TrackUAV – Fault Detection and Predictive Maintenance of Drones

1. Introduction

L'usage des drones connaît une croissance rapide dans de nombreux secteurs : inspection d'infrastructures, logistique, agriculture, sécurité ou collecte de données. Dans ces contextes, la **fiabilité** et la **sécurité** des systèmes deviennent des enjeux majeurs. Une défaillance en vol, en particulier au niveau des hélices (propellers), peut entraîner la perte du drone, des dommages matériels ou des risques pour les personnes.

L'objectif de ce projet est de développer une approche de **maintenance prédictive** pour un drone quadricoptère, en s'appuyant exclusivement sur des données capteurs et des modèles de Machine Learning. Plus précisément, nous visons à :

- détecter automatiquement la présence ou non d'un défaut sur les hélices,
- identifier le **type de défaut** (crack, edge cut, surface cut),
- estimer la **sévérité** de ce défaut pour anticiper la maintenance.

Le projet s'appuie sur le dataset open-source **DronePropA** et sur l'écosystème **IBM watsonx.ai**, utilisé pour tout l'entraînement et l'expérimentation des modèles (notebooks et AutoAI). Seule la phase de prétraitement brut (conversion des `.mat` vers `.npy`) a été réalisée en local avant d'être chargée sur la plateforme.

2. Contexte et objectifs

2.1. Contexte de la maintenance prédictive

Dans l'industrie, la maintenance évolue progressivement :

- de la **maintenance corrective** (intervention après la panne),
- vers la **maintenance préventive et prédictive**, qui vise à détecter les signaux faibles annonçant une défaillance.

Pour les drones, les hélices sont des composants particulièrement exposés : chocs, fissures, coupures de bord ou de surface. Pouvoir surveiller leur état grâce aux signaux issus des IMU et des capteurs embarqués permettrait d'augmenter la disponibilité de la flotte, de réduire les risques de crash et d'optimiser les coûts de maintenance.

2.2. Objectifs du projet TrackUAV

Les objectifs techniques du projet peuvent être résumés ainsi :

1. **Construire un pipeline de données complet**, capable de transformer les fichiers bruts `.mat` du dataset DronePropA en fenêtres temporelles prêtes pour le Machine Learning.
2. **Entraîner plusieurs modèles** sur la plateforme **watsonx.ai** pour :
    - la détection de défaut (binaire),
    - la classification du type de défaut (multi-classe),
    - l'estimation de la sévérité (niveaux 0 à 3).
3. **Comparer différentes familles de modèles** :
    - modèles classiques de type **XGBoost**,
    - un **modèle deep learning multi-tâches** (CNN 1D),
    - un **modèle KNN GPU-ready** avec labels combinés (fault + type + severity),
    - et les pipelines générés automatiquement par **AutoAI**.
4. **Mettre en place une interface de démonstration** permettant de charger un vol et d'observer en continu l'état de santé prédit du drone (probabilité de défaut, type, sévérité), sous forme de monitoring quasi temps réel.

3. Dataset DronePropA

3.1. Description générale

Le dataset **DronePropA** contient **130 vols expérimentaux** effectués en intérieur sur un drone commercial. Les auteurs y ont injecté différents défauts sur une hélice, selon :

- **3 types de défauts** :
    - *crack* (fissure),
    - *edge cut* (détérioration du bord),
    - *surface cut* (détérioration de surface),
- **3 niveaux de sévérité** par type,
- 1 condition saine (aucun défaut).

Les vols suivent **5 trajectoires** :

1. trajectoire en croix (diagonales d'un carré 1×1 m),
2. trajectoire carrée,
3. montée par paliers d'altitude (0.2 m → 0.8 m),
4. montée directe,
5. manœuvre de lacet (yaw) avec rotations ±45° et ±90°.

Les expériences couvrent deux scénarios de vitesse maximale :

- **SP1** : (2.0 m/s, 3.14 rad/s),
- **SP2** : (0.33 m/s, 0.52 rad/s).

Nous avons choisi d'utiliser **l'intégralité du dataset (~4 Go)**, sans troncature, afin de conserver la variabilité temporelle réelle des vols.

On a utilisé le code matlab dans le papier de recherche pour explorer les données.

3.2. Structure des fichiers

Chaque fichier `.mat` contient notamment :

- `QDrone_data` : données capteurs (IMU, états, etc.),
- `commander_data` : consignes de commande, thrust, etc.,
- `stabilizer_data` : états internes du contrôleur.

La première ligne de chaque matrice représente le temps, les lignes suivantes les signaux.

Le nom du fichier encode les conditions de vol, par exemple :

`F3_SV2_SP1_t4.mat` → type de défaut 3 (surface_cut), sévérité 2, scénario de vitesse 1, trajectoire 4.

Un module dédié (`data/metadata.py`) extrait automatiquement les labels :

- `healthy` (1 = sain, 0 = défaut),
- `fault_type` ∈ {none, crack, edge_cut, surface_cut},
- `severity` ∈ {0,1,2,3},
- `trajectory` ∈ {1..5},
- `drone_id` ∈ {D1, D2, D3},
- `speed_case` ∈ {SP1, SP2}.

3.3. Volume de données après préparation

Après prétraitement (section 5), nous obtenons un dataset de fenêtres :

- **X_windows** : `(19 923, 100, 111)` → environ 20 000 fenêtres de 1 seconde (100 échantillons à 100 Hz) avec 111 features normalisées,
- **y_fault, y_type, y_sev** : vecteurs de labels de longueur 19 923.

Ces fichiers (`X_windows.npy`, `y_fault.npy`, `y_type.npy`, `y_sev.npy`, `normalization_stats.npz`) ont été **générés en local**, puis **uploadés sur watsonx.ai** pour tous les entraînements de modèles.

4. Architecture logicielle

Le dépôt est structuré en modules :

- `data/` :
  - `loader.py` : chargement des `.mat` et conversion en `DataFrame`,
  - `metadata.py` : parsing des noms de fichiers et génération des labels,
  - `preprocessing.py` : resampling, normalisation, découpe en fenêtres.
- `features/` : extraction des **features temporelles** et **fréquentielles** à partir des fenêtres.
- `models/` :
  - `classical_xgb.py` : modèles XGBoost pour les trois tâches,
  - `deep_mtl.py` : CNN 1D multi-tâches,
  - `knn_cuda.py` : KNN GPU-ready avec labels combinés.
- `training/` : fonctions d'évaluation communes et boucle d'entraînement pour le modèle MTL.
- Scripts principaux :
  - `main_prepare_data.py` : pipeline de préparation (local),
  - `main_train_xgb_gpu.py` : entraînement des modèles XGBoost (sur watsonx notebooks, CPU),
  - `main_train_deep_mtl.py` : entraînement du CNN multi-tâches,
  - `main_train_knn_cuda.py` : entraînement du KNN combiné,
  - `app.py` : interface Streamlit de démonstration et de monitoring.

5. Prétraitement des données

5.1. Étapes globales

Le prétraitement s'effectue en plusieurs étapes dans `main_prepare_data.py` :

1. Chargement de tous les fichiers `.mat` via `load_all_flights`.
2. Resampling à fréquence fixe (100 Hz).
3. Calcul de la moyenne et de l'écart-type globaux.
4. Normalisation de chaque vol.
5. Découpage en fenêtres glissantes.
6. Construction des labels pour chaque fenêtre.

7. Sauvegarde des matrices numpy (`.npy`) et des statistiques de normalisation (`.npz`).

Cette étape a été réalisée localement, puis les fichiers résultants ont été importés dans **watsonx.ai**. Toutes les phases de **modélisation, entraînement et évaluation** ont ensuite été effectuées dans des notebooks watsonx.

5.2. Resampling et gestion des valeurs manquantes

La fonction `resample_signal` (modifiée dans `data/preprocessing.py`) :

- récupère le vecteur temps `t`,
- calcule une grille temporelle régulière à 100 Hz,
- interpole chaque colonne par interpolation linéaire :

```
for col in df.columns:
    if col == "time":
        continue
    out[col] = np.interp(new_t, t, df[col].values)
```

Ce choix a deux avantages :

1. imposer une fréquence d'échantillonnage uniforme pour tous les vols,
2. **éliminer implicitement les trous de données** (si un capteur est momentanément manquant, la valeur est interpolée à partir des points adjacents).

Après resampling, nous avons vérifié l'absence de valeurs NaN. Le prétraitement est donc robuste à d'éventuels manques de mesures ponctuels.

5.3. Normalisation globale

La fonction `compute_global_mean_std` concatène tous les DataFrames resamplés (hors temps) et calcule :

- `mean` (Series par feature),
- `std` (Series par feature).

La fonction `normalize_df` applique ensuite :

$$x_{norm} = \frac{x - \mu}{\sigma + 10^{-8}}$$

pour chaque colonne (hors `time`). Les statistiques `mean` et `std` sont sauvegardées dans `normalization_stats.npz` et réutilisées :

- pour préparer `X_windows.npy`,
- dans l'application Streamlit pour normaliser un nouveau vol chargé par l'utilisateur.

5.4. Fenêtrage glissant

La fonction `sliding_windows` découpe chaque vol normalisé en fenêtres glissantes :

- durée de fenêtre : `win_sec = 1.0 s` → 100 points à 100 Hz,
- pas : `step_sec = 0.5 s` (chevauchement de 50 %),
- résultat : un tableau `windows` de forme `(n_win, win_len, n_feat)`.

Les valeurs de temps sont utilisées uniquement pour positionner les fenêtres, mais ne sont pas stockées comme feature.

5.5. Export et upload sur watsonx.ai

À la fin de `main_prepare_data.py`, les tableaux suivants sont sauvegardés :

- `X_windows.npy`,
- `y_fault.npy`, `y_type.npy`, `y_sev.npy`,
- `normalization_stats.npz`,
- `fault_type_mapping.json`.

Ces fichiers ont ensuite été **uploadés sur la plateforme watsonx.ai** (Object Storage) et utilisés comme source de données dans les **notebooks Python** pour tous les entraînements de modèles.

6. Méthodologie de modélisation

Nous avons entraîné et comparé trois familles de modèles supervisés sur watsonx.ai :

1. **XGBoost** (modèles classiques) pour les trois tâches,
2. **CNN 1D multi-tâches** (Deep MTL),
3. **KNN GPU-ready avec labels combinés**.

En complément, nous avons utilisé **AutoAI** pour générer automatiquement des pipelines et comparer nos résultats.

Tous les entraînements dans les notebooks ont été exécutés sur **CPU** sur la plateforme watsonx.ai, avec un temps moyen d'environ **50 minutes par entraînement complet** (extraction de features incluse).

6.1. Modèles XGBoost

6.1.1. Extraction des features (temps + fréquence)

À partir de `X_windows.npy`, nous extrayons :

- des **features temporelles** (`features/time_domain.py`) :
  moyenne, min, max, écart-type, RMS, skewness, kurtosis, etc.,
- des **features fréquentielles** (`features/freq_domain.py`) :
  énergie spectrale, énergie par bandes, fréquence dominante, entropie spectrale, etc.

Les deux DataFrames sont concaténés pour chaque fenêtre, produisant une matrice de dimension :

- **X_feat : (19 923, 1 554)**.

6.1.2. Tâches supervisées

Trois modèles XGBoost sont entraînés (`models/classical_xgb.py`) :

1. **Détection de défaut (binaire)**
   - objectif : prédire `y_fault ∈ {0,1}`,
   - modèle : `XGBClassifier` (objective `"binary:logistic"`).
2. **Type de défaut (multi-classe)**
   - objectif : prédire `y_type ∈ {0,1,2,3}`,
   - classes mappées à `{0: none, 1: crack, 2: edge_cut, 3: surface_cut}`,
   - modèle : `XGBClassifier` (objective `"multi:softprob"`).
3. **Sévérité (régression)**
   - objectif : prédire `y_sev ∈ {0,1,2,3}`,
   - modèle : `XGBRegressor` (objective `"reg:squarederror"`).

La division train/test se fait via `train_test_split` (80/20) avec stratification sur `y_fault`.

6.1.3. Hyperparamètres

Paramètres communs :

- `n_estimators = 300`,
- `max_depth = 8`,
- `learning_rate = 0.1`,
- `subsample = 0.8`,
- `colsample_bytree = 0.8`,
- `tree_method = "hist"`.

Dans l'environnement watsonx.ai, les modèles ont été exécutés sur CPU (même si le code est compatible GPU), ce qui explique un temps d'entraînement d'environ **50 minutes** par run complet.

Les métriques d'évaluation utilisées :

- **Détection de défaut** : Accuracy, F1-score,
- **Type de défaut** : Accuracy, F1-macro, matrice de confusion,
- **Sévérité** : MAE en continu + MAE après arrondi (0–3).

Les modèles entraînés et la liste des colonnes de features sont sauvegardés dans : `classical_xgb_timefreq_gpu.joblib` (malgré le nom, l'entraînement décrit ici a été effectué sur CPU dans watsonx).

6.2. Modèle Deep Learning Multi-Task Learning (CNN 1D)

6.2.1. Architecture

Le modèle `CNNMTL` (`models/deep_mtl.py`) est un réseau convolutionnel 1D multi-tâches :

- Entrée : `(batch, seq_len=100, n_feat=111)`.
- Backbone :

```
Conv1d(111 → 64, k=5) + BatchNorm + ReLU + MaxPool
Conv1d(64 → 128, k=5) + BatchNorm + ReLU + MaxPool
Conv1d(128 → 256, k=3) + BatchNorm + ReLU + AdaptiveAvgPool1d(1)
```

- Têtes de sortie :
  - `head_fault` → 2 classes (défaut/pas défaut),
  - `head_type` → n_fault_types classes,
  - `head_sev` → 4 classes (niveaux de sévérité 0..3).

6.2.2. Perte multi-tâches

La fonction de perte combine trois pertes de cross-entropy :

$$L = \lambda_{\text{fault}} L_{\text{fault}} + \lambda_{\text{type}} L_{\text{type}} + \lambda_{\text{sev}} L_{\text{sev}}$$

avec les poids définis dans `config.py` :

- λ_fault = 2.0,
- λ_type = 1.0,

- $\lambda\_sev = 1.0$.

6.2.3. Entraînement sur watsonx.ai

L'entraînement du CNN s'effectue dans `main_train_deep_mtl.py` :

- split train/validation 80/20,
- `batch_size = 64`,
- `num_epochs = 50`,
- optimiseur **Adam (lr = 1e-3)**.

L'ensemble du training est exécuté sur **CPU** dans un notebook watsonx.ai, avec un temps d'exécution similaire à XGBoost (de l'ordre de 50 minutes).

Le meilleur modèle (selon la loss de validation) est sauvegardé dans `deep_mtl_cnn1d.pth` avec ses hyperparamètres et ses métriques de validation.

6.3. Modèle KNN GPU-ready à labels combinés

Pour explorer une approche plus simple mais intéressante pour l'interprétation, nous avons implémenté un modèle **KNN** optimisé pour fonctionner sur GPU (quand il est disponible) dans `models/knn_cuda.py`. Dans notre cas, sur watsonx.ai, le code a tourné sur CPU, mais l'implémentation reste compatible CUDA.

6.3.1. Encodage des labels combinés

L'idée est de combiner les trois informations `(fault, type, severity)` en un **seul label discrétisé** :

```
code = fault * 100 + type * 10 + sev
```

- `fault` ∈ {0,1},
- `type` ∈ {0,1,2,3},
- `sev` ∈ {0,1,2,3}.

Des fonctions utilitaires `encode_combined_labels` et `decode_combined_labels` permettent :

- d'encoder les triplets (fault, type, sev) en un code entier unique,
- et de retrouver ce triplet à partir du code.

Cela permet d'entraîner un KNN dont chaque classe représente directement une combinaison `(fault, type, severity)`.

6.3.2. Entraînement et évaluation

Le script `main_train_knn_cuda.py` :

1. Charge `X_windows` et les vecteurs `y_fault`, `y_type`, `y_sev`.
2. Extrait les mêmes features temps + fréquence que pour XGBoost.
3. Effectue un split train/test (80/20) stratifié sur `y_fault`.
4. Crée un objet `CudaKNNCombined(k=5)` :
    - device = `"cuda"` si disponible, sinon `"cpu"`.
5. Entraîne le KNN en stockant les embeddings du train en mémoire (sur GPU ou CPU).
6. Prédit les labels combinés pour le set de test, puis les décode en `(fault, type, severity)`.

Les métriques calculées :

- **accuracy** et **F1-macro** sur les **labels combinés**,
- accuracy et F1 binaire sur `fault` seul,
- métriques classiques sur `type` (accuracy, F1-macro, confusion matrix),
- MAE sur `severity` (valeur continue issue du voisinage majoritaire, puis arrondi 0..3).

Les résultats montrent que le KNN combiné est cohérent, mais reste légèrement en dessous des performances de XGBoost, ce qui est logique compte tenu du caractère très high-dimensional du problème.

Le modèle est sauvegardé dans `knn_cuda_combined.pth` avec :

- l'état du KNN (tensors des points de référence),
- les colonnes de features,
- la liste des classes combinées observées.

6.4. AutoAI sur watsonx.ai

En complément des entraînements manuels, nous avons utilisé **AutoAI** pour :

- valider nos choix de features et de modèles,
- explorer automatiquement différents pipelines (feature engineering, sélection de modèles, HPO).

6.4.1. Pipelines générés

AutoAI a généré plusieurs pipelines basés notamment sur **LightGBM (LGBM)**, avec optimisation des hyperparamètres et éventuellement du feature engineering.

Exemple de classement obtenu (pour une tâche de classification sur la sévérité ou le type, selon les expériences) :

Rang	Pipeline	Algorithme principal	Exactitude (validation croisée)
1	Pipeline 3	Discriminant LGBM	**0.950**
2	Pipeline 2	Discriminant LGBM	0.943
3	Pipeline 1	Discriminant LGBM	0.933

Les métriques de type F1-macro, précision, rappel et log-loss sont cohérentes avec nos résultats XGBoost.

6.4.2. Limite liée aux crédits

AutoAI **consomme beaucoup de crédits** sur la plateforme watsonx.ai. Nous n'avions pas un quota suffisant pour :

- lancer des pipelines AutoAI séparés sur les trois cibles (fault, type, severity),
- ni pour faire des tours de raffinement supplémentaires.

Nous avons donc **limité les essais AutoAI** à quelques expériences ciblées (principalement classification) et nous avons réutilisé les enseignements obtenus (choix d'algorithmes, importance des features) pour ajuster nos propres modèles, tout en gardant le budget de crédits sous contrôle.

7. Environnement de calcul et contraintes

- Plateforme principale : **IBM watsonx.ai**
  - Notebooks Python pour tout l'entraînement des modèles (XGBoost, CNN MTL, KNN).
  - AutoAI pour la génération automatique de pipelines.
- Prétraitement brut (`main_prepare_data.py`) exécuté **en local** pour convertir les fichiers `.mat` volumineux en `.npy` plus compacts, puis upload sur watsonx.
- Entraînements **sur CPU** (aucun runtime GPU disponible sur notre compte étudiant).
- Temps moyen d'entraînement par modèle (XGBoost ou CNN) : ≈ **50 minutes** sur watsonx notebooks.
- Usage d'AutoAI limité par le **budget de crédits**, ce qui a restreint le nombre d'expériences (notamment pour la sévérité).

8. Résultats

8.1. XGBoost (features temps+freq)

Les résultats sur le set de test (20 % des fenêtres) sont les suivants :

8.1.1. Détection de défaut (binaire)

- **Accuracy** : **0,958**
- **F1-score** : **0,970**

Le modèle distingue très bien les fenêtres issues de vols sains de celles issues de vols défectueux.

8.1.2. Type de défaut (multi-classe)

- **Accuracy** : **0,932**
- **F1-macro** : **0,930**

Matrice de confusion (extrait console) :

```
[[1171   21   21    9]
 [  58  821   19   23]
 [  25   20  850   23]
 [   4   13   16  872]]
```

Les trois types de défaut sont correctement discernés, malgré leur proximité physique (tous affectent une hélice).

8.1.3. Sévérité (régression)

- **MAE (valeur continue)** : **0,466**
- **MAE (niveaux arrondis 0..3)** : **0,402**

L'erreur moyenne d'environ 0,4 sur une échelle de 0 à 3 signifie que le modèle se trompe généralement d'au plus un niveau de sévérité.

8.2. CNN Multi-Task

Le modèle CNN 1D multi-tâches obtient des performances légèrement inférieures à XGBoost :

- Accuracy de détection du défaut autour de **0,94**,
- MAE de sévérité légèrement supérieure à celle de XGBoost.

L'intérêt principal de ce modèle est :

- l'apprentissage direct sur les signaux bruts (sans features manuelles),
- la mutualisation de l'information entre les trois tâches,
- une architecture naturellement adaptée à un futur déploiement temps réel.

8.3. KNN GPU-ready à labels combinés

Le modèle KNN combiné fournit :

- une représentation intuitive de l'état complet `(fault, type, severity)` sous forme d'une seule classe,
- des **metrics raisonnables** (accuracy, F1-macro et MAE) mais inférieures à celles de XGBoost, ce qui est cohérent pour un algorithme non paramétrique dans un espace de dimension aussi élevée,
- un outil intéressant pour analyser des cas particuliers : on peut facilement inspecter les voisins les plus proches d'une fenêtre donnée et comprendre de quels vols et de quelles conditions elle se rapproche le plus.

8.4. AutoAI

Les pipelines générés par AutoAI confirment :

- la **pertinence des modèles de type gradient boosting** (LGBM/XGBoost),
- l'importance des features temps + fréquence pour séparer correctement les classes,
- des performances comparables à nos modèles faits "à la main" (accuracy ≈ 0,95).

La contrainte de crédits n'a pas permis de couvrir exhaustivement toutes les tâches (surtout la régression de sévérité), mais les résultats obtenus confortent nos choix de modélisation.

9. Discussion

9.1. Analyse des performances

Les modèles de type gradient boosting (XGBoost / LGBM AutoAI) offrent le **meilleur compromis** entre :

- performance (accuracy et F1 très élevées),

- temps de calcul acceptable,
- interprétabilité (importance des features).

Le CNN multi-tâches est légèrement en retrait en termes de scores, mais ouvre une voie intéressante pour :

- tirer parti de signaux plus complexes (séquences plus longues, signaux bruts),
- fusionner plus facilement de nouvelles modalités (par exemple, audio ou vibration).

Le KNN combiné, même s'il n'est pas le plus performant, illustre bien une autre manière de structurer le problème (classe combinée) et de visualiser la similarité globale entre états de santé.

9.2. Limites

- Dataset limité à des **conditions indoor** ; il faudrait tester les modèles sur des vols outdoor soumis au vent et à des perturbations plus variées.
- La sévérité est représentée par un niveau discret (0–3) issu du protocole expérimental ; en pratique, une mesure plus continue ou basée sur les heures de vol pourrait être envisagée.
- Tous les entraînements ont été faits sur CPU : l'utilisation d'un **runtime GPU** permettrait d'explorer des architectures plus profondes (Transformers, CNN plus larges) et de faire plus d'essais.

10. Interface de démonstration (Streamlit)

Pour valoriser les modèles et rendre le projet plus concret, nous avons développé une interface interactive en **Streamlit**(`app.py`) baptisée :

TrackUAVFault – Predictive Maintenance for Drones

Cette interface fait le lien entre le pipeline de données, le modèle XGBoost et une visualisation riche de l'état de santé du drone, sous forme de **monitoring quasi temps réel** et de **trajectoire 3D**.

10.1. Chargement des modèles et paramètres de normalisation

Au lancement de l'application :

- le script charge le **modèle XGBoost** entraîné (`classical_xgb_timefreq_gpu.joblib`) via la fonction `load_xgb_model()` ;
- il force explicitement les 3 modèles XGBoost (fault, type, severity) à tourner sur **CPU** pour éviter tout problème de compatibilité GPU avec Streamlit ;

- il charge également les **statistiques de normalisation** (moyenne et écart-type) depuis `normalization_stats.npz` via `load_normalization_stats()`.

Ces éléments garantissent que les données d'un nouveau vol sont prétraitées de la **même manière** que les données d'entraînement.

10.2. Modes d'utilisation : vol de démonstration ou fichier `.mat`

L'utilisateur choisit un mode dans la sidebar :

1. **Demo flight (preprocessed data)**
    - L'application charge directement un sous-ensemble de `X_windows.npy` (≈ 200 fenêtres) comme vol de démonstration.
    - Les timestamps synthétiques sont générés à partir du pas de fenêtre (`cfg.data.step_sec`).
    - Ce mode est particulièrement utile pour la vidéo de démonstration, car il ne dépend pas d'un fichier externe.
2. **Upload .mat DronePropA**
    - L'utilisateur uploade un fichier `.mat` d'un vol DronePropA.
    - Le fichier est sauvegardé temporairement (`tmp_uploaded.mat`), puis converti en `DataFrame` avec `load_flight()`.
    - Le nom du fichier est parsé via `parse_filename()` pour récupérer, entre autres, l'**ID de trajectoire** (`trajectory`).
    - Le vol brut est affiché dans un premier graphe ("Flight signals overview").
    - Ensuite, le vol est prétraité via `preprocess_flight_df()` :
        - resampling à 100 Hz,
        - normalisation avec les statistiques globales,
        - découpe en fenêtres glissantes,
        - calcul du temps central de chaque fenêtre (`t_windows`).

Dans les deux cas, on obtient un ensemble de fenêtres `X_windows` et un vecteur de temps `t_windows` prêt pour la prédiction.

10.3. Simulation de trajectoire 3D

Une des nouveautés importantes du `app.py` est la **simulation de trajectoire 3D** du drone, même si les données de position ne sont pas explicitement présentes dans le dataset.

On distingue deux cas :

- si le vol provient d'un fichier `.mat` et que l'on dispose de colonnes `x`, `y`, `z`, celles-ci sont rééchantillonnées et utilisées comme trajectoire réelle ;
- sinon, on génère une **trajectoire synthétique** en 2D/3D à l'aide de fonctions dédiées :
    - `compute_trajectory_xyz()` pour adapter ou générer X, Y, Z,

- `generate_synthetic_trajectory()` pour dessiner différentes formes : lemniscate, cercle, carré, spirale, Lissajous, etc.

En mode **démo**, l'utilisateur peut régler finement la trajectoire dans la sidebar :

- forme (`shape`),
- échelle (`scale`),
- nombre de boucles (`loops`),
- niveau de bruit (`noise`),
- rotation de la trajectoire (`rotation_deg` en degrés).

Un premier graphique 3D Plotly affiche cette trajectoire fictive. Ensuite, lors de l'analyse globale et de la simulation, la trajectoire est réutilisée pour visualiser la position du drone et surligner les segments en défaut.

10.4. Pipeline de prédiction sur fenêtres

La fonction `predict_on_windows()` orchestre la prédiction sur toutes les fenêtres :

1. Extraction des **features temporelles** (`extract_time_features_from_windows`) et **fréquentielles**(`extract_freq_features_from_windows`), comme pour l'entraînement.
2. Sélection des colonnes de features dans le même ordre que lors du training (`feature_columns`).
3. Appel du modèle XGBoost via `predict_with_xgb()` pour obtenir :
   - probabilité de défaut (`Fault_Prob`),
   - type de défaut (`Fault_Type`),
   - sévérité continue (`Severity_Continuous`), ensuite arrondie en niveau 0–3 (`Severity_Level`).
4. Ajout d'une colonne lisible de type de défaut (`Fault_Type_Name`) à partir d'un dictionnaire `{0: none, 1: crack, 2: edge_cut, 3: surface_cut}`.

En **mode démo**, pour rendre l'animation plus contrôlable et visuellement intéressante, on peut **overrider** les prédictions avec un scénario synthétique :

- un pourcentage de fenêtres en défaut (`synth_fault_ratio`),
- des types de défauts tirés aléatoirement parmi {crack, edge_cut, surface_cut},
- des niveaux de sévérité 1 à 3,
- et des probabilités de défaut cohérentes (proba élevée en cas de défaut, faible sinon).

Cela permet de simuler des cas variés pour la vidéo sans dépendre de la répartition réelle du dataset.

10.5. Analyse globale du vol

La section **“Predictive maintenance analysis”** permet de lancer l’analyse globale du vol sur tous les instants :

- **Résumé global** via trois métriques :
  - probabilité moyenne de défaut,
  - pourcentage du temps passé en état défectueux,
  - sévérité moyenne.
- **Distribution des niveaux de sévérité** (histogramme Matplotlib).
- **Timeline de l’état de santé** avec un graphique Streamlit :
  - sévérité,
  - probabilité de défaut,
  - label binaire (normal vs défaut).
- **Trajectoire 3D du drone** avec Plotly :
  - la trajectoire complète en gris,
  - les points “normaux” en bleu,
  - les points en défaut colorés selon la sévérité (échelle de rouge et taille du marker dépendant du niveau 0–3).

Enfin, un tableau affiche les premières lignes des prédictions (`Fault_Prob`, `Fault_Label`, `Fault_Type_Name`, `Severity_Level`), ce qui permet d’inspecter les résultats numériquement.

10.6. Simulation de vol et monitoring temps réel

La section **“Flight simulation & real-time monitoring”** est conçue pour la vidéo de démo :

- à chaque itération (fenêtre temporelle), l’application :
  - affiche un **en-tête** avec l’état actuel :
    - temps `t`,
    - état NORMAL/FAULT,
    - sévérité,
    - type de défaut ;
  - tente de charger une **image correspondant au défaut** (`images/F{fault_type}SV{sev_level}.png`), qui illustre visuellement le type et la sévérité (par exemple un propeller fissuré) ;
  - met à jour des **indicateurs** (metrics) : probabilité de défaut, sévérité, type ;
  - met à jour un **graphe temps réel** de la sévérité et de la probabilité, jusqu’à l’instant courant ;
  - raffiche une **trajectoire 3D dynamique** :
    - la trajectoire déjà parcourue,
    - les zones défectueuses,
    - la position actuelle du drone en jaune.

La simulation est limitée à un certain nombre de fenêtres (`max_steps`, typiquement 150) afin d'obtenir une animation fluide et exploitable en vidéo. La vitesse d'animation est contrôlée par un slider (`sim_speed`).

10.7. Exploration d'un instant précis

Enfin, la section **"Explore a flight instant"** permet à l'utilisateur de :

- choisir un instant `t` via un slider,
- récupérer automatiquement la fenêtre la plus proche dans `t_windows`,
- afficher :
    - l'état à cet instant (NORMAL/FAULT),
    - la probabilité de défaut,
    - la sévérité,
    - le type de défaut,
- visualiser :
    - la timeline jusqu'à cet instant (sévérité & probabilité),
    - la trajectoire 3D complète avec un marqueur sur la **position sélectionnée**.

Cette fonctionnalité est utile pour analyser en détail un moment précis du vol et faire des captures ciblées pour le rapport ou la présentation.

11. Conclusion et perspectives

Nous avons développé, dans le cadre de ce projet, une solution complète de **détection de défauts et de maintenance prédictive pour drones**, en nous appuyant sur :

- le dataset DronePropA,
- un pipeline robuste de prétraitement (resampling, normalisation, fenêtrage),
- plusieurs modèles supervisés (XGBoost, CNN multi-tâches, KNN combiné),
- la plateforme **IBM watsonx.ai** pour l'ensemble des entraînements et expérimentations,
- une interface de visualisation et de simulation de vol.

Parmi tous les modèles testés, le **modèle CNN 1D multi-tâches (MTL)** s'est révélé **particulièrement performant**, avec des résultats au moins comparables, et dans certains cas meilleurs, que ceux obtenus par XGBoost. Le CNN présente plusieurs atouts majeurs :

- il apprend directement sur les **signaux bruts**, sans dépendre uniquement de features manuelles,
- il traite en **même temps** les trois tâches (détection, type, sévérité) grâce au partage de représentation,

- il montre une bonne capacité de généralisation sur l'ensemble des vols, tout en capturant des patterns temporels plus fins que les modèles classiques.

Les modèles **XGBoost** restent néanmoins une référence solide : ils offrent des performances très élevées, une bonne interprétabilité via les features temps/fréquence et ont servi de **baseline forte** pour comparer le CNN. Le **KNN à labels combinés** apporte, lui, une vision intéressante des états globaux (fault + type + severity) et une interprétation intuitive par voisinage, même s'il est moins performant que le CNN et XGBoost.

Les expériences menées avec **AutoAI** confirment ces conclusions : les pipelines automatiques basés sur des modèles de type gradient boosting (LightGBM) atteignent des performances proches de nos meilleurs modèles, mais ne surpassent pas clairement le CNN multi-tâches lorsque l'on exploite pleinement l'information temporelle.

Globalement, le projet montre qu'il est possible de **surveiller l'état de santé des drones** de manière fiable à partir des seules données capteurs, et que les architectures **deep learning temporelles multi-tâches** (comme notre CNN) constituent une piste très prometteuse pour la maintenance prédictive.

Perspectives

Plusieurs prolongements sont envisageables :

- tester le modèle CNN sur des **scénarios de vol plus variés** (outdoor, vent, perturbations fortes),
- intégrer de nouveaux signaux (vibration, audio, images des hélices) dans la même architecture multi-tâches,
- exploiter un environnement **GPU** dédié pour explorer des modèles encore plus riches (CNN plus profonds, Transformers temporels),
- déployer le modèle sur une **plateforme embarquée** afin d'effectuer la détection de défaut en quasi temps réel à bord du drone,
- utiliser des techniques d'**explicabilité pour deep learning** (Grad-CAM, Integrated Gradients, etc.) afin de mieux comprendre quelles zones temporelles et quels capteurs guident les décisions du CNN.

En résumé, **le CNN multi-tâches apparaît comme le cœur du système TrackUAV**, avec XGBoost et KNN comme modèles de comparaison et de support. Ensemble, ils constituent une base solide pour un futur système industriel de maintenance prédictive des drones.

Pistes futures

- Étendre l'étude à des **vols outdoor** et à d'autres types de drones.
- Intégrer des capteurs supplémentaires (son, vibration, vidéo) pour enrichir la signature des défauts.

- Tester des architectures plus avancées (RNN, Transformers) grâce à un environnement GPU dédié.
- Déployer les modèles sur une plateforme embarquée ou en edge computing pour une **détection en temps réel**directement à bord du drone.
- Exploiter l'explicabilité (SHAP, importance des features) pour aider les ingénieurs maintenance à comprendre précisément quelles caractéristiques du signal trahissent chaque type de défaut.