جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

ENSAM
المدرسة الوطنية العليا للفنون و المهن
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS

Rapport du projet DataMining

Predictive Maintenance of Industrial Machines : NASA Turbofan Engine Degradation Dataset (C-MAPSS)

Réalisé par:

El Hend Amine
Boulaid Mouad
Naji Zakaria
Ait Abdou Aymane
Mohamed Taha Ouhadda
Ben-amar Abdessamad

Encadré par: Pr. Hosni

Introduction

Predictive Maintenance is a proactive maintenance practice that aims to foretell equipment failure during service so that maintenance can be scheduled at the time of imminent failure. This minimizes breakdown downtime, optimizes maintenance resources, and optimizes operational safety. In high-value asset industries like aerospace, where engine failure can be catastrophic and have severe economic consequences, efficient predictive maintenance is essential. Turbofan engines, complex assemblies of rotating machinery that run in severe environments, are prime candidates for predictive maintenance since they serve such a critical role and are susceptible to progressive deterioration. This project is targeted at the development of data-driven models for the predictive maintenance of these engines.

The primary objective of this research endeavor is to predict the Remaining Useful Life (RUL) of turbofan engines. RUL is the number of operating cycles left prior to when an engine is likely to fail or need critical maintenance. Precise estimation of RUL allows operators to make data-driven decisions regarding maintenance scheduling, inventory planning, and operations planning. The difficulty is in using historical sensor readings, which tend to have complicated patterns and noise, to predict this future state.

Dataset Understanding and Description

The data employed in the present study is the well-known NASA Turbofan Engine Degradation Dataset. The dataset was generated artificially with the Commercial Modular Aero-Propulsion System Simulation (C-MAPSS) tool, which was established by NASA. The reason for this was to have a standard benchmark to allow researchers to develop and validate prognostic algorithms.
C-MAPSS is a fidelity simulation framework for commercial turbofan engines. C-MAPSS simulates engine performance over a variety of flight conditions (altitude, Mach number, throttle resolver angle - TRA) and allows simulation of degradation in the engine modules (e.g., Fan, LPC, HPC, HPT, LPT) by modifying health parameters like flow capacity and efficiency.
A critical term for data generation is the "health index." The health index is not a direct output in the dataset but was used to determine when an engine simulation needs to terminate (i.e., failure).
Health indicator H(t) is the minimum of a predefined number of normalized operating margins at time t: H(t)=min(mFan,mHPC,mHPT,mEGT), where the margins are associated with stall limits and exhaust gas temperature (EGT) limitations.
. 1 represents a healthy system in pristine condition, while 0 signifies that the margin has degraded entirely to a pre-defined threshold (e.g., 15% for HPC stall margin, 2% for EGT margin).
Failure is quantified as the instant when the health index H(t) hits zero.

Data Structure and Features

The dataset is provided as text files, where each row represents a snapshot of data taken during a single operational cycle. The columns include :

1.unit_number: Identifier for the engine.
2.time_cycles: The operational cycle number.
3.setting_1, setting_2, setting_3: Operational settings (e.g., altitude, Mach number, TRA).
4.sensor_1 to sensor_21: Measurements from 21 different sensors.

Data Preprocessing

Rationale for Preprocessing

Raw sensor data is often not directly suitable for machine learning models. Preprocessing is essential to:
. Structure the data correctly.
. Create the target variable (RUL) for training.
. Ensure features are on a comparable scale, which is important for many algorithms.
. Handle any inconsistencies or missing information.
The preprocessing steps are implemented in the load_data and preprocess_data functions within the provided app.py script.

Data Loading and Initial Inspection

Process The train_FD001.txt, test_FD001.txt, and RUL_FD001.txt files are loaded into pandas DataFrames. Standardized column names are assigned: 'unit_number', 'time_cycles', three settings, and 21 sensors.

Justification Pandas DataFrames provide a robust and flexible structure for data manipulation in Python. Consistent naming is crucial for reproducibility and clarity. Initial inspection (e.g., df.head(), df.info(), df.describe()) helps verify correct loading and provides a first look at the data's characteristics.

Calculation of RUL for Training Data

Methodology

For each engine unit in the training data, the RUL is calculated as the difference between its maximum recorded cycle (time of failure) and its current cycle count.
1. max_cycle = train_data.groupby('unit_number')['time_cycles'].transform('max')
2. RUL = max_cycle - train_data['time_cycles']
This piecewise linear degradation assumption for RUL is a common approach for this dataset.

Justification

Since the training data includes run-to-failure trajectories, the last cycle recorded for an engine represents its failure point. The RUL at any earlier cycle is simply the time remaining until failure. This thus defines the ground truth target variable needed for supervised learning. This method ties in directly with the RUL definition of "remaining operational cycles before failure."

Data Scaling (Normalization)

The MinMaxScaler from sklearn.preprocessing is used to scale the operational settings and sensor measurement columns to a range of [0, 1].

Process

1. The scaler is fitted only on the training data (scaler.fit_transform(train_data[scale_cols])).
2. The same fitted scaler is then used to transform the test data (scaler.transform(test_data[scale_cols])).

Justification

- *Algorithm Sensitivity:* Many machine learning algorithms (e.g., SVMs, k-NN, neural networks, and gradient-based methods like those in XGBoost or linear regression with regularization) are sensitive to feature scales. Features with larger value ranges can dominate those with smaller ranges, leading to suboptimal model performance. Normalization brings all features to a comparable scale.
- *Improved Convergence:* For gradient-based optimization algorithms, normalization can lead to faster convergence.
- *MinMaxScaler Rationale:* This scaler is chosen because it preserves the shape of the original distribution and does not distort relationships between values. It's effective when the data does not have strong outliers or when a strict bounding of feature values is desired.
- *Preventing Data Leakage:* Fitting the scaler only on the training data and then applying it to the test data is crucial to prevent data leakage. Using information from the test set (like its min/max values) during training would lead to overly optimistic performance estimates.

Handling Operating Conditions

Approach for Single vs. Multiple Conditions

- *FD001 & FD003 (Single Condition):* A new column op_condition is created and assigned a constant value (e.g., 0).
- *FD002 & FD004 (Multiple Conditions):* K-Means clustering (n_clusters=6) is applied to the three operational setting columns (setting_1, setting_2, setting_3) to derive an op_condition label for each data point. The readme.txt mentions six operating conditions for these datasets.

Justification for FD001

For FD001, which operates under a single condition (Sea Level), creating a constant op_condition column ensures a consistent feature set if the analysis framework is later applied to datasets with multiple operating conditions. While not strictly necessary for modeling FD001 in isolation, it aids in building a generalized pipeline.
The C-MAPSS paper also notes that degradation was simulated under various combinations of operational conditions, making explicit handling of these conditions important for datasets like FD002 and FD004.

Handling Operating Conditions

The raw C-MAPSS datasets (FD001-FD004) typically do not have missing values in the provided sensor and setting columns. An initial check (train_data.isnull().sum()) in app.py confirms this. Missing values might be introduced during feature engineering (e.g., for rolling statistics at the beginning of a series), and these are handled by fillna strategies.

Exploratory Data Analysis (EDA)

Objectives of EDA

- Gain insights into the data's structure and distributions.
- Identify patterns, trends, and anomalies in sensor readings.
- Understand relationships between features and the target variable (RUL).
- Inform feature engineering and model selection choices.

The app.py script facilitates interactive EDA primarily through visualizations.

Sensor Trends Visualization

Process

Line plots are generated showing the evolution of selected sensor readings over time_cycles for individual engine units (e.g., unit 1). The app.py allows users to select sensors for plotting.

Interpretation

These plots help visualize:

- *Degradation Signatures:* Sensors that exhibit clear increasing or decreasing trends as the engine approaches failure (RUL decreases) are good candidates for predictive modeling.
- *Noise Levels:* The plots reveal the extent of noise and fluctuations in sensor readings.
- *Sensor Behavior:* Different sensors might show degradation at different stages or rates. Some sensors might remain stable until close to failure, while others show gradual changes.

Correlation Analysis

Process

Pearson correlation coefficients are calculated between each sensor reading and the RUL. The app.py displays the top 5 sensors with the highest absolute correlation.

Interpretation

- Quantifies the linear relationship between sensor values and RUL.
- Positive correlation: As sensor value increases, RUL tends to increase (or vice-versa if RUL decreases with time).
- Negative correlation: As sensor value increases, RUL tends to decrease.
- Helps in identifying sensors that are strongly indicative of the engine's health state.

Feature Engineering

Importance of Feature Engineering in RUL Prediction

Raw sensor readings, even when normalized, might not capture the full dynamics of engine degradation. Feature engineering aims to create new, more informative features that can help machine learning models better understand the engine's state and predict RUL. This involves transforming existing features to highlight trends, changes in variability, and temporal dependencies. The engineer_features function in app.py implements these techniques.

Rolling Statistics (Moving Averages and Standard Deviations)

Process

For each selected sensor, rolling mean and rolling standard deviation are calculated over a fixed-size sliding window.

- train_data[f'{sensor}_roll_mean'] = train_data.groupby(['unit_number', 'op_condition'])[sensor].transform(lambda x: x.rolling(window=window, min_periods=1).mean()).fillna(train_data[sensor])
- train_data[f'{sensor}_roll_std'] = train_data.groupby(['unit_number', 'op_condition'])[sensor].transform(lambda x: x.rolling(window=window, min_periods=1).std()).fillna(0)

Window Size (window = 5 in app.py)

A window size of 5 cycles is used. This choice represents a trade-off:

- Smaller windows are more responsive to rapid changes but can be noisy.
- Larger windows provide smoother trends but might lag in detecting sudden shifts.

A window of 5 is a common heuristic for this dataset, aiming to capture short-term trends without excessive smoothing.

Handling NaNs

min_periods=1 ensures that a value is computed even if the window is not full (at the start of a series). fillna(train_data[sensor]) for mean and fillna(0) for std are used to backfill any NaNs, typically at the very beginning of each engine's time series. Using the original sensor value for mean backfill and 0 for std backfill (implying no variability initially) are reasonable imputation strategies here.

Lagged Features

Process

Lagged features are created by taking the sensor values from previous time steps (cycles).

- train_data[f'{sensor}_lag1'] = train_data.groupby(['unit_number', 'op_condition'])[sensor].shift(1).fillna(train_data[sensor])
- train_data[f'{sensor}_lag2'] = train_data.groupby(['unit_number', 'op_condition'])[sensor].shift(2).fillna(train_data[sensor])

Lags of 1 and 2 cycles are created.

Handling NaNs

fillna(train_data[sensor]) is used to backfill NaNs at the beginning of each series, imputing with the current sensor value (effectively assuming no change from the unobserved prior state).

Purpose

Provide the model with explicit information about the sensor's recent history. Many degradation processes are path-dependent, meaning the current state depends on previous states.

Impact

Allows models (especially those without inherent memory like standard feed-forward networks or tree-based models) to learn temporal dependencies and how the rate of change or recent sensor values influence RUL.

Model Selection and Training

The primary objective is to develop a regression model capable of accurately predicting the RUL of turbofan engines using the preprocessed and engineered features. The chosen model should generalize well to unseen engine data.

K-Fold Cross-Validation

A KFold cross-validation approach is used, as implemented in app.py, with n_splits=3, shuffle=True, and random_state=42.

K-Fold CV provides a more reliable estimate of a model's generalization performance than a single train-test split by using different subsets of the data for training and validation across K iterations. All data points are used for both training and validation across the folds.

- n_splits=3: Offers a reasonable balance between the computational cost of training multiple models and the stability of the performance estimate.
- shuffle=True, random_state=42: Ensures that the splits are random but reproducible.

Importance of Splitting by Engine Unit

The KFold splitting is performed on the unique unit_number (engine IDs), not on individual data rows. This means all data from a particular engine belongs to either the training set or the validation set within a given fold.

This is a critical aspect for time-series data involving multiple independent units.
If splits were done randomly on rows, data points from the same engine (which are highly correlated temporally) could appear in both training and validation sets. This would lead to the model learning engine-specific idiosyncrasies rather than general degradation patterns, resulting in overly optimistic and unrealistic performance estimates.
In practice, a PdM model is trained on historical data from a fleet of engines and then used to predict RUL for other, potentially new, engines. Splitting by engine unit mimics this scenario.

Machine Learning Models Considered

Linear Regression (Baseline)

Serves as a simple, interpretable baseline. Its performance indicates the extent to which RUL can be predicted by a linear combination of the features.

Random Forest Regressor

A powerful ensemble learning method known for its high accuracy, robustness to overfitting (with proper tuning), ability to capture non-linear relationships, and inherent feature importance estimation. n_jobs=-1 enables parallel processing for faster training.

XGBoost Regressor

An advanced gradient boosting algorithm, often achieving state-of-the-art results. It is known for its performance, speed, and regularization capabilities to prevent overfitting.

Justification for Model Choices

The selection includes a simple linear model, a robust ensemble (Random Forest), and a powerful gradient boosting model (XGBoost). This allows for a comparison across different model complexities and learning paradigms.
Linear Regression offers high interpretability, while Random Forest and XGBoost typically offer higher predictive performance at the cost of some interpretability (though feature importance helps).

Training Protocol

For larger datasets (FD002, FD004), the subsample option in app.py allows training on a random 70% subset of engines to reduce computation time. This is not typically needed for FD001.
For each model selected (Random Forest always; Linear Regression and XGBoost if quick_mode is False):

- The training data (engine units) is divided into K folds.
- In each iteration, K-1 folds of engine data are used for training (X_train, y_train).
- The remaining fold of engine data is used for validation (X_val, y_val).
- The model is fitted on X_train, y_train.
- Predictions are made on X_val.
- Evaluation metrics (RMSE, MAE, R2, Custom Score) are computed for the fold.

The metrics are averaged across all K folds to get the final CV performance for each model.

Final Model for Test Predictions

- After cross-validation, a single Random Forest model is trained on the entire training dataset (train_data[feature_cols], train_data['RUL']) using the user-specified n_estimators and max_depth from the UI.
- This final model is then used to make predictions on the test set. For the test data, predictions are made on the last available cycle for each engine unit (test_last = test_data.groupby('unit_number').last().reset_index()).

- Training the final deployment model on all available training data is standard practice to leverage the maximum amount of information. Random Forest is often chosen as the default final model due to its generally strong and robust performance.

Evaluation of Models

A combination of standard regression metrics and a domain-specific score is used to provide a comprehensive assessment of model performance.

Root Mean Squared Error (RMSE)

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

Measures the standard deviation of the prediction errors (residuals). It penalizes larger errors more heavily due to the squaring term. Lower RMSE values indicate better fit. Units are the same as RUL (cycles).

Mean Absolute Error (MAE)

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Measures the standard deviation of the prediction errors (residuals). It penalizes larger errors more heavily due to the squaring term. Lower RMSE values indicate better fit. Units are the same as RUL (cycles).

R-squared (R2 Score)

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

Indicates the proportion of the variance in the RUL that is predictable from the features. An R2 score closer to 1 suggests a good fit. A score of 0 means the model performs no better than predicting the mean RUL. Negative R2 indicates the model performs worse than predicting the mean.

Custom Score (PHM08 Challenge Asymmetric Score)

$$S = \sum_{i=1}^{n} \begin{cases} e^{-d_i/10} - 1 & \text{if } d_i < 0 \\ e^{d_i/13} - 1 & \text{if } d_i \geq 0 \end{cases} \text{where } d_i = \hat{y}_i - y_i$$

This score, defined in the C-MAPSS paper, asymmetrically penalizes prediction errors. Late predictions (predicting failure later than actual, y^i>yi) are penalized more heavily (denominator of 13) than early predictions (denominator of 10). This reflects the higher cost associated with unexpected failures in real-world scenarios. Lower scores are better.

Justification for Metric Selection

- RMSE & MAE: Provide direct measures of prediction error magnitude.
- R2 Score: Offers a relative measure of goodness of fit.
- Custom Score: Crucial for this problem domain as it aligns with the practical preference for early predictions in maintenance.

Using this suite provides a balanced view of model accuracy, fit, and practical utility.

Visualization of Predicted vs. True RUL

A scatter plot of predicted RUL versus true RUL for the test set is generated. An ideal y=x line is often included.

- Points clustered tightly around the y=x line indicate high accuracy.
- Points above the line represent overestimations of RUL (late predictions).
- Points below the line represent underestimations of RUL (early predictions).
- The plot visually reveals any systematic biases or patterns in prediction errors.

Interpretation of Results & Critical Discussion

Comparative Analysis of Model Performance

Cross-Validation (CV) Insights

Typically, Random Forest and XGBoost significantly outperform Linear Regression on this dataset, especially in terms of R2 and RMSE. This suggests that the relationship between the sensor data (and engineered features) and RUL is non-linear and complex, which tree-based ensembles are better equipped to handle.

For FD001, due to its single operating condition and fault mode, models generally achieve better performance (e.g., higher R2, lower RMSE/Custom Score) compared to more complex datasets like FD002 or FD004. The "Damage Propagation Modeling" paper also implies that simpler scenarios should yield clearer degradation signals.

Test Set Performance vs. CV

Ideally, test set performance should be close to CV performance. A significant drop might indicate overfitting to the training set, or that the test set contains engine characteristics not well represented in the training data.

The appv0.ipynb showed a very low R2 (0.09) for FD001's test set in one run, which is atypical if features are well-engineered and the model is robust. This could be due to specific feature subsets used in that notebook instance or suboptimal hyperparameters. The app.py framework, with its interactive settings, allows for exploring this.

Strengths of the Implemented Approach

1.Structured Pipeline: The project follows a systematic data mining workflow from data ingestion to model evaluation, as outlined in the project guidelines [2].
2.Robust Modeling Choices: Random Forest and XGBoost are powerful, well-suited algorithms for this type of regression task.
3.Domain-Relevant Evaluation: The use of the custom PHM08 score alongside standard metrics ensures evaluation is aligned with practical predictive maintenance goals.
4.Effective Feature Engineering: The chosen feature engineering techniques (rolling stats, lags) are standard and effective for time-series sensor data.
5.Sound Cross-Validation: Splitting by engine unit is critical for obtaining realistic performance estimates and avoiding data leakage.

Limitations and Challenges Encountered

1.Hyperparameter Optimization: The app.py allows manual selection of some hyperparameters (e.g., n_estimators, max_depth). A more systematic approach like GridSearch or RandomizedSearch could potentially find better configurations but would be more computationally expensive.
2.Interpretability of Complex Models: While Random Forest provides feature importances, the internal workings of individual predictions are less transparent than, for example, Linear Regression.
3.Computational Resources: Training complex models, especially with many features or extensive CV, can be time-consuming. The quick_mode and subsample options in app.py are practical mitigations.
4.Piecewise Linear RUL Assumption: The RUL in the training data is calculated as a linear decrease from the point of failure. Real-world degradation might not always be linear, and some advanced models try to learn a more nuanced degradation curve.
5.Sensitivity to Window Size: The performance of rolling statistics can be sensitive to the chosen window size (5 cycles in this implementation). This is a parameter that could be tuned.