NASA CMAPSS

Sensor Validation & Fault Identification

Complete Project Notes & Beginner's Guide to ML

DRDO / GTRE | Turbofan Engine Health Monitoring | ML Pipeline v2.0

Final Result: 96% Accuracy | AUC-ROC 0.97+ | RMSE < 15 cycles

INFO

How to Read This Document

This document explains every part of the project from scratch -- no prior machine learning knowledge needed.

It covers: what the problem is, how each file works, why every decision was made, what the results mean, and how to make changes.

Read top-to-bottom the first time. After that use it as a reference -- open any file and look up what it does here.

Section 1 -- The Big Picture: What Are We Building and Why?

What is the Problem?

The DRDO/GTRE turbofan engine has 21 sensors recording data every flight cycle -- temperature, pressure, fan speed, exhaust flow. As the engine ages, these readings change in subtle ways. This project teaches a computer to recognise those changes and answer three questions automatically:

* When will this engine need maintenance? (Remaining Useful Life / RUL prediction)
* Is this engine currently entering a failure zone? (Fault Detection)
* Which specific sensor is showing abnormal behaviour? (Sensor Fault Identification)

WHY

Why This Matters -- Real-World Impact

A jet engine failure mid-flight is catastrophic. But replacing engines too early wastes crores of rupees.

If we can predict exactly when an engine needs maintenance -- not too early, not too late -- we save both lives and money.

This field is called Predictive Maintenance or Prognostics and Health Management (PHM).

The NASA CMAPSS Dataset

NASA created a simulation called CMAPSS (Commercial Modular Aero-Propulsion System Simulation) that mimics a real turbofan engine degrading over time. The dataset we use (FD001) contains:

* 100 engines that ran from brand new until they failed
* 21 sensors recording data every cycle (one cycle = one flight)
* Training set: 20,631 rows -- we know exactly when each engine failed
* Test set: 13,096 rows -- held out for honest model evaluation

Term / Concept	What It Means (Plain English)
Sensor (s1-s21)	One measuring instrument -- like a thermometer or pressure gauge. Reads a value every cycle.
Cycle	One complete flight. An engine runs for hundreds of cycles before failing.
RUL (Remaining Useful Life)	How many more cycles before this engine must be replaced. RUL=0 means failed now.
Fault label	We created this: if RUL < 30 cycles, engine is in the danger zone = Fault=1. Otherwise Fault=0.
Healthy state	The first 30% of an engine's life -- running perfectly, sensor values are stable.
Degraded state	The remaining 70% of engine life -- wear causes sensor readings to slowly shift.

Section 2 -- Project Structure: Every File Explained

Why This Structure?

A production project is organised like a factory where each department has one job. If you need to change data cleaning, you only edit one file. If you want to swap the model algorithm, you only change the model file. Everything is separated by responsibility. This is called Separation of Concerns.

File / Folder	Purpose	Milestone
`main.py`	The front door. Run this to start everything. Connects all stages together.	**All**

`config/config.yaml`	Master settings file. ALL parameters in one place. Change a number here, it affects the whole project.	**All**
`src/data/make_dataset.py`	Loads the raw NASA .txt files. Computes RUL labels for every row.	**M1**
`src/data/preprocess.py`	Cleans data: drops useless sensors, removes noise, clips outliers, scales values to [0,1].	**M1**
`src/features/build_features.py`	Creates new features: rolling statistics, health index, PCA fusion, KMeans clustering.	**M2, M3**
`src/models/train.py`	Trains all 4 models: XGBoost RUL, XGBoost Fault, Isolation Forest, LOF.	**M4**
`src/models/predict.py`	Runs trained models on new data. Contains SensorValidator for real-time use.	**M4**
`src/models/evaluate.py`	Calculates accuracy metrics. Compares predictions against true labels.	**M4**
`src/visualization/visualize.py`	Generates all 13 plots and saves as PNG files in reports/figures/.	**All**
`src/utils/helpers.py`	Shared utilities used by every module -- logging, saving/loading models, timing.	**All**
`notebooks/NASA_CMAPSS_Complete.ipynb`	Interactive version of the whole pipeline. Best place to learn.	**All**
`tests/test_pipeline.py`	25 automated unit tests verifying every function works correctly.	**All**
`scripts/setup.bat`	One-click Windows setup: creates virtual environment and installs all packages.	**Setup**

```
# Run the complete pipeline with one command:
$ python main.py all

# Or individual stages:
$ python main.py train          # train all 4 models
$ python main.py evaluate       # compute and print metrics
$ python main.py visualize      # generate all 13 plots
```

Section 3 -- Milestone M1: Data Analysis and Pre-Processing

What M1 Does

Raw sensor data is always messy. Sensors get stuck at constant values, single bad readings spike to impossible numbers, and cycle-to-cycle noise hides the real degradation signal. M1 fixes all of this through a 5-step pipeline.

Step 1 -- Loading Raw Data (make_dataset.py)

The NASA files are plain text with no column headers -- just numbers separated by spaces. The loader assigns column names and then computes the RUL label for each row of training data:

MATH

How RUL is Calculated

RUL at any cycle = max_cycle_of_this_engine - current_cycle

Example: Engine ran 200 cycles total. At cycle 150, RUL = 200 - 150 = 50 cycles remaining. We cap RUL at 125 cycles. Why? A brand-new engine looks identical whether it has 200 or 500 cycles left. Capping at 125 prevents the model wasting effort on this region where sensor readings don't yet differ.

Step 2 -- Sensor Selection (preprocess.py)

Not all 21 sensors carry useful information. We test each sensor: if its standard deviation (how much it varies) is near zero, it is constant and tells us nothing about degradation. These 7 sensors are permanently dropped:

Term / Concept	What It Means (Plain English)
Dropped (7 sensors)	s1, s5, s6, s10, s16, s18, s19 -- constant values for every engine at every cycle. Zero information content.
Kept (14 sensors)	s2, s3, s4, s7, s8, s9, s11, s12, s13, s14, s15, s17, s20, s21 -- these change as the engine degrades.
Standard Deviation	Measures spread of values. Std=0 means all values identical (useless). Std>0 means values change (useful).

Step 3 -- Outlier Clipping (preprocess.py)

Occasionally a sensor gives a physically impossible reading (electrical noise spike). These outliers confuse the model. We use IQR clipping to remove them:

```
# IQR Method -- only clips truly extreme values
Q1 = 25th percentile value   (lower quarter)
```

```
Q3 = 75th percentile value    (upper quarter)
IQR = Q3 - Q1                 (middle 50% spread)

lower_bound = Q1 - 3.0 * IQR    # anything below this is extreme
upper_bound = Q3 + 3.0 * IQR    # anything above this is extreme

# Values outside bounds are CLIPPED (not deleted) to the boundary value
# factor=3.0 is conservative -- only catches truly extreme outliers
```

Step 4 -- Rolling Mean Smoothing (preprocess.py)

Sensor readings fluctuate randomly cycle-to-cycle even when the engine is perfectly healthy. This noise hides the underlying degradation trend. We apply a 5-cycle rolling mean -- each reading is replaced by the average of itself and the 4 previous cycles:

EX **Before vs After Smoothing**

BEFORE: s7 readings might be 553, 551, 558, 550, 555, 552, 557... (noisy, hard to see trend)
AFTER: s7 smoothed values are 553, 552, 554, 553, 553, 553, 554... (clean, trend visible)
The smoothing is applied PER ENGINE -- so engine #1's last cycles don't bleed into engine #2's first cycles.

Step 5 -- MinMax Scaling (preprocess.py)

Different sensors have very different value ranges. Temperature (s3) ranges 1580-1616. Vibration (s21) ranges 8-23. If we feed raw values to the model, it would weight temperature 100x more just because the numbers are bigger -- even though both sensors matter equally.

MinMax scaling rescales every sensor to [0, 1]. After scaling: 0 = minimum recorded value, 1 = maximum recorded value. IMPORTANT: The scaler is fitted ONLY on training data, never on test data, to prevent data leakage.

Section 4 -- Milestone M2: Sensor Characterisation and Feature Engineering

What M2 Does

Now the data is clean, we analyse each sensor statistically to understand which ones actually respond to engine degradation. We also create new derived features that give the model richer information than raw sensor readings alone.

T-Test: Which Sensors Actually Change? (build_features.py)

We split each engine's life into two groups: Healthy (first 30% of cycles) and Degraded (remaining 70%). We run a statistical t-test on each sensor asking: 'Is the difference between these two groups real, or just random noise?'

STAT

What is a T-Test? (Plain English)

Imagine measuring the height of Class A students vs Class B students. A t-test tells you: is the height difference between classes genuinely real, or could it be just random chance?

p-value < 0.05 means: less than 5% chance the difference is random. We call this statistically significant.

Result for FD001: All 14 active sensors pass the t-test -- they all change meaningfully as the engine degrades.

3-Sigma Confidence Bounds (build_features.py)

Using only the healthy portion of data (first 30% of engine life), we compute the mean and standard deviation for each sensor. Then we set 3-sigma bounds -- a normal operating envelope. Any reading outside this envelope triggers an alert in the SensorValidator.

```
# Example: Sensor s7 healthy statistics
mean_healthy =  554.3   (average reading when engine is healthy)
std_healthy  =    1.8   (how much it normally varies)

lower_bound = 554.3 - (3 x 1.8) = 548.9    # 3 standard deviations below
upper_bound = 554.3 + (3 x 1.8) = 559.7    # 3 standard deviations above

# 99.7% of healthy readings fall within these bounds (empirical rule)
# Saved to: data/processed/sensor_confidence_intervals.csv
```

Health Index -- One Number Summary (build_features.py)

Instead of tracking 14 separate sensors, we compute a single composite 'Health Index' for each cycle. It is the mean z-score across all significant sensors. Near 0 = engine is healthy. Rising value = engine is degrading.

MATH

What is a Z-Score? (Plain English)

Z-score = (current value - healthy mean) / healthy standard deviation

Z-score = 0 means: reading is exactly at the healthy average. Normal.

Z-score = 2 means: reading is 2 standard deviations above normal. Slightly unusual.

Z-score = 5 means: reading is far from normal. Something is clearly wrong.

Rolling Statistics Features (build_features.py)

For each significant sensor, we compute a 10-cycle rolling mean and rolling standard deviation. The rolling standard deviation is especially powerful -- as an engine approaches failure, sensor readings become more erratic, so the rolling std INCREASES. This gives the model a direct measure of instability.

Section 5 -- Milestone M3: Data Fusion and Clustering

What M3 Does

M3 has two goals. First, use PCA to compress 14 correlated sensors into a smaller set of independent components (Data Fusion). Second, use KMeans clustering to discover whether engines operate in distinct 'states' during different flight phases.

PCA -- Principal Component Analysis (build_features.py)

Our 14 sensors are highly correlated -- when temperature rises, pressure also rises, exhaust speed also rises. This means they carry overlapping information. PCA finds new axes (principal components) along which the data varies independently, then ranks them by importance.

ANALOGY

PCA in Plain English

Imagine 14 people all describing the same elephant from slightly different angles. Most are describing the same thing with different words. PCA is a translator that combines all 14 descriptions into 3 independent, non-overlapping statements that together capture 95% of what all 14 were saying.

Result for FD001: 14 sensors compressed to 3 principal components that explain 95.2% of total variation.

PC1 captures the main degradation trend. PC2 captures operating condition differences. PC3 captures fine-grained residual variation.

KMeans Clustering -- Operating State Discovery (build_features.py)

KMeans groups data points that are 'close' to each other in sensor space. We use it to discover if engines operate in distinct regimes -- for example, high-power cruise vs low-power taxi. For FD001, the optimal number of clusters is K=2.

```
# How we find the best K (number of clusters):
# Try K = 2, 3, 4, 5, 6, 7, 8 -- compute Silhouette Score for each

Silhouette Score = how well-separated the clusters are
Score range: -1 (terrible) to +1 (perfect)
Score > 0.5 = good, well-separated clusters

# Best K = the one with the highest Silhouette Score
# FD001 result: K=2 is optimal (two operating states)
```

Section 6 -- Milestone M4: Training the Models

What M4 Does

M4 is the core machine learning stage. We train four models on the processed and engineered features, then evaluate them on completely unseen test engines. This is where the 96% accuracy comes from.

The Critical Data Split -- No Leakage (train.py)

Before training we split 100 training engines into 80 training engines and 20 test engines. This split is by ENGINE ID, not by row. This is critical:

WARNING

Why Engine-Level Split is Critical

WRONG way (row split): Engine #5 cycles 1-150 in train, cycles 151-200 in test. Model has seen engine #5 and knows its behaviour. This is cheating -- called data leakage. Gives fake accuracy.

RIGHT way (our method): ALL cycles of engine #5 are EITHER all in train OR all in test. Test engines are completely unseen.

Without this fix, we previously got R2=-1.8 and AUC=0.51 -- effectively random. With the fix: R2=0.91 and AUC=0.97.

Model 1 -- XGBoost RUL Regressor (train.py)

XGBoost builds an ensemble of 500 decision trees where each new tree corrects the mistakes of all previous trees combined. The final prediction is the sum of all tree outputs. This is called Gradient Boosting.

Term / Concept	What It Means (Plain English)
Decision Tree	A flowchart of yes/no questions: 'Is s7 > 0.7? Yes -> go right. Is health_index > 1.2? Yes -> predict RUL=15'
Gradient Boosting	Build 500 trees one-by-one. Each new tree focuses on fixing the errors of the previous 499 trees.
Early Stopping	Stop adding trees when validation error stops improving. Prevents the model memorising training data.
n_estimators=500	Maximum 500 trees allowed. With early stopping, FD001 actually stopped at ~78 trees.
learning_rate=0.05	Each new tree only contributes 5% of its correction. Slow cautious learning = better generalisation.
subsample=0.8	Each tree only sees 80% of training rows (random sample). Prevents any one tree dominating.

Model 2 -- XGBoost Fault Classifier (train.py)

Same XGBoost algorithm but for binary classification: predict Fault=0 (healthy) or Fault=1 (danger zone, RUL < 30 cycles). The main challenge is class imbalance.

BALANCE

Class Imbalance and scale_pos_weight

In FD001: ~85.5% of rows are healthy (fault=0), only ~14.5% are fault=1.

Without correction: a model that always predicts 'healthy' gets 85.5% accuracy but catches ZERO faults. Useless and dangerous.

Fix: scale_pos_weight = 85.5 / 14.5 = roughly 5.9. XGBoost treats each fault sample 5.9x more important during training.

Effect: the model pays 6x more attention to rare fault cases, dramatically improving Recall.

Model 3 -- Isolation Forest (train.py)

An unsupervised anomaly detector -- requires NO labels. The algorithm builds random trees and measures how quickly each point gets 'isolated'. Normal points are deep in dense clusters (hard to isolate). Anomalies are isolated quickly because they are rare and far from other points.

Model 4 -- Local Outlier Factor / LOF (train.py)

LOF compares each point's local density to its neighbours. If a point is in a sparse region surrounded by a dense neighbourhood, it is flagged as an outlier. LOF and Isolation Forest use different mathematics, so they catch different types of anomalies. Requiring BOTH to agree greatly reduces false alarms.

ENSEMBLE

Why Use Two Anomaly Detectors?

Isolation Forest: finds global outliers -- points far from everyone.

LOF: finds local outliers -- points that look fine globally but are unusual near their neighbours.

Ensemble rule: only flag anomaly if BOTH detectors agree (min_votes=2).

Result: IsoForest alone flagged 33. LOF alone flagged 2,912. Together (both agreed): only 22 high-confidence anomalies.

Section 7 -- Final Results and What They Mean

Performance on Unseen Test Engines

All metrics below were computed on the 20 held-out test engines that the models had NEVER seen during training. This gives an honest, unbiased estimate of real-world performance.

Metric	Achieved	Target	What It Means
RMSE	**~13 cycles**	< 15	*Average RUL prediction error is only*

			13 cycles
MAE	**~10 cycles**	< 12	*Half of all predictions are within 10 cycles of truth*
R2 Score	**~0.91**	> 0.90	*The model explains 91% of the variation in RUL*
NASA Score	**~12,000**	lower	*Asymmetric penalty score -- lower is better*
Accuracy	**96%**	> 90%	*96 out of 100 fault/healthy predictions are correct*
Recall	**~93%**	maximise	*93% of all real faults are correctly identified*
Precision	**~91%**	> 90%	*When we flag a fault, we are correct 91% of the time*
F1 Score	**~92%**	> 90%	*Balanced harmonic mean of Precision and Recall*
AUC-ROC	**~0.97**	> 0.97	*Near-perfect separation of healthy vs fault classes*

Understanding Recall vs Accuracy vs Precision

For a safety-critical application like jet engines, these three metrics are NOT equal. Here is why Recall is prioritised above all others:

Term / Concept	What It Means (Plain English)
True Positive (TP)	Engine is failing AND we correctly predicted fault=1. Correct alarm. Good.
True Negative (TN)	Engine is healthy AND we correctly predicted fault=0. Correct all-clear. Good.
False Positive (FP)	Engine is healthy but we predicted fault=1. Unnecessary maintenance. Costs money but SAFE.
False Negative (FN)	Engine is failing but we predicted fault=0. MISSED FAULT. DANGEROUS. Must minimise this.
Recall = TP/(TP+FN)	What fraction of all real faults did we catch? 93% = we missed only 7% of true failures.
AUC-ROC = 0.97	0.5 = random guessing. 1.0 = perfect. 0.97 = near-perfect fault discrimination.
NASA Score	Penalises missing a failure (predicting RUL too high) 30% more than

	over-predicting. Reflects real risk.

Section 8 -- The SensorValidator: Real-Time Fault Checking

What the SensorValidator Does

While XGBoost predicts WHEN the engine will fail (RUL), the SensorValidator answers 'WHICH sensor is behaving abnormally RIGHT NOW?' -- giving maintenance engineers a specific sensor to inspect immediately.

During training, the SensorValidator learns the 3-sigma healthy bounds for every sensor. At runtime it compares live readings against those bounds and flags any sensor outside the normal envelope.

```
# How to use SensorValidator in your own Python code:
import joblib

# Load the pre-trained validator
validator = joblib.load('models/saved_models/sensor_validator.pkl')

# Feed one complete set of sensor readings
readings = { 's2':0.712, 's3':0.891, 's7':0.634, ... }

report = validator.validate(readings)

# Check overall engine health
print(report['OVERALL'])    # 'HEALTHY' or 'FAULT DETECTED'

# Check each sensor individually
print(report['s3'])         # {'value':0.891, 'z_score':4.7, 'status':'FAULT'}
print(report['s7'])         # {'value':0.634, 'z_score':1.2, 'status':'OK'}
```

Wasserstein Distance -- Ranking Fault Severity (predict.py)

Once anomalous rows are identified, we rank sensors by how differently they behave during anomalies vs normal operation. We use Wasserstein Distance (Earth Mover's Distance) -- it measures how much 'work' it takes to reshape one distribution into another.

ANALOGY

Wasserstein Distance -- Plain English

Picture two histograms as piles of sand. Wasserstein Distance = how much effort (mass x distance) it takes to reshape one pile into the other pile's shape.

If sensor s3's histogram looks very different during normal vs anomaly operation, it has HIGH Wasserstein Distance.

High distance = this sensor changes a lot when something goes wrong = most likely cause of the fault.
Output: a ranked bar chart of sensors from 'most faulty' to 'least faulty' -- tells engineers exactly what to inspect.

Section 9 -- All 13 Plots and What They Tell You

Your plots are saved in: reports/figures/

File / Folder	Purpose	Milestone
`01_sensor_distributions.png`	Histogram of each sensor. Shows value range and shape. Bell curve = normal. Skewed = possible outlier issue.	M1
`02_sensor_trends_u1.png`	How each sensor reading changes over Engine 1's full life. You should see gradual drifting trends.	M1
`03_correlation_heatmap.png`	Which sensors move together. Dark red = highly correlated (carrying same info). Motivates PCA.	M1
`04_healthy_vs_degraded.png`	Side-by-side boxplots: healthy engine (first 30%) vs degraded (rest). Visible shift proves the sensor responds to wear.	M2
`05_pca_variance.png`	Scree plot showing how much variance each PC explains. Confirms 3 components capture 95% of information.	M3
`06_pca_2d_rul.png`	All data in PCA space, coloured by RUL. Green=high RUL (healthy), red=low RUL (near failure). Shows clear gradient.	M3
`07_kmeans_clusters.png`	KMeans cluster assignments in PCA space. Each colour = one operating state.	M3
`08_kmeans_elbow.png`	Elbow chart and Silhouette scores used to select best K. Peak Silhouette = optimal cluster count.	M3
`09_anomaly_detection.png`	Blue=normal, red=anomaly. Shows WHERE in PCA space the ensemble detector flagged unusual engine states.	M4
`10_sensor_fault_scores.png`	Bar chart ranking sensors by Wasserstein fault score. Red bars = most suspicious sensors.	M4

11_rul_prediction.png	Scatter: actual vs predicted RUL. Points close to the diagonal line = accurate predictions.	M4
12_fault_classification.png	Left: confusion matrix (TP/TN/FP/FN counts). Right: ROC curve showing AUC=0.97.	M4
13_shap_importance.png	SHAP beeswarm: which features drive the RUL model. Top feature = most important. Colour shows direction of effect.	M4

Section 10 -- How to Modify the Project (Beginner's Guide)

The Golden Rule: Change config.yaml, Not Code

Almost every parameter is controlled by config/config.yaml. You should rarely need to edit Python files for common adjustments. Here are the six most useful things to change:

1 Switch to a Different Dataset (FD002, FD003, or FD004)

In config/config.yaml change: dataset_id: 'FD002' and update train_file, test_file, rul_file to the matching filenames. Then run: python main.py all

2 Warn Earlier (Change the Fault Threshold)

To alert earlier -- when RUL < 50 cycles instead of < 30 -- change: fault_threshold_cycles: 50 in config.yaml. Retrain: python main.py train

3 Make the Model More Accurate (at the cost of speed)

Under xgb_regressor in config.yaml: increase n_estimators to 1000 and decrease learning_rate to 0.02. Then: python main.py train

4 Reduce False Anomaly Alarms

Under anomaly in config.yaml: decrease contamination from 0.05 to 0.02. This means the detector expects only 2% of data to be anomalous. Retrain to apply.

5 Re-evaluate Without Retraining

If you only change the fault_threshold_cycles, you don't need to retrain. Just run: python main.py evaluate to recompute all metrics with the new threshold.

6 Regenerate All Plots

Run: python main.py visualize -- this reads the saved predictions.csv and model files to regenerate all 13 plots without rerunning training.

Section 11 -- Complete Glossary: Every Technical Term Explained

Machine Learning Terms

Term / Concept	What It Means (Plain English)
Machine Learning (ML)	Teaching a computer to find patterns in data by showing it examples, rather than programming explicit rules.
Supervised Learning	Training with labelled examples. We provide (input data, correct answer) pairs. The model learns the pattern.
Unsupervised Learning	Training without labels. The model finds its own structure -- like grouping similar items together.
Feature	An input variable used for prediction. Our features include sensors, rolling stats, health index, PCA components.
Label / Target	The answer we want to predict. For RUL: the true number of cycles remaining. For fault: 0 or 1.
Training	Adjusting model parameters so predictions match the true labels on training data.
Overfitting	Model memorises training data perfectly but fails on new data. Like memorising exam answers without understanding.
Early Stopping	Halt training when validation performance stops improving, preventing overfitting.
Hyperparameter	Settings chosen BEFORE training (like n_estimators, learning_rate). Not learned from data -- set by you.
Pickle / .pkl	Python's way of saving any object (model, scaler, list) to a file for later reloading.
StandardScaler	Transforms features to mean=0, std=1. Applied to feature matrix before model input.
MinMaxScaler	Transforms sensor values to [0, 1]. Applied to raw sensor columns during preprocessing.
Data Leakage	When training data accidentally contains test information, causing inflated accuracy that vanishes in production.

Statistics Terms

Term / Concept	What It Means (Plain English)
Mean	The average. Sum all numbers, divide by count.
Standard Deviation (std)	How spread out values are around the mean. Small std = tightly clustered. Large std = widely spread.
Z-score	(value - mean) / std. Measures how many standard deviations a value is from normal.
IQR (Interquartile Range)	Q3 - Q1 = spread of middle 50% of values. Robust measure of variability not affected by outliers.
p-value	Probability that an observed difference happened by chance. $p < 0.05$ = statistically significant.
t-test	Statistical test comparing means of two groups to determine if the difference is real or random.
Correlation	How much two variables move together. r=+1 perfect positive, r=-1 perfect negative, r=0 no relationship.
Wasserstein Distance	How different two distributions are, measured as 'work' needed to reshape one into the other.
Silhouette Score	Measures how well-separated clusters are. Ranges from -1 to +1. Higher is better.

Project-Specific Terms

Term / Concept	What It Means (Plain English)
CMAPSS	Commercial Modular Aero-Propulsion System Simulation. The NASA turbofan degradation dataset.
RUL	Remaining Useful Life -- cycles until the engine must be replaced.
PHM	Prognostics and Health Management -- the engineering field of predicting equipment health.
PCA	Principal Component Analysis -- compresses correlated features into independent components.
KMeans	Clustering algorithm assigning each point to the nearest of K cluster centres.
XGBoost	Extreme Gradient Boosting -- an ensemble of decision trees, considered state-of-the-art for tabular ML.

Isolation Forest	Anomaly detector: builds random trees and identifies points that isolate quickly (sparse outliers).
LOF	Local Outlier Factor -- anomaly detector comparing point density to its local neighbourhood.
AUC-ROC	Area Under the Receiver Operating Characteristic Curve. 0.5=random, 1.0=perfect.
NASA Score	Asymmetric RUL loss function. Penalises predicting RUL too high (late warning) 30% more than too low.
Engine-Level Split	Splitting so all cycles of each engine are EITHER all in train OR all in test. Prevents leakage.
life_pct	Current cycle / total engine life. 0 = brand new engine, 1.0 = has just failed.
health_index	Composite score = mean z-score of significant sensors. Rises as engine degrades.
scale_pos_weight	XGBoost parameter weighting minority class (faults) more during training to handle class imbalance.
Rolling Mean	Average of sliding N-cycle window. Smooths noise while preserving trend direction.
Rolling Std	Standard deviation of sliding N-cycle window. Increases as degradation causes instability.

Section 12 -- Quick Reference: Commands You Will Actually Use

Daily Use Commands

```
# ---- First time setup (run once only) -----------------
scripts\setup.bat                   # Windows
bash scripts/setup.sh               # Linux / Mac

# ---- Every new terminal session -----------------------
venv\Scripts\activate               # Windows
source venv/bin/activate            # Linux / Mac

# ---- Main pipeline commands ---------------------------
python main.py all                  # Full pipeline: data->train->evaluate->plots
```

```
python main.py train                # Re-train all models
python main.py evaluate             # Re-evaluate metrics only (no retraining)
python main.py visualize            # Regenerate all 13 plots only

# ---- Jupyter Notebook ---------------------------------
jupyter notebook                    # Open browser-based Jupyter

# ---- Testing ------------------------------------------
pytest tests/ -v                    # Run all 25 unit tests
pytest tests/ -v -k Sensor         # Run only SensorValidator tests

# ---- Git version control ------------------------------
git add .
git commit -m 'describe your change here'
git tag M2-complete                 # Tag a milestone checkpoint
git log --oneline                   # View full history
```

VS Code Keyboard Shortcuts

Term / Concept	What It Means (Plain English)
F5	Start debugging. Runs the selected launch profile. Use 'Run Full Pipeline'.
Ctrl + Shift + P	Command palette. Type any VS Code command here (e.g. 'Python: Select Interpreter').
Ctrl + backtick	Open / close the integrated terminal without leaving VS Code.
Ctrl + click	Jump to any function definition. Great for exploring how functions connect.
Ctrl + Shift + F	Search all files in the project for any text or function name.
F12	Go to definition of whatever your cursor is resting on.
Ctrl + Z	Undo last change in the editor.
Alt + Up/Down	Move the current line up or down without cut-paste.

Project Complete

96% Accuracy | AUC-ROC 0.97 | RMSE < 15 cycles | M1 through M4 Delivered

NASA CMAPSS | DRDO / GTRE | Turbofan Engine Health Monitoring