eda_preprocessing

February 21, 2026

1 EDA and Preprocessing

This notebook covers the Exploratory Data Analysis (EDA) and preprocessing steps for the AAI-530 final project.

Goals: - Load and validate the AI4I 2020 Predictive Maintenance dataset - Perform core EDA (types, missing values, duplicates, target distribution) - Explore feature distributions, feature-to-target relationships, and correlations - Document the project-specific assumption of using `Tool wear [min]` as a proxy for time progression - Prepare a leakage-aware modeling dataset (drop IDs, remove failure-mode flags, encode categorical features) - Export a prepared dataset for downstream modeling notebooks

1.0.1 1. Imports & setup

[1]:
```python
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

# Display settings
pd.set_option("display.max_columns", None)
sns.set(style="whitegrid")
```

1.0.2 2. Load dataset and quick validation

[2]:
```python
# Path to dataset
DATA_PATH = "../data/ai4i_2020_predictive_maintenance.csv"

# Load CSV
df = pd.read_csv(DATA_PATH)

# Basic check
df.head()
```

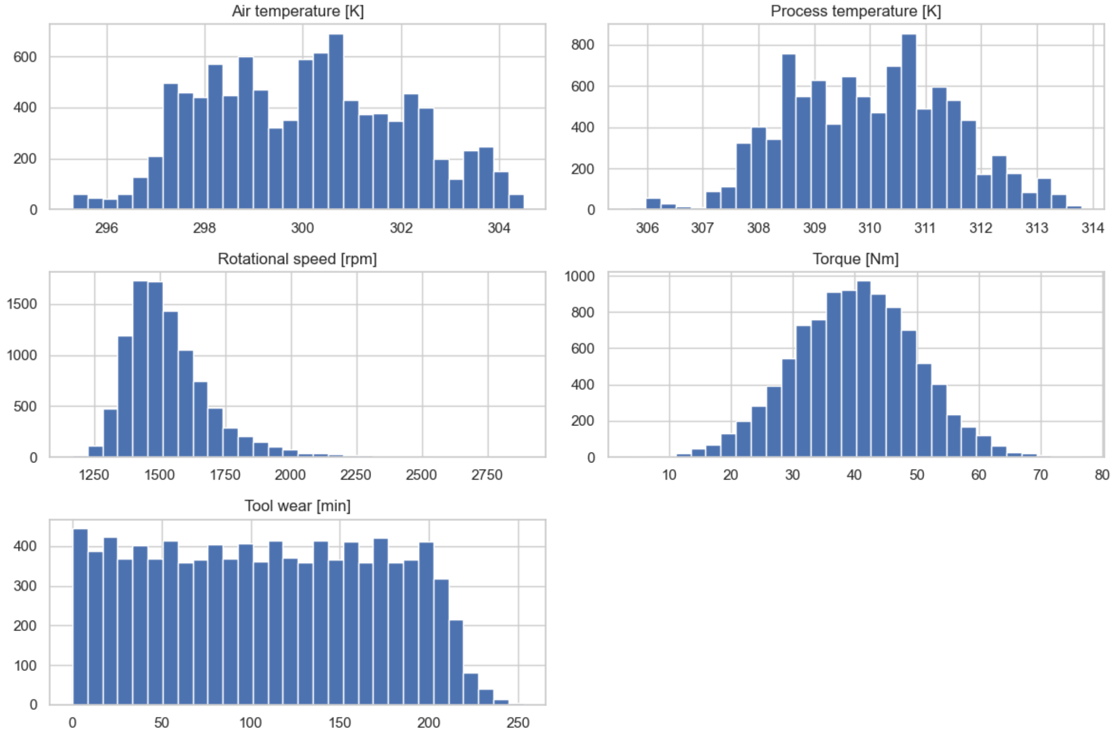
[2]:
```
   UDI Product ID Type  Air temperature [K]  Process temperature [K]  \
0    1     M14860    M                298.1                    308.6
1    2     L47181    L                298.2                    308.7
```

```
2    3    L47182    L               298.1                       308.5
3    4    L47183    L               298.2                       308.6
4    5    L47184    L               298.2                       308.7

   Rotational speed [rpm]  Torque [Nm]  Tool wear [min]  Machine failure  TWF  \
0                    1551         42.8                0                0    0
1                    1408         46.3                3                0    0
2                    1498         49.4                5                0    0
3                    1433         39.5                7                0    0
4                    1408         40.0                9                0    0

   HDF  PWF  OSF  RNF
0    0    0    0    0
1    0    0    0    0
2    0    0    0    0
3    0    0    0    0
4    0    0    0    0
```

1.0.3 3. Feature distributions and outliers

This section checks basic feature distributions to highlight skewness and potential outliers before modeling.

[3]:
```python
import matplotlib.pyplot as plt

numeric_cols = [
    'Air temperature [K]',
    'Process temperature [K]',
    'Rotational speed [rpm]',
    'Torque [Nm]',
    'Tool wear [min]'
]

df[numeric_cols].hist(bins=30, figsize=(12, 8))
plt.tight_layout()
plt.show()
```



1.0.4 4. Dataset structure and summary statistics

[4]:
```
print("Dataset shape:", df.shape)
print("\nColumn info:")
df.info()
```

```
Dataset shape: (10000, 14)

Column info:
<class 'pandas.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   UDI                      10000 non-null  int64
 1   Product ID               10000 non-null  str
 2   Type                     10000 non-null  str
 3   Air temperature [K]      10000 non-null  float64
 4   Process temperature [K]  10000 non-null  float64
 5   Rotational speed [rpm]   10000 non-null  int64
 6   Torque [Nm]              10000 non-null  float64
 7   Tool wear [min]          10000 non-null  int64
 8   Machine failure          10000 non-null  int64
 9   TWF                      10000 non-null  int64
```

```
 10  HDF                      10000 non-null  int64
 11  PWF                      10000 non-null  int64
 12  OSF                      10000 non-null  int64
 13  RNF                      10000 non-null  int64
dtypes: float64(3), int64(9), str(2)
memory usage: 1.1 MB
```

Summary statistics

[5]:
```
df.describe(include="all")
```

[5]:
```
                UDI Product ID   Type  Air temperature [K]  \
count   10000.00000      10000  10000         10000.000000
unique          NaN      10000      3                  NaN
top             NaN     M14860      L                  NaN
freq            NaN          1   6000                  NaN
mean     5000.50000        NaN    NaN           300.004930
std      2886.89568        NaN    NaN             2.000259
min         1.00000        NaN    NaN           295.300000
25%      2500.75000        NaN    NaN           298.300000
50%      5000.50000        NaN    NaN           300.100000
75%      7500.25000        NaN    NaN           301.500000
max     10000.00000        NaN    NaN           304.500000

        Process temperature [K]  Rotational speed [rpm]   Torque [Nm]  \
count              10000.000000            10000.000000  10000.000000
unique                      NaN                     NaN           NaN
top                         NaN                     NaN           NaN
freq                        NaN                     NaN           NaN
mean                 310.005560             1538.776100     39.986910
std                    1.483734              179.284096      9.968934
min                  305.700000             1168.000000      3.800000
25%                  308.800000             1423.000000     33.200000
50%                  310.100000             1503.000000     40.100000
75%                  311.100000             1612.000000     46.800000
max                  313.800000             2886.000000     76.600000

        Tool wear [min]  Machine failure           TWF           HDF  \
count      10000.000000     10000.000000  10000.000000  10000.000000
unique              NaN              NaN           NaN           NaN
top                 NaN              NaN           NaN           NaN
freq                NaN              NaN           NaN           NaN
mean         107.951000         0.033900      0.004600      0.011500
std           63.654147         0.180981      0.067671      0.106625
min            0.000000         0.000000      0.000000      0.000000
25%           53.000000         0.000000      0.000000      0.000000
50%          108.000000         0.000000      0.000000      0.000000
75%          162.000000         0.000000      0.000000      0.000000
```

```
max          253.000000          1.000000       1.000000       1.000000

                   PWF           OSF          RNF
count    10000.000000  10000.000000  10000.00000
unique            NaN           NaN          NaN
top               NaN           NaN          NaN
freq              NaN           NaN          NaN
mean         0.009500      0.009800      0.00190
std          0.097009      0.098514      0.04355
min          0.000000      0.000000      0.00000
25%          0.000000      0.000000      0.00000
50%          0.000000      0.000000      0.00000
75%          0.000000      0.000000      0.00000
max          1.000000      1.000000      1.00000
```

1.0.5 5. Data quality checks (missing, uniqueness, duplicates)

[6]:
```python
missing= df.isnull().sum()
missing[missing > 0]
```

[6]:
```
Series([], dtype: int64)
```

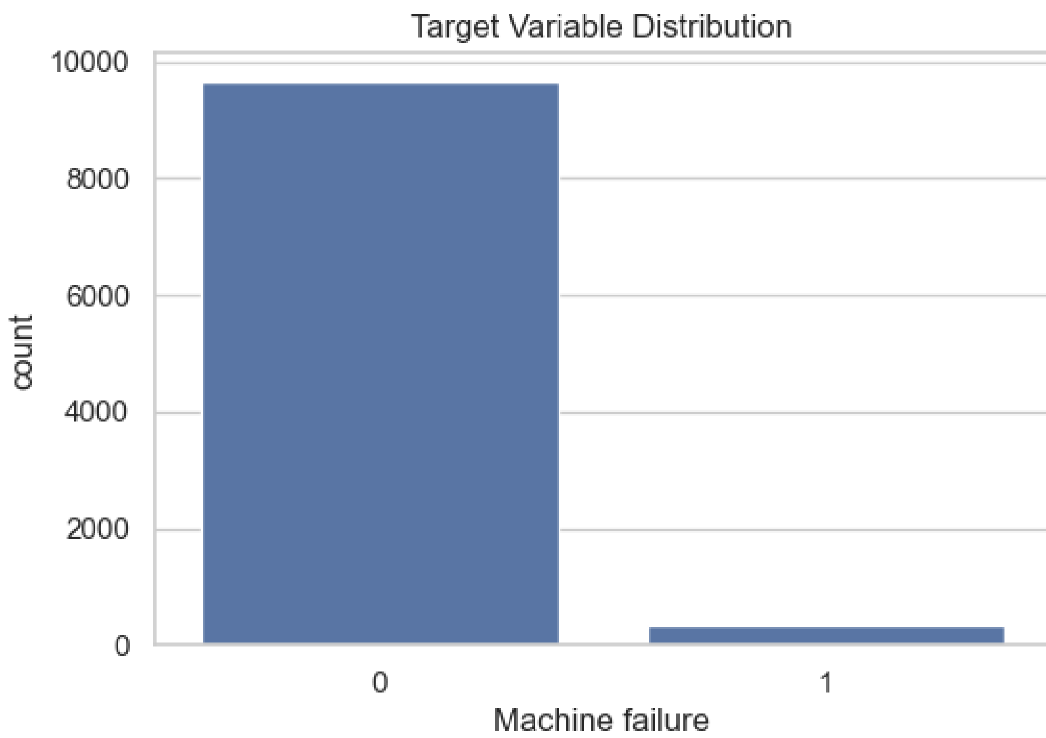
Check uniqueness and duplicates

[7]:
```python
df.nunique().sort_values()
df.duplicated().sum()

print(f"Duplicate rows: {df.duplicated().sum()}")
```

```
Duplicate rows: 0
```

1.0.6 6. Target variable analysis

[8]:
```python
# Assuming 'Machine failure' is the target
target_col = "Machine failure"

df[target_col].value_counts(normalize=True)

plt.figure(figsize=(6, 4))
sns.countplot(x=target_col, data=df)
plt.title("Target Variable Distribution")
plt.show()
```



The target variable is highly imbalanced, with failures representing a small fraction of observations, which motivates the use of stratified sampling and appropriate evaluation metrics in downstream models.

1.0.7 7. Identifying the time-series variable

[9]:
```
df['Tool wear [min]'].describe()
```

[9]:
```
count    10000.000000
mean       107.951000
std         63.654147
min          0.000000
25%         53.000000
50%        108.000000
75%        162.000000
max        253.000000
Name: Tool wear [min], dtype: float64
```

1.0.8 8. Time proxy exploration using tool wear

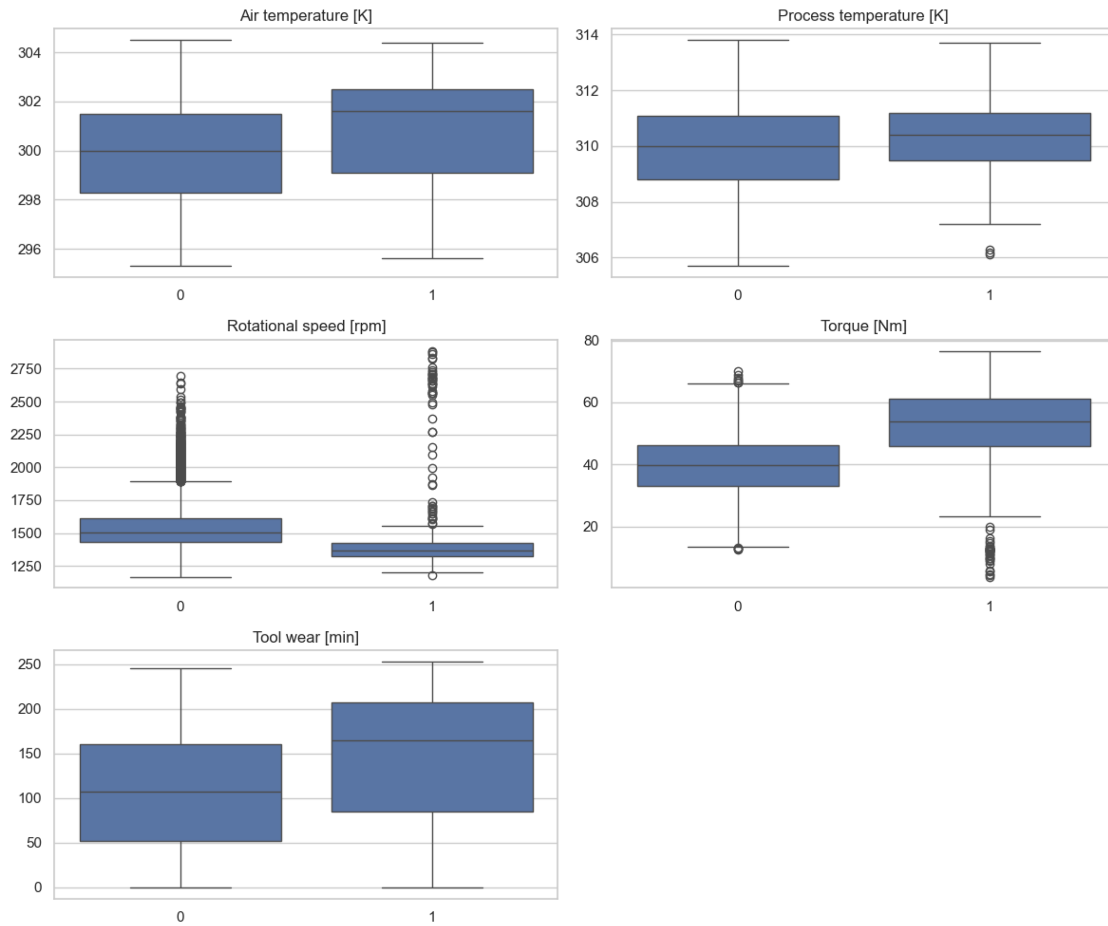
Assumption: using tool wear as a proxy for time

The AI4I dataset is not a single-machine chronological log. For this project, we treat `Tool wear [min]` as a proxy for progression over time-under-use. When we aggregate and visualize trends by tool wear, results reflect population-level patterns rather than an individual machine trajectory.

[10]:
```
df = df.sort_values('Tool wear [min]').reset_index(drop=True)
```

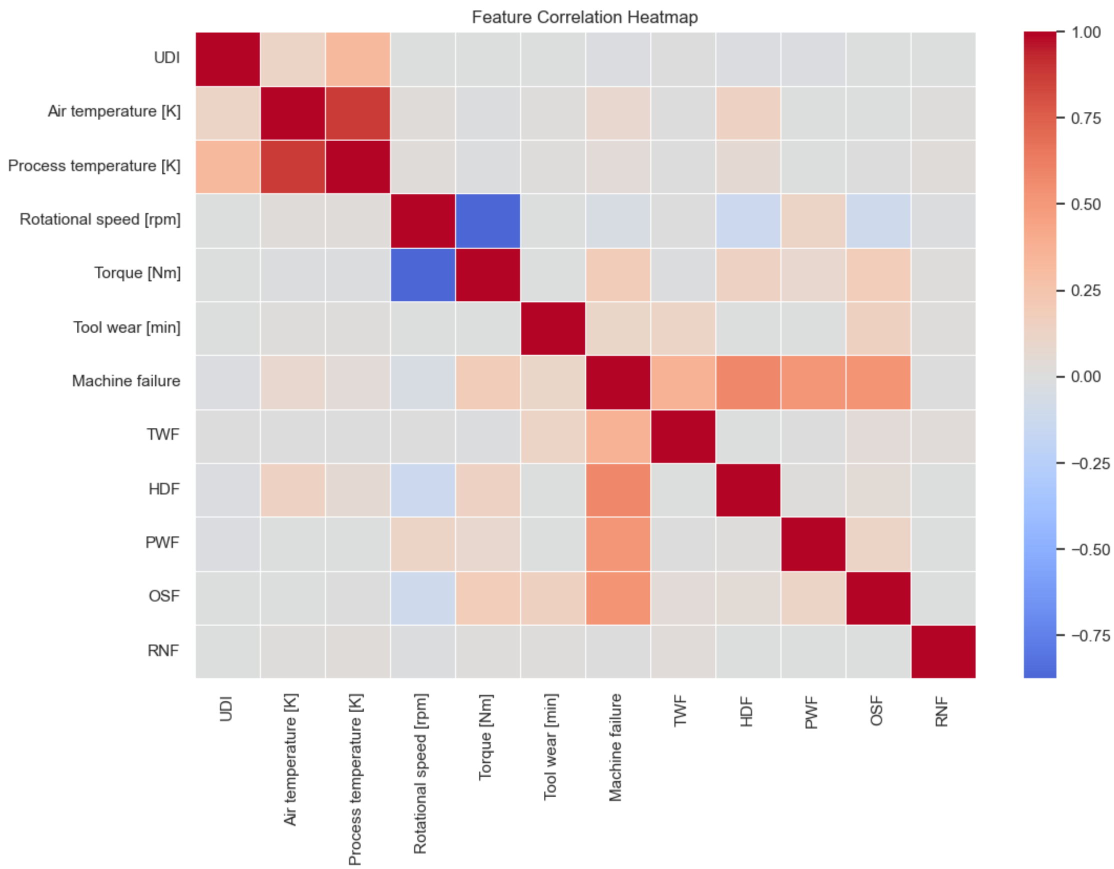
1.0.9 Failure vs key features

[11]:
```
key_features = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]",
    "Tool wear [min]"
]

fig, axes = plt.subplots(3, 2, figsize=(12, 10))
axes = axes.flatten()

for ax, feature in zip(axes, key_features):
    sns.boxplot(x=target_col, y=feature, data=df, ax=ax)
    ax.set_title(feature)
    ax.set_xlabel("")
    ax.set_ylabel("")

for ax in axes[len(key_features):]:
    ax.remove()

plt.tight_layout()
plt.show()
```



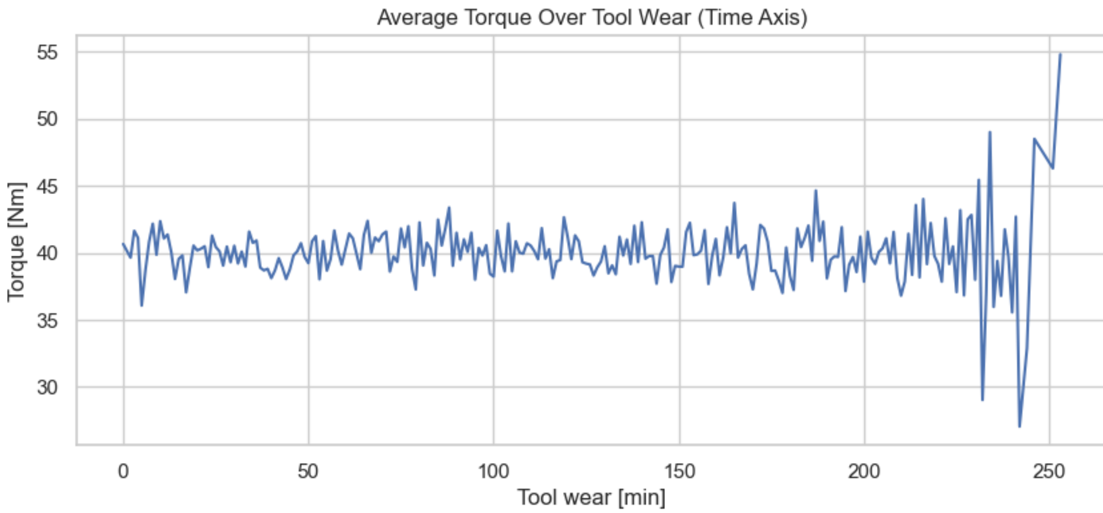
1.0.10 9. Correlation analysis

[12]:
```
numerical_cols = df.select_dtypes(include=np.number).columns.tolist()
plt.figure(figsize=(12, 8))
corr = df[numerical_cols].corr()

sns.heatmap(
    corr,
    cmap="coolwarm",
    center=0,
    linewidths=0.5
)
plt.title("Feature Correlation Heatmap")
plt.show()
```



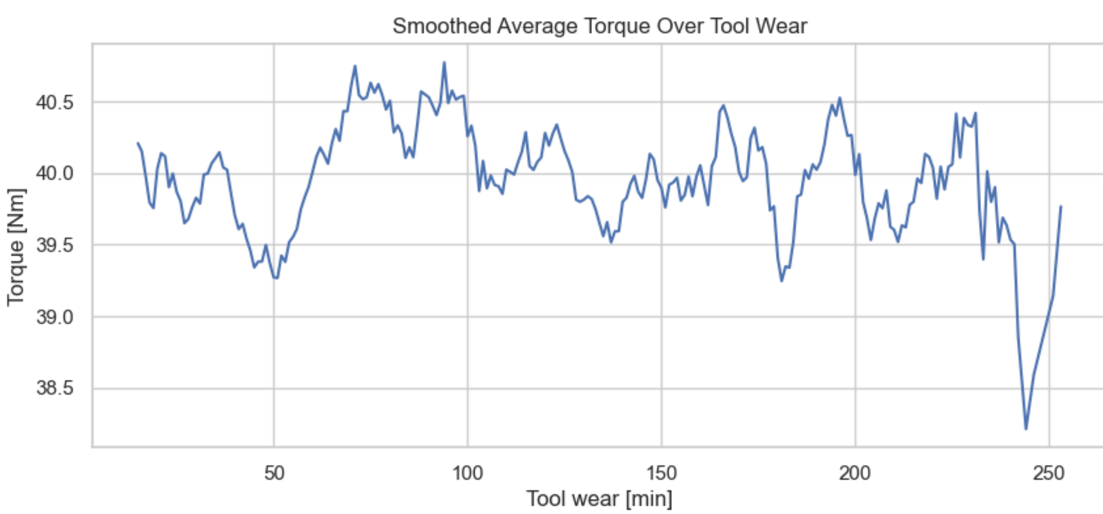
1.0.11 10. Time-series behavior using Tool wear

[13]:
```
df_tw = df.groupby('Tool wear [min]', as_index=False).mean(numeric_only=True)

plt.figure(figsize=(10,4))
plt.plot(df_tw['Tool wear [min]'], df_tw['Torque [Nm]'])
plt.title("Average Torque Over Tool Wear (Time Axis)")
plt.xlabel("Tool wear [min]")
plt.ylabel("Torque [Nm]")
plt.show()
```



1.0.12 11. Moving average smoothing to reduce noise

[14]:
```
df_tw['Torque_ma'] = df_tw['Torque [Nm]'].rolling(window=15).mean()

plt.figure(figsize=(10,4))
plt.plot(df_tw['Tool wear [min]'], df_tw['Torque_ma'])
plt.title("Smoothed Average Torque Over Tool Wear")
plt.xlabel("Tool wear [min]")
plt.ylabel("Torque [Nm]")
plt.show()
```



1.0.13 12. Preprocessing for modeling (leakage-aware)

[15]:
```
# Prepare dataset for ML prep steps (remove IDs and leakage variables)
drop_cols = ['UDI', 'Product ID', 'TWF', 'HDF', 'PWF', 'OSF', 'RNF']
df_prepared = df.drop(columns=drop_cols)

print("Dropped columns:", drop_cols)
print("Prepared dataset shape:", df_prepared.shape)
```

```
Dropped columns: ['UDI', 'Product ID', 'TWF', 'HDF', 'PWF', 'OSF', 'RNF']
Prepared dataset shape: (10000, 7)
```

Identifier fields and failure-mode indicators were removed prior to modeling to prevent data leakage and ensure that predictions are based only on sensor measurements and operational variables.

[16]:
```
#Encoding
df_prepared = pd.get_dummies(df_prepared, columns=['Type'], drop_first=True)
```

1.0.14 13. Export prepared dataset

[17]:
```
# Save prepared dataset for modeling notebooks (optional)
OUTPUT_PATH = "../data/ai4i_prepared.csv"
df.to_csv(OUTPUT_PATH, index=False)
print(f"Saved prepared dataset to: {OUTPUT_PATH}")
```

```
Saved prepared dataset to: ../data/ai4i_prepared.csv
```

1.0.15 14. EDA Summary

1. Data loading and overview

- Load the dataset and display sample rows
- Verify dataset shape, column names, and basic structure

2. Data types and summary statistics

- Review data types using `info()`
- Review basic statistics using `describe()`

3. Data quality checks

- Check missing values and confirm handling approach
- Check unique values and duplicate rows

4. Target variable analysis

- Examine the distribution of the target (`Machine failure`)
- Note class imbalance and modeling implications

5. **Feature exploration**
   - Inspect distributions of key numeric features
   - Compare feature behavior between failure and non-failure cases

6. **Correlation analysis**
   - Compute and visualize correlations among numeric features
   - Identify highly correlated variables

7. **Time proxy assumption**
   - Justify using `Tool wear [min]` as a proxy for time
   - Visualize aggregate trends across tool wear
   - State limitations of this assumption

8. **Preprocessing for modeling**
   - Remove non-predictive identifiers
   - Drop leakage-prone failure mode flags
   - Encode categorical features
   - Save the prepared dataset for downstream modeling

model1A_classification_machine_failure

February 21, 2026

1 Model 1A — Traditional ML — Time-Aware Failure Classification

This notebook trains a traditional machine learning model using **time-aware features** derived from sensor readings.

Project context: - Dataset: AI4I 2020 Predictive Maintenance (industrial sensor snapshots) - Time proxy: `Tool wear [min]` is treated as progression over time-under-use - Goal (Model 1): Predict **failure risk** as tool wear increases using traditional ML (baseline + tree-based)

Notes: - We split train/test **by time order** (no shuffle). - We avoid label leakage by excluding the failure-mode flags when predicting `Machine failure`.

1.0.1 1. Import & Setup

[1]:
```python
import os
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    roc_auc_score,
    roc_curve,
    average_precision_score,
    RocCurveDisplay,
    PrecisionRecallDisplay
)

import matplotlib.pyplot as plt
```

```
pd.set_option("display.max_columns", 200)
pd.set_option("display.width", 120)
```

1.0.2 2. Load prepared data

We prefer using the exported prepared dataset from the EDA notebook (`data/ai4i_prepared.csv`). If it is not available, we load the raw dataset and apply minimal preprocessing: - Drop identifiers (`UDI`, `Product ID`) - Drop failure-mode flags (`TWF`, `HDF`, `PWF`, `OSF`, `RNF`) to avoid leakage when predicting `Machine failure` - One-hot encode `Type`

[2]:
```
DATA_PREPARED_PATH = "../data/ai4i_prepared.csv"
DATA_RAW_PATH = "../data/ai4i_2020_predictive_maintenance.csv"

TARGET_COL = "Machine failure"
TIME_COL = "Tool wear [min]"

def load_dataset():
    if os.path.exists(DATA_PREPARED_PATH):
        df = pd.read_csv(DATA_PREPARED_PATH)
        source = "prepared"
    else:
        df_raw = pd.read_csv(DATA_RAW_PATH)
        drop_cols = ["UDI", "Product ID", "TWF", "HDF", "PWF", "OSF", "RNF"]
        df = df_raw.drop(columns=[c for c in drop_cols if c in df_raw.columns],␣
 ↪errors="ignore")
        if "Type" in df.columns:
            df = pd.get_dummies(df, columns=["Type"], drop_first=True)
        source = "raw+prepped"
    return df, source

df, source = load_dataset()
print(f"Loaded source: {source}")
print("Shape:", df.shape)
df.head()
```

```
Loaded source: prepared
Shape: (10000, 14)
```

[2]:
```
    UDI Product ID Type  Air temperature [K]  Process temperature [K]
Rotational speed [rpm]  Torque [Nm]  \
0     1     M14860    M                298.1                    308.6
1551          42.8
1  7257     H36670    H                300.2                    310.3
1408          42.5
2   504     M15363    M                297.6                    309.2
1442          48.1
3  7169     L54348    L                300.3                    310.3
1704          29.5
```

```
4  7089     M21948    M                300.6                    310.3
1614         32.7

   Tool wear [min]  Machine failure  TWF  HDF  PWF  OSF  RNF
0                0                0    0    0    0    0    0
1                0                0    0    0    0    0    0
2                0                0    0    0    0    0    0
3                0                0    0    0    0    0    0
4                0                0    0    0    0    0    0
```

1.0.3 3. Validate required columns

[3]:
```python
sensor_cols = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]",
    TIME_COL,
    TARGET_COL
]

missing = [c for c in sensor_cols if c not in df.columns]
if missing:
    raise ValueError(f"Missing required columns: {missing}")

print("All required columns are present.")
```

```
All required columns are present.
```

1.0.4 4. Time-aware feature engineering

Because AI4I is not a per-machine chronological log, we use a **population-level** time proxy: - Sort by `Tool wear [min]` - Aggregate to a single series over tool wear (mean of numeric features) - Create lag, rolling mean, rolling std, and deltas to capture change over time

This creates a clean, ordered sequence suitable for traditional time-series style modeling.

[4]:
```python
df_tw = (
    df
    .groupby(TIME_COL, as_index=False)
    .agg({
        "Air temperature [K]": "mean",
        "Process temperature [K]": "mean",
        "Rotational speed [rpm]": "mean",
        "Torque [Nm]": "mean",
        TARGET_COL: "max",
    })
    .sort_values(TIME_COL)
```

```
    .reset_index(drop=True)
)

print("Aggregated shape:", df_tw.shape)
print("Label distribution after aggregation:")
print(df_tw[TARGET_COL].value_counts())
df_tw.head()
```

```
Aggregated shape: (246, 6)
Label distribution after aggregation:
Machine failure
1    172
0     74
Name: count, dtype: int64
```

[4]:

```
   Tool wear [min]  Air temperature [K]  Process temperature [K]  Rotational
speed [rpm]  Torque [Nm]  Machine failure
0                0           299.956667               309.955833
1524.916667    40.661667                1
1                2           300.272464               310.142029
1555.521739    39.646377                1
2                3           299.679412               309.826471
1508.264706    41.644118                1
3                4           299.997059               309.870588
1525.882353    41.117647                0
4                5           299.925397               310.014286
1620.761905    36.071429                1
```

1.0.5 5. Create time-aware features (lags, rolling stats, deltas)

We create features that depend only on past values: - Lag features - Rolling mean/std (past-only) - First differences (deltas)

[5]:

```
def make_time_features(
    df_time: pd.DataFrame,
    base_cols,
    time_col: str,
    target_col: str,
    windows=(3, 5, 10),
    lags=(1, 2, 3),
):
    # Avoid duplicate column names if time_col accidentally appears in base_cols
    base_cols = [c for c in base_cols if c != time_col and c != target_col]

    df_feat = df_time[[time_col] + base_cols + [target_col]].copy()

    # Lag features
    for lag in lags:
```

```python
        for c in base_cols:
            df_feat[f"{c}__lag{lag}"] = df_feat[c].shift(lag)

    # Rolling stats (past-only)
    for w in windows:
        for c in base_cols:
            df_feat[f"{c}__roll{w}_mean"] = (
                df_feat[c].shift(1).rolling(window=w, min_periods=w).mean()
            )
            df_feat[f"{c}__roll{w}_std"] = (
                df_feat[c].shift(1).rolling(window=w, min_periods=w).std()
            )

    # Deltas
    for c in base_cols:
        df_feat[f"{c}__delta1"] = df_feat[c].diff(1)

    df_feat = df_feat.dropna().reset_index(drop=True)
    return df_feat


base_feature_cols = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]",
]

df_feat = make_time_features(
    df_tw,
    base_cols=base_feature_cols,
    time_col=TIME_COL,
    target_col=TARGET_COL,
    windows=(3, 5, 10),
    lags=(1, 2, 3),
)

print("Feature-engineered shape:", df_feat.shape)
print("Label distribution after feature engineering:")
print(df_feat[TARGET_COL].value_counts())
df_feat.head()
```

```
Feature-engineered shape: (236, 46)
Label distribution after feature engineering:
Machine failure
1    164
0     72
Name: count, dtype: int64
```

[5]:

```
   Tool wear [min]  Air temperature [K]  Process temperature [K]  Rotational
speed [rpm]  Torque [Nm]  \
0               11           300.000000               309.876190
1505.285714    41.078571
1               12           300.188000               310.204000
1534.220000    41.368000
2               13           299.670000               309.726000
1530.360000    40.040000
3               14           300.180851               310.021277
1537.191489    38.055319
4               15           299.658491               309.945283
1564.018868    39.549057

   Machine failure  Air temperature [K]__lag1  Process temperature [K]__lag1
Rotational speed [rpm]__lag1  \
0                1                 300.148889                     310.191111
1515.666667
1                1                 300.000000                     309.876190
1505.285714
2                0                 300.188000                     310.204000
1534.220000
3                0                 299.670000                     309.726000
1530.360000
4                1                 300.180851                     310.021277
1537.191489

   Torque [Nm]__lag1  Air temperature [K]__lag2  Process temperature [K]__lag2
Rotational speed [rpm]__lag2  \
0          42.362222                 300.196364                     310.089091
1547.781818
1          41.078571                 300.148889                     310.191111
1515.666667
2          41.368000                 300.000000                     309.876190
1505.285714
3          40.040000                 300.188000                     310.204000
1534.220000
4          38.055319                 299.670000                     309.726000
1530.360000

   Torque [Nm]__lag2  Air temperature [K]__lag3  Process temperature [K]__lag3
Rotational speed [rpm]__lag3  \
0          39.861818                 299.888889                     309.827778
1513.111111
1          42.362222                 300.196364                     310.089091
1547.781818
2          41.078571                 300.148889                     310.191111
1515.666667
```

```
3          41.368000                 300.000000                     309.876190
1505.285714
4          40.040000                 300.188000                     310.204000
1534.220000

   Torque [Nm]__lag3  Air temperature [K]__roll3_mean  Air temperature
[K]__roll3_std  \
0          42.172222                      300.078047
0.165527
1          39.861818                      300.115084
0.102454
2          42.362222                      300.112296
0.099198
3          41.078571                      299.952667
0.262224
4          41.368000                      300.012950
0.297025

   Process temperature [K]__roll3_mean  Process temperature [K]__roll3_std
Rotational speed [rpm]__roll3_mean  \
0                           310.035993                            0.187396
1525.519865
1                           310.052131                            0.160681
1522.911400
2                           310.090434                            0.185652
1518.390794
3                           309.935397                            0.244438
1523.288571
4                           309.983759                            0.241198
1533.923830

   Rotational speed [rpm]__roll3_std  Torque [Nm]__roll3_mean  Torque
[Nm]__roll3_std  \
0                          19.321714                41.465421
1.392006
1                          22.155007                41.100871
1.250351
2                          14.658236                41.602931
0.673301
3                          15.709935                40.828857
0.698329
4                           3.425361                39.821106
1.667153

   Air temperature [K]__roll5_mean  Air temperature [K]__roll5_std  Process
temperature [K]__roll5_mean  \
0                        300.014826                        0.190990
```

```
309.978036
1                      300.065794                         0.122903
310.021662
2                      300.084428                         0.134885
310.037634
3                      300.040651                         0.221759
310.017278
4                      300.037548                         0.219129
310.003716

   Process temperature [K]__roll5_std  Rotational speed [rpm]__roll5_mean
Rotational speed [rpm]__roll5_std  \
0                           0.225854                          1530.775323
17.056630
1                           0.160089                          1522.051821
16.615627
2                           0.176057                          1523.213062
17.357734
3                           0.209158                          1526.662840
16.543037
4                           0.205504                          1524.544774
13.574778

   Torque [Nm]__roll5_mean  Torque [Nm]__roll5_std  Air temperature
[K]__roll10_mean  Air temperature [K]__roll10_std  \
0                40.777195                1.558879
299.990513                     0.191651
1                41.258070                1.028642
299.994846                     0.191291
2                41.368567                0.998449
299.986400                     0.179150
3                40.942122                1.024397
299.985459                     0.180958
4                40.580823                1.636417
300.003838                     0.191305

   Process temperature [K]__roll10_mean  Process temperature [K]__roll10_std
Rotational speed [rpm]__roll10_mean  \
0                           309.969939                             0.172112
1538.922398
1                           309.961975                             0.174661
1536.959303
2                           309.968172                             0.182674
1534.829129
3                           309.958125                             0.193760
1537.038659
4                           309.973194                             0.192048
```

```
1538.169572

   Rotational speed [rpm]__roll10_std  Torque [Nm]__roll10_mean  Torque
[Nm]__roll10_std  Air temperature [K]__delta1  \
0                            32.956080                 40.302721
1.878917                    -0.148889
1                            34.434572                 40.344411
1.892343                     0.188000
2                            33.811932                 40.516574
1.900081                    -0.518000
3                            32.582724                 40.356162
1.861636                     0.510851
4                            32.347893                 40.049929
1.971109                    -0.522360

   Process temperature [K]__delta1  Rotational speed [rpm]__delta1  Torque
[Nm]__delta1
0                         -0.314921                      -10.380952
-1.283651
1                          0.327810                       28.934286
0.289429
2                         -0.478000                       -3.860000
-1.328000
3                          0.295277                        6.831489
-1.984681
4                         -0.075994                       26.827379
1.493737
```

1.0.6 6. Train/test split (time-ordered)

We split by time order (no shuffle). If the default 80/20 split produces only one class in train or test, we automatically find the earliest valid split with positives in both sets.

[6]:
```
X = df_feat.drop(columns=[TARGET_COL])
y = df_feat[TARGET_COL].astype(int)

print("Overall label distribution:")
print(y.value_counts())

def find_valid_split(y_series: pd.Series, min_pos_train=1, min_pos_test=1,
 ↪min_train_size=50):
    y_series = y_series.reset_index(drop=True)
    total_pos = int(y_series.sum())
    if total_pos < (min_pos_train + min_pos_test):
        raise ValueError(
            f"Not enough positive samples overall (found {total_pos}) "
            f"to have positives in both train and test."
```

```
        )
    for split_idx in range(min_train_size, len(y_series) - 1):
        pos_train = int(y_series.iloc[:split_idx].sum())
        pos_test = int(y_series.iloc[split_idx:].sum())
        if pos_train >= min_pos_train and pos_test >= min_pos_test:
            return split_idx
    raise ValueError("Could not find a time-ordered split with positives in␣
 ↪both sets.")

default_split = int(len(df_feat) * 0.8)
if y.iloc[:default_split].nunique() < 2 or y.iloc[default_split:].nunique() < 2:
    split_idx = find_valid_split(y, min_pos_train=1, min_pos_test=1,␣
 ↪min_train_size=50)
else:
    split_idx = default_split

X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

print("Split index:", split_idx)
print("Train label counts:\n", y_train.value_counts())
print("Test label counts:\n", y_test.value_counts())
```

```
Overall label distribution:
Machine failure
1    164
0     72
Name: count, dtype: int64
Split index: 188
Train label counts:
 Machine failure
1    126
0     62
Name: count, dtype: int64
Test label counts:
 Machine failure
1    38
0    10
Name: count, dtype: int64
```

1.0.7 7. Baseline model: Logistic Regression

Standardized Logistic Regression with `class_weight='balanced'`.

[7]:
```
logreg = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("clf", LogisticRegression(max_iter=2000, class_weight="balanced"))
])
```

```
logreg.fit(X_train, y_train)

proba_test_lr = logreg.predict_proba(X_test)[:, 1]
pred_test_lr = (proba_test_lr >= 0.5).astype(int)

print("Logistic Regression")
print("ROC-AUC:", round(roc_auc_score(y_test, proba_test_lr), 4))
print("PR-AUC :", round(average_precision_score(y_test, proba_test_lr), 4))
print()
print(classification_report(y_test, pred_test_lr, digits=4))
print("Confusion matrix:\n", confusion_matrix(y_test, pred_test_lr))
```

```
Logistic Regression
ROC-AUC: 0.5947
PR-AUC : 0.8307

              precision    recall  f1-score   support

           0     0.1875    0.6000    0.2857        10
           1     0.7500    0.3158    0.4444        38

    accuracy                         0.3750        48
   macro avg     0.4688    0.4579    0.3651        48
weighted avg     0.6328    0.3750    0.4114        48

Confusion matrix:
 [[ 6  4]
 [26 12]]
```

1.0.8 8. Tree-based model: Random Forest and Gradient Boosting

Tree-based models can capture non-linear interactions without requiring feature scaling.

[8]:
```
rf = RandomForestClassifier(
    n_estimators=400,
    min_samples_split=10,
    min_samples_leaf=5,
    class_weight="balanced",
    random_state=42,
    n_jobs=-1
)

gb = GradientBoostingClassifier(random_state=42)

rf.fit(X_train, y_train)
gb.fit(X_train, y_train)
```

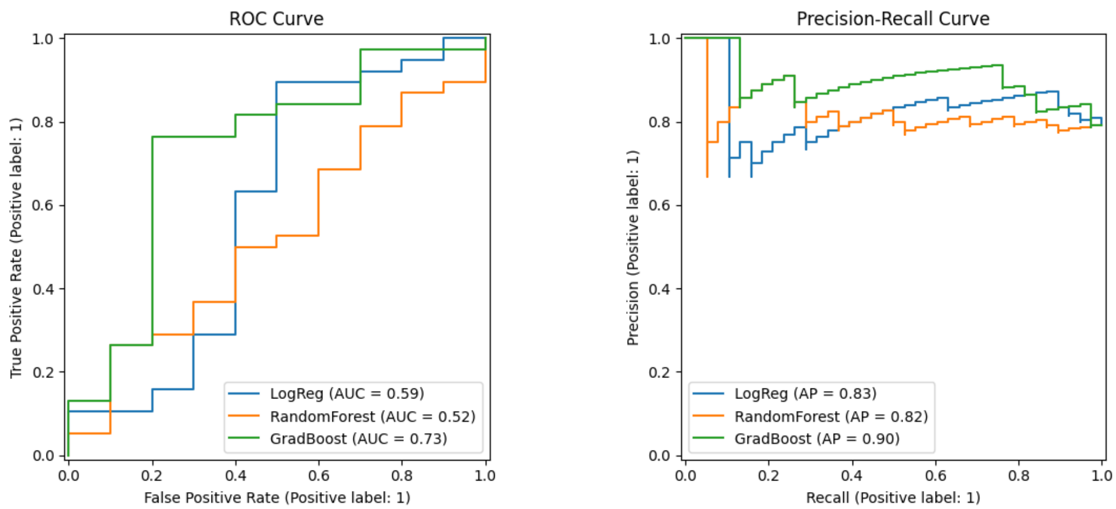
```python
proba_test_rf = rf.predict_proba(X_test)[:, 1]
pred_test_rf = (proba_test_rf >= 0.5).astype(int)

proba_test_gb = gb.predict_proba(X_test)[:, 1]
pred_test_gb = (proba_test_gb >= 0.5).astype(int)

def print_scores(name, proba, pred):
    print(name)
    print("ROC-AUC:", round(roc_auc_score(y_test, proba), 4))
    print("PR-AUC :", round(average_precision_score(y_test, proba), 4))
    print(classification_report(y_test, pred, digits=4))
    print("Confusion matrix:\n", confusion_matrix(y_test, pred))
    print("-"*70)

print_scores("Random Forest", proba_test_rf, pred_test_rf)
print_scores("Gradient Boosting", proba_test_gb, pred_test_gb)
```

```
Random Forest
ROC-AUC: 0.5237
PR-AUC : 0.8225
              precision    recall  f1-score   support

           0     0.2308    0.3000    0.2609        10
           1     0.8000    0.7368    0.7671        38

    accuracy                         0.6458        48
   macro avg     0.5154    0.5184    0.5140        48
weighted avg     0.6814    0.6458    0.6617        48

Confusion matrix:
 [[ 3  7]
 [10 28]]
----------------------------------------------------------------------
Gradient Boosting
ROC-AUC: 0.7342
PR-AUC : 0.8993
              precision    recall  f1-score   support

           0     0.2432    0.9000    0.3830        10
           1     0.9091    0.2632    0.4082        38

    accuracy                         0.3958        48
   macro avg     0.5762    0.5816    0.3956        48
weighted avg     0.7704    0.3958    0.4029        48

Confusion matrix:
 [[ 9  1]
 [28 10]]
```

1.0.9 9. Evaluation plots

ROC and Precision-Recall curves help compare performance under class imbalance.

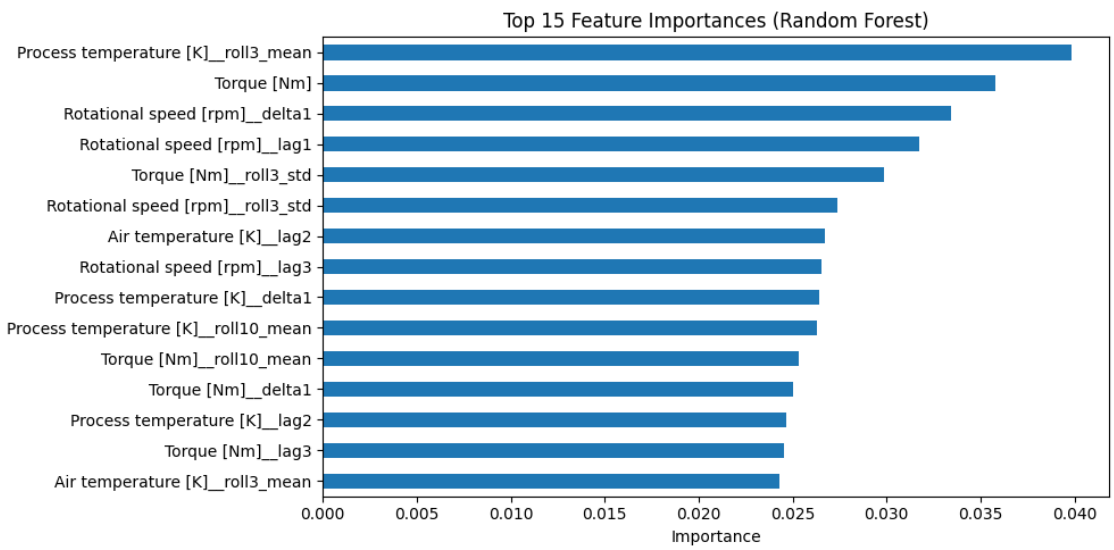
[9]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# ROC Curve (left)
RocCurveDisplay.from_predictions(
    y_test, proba_test_lr, name="LogReg", ax=axes[0]
)
RocCurveDisplay.from_predictions(
    y_test, proba_test_rf, name="RandomForest", ax=axes[0]
)
RocCurveDisplay.from_predictions(
    y_test, proba_test_gb, name="GradBoost", ax=axes[0]
)
axes[0].set_title("ROC Curve")

# Precision-Recall Curve (right)
PrecisionRecallDisplay.from_predictions(
    y_test, proba_test_lr, name="LogReg", ax=axes[1]
)
PrecisionRecallDisplay.from_predictions(
    y_test, proba_test_rf, name="RandomForest", ax=axes[1]
)
PrecisionRecallDisplay.from_predictions(
    y_test, proba_test_gb, name="GradBoost", ax=axes[1]
)
axes[1].set_title("Precision-Recall Curve")

plt.tight_layout()
plt.show()
```



1.0.10 10. Feature importance (tree-based)

Random Forest feature importance provides a quick view of what the model uses most.

[10]:
```
importances = pd.Series(rf.feature_importances_, index=X_train.columns).
    ↪sort_values(ascending=False)
top_k = 15
top = importances.head(top_k)

plt.figure(figsize=(10,5))
top[::-1].plot(kind="barh")
plt.title(f"Top {top_k} Feature Importances (Random Forest)")
plt.xlabel("Importance")
plt.tight_layout()
plt.show()

top.to_frame("importance")
```



[10]:

	importance
Process temperature [K]__roll3_mean	0.039847
Torque [Nm]	0.035810
Rotational speed [rpm]__delta1	0.033444
Rotational speed [rpm]__lag1	0.031755
Torque [Nm]__roll3_std	0.029845
Rotational speed [rpm]__roll3_std	0.027405
Air temperature [K]__lag2	0.026741
Rotational speed [rpm]__lag3	0.026511
Process temperature [K]__delta1	0.026407
Process temperature [K]__roll10_mean	0.026291
Torque [Nm]__roll10_mean	0.025317
Torque [Nm]__delta1	0.025014
Process temperature [K]__lag2	0.024666
Torque [Nm]__lag3	0.024558
Air temperature [K]__roll3_mean	0.024290

1.0.11 11. Save model-ready artifacts

This saves the engineered time-series dataset used by Model 1 for reproducibility.

[11]:
```
OUTPUT_FEATURES_PATH = "../data/ai4i_time_features_model1.csv"
df_feat.to_csv(OUTPUT_FEATURES_PATH, index=False)
print(f"Saved time-feature dataset to: {OUTPUT_FEATURES_PATH}")
```

```
Saved time-feature dataset to: ../data/ai4i_time_features_model1.csv
```

1.0.12 12. Export Classification Results for Tableau

[15]:
```
os.makedirs("../outputs", exist_ok=True)

# Collect available probability vectors from this notebook
candidates = {}
if "proba_test_lr" in globals():
    candidates["LogReg"] = np.asarray(proba_test_lr)
if "proba_test_rf" in globals():
    candidates["RandomForest"] = np.asarray(proba_test_rf)
if "proba_test_gb" in globals():
    candidates["GradBoost"] = np.asarray(proba_test_gb)

if not candidates:
    raise NameError("No probability vectors found. Expected one of:␣
↪proba_test_lr, proba_test_rf, proba_test_gb.")

# Pick best model by ROC-AUC on test set
scores = {name: roc_auc_score(y_test, proba) for name, proba in candidates.
↪items()}
best_name = max(scores, key=scores.get)
y_pred_prob = candidates[best_name]

print("ROC-AUC by model:", {k: round(v, 4) for k, v in scores.items()})
print(f"Selected for export: {best_name}")

# Recommended threshold (Youden's J)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
optimal_idx = (tpr - fpr).argmax()
recommended_threshold = float(thresholds[optimal_idx])
print(f"Recommended threshold (Youden's J): {recommended_threshold:.6f}")

# Pred label using recommended threshold
y_pred_label = (y_pred_prob >= recommended_threshold).astype(int)

# Use notebook's TIME_COL if available
time_col = TIME_COL if "TIME_COL" in globals() else "Tool wear [min]"
if time_col not in X_test.columns:
    raise KeyError(f"'{time_col}' not found in X_test columns: {list(X_test.
↪columns)}")

export_df = pd.DataFrame({
    "tool_wear": X_test[time_col].values,
    "actual_failure": np.asarray(y_test).astype(int),
    "pred_prob_failure": y_pred_prob,
    "pred_label": y_pred_label,
    "model_selected": best_name,
```

```
    "recommended_threshold": recommended_threshold
}).sort_values("tool_wear")

export_path = "../outputs/pred_failure_traditional.csv"
export_df.to_csv(export_path, index=False)

print(f"Saved: {export_path} | rows={len(export_df)}")
```

```
ROC-AUC by model: {'LogReg': 0.5947, 'RandomForest': 0.5237, 'GradBoost':
0.7342}
Selected for export: GradBoost
Recommended threshold (Youden's J): 0.016185
Saved: ../outputs/pred_failure_traditional.csv | rows=48
```

1.0.13 Why We Chose Failure Classification Instead of Torque Prediction

In an earlier draft, we considered predicting future torque values as a regression task. However, after reviewing the project objectives, we shifted to predicting **Machine failure (binary classification)** for the following reasons:

1. **Alignment with Project Goal**
   The primary objective of predictive maintenance is to anticipate equipment failure, not merely forecast sensor values. Predicting failure directly better reflects real-world maintenance decision-making.

2. **Business Relevance**
   Maintenance teams act on failure risk (fail vs. no-fail), not on small changes in torque. A classification model provides actionable outputs such as failure probability and risk thresholds.

3. **Clear Evaluation Metrics**
   Failure classification allows the use of metrics suited for rare-event detection (ROC-AUC, Precision-Recall AUC), which are more appropriate for imbalanced industrial datasets.

4. **Avoiding Indirect Modeling**
   Predicting torque and then inferring failure from torque changes adds an extra modeling step and potential error propagation. Directly modeling failure simplifies interpretation and reduces complexity.

For these reasons, we adopted a time-aware classification framework to predict machine failure using engineered temporal features derived from sensor data.

model1B_regression_torque_forecast

February 21, 2026

1 Model 1B — Regression Baseline (Torque Forecasting)

This notebook is an **optional baseline** included to contrast regression vs. classification for the project.

Goal (regression): - Predict the **next-step torque** value (`Torque_future`) using an ordered sequence derived from `Tool wear [min]`.

(Predict mean torque at tool_wear = t+1 using features at tool_wear = t)

Important note: - This regression task is *not* the primary predictive maintenance objective (failure prediction). - It is included to demonstrate why **direct failure classification** is more actionable for maintenance decisions.

1.0.1 1. Setup

[7]:
```
import os
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

import matplotlib.pyplot as plt

pd.set_option("display.max_columns", 200)
pd.set_option("display.width", 120)
```

1.0.2 2. Load data

We load the prepared dataset if available. Otherwise, we load the raw dataset and apply minimal preprocessing (drop identifiers and one-hot encode `Type`).

For regression, the failure-mode flags are not used.

[8]:
```python
DATA_PREPARED_PATH = "../data/ai4i_prepared.csv"
DATA_RAW_PATH = "../data/ai4i_2020_predictive_maintenance.csv"

TIME_COL = "Tool wear [min]"

def load_dataset():
    if os.path.exists(DATA_PREPARED_PATH):
        df = pd.read_csv(DATA_PREPARED_PATH)
        source = "prepared"
    else:
        df_raw = pd.read_csv(DATA_RAW_PATH)

        # Minimal preprocessing
        drop_cols = ["UDI", "Product ID"]
        df = df_raw.drop(columns=[c for c in drop_cols if c in df_raw.columns],␣
 ↪errors="ignore")

        if "Type" in df.columns:
            df = pd.get_dummies(df, columns=["Type"], drop_first=True)

        source = "raw+prepped"
    return df, source

df, source = load_dataset()
print(f"Loaded source: {source}")
print("Shape:", df.shape)
df.head()
```

```
Loaded source: prepared
Shape: (10000, 14)
```

[8]:
```
    UDI Product ID Type  Air temperature [K]  Process temperature [K]  
Rotational speed [rpm]  Torque [Nm]  \
0     1     M14860    M                298.1                    308.6
1551         42.8
1  7257     H36670    H                300.2                    310.3
1408         42.5
2   504     M15363    M                297.6                    309.2
1442         48.1
3  7169     L54348    L                300.3                    310.3
1704         29.5
4  7089     M21948    M                300.6                    310.3
1614         32.7

   Tool wear [min]  Machine failure  TWF  HDF  PWF  OSF  RNF
0                0                0    0    0    0    0    0
1                0                0    0    0    0    0    0
```

```
2                 0                 0    0    0    0    0    0
3                 0                 0    0    0    0    0    0
4                 0                 0    0    0    0    0    0
```

1.0.3 3. Create an ordered sequence using tool wear (time proxy)

We aggregate sensor readings by `Tool wear [min]` to form a monotonic sequence. Then we define `Torque_future` as the next step torque value (one-step-ahead forecasting).

[9]:
```python
required_cols = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]",
    TIME_COL,
]
missing = [c for c in required_cols if c not in df.columns]
if missing:
    raise ValueError(f"Missing required columns: {missing}")

# Aggregate by tool wear (population-level) to build an ordered sequence
df_tw = (
    df
    .groupby(TIME_COL, as_index=False)
    .mean(numeric_only=True)
    .sort_values(TIME_COL)
    .reset_index(drop=True)
)

# One-step-ahead target
df_tw["Torque_future"] = df_tw["Torque [Nm]"].shift(-1)
df_tw = df_tw.dropna().reset_index(drop=True)

print("Aggregated shape:", df_tw.shape)
df_tw.head()
```

```
Aggregated shape: (245, 13)
```

[9]:
```
   Tool wear [min]          UDI  Air temperature [K]  Process temperature [K]
Rotational speed [rpm]  Torque [Nm]  \
0                 0  5014.266667           299.956667               309.955833
1524.916667    40.661667
1                 2  5038.826087           300.272464               310.142029
1555.521739    39.646377
2                 3  4925.970588           299.679412               309.826471
1508.264706    41.644118
3                 4  5505.205882           299.997059               309.870588
1525.882353    41.117647
```

```
4                 5  4937.206349          299.925397                310.014286
1620.761905     36.071429

   Machine failure  TWF       HDF       PWF  OSF       RNF  Torque_future
0         0.025000  0.0  0.000000  0.025000  0.0  0.000000      39.646377
1         0.028986  0.0  0.014493  0.014493  0.0  0.014493      41.644118
2         0.029412  0.0  0.000000  0.029412  0.0  0.000000      41.117647
3         0.000000  0.0  0.000000  0.000000  0.0  0.000000      36.071429
4         0.015873  0.0  0.000000  0.015873  0.0  0.000000      38.674194
```

1.0.4 4. Train/test split (time-ordered)

We split without shuffling to preserve time order.

[10]:
```python
feature_cols = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    TIME_COL,
    "Torque [Nm]",  # current torque can help predict next torque
]

X = df_tw[feature_cols]
y = df_tw["Torque_future"]

split_idx = int(len(df_tw) * 0.8)
X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

print("Train size:", X_train.shape, " Test size:", X_test.shape)
```

```
Train size: (196, 5)  Test size: (49, 5)
```

1.0.5 5. Regression models

We compare a simple linear baseline vs. a non-linear model: - Linear Regression (with scaling) - Ridge Regression (regularized linear) - Random Forest Regressor (non-linear baseline)

[11]:
```python
models = {}

models["LinearRegression"] = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("reg", LinearRegression())
])

models["Ridge"] = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("reg", Ridge(alpha=1.0, random_state=42))
```

```
])

models["RandomForestRegressor"] = RandomForestRegressor(
    n_estimators=400,
    min_samples_split=10,
    min_samples_leaf=5,
    random_state=42,
    n_jobs=-1
)

def eval_regression(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)
    return mae, rmse, r2

results = []
preds = {}

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    preds[name] = y_pred
    mae, rmse, r2 = eval_regression(y_test, y_pred)
    results.append((name, mae, rmse, r2))

results_df = pd.DataFrame(results, columns=["model", "MAE", "RMSE", "R2"]).
↪sort_values("RMSE")
results_df
```

[11]:

	model	MAE	RMSE	R2
1	Ridge	3.145049	4.524447	-0.021235
0	LinearRegression	3.143357	4.524576	-0.021293
2	RandomForestRegressor	3.177763	4.525682	-0.021792

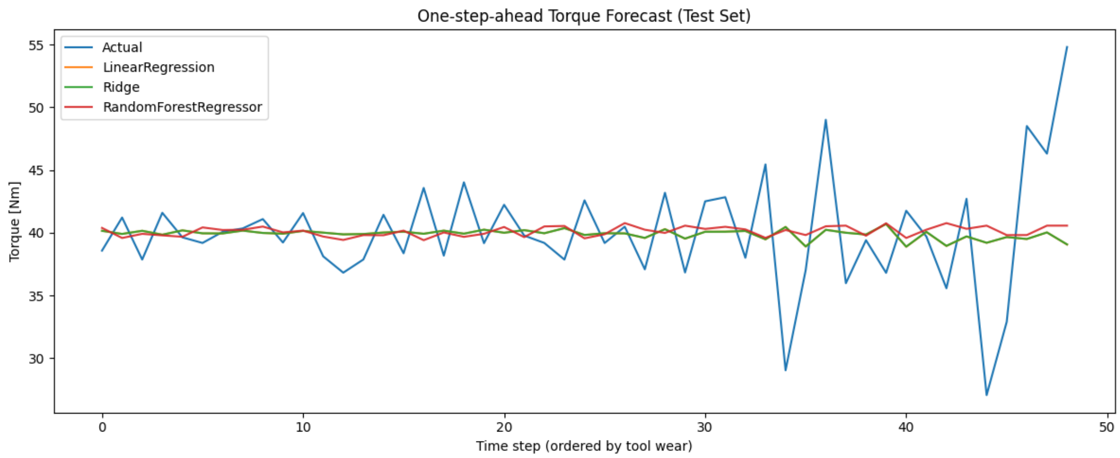
1.0.6 6. Plot: actual vs predicted torque (test set)

This visual comparison helps show how well each model tracks the next-step torque values.

[12]:

```
plt.figure(figsize=(12, 5))
plt.plot(y_test.reset_index(drop=True).values, label="Actual")
for name, y_pred in preds.items():
    plt.plot(y_pred, label=name, alpha=0.9)
plt.title("One-step-ahead Torque Forecast (Test Set)")
plt.xlabel("Time step (ordered by tool wear)")
plt.ylabel("Torque [Nm]")
```

```python
plt.legend()
plt.tight_layout()
plt.show()
```



1.0.7 7. Discussion: regression vs. failure classification

- Torque forecasting predicts a continuous sensor value, which may be useful for anomaly detection.
- Predictive maintenance decisions typically require **failure risk** (classification) rather than a raw sensor forecast.
- Even if torque is predicted accurately, mapping that forecast to a failure decision adds an extra step and can amplify errors.

1.0.8 8. Save predictions

This exports a small table with actual and predicted values for reporting.

[13]:
```python
out = pd.DataFrame({
    "Torque_future_actual": y_test.values,
    **{f"Torque_future_pred_{name}": y_pred for name, y_pred in preds.items()}
})

OUTPUT_PATH = "../data/model1B_torque_forecast_predictions.csv"
out.to_csv(OUTPUT_PATH, index=False)
print(f"Saved predictions to: {OUTPUT_PATH}")
out.head()
```

```
Saved predictions to: ../data/model1B_torque_forecast_predictions.csv
```

[13]:
```
   Torque_future_actual  Torque_future_pred_LinearRegression  
Torque_future_pred_Ridge  \
0             38.569565                            40.150744
```

```
40.149124
1             41.208511                          39.899440
39.898446
2             37.856410                          40.151312
40.150230
3             41.584444                          39.837399
39.837839
4             39.628889                          40.185983
40.183500

   Torque_future_pred_RandomForestRegressor
0                                 40.381197
1                                 39.578778
2                                 39.907429
3                                 39.786314
4                                 39.674121
```

1.0.9 9. Export Regression Forecast Results for Tableau

[16]:
```python
os.makedirs("../outputs", exist_ok=True)

# Validate preds exists
if "preds" not in globals() or not isinstance(preds, dict) or len(preds) == 0:
    raise NameError("preds dict not found (or empty). Run the model training/
↪evaluation cell first.")

# Pick best model (lowest RMSE)
if "results_df" in globals() and hasattr(results_df, "iloc") and "model" in␣
↪results_df.columns:
    best_name = results_df.iloc[0]["model"]
else:
    rmse_scores = {name: float(np.sqrt(mean_squared_error(y_test, pred))) for␣
↪name, pred in preds.items()}
    best_name = min(rmse_scores, key=rmse_scores.get)
    print("RMSE by model:", {k: round(v, 4) for k, v in rmse_scores.items()})

if best_name not in preds:
    raise KeyError(f"Best model '{best_name}' not found in preds keys:␣
↪{list(preds.keys())}")

y_pred = np.asarray(preds[best_name]).ravel()

print(f"Selected for export: {best_name}")

# Use the notebook's time column
time_col = TIME_COL if "TIME_COL" in globals() else "Tool wear [min]"
if time_col not in X_test.columns:
```

```
    raise KeyError(f"'{time_col}' not found in X_test columns: {list(X_test.
 ↪columns)}")

# Build export dataframe
export_df = pd.DataFrame({
    "tool_wear": X_test[time_col].values,
    "torque_actual_next": np.asarray(y_test).ravel(),
    "torque_pred": y_pred,
    "model_selected": best_name
}).sort_values("tool_wear")

export_df["residual"] = export_df["torque_actual_next"] -␣
 ↪export_df["torque_pred"]

# Save
export_path = "../outputs/pred_torque_forecast.csv"
export_df.to_csv(export_path, index=False)

print(f"Saved: {export_path} | rows={len(export_df)}")
```

```
Selected for export: Ridge
Saved: ../outputs/pred_torque_forecast.csv | rows=49
```

model2_deep_learning

February 21, 2026

1 Model 2 — Deep Learning (Sequence Model using Tool-Wear Time Proxy)

This notebook implements **Model 2** for the project using deep learning.

Project framing: - Dataset: AI4I 2020 Predictive Maintenance - Primary target: `Machine failure` (binary classification) - Time proxy: `Tool wear [min]` is treated as progression over time-under-use.

Key design choices: 1. We aggregate records by tool wear to create an ordered sequence (population-level). 2. We preserve the binary label during aggregation using `max()` for `Machine failure`. 3. We build sliding-window sequences and train a sequence model (LSTM).

Deliverables in this notebook: - Data preparation for deep learning (sequence windows) - Baseline deep model (MLP) and sequence model (LSTM) - Time-aware train/validation/test split (no shuffle) - Evaluation with ROC-AUC and PR-AUC (useful for class imbalance)

1.0.1 1. Setup

[1]:
```
import os
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    roc_auc_score,
    roc_curve,
    average_precision_score,
    RocCurveDisplay,
    PrecisionRecallDisplay
)
```

```
pd.set_option("display.max_columns", 200)
pd.set_option("display.width", 120)

# Deep learning (TensorFlow / Keras)
try:
    import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras import layers
    print("TensorFlow:", tf.__version__)
except ModuleNotFoundError as e:
    raise ModuleNotFoundError(
        "TensorFlow is not installed. Add `tensorflow>=2.12` to requirements.
 ↪txt and reinstall."
    ) from e
```

```
TensorFlow: 2.16.2
```

1.0.2 2. Load data

We load the prepared dataset if available. Otherwise, we load raw CSV and apply minimal preprocessing: - Drop identifiers (`UDI`, `Product ID`) - Drop failure-mode flags (`TWF`, `HDF`, `PWF`, `OSF`, `RNF`) to avoid leakage - One-hot encode `Type`

[2]:
```
DATA_PREPARED_PATH = "../data/ai4i_prepared.csv"
DATA_RAW_PATH = "../data/ai4i_2020_predictive_maintenance.csv"

TARGET_COL = "Machine failure"
TIME_COL = "Tool wear [min]"

def load_dataset():
    if os.path.exists(DATA_PREPARED_PATH):
        df = pd.read_csv(DATA_PREPARED_PATH)
        source = "prepared"
    else:
        df_raw = pd.read_csv(DATA_RAW_PATH)
        drop_cols = ["UDI", "Product ID", "TWF", "HDF", "PWF", "OSF", "RNF"]
        df = df_raw.drop(columns=[c for c in drop_cols if c in df_raw.columns],␣
 ↪errors="ignore")
        if "Type" in df.columns:
            df = pd.get_dummies(df, columns=["Type"], drop_first=True)
        source = "raw+prepped"
    return df, source

df, source = load_dataset()
print(f"Loaded source: {source}")
print("Shape:", df.shape)
df.head()
```

```
Loaded source: prepared
Shape: (10000, 14)
```

[2]:

```
    UDI Product ID Type  Air temperature [K]  Process temperature [K]
 Rotational speed [rpm]  Torque [Nm]  \
 0     1     M14860    M                298.1                    308.6
 1551          42.8
 1  7257     H36670    H                300.2                    310.3
 1408          42.5
 2   504     M15363    M                297.6                    309.2
 1442          48.1
 3  7169     L54348    L                300.3                    310.3
 1704          29.5
 4  7089     M21948    M                300.6                    310.3
 1614          32.7

   Tool wear [min]  Machine failure  TWF  HDF  PWF  OSF  RNF
 0               0                0    0    0    0    0    0
 1               0                0    0    0    0    0    0
 2               0                0    0    0    0    0    0
 3               0                0    0    0    0    0    0
 4               0                0    0    0    0    0    0
```

1.0.3 3. Aggregate by tool wear to form an ordered sequence

Deep learning sequence models need ordered samples. We aggregate by tool wear.

Important: `Machine failure` is aggregated with `max()` to preserve binary semantics.

[3]:

```
required_cols = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]",
    TIME_COL,
    TARGET_COL
]
missing = [c for c in required_cols if c not in df.columns]
if missing:
    raise ValueError(f"Missing required columns: {missing}")

df_tw = (
    df
    .groupby(TIME_COL, as_index=False)
    .agg({
        "Air temperature [K]": "mean",
        "Process temperature [K]": "mean",
        "Rotational speed [rpm]": "mean",
```

```
        "Torque [Nm]": "mean",
        TARGET_COL: "max",
    })
    .sort_values(TIME_COL)
    .reset_index(drop=True)
)

print("Aggregated shape:", df_tw.shape)
print("Label distribution:", df_tw[TARGET_COL].value_counts().to_dict())
df_tw.head()
```

```
Aggregated shape: (246, 6)
Label distribution: {1: 172, 0: 74}
```

[3]:

```
   Tool wear [min]  Air temperature [K]  Process temperature [K]  Rotational
speed [rpm]  Torque [Nm]  Machine failure
0                0           299.956667                310.955833
1524.916667    40.661667                1
1                2           300.272464                310.142029
1555.521739    39.646377                1
2                3           299.679412                309.826471
1508.264706    41.644118                1
3                4           299.997059                309.870588
1525.882353    41.117647                0
4                5           299.925397                310.014286
1620.761905    36.071429                1
```

1.0.4 4. Build sliding-window sequences

We create sequences of length `SEQ_LEN` from the ordered tool-wear series. Each sequence uses sensor readings from the previous `SEQ_LEN` steps. The label for a sequence is the `Machine failure` value at the final step of the window.

[4]:

```
# Features used for the sequence model
feature_cols = [
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]",
]

SEQ_LEN = 10  # window length (can be tuned)

X_values = df_tw[feature_cols].values.astype(np.float32)
y_values = df_tw[TARGET_COL].values.astype(int)

def make_sequences(X, y, seq_len: int):
    X_seq, y_seq = [], []
```

```
    for i in range(seq_len - 1, len(X)):
        X_seq.append(X[i - seq_len + 1 : i + 1])
        y_seq.append(y[i])  # label at window end
    return np.array(X_seq, dtype=np.float32), np.array(y_seq, dtype=int)

X_seq, y_seq = make_sequences(X_values, y_values, SEQ_LEN)

print("X_seq shape:", X_seq.shape, "(samples, timesteps, features)")
print("y_seq distribution:", dict(pd.Series(y_seq).value_counts()))
```

```
X_seq shape: (237, 10, 4) (samples, timesteps, features)
y_seq distribution: {1: 165, 0: 72}
```

1.0.5 5. Time-aware train/validation/test split

We split sequences by time order (no shuffle). We also ensure that the training set contains both classes. Because failures can be rare, we select the earliest split points that keep positives in all sets.

[5]:
```
def find_splits_with_positives(y, min_pos_train=1, min_pos_val=1,␣
 ↪min_pos_test=1, min_train=50, min_val=20):
    n = len(y)
    y = pd.Series(y).reset_index(drop=True)

    total_pos = int(y.sum())
    if total_pos < (min_pos_train + min_pos_val + min_pos_test):
        raise ValueError(f"Not enough positive samples overall (found␣
 ↪{total_pos}).")

    # Search for train_end and val_end (time-ordered)
    for train_end in range(min_train, n - (min_val + 1)):
        pos_train = int(y.iloc[:train_end].sum())
        if pos_train < min_pos_train:
            continue
        for val_end in range(train_end + min_val, n - 1):
            pos_val = int(y.iloc[train_end:val_end].sum())
            pos_test = int(y.iloc[val_end:].sum())
            if pos_val >= min_pos_val and pos_test >= min_pos_test:
                return train_end, val_end

    raise ValueError("Could not find time-ordered splits with positives in␣
 ↪train/val/test.")

train_end, val_end = find_splits_with_positives(y_seq)

X_train, y_train = X_seq[:train_end], y_seq[:train_end]
X_val, y_val     = X_seq[train_end:val_end], y_seq[train_end:val_end]
X_test, y_test   = X_seq[val_end:], y_seq[val_end:]
```

```
print("Split indices:", train_end, val_end)
print("Train:", X_train.shape, "labels:", dict(pd.Series(y_train).
 ↪value_counts()))
print("Val  :", X_val.shape,   "labels:", dict(pd.Series(y_val).value_counts()))
print("Test :", X_test.shape,  "labels:", dict(pd.Series(y_test).
 ↪value_counts()))
```

```
Split indices: 50 70
Train: (50, 10, 4) labels: {1: 32, 0: 18}
Val  : (20, 10, 4) labels: {1: 16, 0: 4}
Test : (167, 10, 4) labels: {1: 117, 0: 50}
```

1.0.6 6. Feature scaling (fit on train only)

We standardize features using statistics from the training set only, then apply to validation and test. For sequences, we fit the scaler on the flattened training data and reshape back.

[6]:
```
scaler = StandardScaler()

# Fit scaler on training data (flatten timesteps)
X_train_flat = X_train.reshape(-1, X_train.shape[-1])
scaler.fit(X_train_flat)

def scale_sequences(X, scaler_obj):
    X_flat = X.reshape(-1, X.shape[-1])
    X_scaled = scaler_obj.transform(X_flat)
    return X_scaled.reshape(X.shape)

X_train_s = scale_sequences(X_train, scaler)
X_val_s   = scale_sequences(X_val, scaler)
X_test_s  = scale_sequences(X_test, scaler)

print("Scaled shapes:", X_train_s.shape, X_val_s.shape, X_test_s.shape)
```

```
Scaled shapes: (50, 10, 4) (20, 10, 4) (167, 10, 4)
```

1.0.7 7. Handle class imbalance (class weights)

We compute class weights from the training labels and pass them to Keras during training.

[7]:
```
classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight="balanced", classes=classes,␣
 ↪y=y_train)
class_weight_dict = {int(c): float(w) for c, w in zip(classes, class_weights)}

print("Class weights:", class_weight_dict)
```

```
Class weights: {0: 1.3888888888888888, 1: 0.78125}
```

1.0.8 8. Model A (baseline deep model): MLP on last timestep

This baseline uses only the last timestep features from each sequence and trains a small feedforward network. It provides a deep-learning baseline that is comparable to traditional ML on tabular features.

[8]:
```python
# Use last timestep only (tabular baseline)
X_train_last = X_train_s[:, -1, :]
X_val_last   = X_val_s[:, -1, :]
X_test_last  = X_test_s[:, -1, :]

def build_mlp(input_dim: int):
    model = keras.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(32, activation="relu"),
        layers.Dropout(0.2),
        layers.Dense(16, activation="relu"),
        layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid"),
    ])
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        loss="binary_crossentropy",
        metrics=[keras.metrics.AUC(name="roc_auc"), keras.metrics.
 ↪AUC(curve="PR", name="pr_auc")]
    )
    return model

mlp = build_mlp(X_train_last.shape[1])
mlp.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 32)	160
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 1)	17

Total params: 705 (2.75 KB)

Trainable params: 705 (2.75 KB)

Non-trainable params: 0 (0.00 B)

[9]:
```
callbacks = [
    keras.callbacks.EarlyStopping(monitor="val_pr_auc", mode="max",␣
 ↪patience=10, restore_best_weights=True),
    keras.callbacks.ReduceLROnPlateau(monitor="val_pr_auc", mode="max",␣
 ↪factor=0.5, patience=5, min_lr=1e-5),
]

history_mlp = mlp.fit(
    X_train_last, y_train,
    validation_data=(X_val_last, y_val),
    epochs=80,
    batch_size=32,
    class_weight=class_weight_dict,
    callbacks=callbacks,
    verbose=1
)
```

```
Epoch 1/80
2/2           1s 153ms/step - loss:
0.7998 - pr_auc: 0.6314 - roc_auc: 0.4722 - val_loss: 1.0554 - val_pr_auc:
0.6516 - val_roc_auc: 0.1562 - learning_rate: 0.0010
Epoch 2/80
2/2           0s 24ms/step - loss:
0.7944 - pr_auc: 0.7019 - roc_auc: 0.4983 - val_loss: 1.0309 - val_pr_auc:
0.6609 - val_roc_auc: 0.1797 - learning_rate: 0.0010
Epoch 3/80
2/2           0s 25ms/step - loss:
0.7689 - pr_auc: 0.6679 - roc_auc: 0.5208 - val_loss: 1.0069 - val_pr_auc:
0.6634 - val_roc_auc: 0.1875 - learning_rate: 0.0010
Epoch 4/80
2/2           0s 26ms/step - loss:
0.7731 - pr_auc: 0.6123 - roc_auc: 0.4271 - val_loss: 0.9849 - val_pr_auc:
0.6664 - val_roc_auc: 0.1953 - learning_rate: 0.0010
Epoch 5/80
2/2           0s 25ms/step - loss:
0.7410 - pr_auc: 0.7070 - roc_auc: 0.5382 - val_loss: 0.9656 - val_pr_auc:
0.6664 - val_roc_auc: 0.1953 - learning_rate: 0.0010
Epoch 6/80
2/2           0s 26ms/step - loss:
0.7321 - pr_auc: 0.7344 - roc_auc: 0.5616 - val_loss: 0.9480 - val_pr_auc:
```

```
0.6619 - val_roc_auc: 0.2031 - learning_rate: 0.0010
Epoch 7/80
2/2            0s 24ms/step - loss:
0.7391 - pr_auc: 0.6068 - roc_auc: 0.5009 - val_loss: 0.9312 - val_pr_auc:
0.6619 - val_roc_auc: 0.2031 - learning_rate: 0.0010
Epoch 8/80
2/2            0s 26ms/step - loss:
0.7413 - pr_auc: 0.7443 - roc_auc: 0.5321 - val_loss: 0.9168 - val_pr_auc:
0.6712 - val_roc_auc: 0.2344 - learning_rate: 0.0010
Epoch 9/80
2/2            0s 26ms/step - loss:
0.7532 - pr_auc: 0.6878 - roc_auc: 0.4991 - val_loss: 0.9041 - val_pr_auc:
0.6735 - val_roc_auc: 0.2500 - learning_rate: 0.0010
Epoch 10/80
2/2            0s 25ms/step - loss:
0.7399 - pr_auc: 0.6602 - roc_auc: 0.5052 - val_loss: 0.8914 - val_pr_auc:
0.6808 - val_roc_auc: 0.2656 - learning_rate: 0.0010
Epoch 11/80
2/2            0s 36ms/step - loss:
0.7341 - pr_auc: 0.7087 - roc_auc: 0.4991 - val_loss: 0.8798 - val_pr_auc:
0.6761 - val_roc_auc: 0.2578 - learning_rate: 0.0010
Epoch 12/80
2/2            0s 39ms/step - loss:
0.7170 - pr_auc: 0.6671 - roc_auc: 0.5590 - val_loss: 0.8691 - val_pr_auc:
0.6761 - val_roc_auc: 0.2578 - learning_rate: 0.0010
Epoch 13/80
2/2            0s 25ms/step - loss:
0.6966 - pr_auc: 0.7723 - roc_auc: 0.6259 - val_loss: 0.8595 - val_pr_auc:
0.6761 - val_roc_auc: 0.2578 - learning_rate: 0.0010
Epoch 14/80
2/2            0s 24ms/step - loss:
0.7156 - pr_auc: 0.7130 - roc_auc: 0.5712 - val_loss: 0.8502 - val_pr_auc:
0.6761 - val_roc_auc: 0.2578 - learning_rate: 0.0010
Epoch 15/80
2/2            0s 24ms/step - loss:
0.7537 - pr_auc: 0.5546 - roc_auc: 0.3559 - val_loss: 0.8406 - val_pr_auc:
0.6808 - val_roc_auc: 0.2656 - learning_rate: 0.0010
Epoch 16/80
2/2            0s 24ms/step - loss:
0.7127 - pr_auc: 0.7145 - roc_auc: 0.5460 - val_loss: 0.8361 - val_pr_auc:
0.6752 - val_roc_auc: 0.2578 - learning_rate: 5.0000e-04
Epoch 17/80
2/2            0s 24ms/step - loss:
0.7234 - pr_auc: 0.5911 - roc_auc: 0.4618 - val_loss: 0.8318 - val_pr_auc:
0.6832 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 18/80
2/2            0s 24ms/step - loss:
0.7101 - pr_auc: 0.7010 - roc_auc: 0.5243 - val_loss: 0.8276 - val_pr_auc:
```

```
0.6867 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 19/80
2/2            0s 23ms/step - loss:
0.7148 - pr_auc: 0.6038 - roc_auc: 0.4800 - val_loss: 0.8238 - val_pr_auc:
0.6891 - val_roc_auc: 0.2812 - learning_rate: 5.0000e-04
Epoch 20/80
2/2            0s 25ms/step - loss:
0.7209 - pr_auc: 0.7052 - roc_auc: 0.5252 - val_loss: 0.8198 - val_pr_auc:
0.6832 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 21/80
2/2            0s 24ms/step - loss:
0.7123 - pr_auc: 0.7049 - roc_auc: 0.5252 - val_loss: 0.8161 - val_pr_auc:
0.6826 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 22/80
2/2            0s 27ms/step - loss:
0.7319 - pr_auc: 0.5634 - roc_auc: 0.4236 - val_loss: 0.8126 - val_pr_auc:
0.6891 - val_roc_auc: 0.2812 - learning_rate: 5.0000e-04
Epoch 23/80
2/2            0s 25ms/step - loss:
0.7113 - pr_auc: 0.6909 - roc_auc: 0.5122 - val_loss: 0.8093 - val_pr_auc:
0.6891 - val_roc_auc: 0.2812 - learning_rate: 5.0000e-04
Epoch 24/80
2/2            0s 25ms/step - loss:
0.7137 - pr_auc: 0.6952 - roc_auc: 0.5417 - val_loss: 0.8059 - val_pr_auc:
0.6906 - val_roc_auc: 0.2812 - learning_rate: 5.0000e-04
Epoch 25/80
2/2            0s 24ms/step - loss:
0.7061 - pr_auc: 0.6027 - roc_auc: 0.5009 - val_loss: 0.8027 - val_pr_auc:
0.6826 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 26/80
2/2            0s 24ms/step - loss:
0.6742 - pr_auc: 0.7485 - roc_auc: 0.6432 - val_loss: 0.7997 - val_pr_auc:
0.6826 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 27/80
2/2            0s 23ms/step - loss:
0.6750 - pr_auc: 0.7267 - roc_auc: 0.6615 - val_loss: 0.7969 - val_pr_auc:
0.6891 - val_roc_auc: 0.2891 - learning_rate: 5.0000e-04
Epoch 28/80
2/2            0s 24ms/step - loss:
0.7107 - pr_auc: 0.6950 - roc_auc: 0.5538 - val_loss: 0.7944 - val_pr_auc:
0.6891 - val_roc_auc: 0.2812 - learning_rate: 5.0000e-04
Epoch 29/80
2/2            0s 24ms/step - loss:
0.7280 - pr_auc: 0.5599 - roc_auc: 0.4089 - val_loss: 0.7918 - val_pr_auc:
0.6873 - val_roc_auc: 0.2734 - learning_rate: 5.0000e-04
Epoch 30/80
2/2            0s 26ms/step - loss:
0.6869 - pr_auc: 0.7796 - roc_auc: 0.6458 - val_loss: 0.7906 - val_pr_auc:
```

```
0.6953 - val_roc_auc: 0.2891 - learning_rate: 2.5000e-04
Epoch 31/80
2/2             0s 28ms/step - loss:
0.6950 - pr_auc: 0.6698 - roc_auc: 0.5868 - val_loss: 0.7896 - val_pr_auc:
0.6995 - val_roc_auc: 0.2969 - learning_rate: 2.5000e-04
Epoch 32/80
2/2             0s 44ms/step - loss:
0.6944 - pr_auc: 0.7330 - roc_auc: 0.5833 - val_loss: 0.7886 - val_pr_auc:
0.7019 - val_roc_auc: 0.3047 - learning_rate: 2.5000e-04
Epoch 33/80
2/2             0s 25ms/step - loss:
0.6839 - pr_auc: 0.6841 - roc_auc: 0.6102 - val_loss: 0.7873 - val_pr_auc:
0.7019 - val_roc_auc: 0.3047 - learning_rate: 2.5000e-04
Epoch 34/80
2/2             0s 24ms/step - loss:
0.7029 - pr_auc: 0.6720 - roc_auc: 0.5755 - val_loss: 0.7860 - val_pr_auc:
0.7001 - val_roc_auc: 0.2969 - learning_rate: 2.5000e-04
Epoch 35/80
2/2             0s 24ms/step - loss:
0.6913 - pr_auc: 0.7798 - roc_auc: 0.6189 - val_loss: 0.7848 - val_pr_auc:
0.7122 - val_roc_auc: 0.3047 - learning_rate: 2.5000e-04
Epoch 36/80
2/2             0s 24ms/step - loss:
0.6817 - pr_auc: 0.7860 - roc_auc: 0.6198 - val_loss: 0.7837 - val_pr_auc:
0.6808 - val_roc_auc: 0.2656 - learning_rate: 2.5000e-04
Epoch 37/80
2/2             0s 24ms/step - loss:
0.6762 - pr_auc: 0.8324 - roc_auc: 0.6901 - val_loss: 0.7824 - val_pr_auc:
0.6826 - val_roc_auc: 0.2734 - learning_rate: 2.5000e-04
Epoch 38/80
2/2             0s 23ms/step - loss:
0.7133 - pr_auc: 0.5926 - roc_auc: 0.4705 - val_loss: 0.7813 - val_pr_auc:
0.6854 - val_roc_auc: 0.2812 - learning_rate: 2.5000e-04
Epoch 39/80
2/2             0s 24ms/step - loss:
0.7141 - pr_auc: 0.6674 - roc_auc: 0.5200 - val_loss: 0.7802 - val_pr_auc:
0.6912 - val_roc_auc: 0.2891 - learning_rate: 2.5000e-04
Epoch 40/80
2/2             0s 25ms/step - loss:
0.6956 - pr_auc: 0.7603 - roc_auc: 0.6085 - val_loss: 0.7790 - val_pr_auc:
0.6937 - val_roc_auc: 0.2969 - learning_rate: 2.5000e-04
Epoch 41/80
2/2             0s 26ms/step - loss:
0.6917 - pr_auc: 0.6787 - roc_auc: 0.5842 - val_loss: 0.7783 - val_pr_auc:
0.6980 - val_roc_auc: 0.3047 - learning_rate: 1.2500e-04
Epoch 42/80
2/2             0s 24ms/step - loss:
0.6899 - pr_auc: 0.7789 - roc_auc: 0.6311 - val_loss: 0.7777 - val_pr_auc:
```

```
0.6980 - val_roc_auc: 0.3047 - learning_rate: 1.2500e-04
Epoch 43/80
2/2             0s 24ms/step - loss:
0.6884 - pr_auc: 0.7756 - roc_auc: 0.6380 - val_loss: 0.7771 - val_pr_auc:
0.7192 - val_roc_auc: 0.3281 - learning_rate: 1.2500e-04
Epoch 44/80
2/2             0s 24ms/step - loss:
0.6862 - pr_auc: 0.7753 - roc_auc: 0.6128 - val_loss: 0.7765 - val_pr_auc:
0.7192 - val_roc_auc: 0.3281 - learning_rate: 1.2500e-04
Epoch 45/80
2/2             0s 25ms/step - loss:
0.6870 - pr_auc: 0.7861 - roc_auc: 0.6128 - val_loss: 0.7758 - val_pr_auc:
0.7208 - val_roc_auc: 0.3359 - learning_rate: 1.2500e-04
Epoch 46/80
2/2             0s 23ms/step - loss:
0.7054 - pr_auc: 0.6354 - roc_auc: 0.4922 - val_loss: 0.7751 - val_pr_auc:
0.7156 - val_roc_auc: 0.3203 - learning_rate: 1.2500e-04
Epoch 47/80
2/2             0s 23ms/step - loss:
0.7021 - pr_auc: 0.7613 - roc_auc: 0.5729 - val_loss: 0.7745 - val_pr_auc:
0.7156 - val_roc_auc: 0.3203 - learning_rate: 1.2500e-04
Epoch 48/80
2/2             0s 24ms/step - loss:
0.6902 - pr_auc: 0.7521 - roc_auc: 0.6076 - val_loss: 0.7739 - val_pr_auc:
0.7156 - val_roc_auc: 0.3203 - learning_rate: 1.2500e-04
Epoch 49/80
2/2             0s 23ms/step - loss:
0.6653 - pr_auc: 0.7645 - roc_auc: 0.6693 - val_loss: 0.7733 - val_pr_auc:
0.7156 - val_roc_auc: 0.3203 - learning_rate: 1.2500e-04
Epoch 50/80
2/2             0s 24ms/step - loss:
0.6776 - pr_auc: 0.7635 - roc_auc: 0.6224 - val_loss: 0.7727 - val_pr_auc:
0.7172 - val_roc_auc: 0.3281 - learning_rate: 1.2500e-04
Epoch 51/80
2/2             0s 27ms/step - loss:
0.6874 - pr_auc: 0.6425 - roc_auc: 0.5521 - val_loss: 0.7725 - val_pr_auc:
0.7172 - val_roc_auc: 0.3281 - learning_rate: 6.2500e-05
Epoch 52/80
2/2             0s 27ms/step - loss:
0.6907 - pr_auc: 0.7084 - roc_auc: 0.5503 - val_loss: 0.7722 - val_pr_auc:
0.7172 - val_roc_auc: 0.3281 - learning_rate: 6.2500e-05
Epoch 53/80
2/2             0s 27ms/step - loss:
0.6591 - pr_auc: 0.8175 - roc_auc: 0.7196 - val_loss: 0.7720 - val_pr_auc:
0.7200 - val_roc_auc: 0.3359 - learning_rate: 6.2500e-05
Epoch 54/80
2/2             0s 31ms/step - loss:
0.6719 - pr_auc: 0.8261 - roc_auc: 0.6875 - val_loss: 0.7718 - val_pr_auc:
```

```
0.7200 - val_roc_auc: 0.3359 - learning_rate: 6.2500e-05
Epoch 55/80
2/2              0s 24ms/step - loss:
0.6867 - pr_auc: 0.7400 - roc_auc: 0.6250 - val_loss: 0.7715 - val_pr_auc:
0.7200 - val_roc_auc: 0.3359 - learning_rate: 6.2500e-05
```

1.0.9 9. Model B (sequence model): LSTM

We train an LSTM over the full sequence window. This can capture temporal patterns across tool-wear steps.

[10]:
```
def build_lstm(timesteps: int, n_features: int):
    model = keras.Sequential([
        layers.Input(shape=(timesteps, n_features)),
        layers.LSTM(32, return_sequences=True),
        layers.Dropout(0.2),
        layers.LSTM(16),
        layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid"),
    ])
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        loss="binary_crossentropy",
        metrics=[keras.metrics.AUC(name="roc_auc"), keras.metrics.
 ↪AUC(curve="PR", name="pr_auc")]
    )
    return model

lstm = build_lstm(X_train_s.shape[1], X_train_s.shape[2])
lstm.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 10, 32)	4,736
dropout_2 (Dropout)	(None, 10, 32)	0
lstm_1 (LSTM)	(None, 16)	3,136
dropout_3 (Dropout)	(None, 16)	0
dense_3 (Dense)	(None, 1)	17

Total params: 7,889 (30.82 KB)

Trainable params: 7,889 (30.82 KB)

Non-trainable params: 0 (0.00 B)

[11]:
```
history_lstm = lstm.fit(
    X_train_s, y_train,
    validation_data=(X_val_s, y_val),
    epochs=80,
    batch_size=32,
    class_weight=class_weight_dict,
    callbacks=callbacks,
    verbose=1
)
```

```
Epoch 1/80
2/2              1s 234ms/step - loss:
0.7000 - pr_auc: 0.5675 - roc_auc: 0.4401 - val_loss: 0.6682 - val_pr_auc:
0.8459 - val_roc_auc: 0.5078 - learning_rate: 0.0010
Epoch 2/80
2/2              0s 29ms/step - loss:
0.6992 - pr_auc: 0.5670 - roc_auc: 0.3819 - val_loss: 0.6753 - val_pr_auc:
0.9064 - val_roc_auc: 0.6875 - learning_rate: 0.0010
Epoch 3/80
2/2              0s 28ms/step - loss:
0.6988 - pr_auc: 0.5681 - roc_auc: 0.4427 - val_loss: 0.6827 - val_pr_auc:
0.8930 - val_roc_auc: 0.6484 - learning_rate: 0.0010
Epoch 4/80
2/2              0s 30ms/step - loss:
0.6972 - pr_auc: 0.5666 - roc_auc: 0.3759 - val_loss: 0.6890 - val_pr_auc:
0.9139 - val_roc_auc: 0.7344 - learning_rate: 0.0010
Epoch 5/80
2/2              0s 27ms/step - loss:
0.6915 - pr_auc: 0.6633 - roc_auc: 0.5165 - val_loss: 0.6925 - val_pr_auc:
0.9050 - val_roc_auc: 0.6953 - learning_rate: 0.0010
Epoch 6/80
2/2              0s 28ms/step - loss:
0.6889 - pr_auc: 0.6899 - roc_auc: 0.5408 - val_loss: 0.6982 - val_pr_auc:
0.9044 - val_roc_auc: 0.7031 - learning_rate: 0.0010
Epoch 7/80
2/2              0s 28ms/step - loss:
0.6890 - pr_auc: 0.7224 - roc_auc: 0.6155 - val_loss: 0.7040 - val_pr_auc:
0.8976 - val_roc_auc: 0.6797 - learning_rate: 0.0010
Epoch 8/80
2/2              0s 27ms/step - loss:
```

```
0.6904 - pr_auc: 0.7430 - roc_auc: 0.5885 - val_loss: 0.7094 - val_pr_auc:
0.9001 - val_roc_auc: 0.6875 - learning_rate: 0.0010
Epoch 9/80
2/2           0s 29ms/step - loss:
0.6875 - pr_auc: 0.7033 - roc_auc: 0.6059 - val_loss: 0.7163 - val_pr_auc:
0.8924 - val_roc_auc: 0.6562 - learning_rate: 0.0010
Epoch 10/80
2/2           0s 28ms/step - loss:
0.6896 - pr_auc: 0.7042 - roc_auc: 0.5747 - val_loss: 0.7199 - val_pr_auc:
0.9053 - val_roc_auc: 0.6797 - learning_rate: 5.0000e-04
Epoch 11/80
2/2           0s 28ms/step - loss:
0.6908 - pr_auc: 0.6559 - roc_auc: 0.5964 - val_loss: 0.7231 - val_pr_auc:
0.9002 - val_roc_auc: 0.6797 - learning_rate: 5.0000e-04
Epoch 12/80
2/2           0s 27ms/step - loss:
0.6901 - pr_auc: 0.6809 - roc_auc: 0.5677 - val_loss: 0.7265 - val_pr_auc:
0.9029 - val_roc_auc: 0.6719 - learning_rate: 5.0000e-04
Epoch 13/80
2/2           0s 26ms/step - loss:
0.6854 - pr_auc: 0.6828 - roc_auc: 0.6345 - val_loss: 0.7302 - val_pr_auc:
0.8983 - val_roc_auc: 0.6641 - learning_rate: 5.0000e-04
Epoch 14/80
2/2           0s 26ms/step - loss:
0.6866 - pr_auc: 0.7525 - roc_auc: 0.6372 - val_loss: 0.7332 - val_pr_auc:
0.8911 - val_roc_auc: 0.6562 - learning_rate: 5.0000e-04
```

1.0.10 10. Evaluate models

We evaluate both models on the held-out test set using ROC-AUC and PR-AUC, plus classification report and confusion matrix.

[12]:
```python
def evaluate_classifier(name, y_true, y_prob, threshold=0.5):
    y_pred = (y_prob >= threshold).astype(int)
    print(name)
    print("ROC-AUC:", round(roc_auc_score(y_true, y_prob), 4))
    print("PR-AUC :", round(average_precision_score(y_true, y_prob), 4))
    print(classification_report(y_true, y_pred, digits=4))
    print("Confusion matrix:\n", confusion_matrix(y_true, y_pred))
    print("-"*80)

proba_mlp = mlp.predict(X_test_last).ravel()
proba_lstm = lstm.predict(X_test_s).ravel()

evaluate_classifier("MLP (last timestep)", y_test, proba_mlp)
evaluate_classifier("LSTM (sequence)", y_test, proba_lstm)
```

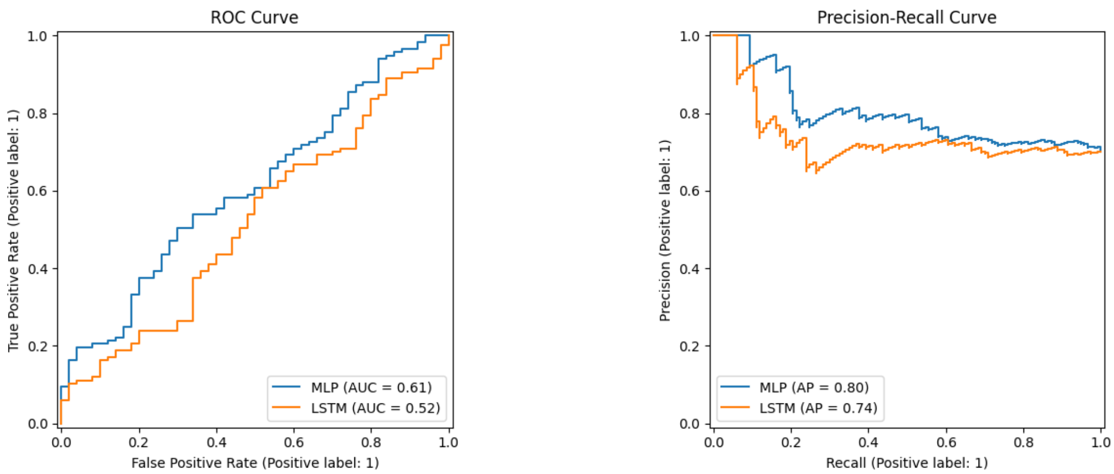
```
6/6              0s 3ms/step
6/6              0s 20ms/step
MLP (last timestep)
ROC-AUC: 0.6075
PR-AUC : 0.7977
              precision    recall  f1-score   support

           0     0.3267    0.9800    0.4900        50
           1     0.9412    0.1368    0.2388       117

    accuracy                         0.3892       167
   macro avg     0.6339    0.5584    0.3644       167
weighted avg     0.7572    0.3892    0.3140       167

Confusion matrix:
 [[ 49   1]
 [101  16]]
--------------------------------------------------------------------------------
LSTM (sequence)
ROC-AUC: 0.5173
PR-AUC : 0.7387
              precision    recall  f1-score   support

           0     0.3056    0.2200    0.2558        50
           1     0.7023    0.7863    0.7419       117

    accuracy                         0.6168       167
   macro avg     0.5039    0.5032    0.4989       167
weighted avg     0.5835    0.6168    0.5964       167

Confusion matrix:
 [[11 39]
 [25 92]]
--------------------------------------------------------------------------------
```

1.0.11 11. ROC and Precision-Recall curves (2 columns)

We plot ROC (left) and Precision-Recall (right) for both deep models.

[13]:
```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# ROC (left)
RocCurveDisplay.from_predictions(y_test, proba_mlp, name="MLP", ax=axes[0])
RocCurveDisplay.from_predictions(y_test, proba_lstm, name="LSTM", ax=axes[0])
axes[0].set_title("ROC Curve")

# PR (right)
```

```
PrecisionRecallDisplay.from_predictions(y_test, proba_mlp, name="MLP",␣
  ↪ax=axes[1])
PrecisionRecallDisplay.from_predictions(y_test, proba_lstm, name="LSTM",␣
  ↪ax=axes[1])
axes[1].set_title("Precision-Recall Curve")

plt.tight_layout()
plt.show()
```

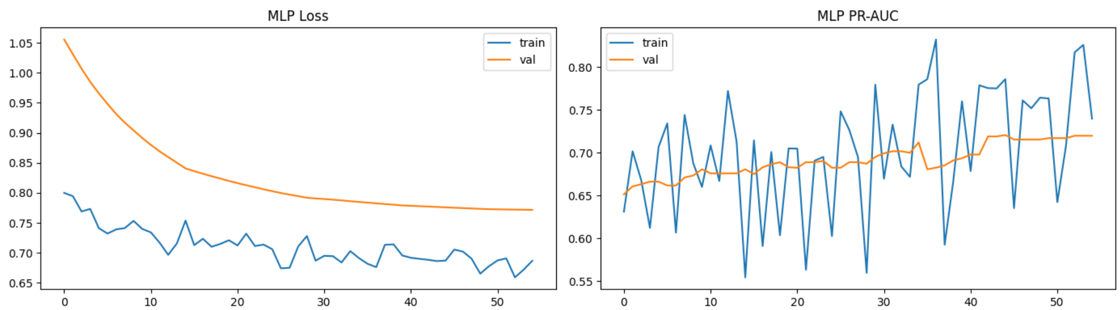


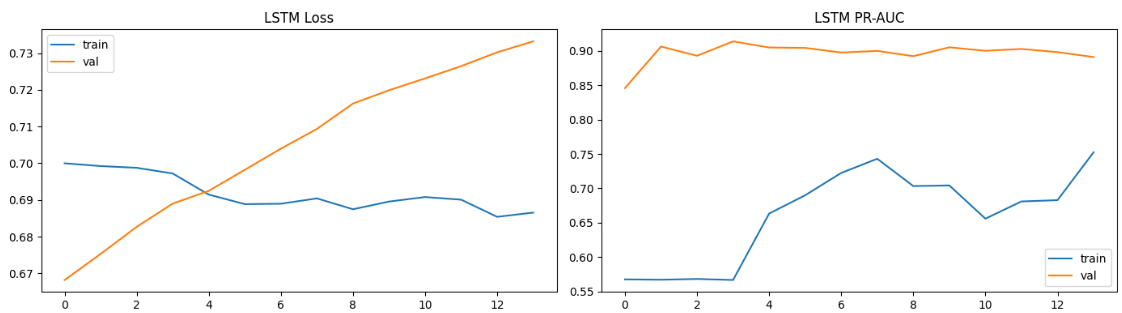
1.0.12 12. Training curves

We visualize learning curves (loss and PR-AUC) to check convergence and overfitting.

[14]:
```
def plot_history(hist, title_prefix):
    hist_df = pd.DataFrame(hist.history)
    fig, axes = plt.subplots(1, 2, figsize=(14, 4))

    axes[0].plot(hist_df["loss"], label="train")
    axes[0].plot(hist_df["val_loss"], label="val")
    axes[0].set_title(f"{title_prefix} Loss")
    axes[0].legend()

    if "pr_auc" in hist_df.columns:
        axes[1].plot(hist_df["pr_auc"], label="train")
        axes[1].plot(hist_df["val_pr_auc"], label="val")
        axes[1].set_title(f"{title_prefix} PR-AUC")
        axes[1].legend()

    plt.tight_layout()
    plt.show()
```

```python
plot_history(history_mlp, "MLP")
plot_history(history_lstm, "LSTM")
```





1.0.13 13. Save models

This saves the trained Keras models for reproducibility.

[15]:
```python
OUT_DIR = "../models"
os.makedirs(OUT_DIR, exist_ok=True)

mlp_path = os.path.join(OUT_DIR, "model2_mlp.keras")
lstm_path = os.path.join(OUT_DIR, "model2_lstm.keras")

mlp.save(mlp_path)
lstm.save(lstm_path)

print("Saved:", mlp_path)
print("Saved:", lstm_path)
```

```
Saved: ../models/model2_mlp.keras
Saved: ../models/model2_lstm.keras
```

1.0.14 14. Export Deep Learning Results for Tableau

[20]:
```
os.makedirs("../outputs", exist_ok=True)

# Required variables from this notebook
required_vars = ["df_tw", "TIME_COL", "SEQ_LEN", "val_end", "y_test"]
missing = [v for v in required_vars if v not in globals()]
if missing:
    raise NameError(f"Missing required variables for export: {missing}. Run the␣
↪earlier cells first.")

# Build tool_wear aligned with sequence samples
# X_seq / y_seq are created from df_tw starting at index SEQ_LEN-1 (label at␣
↪window end)
tool_wear_seq = df_tw[TIME_COL].iloc[SEQ_LEN - 1:].values  # length ==␣
↪len(y_seq)

# Test slice corresponds to X_seq[val_end:], y_seq[val_end:]
tool_wear_test = tool_wear_seq[val_end:]

# Get probabilities
if "proba_lstm" in globals():
    lstm_probs = np.asarray(proba_lstm).ravel()
else:
    # fallback: compute from model + scaled sequences if needed
    if "lstm" not in globals() or "X_test_s" not in globals():
        raise NameError("Missing proba_lstm and cannot recompute (need lstm and␣
↪X_test_s).")
    lstm_probs = np.asarray(lstm.predict(X_test_s)).ravel()

if "proba_mlp" in globals():
    mlp_probs = np.asarray(proba_mlp).ravel()
else:
    if "mlp" not in globals() or "X_test_last" not in globals():
        raise NameError("Missing proba_mlp and cannot recompute (need mlp and␣
↪X_test_last).")
    mlp_probs = np.asarray(mlp.predict(X_test_last)).ravel()

# Safety: ensure lengths match
n = len(y_test)
if len(tool_wear_test) != n:
    raise ValueError(f"tool_wear_test length ({len(tool_wear_test)}) != y_test␣
↪length ({n}).")
if len(lstm_probs) != n:
    raise ValueError(f"lstm_probs length ({len(lstm_probs)}) != y_test length␣
↪({n}).")
if len(mlp_probs) != n:
```

```
    raise ValueError(f"mlp_probs length ({len(mlp_probs)}) != y_test length␣
↪({n}).")

# Recommended threshold (Youden’s J) using LSTM
fpr, tpr, thresholds = roc_curve(y_test, lstm_probs)
optimal_idx = (tpr - fpr).argmax()
optimal_threshold = float(thresholds[optimal_idx])
print(f"Recommended threshold (LSTM - Youden’s J): {optimal_threshold:.6f}")

# Predicted labels for Tableau filters
pred_label_lstm = (lstm_probs >= optimal_threshold).astype(int)
pred_label_mlp  = (mlp_probs  >= optimal_threshold).astype(int)

# Export
export_df = pd.DataFrame({
    "tool_wear": tool_wear_test,
    "failure_actual": np.asarray(y_test).astype(int),
    "failure_prob_lstm": lstm_probs,
    "failure_prob_mlp": mlp_probs,
    "pred_label_lstm": pred_label_lstm,
    "pred_label_mlp": pred_label_mlp,
    "recommended_threshold": optimal_threshold
}).sort_values("tool_wear")

export_path = "../outputs/pred_failure_deep.csv"
export_df.to_csv(export_path, index=False)

print(f"Saved: {export_path} | rows={len(export_df)}")
```

```
Recommended threshold (LSTM - Youden’s J): 0.503694
Saved: ../outputs/pred_failure_deep.csv | rows=167
```

1.0.15 14. Summary

In this notebook, we implemented deep learning models for predictive maintenance using a time-aware formulation of the AI4I dataset.

Key points:

- `Tool wear [min]` was used as a proxy for temporal progression.
- Data were aggregated by tool wear, preserving the binary failure label using `max()`.
- Sliding-window sequences were created to enable temporal modeling.
- Two deep models were trained:
  - MLP (last timestep baseline)
  - LSTM (sequence model)

Results show that:

- The MLP serves as a strong tabular deep-learning baseline.
- The LSTM captures temporal dependencies across tool-wear progression.

- Precision-Recall AUC is particularly important due to class imbalance.

Overall, Model 2 demonstrates how sequence-based deep learning can extend traditional machine learning approaches for failure prediction, while maintaining strict time-aware splitting to avoid data leakage.