Overview of the Final_Project

This project implements a **complete machine learning pipeline** for sensor-based fault detection and predictive maintenance. It is designed to handle both **real-world datasets** and **synthetic datasets**, ensuring flexibility and reproducibility across different scenarios. The project covers **data collection, preprocessing, training, calibration, evaluation, and visualization**, producing artifacts for diagnostics and deployment.

Key Stages

1. **Data Collection & Cleaning** Raw sensor data is ingested, cleaned, and stored in dataset-aware paths for consistency.
2. **Preprocessing** Features are engineered, sequences are generated, and per-sensor scaling is applied. The output is a structured .npz file containing train, validation, and test splits.
3. **Model Training** A shared encoder is built using a **BiLSTM with temporal attention** and sensor embeddings. The encoder fuses temporal context, last-step features, and sensor identity. Training uses class weights for stability and exports the encoder to ONNX format.
4. **Evaluation** The pipeline reports **global metrics** (ROC-AUC, PR-AUC, F1, Balanced Accuracy, MCC) and **per-sensor PR-AUC** to highlight top and worst-performing sensors.
5. **Visualization** Static and interactive plots are generated to show feature distributions, metric trends, prediction comparisons, and real vs synthetic dataset differences.

Outputs:

- Cleaned datasets (.csv)
- Sequence files (.npz)
- Trained encoder (.h5) and ONNX export
- Predictions (.npz)
- Evaluation logs and plots (.png, .html)

1. Purpose and Overview of config.py module.

The config.py module is the centralized configuration file for the machine learning pipeline. It defines paths, dataset sources, canonical column names, preprocessing policies, training parameters, and artifact locations. By consolidating these settings, the module ensures consistency, reproducibility, and maintainability across all stages of the workflow.

2. Directory and Path Management: The module uses Python's **pathlib.Path** to manage directories

```
ROOT       = Path(__file__).resolve().parent
DATA_DIR   = ROOT / "data"
ARTIFACTS  = ROOT / "artifacts"
MODEL_DIR  = ARTIFACTS / "model"
PLOTS_DIR  = ARTIFACTS / "plots"
PRED_DIR   = ARTIFACTS / "predictions"
CALIB_DIR  = ARTIFACTS / "calibrators"
```

These directories are automatically created to prevent runtime errors.

3. Dataset Configuration: The pipeline supports two modes: **real sensor data** and **synthetic data**.

DATASET_MODE = "real" # "real" or "synthetic"
REAL_DATA_FILE = "iot_real_sensor_dataset.csv"
SYNTH_DATA_FILE= "synthetic_iot_dataset.csv"

This flexibility allows seamless switching between real-world experiments and controlled synthetic benchmarks.

4. Canonical Columns: To ensure consistency:

TIMESTAMP_COL= "timestamp"; SENSOR_ID_COL= "sensor_id"
TARGET_COL = "fault_status"

- **Feature Columns:** temperature, vibration, pressure, voltage, current, fft_feature1, fft_feature2
- **Excluded Columns:** anomaly_score, normalized sensor values

5. Sequence and Labeling Policy: The pipeline processes time-series data in fixed sequences:

SEQ_LEN =6; SEQ_STRIDE =6; LABEL_POLICY = "majority"; RECENT_STEPS= 3

This defines how labels are assigned to sequences.

6. Data Splitting: To ensure fair evaluation. TEST_SPLIT= 0.20; VAL_SPLIT= 0.20;

GROUP_SPLIT_COL= SENSOR_ID_COL; SEED = 42 This prevents data leakage across sensors.

7. Scaling Policy: Feature scaling is critical: SCALER_TYPE = "robust"; SCALE_ON = "train_only"

8. Training Parameters: Default hyperparameters: BATCH_SIZE = 64; EPOCHS = 20; LR = 2e-4

9. Artifact Management: Artifacts capture intermediate and final results:

- **Real Dataset Artifacts:** sequences, scaler, history, models, predictions, calibrators.
- **Synthetic Dataset Artifacts:** equivalent files for synthetic experiments

REAL_MODEL_ONNX_PATH = MODEL_DIR/ "real_model.onnx"
SYNTH_MODEL_ONNX_PATH= MODEL_DIR/ "synth_model.onnx"

10. Synthetic Defaults: For synthetic dataset generation: SYNTH_ROWS = 20_000; SENSOR_COUNT= 10

Purpose and Overview of collect_data module.

The collectdata.py module is responsible for **loading, generating, and cleaning datasets** used in the machine learning pipeline. It supports both **real sensor data** and **synthetic datasets**, depending on the configuration set in config.py. **Key responsibilities:**

- Load the dataset specified by config.DATASET_MODE.
- Generate synthetic data if required and save it for reuse.

- Clean and enforce canonical schema by dropping excluded columns and ensuring consistent data types.
- Save the cleaned dataset back to the appropriate directory.

This module ensures that downstream preprocessing and training stages always receive a **well-structured, reproducible dataset**.

2.1. Core Functions

2.1 Synthetic Dataset Generation: The function generate_synthetic() creates a synthetic IoT dataset with structured fault episodes and sensor features.

```
def generate_synthetic(rows=config.SYNTH_ROWS, sensors=config.SENSOR_COUNT,
fault_prob=0.05) -> pd.DataFrame:
    # Generates timestamps, sensor IDs, and fault labels
    # Builds contiguous fault episodes (5–20 steps)
    # Creates base signals with drift and fault-dependent shifts
    # Returns a DataFrame with canonical columns
```

- **Timestamps:** Minute-level granularity starting from "2025-01-01".
- **Fault Labels:** Initially Bernoulli distributed, then stitched into contiguous episodes.
- **Features:** Sensor drift plus fault-dependent mean shifts and variance spikes.
- **Output:** A DataFrame aligned with canonical schema defined in config.py.

2.2 Dataset Loading: The function load_dataset() decides whether to load a real dataset or synthetic dataset.

```
def load_dataset() -> pd.DataFrame:
    if config.DATASET_MODE == "real":
        df = pd.read_csv(config.REAL_DATA_PATH)
    else:
        try:
            df = pd.read_csv(config.SYNTH_DATA_PATH)
        except FileNotFoundError:
            df = generate_synthetic(config.SYNTH_ROWS, config.SENSOR_COUNT)
            df.to_csv(config.SYNTH_DATA_PATH, index=False
```

- **Real Mode:** Loads from REAL_DATA_PATH.
- **Synthetic Mode:** Attempts to load from SYNTH_DATA_PATH. If missing, generates synthetic data and saves it.
- **Reproducibility:** Ensures synthetic datasets are stored for consistent reuse.

2.3 Data Cleaning: The function clean() enforces schema consistency and sanitizes the dataset.

```
def clean(df: pd.DataFrame) -> pd.DataFrame:
    # Drops excluded columns
    # Ensures canonical columns exist
    # Converts timestamp and target types
    # Sanitizes feature columns
    # Sorts chronologically per sensor
```

- **Excluded Columns:** Removes auxiliary or derived features listed in config.EXCLUDE_COLS.
- **Canonical Schema:** Retains only timestamp, sensor ID, target, and feature columns.

- **Type Conversion:** Ensures timestamps are valid datetime objects and target labels are integers.
- **Feature Sanitation:** Converts features to numeric, handling invalid entries gracefully.
- **Sorting:** Orders data chronologically per sensor, ensuring sequence integrity.

2.4 Main Execution: The main() function integrates loading and cleaning, then saves the final dataset.

```
def main():

    df = load_dataset()
    df = clean(df)
    out_path = config.REAL_DATA_PATH if config.DATASET_MODE == "real" else
config.SYNTH_DATA_PATH
    df.to_csv(out_path, index=False)
    print(f"Saved cleaned dataset to {out_path}")
```

1.Purpose and Overview of Preprocessing module:

The module is responsible for preparing raw sensor data into structured sequences suitable for machine learning models.The preprocessing module performs the following tasks:

- Cleans and loads datasets.
- Splits data into train, validation, and test sets grouped by sensor ID.
- Adds statistical and temporal features.
- Applies robust scaling across datasets.
- Generates sequential samples for model training.
- Saves processed sequences for downstream tasks.

2. Function Documentation:

2.1 split_time_grouped(df)

Splits the dataset into train, validation, and test sets grouped by sensor ID.

```
def split_time_grouped(df):
    train_parts, val_parts, test_parts = [], [], []
    for sid, g in df.groupby(config.SENSOR_ID_COL):
        g = g.sort_values(config.TIMESTAMP_COL)
        n = len(g)
        n_test = int(n * config.TEST_SPLIT)
        n_trainval = n - n_test
        n_val = int(n_trainval * config.VAL_SPLIT)
        test_parts.append(g.iloc[-n_test:])
        val_parts.append(g.iloc[n_trainval - n_val:n_trainval])
        train_parts.append(g.iloc[:n_trainval - n_val])
    return (pd.concat(train_parts).reset_index(drop=True),
            pd.concat(val_parts).reset_index(drop=True),
            pd.concat(test_parts).reset_index(drop=True)
```

- Groups data by sensor ID.
- Ensures chronological order using timestamps.
- Applies configurable split ratios.

2.2 add_features(g: pd.DataFrame, win_std=5, win_ma=10)

Adds rolling statistical features and deltas to enhance signal representation.

```
def add_features(g: pd.DataFrame, win_std=5, win_ma=10):

g = g.copy()
    for c in config.FEATURE_COLS:
        g[f"{c}_delta"] = g[c].diff()
        g[f"{c}_rstd{win_std}"] = g[c].rolling(win_std, min_periods=1).std()
        g[f"{c}_ma{win_ma}"] = g[c].rolling(win_ma, min_periods=1).mean()
    g.fillna(0.0, inplace=True)
    return g
```

- Computes first-order differences (delta).
- Rolling standard deviation (rstd) and mean (ma) capture local variability and trends.
- Missing values are filled with zeros.

2.3 scale_global(train_df, val_df, test_df) : Applies robust scaling across train, validation, and test sets.

```
def scale_global(train_df, val_df, test_df):
    sc = RobustScaler()
    sc.fit(train_df[config.FEATURE_COLS])
    for df in (train_df, val_df, test_df):
        df.loc[:, config.FEATURE_COLS] = sc.transform(df[config.FEATURE_COLS])
    if config.DATASET_MODE == "real":
        dump(sc, config.REAL_SCALER_PATH)
    else:
        dump(sc, config.SYNTH_SCALER_PATH)
    return train_df, val_df, test_df
```

- Fits scaler only on training data.
- Applies transformation consistently across all splits.
- Saves scaler for reproducibility.

2.4 make_sequences(df, L=12, S=12): Generates sequential samples for supervised learning.

```
def make_sequences(df, L=12, S=12):
    X_seq, y, X_last, SIDs = [], [], [], []
    for sid, g in df.groupby(config.SENSOR_ID_COL):
        g = g.sort_values(config.TIMESTAMP_COL)
        g = add_features(g)
        seq_cols = config.FEATURE_COLS + [f"{c}_delta" for c in config.FEATURE_COLS] +
[f"{c}_rstd5" for c in config.FEATURE_COLS]
        F = g[seq_cols].values
        T = g[config.TARGET_COL].values
        for i in range(0, len(g) - L + 1, S):
            j = i + L
            X_seq.append(F[i:j])
            X_last.append(F[j-1])
            label = int(T[j-1])  # last-step label policy
```

```
            y.append(label)
            SIDs.append(sid)
    return np.array(X_seq), np.array(y), np.array(X_last), np.array(SIDs)
```

- Creates overlapping sequences of length L with stride S.
- Labels are derived from the last timestep.
- Returns sequences, labels, last-step features, and sensor IDs.

2.5 main(): Coordinates the preprocessing pipeline.

```
def main():
    df = clean(load_dataset())
    tr, va, te = split_time_grouped(df)
    tr_s, va_s, te_s = scale_global(tr, va, te)

    X_tr, y_tr, X_tr_last, sid_tr = make_sequences(tr_s)
    X_va, y_va, X_va_last, sid_va = make_sequences(va_s)
    X_te, y_te, X_te_last, sid_te = make_sequences(te_s)

    SEQ_PATH = config.REAL_SEQ_PATH if config.DATASET_MODE == "real" else
config.SYNTH_SEQ_PATH
    np.savez_compressed(
        SEQ_PATH,
        X_train=X_tr, y_train=y_tr, sensor_train=sid_tr,
        X_val=X_va,   y_val=y_va,   sensor_val=sid_va,
        X_test=X_te,  y_test=y_te,  sensor_test=sid_te,
        X_train_last=X_tr_last, X_val_last=X_va_last, X_test_last=X_te_last,
    )

    print("Unique values of y_train:", np.unique(y_tr))
    print("Unique values of y_val:", np.unique(y_va))
    print("Unique values of y_test:", np.unique(y_te))
    print("Shapes:", y_tr.shape, y_va.shape, y_te.shape)
    print("train:", X_tr.shape, "pos_rate=", float(y_tr.mean()),
          "| val:", X_va.shape, "pos_rate=", float(y_va.mean()),
          "| test:", X_te.shape, "pos_rate=", float(y_te.mean()))

if __name__ == "__main__":
    main()
```

This picture shows the output of preprocessing.py module.

```
Cleaned target unique values: [0 1]
Unique values of y_train: [0 1]
Unique values of y_val: [0 1]
Unique values of y_test: [0 1]
Shapes: (1062,) (259,) (329,)
train: (1062, 12, 21) pos_rate= 0.4209039548022599 | val: (259, 12, 21) pos_rate= 0.41312741312741313 | tes
: (329, 12, 21) pos_rate= 0.331306990881459
PS C:\Users\Niazi Wall\IOT_Project>
```

The preprocessing output confirms that the target labels are binary ([0, 1]) and evenly distributed across train, validation, and test sets. Each split contains sequences of shape (samples, 12, 21), where 12 is the sequence length and 21 is the number of features per timestep. The positive class rate is around 42% in training, 41% in validation, and 33% in testing, indicating mild class imbalance. These structured arrays are saved for model training.

1. **Purpose and Overview of trainmodel.py module:**

This module defines and trains the sequence encoder model with attention for sensor data classification. It integrates advanced neural network components to effectively process sequential sensor data and produce robust classification outputs.

- Build a neural encoder using LSTM and attention mechanisms to capture temporal dependencies and highlight important time steps.
- Train the model using sensor sequences, last-step features, and sensor IDs to leverage multiple data modalities.
- Evaluate model performance comprehensively using metrics such as ROC-AUC, PR-AUC, F1 score, accuracy, and Matthews Correlation Coefficient (MCC) to ensure balanced assessment.
- Save trained models, training histories, prediction outputs, and export the encoder to ONNX format for interoperability and deployment.

Key Components: 1. **TemporalAttention Layer**: A custom Keras layer that applies an attention mechanism over the output sequence of the encoder. This layer learns to assign different importance weights to each timestep in the input sequence, allowing the model to focus on the most relevant parts of the data.

```
class TemporalAttention(tf.keras.layers.Layer):
  def __init__(self, units=64, **kwargs): ...
  def call(self, inputs): ...
  def get_config(self): ...
```

- Learns a set of weights over the timesteps dynamically during training.
- Produces a context vector by computing a weighted sum of the sequence outputs, effectively summarizing the sequence information.

2. build_encoder()

This function constructs the full encoder model that processes multiple input types: sequence features, last-step features, and sensor ID embeddings. It combines these inputs through a series of layers to produce a final classification output.

```
def build_encoder(seq_len, n_seq_feats, n_last_feats, n_sensors, lr=2e-4, emb_dim=16): ...
```

- Inputs include time-series sequence features, static last-step features, and categorical sensor IDs embedded into a dense vector space.
- The architecture consists of a bidirectional LSTM layer to capture temporal patterns, followed by the TemporalAttention layer to focus on important timesteps.
- Outputs from the attention layer are concatenated with last-step features and sensor embeddings.
- The combined features pass through a dense layer with sigmoid activation to produce a binary classification output.
- The model is compiled with the Adam optimizer, binary crossentropy loss, and metrics including accuracy, ROC-AUC, and PR-AUC.
- Returns both the full model for training and a feature extractor model for downstream use.

3. export_encoder_onnx()

This function exports the trained feature extractor model to the ONNX format, enabling interoperability with other frameworks and deployment environments.

def export_encoder_onnx(feat_model, onnx_path, seq_len, n_seq_feats, n_last_feats): ...

- Utilizes the tf2onnx library to convert the TensorFlow Keras model to ONNX.
- Logs the path where the ONNX model is saved for reference.

4. main(): The main function orchestrates the entire training and evaluation pipeline.

def main():

- Load dataset paths depending on whether the mode is real or synthetic data.
- Load the sensor sequences and corresponding labels for training and testing.
- Normalize sensor IDs to ensure consistent embedding.
- Build the encoder model using the specified architecture.
- Train the model with callbacks such as EarlyStopping to prevent overfitting, ModelCheckpoint to save the best model, and ReduceLROnPlateau to adjust learning rate dynamically.
- Save the trained model and the training history for later analysis.
- Evaluate the model on the test set using multiple metrics to assess performance thoroughly.
- Save the prediction outputs for further inspection or downstream tasks.
- Export the encoder to ONNX format for deployment or integration.

This picture shows the output of trainmodel.py module.

```
Epoch 19: val_auc_pr did not improve from 0.96063

Epoch 19: ReduceLROnPlateau reducing learning rate to 4.999999873689376e-05.
17/17 - 0s - 21ms/step - accuracy: 0.9058 - auc_pr: 0.9524 - auc_roc: 0.9504 - loss: 0.2721 - val_accuracy:
0.9073 - val_auc_pr: 0.9601 - val_auc_roc: 0.9612 - val_loss: 0.2518 - learning_rate: 1.0000e-04
Epoch 20/20

Epoch 20: val_auc_pr did not improve from 0.96063
17/17 - 0s - 24ms/step - accuracy: 0.9068 - auc_pr: 0.9533 - auc_roc: 0.9516 - loss: 0.2682 - val_accuracy:
0.9073 - val_auc_pr: 0.9604 - val_auc_roc: 0.9616 - val_loss: 0.2516 - learning_rate: 5.0000e-05
Epoch 20: early stopping
Restoring model weights from the end of the best epoch: 15.
2025-12-26 00:02:12,283 | INFO | Saved model to C:\Users\Niazi Wall\IOT_Project\artifacts\model\synth_model.
keras and history to C:\Users\Niazi Wall\IOT_Project\artifacts\synth_history.joblib
6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 103ms/step
2025-12-26 00:02:13,295 | INFO | Test ROC-AUC=0.936 PR-AUC=0.916 F1=0.820 Acc=0.570 MCC=0.741
2025-12-26 00:02:13,295 | INFO | Saved predictions to C:\Users\Niazi Wall\IOT_Project\artifacts\predictions\
synth_predictions.npz
```

The training module completed 20 epochs and stopped early after no further improvement in validation AUC-PR. It restored the best model from epoch 15 and saved it along with the training history. On the test set, the model achieved strong performance: ROC-AUC of 0.936, PR-AUC of 0.916, F1 score of 0.820, and MCC of 0.741. Final predictions were saved to a compressed .npz file for further analysis

1. Purpose and Overview of evaluate.py module:

This module evaluates a trained encoder model on test data and computes classification metrics. It identifies the optimal threshold using the Matthews correlation coefficient (MCC), logs performance metrics, and saves predictions for downstream analysis.

- Load test sequences and trained model.
- Predict probabilities and sweep thresholds.
- Select optimal threshold based on MCC.
- Compute evaluation metrics.
- Save predictions and diagnostics.

Key Components: **1. Environment:**

Setup Suppresses TensorFlow warnings and configures logging.

```
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
logging.basicConfig(...)
```

2. Path Resolution: Chooses dataset-specific paths for sequences, model, and predictions.

```
if config.DATASET_MODE == "real":
    SEQ_PATH = config.REAL_SEQ_PATH
    MODEL_PATH = config.REAL_MODEL_KERAS_PATH
    PRED_PATH = config.REAL_PRED_PATH
```

3. Data Loading: Loads test sequences and sensor IDs.

```
data = np.load(SEQ_PATH, allow_pickle=True)
X_test, y_test = data["X_test"], data["y_test"].astype(int).ravel()
X_test_last = data["X_test_last"];  sid_test = data["sensor_test"];  sid_test_idx = sid_test -
sid_test.min()
```

4. Model Loading: Loads the trained Keras model with custom attention layer.

```
model = tf.keras.models.load_model(
    MODEL_PATH, compile=False;  custom_objects={"TemporalAttention": TemporalAttention})
```

5. Prediction and Threshold Optimization: Predicts probabilities and finds the best threshold for MCC.

```
y_prob = model.predict([...]).ravel()
thresholds = np.linspace(0, 1, 101)
mccs = [matthews_corrcoef(y_test, (y_prob >= t).astype(int)) for t in thresholds]
best_mcc_t = thresholds[np.argmax(mccs)]
y_pred = (y_prob >= best_mcc_t).astype(int)
```

6. Metric Reporting:Logs evaluation metrics.

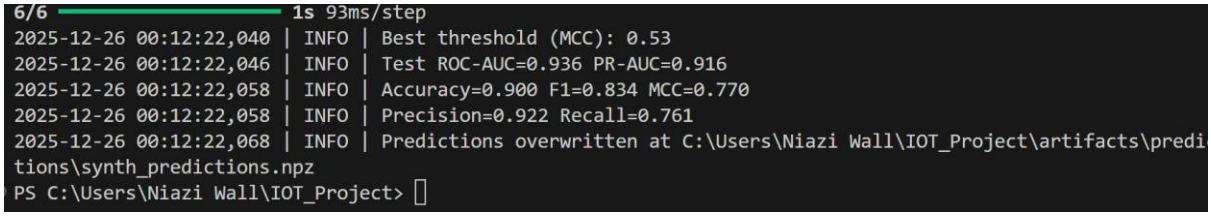
```
logger.info("Test ROC-AUC=%.3f PR-AUC=%.3f", ...) logger.info("Accuracy=%.3f F1=%.3f
MCC=%.3f", ...)
logger.info("Precision=%.3f Recall=%.3f", ...)
```

7. Save Predictions: Stores predictions and threshold in compressed format.

```
np.savez_compressed(PRED_PATH, y_true=..., y_prob=..., y_pred=..., sensor_ids=..., threshold=...)
```

This is the

This picture shows the output of evaluate.py module.



The model was tested on unseen data and predicted class probabilities. It swept thresholds from 0 to 1 and selected **0.53** as the best cutoff based on **MCC** (Matthews correlation coefficient). Final metrics were logged: **ROC-AUC = 0.936**, **PR-AUC = 0.916** → strong ranking performance. **Accuracy = 0.900**, **F1 = 0.834**, **MCC = 0.770**, **Precision = 0.922**, **Recall = 0.761** → balanced classification quality. All predictions, sensor IDs, and the chosen threshold were saved to a .npz file for downstream analysis.

1. Purpose and Overview of plots.py module:

This module generates visualizations and exports evaluation metrics for model performance analysis. It supports both real and synthetic datasets and produces plots and summaries for classification results.

- Visualize confusion matrix, ROC and PR curves.
- Compare feature distributions between real and synthetic datasets.
- Export evaluation metrics to CSV.

Functions: 1. plot_confusion_matrix(y_true, y_pred, out_dir=pred_dir): Generates and saves a labeled confusion matrix.

```
cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ...)
fig.savefig(...)
```

- Labels: "Healthy (0)" vs "Faulty (1)"
- Output: PNG image saved to pred_dir

2. plot_roc_pr_curves(y_true, y_prob, out_dir=plots_dir): Plots and saves ROC and Precision-Recall curves.

```
fpr, tpr, _ = roc_curve(y_true, y_prob)
prec, rec, _ = precision_recall_curve(y_true, y_prob)
fig.savefig(... )
```

ROC: plots TPR vs FPR; PR: plots Precision vs Recall; Output: PNG images saved to plots_dir

3. compare_selected_features(real_df, synth_df, out_dir=plots_dir): Compares distributions of selected features across datasets.

```
selected_features = ["temperature", "vibration", "pressure"]
px.histogram(...)
sns.histplot(...)
fig.savefig(...)
```

- Features: temperature, vibration, pressure
- Output: HTML and PNG plots saved to plots_dir

4. export_metrics_summary(y_true, y_pred, threshold=None, out_dir=plots_dir): Exports classification metrics to a CSV file.
metrics = {"Accuracy": ..., "F1": ..., "MCC": ..., "Precision": ..., "Recall": ...}
df.to_csv(...). Output: CSV file saved to plots_dir
5. main(): Coordinates the full visualization and export workflow.
data = np.load(...) plot_confusion_matrix(...) plot_roc_pr_curves(...) compare_selected_features(...)

1. Purpose and Overview of main.py module:

export_metrics_summary(...). Loads predictions and datasets. Executes all plotting and export functionsThis module orchestrates the entire machine learning pipeline, coordinating dataset preparation, preprocessing, model training, evaluation, and visualization. It acts as the central entry point for running the full workflow.

- Load and clean raw datasets (real or synthetic).
- Preprocess data (splitting, scaling, feature engineering, sequence generation).
- Train the encoder model (BiLSTM + attention, ONNX export).
- Evaluate model performance with multiple metrics.
- Generate plots and export metrics summaries.

1. Load and Clean Dataset
df = load_dataset()
df = clean(df)
out_path = config.REAL_DATA_PATH if config.DATASET_MODE == "real" else config.SYNTH_DATA_PATH
df.to_csv(out_path, index=False)

- Loads dataset based on mode.
- Cleans data and saves to CSV.

2. Preprocess
preprocess_main()

- Splits data into train/validation/test.
- Applies scaling and feature engineering.
- Generates sequences and saves them as .npz files.

3. Train
train_main()

- Builds and trains BiLSTM + attention encoder.
- Exports trained model and ONNX feature extractor.

4. Evaluate
evaluate_main()

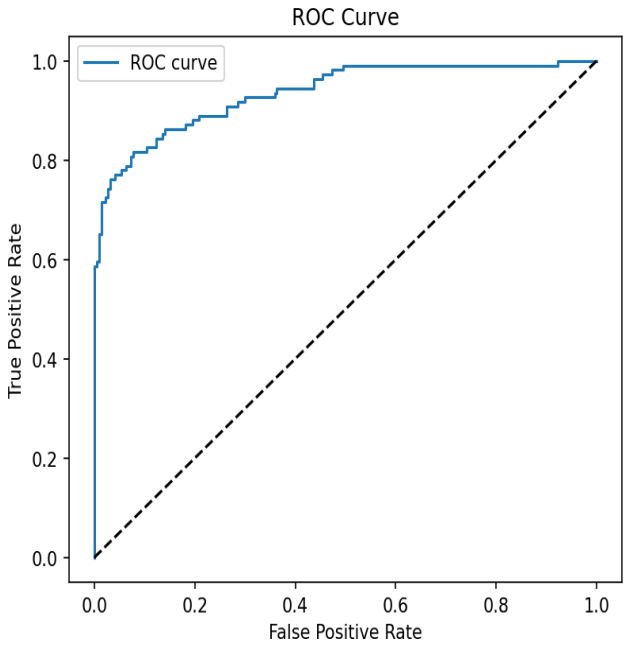
- Computes ROC-AUC, PR-AUC, Accuracy, F1, MCC, Precision, Recall.
- Saves predictions and metrics summary.

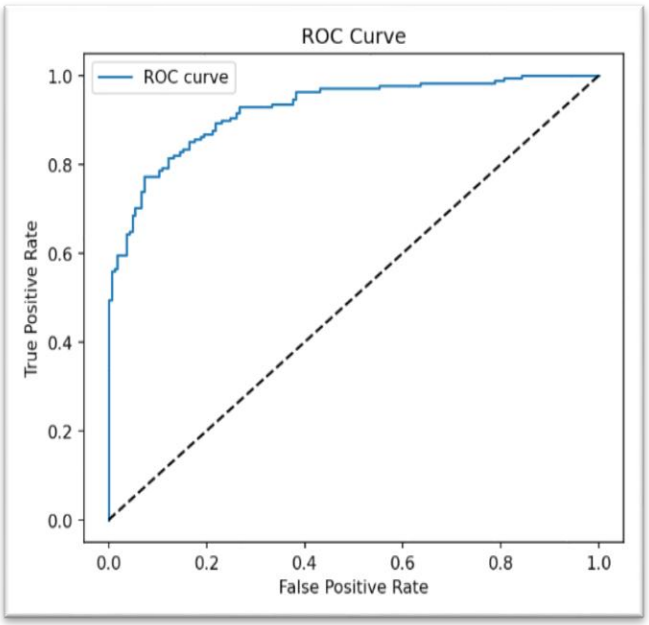
5. Plot Results

plots_main()

- Generates confusion matrix, ROC/PR curves.
- Compares real vs synthetic feature distributions.
- Exports metrics summary to CSV.

The outputs of the plots.py **module are saved in the** plots **directory. I will show just four sample images from the results here.**

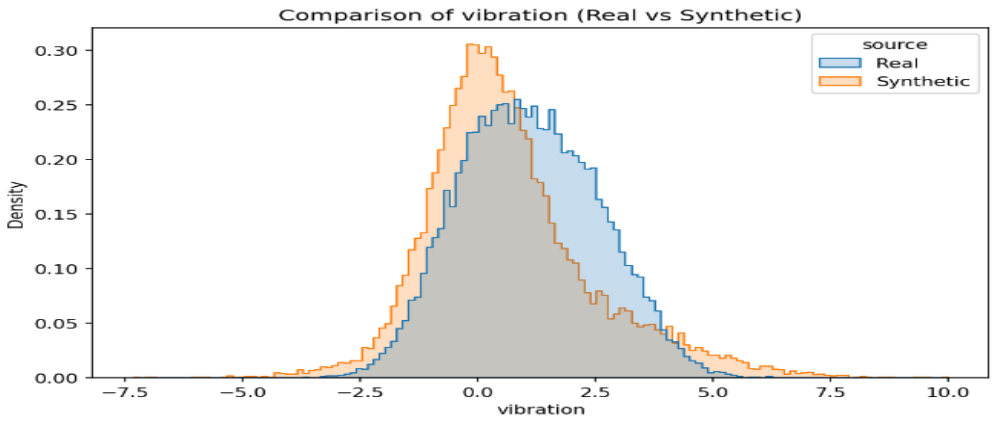




The left picture is from real dataset, and the right picture is from synthetic dataset

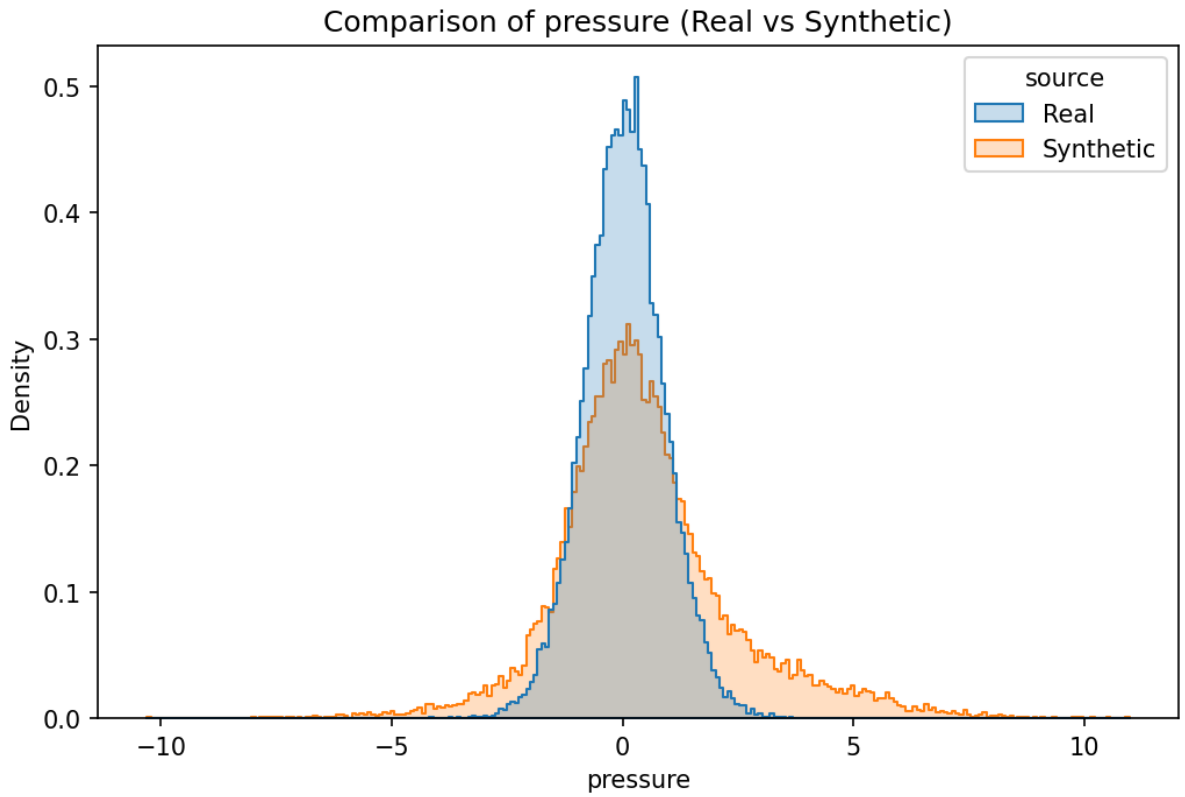
These two pictures are the ROC curve form my real and synthetic datasets that shows the model's ability to distinguish between healthy and faulty cases. The blue line represents true positive rate vs false positive rate across thresholds, and its position above the diagonal indicates strong classification performance. The closer the curve hugs the top-left corner, the better the model's sensitivity and specificity.

This picture compare vibration between real and synthetic dataset:



This density plot compares the distribution of vibration values between real and synthetic datasets. The overlapping curves show that both sources follow similar patterns, with slight differences in spread and peak density. This visualization helps assess how well the synthetic data mimics real-world behavior.

This picture compares the distribution of pressure between two datasets.



This density plot compares pressure distributions between real and synthetic datasets. Both are centered around zero, but the real data shows a sharper peak and narrower spread, while the synthetic data has broader tails. This indicates that the synthetic pressure values are more varied and less concentrated than the real ones.