AI ASSISTED CODING END-EXAM LAB EXAM

NAME:T.SHIRISHA

HTNO:2403A52077

BATCH-04

SUBSET:04

Subset 4 — Debugging Data Inconsistencies Across Sensors
Q1: Diagnose time-synchronization drift between sensors.
• Task 1: Use AI to analyze timestamps and suggest corrections.
• Task 2: Implement synchronization layer and test.

PROMPT:
Generate a python code Debugging Data Inconsistencies Across Sensors
Task 1: Use AI to analyze timestamps and suggest corrections.
“Analyze the uploaded sensor dataset to detect time-synchronization drift between sensors. Identify timestamp mismatches, estimate drift amount, and suggest corrections.
Task 2: Implement synchronization layer and test.
Then propose a simple synchronization layer and outline how to test it.”

CODE:

```python
import numpy as np
from sklearn.linear_model import RANSACRegressor, LinearRegression
# --- Replace these with your real timestamp arrays (seconds since epoch) ---
# Example synthetic data for demo; remove and load your real arrays instead.
t_sensor = np.array([1_600_000_000.0, 1_600_000_100.0, 1_600_000_200.0, 1_600_000_300.0, 1_600_000_400.0])
t_ref    = np.array([1_600_000_002.5, 1_600_000_102.8, 1_600_000_203.1, 1_600_000_303.6, 1_600_000_404.0])
# Basic validation
if t_sensor.size == 0 or t_ref.size == 0:
    raise SystemExit("t_sensor or t_ref is empty — provide numeric arrays.")
if t_sensor.shape != t_ref.shape:
    raise SystemExit("t_sensor and t_ref must have the same shape.")
if not (np.isfinite(t_sensor).all() and np.isfinite(t_ref).all()):
    raise SystemExit("t_sensor/t_ref contain non-finite values (NaN/inf).")
# Robust linear fit: t_ref = a + b * t_sensor
base_estimator = LinearRegression()
robust = RANSACRegressor(estimator=base_estimator, residual_threshold=1.0, random_state=42)
try:
    robust.fit(t_sensor.reshape(-1, 1), t_ref)
except Exception as e:
    raise SystemExit(f"RANSAC fit failed: {e}")
b = float(robust.estimator_.coef_[0])
a = float(robust.estimator_.intercept_)
print(f"Estimated mapping: t_ref ≈ a + b * t_sensor")
print(f"  offset a = {a:.6f} seconds")
print(f"  slope  b = {b:.12f}  (skew => {(b-1.0)*1e6:.3f} ppm)")
# Correction function (accepts scalar or array-like)
def correct(ts):
    return a + b * np.array(ts, dtype=float)
# Quick test / example
sample_sensor = np.array([1_600_000_500.0])
print("Sensor time:", sample_sensor)
print("Corrected time:", correct(sample_sensor))
```

OUTPUT:

```
on.exe" "c:/Users/kurapati pruthvi/OneDrive/Desktop/AI-endexam.py"
Estimated mapping: t_ref ≈ a + b * t_sensor
  offset a = -6079997.483705 seconds
  slope  b = 1.003799999952  (skew => 3800.000 ppm)
Sensor time: [1.6000005e+09]
Corrected time: [1.6000005e+09]
PS C:\Users\kurapati pruthvi\AppData\Local\Programs\Microsoft VS Code>
```

OBSERVATION:

1. The code loads two timestamp arrays and performs basic validation to ensure they are non-empty, same-size, and contain only valid numeric values.

2. It uses a robust RANSAC regression model to find the relationship between sensor time and reference time, reducing the impact of outliers.

3. After fitting, it extracts the offset (a) and clock skew (b), which describe how much the sensor clock is shifted and how fast/slow it runs.

4. A correction function is created that applies the linear mapping so sensor timestamps can be converted into corrected reference-aligned timestamps.

5. Finally, the code tests the correction function using a sample timestamp and prints the corrected result.

Q2: Implement device heartbeat monitoring.
• Task 1: Use AI to generate aggregator and alert trigger.
• Task 2: Add retry & exponential backoff.

PROMPT:
Generate a python code Implement device heartbeat monitoring.
Task 1: Use AI to generate aggregator and alert trigger.
“Create a device-heartbeat monitoring system. Generate an AI-assisted heartbeat aggregator and alert trigger that detects missed heartbeats.
Task 2: Add retry & exponential backoff.
Implement retry logic with exponential backoff for unstable devices. Provide clean, production-ready code.”

CODE:

```
import numpy as np
import pandas as pd
from scipy.interpolate import interp1d
from sklearn.metrics import mean_absolute_error, mean_squared_error
# ==============================================================================
# PART 1: GENERATE SYNTHETIC DATA WITH MISSING PACKETS
# ==============================================================================
def generate_sensor_data(n_points=100, seed=42):
    """Generate synthetic sensor data."""
    np.random.seed(seed)
    time = np.arange(0, n_points, 1.0)
    # True signal: sine wave + trend
    true_signal = 20.0 + 5.0 * np.sin(2 * np.pi * time / n_points) + 0.05 * time
    # Add noise
    noise = np.random.normal(0, 0.5, n_points)
    signal = true_signal + noise
    return time, signal, true_signal
def introduce_missing_packets(time, signal, missing_ratio=0.1):
    """Randomly remove packets (simulate packet loss)."""
    n_missing = int(len(signal) * missing_ratio)
    missing_indices = np.random.choice(len(signal), n_missing, replace=False)
    missing_indices = sorted(missing_indices)
    mask = np.ones(len(signal), dtype=bool)
    mask[missing_indices] = False
    time_sparse = time[mask]
    signal_sparse = signal[mask]
    return time, signal, time_sparse, signal_sparse, missing_indices, mask
# ==============================================================================
# PART 2: INTERPOLATION METHODS
# ==============================================================================
def linear_interpolation(time, signal_sparse, time_full, mask):
    f = interp1d(time[mask], signal_sparse, kind='linear', fill_value='extrapolate')
    return f(time_full)
def cubic_interpolation(time, signal_sparse, time_full, mask):
    if np.sum(mask) < 4:
        return linear_interpolation(time, signal_sparse, time_full, mask)
    f = interp1d(time[mask], signal_sparse, kind='cubic', fill_value='extrapolate')
```

```python
def cubic_interpolation(time, signal_sparse, time_full, mask):
    return f(time_full)
def nearest_neighbor_interpolation(time, signal_sparse, time_full, mask):
    f = interp1d(time[mask], signal_sparse, kind='nearest', fill_value='extrapolate')
    return f(time_full)
def forward_fill(time, signal_sparse, time_full, mask):
    result = np.full_like(time_full, np.nan, dtype=float)
    sparse_idx = 0
    for i in range(len(time_full)):
        if mask[i]:
            result[i] = signal_sparse[sparse_idx]
            last_value = signal_sparse[sparse_idx]
            sparse_idx += 1
        else:
            result[i] = last_value if 'last_value' in locals() else np.nan
    return result
def polynomial_interpolation(time, signal_sparse, time_full, mask, degree=3):
    coeffs = np.polyfit(time[mask], signal_sparse, degree)
    poly = np.poly1d(coeffs)
    return poly(time_full)
# ============================================================================
# PART 3: ACCURACY EVALUATION
# ============================================================================
def evaluate_interpolation(true_signal, interpolated_signal, missing_indices, method_name):
    true_missing = true_signal[missing_indices]
    pred_missing = interpolated_signal[missing_indices]
    mae = mean_absolute_error(true_missing, pred_missing)
    rmse = np.sqrt(mean_squared_error(true_missing, pred_missing))
    mape = np.mean(np.abs((true_missing - pred_missing) / np.abs(true_missing))) * 100
    print(f"\n{method_name}:")
    print(f"  MAE:  {mae:.6f}")
    print(f"  RMSE: {rmse:.6f}")
    print(f"  MAPE: {mape:.3f}%")
    return {"method": method_name, "mae": mae, "rmse": rmse, "mape": mape}
# ============================================================================
# PART 4: MAIN PIPELINE
# ============================================================================
```

```python
def main():
    print("=" * 70)
    print("Missing Packet Analysis & Interpolation Evaluation")
    print("=" * 70)
    print("\n[Step 1] Generating synthetic sensor data...")
    time, signal, true_signal = generate_sensor_data(n_points=100, seed=42)
    print(f"  Generated {len(time)} data points")
    print("\n[Step 2] Introducing missing packets (10% loss)...")
    time, signal, time_sparse, signal_sparse, missing_indices, mask = introduce_missing_packets(
        time, signal, missing_ratio=0.1
    )
    n_missing = len(missing_indices)
    print(f"  Missing packets: {n_missing} out of {len(time)} ({n_missing/len(time)*100:.1f}%)")
    print(f"  Missing indices: {missing_indices[:10]}{'...' if n_missing > 10 else ''}")
    print("\n[Step 3] Testing interpolation methods...")
    results = []
    # Linear
    interp_linear = linear_interpolation(time, signal_sparse, time, mask)
    results.append(evaluate_interpolation(true_signal, interp_linear, missing_indices, "Linear"))
    # Cubic spline
    interp_cubic = cubic_interpolation(time, signal_sparse, time, mask)
    results.append(evaluate_interpolation(true_signal, interp_cubic, missing_indices, "Cubic Spline"))
    # Nearest neighbor
    interp_nn = nearest_neighbor_interpolation(time, signal_sparse, time, mask)
    results.append(evaluate_interpolation(true_signal, interp_nn, missing_indices, "Nearest Neighbor"))
    # Forward fill
    interp_ff = forward_fill(time, signal_sparse, time, mask)
    results.append(evaluate_interpolation(true_signal, interp_ff, missing_indices, "Forward Fill"))
    # Polynomial
    interp_poly = polynomial_interpolation(time, signal_sparse, time, mask, degree=3)
    results.append(evaluate_interpolation(true_signal, interp_poly, missing_indices, "Polynomial (deg 3)"))
    print("\n[Step 4] Summary of Results:")
    print("-" * 70)
    df_results = pd.DataFrame(results)
    print(df_results.to_string(index=False))
    best_idx = df_results['mae'].idxmin()
```

```python
    best_idx = df_results['mae'].idxmin()
    best_method = df_results.loc[best_idx, 'method']
    print(f"\n✓ Best method: {best_method} (lowest MAE)")
    print("\n" + "=" * 70)
    print("Recommendations:")
    print("  - Use Cubic Spline for smooth signals")
    print("  - Use Linear for fast-changing signals")
    print("  - Use Forward Fill for IoT/edge devices")
    print("  - Always evaluate accuracy on your data")
    print("=" * 70)
# ============================================================================
# FIXED ENTRY POINT
# ============================================================================
if __name__ == "__main__":
    main()
```

OUTPUT:

```
======================================================================
Missing Packet Analysis & Interpolation Evaluation
======================================================================

[Step 1] Generating synthetic sensor data...
  Generated 100 data points

[Step 2] Introducing missing packets (10% loss)...
  Missing packets: 10 out of 100 (10.0%)
  Missing indices: [np.int32(3), np.int32(7), np.int32(13), np.int32(14), np.int32(20), np.int32(30), np.int32(41), np.int32(48), np.int32(56), np.int32(95)]

[Step 3] Testing interpolation methods...

Linear:
  MAE:  0.187242
  RMSE: 0.226368
  MAPE: 0.778%
```

```
Cubic Spline:
  MAE:  0.271239
  RMSE: 0.324243
  MAPE: 1.124%

Nearest Neighbor:
  MAE:  0.426884
  RMSE: 0.523606
  MAPE: 1.822%

Forward Fill:
  MAE:  0.463907
  RMSE: 0.539194
  MAPE: 1.972%

Polynomial (deg 3):
  MAE:  0.267118
  RMSE: 0.313407
  MAPE: 1.114%

[Step 4] Summary of Results:
----------------------------------------------------------------------
            method      mae     rmse     mape
            Linear 0.187242 0.226368 0.777740
      Cubic Spline 0.271239 0.324243 1.124067
  Nearest Neighbor 0.426884 0.523606 1.821511
      Forward Fill 0.463907 0.539194 1.972300
Polynomial (deg 3) 0.267118 0.313407 1.114229

✓ Best method: Linear (lowest MAE)

======================================================================
Recommendations:
  - Use Cubic Spline for smooth signals
  - Use Linear for fast-changing signals
  - Use Forward Fill for IoT/edge devices
```

OBSERVATION:

1. The code generates synthetic sensor data using a sine wave plus noise to simulate real sensor readings.
2. It randomly removes a percentage of data points to mimic missing packets and prepares the sparse dataset.
3. Several interpolation methods are applied to reconstruct the missing values, including linear, cubic, nearest neighbor, forward fill, and polynomial.
4. Each method's reconstructed values are compared with the true signal using MAE, RMSE, and MAPE to measure accuracy.
5. The code prints a summary table, identifies the best-performing interpolation method, and provides recommendations on which method to use.