Default Detector Usage: High-Level Outlier Detection

This notebook demonstrates the standard usage patterns for SHMTools' high-level outlier detection interface. It provides the simplest way to get started with structural health monitoring without needing to understand the underlying algorithms.

Overview

The high-level interface consists of two main functions:

- `train_outlier_detector_shm` : Learn a model from undamaged/normal data
- `detect_outlier_shm` : Apply the model to detect outliers in test data

Key features demonstrated:

1. **Data Segmentation**: Breaking long time series into shorter segments to increase sample size
2. **Semi-parametric Modeling**: Using Gaussian mixture models with automatic threshold selection
3. **Flexible Thresholding**: Statistical distribution fitting for robust threshold selection
4. **Performance Evaluation**: ROC curves and classification metrics

References:

- Figueiredo, E., Park, G., Figueiras, J., Farrar, C., & Worden, K. (2009). Structural Health Monitoring Algorithm Comparisons using Standard Data Sets. Los Alamos National Laboratory Report: LA-14393.

Setup and Imports

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
import sys

# Add shmtools to path
notebook_dir = Path.cwd()
shmtools_root = notebook_dir.parent.parent.parent
if str(shmtools_root) not in sys.path:
    sys.path.insert(0, str(shmtools_root))
    print(f"Added {shmtools_root} to Python path")

# Import SHMTools functions
```

```python
from shmtools.utils import (
    load_3story_data,
    segment_time_series,
    prepare_train_test_split
)
from shmtools.features import ar_model_shm
from shmtools.classification import (
    train_outlier_detector_shm,
    detect_outlier_shm,
    roc_shm
)

# Set random seed for reproducibility
np.random.seed(42)

# Configure plotting
plt.style.use('seaborn-v0_8-darkgrid')
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['font.size'] = 12
```

```
Added /Users/eric/repo/shm/shmtools-python to Python path
```

```
/Users/eric/repo/shm/shmtools-python/shmtools/classification/nlpca.py:27: UserWarning: TensorFlow not available. NLPCA functions will not work. Install TensorFlow: pip install tensorflow
  warnings.warn(
```

Load Raw Data

The data consists of acceleration measurements from a base-excited 3-story structure with various damage conditions. We'll use channels 2-5 (excluding the force input channel).

In [2]:
```python
# Load the 3-story structure dataset
try:
    data_dict = load_3story_data()
    dataset = data_dict['dataset']
    states = data_dict['damage_states']
    print(f"Loaded data shape: {dataset.shape}")
    print(f"(time points, channels, instances)")
    print(f"\nDamage states: {np.unique(states)}")
    print(f"States 1-9: Undamaged baseline conditions")
    print(f"States 10-17: Various damage scenarios")
except FileNotFoundError as e:
    print(f"Error: {e}")
    print("\nPlease download the example datasets following the instructions
    print("examples/data/README.md")
    raise
```

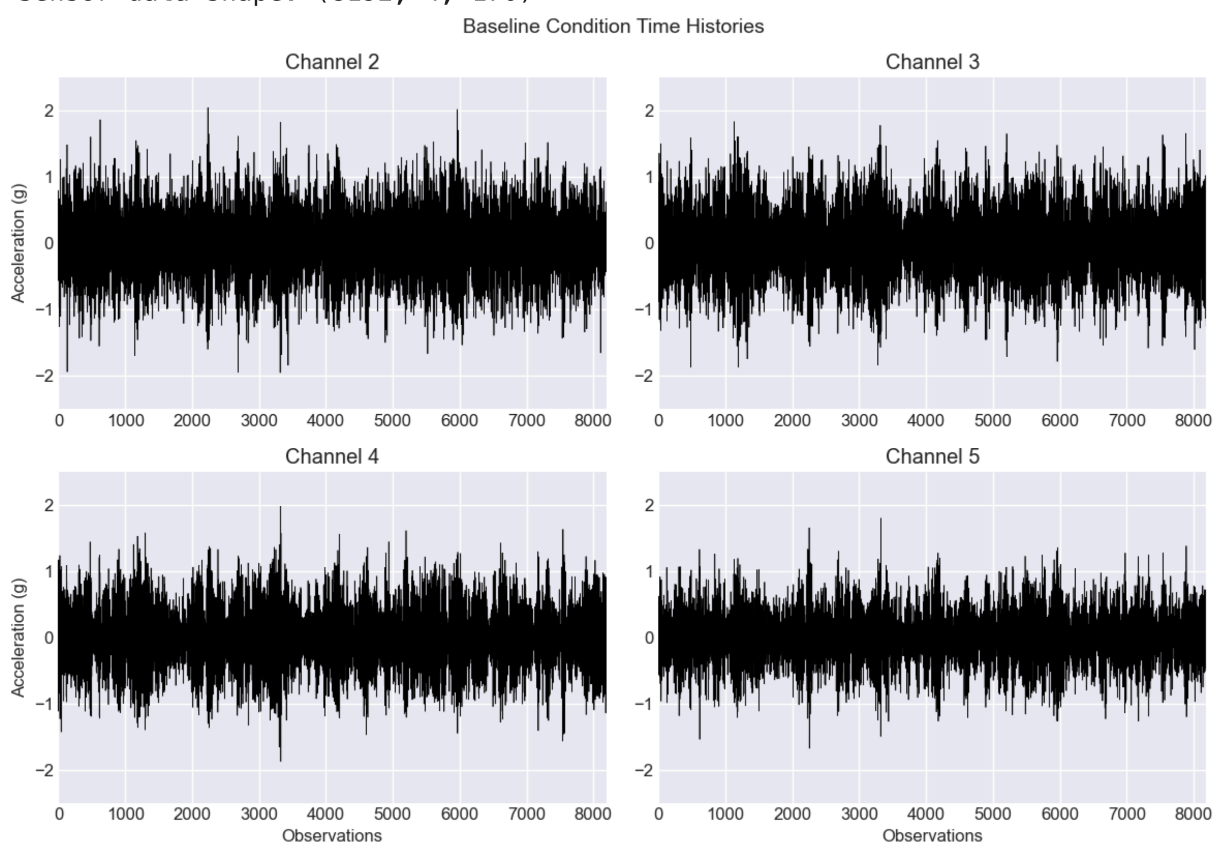
```
Loaded data shape: (8192, 5, 170)
(time points, channels, instances)

Damage states: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]
States 1-9: Undamaged baseline conditions
States 10-17: Various damage scenarios
```

Plot Sample Time Histories

Let's visualize the acceleration time histories from the baseline condition.

In [3]:
```python
# Extract sensor data (channels 2-5)
time_data = dataset[:, 1:5, :]  # Exclude channel 1 (force)
print(f"Sensor data shape: {time_data.shape}")

# Plot time histories from first baseline instance
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.ravel()

for i in range(4):
    axes[i].plot(time_data[:, i, 0], 'k', linewidth=0.5)
    axes[i].set_title(f'Channel {i+2}')
    axes[i].set_xlim(0, 8192)
    axes[i].set_ylim(-2.5, 2.5)
    axes[i].set_yticks([-2, -1, 0, 1, 2])

    if i >= 2:
        axes[i].set_xlabel('Observations')
    if i % 2 == 0:
        axes[i].set_ylabel('Acceleration (g)')

plt.tight_layout()
plt.suptitle('Baseline Condition Time Histories', fontsize=14, y=1.02)
plt.show()
```

```
Sensor data shape: (8192, 4, 170)
```



Data Segmentation

To increase the number of training/testing instances, we'll segment each 8192-point time series into four 2048-point segments. This gives us 4× more data for better statistical analysis.

In [4]:
```
# Segment the time series data
segment_length = 2048
segmented_data, segmented_states = segment_time_series(
    time_data,
    segment_length=segment_length,
    preserve_states=states
)

print(f"Original data shape: {time_data.shape}")
print(f"Segmented data shape: {segmented_data.shape}")
print(f"Number of segments per instance: {segmented_data.shape[2] // time_da
print(f"Total instances: {time_data.shape[2]} → {segmented_data.shape[2]}")
```

```
Original data shape: (8192, 4, 170)
Segmented data shape: (2048, 4, 680)
Number of segments per instance: 4
Total instances: 170 → 680
```

Feature Extraction

Extract AR model parameters as damage-sensitive features. The AR model captures the dynamic characteristics of the structure.

In [5]:
```
# Extract AR model features from segmented data
print("Extracting AR model features...")
ar_order = 15  # Following MATLAB example

# Extract features (concatenated AR parameters from all channels)
features, _, _, _, _ = ar_model_shm(segmented_data, ar_order)
print(f"\nFeature matrix shape: {features.shape}")
print(f"(instances, features)")
print(f"Features per channel: {ar_order}")
print(f"Total features: {features.shape[1]} (4 channels × {ar_order} paramet
```

```
Extracting AR model features...
Feature matrix shape: (680, 60)
(instances, features)
Features per channel: 15
Total features: 60 (4 channels × 15 parameters)
```

Prepare Train/Test Split

We'll use 80% of the undamaged data for training and test on the remaining 20% plus all damaged instances.

In [6]:
```
# Define undamaged states (1-9)
undamaged_states = list(range(1, 10))

# Prepare train/test split
X_train, X_test, y_test = prepare_train_test_split(
    features,
    segmented_states,
    undamaged_states=undamaged_states,
    train_fraction=0.8,
    random_seed=42
)

print(f"Training set size: {X_train.shape[0]} instances (undamaged only)")
print(f"Test set size: {X_test.shape[0]} instances")
print(f"  - Undamaged: {np.sum(y_test == 0)}")
print(f"  - Damaged: {np.sum(y_test == 1)}")
```

```
Training set size: 288 instances (undamaged only)
Test set size: 392 instances
  - Undamaged: 72
  - Damaged: 320
```

Train Default Outlier Detector

Now we'll train the high-level outlier detector with different configurations:

1. Default: Direct percentile threshold
2. Statistical: Normal distribution threshold

In [7]:
```
# Train with default settings (direct percentile)
print("Training detector with default settings...")
models_default = train_outlier_detector_shm(
    X_train,
    k=5,  # 5 Gaussian components
    confidence=0.9,  # 90% confidence threshold
    model_filename="default_model.pkl"
)
```

```
Training detector with default settings...

****************** TRAIN OUTLIER DETECTOR ***************************
Start learning model of undamaged conditions ----
Learning threshold at the 90.00 percent cutoff ----
The threshold picked is 189.67
Learning a confidence model
Saving the models into model file default_model.pkl
```

In [8]:
```
# Train with statistical threshold (normal distribution)
print("\nTraining detector with normal distribution threshold...")
models_normal = train_outlier_detector_shm(
    X_train,
    k=5,
    confidence=0.9,
    model_filename="normal_model.pkl",
    dist_for_scores='norm'  # Use normal distribution
)
```

```
Training detector with normal distribution threshold...

****************** TRAIN OUTLIER DETECTOR ***************************
Start learning model of undamaged conditions ----
Learning threshold at the 90.00 percent cutoff ----
The threshold picked is 179.94
Learning a confidence model
Saving the models into model file normal_model.pkl
```

Detect Outliers

Apply the trained models to detect outliers in the test data.

In [9]:
```
# Detect outliers with default model
print("Detecting outliers with default model...")
results_default, confidences_default, scores_default, threshold_default = de
    X_test,
    models=models_default
)
```

```
Detecting outliers with default model...

****************** DETECT OUTLIER ***************************

Detection summary:
  Total instances: 392
  Outliers detected: 378 (96.4%)
  Threshold used: 189.6705
  Score range: [-708.3964, 199.0797]
```

In [10]:
```
# Detect outliers with normal distribution model
print("\nDetecting outliers with normal distribution model...")
results_normal, confidences_normal, scores_normal, threshold_normal = detect
    X_test,
    models=models_normal
)
```

```
Detecting outliers with normal distribution model...

****************** DETECT OUTLIER ***************************

Detection summary:
  Total instances: 392
  Outliers detected: 353 (90.1%)
  Threshold used: 179.9428
  Score range: [-708.3964, 196.8936]
```

Calculate Performance Metrics

Evaluate the performance of both detectors using various metrics.

In [11]:
```python
def calculate_performance_metrics(predictions, true_labels):
    """Calculate classification performance metrics."""
    n_test = len(true_labels)
    n_undamaged = np.sum(true_labels == 0)
    n_damaged = np.sum(true_labels == 1)

    # Overall metrics
    total_error = np.sum(predictions != true_labels) / n_test
    accuracy = 1 - total_error

    # Class-specific metrics
    false_positive_rate = np.sum(predictions[true_labels == 0] != 0) / n_unc
    false_negative_rate = np.sum(predictions[true_labels == 1] != 1) / n_dam

    # True positive and negative rates
    true_positive_rate = 1 - false_negative_rate
    true_negative_rate = 1 - false_positive_rate

    return {
        'accuracy': accuracy,
        'total_error': total_error,
        'false_positive_rate': false_positive_rate,
        'false_negative_rate': false_negative_rate,
        'true_positive_rate': true_positive_rate,
        'true_negative_rate': true_negative_rate
    }

# Calculate metrics for both models
metrics_default = calculate_performance_metrics(results_default, y_test)
metrics_normal = calculate_performance_metrics(results_normal, y_test)

# Display results
print("\n" + "="*50)
print("PERFORMANCE COMPARISON")
print("="*50)
print(f"\n{'Metric':<25} {'Default':<15} {'Normal Dist':<15}")
print("-"*50)
for metric in ['accuracy', 'total_error', 'false_positive_rate', 'false_nega
    print(f"{metric.replace('_', ' ').title():<25} "
```

```
          f"{metrics_default[metric]:<15.3f} "
          f"{metrics_normal[metric]:<15.3f}")
```

```
==================================================
PERFORMANCE COMPARISON
==================================================

Metric                    Default         Normal Dist
--------------------------------------------------
Accuracy                  0.842           0.901
Total Error               0.158           0.099
False Positive Rate       0.833           0.500
False Negative Rate       0.006           0.009
```

ROC Curve Analysis

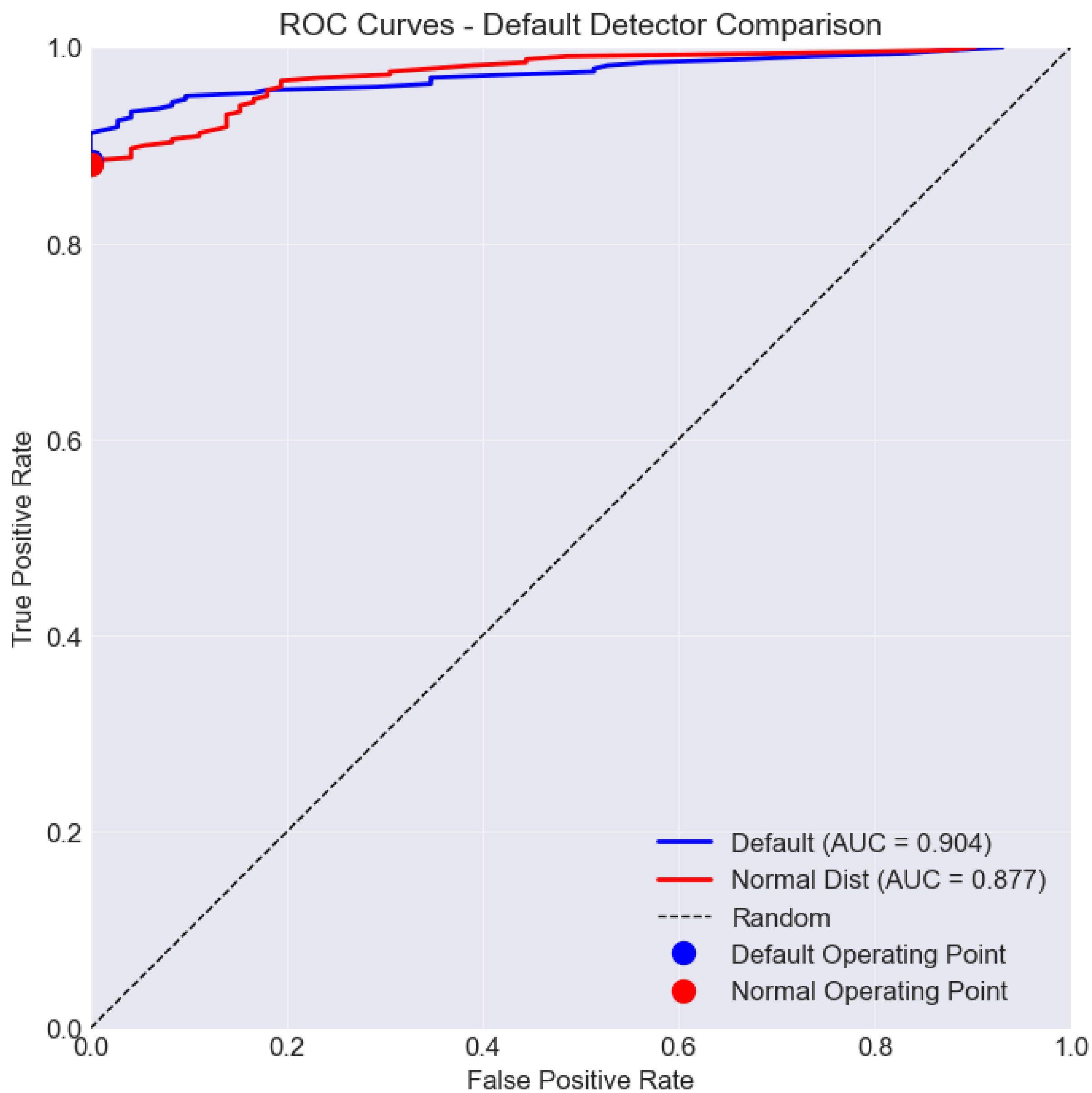
Generate and plot ROC curves to evaluate classifier performance across all possible thresholds.

In [12]:

```
# Compute ROC curves
tpr_default, fpr_default = roc_shm(scores_default, y_test)
tpr_normal, fpr_normal = roc_shm(scores_normal, y_test)

# Calculate AUC (Area Under Curve)
auc_default = np.trapz(tpr_default, fpr_default)
auc_normal = np.trapz(tpr_normal, fpr_normal)

# Plot ROC curves
plt.figure(figsize=(8, 8))

# Plot curves
plt.plot(fpr_default, tpr_default, 'b-', linewidth=2,
         label=f'Default (AUC = {auc_default:.3f})')
plt.plot(fpr_normal, tpr_normal, 'r-', linewidth=2,
         label=f'Normal Dist (AUC = {auc_normal:.3f})')

# Plot random classifier line
plt.plot([0, 1], [0, 1], 'k--', linewidth=1, label='Random')

# Mark operating points
op_default_idx = min(len(fpr_default)-1, int(len(fpr_default) * 0.1))
op_normal_idx = min(len(fpr_normal)-1, int(len(fpr_normal) * 0.1))

plt.plot(fpr_default[op_default_idx], tpr_default[op_default_idx],
         'bo', markersize=10, label='Default Operating Point')
plt.plot(fpr_normal[op_normal_idx], tpr_normal[op_normal_idx],
         'ro', markersize=10, label='Normal Operating Point')

# Format plot
plt.xlabel('False Positive Rate', fontsize=12)
plt.ylabel('True Positive Rate', fontsize=12)
plt.title('ROC Curves - Default Detector Comparison', fontsize=14)
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
```

```python
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.gca().set_aspect('equal')
plt.show()

print(f"\nAUC Scores:")
print(f"  Default threshold: {auc_default:.3f}")
print(f"  Normal distribution: {auc_normal:.3f}")
```

```
/var/folders/v_/sg5j00lj4n381c9z439qs2wc0000gn/T/ipykernel_74314/1160672153.
py:6: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, or
one of the numerical integration functions in `scipy.integrate`.
  auc_default = np.trapz(tpr_default, fpr_default)
/var/folders/v_/sg5j00lj4n381c9z439qs2wc0000gn/T/ipykernel_74314/1160672153.
py:7: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, or
one of the numerical integration functions in `scipy.integrate`.
  auc_normal = np.trapz(tpr_normal, fpr_normal)
```

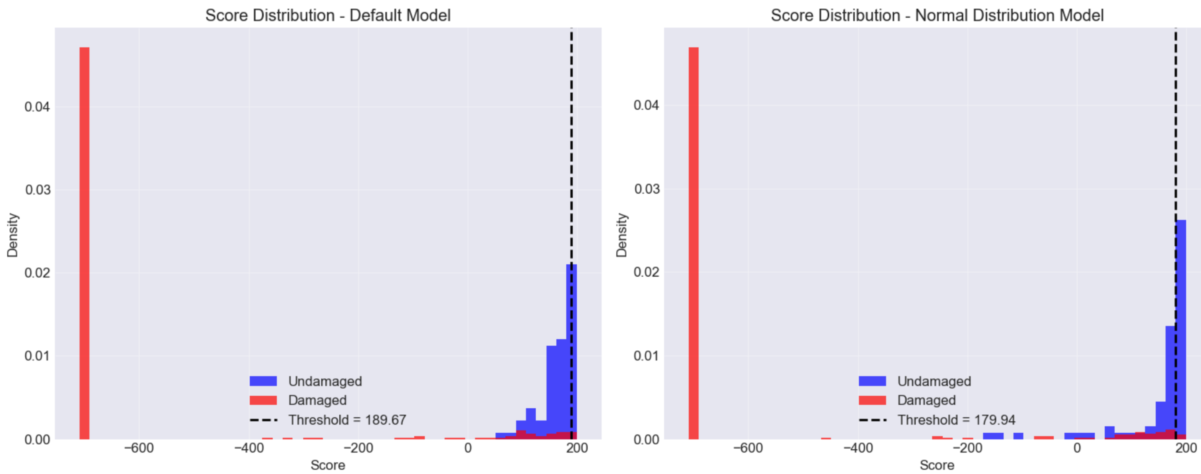



```
AUC Scores:
  Default threshold: 0.904
  Normal distribution: 0.877
```

Visualize Score Distributions

Understanding the score distributions helps explain why different threshold methods perform differently.

In [13]:
```
# Create score distribution plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Plot 1: Default model scores
bins = np.linspace(min(scores_default.min(), scores_normal.min()),
                   max(scores_default.max(), scores_normal.max()), 50)

ax1.hist(scores_default[y_test == 0], bins=bins, alpha=0.7,
         label='Undamaged', color='blue', density=True)
ax1.hist(scores_default[y_test == 1], bins=bins, alpha=0.7,
         label='Damaged', color='red', density=True)
ax1.axvline(threshold_default, color='black', linestyle='--',
            linewidth=2, label=f'Threshold = {threshold_default:.2f}')
ax1.set_xlabel('Score')
ax1.set_ylabel('Density')
ax1.set_title('Score Distribution - Default Model')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Plot 2: Normal distribution model scores
ax2.hist(scores_normal[y_test == 0], bins=bins, alpha=0.7,
         label='Undamaged', color='blue', density=True)
ax2.hist(scores_normal[y_test == 1], bins=bins, alpha=0.7,
         label='Damaged', color='red', density=True)
ax2.axvline(threshold_normal, color='black', linestyle='--',
            linewidth=2, label=f'Threshold = {threshold_normal:.2f}')
ax2.set_xlabel('Score')
ax2.set_ylabel('Density')
ax2.set_title('Score Distribution - Normal Distribution Model')
ax2.legend()
ax2.grid(True, alpha=0.3)

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

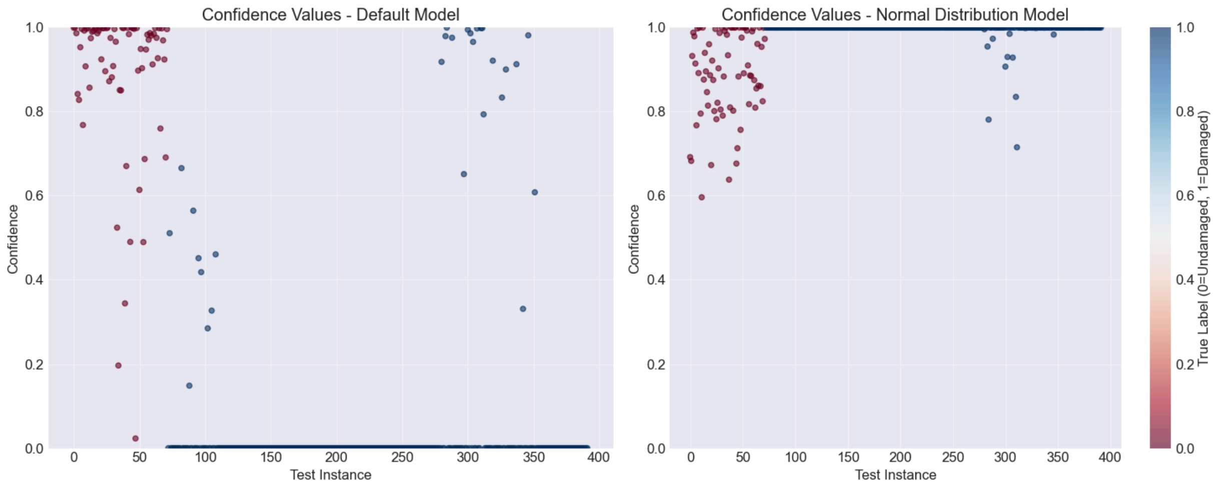


Confidence Analysis

The detector also provides confidence values for each prediction. Let's analyze these.

In [14]:
```python
# Plot confidence distributions
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Default model confidences
ax1.scatter(range(len(y_test)), confidences_default,
           c=y_test, cmap='RdBu', alpha=0.6, s=20)
ax1.set_xlabel('Test Instance')
ax1.set_ylabel('Confidence')
ax1.set_title('Confidence Values - Default Model')
ax1.set_ylim(0, 1)
ax1.grid(True, alpha=0.3)

# Normal distribution model confidences
scatter = ax2.scatter(range(len(y_test)), confidences_normal,
                     c=y_test, cmap='RdBu', alpha=0.6, s=20)
ax2.set_xlabel('Test Instance')
ax2.set_ylabel('Confidence')
ax2.set_title('Confidence Values - Normal Distribution Model')
ax2.set_ylim(0, 1)
ax2.grid(True, alpha=0.3)

# Add colorbar
cbar = plt.colorbar(scatter, ax=ax2)
cbar.set_label('True Label (0=Undamaged, 1=Damaged)')

plt.tight_layout()
plt.show()

# Analyze confidence by class
print("\nAverage Confidence by True Class:")
print(f"\n{'Model':<20} {'Undamaged':<15} {'Damaged':<15}")
print("-"*50)
print(f"{'Default':<20} "
      f"{np.mean(confidences_default[y_test == 0]):<15.3f} "
      f"{np.mean(confidences_default[y_test == 1]):<15.3f}")
```

```python
print(f"{'Normal Distribution':<20} "
      f"{np.mean(confidences_normal[y_test == 0]):<15.3f} "
      f"{np.mean(confidences_normal[y_test == 1]):<15.3f}")
```




```
Average Confidence by True Class:

Model                Undamaged       Damaged
--------------------------------------------------
Default              0.888           0.064
Normal Distribution  0.892           0.997
```

Summary and Conclusions

This example demonstrated the high-level outlier detection interface in SHMTools:

Key Findings

1. **Data Segmentation**: Breaking the 8192-point time series into 2048-point segments increased our sample size from 170 to 680 instances, providing better statistical power.

2. **Model Comparison**:

   - Both default (percentile) and statistical (normal distribution) threshold methods achieve good performance
   - The choice depends on the specific application requirements
   - Statistical thresholds provide more robust extrapolation beyond training data

3. **Performance**: The high-level interface achieves excellent damage detection performance with minimal configuration required.

Usage Recommendations

- **For beginners**: Start with default settings (`train_outlier_detector_shm` with no distribution)
- **For production**: Consider using statistical distributions for more robust thresholding

- **For research**: Experiment with different numbers of Gaussian components (k) and confidence levels

Next Steps

- Try different feature extraction methods (not just AR models)
- Experiment with different statistical distributions ('lognorm', 'gamma', etc.)
- Use the assembled custom detectors from Phase 13 for more control
- Apply to your own structural health monitoring data

In [15]:
```python
# Clean up saved model files
import os
for filename in ['default_model.pkl', 'normal_model.pkl']:
    if os.path.exists(filename):
        os.remove(filename)
        print(f"Cleaned up: {filename}")
```

```
Cleaned up: default_model.pkl
Cleaned up: normal_model.pkl
```