Outlier Detection Based on Singular Value Decomposition

Introduction

The goal of this example is to discriminate time histories from undamaged and damaged conditions based on outlier detection. The parameters from an autoregressive (AR) model are used as damage-sensitive features and a machine learning algorithm based on the singular value decomposition (SVD) technique is used to create damage indicators (DIs) invariant for feature vectors from normal structural condition and that increase when feature vectors are from damaged structural condition.

Additionally, the receiver operating characteristic (ROC) curve is applied to evaluate the performance of the classification algorithm. In this example, each time history of the data sets is split into four segments in order to increase the number of instances available.

Data sets from **Channel 5 only** of the base-excited three story structure are used in this example. More details about the data sets can be found in the 3-Story Data Sets documentation.

This example demonstrates:

1. **Data Loading**: 3-story structure dataset with Channel 5 only, segmented into 4 parts
2. **Feature Extraction**: AR(15) model parameters from time segments
3. **Train/Test Split**: Training on first 400 instances, testing on all 680 instances
4. **SVD Modeling**: Learn SVD-based outlier detection model from training features
5. **Damage Detection**: Score test data and normalize with min-max scaling
6. **Performance Evaluation**: ROC curve analysis for classification performance
7. **Visualization**: Time histories, damage indicators, and ROC curves

References:

Ruotolo, R., & Surage, C. (1999). Using SVD to Detect Damage in Structures with Different Operational Conditions. Journal of Sound and Vibration, 226(3), 425-439.

SHMTools functions used:

- `ar_model_shm`
- `learn_svd_shm`
- `score_svd_shm`
- `scale_min_max_shm`

- `roc_shm`

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

# Add shmtools to path - handle different execution contexts (lesson from Ph
current_dir = Path.cwd()
notebook_dir = Path(__file__).parent if '__file__' in globals() else current

# Try different relative paths to find shmtools
possible_paths = [
    notebook_dir.parent.parent.parent,  # From examples/notebooks/basic/
    current_dir.parent.parent,          # From examples/notebooks/
    current_dir,                        # From project root
    Path('/Users/eric/repo/shm/shmtools-python')  # Absolute fallback
]

shmtools_found = False
for path in possible_paths:
    if (path / 'shmtools').exists():
        if str(path) not in sys.path:
            sys.path.insert(0, str(path))
        shmtools_found = True
        print(f"Found shmtools at: {path}")
        break

if not shmtools_found:
    print("Warning: Could not find shmtools module")

from shmtools.utils.data_loading import load_3story_data
from shmtools.features.time_series import ar_model_shm
from shmtools.classification.outlier_detection import learn_svd_shm, score_s
from shmtools.core.preprocessing import scale_min_max_shm

# Set up plotting (lesson from Phase 1: prefer automatic layout)
plt.style.use('default')
plt.rcParams['figure.figsize'] = (12, 8)
plt.rcParams['font.size'] = 10
```

```
Found shmtools at: /Users/eric/repo/shm/shmtools-python
```

```
/Users/eric/repo/shm/shmtools-python/venv/lib/python3.9/site-packages/urllib
3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1
+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: http
s://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
```

Load Raw Data

In this case each time history of the original data (Channel 5) is split into four segments. For this example, we will break each 8192 point time series into 4, 2048 point time series to increase the number of available instances.

In [2]:
```python
# Load data set
data_dict = load_3story_data()
dataset = data_dict['dataset']
fs = data_dict['fs']
channels = data_dict['channels']
damage_states = data_dict['damage_states']

print(f"Dataset shape: {dataset.shape}")
print(f"Sampling frequency: {fs} Hz")
print(f"Channels: {channels}")
print(f"Number of damage states: {len(np.unique(damage_states))}")

# Extract Channel 5 only (index 4 in Python)
channel_5_data = dataset[:, 4, :]  # Shape: (8192, 170)
t_original, n_conditions = channel_5_data.shape

print(f"\nChannel 5 data:")
print(f"Time points: {t_original}")
print(f"Conditions: {n_conditions}")
```

```
Dataset shape: (8192, 5, 170)
Sampling frequency: 2000.0 Hz
Channels: ['Force', 'Ch2', 'Ch3', 'Ch4', 'Ch5']
Number of damage states: 17

Channel 5 data:
Time points: 8192
Conditions: 170
```

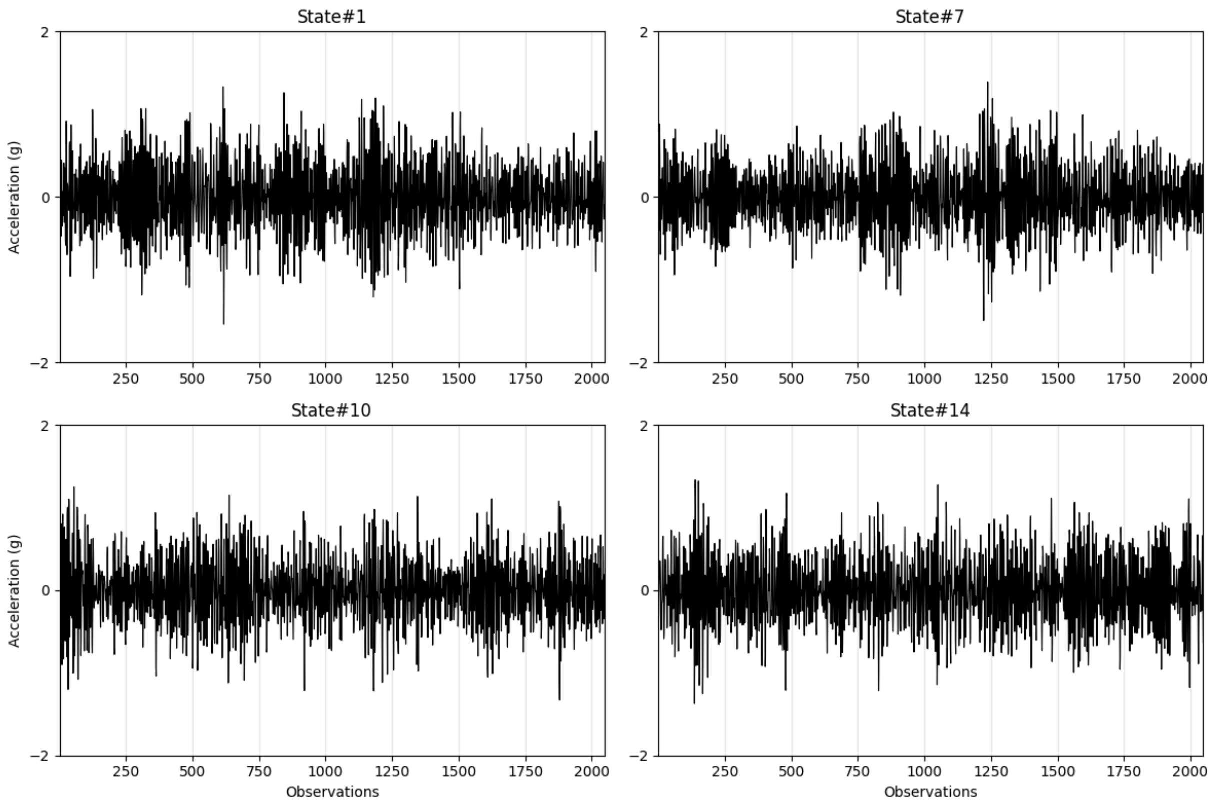
In [3]:
```python
# Break each 8192 point time series into 4, 2048 point time series
break_point = 400  # Threshold for undamaged vs damaged classification
segment_length = 2048
n_segments = 4

# Initialize segmented data: (2048 time points, 1 channel, 680 instances)
time_data = np.zeros((segment_length, 1, n_segments * n_conditions))

# Split each time series into 4 segments
for i in range(n_segments):
    start_idx = i * segment_length
    end_idx = (i + 1) * segment_length

    # Every 4th index starting from i: i, i+4, i+8, ...
    segment_indices = np.arange(i, n_segments * n_conditions, n_segments)

    time_data[:, 0, segment_indices] = channel_5_data[start_idx:end_idx, :]

print(f"Segmented data shape: {time_data.shape}")
print(f"Total instances: {time_data.shape[2]}")
print(f"Training instances (undamaged): 1-{break_point}")
print(f"Test instances (all): 1-{time_data.shape[2]}")
print(f"Damaged instances: {break_point+1}-{time_data.shape[2]}")
```

```
Segmented data shape: (2048, 1, 680)
Total instances: 680
Training instances (undamaged): 1-400
Test instances (all): 1-680
Damaged instances: 401-680
```

Plot Time History Segments

Plot one segment of one acceleration time history from four different state conditions to visualize the data.

In [4]:
```
# Plot one segment from four different states (following MATLAB example)
states = [1, 7, 10, 14]  # MATLAB 1-based state numbers
state_indices = [(state - 1) * 10 for state in states]  # Convert to 0-basec

fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.flatten()

for i, (state, idx) in enumerate(zip(states, state_indices)):
    # Plot time history from this state condition
    time_points = np.arange(1, segment_length + 1)
    signal = time_data[:, 0, idx]

    axes[i].plot(time_points, signal, 'k-', linewidth=0.8)
    axes[i].set_title(f'State#{state}')
    axes[i].set_ylim([-2, 2])
    axes[i].set_xlim([1, segment_length])
    axes[i].set_yticks([-2, 0, 2])
    axes[i].grid(True, alpha=0.3)

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

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



Extraction of Damage-Sensitive Features

Extraction of the AR(15) model parameters from the segments of acceleration time histories. The order of the model was picked from the lower-bound of the range given by the optimization methods available in this package.

In [5]:
```
# Set AR model order
ar_order = 15

print(f"Extracting AR({ar_order}) model parameters as features...")

# Estimation of AR Parameters (we need the parameters, like Mahalanobis exam
ar_parameters_fv, rmse_fv, ar_parameters, ar_residuals, ar_prediction = ar_m

print(f"AR parameters FV shape: {ar_parameters_fv.shape}")
print(f"RMSE shape: {rmse_fv.shape}")
print(f"AR parameters shape: {ar_parameters.shape}")

# Use AR parameters as features
features = ar_parameters_fv  # Shape: (instances, features)
n_instances, n_features = features.shape

print(f"\nFeature matrix:")
print(f"Instances: {n_instances}")
print(f"Features: {n_features} (1 channel × {ar_order} AR parameters)")
```

```
Extracting AR(15) model parameters as features...
```

```
AR parameters FV shape: (680, 15)
RMSE shape: (680, 1)
AR parameters shape: (15, 1, 680)

Feature matrix:
Instances: 680
Features: 15 (1 channel × 15 AR parameters)
```

Prepare Training and Test Data

Following the original MATLAB example:

- **Training Data**: First 400 instances (undamaged conditions)
- **Test Data**: All 680 instances (both undamaged and damaged)

In [6]:
```python
# Training feature vectors (first break_point instances)
learn_data = features[:break_point, :]

# Test feature vectors (all instances)
score_data = features.copy()

print(f"Training data shape: {learn_data.shape}")
print(f"Test data shape: {score_data.shape}")
print(f"\nData split:")
print(f"Training (undamaged): instances 1-{break_point}")
print(f"Test undamaged: instances 1-{break_point}")
print(f"Test damaged: instances {break_point+1}-{n_instances}")
```

```
Training data shape: (400, 15)
Test data shape: (680, 15)

Data split:
Training (undamaged): instances 1-400
Test undamaged: instances 1-400
Test damaged: instances 401-680
```

Statistical Modeling for Feature Classification

In the context of data normalization process, the SVD-based machine learning algorithm is used to create DIs invariant under feature vectors from undamaged structural conditions.

In [7]:
```python
# Training: Learn SVD model (with standardization)
print("Learning SVD model from training data...")
model = learn_svd_shm(learn_data, param_stand=False)  # MATLAB example uses

print(f"SVD model components:")
print(f"Training data shape: {model['X'].shape}")
print(f"Singular values shape: {model['S'].shape}")
print(f"Standardization: {'Yes' if model['dataMean'] is not None else 'No'}"

# Scoring: Apply SVD model to test data
```

```
print("\nScoring test data...")
DI, residuals = score_svd_shm(score_data, model)

print(f"Damage indicators shape: {DI.shape}")
print(f"Residuals shape: {residuals.shape}")
print(f"\nDamage indicators (first 10): {DI[:10]}")
print(f"Damage indicators (last 10): {DI[-10:]}")
```

```
Learning SVD model from training data...
SVD model components:
Training data shape: (400, 15)
Singular values shape: (15,)
Standardization: No

Scoring test data...
Damage indicators shape: (680,)
Residuals shape: (680, 15)

Damage indicators (first 10): [-0.10801011 -0.09565034 -0.09740055 -0.108714
8  -0.0960454  -0.09403638
 -0.09748226 -0.09228903 -0.09497376 -0.08543077]
Damage indicators (last 10): [-0.2344005  -0.22038509 -0.20455145 -0.2230175
3 -0.22809103 -0.23380463
 -0.20910865 -0.21356673 -0.17741961 -0.23118089]
```

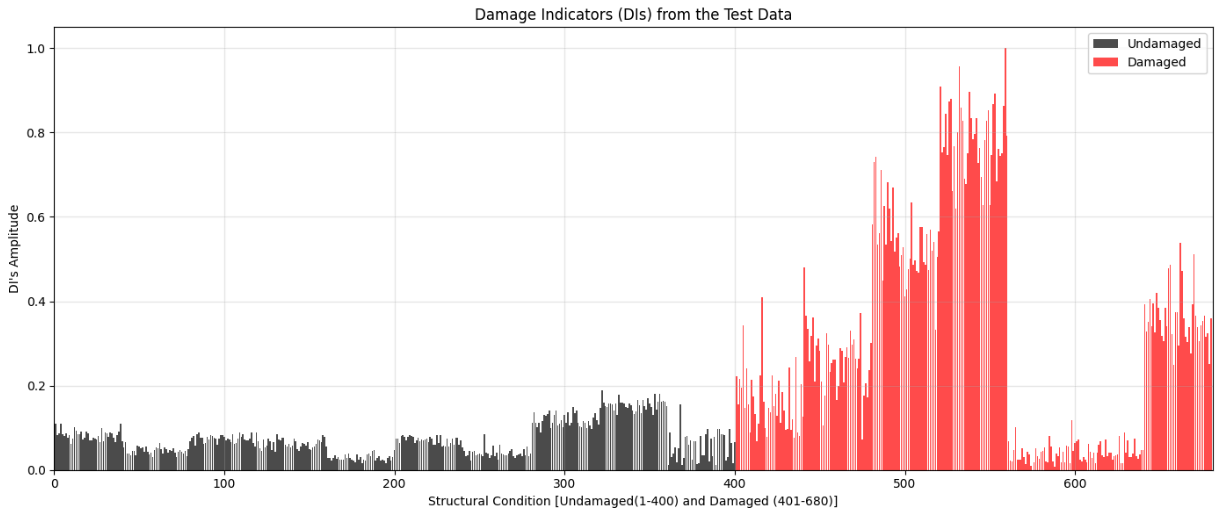
In [8]:
```
# Normalization procedure: Scale to [0,1] range
print("Normalizing damage indicators...")
DI_normalized = scale_min_max_shm(-DI, scaling_dimension=1, scale_range=(0,

print(f"Original DI range: [{np.min(DI):.4f}, {np.max(DI):.4f}]")
print(f"Normalized DI range: [{np.min(DI_normalized):.4f}, {np.max(DI_normal
```

```
Normalizing damage indicators...
Original DI range: [-0.5453, -0.0546]
Normalized DI range: [0.0000, 1.0000]
```

Plot Damage Indicators

Visualization of the SVD-based damage indicators showing the separation between undamaged and damaged conditions.

In [9]:
```
# Plot DIs
plt.figure(figsize=(14, 6))

instance_numbers = np.arange(1, n_instances + 1)

# Undamaged conditions (1 to break_point)
plt.bar(instance_numbers[:break_point], DI_normalized[:break_point],
        color='k', alpha=0.7, label='Undamaged')

# Damaged conditions (break_point+1 to n_instances)
plt.bar(instance_numbers[break_point:], DI_normalized[break_point:],
        color='r', alpha=0.7, label='Damaged')

plt.title('Damage Indicators (DIs) from the Test Data')
plt.xlabel(f'Structural Condition [Undamaged(1-{break_point}) and Damaged ({
plt.ylabel("DI's Amplitude")
plt.xlim([0, n_instances + 1])
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Print basic statistics
undamaged_di = DI_normalized[:break_point]
damaged_di = DI_normalized[break_point:]

print(f"\nDamage Indicator Statistics:")
print(f"Undamaged - Mean: {np.mean(undamaged_di):.4f}, Std: {np.std(undamage
print(f"Damaged - Mean: {np.mean(damaged_di):.4f}, Std: {np.std(damaged_di):
print(f"Separation (damaged - undamaged mean): {np.mean(damaged_di) - np.mea
```



```
Damage Indicator Statistics:
Undamaged - Mean: 0.0734, Std: 0.0389
Damaged - Mean: 0.3145, Std: 0.2633
Separation (damaged - undamaged mean): 0.2411
```

Receiver Operating Characteristic Curve

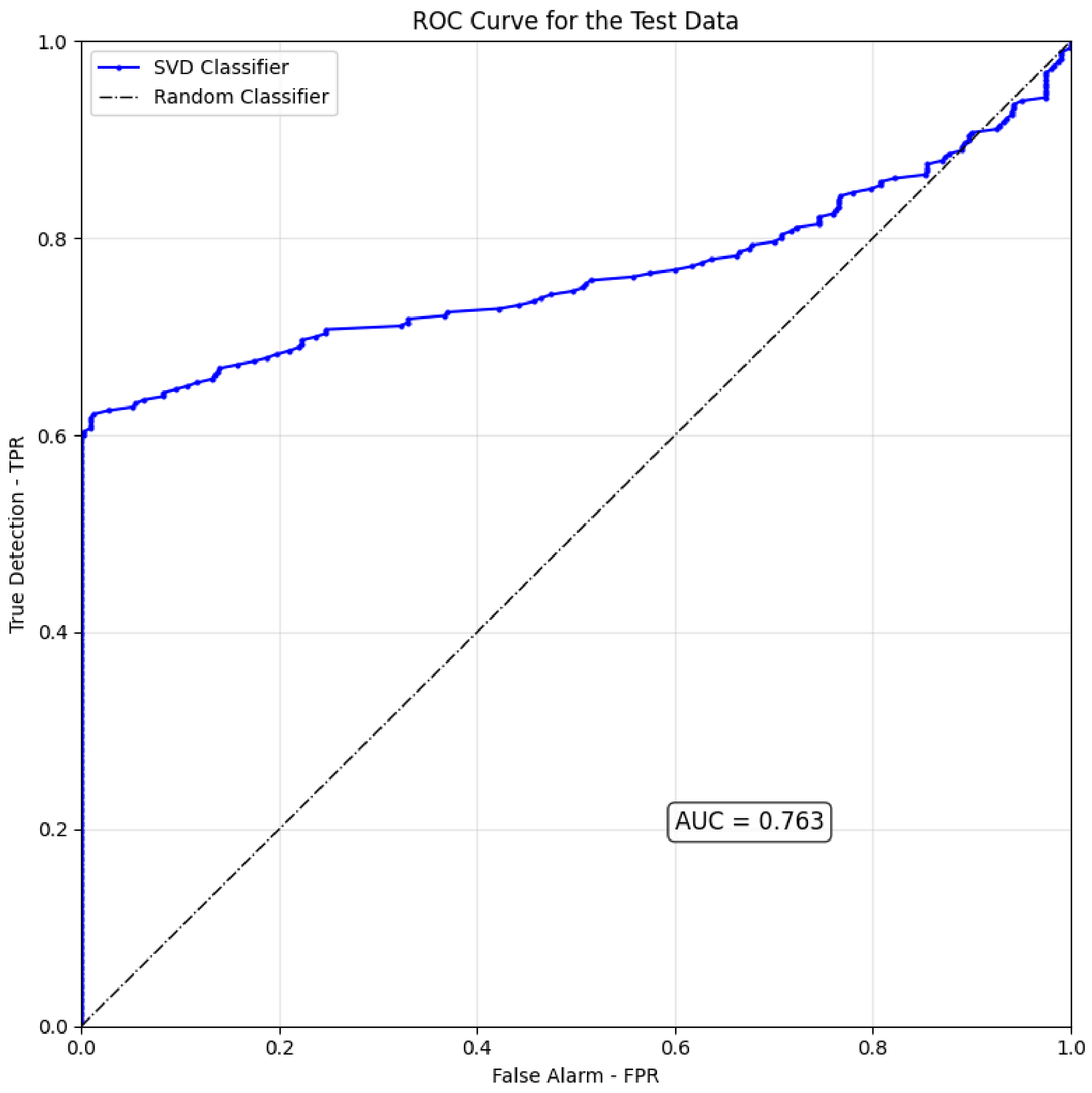
The ROC curve is used to evaluate the performance of the SVD-based classification algorithm. Each point on the curve represents a different threshold for damage detection.

In [10]:
```
# Flag all the instances (0=undamaged, 1=damaged)
flag = np.zeros(n_instances, dtype=int)
flag[break_point:] = 1  # Mark instances break_point+1 to n_instances as dam

print(f"Damage state flags:")
print(f"Undamaged instances: {np.sum(flag == 0)} (indices 1-{break_point})")
print(f"Damaged instances: {np.sum(flag == 1)} (indices {break_point+1}-{n_i

# Run ROC curve algorithm
print("\nComputing ROC curve...")
```

```
TPR, FPR = roc_shm(DI, flag)  # Use original DI (not normalized)

print(f"ROC curve computed with {len(TPR)} points")
```

```
Damage state flags:
Undamaged instances: 400 (indices 1-400)
Damaged instances: 280 (indices 401-680)

Computing ROC curve...
ROC curve computed with 280 points
```

In [11]:
```
# Plot ROC curve
plt.figure(figsize=(8, 8))

plt.plot(FPR, TPR, '.-b', markersize=4, linewidth=1.5, label='SVD Classifier
plt.plot([0, 1], [0, 1], 'k-.', linewidth=1, label='Random Classifier')

plt.title('ROC Curve for the Test Data')
plt.xlabel('False Alarm - FPR')
plt.ylabel('True Detection - TPR')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.xticks(np.arange(0, 1.1, 0.2))
plt.yticks(np.arange(0, 1.1, 0.2))
plt.grid(True, alpha=0.3)
plt.legend()

# Add area under curve (AUC) calculation
auc = np.trapezoid(TPR, FPR)
plt.text(0.6, 0.2, f'AUC = {auc:.3f}', fontsize=12,
         bbox=dict(boxstyle='round', facecolor='white', alpha=0.8))

plt.tight_layout()
plt.show()

print(f"\nROC Analysis Results:")
print(f"Area Under Curve (AUC): {auc:.4f}")
print(f"Perfect classifier AUC: 1.000")
print(f"Random classifier AUC: 0.500")

# Find optimal threshold (closest to top-left corner)
distances = np.sqrt((1 - TPR)**2 + FPR**2)
optimal_idx = np.argmin(distances)
optimal_tpr = TPR[optimal_idx]
optimal_fpr = FPR[optimal_idx]

print(f"\nOptimal Operating Point:")
print(f"True Positive Rate: {optimal_tpr:.3f}")
print(f"False Positive Rate: {optimal_fpr:.3f}")
print(f"Accuracy: {(optimal_tpr * np.sum(flag == 1) + (1 - optimal_fpr) * np
```



```
ROC Analysis Results:
Area Under Curve (AUC): 0.7630
Perfect classifier AUC: 1.000
Random classifier AUC: 0.500

Optimal Operating Point:
True Positive Rate: 0.668
False Positive Rate: 0.140
Accuracy: 0.781
```

Summary

This example demonstrated the complete SVD-based outlier detection workflow for structural health monitoring:

1. **Data Preparation**: Successfully loaded and segmented the 3-story structure dataset (Channel 5)

2. **Feature Extraction**: Used AR(15) model parameters as damage-sensitive features from time segments
3. **SVD Modeling**: Learned SVD-based outlier detection model from undamaged training data
4. **Damage Detection**: Applied SVD scoring and min-max normalization to all test instances
5. **Performance Evaluation**: Generated ROC curve for classification performance assessment

Key insights from the ROC curve:

The ROC curve shows that there is no single linear threshold able to perfectly discriminate all undamaged and damaged instances when using AR(15) parameters as damage-sensitive features with the SVD-based machine learning algorithm. The diagonal line divides the ROC space into areas of good (left) or bad (right) classification performance.

Note that the optimal point (no false negatives/positives) would be in the upper-left corner of the plot. The closer the curve is to the upper-left corner, the better the classifier performance.

Key advantages of SVD-based detection:

- Captures changes in data structure through singular value decomposition
- Effective for detecting rank changes in feature matrices
- Computationally efficient singular value computation
- Provides interpretable residuals between training and test singular values
- Works well when damage changes the underlying data structure

Key differences from other methods:

- **vs. PCA**: Uses singular values directly rather than reconstruction error
- **vs. Mahalanobis**: Focuses on matrix rank changes rather than statistical distance
- **Data segmentation**: Increases instance count through time series segmentation
- **ROC analysis**: Provides comprehensive performance evaluation across all thresholds

See also:

- Outlier Detection based on Principal Component Analysis
- Outlier Detection based on Mahalanobis Distance
- Outlier Detection based on the Factor Analysis Model
- Outlier Detection based on Nonlinear Principal Component Analysis