Custom Detector Assembly

This notebook demonstrates how to assemble custom outlier detectors by mixing and matching learning/scoring function pairs from different detector categories. The custom detector assembly framework allows you to:

1. **Parametric Detectors**: PCA, Mahalanobis, SVD, Factor Analysis
2. **Non-parametric Detectors**: Kernel density estimation with various kernels
3. **Semi-parametric Detectors**: Gaussian Mixture Models with partitioning algorithms

Overview

The `assemble_outlier_detector_shm` function provides an interactive framework for creating custom detectors. You can:

- Select from pre-built detector combinations
- Configure parameters for each detector type
- Generate custom training functions that work with the universal `detect_outlier_shm` interface
- Save and load detector configurations for reproducibility

Setup and Imports

In [1]:
```
import sys
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
from typing import Dict, Any, List

# Add shmtools to path if needed
notebook_dir = Path.cwd()
if 'shmtools-python' in str(notebook_dir):
    project_root = notebook_dir
    while project_root.name != 'shmtools-python' and project_root.parent !=
        project_root = project_root.parent
else:
    # Try common paths
    possible_paths = [
        notebook_dir.parent.parent.parent,  # From examples/notebooks/advanc
        notebook_dir.parent.parent,          # From examples/notebooks/
        notebook_dir,                        # From project root
        Path('/Users/eric/repo/shm/shmtools-python')  # Absolute fallback
    ]

    project_root = None
    for path in possible_paths:
```

```python
        if (path / 'shmtools').exists():
            project_root = path
            break

    if project_root is None:
        raise RuntimeError("Could not find shmtools-python project root")

if str(project_root) not in sys.path:
    sys.path.insert(0, str(project_root))

print(f"Found shmtools at: {project_root}")

# Import SHMTools functions
from shmtools.utils.data_loading import load_3story_data
from shmtools.features import ar_model_shm
from shmtools.classification import (
    assemble_outlier_detector_shm,
    save_detector_assembly,
    load_detector_assembly,
    detector_registry,
    train_outlier_detector_shm,
    detect_outlier_shm,
    roc_shm
)

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

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

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

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

Load and Prepare Data

We'll use the 3-story structure dataset and extract AR model features:

In [2]:
```python
# Load the 3-story structure dataset
data = load_3story_data()
dataset = data['dataset']
damage_states = data['damage_states']

# Extract channels 2-5 (accelerations) - skip channel 0 (force)
acceleration_data = dataset[:, 1:, :]
print(f"Data shape: {acceleration_data.shape} (time_points, channels, instar

# Extract AR model features
ar_order = 15
```

```python
ar_features, _, _, _, _ = ar_model_shm(acceleration_data, ar_order)
print(f"AR features shape: {ar_features.shape} (instances, features)")

# Separate undamaged and damaged data
undamaged_mask = damage_states <= 9
damaged_mask = damage_states > 9

undamaged_features = ar_features[undamaged_mask]
damaged_features = ar_features[damaged_mask]

print(f"\nUndamaged instances: {undamaged_features.shape[0]}")
print(f"Damaged instances: {damaged_features.shape[0]}")
```

```
Data shape: (8192, 4, 170) (time_points, channels, instances)
AR features shape: (170, 60) (instances, features)

Undamaged instances: 90
Damaged instances: 80
```

Explore Available Detectors

Let's see what detectors are available in each category:

In [3]:
```python
# Display available detectors from the registry
print("=== PARAMETRIC DETECTORS ===")
for name, info in detector_registry.parametric_detectors.items():
    print(f"\n{name}:")
    print(f"  Display Name: {info['display_name']}")
    print(f"  Description: {info['description']}")
    print(f"  Learn Function: {info['learn_function']}")
    print(f"  Score Function: {info['score_function']}")

print("\n=== NON-PARAMETRIC DETECTORS ===")
for name, info in detector_registry.nonparametric_detectors.items():
    print(f"\n{name}:")
    print(f"  Display Name: {info['display_name']}")
    print(f"  Description: {info['description']}")
    print(f"  Available Kernels: {', '.join(detector_registry.available_kerr

print("\n=== SEMI-PARAMETRIC DETECTORS ===")
for name, info in detector_registry.semiparametric_detectors.items():
    print(f"\n{name}:")
    print(f"  Display Name: {info['display_name']}")
    print(f"  Description: {info['description']}")
    print(f"  Partitioning Algorithms: {', '.join(detector_registry.partitic
```

```
=== PARAMETRIC DETECTORS ===

pca:
  Display Name: Principal Component Analysis
  Description: PCA-based outlier detection using principal component scores
  Learn Function: learn_pca_shm
  Score Function: score_pca_shm

mahalanobis:
  Display Name: Mahalanobis Distance
  Description: Mahalanobis distance-based outlier detection
  Learn Function: learn_mahalanobis_shm
  Score Function: score_mahalanobis_shm

svd:
  Display Name: Singular Value Decomposition
  Description: SVD-based outlier detection using reconstruction errors
  Learn Function: learn_svd_shm
  Score Function: score_svd_shm

factor_analysis:
  Display Name: Factor Analysis
  Description: Factor analysis-based outlier detection
  Learn Function: learn_factor_analysis_shm
  Score Function: score_factor_analysis_shm

=== NON-PARAMETRIC DETECTORS ===

kernel_density:
  Display Name: Kernel Density Estimation
  Description: Non-parametric kernel density estimation for outlier detectio
n
  Available Kernels: gaussian, epanechnikov, quartic, triangle, triweight, u
niform, cosine

=== SEMI-PARAMETRIC DETECTORS ===

gmm_semi:
  Display Name: Gaussian Mixture Model (Semi-parametric)
  Description: Semi-parametric GMM-based outlier detection with partitioning
  Partitioning Algorithms: kmeans, kmedians, kdtree, pdtree, rptree
```

Example 1: Assemble a Parametric Detector (PCA)

First, let's assemble a PCA-based detector programmatically (non-interactive mode):

In [4]:
```python
# Assemble a PCA detector with custom parameters
pca_detector = assemble_outlier_detector_shm(
    suffix="PCA_Custom",
    detector_type="parametric",
    detector_name="pca",
    parameters={
        "per_var": 0.95,  # Retain 95% of variance
        "stand": 0        # Use standardization
```

```
    },
    interactive=False
)

print("Assembled PCA Detector:")
print(f"  Type: {pca_detector['type']}")
print(f"  Name: {pca_detector['name']}")
print(f"  Learn Function: {pca_detector['learn_function']}")
print(f"  Score Function: {pca_detector['score_function']}")
print(f"  Parameters: {pca_detector['parameters']}")
print(f"  Training Function: {pca_detector['training_function'].__name__}")
```

```
\n=== SHMTools Custom Detector Assembly ===
Assembling custom outlier detector with configurable components.\n
\n✅ Custom detector 'pca_PCA_Custom' assembled successfully!
   Type: parametric
   Learning function: learn_pca_shm
   Scoring function: score_pca_shm
   Parameters: {'per_var': 0.95, 'stand': 0}
Assembled PCA Detector:
  Type: parametric
  Name: pca
  Learn Function: learn_pca_shm
  Score Function: score_pca_shm
  Parameters: {'per_var': 0.95, 'stand': 0}
  Training Function: train_detector_PCA_Custom
```

Use the Assembled PCA Detector

Now let's use the assembled detector to train and test on our data:

In [5]:

```
# Split data for training and testing
train_split = 0.8
n_train = int(train_split * len(undamaged_features))

# Training data: 80% of undamaged
train_features = undamaged_features[:n_train]

# Test data: 20% undamaged + all damaged
test_features = np.vstack([
    undamaged_features[n_train:],
    damaged_features
])

# Create labels for test data
test_labels = np.concatenate([
    np.zeros(len(undamaged_features[n_train:])),  # Undamaged = 0
    np.ones(len(damaged_features))                # Damaged = 1
]).astype(int)

print(f"Training samples: {len(train_features)}")
print(f"Test samples: {len(test_features)} ({np.sum(test_labels == 0)} undam

# Train using the assembled detector's training function
models = pca_detector['training_function'](
```

```python
    train_features,
    k=5,
    confidence=0.95,
    model_filename="assembled_pca_model.pkl"
)

# Detect outliers
results, confidences, scores, threshold = detect_outlier_shm(
    test_features,
    models=models
)

# Calculate performance metrics
accuracy = np.mean(results == test_labels)
false_positive_rate = np.mean(results[test_labels == 0] == 1)
false_negative_rate = np.mean(results[test_labels == 1] == 0)

print(f"\nPerformance Metrics:")
print(f"  Accuracy: {accuracy:.3f}")
print(f"  False Positive Rate: {false_positive_rate:.3f}")
print(f"  False Negative Rate: {false_negative_rate:.3f}")
```

```
Training samples: 72
Test samples: 98 (18 undamaged, 80 damaged)
**** Training custom detector: pca_PCA_Custom ****
Model saved to: assembled_pca_model.pkl

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

Detection summary:
  Total instances: 98
  Outliers detected: 97 (99.0%)
  Threshold used: -2.2845
  Score range: [-27.1947, -1.8524]

Performance Metrics:
  Accuracy: 0.827
  False Positive Rate: 0.944
  False Negative Rate: 0.000
```

Example 2: Assemble a Non-Parametric Detector (Kernel Density)

Let's assemble a kernel density detector with Epanechnikov kernel:

In [6]:

```python
# Assemble a kernel density detector
kde_detector = assemble_outlier_detector_shm(
    suffix="KDE_Epanechnikov",
    detector_type="nonparametric",
    detector_name="kernel_density",
    parameters={
        "kernel_function": "epanechnikov",
        "bandwidth_method": "scott"  # Use Scott's rule for bandwidth
    },
```

```
    interactive=False
)

print("Assembled KDE Detector:")
print(f"  Type: {kde_detector['type']}")
print(f"  Name: {kde_detector['name']}")
print(f"  Parameters: {kde_detector['parameters']}")

# Train and test with KDE detector
kde_models = kde_detector['training_function'](
    train_features,
    k=3,
    confidence=0.95,
    model_filename="assembled_kde_model.pkl"
)

# Detect outliers
kde_results, kde_confidences, kde_scores, kde_threshold = detect_outlier_shm
    test_features,
    models=kde_models
)
```

```
\n=== SHMTools Custom Detector Assembly ===
Assembling custom outlier detector with configurable components.\n
\n✅ Custom detector 'kernel_density_KDE_Epanechnikov' assembled successfull
y!
   Type: nonparametric
   Learning function: learn_kernel_density_shm
   Scoring function: score_kernel_density_shm
   Parameters: {'kernel_function': 'epanechnikov', 'bandwidth_method': 'scot
t'}
Assembled KDE Detector:
  Type: nonparametric
  Name: kernel_density
  Parameters: {'kernel_function': 'epanechnikov', 'bandwidth_method': 'scot
t'}
**** Training custom detector: kernel_density_KDE_Epanechnikov ****
Model saved to: assembled_kde_model.pkl

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

Detection summary:
  Total instances: 98
  Outliers detected: 98 (100.0%)
  Threshold used: 64.1672
  Score range: [-708.3964, 62.2952]
```

Example 3: Assemble a Semi-Parametric Detector (GMM)

Let's assemble a GMM-based semi-parametric detector:

In [7]:
```
# Assemble a GMM semi-parametric detector
gmm_detector = assemble_outlier_detector_shm(
```

```python
    suffix="GMM_KMeans",
    detector_type="semiparametric",
    detector_name="gmm_semi",
    parameters={
        "partitioning_algorithm": "kmeans",
        "n_components": 5
    },
    interactive=False
)

print("Assembled GMM Detector:")
print(f"  Type: {gmm_detector['type']}")
print(f"  Name: {gmm_detector['name']}")
print(f"  Parameters: {gmm_detector['parameters']}")

# Train and test with GMM detector
gmm_models = gmm_detector['training_function'](
    train_features,
    k=5,
    confidence=0.95,
    model_filename="assembled_gmm_model.pkl",
    dist_for_scores="norm"  # Use normal distribution for threshold
)

# Detect outliers
gmm_results, gmm_confidences, gmm_scores, gmm_threshold = detect_outlier_shm
    test_features,
    models=gmm_models
)
```

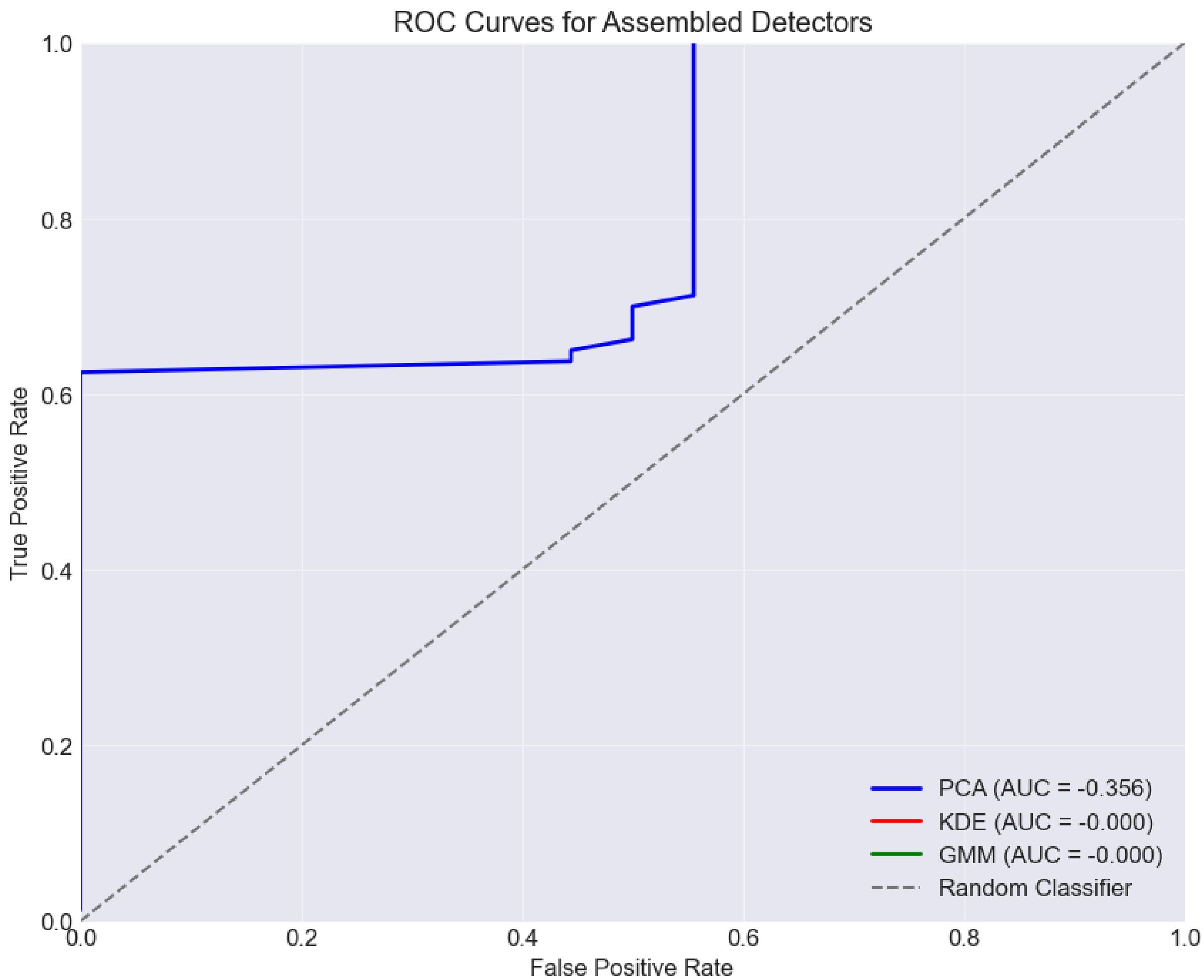
```
\n=== SHMTools Custom Detector Assembly ===
Assembling custom outlier detector with configurable components.\n
\n✅ Custom detector 'gmm_semi_GMM_KMeans' assembled successfully!
   Type: semiparametric
   Learning function: learn_gmm_semiparametric_model_shm
   Scoring function: score_gmm_semiparametric_model_shm
   Parameters: {'partitioning_algorithm': 'kmeans', 'n_components': 5}
Assembled GMM Detector:
  Type: semiparametric
  Name: gmm_semi
  Parameters: {'partitioning_algorithm': 'kmeans', 'n_components': 5}
**** Training custom detector: gmm_semi_GMM_KMeans ****
Model saved to: assembled_gmm_model.pkl

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

Detection summary:
  Total instances: 98
  Outliers detected: 98 (100.0%)
  Threshold used: 229.1420
  Score range: [-708.3964, -708.3964]
```

Compare Detector Performance

Let's compare the performance of all three assembled detectors:

In [8]:
```
# Calculate ROC curves for all detectors
pca_tpr, pca_fpr = roc_shm(scores, test_labels)
kde_tpr, kde_fpr = roc_shm(kde_scores, test_labels)
gmm_tpr, gmm_fpr = roc_shm(gmm_scores, test_labels)

# Plot ROC curves
plt.figure(figsize=(10, 8))

# Calculate AUC using trapezoidal rule
pca_auc = -np.trapz(pca_tpr, pca_fpr)
kde_auc = -np.trapz(kde_tpr, kde_fpr)
gmm_auc = -np.trapz(gmm_tpr, gmm_fpr)

plt.plot(pca_fpr, pca_tpr, 'b-', linewidth=2, label=f'PCA (AUC = {pca_auc:.3
plt.plot(kde_fpr, kde_tpr, 'r-', linewidth=2, label=f'KDE (AUC = {kde_auc:.3
plt.plot(gmm_fpr, gmm_tpr, 'g-', linewidth=2, label=f'GMM (AUC = {gmm_auc:.3
plt.plot([0, 1], [0, 1], 'k--', alpha=0.5, label='Random Classifier')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Assembled Detectors')
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.show()

# Performance summary table
print("\n=== PERFORMANCE SUMMARY ===")
print(f"{'Detector':<15} {'Accuracy':<10} {'FPR':<10} {'FNR':<10} {'AUC':<10
print("-" * 55)

# PCA performance
pca_acc = np.mean(results == test_labels)
pca_fpr_val = np.mean(results[test_labels == 0] == 1)
pca_fnr = np.mean(results[test_labels == 1] == 0)
print(f"{'PCA':<15} {pca_acc:<10.3f} {pca_fpr_val:<10.3f} {pca_fnr:<10.3f} {

# KDE performance
kde_acc = np.mean(kde_results == test_labels)
kde_fpr_val = np.mean(kde_results[test_labels == 0] == 1)
kde_fnr = np.mean(kde_results[test_labels == 1] == 0)
print(f"{'KDE':<15} {kde_acc:<10.3f} {kde_fpr_val:<10.3f} {kde_fnr:<10.3f} {

# GMM performance
gmm_acc = np.mean(gmm_results == test_labels)
gmm_fpr_val = np.mean(gmm_results[test_labels == 0] == 1)
gmm_fnr = np.mean(gmm_results[test_labels == 1] == 0)
print(f"{'GMM':<15} {gmm_acc:<10.3f} {gmm_fpr_val:<10.3f} {gmm_fnr:<10.3f} {
```

```
/var/folders/v_/sg5j00lj4n381c9z439qs2wc0000gn/T/ipykernel_92156/923934440.p
y:10: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, or
one of the numerical integration functions in `scipy.integrate`.
  pca_auc = -np.trapz(pca_tpr, pca_fpr)
/var/folders/v_/sg5j00lj4n381c9z439qs2wc0000gn/T/ipykernel_92156/923934440.p
y:11: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, or
one of the numerical integration functions in `scipy.integrate`.
  kde_auc = -np.trapz(kde_tpr, kde_fpr)
/var/folders/v_/sg5j00lj4n381c9z439qs2wc0000gn/T/ipykernel_92156/923934440.p
y:12: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, or
one of the numerical integration functions in `scipy.integrate`.
  gmm_auc = -np.trapz(gmm_tpr, gmm_fpr)
```




```
=== PERFORMANCE SUMMARY ===
Detector        Accuracy   FPR        FNR        AUC
-------------------------------------------------------
PCA             0.827      0.944      0.000      -0.356
KDE             0.816      1.000      0.000      -0.000
GMM             0.816      1.000      0.000      -0.000
```

Save and Load Detector Configurations

Detector assemblies can be saved and loaded for reproducibility:

In [9]:
```
# Save detector configurations
save_detector_assembly(pca_detector, "pca_detector_config.json")
save_detector_assembly(kde_detector, "kde_detector_config.json")
save_detector_assembly(gmm_detector, "gmm_detector_config.json")

print("Detector configurations saved!")

# Load a detector configuration
loaded_pca = load_detector_assembly("pca_detector_config.json")

print("\nLoaded PCA detector configuration:")
print(f"  Type: {loaded_pca['type']}")
print(f"  Name: {loaded_pca['name']}")
print(f"  Parameters: {loaded_pca['parameters']}")
```

```
Detector assembly saved to: pca_detector_config.json
Detector assembly saved to: kde_detector_config.json
Detector assembly saved to: gmm_detector_config.json
Detector configurations saved!
Detector assembly loaded from: pca_detector_config.json
Note: Use assemble_outlier_detector_shm to regenerate the training function.

Loaded PCA detector configuration:
  Type: parametric
  Name: pca
  Parameters: {'per_var': 0.95, 'stand': 0}
```

Interactive Assembly Example

For interactive assembly (when `interactive=True`), the function will prompt you for:

1. Detector type selection
2. Specific detector algorithm selection
3. Parameter configuration

This is useful for exploring different detector configurations without writing code.

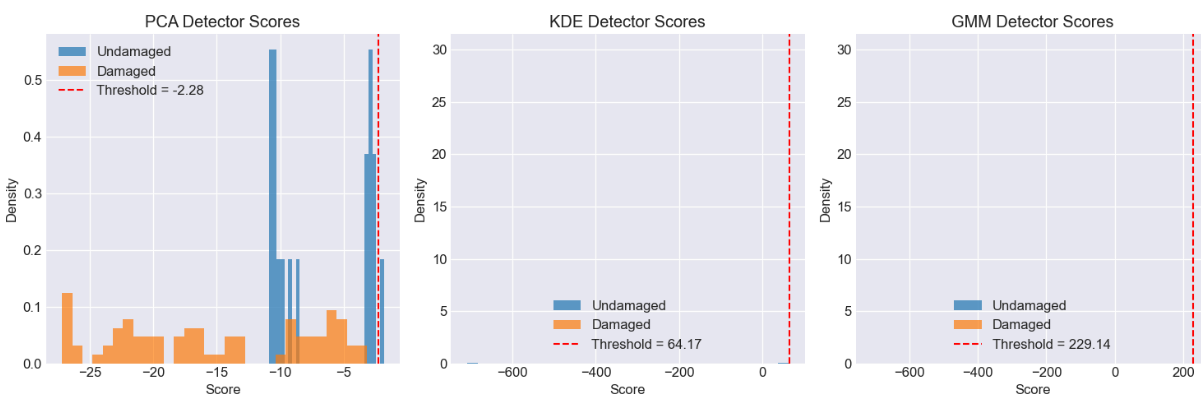
```
# Example of interactive assembly (commented out for notebook
execution)
# interactive_detector =
assemble_outlier_detector_shm(interactive=True)
```

Visualize Score Distributions

Let's visualize how different detectors score the data:

In [10]:
```
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# PCA scores
axes[0].hist(scores[test_labels == 0], bins=30, alpha=0.7, density=True, lab
axes[0].hist(scores[test_labels == 1], bins=30, alpha=0.7, density=True, lab
```

```python
axes[0].axvline(threshold, color='r', linestyle='--', label=f'Threshold = {t
axes[0].set_xlabel('Score')
axes[0].set_ylabel('Density')
axes[0].set_title('PCA Detector Scores')
axes[0].legend()

# KDE scores
axes[1].hist(kde_scores[test_labels == 0], bins=30, alpha=0.7, density=True,
axes[1].hist(kde_scores[test_labels == 1], bins=30, alpha=0.7, density=True,
axes[1].axvline(kde_threshold, color='r', linestyle='--', label=f'Threshold
axes[1].set_xlabel('Score')
axes[1].set_ylabel('Density')
axes[1].set_title('KDE Detector Scores')
axes[1].legend()

# GMM scores
axes[2].hist(gmm_scores[test_labels == 0], bins=30, alpha=0.7, density=True,
axes[2].hist(gmm_scores[test_labels == 1], bins=30, alpha=0.7, density=True,
axes[2].axvline(gmm_threshold, color='r', linestyle='--', label=f'Threshold
axes[2].set_xlabel('Score')
axes[2].set_ylabel('Density')
axes[2].set_title('GMM Detector Scores')
axes[2].legend()

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



Summary

This notebook demonstrated:

1. **Custom Detector Assembly**: How to create custom outlier detectors by combining different learning and scoring functions
2. **Detector Categories**: Working with parametric (PCA), non-parametric (KDE), and semi-parametric (GMM) detectors
3. **Parameter Configuration**: Setting specific parameters for each detector type
4. **Performance Comparison**: Evaluating multiple detectors on the same dataset
5. **Configuration Management**: Saving and loading detector configurations for reproducibility

The custom detector assembly framework provides flexibility to:

- Mix and match algorithms based on your specific application
- Fine-tune parameters for optimal performance
- Create reproducible detection workflows
- Integrate seamlessly with the universal `detect_outlier_shm` interface

This framework is particularly useful when:

- Default detectors don't meet your specific requirements
- You need to explore different algorithmic approaches
- You want to create application-specific detection pipelines
- Reproducibility and configuration management are important