Example Usage: Direct Use of Semi-Parametric Routines

Introduction

Here we show how to directly use the Semiparametric routines while bypassing the "trainOutlierDetector" routine.

The data used in this example is from the 3-story structure. More details about the data sets can be found in 3-Story Data Sets.

Requires data3SS.mat dataset.

SHMTools functions called:

- arModel_shm
- learnGMMSemiParametricModel_shm
- scoreGMM_shm

Author: Samory Kpotufe

Date Created: August 19, 2009

LA-CC-14-046

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
from shmtools.features import ar_model_shm
from shmtools.classification import (
    learn_gmm_semiparametric_model_shm,
    score_gmm_shm,
    score_gmm_semiparametric_model_shm,
    k_medians_shm,
    roc_shm
)
from shmtools.utils.data_loading import load_3story_data
```

```
/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 data

The data here is in the form of time series in a 3 dimensional matrix (time, sensors, instances) and also a state vector representing the various environmental conditions under which the data is collected.

In [2]:
```
data = load_3story_data()
dataset = data['dataset']
states = data['damage_states']
```

For this example, we will break each 8192 point time series into 4, 2048 point time series.

In [3]:
```
time_data = np.zeros((2048, 5, 680))
time_data_states = np.zeros(680)
for i in range(4):
    start_idx = 2048 * i
    end_idx = 2048 * (i + 1)
    time_data[:, :, i::4] = dataset[start_idx:end_idx, :, :]
    time_data_states[i::4] = states
```

Extract some features using your favorite function, but first pick N of the instances (each time series reading over all sensors). Each instance is then transformed into a feature vector: the returned matrix has the form (instances, features).

In [4]:
```
N = 400
np.random.seed(42)  # For reproducibility
idx = np.random.permutation(time_data.shape[2])[:N]
X_data = ar_model_shm(time_data[:, :, idx])[1]  # Get RMS residuals feature
X_states = time_data_states[idx]
```

Now set 80% of states 1:9 aside as the training data, these states correspond to undamaged readings. We'll then test on the remaining 20% of 1:9 and on the "damaged" states 10:17.

In [5]:
```
idx = np.isin(X_states, range(1, 10))  # States 1:9
X_undamaged = X_data[idx, :]
n_undamaged = X_undamaged.shape[0]
n_train = round(0.8 * n_undamaged)
X_train = X_undamaged[:n_train, :]
X_test = np.vstack([X_undamaged[n_train:, :], X_data[~idx, :]])
n_test = X_test.shape[0]
```

Now set labels for the test data, 0 corresponds to undamaged, and 1 to damaged.

number of undamaged in test.

In [6]:
```
n_test_0 = n_undamaged - n_train
```

test labels

In [7]:
```
test_labels = np.concatenate([np.zeros(n_test_0), np.ones(n_test - n_test_0)
```

Train a model over the undamaged data

The next call learns a mixture of k gaussians over the undamaged data and returns the parameters of this model in dModel. The partition function is one of those in "SemiParametricDetectors/PartitioningAlgorithms/" or should have the same behavior as one of those functions (including signature). The "MMFun" is a Mixture Model function from "SemiParametricDetectors/ParametricMixtures" or should have the same behavior.

In [8]:
```
partition_fun = k_medians_shm
k = 5
d_model = learn_gmm_semiparametric_model_shm(X_train, partition_fun, k)
```

Pick a threshold from the training data

We will first obtain the "scores" over the training data, that is the log-likelihoods that are given by the learned distribution. Then we learn a distribution of these scores, and pick a threshold so that 90% of the training data (undamaged data) has scores above this threshold (according to the distribution of scores).

In [9]:
```
likelihoods = score_gmm_shm(X_train, d_model)
```

learn a normal distribution over the scores

In [10]:
```
model_p = stats.norm.fit(likelihoods)
```

pick the threshold

In [11]:
```
confidence = 0.9
threshold = stats.norm.ppf(1 - confidence, model_p[0], model_p[1])
```

Test the detector

Now the detector consists simply of getting the distribution of scores over the test data, under the distribution learned on the undamaged training data (dModel). We simply flag a test point as "damaged" whenever it falls below our threshold.

Test scores

In [12]:
```
scores = score_gmm_semiparametric_model_shm(X_test, d_model)
```

Results contains a 1 whenever we think the point is damaged, a 0 otherwise.

In [13]:
```
results = scores <= threshold
```

Report the detector's performance

Various error rates

In [14]:
```
total_err = np.sum(results != test_labels) / n_test
false_positive_err = np.sum(results[:n_test_0] != 0) / n_test_0
false_negative_err = np.sum(results[n_test_0:] != 1) / (n_test - n_test_0)
print(f'\n Total error: {total_err:.2f}\n False Positive rate: {false_positi
```

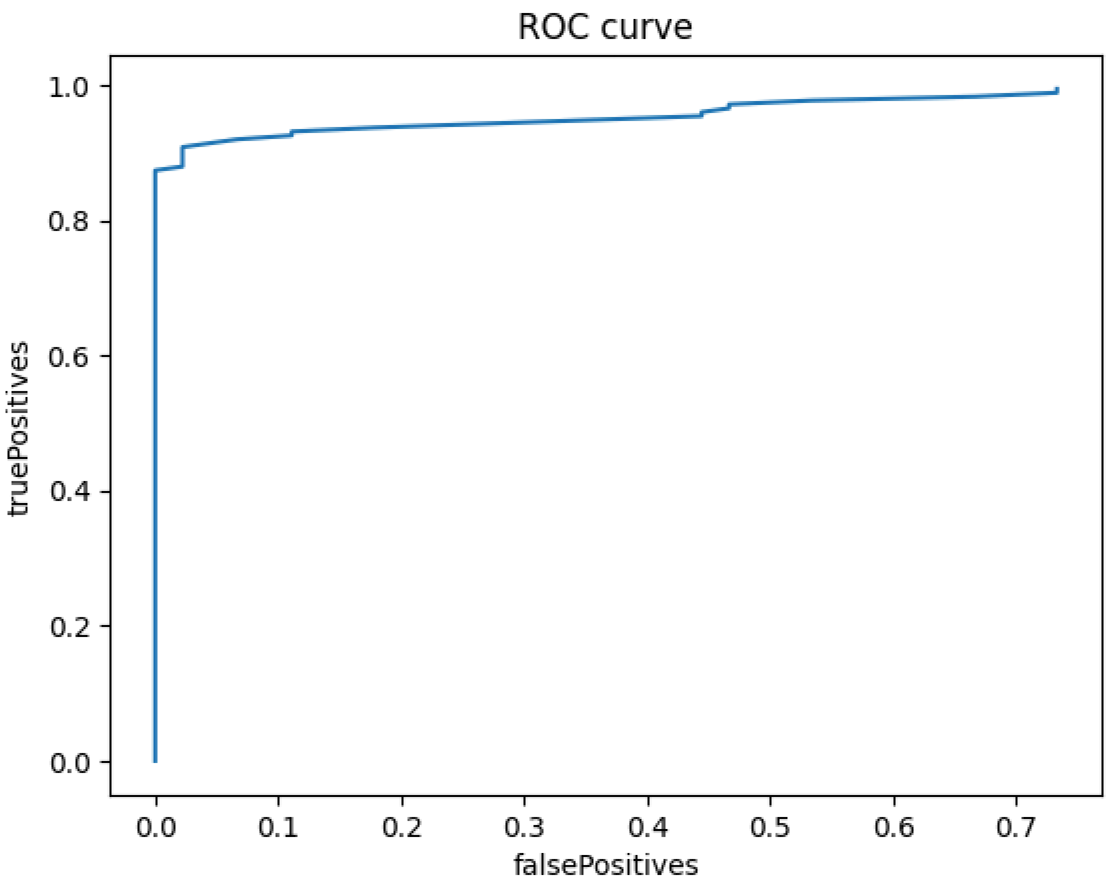
```
Total error: 0.09
False Positive rate: 0.20
False Negative rate: 0.06
```

ROC curve

In [15]:
```
true_positives, false_positives = roc_shm(scores, test_labels)
```

Now plot the curve

In [16]:
```
plt.figure()
plt.plot(false_positives, true_positives)
plt.xlabel('falsePositives')
plt.ylabel('truePositives')
plt.title('ROC curve')
plt.show()
```



ROC curve
truePositives
falsePositives
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7