One-Class SVM: Overview and Purpose

One-Class SVM (Support Vector Machine) is primarily used for anomaly detection or novelty detection in an unsupervised learning setting. The main purpose is to identify data points that do not conform to the expected pattern of the majority of the data. This is particularly useful in scenarios where you have a large amount of data and need to detect outliers or rare events.

When to Use One-Class SVM

One-Class SVM is suitable for:

1. **Anomaly Detection**: Identifying fraudulent transactions in finance, unusual network traffic in cybersecurity, or defects in manufacturing.
2. **Novelty Detection**: Detecting new or previously unseen data points that differ significantly from the training data.

Suitable Data Types

One-Class SVM works well with data where the majority class or normal behavior is well-defined and anomalies are rare. It can handle both structured and unstructured data, including:

- Numerical data
- Time series data
- Text data (after appropriate preprocessing)

Examples of Use Cases

1. **Fraud Detection**: Detecting unusual credit card transactions that may indicate fraud.
2. **Network Security**: Identifying abnormal patterns in network traffic that could signify a cyber attack.
3. **Manufacturing**: Spotting defective products in an assembly line.

Implementing One-Class SVM on a Real-Life Dataset

Let's implement One-Class SVM using Python on a real-life dataset. We will use the `pyod` library, which is designed for outlier detection.

```
# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from pyod.models.ocsvm import OCSVM

from pyod.utils.data import generate_data

from pyod.utils.example import visualize
```

```
from pyod.utils.data import evaluate_print

# Step 1: Generate Sample Data
contamination = 0.1  # proportion of outliers
n_train = 200  # number of training points
n_test = 100  # number of testing points
X_train, X_test, y_train, y_test = generate_data(n_train=n_train, n_test=n_test, n_features=2,
contamination=contamination)

# Visualize the dataset before applying One-Class SVM
plt.figure(figsize=(10, 5))
plt.scatter(X_train[:, 0], X_train[:, 1], c='b', marker='o', label='Training data')
plt.scatter(X_test[:, 0], X_test[:, 1], c='r', marker='x', label='Testing data')
plt.title("Dataset before applying One-Class SVM")
plt.legend()
plt.show()

# Step 2: Train One-Class SVM
clf = OCSVM()
clf.fit(X_train)

# Get the prediction labels and decision function scores
y_train_pred = clf.labels_  # binary labels (0: inliers, 1: outliers)
y_train_scores = clf.decision_scores_  # raw outlier scores
y_test_pred = clf.predict(X_test)  # outlier labels (0 or 1) for test data

# Step 3: Evaluate the Model
evaluate_print('OCSVM', y_test, y_test_pred)
```

```
# Visualize the dataset after applying One-Class SVM

plt.figure(figsize=(10, 5))


# Before applying One-Class SVM

plt.subplot(1, 2, 1)

plt.scatter(X_train[:, 0], X_train[:, 1], c='b', marker='o', label='Training Data')

plt.title('Training Data Before One-Class SVM')

plt.legend()


# After applying One-Class SVM

plt.subplot(1, 2, 2)

plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train_pred, marker='o', cmap='coolwarm', label='Training Data')

plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test_pred, marker='x', cmap='coolwarm', label='Testing Data')

plt.title('Training Data After One-Class SVM')

plt.legend()


plt.show()


# Visualize the results using pyod's built-in visualize function

visualize(clf, X_train, X_test, y_train, y_test, y_train_pred, y_test_pred)
```

Explanation of the Code

- `import numpy as np`: Imports the NumPy library for numerical operations.
- `import matplotlib.pyplot as plt`: Imports Matplotlib for data visualization.
- `from pyod.models.ocsvm import OCSVM`: Imports the One-Class SVM model from the `pyod` (Python Outlier Detection) library.
- `from pyod.utils.data import generate_data`: Imports a utility function to generate synthetic data.
- `from pyod.utils.example import visualize`: Imports a utility function to visualize data and model results.

- `from pyod.utils.data import evaluate_print`: Imports a function to print evaluation metrics.

- `contamination = 0.1`: Sets the proportion of outliers in the dataset to 10%.
- `n_train = 200`: Sets the number of training data points to 200.
- `n_test = 100`: Sets the number of testing data points to 100.
- `X_train, X_test, y_train, y_test = generate_data(...)`: Generates synthetic training and testing data with the specified parameters. `X_train` and `X_test` are the feature matrices for training and testing, while `y_train` and `y_test` are the corresponding labels indicating inliers (0) and outliers (1).

- `plt.figure(figsize=(10, 5))`: Creates a new figure with a specified size.
- `plt.scatter(X_train[:, 0], X_train[:, 1], c='b', marker='o', label='Training data')`: Plots the training data as blue circles.
- `plt.scatter(X_test[:, 0], X_test[:, 1], c='r', marker='x', label='Testing data')`: Plots the testing data as red crosses.
- `plt.title("Dataset before applying One-Class SVM")`: Sets the title of the plot.
- `plt.legend()`: Displays the legend.
- `plt.show()`: Displays the plot.

- `clf = OCSVM()`: Initializes the One-Class SVM model.
- `clf.fit(X_train)`: Trains the One-Class SVM model on the training data.

- `y_train_pred = clf.labels_`: Retrieves the binary labels for the training data (0 for inliers, 1 for outliers).
- `y_train_scores = clf.decision_scores_`: Retrieves the raw outlier scores for the training data.
- `y_test_pred = clf.predict(X_test)`: Predicts the outlier labels for the testing data.

`evaluate_print('OCSVM', y_test, y_test_pred)`: Evaluates the model's performance on the testing data and prints the evaluation metrics (e.g., precision, recall, F1 score).

- `plt.figure(figsize=(10, 5))`: Creates a new figure with a specified size.
- `plt.subplot(1, 2, 1)`: Creates a subplot in a 1x2 grid (first subplot).
- `plt.scatter(X_train[:, 0], X_train[:, 1], c='b', marker='o', label='Training Data')`: Plots the training data as blue circles.
- `plt.title('Training Data Before One-Class SVM')`: Sets the title of the first subplot.
- `plt.legend()`: Displays the legend for the first subplot.
- `plt.subplot(1, 2, 2)`: Creates a subplot in a 1x2 grid (second subplot).
- `plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train_pred, marker='o', cmap='coolwarm', label='Training Data')`: Plots the training data with colors based on the predicted labels.
- `plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test_pred, marker='x', cmap='coolwarm', label='Testing Data')`: Plots the testing data with colors based on the predicted labels.
- `plt.title('Training Data After One-Class SVM')`: Sets the title of the second subplot.

- `plt.legend()`: Displays the legend for the second subplot.
- `plt.show()`: Displays the plot.

`visualize(...)`: Uses the `pyod` library's built-in `visualize` function to create a detailed visualization of the training and testing data, including the predicted inliers and outliers.

Elliptic Envelope in Unsupervised Learning

Purpose

The Elliptic Envelope method is used for robust covariance estimation, which helps in identifying outliers in a dataset. It assumes that the data follows a Gaussian distribution and attempts to estimate the inliers of the dataset while disregarding the outliers. The primary purpose is to model the data as an ellipse (in multiple dimensions) and find the data points that fit within this ellipse, flagging those that fall outside as outliers.

Suitable Data Types

Elliptic Envelope is suitable for continuous, multivariate data where the assumption of a Gaussian distribution is reasonable. It is particularly useful in anomaly detection tasks where the majority of the data is expected to conform to a normal distribution, and the anomalies (outliers) deviate significantly from this distribution.

Examples of Use Cases

1. **Fraud Detection**: Identifying fraudulent transactions in financial datasets where normal transactions follow a certain pattern.
2. **Manufacturing**: Detecting defective products in a production line where the measurements of a normally functioning product follow a specific distribution.
3. **Network Security**: Identifying abnormal network activities that deviate from normal usage patterns.

Implementation and Visualization

Let's implement the Elliptic Envelope on a real-life dataset and visualize the data before and after applying the model. We'll use the "Wine" dataset from the UCI Machine Learning Repository for this example. The dataset contains continuous, multivariate data, making it suitable for this method.

First, we need to load the necessary libraries and the dataset:

```
import numpy as np

import pandas as pd

from sklearn.covariance import EllipticEnvelope
```

```
import matplotlib.pyplot as plt
import seaborn as sns

# Load the wine dataset
from sklearn.datasets import load_wine
data = load_wine()
df = pd.DataFrame(data.data, columns=data.feature_names)

# For visualization, we'll use two features
df = df[['alcohol', 'malic_acid']]

# Plot the original data
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='alcohol', y='malic_acid')
plt.title('Original Data')
plt.show()
# Fit the Elliptic Envelope model
ee = EllipticEnvelope(contamination=0.1)  # Assume 10% of the data are outliers
ee.fit(df)

# Predict inliers and outliers
df['outlier'] = ee.predict(df[['alcohol', 'malic_acid']])

# Plot the data after applying Elliptic Envelope
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='alcohol', y='malic_acid', hue='outlier', palette=['blue', 'red'])
plt.title('Data after Elliptic Envelope')
plt.show()
```

- We used only two features (`alcohol` and `malic_acid`) for easier visualization.

- The `contamination` parameter is set to 0.1, assuming that 10% of the data are outliers.
- We plotted the data before and after applying the Elliptic Envelope to visualize the identified outliers.

Visualization Explanation

- **Original Data Plot**: This scatter plot shows the distribution of the two features (`alcohol` and `malic_acid`) in the dataset.
- **Data after Elliptic Envelope**: This scatter plot highlights the inliers (blue) and outliers (red) as identified by the Elliptic Envelope method. The inliers are those points that fit well within the estimated Gaussian distribution, while the outliers are those that fall outside the distribution.

This implementation and visualization demonstrate how the Elliptic Envelope can be used to identify outliers in a dataset, which is useful for anomaly detection in various real-life applications.