Anomaly Detection

Data Science and Machine Learning Lab

Flavio Giobergia

0.1 Anomaly Detection (overview)

Anomaly Detection (AD) is the process of identifying data points, events, or patterns that deviate significantly from expected behavior.

In **unsupervised AD**, the goal is to detect anomalies without labeled data – the model learns the normal patterns and flags deviations (==> We will mainly focus on this!)

In **supervised AD**, instances of anomalies are available, models learn to separate normal patterns from anomalies.

AD algorithms typically produce, for each sample, an **anomaly score** indicating how likely it is to be an anomaly. A threshold is then applied to classify samples as normal or anomalous.

0.2 Types of Anomalies

0.2.1 Point Anomalies

Individual data points that significantly differ from the rest of the population.

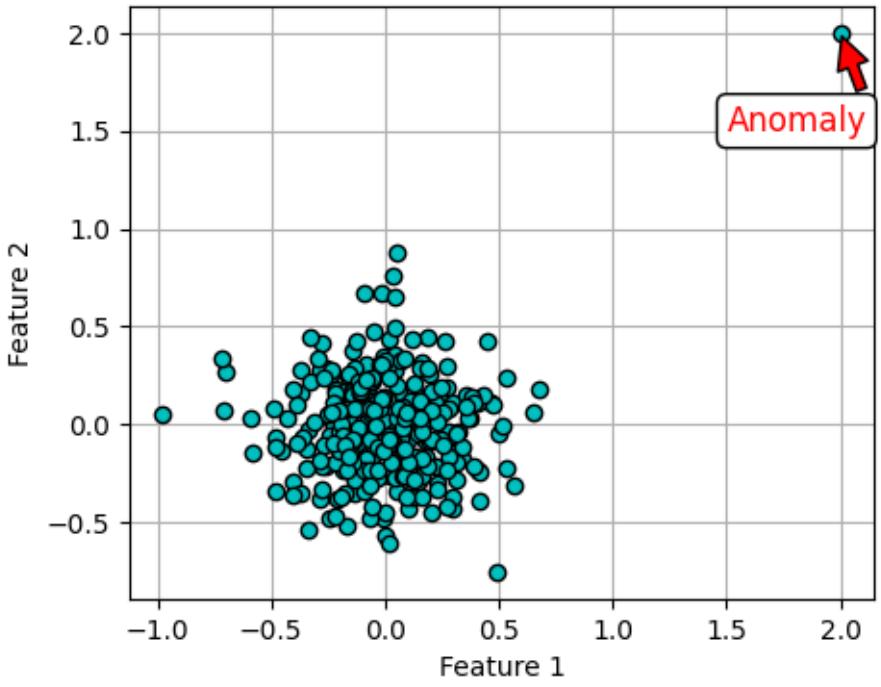
Example: a person withdrawing $10,000 in cash, at 3 AM from an ATM.

[1]:
```
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
```

[2]:
```
# add arrow to an anomalous point
def show_anomaly(ax, point):
    # white background, black border for the text box
    ax.annotate('Anomaly', xy=point, xytext=point - (0.5, 0.5),␣
 ↪arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, color='red',␣
 ↪bbox=dict(boxstyle="round,pad=0.3", edgecolor='black', facecolor='white'))
```

[3]:
```
# generate dataset (sampled from normal distribution, with 1 anomaly)
X_normal = np.random.normal(loc=0.0, scale=0.25, size=(300, 2))
X_anomaly = np.array([[2,2]])
X = np.vstack([X_normal, X_anomaly])

# plot dataset
fig, ax = plt.subplots(figsize=(5, 4))
```

```
ax.scatter(X[:, 0], X[:, 1], c='c', edgecolors='k')
show_anomaly(ax, X_anomaly[0])

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.grid()
ax.set_axisbelow(True)
```



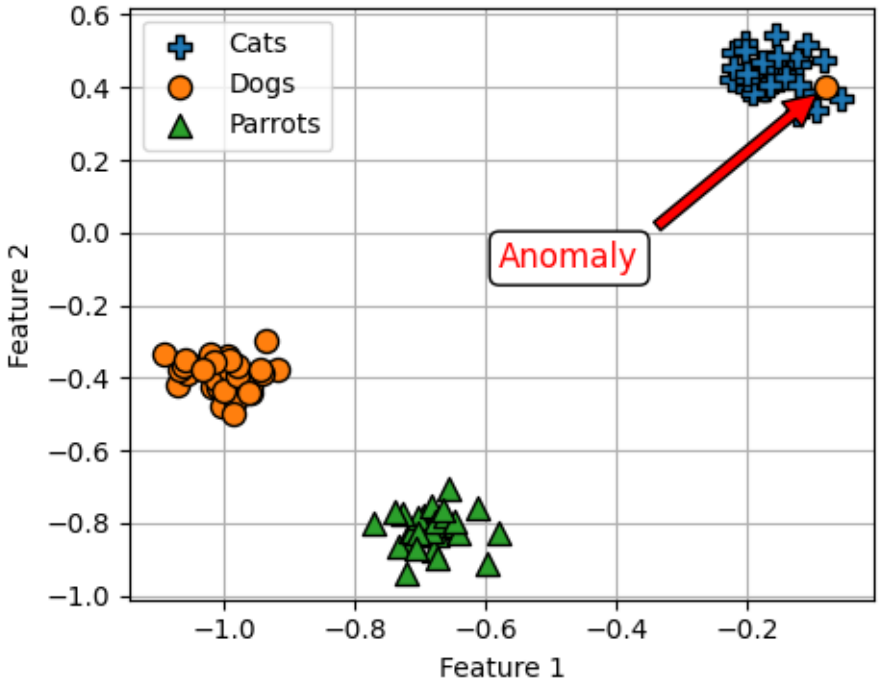
0.2.2 Contextual Anomalies

For contextual anomalies, a data point that is anomalous only within a certain context i.e., it would be "normal" in other circumstances.

Example: a person doing a wire transfer for $500,000. If it was a company, the operation would be less anomalous!

[4]:
```
X, y = make_blobs(n_samples=100, centers=3, cluster_std=0.5, random_state=1)
X *= .1
anomaly_id = (y == 0).nonzero()[0][-1]
y[anomaly_id] = 1

markers = ["P", "o", "^"]
group_labels = ["Cats", "Dogs", "Parrots"]
```

```
fig, ax = plt.subplots(figsize=(5, 4))
for c_id in np.unique(y):
    ax.scatter(X[y==c_id, 0], X[y==c_id, 1], label=group_labels[c_id],␣
 ↪edgecolors='k', marker=markers[c_id], s=75)

show_anomaly(ax, X[anomaly_id])

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.legend()
ax.grid()
ax.set_axisbelow(True)
```



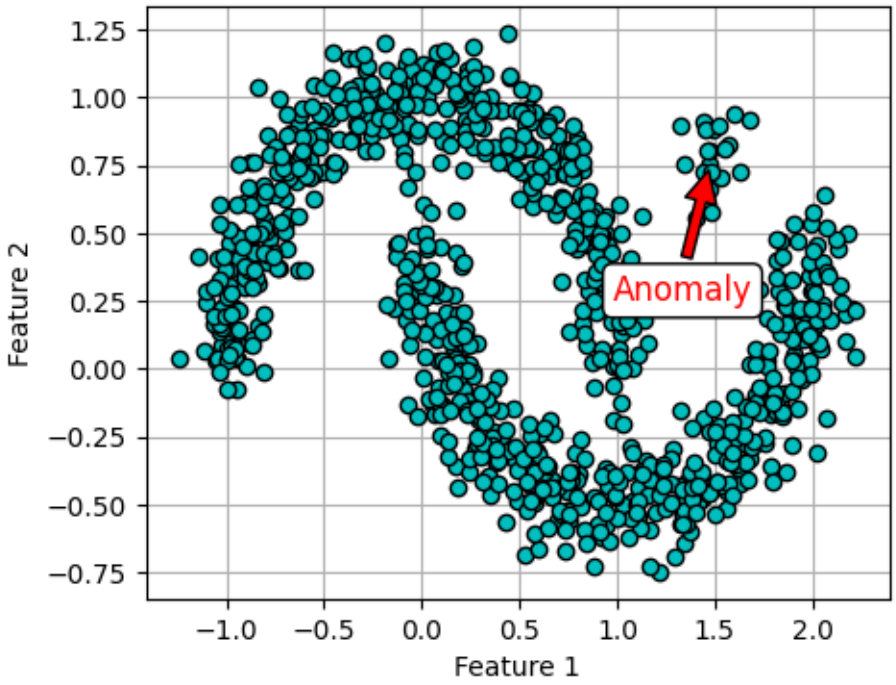
0.2.3 Collective Anomalies

A group of otherwise normal points that collectively form an anomaly.

Example: multiple people depositing cash for $9,000 and then wire-transferring the same amount to a foreign account. A single deposit of $9,000 is not anomalous, but the collective behavior is suspicious.

[5]:
```
from sklearn.datasets import make_moons
```

```
X, y = make_moons(n_samples=900, noise=0.1, random_state=1)
X_noise = np.random.normal(loc=(1.5,0.75), scale=0.075, size=(20, 2))

X = np.vstack([X, X_noise])

fig, ax = plt.subplots(figsize=(5, 4))
ax.scatter(X[:, 0], X[:, 1], c='c', edgecolors='k')

show_anomaly(ax, X_noise[0])

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.grid()
ax.set_axisbelow(True)
```



0.3 Anomaly Detection techniques

- **Statistical methods**:
  - Based on statistical models to identify data points that deviate significantly from the expected distribution
  - Example: Z-score method
- **Clustering-based methods**:
  - Use clustering algorithms to group similar data points together and identify those that do not belong to any cluster as anomalies

  - Example: DBSCAN, K-means
- **Proximity-based methods**:
  - Use distance or density measures to identify anomalies based on their proximity to other data points
  - Example: k-NN, Local Outlier Factor (LOF)
- **Tree-based methods**:
  - Use properties of trees to isolate anomalies in the data
  - Example: Isolation Forest
- **Boundary-based methods**:
  - Use decision boundaries to separate normal data points from anomalies
  - Example: One-Class SVM
- **Neural Network-based methods**:
  - Use neural networks to learn data distributions and identify anomalies (e.g., based on reconstruction errors)
  - Example: Autoencoders

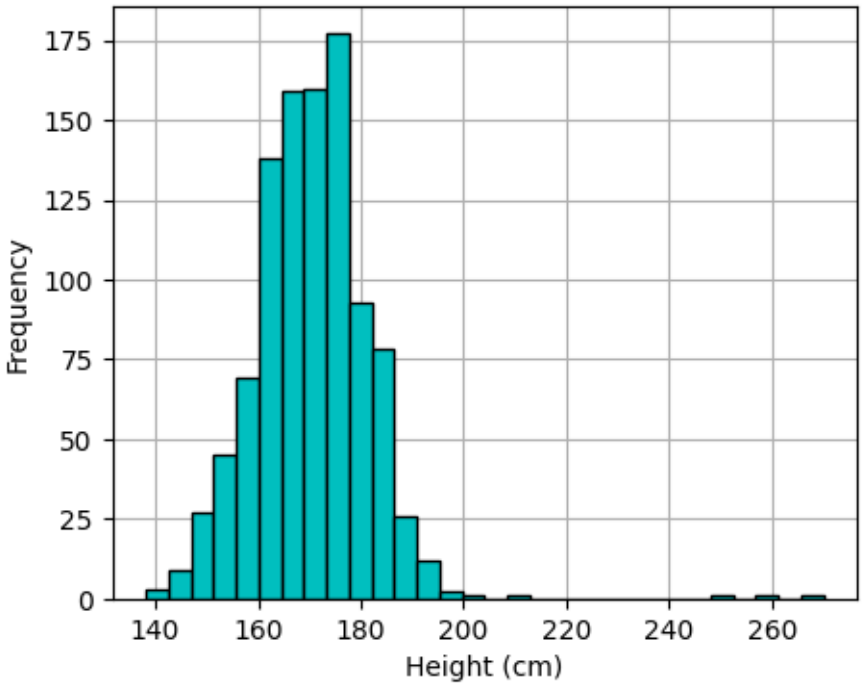
0.3.1 Statistical methods

Quantify the deviation of a data point from the expected distribution using statistical measures. For instance, the Z-score method calculates how many standard deviations a data point is from the mean. Points with high absolute Z-scores are considered anomalies.

Multivariate generalizations can also be used (e.g., Mahalanobis distance).

[6]:
```
heights = np.random.normal(loc=170, scale=10, size=1000)
heights_anomalies = np.array([250, 260, 270])
heights = np.hstack([heights, heights_anomalies])
```

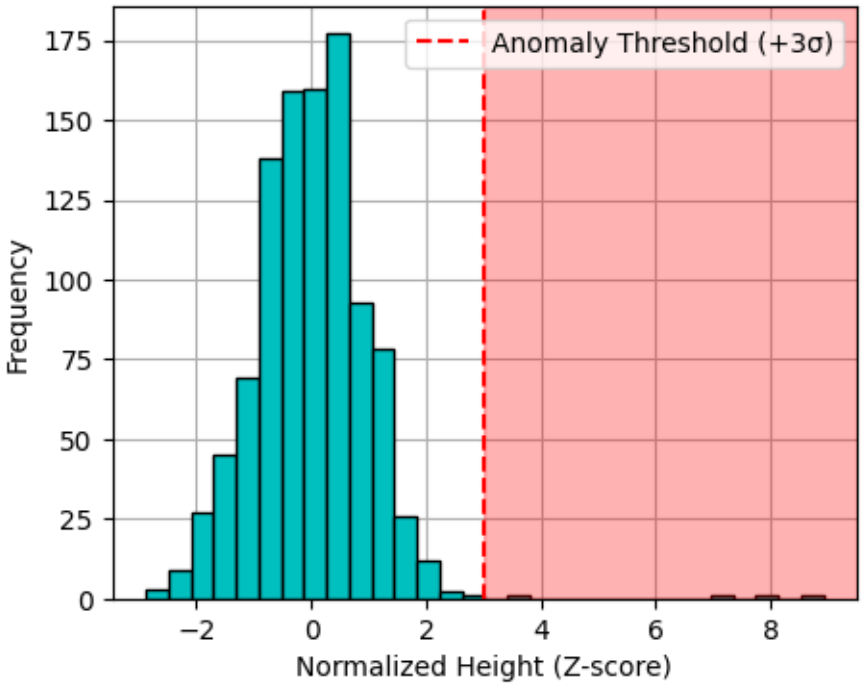
[7]:
```
fig, ax = plt.subplots(figsize=(5, 4))
ax.hist(heights, bins=30, color='c', edgecolor='k')
ax.set_xlabel('Height (cm)')
ax.set_ylabel('Frequency')
ax.grid()
ax.set_axisbelow(True)
```



[8]:
```
z_normalized = (heights - np.mean(heights)) / np.std(heights)

anomaly_threshold = 3

print(f"Anomalies (3σ): {heights[abs(z_normalized) > anomaly_threshold]}")
```

```
Anomalies (3σ): [208.59256742 250.         260.         270.        ]
```

[9]:
```
fig, ax = plt.subplots(figsize=(5, 4))
ax.hist(z_normalized, bins=30, color='c', edgecolor='k')
ax.set_xlabel('Normalized Height (Z-score)')
ax.set_ylabel('Frequency')
ax.grid()
ax.set_axisbelow(True)

ax.axvline(x=anomaly_threshold, color='r', linestyle='--', label=f'Anomaly␣
 ↪Threshold (+{anomaly_threshold}σ)')

xlim = ax.get_xlim()
ax.axvspan(anomaly_threshold, xlim[1], color='red', alpha=0.3)
ax.legend()
ax.set_xlim(xlim);
```

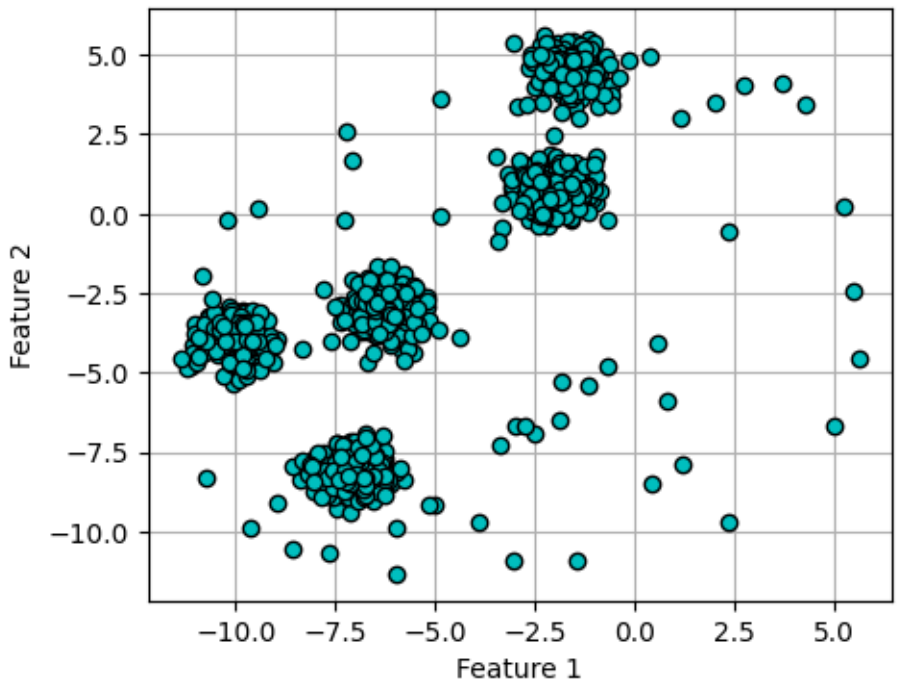


0.3.2 Clustering-based methods

Clustering algorithms group similar data points together. Data points that do not belong to any cluster or are in small clusters are considered anomalies.

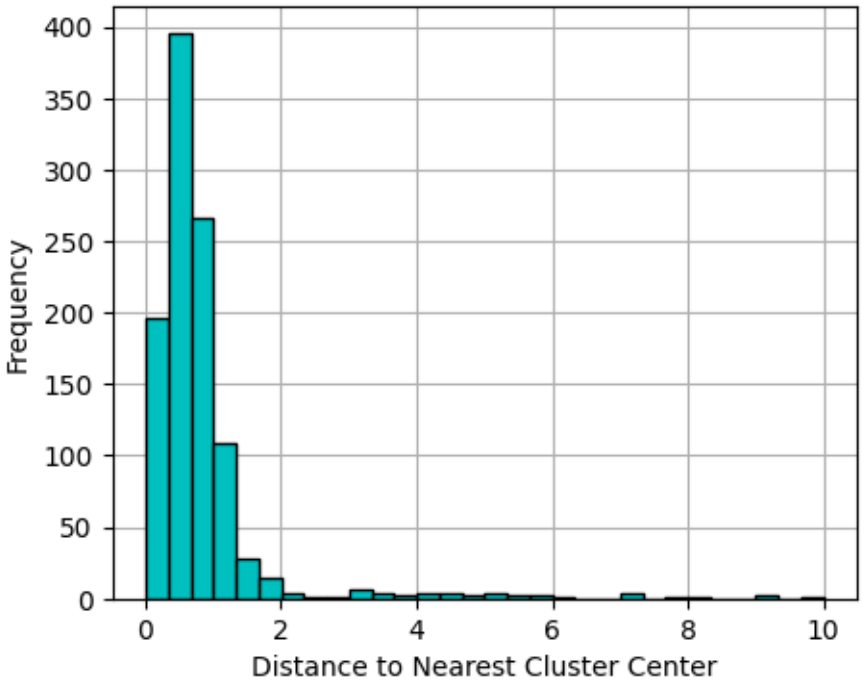
K-means With K-means, for example, points that are far from their assigned cluster centroids can be flagged as anomalies.

[10]:
```
X_blob, y_blob = make_blobs(n_samples=1_000, centers=5, cluster_std=0.5,␣
↪random_state=1)
X_anomalies = np.random.uniform(X_blob.min(), X_blob.max(), size=(50, 2))

X_blob = np.vstack([X_blob, X_anomalies])

fig, ax = plt.subplots(figsize=(5, 4))
ax.scatter(X_blob[:, 0], X_blob[:, 1], c='c', edgecolors='k')
ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.grid()
ax.set_axisbelow(True)
```

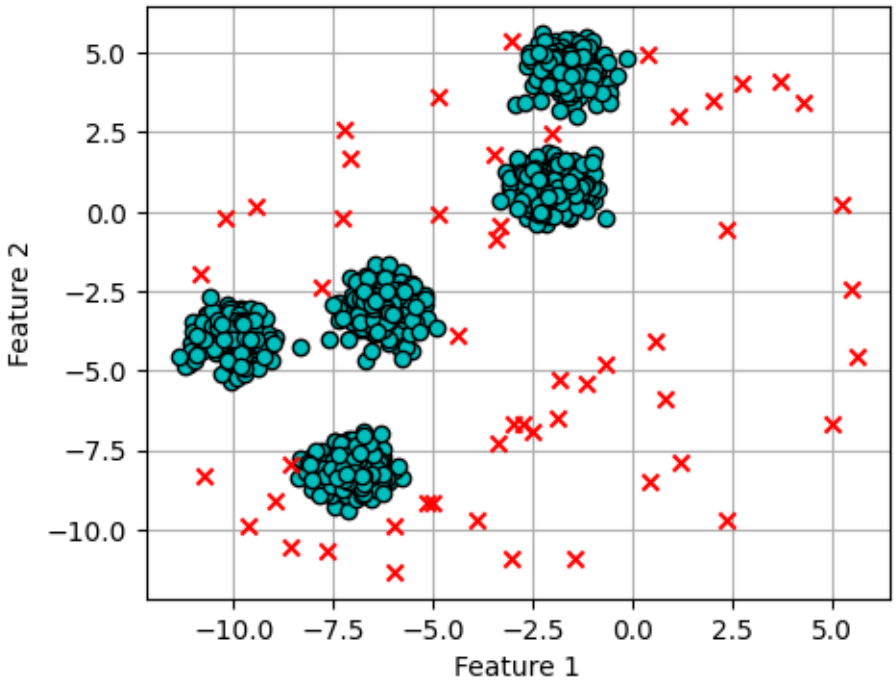


[11]:
```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=5)
model.fit(X_blob)

# transform() computes the distance between each point and each cluster center
# (equivalent to euclidean_distances(X, model.cluster_centers_))
# Each point is assigned to the closest cluster. With min(axis=1) we can extract␣
 ↪that distance
distances = model.transform(X_blob).min(axis=1)

fraction_of_anomalies = 0.05 # we expect ~ 5% of the data to be anomalies

anomalous_indices = distances.argsort()[-int(fraction_of_anomalies *␣
 ↪len(X_blob)):]
y_anomaly = np.zeros(len(X_blob), dtype=int)
y_anomaly[anomalous_indices] = 1
```

[12]:
```
fig, ax = plt.subplots(figsize=(5, 4))
ax.hist(distances, bins=30, color='c', edgecolor='k')
ax.set_xlabel('Distance to Nearest Cluster Center')
ax.set_ylabel('Frequency')
ax.grid()
ax.set_axisbelow(True)
```



[13]:
```
fig, ax = plt.subplots(figsize=(5, 4))
ax.scatter(X_blob[y_anomaly==0, 0], X_blob[y_anomaly==0, 1], c='c',␣
 ↪edgecolors='k')
ax.scatter(X_blob[y_anomaly==1, 0], X_blob[y_anomaly==1, 1], c='r', marker='x')

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.grid()
ax.set_axisbelow(True)
```

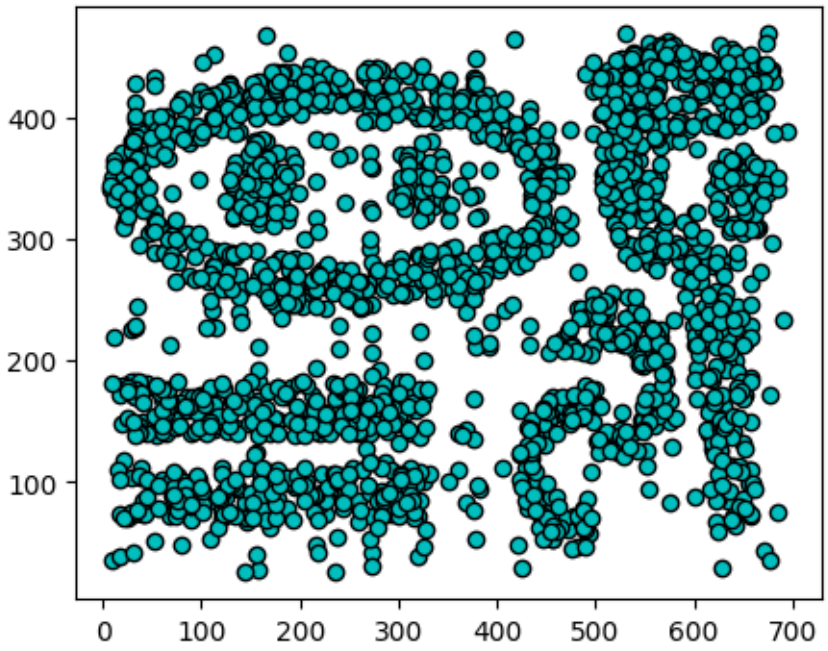


DBSCAN Some clustering algorithms (e.g., DBSCAN) can directly identify outliers as points that do not belong to any cluster.

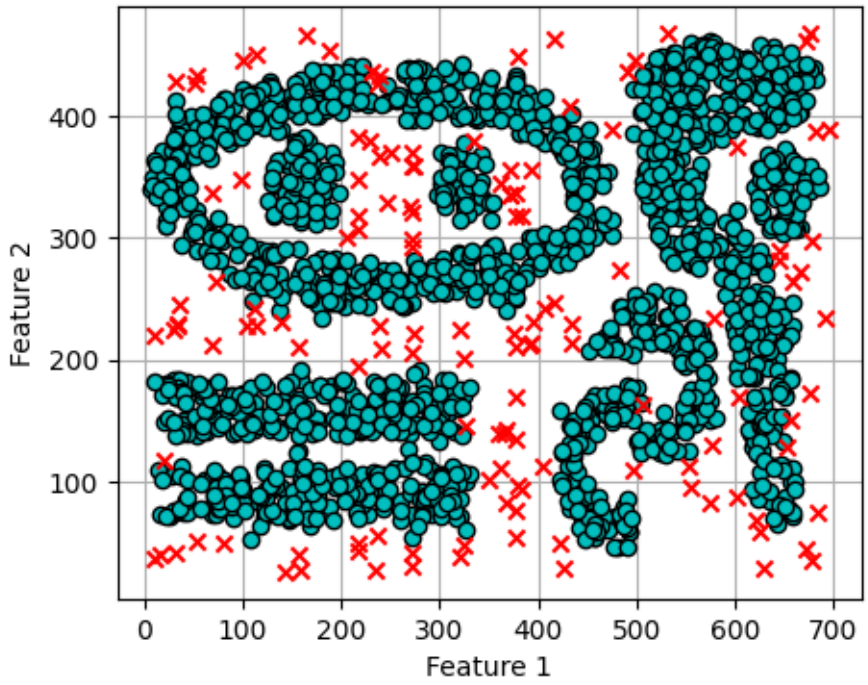
In scikit-learn, these points are labeled with cluster -1.

[14]:
```
from sklearn.cluster import DBSCAN
import pandas as pd

df = pd.read_csv("chameleon.data", sep=" ", header=None, names=['x1', 'x2'])
X = df.values

fig, ax = plt.subplots(figsize=(5, 4))
ax.scatter(X[:, 0], X[:, 1], c='c', edgecolors='k')
```

[14]:
```
<matplotlib.collections.PathCollection at 0x17ab84ad0>
```



[15]:
```
model = DBSCAN(eps=15, min_samples=5)
y_pred = model.fit_predict(X)

fig, ax = plt.subplots(figsize=(5, 4))
ax.scatter(X[y_pred>-1, 0], X[y_pred>-1, 1], c='c', edgecolors='k')
ax.scatter(X[y_pred==-1, 0], X[y_pred==-1, 1], c='r', marker='x')

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.grid()
ax.set_axisbelow(True)
```



(You can try to apply K-means to this dataset. Do you expect the algorithm to work well? How can you make it work to obtain the anomalies?)

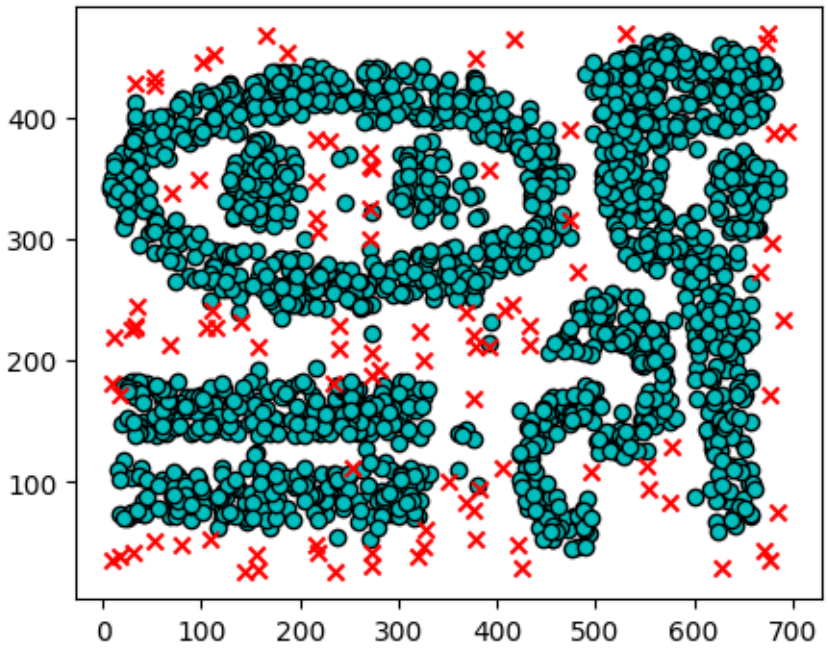
0.3.3 Proximity-based methods

These methods rely on the distance or density of data points. Points that are far from others or in low-density regions are considered anomalies.

Local Outlier Factor (LOF)

[16]:
```
from sklearn.neighbors import LocalOutlierFactor

lof = LocalOutlierFactor(n_neighbors=10, contamination=0.05) # contamination:␣
↪expected fraction of outliers in the data (will be used to define the␣
↪threshold)
y_anomaly = lof.fit_predict(X)

fig, ax = plt.subplots(figsize=(5, 4))
ax.scatter(X[y_anomaly==1, 0], X[y_anomaly==1, 1], c='c', edgecolors='k')
ax.scatter(X[y_anomaly==-1, 0], X[y_anomaly==-1, 1], c='r', marker='x')
```

[16]:
```
<matplotlib.collections.PathCollection at 0x17aded710>
```

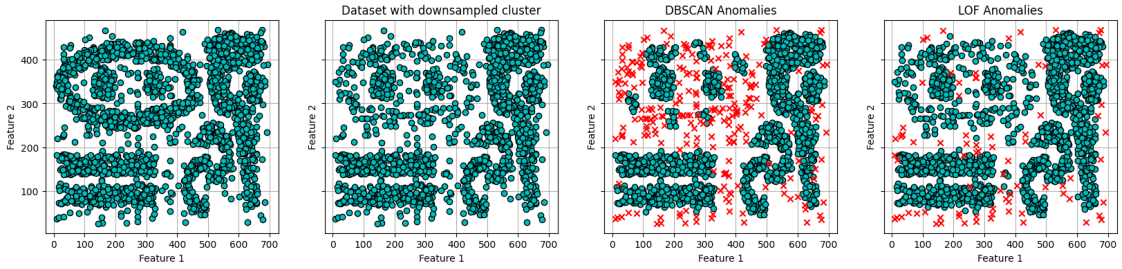


The reults are similar to those obtained with DBSCAN, since the "high density" regions all have approximately the same density. What happens if some region of space becomes less dense?

[17]:
```
# find the biggest cluster (according to DBSCAN), and downsample it (with [::3]␣
↪we are selecting approx. 1/3 of the dataset)

model = DBSCAN(eps=15, min_samples=5)
y_pred = model.fit_predict(X)

clusters, frequencies = np.unique(y_pred, return_counts=True)

biggest_cluster = clusters[frequencies.argmax()]
X_new = np.vstack([X[y_pred != biggest_cluster], X[y_pred == biggest_cluster][::
↪3]])

# now, let's apply DBSCAN once again and LOF to the new dataset (with one␣
↪cluster downsampled) and let's compare the results.

fig, ax = plt.subplots(1, 4, figsize=(20, 4), sharey=True)

ax[0].scatter(X[:, 0], X[:, 1], c='c', edgecolors='k')
ax[0].set_xlabel('Feature 1')
ax[0].set_ylabel('Feature 2')
ax[0].grid()
ax[0].set_axisbelow(True)
```

```
ax[1].scatter(X_new[:, 0], X_new[:, 1], c='c', edgecolors='k')
ax[1].set_xlabel('Feature 1')
ax[1].set_ylabel('Feature 2')
ax[1].grid()
ax[1].set_axisbelow(True)

# DBSCAN
model = DBSCAN(eps=15, min_samples=5)
y_pred = model.fit_predict(X_new)

ax[2].scatter(X_new[y_pred>-1, 0], X_new[y_pred>-1, 1], c='c', edgecolors='k')
ax[2].scatter(X_new[y_pred==-1, 0], X_new[y_pred==-1, 1], c='r', marker='x')
ax[2].set_xlabel('Feature 1')
ax[2].set_ylabel('Feature 2')
ax[2].grid()
ax[2].set_axisbelow(True)

# LOF
lof = LocalOutlierFactor(n_neighbors=10, contamination=0.05) # contamination:␣
 ↪expected fraction of outliers in the data (will be used to define the␣
 ↪threshold)
y_anomaly = lof.fit_predict(X_new)


ax[3].scatter(X_new[y_anomaly==1, 0], X_new[y_anomaly==1, 1], c='c',␣
 ↪edgecolors='k')
ax[3].scatter(X_new[y_anomaly==-1, 0], X_new[y_anomaly==-1, 1], c='r',␣
 ↪marker='x')
ax[3].set_xlabel('Feature 1')
ax[3].set_ylabel('Feature 2')
ax[3].grid()
ax[3].set_axisbelow(True)


ax[1].set_title('Original dataset')
ax[1].set_title('Dataset with downsampled cluster')
ax[2].set_title('DBSCAN Anomalies')
ax[3].set_title('LOF Anomalies')
```

[17]: Text(0.5, 1.0, 'LOF Anomalies')

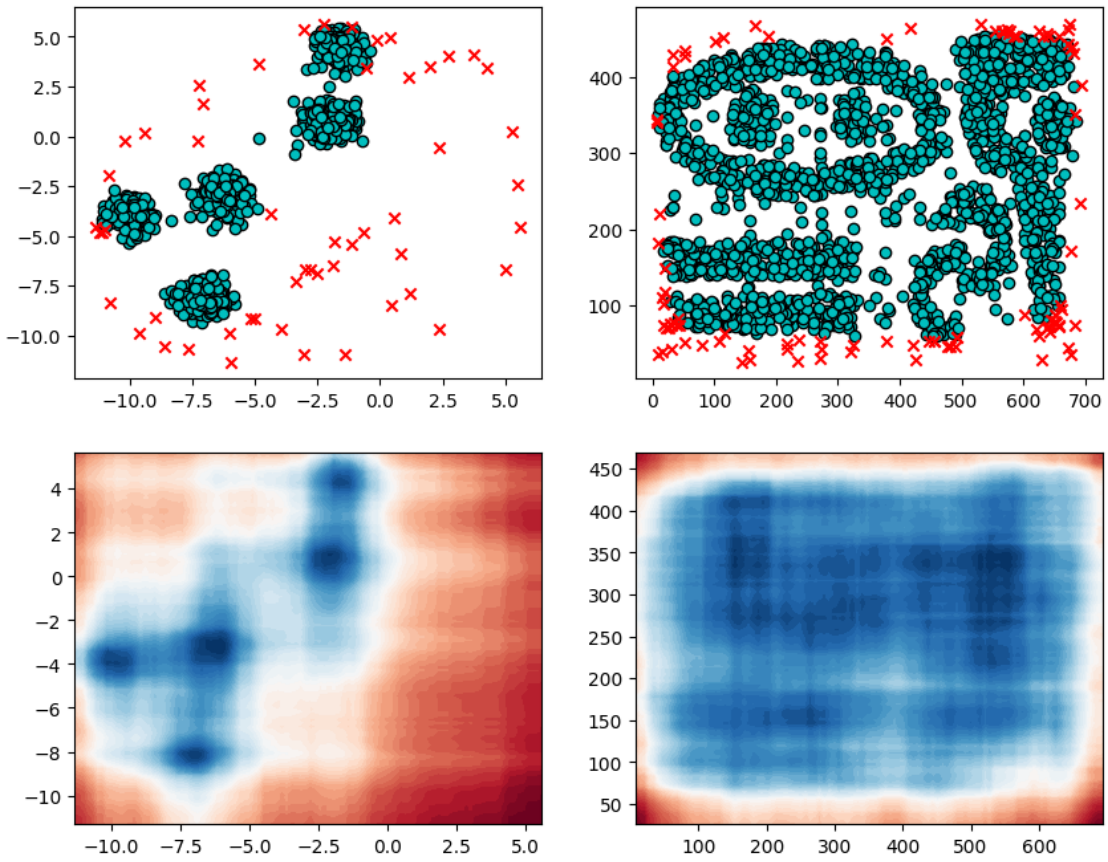


0.3.4 Tree-based methods

Isolation Forests are ensemble methods that isolate anomalies by randomly partitioning the data using random decision trees. Anomalies are easier to isolate and thus have shorter average path lengths in the trees.

[18]:
```
from sklearn.ensemble import IsolationForest


fig, ax = plt.subplots(2, 2, figsize=(10, 8))

isoforest = IsolationForest(n_estimators=100, contamination=0.05, random_state=1)
y_anomaly = isoforest.fit_predict(X_blob)

ax[0, 0].scatter(X_blob[y_anomaly==1, 0], X_blob[y_anomaly==1, 1], c='c',␣
 ↪edgecolors='k')
ax[0, 0].scatter(X_blob[y_anomaly==-1, 0], X_blob[y_anomaly==-1, 1], c='r',␣
 ↪marker='x')


# Heatmap of anomaly scores
ax[1,0].scatter(X_blob[:, 0], X_blob[:, 1], c=isoforest.score_samples(X_blob))

XX, YY = np.meshgrid(np.linspace(X_blob[:,0].min(), X_blob[:,0].max(), 100),
                     np.linspace(X_blob[:,1].min(), X_blob[:,1].max(), 100))
Z = isoforest.score_samples(np.c_[XX.ravel(), YY.ravel()]).reshape(XX.shape)
Z = (Z - Z.min()) / (Z.max() - Z.min())
cm = ax[1,0].contourf(XX, YY, Z, levels=50, cmap='RdBu')


isoforest = IsolationForest(n_estimators=100, contamination=0.05, random_state=1)
y_anomaly = isoforest.fit_predict(X)

ax[0,1].scatter(X[y_anomaly==1, 0], X[y_anomaly==1, 1], c='c', edgecolors='k')
ax[0,1].scatter(X[y_anomaly==-1, 0], X[y_anomaly==-1, 1], c='r', marker='x')
```

```
# Heatmap of anomaly scores
ax[1,1].scatter(X[:, 0], X[:, 1], c=isoforest.score_samples(X))

XX, YY = np.meshgrid(np.linspace(X[:,0].min(), X[:,0].max(), 100),
                     np.linspace(X[:,1].min(), X[:,1].max(), 100))
Z = isoforest.score_samples(np.c_[XX.ravel(), YY.ravel()]).reshape(XX.shape)
Z = (Z - Z.min()) / (Z.max() - Z.min())
cm = ax[1,1].contourf(XX, YY, Z, levels=50, cmap='RdBu')
```



0.3.5 Boundary-based methods

Some approaches (e.g., One-Class SVMs) learn a decision boundary that separates normal data points from anomalies. Points that fall outside this boundary are considered anomalies.

For these approaches, it is important to set the expected fraction of anomalies in the dataset ν, as this influences the shape of the decision boundary (as ν defines how many points can fall outside the learned boundary?)

You can try to apply One-Class SVMs to the previous dataset. Try to change the "kernel" hyperparameter (which defines the shape of the decision boundary). What do the various kernels do?

Next, try to chnge the fraction of anomalies detected. How does it affect the decision boundary?

0.3.6 Neural Network-based methods

A simple approach is to use Autoencoders, which are neural networks trained to reconstruct their input. A Reconstruction Error (RE) can be computed as the "distance" between the input and the reconstructed output (e.g., Mean Squared Error for continuous variables, Binary Cross Entropy for binary variables, etc.).

[19]:
```
import torch
import torch.nn as nn
from torchvision.datasets import MNIST
from torch.utils.data import TensorDataset, DataLoader
```

[20]:
```
dataset = MNIST(root='data', train=True, download=True) # only use training set␣
↪for convenience
X = dataset.data.reshape(-1, 28*28) / 255.0  # normalize to [0, 1] and flatten␣
↪(==> 60000, 784), i.e., treat each image as a vector of 784 features
print(X.shape)
```

```
torch.Size([60000, 784])
```

[21]:
```
class AutoEncoder(nn.Module):
    def __init__(self):
        # Hardcoding all dimensions for simplicity, we technically could pass␣
↪them as parameters
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 256),
            nn.ReLU(),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Linear(64, 10),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(10, 64),
            nn.ReLU(),
            nn.Linear(64, 256),
            nn.ReLU(),
            nn.Linear(256, 784),
            nn.Sigmoid()  # assuming input is normalized between 0 and 1 (as we␣
↪previously did)
```

```
        )

    def forward(self, x):
        code = self.encoder(x)
        x_rec = self.decoder(code)
        return x_rec
```

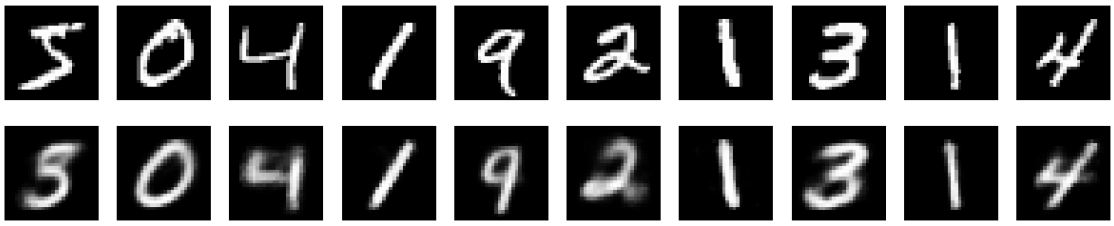
[22]:
```
model = AutoEncoder()
opt = torch.optim.Adam(model.parameters(), lr=1e-2)
loss_fn = nn.MSELoss()
ds = TensorDataset(X)
dl = DataLoader(ds, batch_size=256, shuffle=True)

n_epochs = 10

for epoch in range(n_epochs):
    epoch_loss = 0.0
    for x_batch, in dl:
        opt.zero_grad()
        batch_rec = model(x_batch)
        loss = loss_fn(batch_rec, x_batch)
        loss.backward()
        opt.step()
        epoch_loss += loss.item()
    epoch_loss /= len(dl)
    print(f"Epoch {epoch+1}/{n_epochs}, Loss: {epoch_loss:.6f}")
```

```
Epoch 1/10, Loss: 0.055307
Epoch 2/10, Loss: 0.039010
Epoch 3/10, Loss: 0.034029
Epoch 4/10, Loss: 0.032201
Epoch 5/10, Loss: 0.031260
Epoch 6/10, Loss: 0.030655
Epoch 7/10, Loss: 0.030282
Epoch 8/10, Loss: 0.029869
Epoch 9/10, Loss: 0.029597
Epoch 10/10, Loss: 0.029360
```

[23]:
```
with torch.no_grad():
    X_rec = model(X).numpy()

X_np = X.numpy()
```

[24]:
```
fig, ax = plt.subplots(2, 10, figsize=(15, 3))


for i in range(10):
    ax[0, i].imshow(X_np[i].reshape(28, 28), cmap='gray')
```

```
    ax[0, i].set_axis_off()

    ax[1, i].imshow(X_rec[i].reshape(28, 28), cmap='gray')
    ax[1, i].set_axis_off()
```



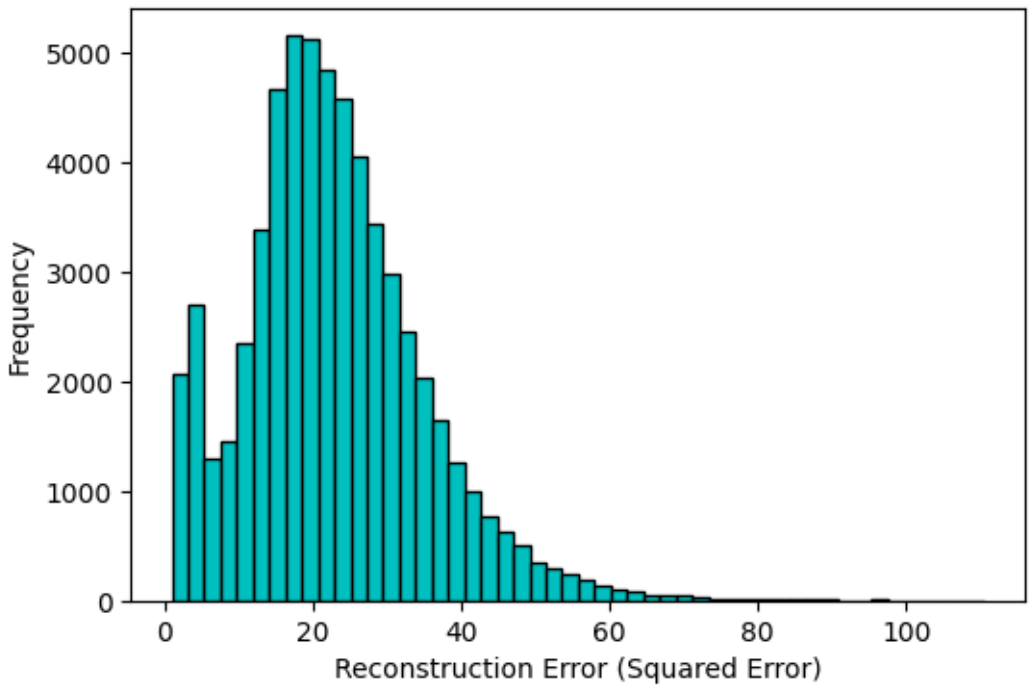
The RE (for instance, squared error across all pixels, in this case) can be used to compute how "poorly" each sample is reconstructed by the Autoencoder. High RE values can be used to flag anomalies in the dataset.

Let's look at the worst-reconstructed images according to the MSE RE.

[25]:
```
re = ((X_np - X_rec) ** 2).sum(axis=1)  # Mean Squared Error per sample
```

[26]:
```
fig, ax = plt.subplots(figsize=(6,4))
ax.hist(re, bins=50, color='c', edgecolor='k')
ax.set_ylabel('Frequency')
ax.set_xlabel('Reconstruction Error (Squared Error)')
```

[26]:
```
Text(0.5, 0, 'Reconstruction Error (Squared Error)')
```



[27]:
```
fig, ax = plt.subplots(2, 10, figsize=(15, 3))


for i, pos in enumerate(re.argsort()[-10:]):
    ax[0, i].imshow(X_np[pos].reshape(28, 28), cmap='gray')
    ax[0, i].set_axis_off()

    ax[1, i].imshow(X_rec[pos].reshape(28, 28), cmap='gray')
    ax[1, i].set_axis_off()
```