A1 - Data Mining - Local Outlier Detection (LOF)

In this assignment i will be implementing the Local Outlier Detection algorithm in python and collect the Local Outlier Factor (LOF) for all points in the given dataset. For validation of the algorithm implementation i will use a built-in solution for the same problem via a python library.

In [181…
```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.neighbors import LocalOutlierFactor

# read the data
data = pd.read_csv('data/artificial_dataset.csv') # can be changed to: cl
```

For this assignment, i tried following the logic of the algorithm as shown in the slides as closely as possible with all of the different steps as an outline for my code.

The first step is simple, we just want to calculate the euclidean distance between all of the data points. In the end i just print the dimensions of the matrix because since we have 19 samples in the dataset, i expect us to get a 19x19 matrix, which we did.

In [182…
```
# Step 1: calculate pairwise distance matrix manually

num_samples = data.shape[0]
distance_matrix = np.zeros((num_samples, num_samples))
for i in range(num_samples):
    for j in range(num_samples):
        distance_matrix[i, j] = np.linalg.norm(data.iloc[i] - data.iloc[j

# print(distance_matrix)
# Print matrix dimensions
print("Distance matrix shape:", distance_matrix.shape)
```

```
Distance matrix shape: (19, 19)
```

Then, i will take each row of the matrix and sort the elements for each row. Then i will take the k-th value in each row and then exclude everything else. I do that by just putting it in a new list so i can use it further in the algorithm.

In [183…
```
# Step 2: Choose k and find k-distance

k = 3 # Change k value here if needed

k_distance = np.zeros(num_samples)
for i in range(num_samples):
    sorted_distances = np.sort(distance_matrix[i])
    k_distance[i] = sorted_distances[k]  # k-th nearest neighbor distance

print(k_distance)
```

```
[1.26400725 0.85339749 1.27157149 0.4776837  0.59034546 1.31952402
 0.5327643  0.83583394 0.5327643  0.94839886 0.68782789 1.31542968
 0.88983426 0.79641614 0.66023746 0.5551712  0.71139371 0.88983426
 0.59935623]
```

Now we calculate the ARD to see how "reachable" a given point is to its neighborhood. In dense areas neighbors have small k-distances so we will use that distance but in sparser areas the actual distances are larger so we use actual distance and we will then take the average of this mix of values to get an average of all distances to the k-nearest neighbors.

So relatively, a lower ARD should mean that a point is in a dense neighborhood and on the other hand a higher ARD means its a sparse region.

In [184…

```
# Step 3: Define the Average Reachability Distance (ARD) of each point

def average_reachability_distance(point_index, neighbors):
    reachability_distances = []
    for neighbor in neighbors:
        reachability_distance = max(k_distance[neighbor], distance_matrix
        reachability_distances.append(reachability_distance)
    return np.mean(reachability_distances)

# Do it but not in a function
average_reachability_distances = np.zeros(num_samples)
for i in range(num_samples):
    # Find k nearest neighbors
    neighbors = np.argsort(distance_matrix[i])[:k+1]  # +1 to include the
    neighbors = neighbors[neighbors != i]  # Exclude the point itself
    average_reachability_distances[i] = average_reachability_distance(i,

print(average_reachability_distances)
```

```
[1.20378106 0.84528395 1.04919658 0.58445216 0.55751358 1.0984628
 0.60228805 0.77285651 0.53359782 0.85128635 0.57356093 1.30711545
 0.98543368 0.64744812 0.60492163 0.64914053 0.81110303 0.81820849
 0.60492163]
```

Then we calculate the LARD which is similar to ARD but it checks the reachability within its own neighborhood instead of on a global scale for the entire dataset.

In [185…

```
# step 3 (cont): Define the Local Average Reachability Distance (LARD) of
local_average_reachability_distances = np.zeros(num_samples)
for i in range(num_samples):
    # Find k nearest neighbors (excluding the point itself)
    neighbors = np.argsort(distance_matrix[i])[:k+1]  # +1 to include the
    neighbors = neighbors[neighbors != i]  # Exclude the point itself
    local_average_reachability_distances[i] = np.mean(average_reachabilit

print(local_average_reachability_distances)
```

```
[0.82486516 0.74474438 0.8170301  0.5698156  0.57344601 0.70034084
 0.58849937 0.6489963  0.58141793 0.72096978 0.61283811 0.88297537
 0.81728575 0.57344601 0.6187834  0.59446806 0.70822094 0.88060689
 0.59588666]
```

Finally, we can calculate the LOF which is a ratio between the ARD/LARD.

In [186…

```
# Step 4: Define Local Outlier Factor (LOF) for each point
local_outlier_factors = np.zeros(num_samples)
for i in range(num_samples):
    if local_average_reachability_distances[i] == 0:
        local_outlier_factors[i] = 0  # Avoid division by zero
    else:
        local_outlier_factors[i] = average_reachability_distances[i] / lo
print("Local Outlier Factors:")
for i in range(num_samples):
    print(f"LOF[{i}] = {local_outlier_factors[i]}")
```

```
Local Outlier Factors:
LOF[0] = 1.4593670820006248
LOF[1] = 1.1349987558779633
LOF[2] = 1.2841590393871694
LOF[3] = 1.025686494771743
LOF[4] = 0.9722163429044278
LOF[5] = 1.5684688587217182
LOF[6] = 1.0234302359082514
LOF[7] = 1.190848863248375
LOF[8] = 0.9177526042390783
LOF[9] = 1.1807517823940012
LOF[10] = 0.9359093742032981
LOF[11] = 1.4803532324597777
LOF[12] = 1.2057394640508463
LOF[13] = 1.1290480835999477
LOF[14] = 0.9775983458348672
LOF[15] = 1.091968714310732
LOF[16] = 1.1452683591858566
LOF[17] = 0.9291415979636373
LOF[18] = 1.0151622296770906
```

In the end for validation i use scikit-learns implementation of the same algorithm and then take the difference between the two solutions. From eyeballing it, i see that i am usually 2-3 decimals off from the scikit solution which i am pretty happy with. I think this comes down to what happens under the hood in scikit learns implementation. They might have their own optimizations in one or more of the steps which gives different final results.

For all of the datasets in this folder, the results are fairly consistents which further validates for me that the implementation is correct.

In [187…

```
# Calculate LOF using scikit-learn for validation

lof = LocalOutlierFactor(n_neighbors=k)
y_pred = lof.fit_predict(data)
lof_scores = -lof.negative_outlier_factor_
print("LOF scores from scikit-learn:")
for i in range(num_samples):
    print(f"LOF[{i}] = {lof_scores[i]}")
print("\n")

# Compare the two LOF scores
print("Difference between manual LOF and scikit-learn LOF:")
print(local_outlier_factors - lof_scores)
```

```
LOF scores from scikit-learn:
LOF[0] = 1.4598276095131641
LOF[1] = 1.1575256624653327
LOF[2] = 1.322508555580498
LOF[3] = 1.0282266304891439
LOF[4] = 0.9747957310865116
LOF[5] = 1.6035921174242127
LOF[6] = 1.0298322428460047
LOF[7] = 1.2281449186838025
LOF[8] = 0.9186806784470107
LOF[9] = 1.2284465013325174
LOF[10] = 0.9376974041923649
LOF[11] = 1.4901109244953854
LOF[12] = 1.2765625255563475
LOF[13] = 1.1320435622068186
LOF[14] = 0.9787511613882703
LOF[15] = 1.0926565591853266
LOF[16] = 1.1869558094739008
LOF[17] = 0.9356773297867095
LOF[18] = 1.0218489030440994


Difference between manual LOF and scikit-learn LOF:
[-0.00046053 -0.02252691 -0.03834952 -0.00254014 -0.00257939 -0.03512326
 -0.00640201 -0.03729606 -0.00092807 -0.04769472 -0.00178803 -0.00975769
 -0.07082306 -0.00299548 -0.00115282 -0.00068784 -0.04168745 -0.00653573
 -0.00668667]
```