COMP4027 Lab Report

HU Xinyue (19250797)
Department of Computer Science
Hong Kong Baptist University

1. Task and Datasets

The mini project focus on the **Outlier Detection** task which is one of the most important topics in this course as well as the fundamental data mining. I use **LOF** (local outlier factor, in density-based model) as a key indicator to analysis the real-world data sets.

I found the two awesome data sets, both from "Unsupervised Anomaly Detection Dataverse" at Harvard (https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/OPQMVF), and they are originally created in the UCI Machine Learning Repository.

Data set 1 (pen-global-unsupervised): The whole data set contains pen-local-unsupervised and pen-global-unsupervised, but pen-local-unsupervised is not included in this project. I intended to find a relatively small dataset, and there are in total **16 attributes** and **800+ rows** which is a perfect match. All input attributes are integers ranging from 0 to 100. The meaning of each attribute could be somewhat ambiguous because they did some pre-processing on the coordinate information and represented digits as constant length feature vectors. Although the algorithm I have can handle 16 attributes, it takes longer time to run the program and hard to do visualization, so I decided to just choose the 3 dimensions among 16. I tried different combinations and use the one that seems would have obvious outliers.

Data set 2 (Shuttle): The shuttle dataset contains **9 attributes** all of which are numerical and there are in total **46000+ rows** which is a super large dataset. Very cool part of this data set is that it is provided by NASA. Consider the huge size of the dataset, it is very difficult for my computer to find all the outliers, but we can input most data and input additional data for individual **query**.

2. Algorithm and Code

A simple function that calculates **Euclidean distance** between two points, it first checks whether the instances have the same length, if not then error will be raised.

```python
from __future__ import division
import warnings
```

```python
def distance_euclidean(instance1, instance2):
    def detect_value_type(attribute):
        from numbers import Number
        attribute_type = None
        if isinstance(attribute, Number):
            attribute_type = float
            attribute = float(attribute)
        else:
            attribute_type = str
            attribute = str(attribute)
        return attribute_type, attribute
    # check if instances are of same length
    if len(instance1) != len(instance2):
        raise AttributeError("Instances have different number of arguments.")
    differences = [0] * len(instance1)
    for i, (attr1, attr2) in enumerate(zip(instance1, instance2)):
        type1, attr1 = detect_value_type(attr1)
        type2, attr2 = detect_value_type(attr2)
        if type1 != type2:
            raise AttributeError("Instances have different data types.")
        if type1 is float:
            differences[i] = attr1 - attr2
        else:
            if attr1 == attr2:
                differences[i] = 0
            else:
                differences[i] = 1
    rmse = (sum(map(lambda x: x**2, differences)) / len(differences))**0.5
    return rmse
```

The **k-distance ε** is expressed as "*k_distance_value*" in the function "*k_distance*", also the k nearest neighbors will be returned.

The **reachability distance** between two points is expressed in the function "*reachability_distance*", this function basically invokes the previous function to get the k-distance and compare it with the Euclidean distance and select the max.

- **K-distance:**
  - Given an integer k and a point o, the k-distance of o is defined as k-distance(o) = ε,
    - where $N_k(o)$ is defined to be the ε-neighborhood of o (**excluding point o**)
    - and ε is the distance between o and the k-th nearest neighbor

- **Reachability distance** of p with respect to o
  - Given two points p and o and an integer k,
    - $\text{Reach\_dist}_k(p, o)$ is defined to be max{dist(p, o), k-distance(o)}

```python
def k_distance(k, instance, instances, distance_function=distance_euclidean):
    distances = {}
    for instance2 in instances:
        distance_value = distance_function(instance, instance2)
        if distance_value in distances:
            distances[distance_value].append(instance2)
        else:
            distances[distance_value] = [instance2]
    distances = sorted(distances.items())
    neighbours = []
    [neighbours.extend(n[1]) for n in distances[:k]]
    k_distance_value = distances[k - 1][0] if len(distances) >= k else distances[-1][0]
    return k_distance_value, neighbours
```

```python
def reachability_distance(k, instance1, instance2, instances, distance_function=distance_euclidean):
    (k_distance_value, neighbours) = k_distance(k, instance2, instances, distance_function=distance_function)
    return max([k_distance_value, distance_function(instance1, instance2)])
```

- The **local outlier factor (LOF)** of a point p equals to

$$LOF_k(p) = \frac{\sum_{o \in N_k(p)} \frac{lrd(o)}{lrd(p)}}{|N_k(p)|}$$

- **Local reachability density** of p (denoted by $lrd_k(p)$) is defined:

$$lrd_k(p) = \frac{1}{\frac{\sum_{o \in N_k(p)} reach\_dist_k(p,o)}{|N_k(p)|}}$$

Local reachability density is calculated as (neighbors' length) / (sum all the reachability distance), referring to "*local_reachability_density*".

Finally, the **LOF** can be calculated by making use of the local reachability density: (sum up all the ratios (lrd_o/lrd_p) array) / neighbors' length, and this is implemented in "*local_outlier_factor*".

```python
def local_reachability_density(min_pts, instance, instances, **kwargs):
    (k_distance_value, neighbours) = k_distance(min_pts, instance, instances, **kwargs)
    reachability_distances_array = [0]*len(neighbours) #n.zeros(len(neighbours))
    for i, neighbour in enumerate(neighbours):
        reachability_distances_array[i] = reachability_distance(min_pts, instance, neighbour, instances, **kwargs)
    if not any(reachability_distances_array):
        warnings.warn("Instance %s (could be normalized) is identical to all the neighbors. Setting local reachability
        return float("inf")
    else:
        return len(neighbours) / sum(reachability_distances_array)
```

```python
def local_outlier_factor(min_pts, instance, instances, **kwargs):
    (k_distance_value, neighbours) = k_distance(min_pts, instance, instances, **kwargs)
    instance_lrd = local_reachability_density(min_pts, instance, instances, **kwargs)
    lrd_ratios_array = [0]* len(neighbours)
    for i, neighbour in enumerate(neighbours):
        instances_without_instance = set(instances)
        instances_without_instance.discard(neighbour)
        neighbour_lrd = local_reachability_density(min_pts, neighbour, instances_without_instance, **kwargs)
        lrd_ratios_array[i] = neighbour_lrd / instance_lrd
    return sum(lrd_ratios_array) / len(neighbours)
```

The function "*outliers*" will be the one that the user/client call and will start the whole things. We classify a point to be an outlier if the LOF value is greater than 1.

```
def outliers(k, instances, **kwargs):
    instances_value_backup = instances
    outliers = []
    for i, instance in enumerate(instances_value_backup):
        instances = list(instances_value_backup)
        instances.remove(instance)
        l = LOF(instances, **kwargs)
        value = l.local_outlier_factor(k, instance)
        if value > 1:
            outliers.append({"lof": value, "instance": instance, "index": i})
    outliers.sort(key=lambda o: o["lof"], reverse=True)
    return outliers
```

This helper class *LOF* will be used to perform computations and do **normalization** automatically.

```
class LOF:

    def __init__(self, instances, normalize=True, distance_function=distance_euclidean):
        self.instances = instances
        self.normalize = normalize
        self.distance_function = distance_function
        if normalize:
            self.normalize_instances()

    def compute_instance_attribute_bounds(self):
        min_values = [float("inf")] * len(self.instances[0]) #n.ones(len(self.instances[0])) * n.inf
        max_values = [float("-inf")] * len(self.instances[0]) #n.ones(len(self.instances[0])) * -1 * n.inf
        for instance in self.instances:
            min_values = tuple(map(lambda x,y: min(x,y), min_values,instance)) #n.minimum(min_values, instance)
            max_values = tuple(map(lambda x,y: max(x,y), max_values,instance)) #n.maximum(max_values, instance)

        diff = [dim_max - dim_min for dim_max, dim_min in zip(max_values, min_values)]
        if not all(diff):
            problematic_dimensions = ", ".join(str(i+1) for i, v in enumerate(diff) if v == 0)
            warnings.warn("No data variation in dimensions: %s. You should check your data or disable normalization." %

        self.max_attribute_values = max_values
        self.min_attribute_values = min_values

    def normalize_instances(self):
        if not hasattr(self, "max_attribute_values"):
            self.compute_instance_attribute_bounds()
        new_instances = []
        for instance in self.instances:
            new_instances.append(self.normalize_instance(instance)) # (instance - min_values) / (max_values - min_value
        self.instances = new_instances
```

```
    def normalize_instance(self, instance):
        return tuple(map(lambda value,max,min: (value-min)/(max-min) if max-min > 0 else 0,
                         instance, self.max_attribute_values, self.min_attribute_values))

    def local_outlier_factor(self, min_pts, instance):
        if self.normalize:
            instance = self.normalize_instance(instance)
        return local_outlier_factor(min_pts, instance, self.instances, distance_function=self.distance_function)
```

3. Experiment and Result

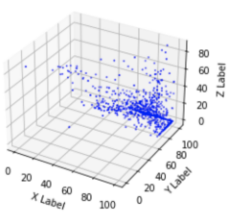
Ex1 using Dataset1:

Before running outlier detection algorithm, we can do an initial observation on data.

```
from matplotlib import pyplot as plt

(b,d,j) = zip(*instances)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(b, d, j, color="#0000FF", s=1)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```



The dataset (I choose b, d, j column among 16 attributes) to be run is plotted as above for visualization, there is some outliers that is visible to the naked eye.

Then I start the algorithm, the top-10 outliers after the experiment are pasted below, to save some space, other outliers would not be pasted here but can be seen from later visualization. I also choose 10 as the k value because I think that would be reasonable.

In [71]:
```
from matplotlib import pyplot as plt
import pandas as pd

data = pd.read_csv (r'Downloads/pen-global-unsupervised-ad.csv')
raw_df = pd.DataFrame(data, columns= ['B','D','J'])
instances = raw_df.apply(lambda x: tuple(x), axis=1).values.tolist()

# (b, d, j) = zip(*instances)
# fig = plt.figure()
# ax = fig.add_subplot(projection='3d')
# ax.scatter(b, d, j, color="#0000FF", s=1)
# ax.set_xlabel('X Label')
# ax.set_ylabel('Y Label')
# ax.set_zlabel('Z Label')

lof = outliers(10, instances)

for outlier in lof:
    print (outlier["lof"],outlier["instance"])
```

```
3.8218037767022977 (52, 0, 37)
3.4994397347310295 (0, 15, 79)
2.6720749923888203 (60, 83, 0)
2.477527921419088 (66, 89, 0)
1.8934387120356164 (46, 64, 75)
1.850540358696421 (38, 29, 57)
1.7601450306566853 (61, 20, 64)
1.6448198667604874 (79, 92, 0)
1.6378953357160324 (100, 41, 43)
1.6118514973982627 (45, 36, 38)
```

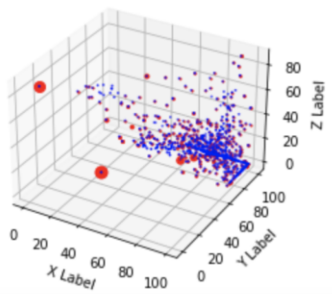
We first plot all the points using blue color in the 3D-space just like the beginning, and then we loop though all the outlier we detect and cover the blue points (outliers cover themselves) with larger red points (might also see a small blue dot inside). **The larger the LOF value, the bigger the point is.** From the 3D-plotting, I suppose the algorithm results perfectly make sense.

```
from matplotlib import pyplot as plt

(b,d,j) = zip(*instances)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(b, d, j, color="#0000FF", s=1)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

for outlier in lof:
    value = outlier["lof"]
    instance = outlier["instance"]
    color = "#FF0000" if value > 1 else "#00FF00"
    ax.scatter(instance[0], instance[1], instance[2], color=color, s=(value-1)**2*10+1)

plt.show()
```



Ex2 using Dataset2:

For the huge one, import the first 40000 rows and 9 columns.

```
In [89]: import pandas as pd

         data = pd.read_csv (r'Downloads/shuttle.csv')

         raw_df = pd.DataFrame(data, columns= ['V1','V2','V3','V4','V5','V6','V7','V8','V9'])

         df = raw_df.head(40000).apply(lambda x: tuple(x), axis=1).values.tolist()

         instances = df

         print (instances)
```

```
0, 34, 34, 0), (55, 0, 81, 0, 54, -10, 25, 26, 2), (37, 0, 100, 0, 34, 6, 64, 67, 4), (43, -3, 86, 2, 44, 14, 43, 42,
0), (46, -5, 84, 0, 46, 0, 38, 37, 0), (45, 0, 81, 0, 44, 0, 36, 37, 0), (48, 0, 86, 0, 50, 30, 38, 37, 0), (45, 5, 9
0, 0, 44, 0, 44, 46, 2), (37, 0, 77, 0, -20, 27, 41, 98, 58), (37, 0, 90, -6, 18, 0, 53, 72, 20), (37, -8, 77, 0, 20,
0, 40, 57, 16), (50, 5, 95, -3, 50, 0, 45, 46, 2), (43, -2, 76, 0, 44, 15, 32, 32, 0), (42, -1, 85, -5, 42, 0, 44, 4
4, 0), (37, 0, 76, 0, 26, 20, 39, 50, 12), (37, 0, 76, 8, 30, 0, 40, 45, 6), (45, 0, 106, 0, 46, 9, 61, 60, 0), (37,
0, 81, 0, 38, 17, 44, 43, 0), (49, 4, 83, 0, 46, 0, 34, 37, 2), (50, 0, 88, 1, 50, 0, 38, 39, 0), (42, 2, 77, 0, 42,
0, 36, 36, 0), (49, 0, 78, 0, 50, 11, 29, 29, 0), (37, 0, 77, 0, 34, -12, 40, 43, 2), (37, 3, 81, 0, 36, 0, 44, 44,
0), (37, 0, 84, 0, 26, -3, 47, 58, 10), (53, 0, 86, 0, 54, 0, 33, 32, 0), (44, 0, 81, -3, 44, 17, 37, 37, 0), (46, 0,
81, 7, 46, 3, 35, 34, 0), (45, -4, 83, 0, 44, -14, 37, 39, 2), (55, -5, 87, 2, 54, 0, 32, 33, 0), (45, 0, 89, -3, 44,
0, 44, 45, 0), (51, 0, 86, 0, 52, 22, 35, 34, 0), (53, 0, 87, -6, 52, -7, 34, 35, 2), (56, 0, 86, 0, 56, 2, 30, 30,
0), (49, 0, 87, 0, 50, 1, 38, 38, 0), (56, 0, 96, 0, 56, 10, 40, 39, 0), (37, 0, 95, 0, 18, -24, 57, 77, 20), (48, 0,
86, -4, 46, 0, 39, 40, 2), (41, 5, 93, 0, 38, 0, 53, 55, 2), (56, 0, 85, 0, 56, 15, 29, 28, 0), (53, 0, 88, 0, 52, -
4, 35, 37, 2), (37, 0, 78, 0, -6, -5, 42, 86, 44), (41, 5, 80, 0, 42, 7, 39, 39, 0), (45, 2, 108, 0, 44, 0, 63, 64,
2), (48, 2, 89, 0, 46, 0, 41, 42, 2), (49, 0, 85, 0, 46, -5, 36, 39, 2), (44, 0, 76, 2, 42, -29, 32, 34, 2), (49, 0,
95, 4, 50, 0, 46, 46, 0), (51, 0, 88, 0, 52, 8, 36, 36, 0), (49, 3, 95, 0, 46, -9, 47, 49, 2), (40, 0, 84, 6, 38, 0,
44, 45, 2), (59, 0, 86, -2, 60, 18, 27, 26, 0), (37, 0, 108, 15, 26, -1, 71, 82, 12), (37, 0, 104, 0, 28, 1, 66, 75,
8), (55, 1, 83, 0, 54, 0, 28, 29, 0), (43, 0, 97, 0, 42, -9, 53, 55, 2), (45, -1, 83, 0, 44, 0, 38, 39, 0), (37, 0, 7
6, -7, 20, -6, 39, 55, 16), (38, 0, 81, 0, 36, -30, 43, 45, 2), (37, 0, 77, 0, 30, -9, 40, 46, 6), (37, 0, 102, 0, 3
6, 0, 64, 66, 2), (52, 5, 78, -1, 52, 0, 26, 26, 0), (43, 0, 88, 0, 42, -11, 45, 47, 2), (42, 0, 86, 1, 42, 0, 45, 4
```

One good thing about this dataset is that the last (tenth) column is about the outlier label! We know which one is the outlier in advance. I rank the rows to make sure that all the first 40000 rows are not outliers. Below are some examples.

23412	37	0	75	0	28	-1	38	46	8	n
23413	37	0	81	0	30	14	45	50	6	n
23414	42	4	76	0	42	0	33	34	2	n
23415	37	0	76	0	28	30	39	48	8	n
23416	45	-1	86	-5	44	-11	40	42	2	n
23417	37	0	79	0	8	-21	43	72	28	n
23418	56	0	87	2	56	6	31	30	0	n
23419	38	-4	76	-1	38	0	38	37	0	n
23420	51	0	79	0	52	8	27	27	0	n
23421	38	1	80	0	38	14	42	41	0	n
23422	37	-3	79	0	18	0	42	61	18	n
23423	59	2	84	0	56	-7	25	27	2	n
23424	51	0	86	-3	52	0	35	34	0	n
23425	53	-3	82	0	52	-21	29	30	2	n
23426	49	0	81	1	46	-7	33	35	2	n
23427	40	2	81	-5	38	0	41	42	2	n
23428	55	0	82	1	54	0	27	28	0	n
23429	37	0	95	2	16	0	58	80	22	n
23430	55	0	83	0	56	22	27	26	0	n
23431	37	0	76	0	34	16	39	43	4	n
23432	41	0	76	0	38	-12	35	37	2	n
23433	37	2	77	0	34	-24	40	43	4	n
23434	42	-1	83	0	42	0	41	41	0	n
23435	58	-5	88	1	56	0	31	31	0	n

45198	49	0	106	-5	46	-13	58	60	2	n
45199	49	5	77	0	46	-10	28	30	2	n
45200	37	0	83	0	28	-4	46	54	8	n
45201	45	0	86	2	44	0	42	42	0	n
45202	44	-1	82	0	44	9	38	38	0	n
45203	37	0	77	0	36	3	39	41	2	n
45204	41	23	78	0	38	-99	37	39	2	n
45205	49	0	87	0	50	21	38	38	0	n
45206	37	-29	106	-2	34	-4	69	72	4	n
45207	44	0	84	0	44	5	40	41	0	n
45208	41	0	76	0	42	23	35	35	0	n
45209	51	0	86	0	50	-9	36	37	2	n
45210	52	-5	81	0	52	-7	29	30	0	n
45211	55	2	96	0	54	0	41	42	2	n
45212	49	0	87	0	50	8	38	38	0	n
45213	37	0	90	0	36	9	53	54	2	n
45214	37	0	80	0	6	0	43	75	32	n
45215	47	-3	79	0	46	0	31	32	0	n
45216	38	4	105	0	38	0	67	66	0	n
45217	37	0	94	0	8	-19	57	86	30	n
45218	43	0	84	5	42	0	42	43	2	n
45219	46	0	85	-6	44	-25	39	41	2	n
45220	53	0	80	0	54	0	27	26	0	n
45221	37	0	77	0	-14	-2	40	92	52	n
45222	37	0	83	0	12	-20	46	70	24	n

Then we can do some queries and see if the result we get is consistent with the actual label.

I randomly choose two points that are originally labeled as n (non-outlier) to be test points (not choosing from the first 40000), and randomly pick a point labeled as o (outlier).

```
In [92]: for instance in [[44,3,77,0,44,14,34,34,0],[37,-2,94,0,34,0,57,61,4],[82,0,87,7,-40,1,5,128,124]]:
             value = lof.local_outlier_factor(10, instance)
             print (value, instance)

         1.9479614927533908 [44, 3, 77, 0, 44, 14, 34, 34, 0]
         0.9990469042908572 [37, -2, 94, 0, 34, 0, 57, 61, 4]
         61.06729264394447 [82, 0, 87, 7, -40, 1, 5, 128, 124]
```

Interestingly, the first one is being misclassified (inconsistent with the label) while other two are correct, I think there are several reasons which may cause this issue:

1. parameter choosing: k=10 might be too small for a 40000+ rows dataset, k=20 should probably be a better choice.
2. only three testing points: this small number will increase the randomness of the experiment. We should have more testing points.
3. the official label may not base on LOF, and it include all points (not just first 40000).

But the single query is also time consuming, so I did not have enough time for the enhancement.