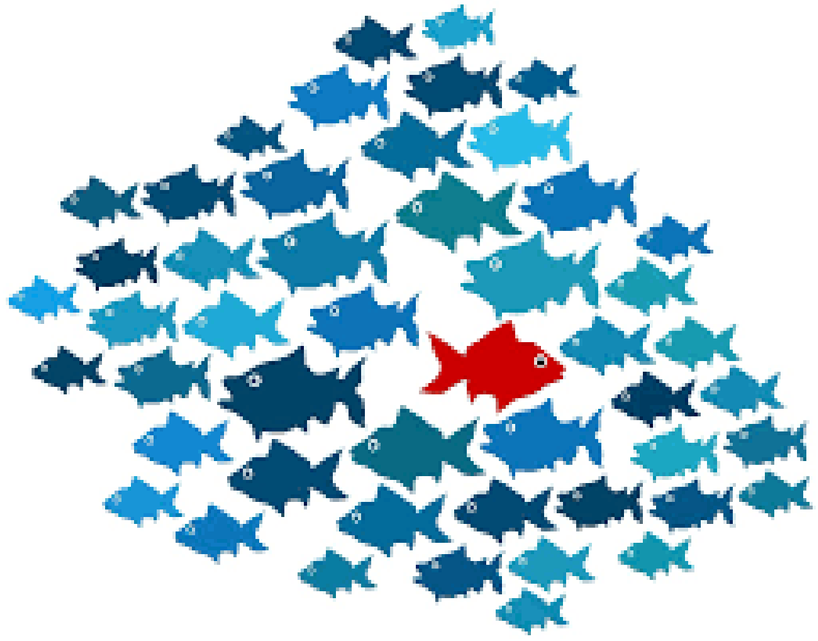
Anomaly detection - Part 3

One should look for what is and not what he thinks should be. -Albert Einstein

Welcome back



Before we get started, let’s take moment to remind ourselves why outliers matter and discover two methods for uncovering them

https://towardsdatascience.com/a-brief-overview-of-outlier-detection-techniques-1e0b2c19e561

Common anomaly detection methods

- We've already covered the clustering method DBSCAN and implemented decision trees on anomalous data to catch outliers
- In this module, we will cover common anomaly detection methods, including the popular **local outlier factor (LOF)** algorithm

Module completion checklist

Objective	Complete
Discuss anomaly detection methods	
Describe local outlier factor algorithm	
Implement LOF	
Optimize LOF by tuning its hyperparameters	

Anomaly detection methods

- Aside from conventional classification models, there are special statistical models to detect anomalies:
  - **Gaussian model**: checks the distribution of the data and models any data point that falls outside the normal distribution
  - **Isolation forest**: a tree-based approach, which we will learn more about later

Density-based anomaly detection

- Density-based algorithms are mainly based on distance
- The general assumption is regular data points occur around a dense region and abnormalities are far away
- There are two algorithms for density-based detection:
  - **K nearest neighbors**, which is based on the fact that regular data points are closer to each other than the anomalous data points and uses distance measures like Euclidean to find the distance between data points
  - **Local outlier factor**, which is based on relative density of data, or “reachability distance”

Clustering-based anomaly detection

- Clustering is one of the most popular concepts in the area of anomaly detection
- It is based on the assumption that data points that are similar tend to belong to similar groups or cluster determined by the local centroid
- k-means is the widely used clustering algorithm, but DBSCAN is most commonly used for anomaly detection

Support vector-machine based anomaly detection

- There is a special SVM algorithm for anomaly detection called **one class SVM**
- This one class SVM is trained only on normal data
- The algorithm learns a soft boundary in order to cluster the normal data instances using training data
- When it sees the testing instance, it tunes itself to identify the abnormalities that fall outside the learned region

Choosing an anomaly detection technique

- We've seen many anomaly detection techniques thus far; which one we use depends on the use case
- We choose any technique useful for the data based on domain expertise
- In this module, we will build a LOF model on the Paysim dataset

Module completion checklist

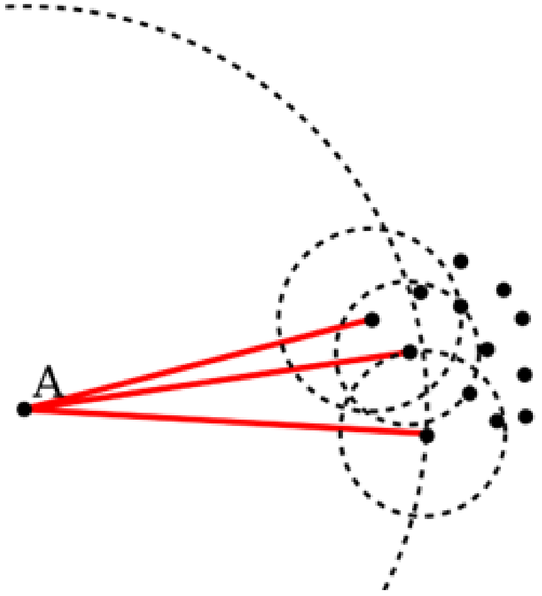
Objective	Complete
Discuss anomaly detection methods	✔
Describe local outlier factor algorithm	
Implement LOF	
Optimize LOF by tuning its hyperparameters	

Local outlier factor

- Local outlier factor (LOF) is a **density-based** outlier detection algorithm
- It uses the density of data points in the distribution as a key factor to detect outliers
- It measures the local density deviation of a given point with respect to its neighbors
- Outliers point that have a substantially lower density than their neighbors

LOF

- The local density of a point is compared with the densities of its neighbors
- We see here that `A` has a much lower density than its neighbors



Source: *https://upload.wikimedia.org/wikipedia/commons/thumb/4/4e/LOF-idea.svg/250px-LOF-idea.svg.png*

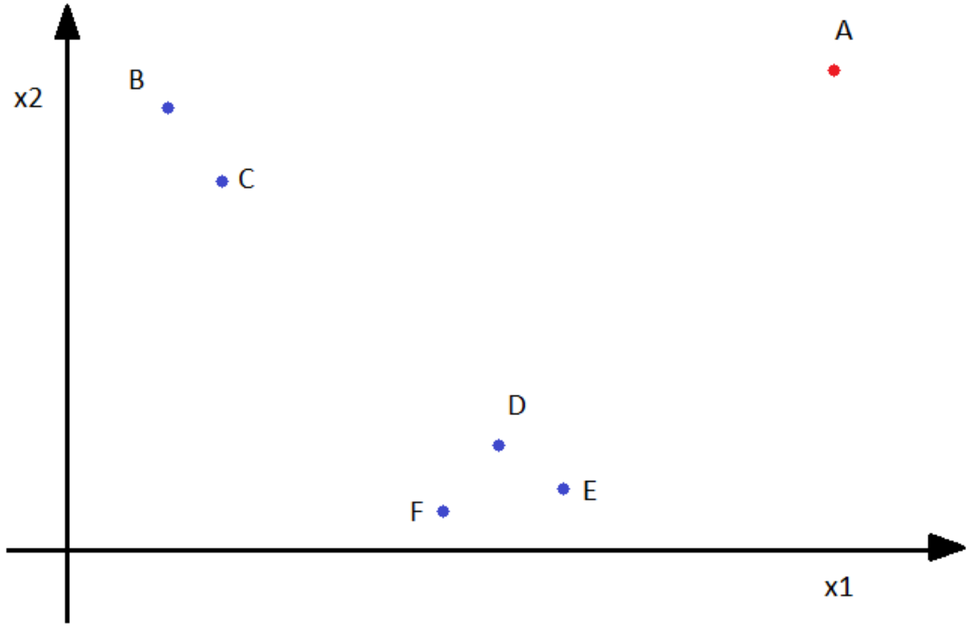
LOF - steps involved

The estimation of LOF involves the following steps:

- Calculate pairwise distances
- Introduce Hyperparameter `k`
- Average reachability distance
- Local reachability density

LOF - sample points

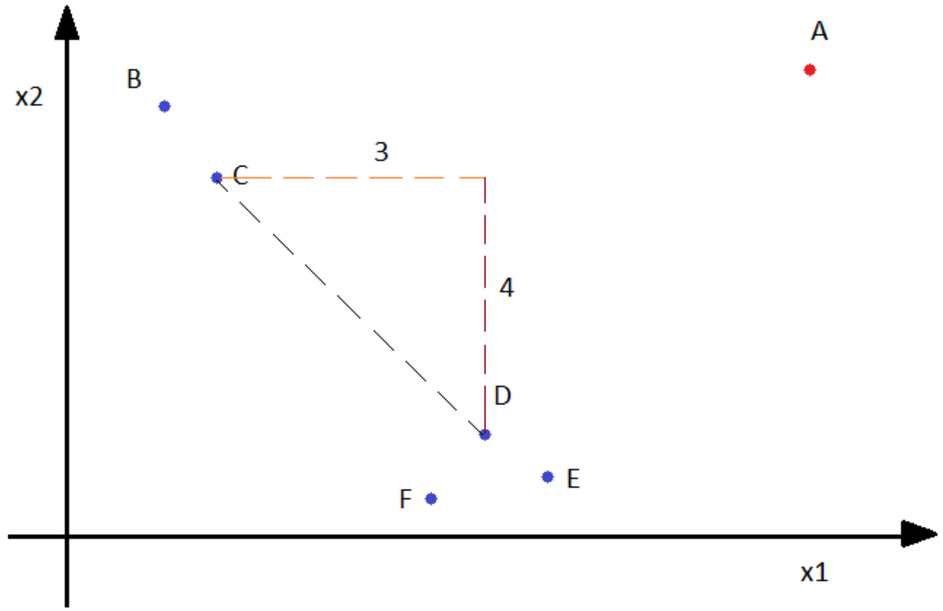
- It is quite evident from the sample points below that `A` is the outlier point
- Let's illustrate the working of LOF and how the local densities are calculated using these points



LOF - pairwise distance

- Before we apply LOF, we will need to calculate pairwise distance between each point and this can be calculated using Pythagoras theorem
- For example, distance between D and C would be:

$$DC^2 = 3^2 + 4^2$$
$$DC^2 = 25$$
$$DC = 5$$



LOF - pairwise distances

- The table shows the pairwise distances calculated using the Pythagorean Theorem for each and every point

Distances	
AB	8.6
AC	8.4
AD	7
AE	6.4
AF	8.5
BC	1
BD	6
BE	6.8
BF	6.5
CD	5
CE	5.8
CF	5.5
DE	0.8
DF	1.2
EF	1.6

LOF - hyperparameter k

- `k` represents the distance of a point to its `kth` neighbor
- Let’s choose the value of `k` equal to 3 for our example
- This means that only the 3rd closest point for any point is relevant from the table
- For simplicity, let’s consider only the points A, B, C, and D for local density estimations

LOF - k-th closest

- For point `A` - 1st closest is E with 6.4, 2nd closest is D with 7, 3rd closest is C with 8.4
- For point `B` - 1st closest is C with 1, 2nd closest is D with 6, 3rd closest is F with 6.5
- Likewise, the 3rd closest points for each A, B, C and D are found as shown

Distances	
AB	8.6
AC	8.4
AD	7
AE	6.4
AF	8.5
BA	8.6
BC	1
BD	6
BE	6.8
BF	6.5
CA	8.4
CB	1
CD	5
CE	5.8
CF	5.5
DA	7
DB	6
DC	5
DE	0.8
DF	1.2

LOF - reachability distance

- Reachability distance is a measure that represents distance at which a point can be “reached” from its neighbors
- This measure is the maximum of the distance between two points and the k-distance of the second point

  `RD(a,b) = max{k-distance(b), dist(a,b)}`
- If point `A` lies outside the `k` neighbors of `B`, then RD (a,b) will be the distance between `A` and `B`. Else, if point `A` lies within the `k` neighbors of `B`, then RD (a,b) will be the k-distance of `B`

LOF - average reachability distance

- The average reachability distance is calculated for each data point
- For point A,

$$AverageRD(A) = \frac{1}{3} * [max(3^{rd}DistB, dist(A,B))$$

$$+max(3^{rd}DistC, dist(A,C)) + max(3^{rd}DistD, dist(A,D))]$$

$$AverageRD(A) = \frac{1}{3} * [max(6.5, 8.6) + max(5.5, 8.4) + max(5, 7)]$$

$$AverageRD(A) = \frac{1}{3} * [8.6 + 8.4 + 7] = 8$$

LOF - average reachability distance

- Likewise, on calculating the average reachability distance for the other points
  - Average RD(B) = 6.7
  - Average RD(C) = 6.6
  - Average RD(D) = 6.8

LOF - local reachability density

- Local reachability density (LRD) is the inverse of the average reachability distance
- In simple terms, the longer the distance a point is to its neighbors, the sparser is the area where the point is located

$$LRD(A) = \frac{1}{AverageRD(A)}$$

- The LRD for the points is as follows:

$$LRD(A) = \frac{1}{8} = 0.125$$

$$LRD(B) = \frac{1}{6.7} = 0.14$$

$$LRD(C) = \frac{1}{6.6} = 0.15$$

$$LRD(D) = \frac{1}{6.8} = 0.14$$

LOF - estimation

- LOF for a point is calculated using the LRDs
- LOF of a point is the average ratio of the LRDs of the neighbors of the point to the LRD of that point
  - LOF(point) ~ 1 means similar density as neighbors
  - LOF(point) < 1 means higher density than neighbors (inlier)
  - LOF(point) > 1 means lower density than neighbors (outlier)

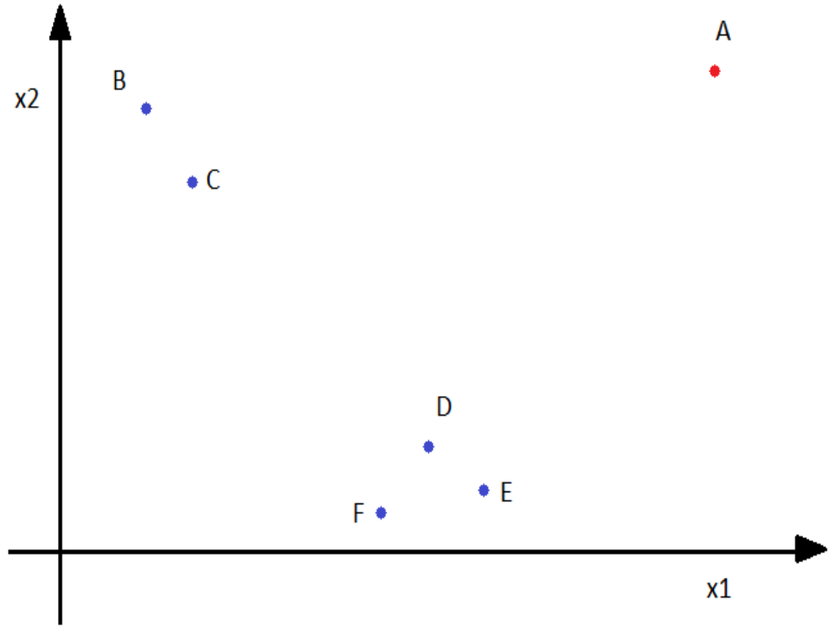
LOF - estimation

- Let’s estimate the LOF for the point A

$$LOF(A) = \frac{1}{3} * \frac{(LRD(B) + LRD(C) + LRD(D))}{LRD(A)}$$

$$LOF(A) = \frac{0.14 + 0.15 + 0.14}{3 * 0.125} = 1.14$$

- LOF of point `A` is greater than 1 indicating that the point is an outlier
- It is also clear from our points that point `A` is an outlier



Knowledge check 1

Module completion checklist

Objective	Complete
Discuss anomaly detection methods	✔
Describe local outlier factor algorithm	✔
Implement LOF	
Optimize LOF by tuning its hyperparameters	

Loading packages

Let’s load the packages we will be using:

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle

from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.ensemble import IsolationForest
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your course materials folder
- `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
from pathlib import Path
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Load the dataset

- Load the `paysim_transactions.csv` dataset and print the head

```
paysim = pd.read_csv(str(data_dir)+"/paysim_transactions.csv")
paysim.head()
```

```
   step      type      amount  ... newbalanceDest  isFraud  isFlaggedFraud
0   308  CASH_OUT    94270.99  ...      486682.07        0               0
1   215  TRANSFER  1068883.00  ...     5165788.35        0               0
2   326  TRANSFER  2485281.21  ...     2663110.80        0               0
3   371   PAYMENT     2243.36  ...           0.00        0               0
4   283   PAYMENT     5845.82  ...           0.00        0               0

[5 rows x 11 columns]
```

Prepare the dataset for modeling

- We will use the numeric variables - `oldbalanceOrg`, `newbalanceOrg`, `OldbalanceDest`, and `newbalanceDest` as our predictors
- The other variables are categorical - for us to understand the technique better, we'll ignore the other variables

```
# Drop columns.
paysim = paysim.drop(['step', 'type','nameOrig', 'nameDest', 'isFlaggedFraud'], axis = 1)
```

```
paysim.columns
```

```
Index(['amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest',
       'newbalanceDest', 'isFraud'],
      dtype='object')
```

LOF on fraud dataset

- Our first step is to split the data into train and test datasets
- We will then separate non fraud observations from the train dataset
- We are going to train the LOF model on the regular observation as discussed

```
train, test = train_test_split(paysim, test_size=.30)

# Split fraud vs non fraud.
non_fraud = train[train['isFraud']==0]
fraud = train[train['isFraud']==1]
non_fraud = non_fraud.drop(['isFraud'], axis = 1)

test = test.append(fraud)
actual_test = test['isFraud']
```

Create and fit LOF model

- We now will instantiate our LOF model and run it on non_fraud data
- At first, we will simply run the model on our training data and predict on test
- We set n_neighbors = **5** and contamination = **0.1** as a random guess. Novelty is set to **True** for anomaly detection on new points

```
lof = LocalOutlierFactor(n_neighbors = 5,
                         metric = "manhattan",
                         contamination = 0.1,
                         novelty = True)

# model fitting
lof.fit(non_fraud)
```

```
LocalOutlierFactor(contamination=0.1, metric='manhattan', n_neighbors=5,
                   novelty=True)
```

Test predictions

- Predict on the test data using the trained LOF model

```
fraud_pred = lof.predict(test.iloc[:,:-1])
fraud_pred
```

```
array([ 1, -1,  1, ..., -1, -1,  1])
```

- We know that a LOF classifies the data points as -1 and +1 instead of 1 and 0
- Let's replace these values into 0 and 1 as we have in our Paysim dataset

```
fraud_pred[fraud_pred == 1] = 0
fraud_pred[fraud_pred == -1] = 1
```

Find TPR and TNR

```
tn, fp, fn, tp = confusion_matrix(actual_test, fraud_pred).ravel()
non_fraud_eval = tn / (tn + fp)
print(non_fraud_eval)
```

```
0.8962075848303394
```

```
fraud_eval = tp / (tp + fn)
print(fraud_eval)
```

```
0.7636363636363637
```

Exercise 1

Module completion checklist

Objective	Complete
Discuss anomaly detection methods	✔
Describe local outlier factor algorithm	✔
Implement LOF	✔
Optimize LOF by tuning its hyperparameters	

LOF model - hyperparameter tuning

- The hyperparameters that we will be tuning for optimizing the LOF model are **n_neighbors** and **contamination**
  - **n_neighbors**: number of neighbors to use; defines the neighborhood for the computation of local density
  - **contamination**: the amount of contamination of the data set, i.e., the proportion of outliers in the data set

LOF model - tuning neighborhood size

- Let’s give the number of neighbors ranging between 3 and 20
- We will store the TNR and TPR values predicted on the test data for each neighborhood size

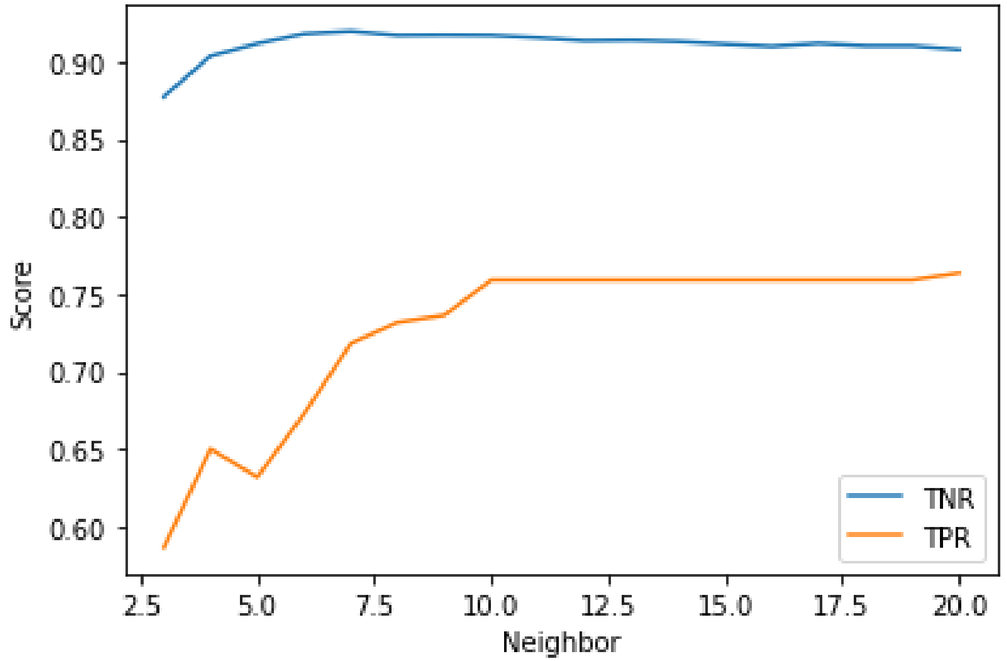
```
lof_df = pd.DataFrame()
for neighbor in range(3,21):
    lof = LocalOutlierFactor(n_neighbors = neighbor, metric = "manhattan", novelty = True)
    lof.fit(non_fraud)
    fraud_pred = lof.predict(test.iloc[:,:-1])
    fraud_pred[fraud_pred == 1] = 0
    fraud_pred[fraud_pred == -1] = 1
    tn, fp, fn, tp = confusion_matrix(actual_test, fraud_pred).ravel()
    non_fraud_eval = tn / (tn + fp)
    fraud_eval = tp / (tp + fn)
    values  =  [neighbor,non_fraud_eval,fraud_eval]
    values = pd.DataFrame(values).T
    lof_df = pd.concat([lof_df,values])

lof_df.columns=['Neighbor','TNR','TPR']
```

LOF model - tuning neighborhood size

- Let's plot the results obtained for each neighborhood size
- Based on the TNR and TPR values, we will need to pick an optimal neighborhood size for this model

```
plt.figure(figsize=(5,5))
plt.plot(lof_df['Neighbor'],lof_df['TNR'], label
= 'TNR')
plt.plot(lof_df['Neighbor'],lof_df['TPR'], label
= 'TPR')
plt.xlabel('Neighbor')
plt.ylabel('Score')
plt.legend(loc="lower right")
plt.show()
```

Note: *This plot may change for different train and test datasets*



- We can see that the TPR seems to have saturated at **10** neighbors.This value can be chosen as our optimal neighborhood size

LOF model - tuning contamination

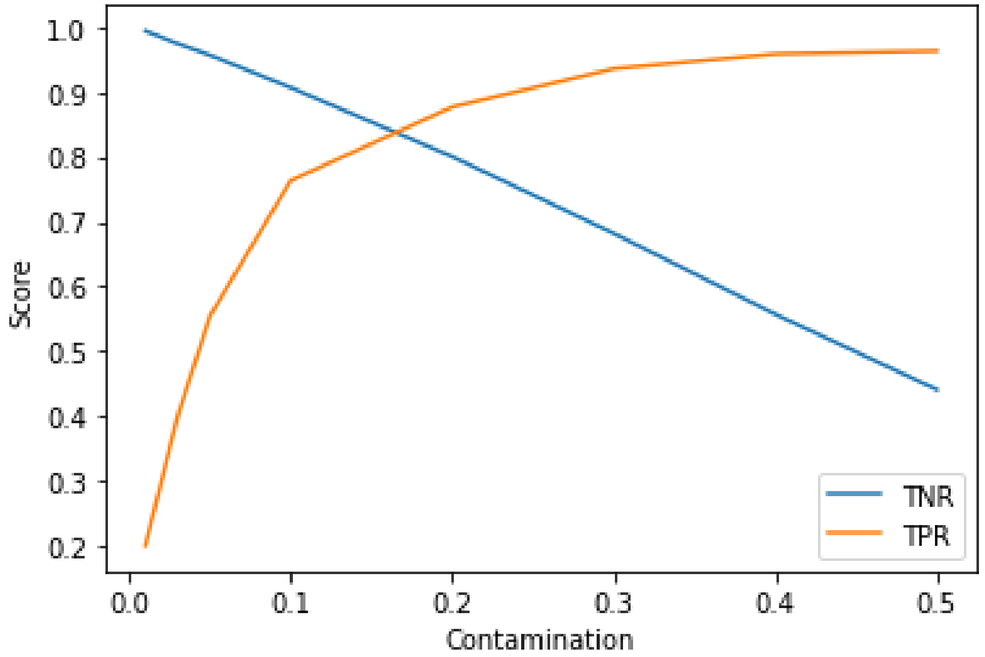
- Let’s give a list of contamination values as shown below
- We will store the TNR and TPR values predicted on the test data for each contamination value

```
contamination_values = [0.01,0.03,0.05,0.1,0.2,0.3,0.4,0.5]
lof_df = pd.DataFrame()
for contamination_value in contamination_values:
    lof = LocalOutlierFactor(n_neighbors = 20, metric = "manhattan",contamination =
contamination_value, novelty = True)
    lof.fit(non_fraud)
    fraud_pred = lof.predict(test.iloc[:,:-1])
    fraud_pred[fraud_pred == 1] = 0
    fraud_pred[fraud_pred == -1] = 1
    tn, fp, fn, tp = confusion_matrix(actual_test, fraud_pred).ravel()
    non_fraud_eval = tn / (tn + fp)
    fraud_eval = tp / (tp + fn)
    values  =  [contamination_value,non_fraud_eval,fraud_eval]
    values = pd.DataFrame(values).T
    lof_df = pd.concat([lof_df,values])

lof_df.columns=['Contamination','TNR','TPR']
```

LOF model - tuning contamination

- Let's plot the results obtained for each contamination value
- Based on the TNR and TPR values, we will need to pick an optimal contamination size for this model

```
plt.plot(lof_df['Contamination'],lof_df['TNR'],
label = 'TNR')
plt.plot(lof_df['Contamination'],lof_df['TPR'],
label = 'TPR')
plt.xlabel('Contamination')
plt.ylabel('Score')
plt.legend(loc="lower right")
plt.show()
```



- We can choose the contamination value to be **0.1** as we have close to maximum values for both TPR and TNR at this point

Optimized LOF model

- We will now fit the LOF model using the chose hyperparameters - n_neighbors = **10** and contamination = **0.1**

```
lof = LocalOutlierFactor(n_neighbors = 10,
                         metric = "manhattan",
                         contamination = 0.1,
                         novelty = True)

# model fitting
lof.fit(non_fraud)
```

```
LocalOutlierFactor(contamination=0.1, metric='manhattan', n_neighbors=10,
                   novelty=True)
```

Test predictions

- Predict on the test data using the trained LOF model and replace predicted values into 0 and 1 as we have in our Paysim dataset

```
fraud_pred = lof.predict(test.iloc[:,:-1])
fraud_pred[fraud_pred == 1] = 0
fraud_pred[fraud_pred == -1] = 1
```

- Find TPR and TNR

```
tn, fp, fn, tp = confusion_matrix(actual_test, fraud_pred).ravel()
non_fraud_eval = tn / (tn + fp)
print(non_fraud_eval)
```

```
0.8975382568196939
```

```
fraud_eval = tp / (tp + fn)
print(fraud_eval)
```

```
0.7772727272727272
```

Load performance_df dataframe

- Load the pickled file and append the scores of the LOF model

```
performance_df = pickle.load(open(str(data_dir)+"/performance_anomalies.sav","rb"))
```

```
s = pd.Series(['LOF', fraud_eval, non_fraud_eval],
              index=['model_name', 'TPR', 'TNR'])
performance_df = performance_df.append(s, ignore_index = True)
performance_df
```

```
                model_name       TPR       TNR
0  Decision_tree_baseline  0.671642  0.999667
1                   SMOTE  0.865672  0.991332
2                     LOF  0.772727  0.890927
3                     LOF  0.759091  0.903280
4                     LOF  0.750000  0.894868
5                     LOF  0.745455  0.900367
6                     LOF  0.754545  0.892982
7                     LOF  0.722727  0.893376
8                     LOF  0.777273  0.897538
```

Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Discuss anomaly detection methods	✔
Describe local outlier factor algorithm	✔
Implement LOF	✔
Optimize LOF by tuning its hyperparameters	✔

Save results as a pickle

- Pickle all generated data for next module

```
pickle.dump(non_fraud, open(str(data_dir) + '/non_fraud.sav', 'wb'))
pickle.dump(test, open(str(data_dir) + '/test.sav', 'wb'))
pickle.dump(actual_test, open(str(data_dir) + '/actual_test.sav', 'wb'))
pickle.dump(performance_df, open(str(data_dir) + '/performance_anomalies.sav', 'wb'))
```

What’s next?

- In this module we:
  - discussed anomaly detection techniques
  - implemented and optimized a LOF model
- In the next module, we will:
  - continue learning more about LOF
  - cover isolation forests and implement the model

Congratulations on completing this module!