- Initialize and Prepare data

```
# initialize
KNN_1_score = []
KNN_5_score = []
KNN_10_score = []
Kmeans_1_score = []
Kmeans_5_score = []
Kmeans_10_score = []
Cosine_distance_score = []
Minkowski_1_score = []
Minkowski_2_score = []
Minkowski_inf_score = []
Mahalanobis_score = []
LOF_score = []
test_data_0, test_label_0, lof_score_0 = None, None, None # be used for visualize      You, 1 s

for i in tqdm.tqdm(range(10)):
    train_data = orig_train_data[orig_train_label == i]
    test_data, test_label = resample(
        orig_test_data, orig_test_label, target_label=i, outlier_ratio=0.1
    )
    # [TODO] prepare training/testing data with label==i labeled as 0, and others labeled as 1
    train_label = np.zeros(len(train_data), dtype=int)
    for j in range(len(test_label)):
        if test_label[j] == i:
            test_label[j] = 0
        else:
            test_label[j] = 1
    test_label = np.array(test_label, dtype=int)
```

一開始先初始化一些 list，用於儲存各個方法在各個 digit 的 ROC-AUC score，最後才可以取平均。

準備 data 的部份，train_data 為全部都是某個 digit，並且視為正常，所以 train_label 全是 0，而 test_data 則是會做 resample，其中包含 10%不同的 digit 視為異常，所以 test_label 就是根據 digit 是否是目前的 i 來分配 0 和 1。

- K Nearest Neighbor

```
# KNN
for k in [1, 5, 10]:
    pred = KNN(train_data, test_data, k)
    score = roc_auc_score(test_label, pred)
    if k == 1:
        KNN_1_score.append(score)
    elif k == 5:
        KNN_5_score.append(score)
    else:
        KNN_10_score.append(score)
```

每個 digit 都跑 k=1，5，10 的 KNN，並分別將 score 放入對應的 list。

```python
def KNN(train_data, test_data, k):
    # Calculate distance
    distances_matrix = pairwise_distances(test_data, train_data, n_jobs=-1)
    k_nearest = np.sort(distances_matrix)[:, :k]
    anomaly_score = np.mean(k_nearest, axis=1)
    sorted_index = np.argsort(anomaly_score)[::-1]  # descending  order
    # Pick top n % as anomaly
    n = 10
    top_n = int(test_data.shape[0] * n * 0.01)
    anomaly_index = sorted_index[:top_n]
    pred = np.zeros(test_data.shape[0], dtype=int)
    pred[anomaly_index] = 1

    return pred
```

KNN 首先計算 test_data 到 train_data 的 pairwise distance，隨後對 test_data 的每個點到 train_data 的 distance 做排序，取出前 k 近的距離，並且將其做平均，做為該點的 anomaly score，最後取 Top n% large 的 score 做為 anomaly。

```
KNN with k=1 score: 0.8894898029408088
KNN with k=5 score: 0.887856101399416
KNN with k=10 score: 0.884372380438195
```

KNN 的 k 的大小在此處 ROC-AUC score 上變化並沒有很多，猜測是因為正常的點的分佈比較密集，異常的密度較小，所以比較容易判斷的出來距離的差異。

- K Means Clustering

```python
# K means
for k in [1, 5, 10]:
    pred = Kmeans(train_data, test_data, k)
    score = roc_auc_score(test_label, pred)
    if k == 1:
        Kmeans_1_score.append(score)
    elif k == 5:
        Kmeans_5_score.append(score)
    else:
        Kmeans_10_score.append(score)
```

每個 digit 都跑 k=1，5，10 的 KNN，並分別將 score 放入對應的 list。

```python
def Kmeans(train_data, test_data, k):
    # Construct clusters
    k_clusters = train_data[np.random.choice(train_data.shape[0], k, replace=False)]
    for i in range(100):  # run for 100 times
        distance_matrix = pairwise_distances(train_data, k_clusters, n_jobs=-1)
        assigned_cluster = np.argmin(distance_matrix, axis=1)
        k_clusters = np.array(
            [train_data[assigned_cluster == j].mean(axis=0) for j in range(k)]
        )
    # Pick n% distance as threshold
    n = 90
    train_distances = pairwise_distances(train_data, k_clusters, n_jobs=-1)
    train_min_distances = np.min(train_distances, axis=1)
    threshold = np.percentile(train_min_distances, n)
    # Calculate distance
    test_distances = pairwise_distances(test_data, k_clusters, n_jobs=-1)
    min_distances = np.min(test_distances, axis=1)
    pred = (min_distances > threshold).astype(int)

    return pred
```

K Means Cluster 首先隨機從 train_data 取 K 個點做為初始中心，隨後計算各個 train_data 到中心的距離，並指派各個點到最近的中心，最後根據各個 cluster 的點去更新中心，此步驟會重複一百次，目的是為了穩定中心。當中心建立完成後，先計算 train_data 各個點到各中心的距離，並取最近的距離，再取 90%遠的距離當作後續 test_data 判斷 anomaly 的 threshold。最後，便計算每個 test_data 的點到各個中心的距離，同樣取最近距離，再與 threshold 做比較，若是大於則為 anomaly。

```
Kmeans with k=1 score: 0.8036984302931737
Kmeans with k=5 score: 0.9032962054836078
Kmeans with k=10 score: 0.9053697032982937
```

K Means Cluster 不同的 k 對不同的 n% distance 的反應有所差異，若是選取 95%遠的 distance 作為 threshold 則 k=1 的分數會上升，而 k=5 和 k=10 的都會下降，猜測是因為 k 比較大時，中心的分佈可能比較分散，那麼各個點到最近中心的距離就不會這麼遠，再取更大%的 distance 當成 threshold 時，反而會將本來 anomaly 的點判斷成正常。

- Cosine Distance

```python
# Cosine distance
pred = Cosine_distance(test_data, k=5)
score = roc_auc_score(test_label, pred)
Cosine_distance_score.append(score)
```

```python
def Cosine_distance(test_data, k=5):
    # Calculate cosine distance
    norms = np.linalg.norm(test_data, axis=1, keepdims=True)
    normalized_data = test_data / norms
    cosine_similarity_matrix = np.dot(normalized_data, normalized_data.T)
    cosine_distance_matrix = (
        1 - cosine_similarity_matrix
    )  # map value to 0(same) ~ 2(different)
    # Find anomaly
    kth_nearest = np.sort(cosine_distance_matrix)[:, k]  # omit the first, namely itself
    sorted_index = np.argsort(kth_nearest)[::-1]  # descending  order
    # Pick top n % as anomaly
    n = 10
    top_n = int(test_data.shape[0] * n * 0.01)
    pred = np.zeros(test_data.shape[0], dtype=int)
    anomaly_index = sorted_index[:top_n]
    pred[anomaly_index] = 1

    return pred
```

Cosine Distance 首先需要計算 Cosine Similarity，使用 np.linalg.norm 可以計算 matrix 的範數，隨後根據公式算出 cosine similarity matrix，而因為此處要用 distance 做為判斷 anomaly，且 distance 的數字是越小越近，所以將 cosine similarity 的值域從-1～1 轉換成 0～2，並根據該 distance 找出各個點第 K 近的 distance 為何，最後則同樣取 Top n% large 的點做為 anomaly。

```
Cosine Distance with k=5 score: 0.9156307616891167
```

Cosine Distance 是我實作所有方式裡第二好的，猜測可能是因為其對於 test_data 之間的關聯性處理得比較細緻，所以在判斷 anomaly 上便會比較精準一點。

- Minkowski Distance

```python
# Minkowski distance
for r in [1, 2, None]:
    pred = Minkowski_distance(test_data, r=r, k=5)
    score = roc_auc_score(test_label, pred)
    if r == 1:
        Minkowski_1_score.append(score)
    elif r == 2:
        Minkowski_2_score.append(score)
    else:
        Minkowski_inf_score.append(score)
```

每個 digit 都跑 r=1，5，inf 的 Minkowski distance，並分別將 score 放入對應的 list。

```python
def Minkowski_distance(test_data, r=1, k=5):
    # Calculate minkowski distance
    diff_abs = np.abs(test_data[:, np.newaxis] - test_data)
    if r == None:
        distance_matrix = np.max(diff_abs, axis=-1)
    else:
        distance_matrix = np.sum(diff_abs**r, axis=-1) ** (1 / r)
    # Find anomaly
    kth_nearest = np.sort(distance_matrix)[:, k]  # omit the first, namely itself
    sorted_index = np.argsort(kth_nearest)[::-1]  # descending  order
    # Pick top n % as anomaly
    n = 10
    top_n = int(test_data.shape[0] * n * 0.01)
    pred = np.zeros(test_data.shape[0], dtype=int)
    anomaly_index = sorted_index[:top_n]
    pred[anomaly_index] = 1

    return pred
```

Minkowski distance 的計算方式很簡單直接，首先計算 test_data 的距離差異，然後根據 r 的大小做次方和再開次方根號，此處 r=inf 時，根據講義是取距離差異中最大者當成 r，所以便使用 np.max 取出最大值，隨後同樣取 Top n% large 的點做為 anomaly。

```
Minkowski Distance with r=1 score: 0.8747589229867497
Minkowski Distance with r=2 score: 0.8790753401609681
Minkowski Distance with r=inf score: 0.8689157894262326
```

Minkowski Distance 在不同的 r 底下變化同樣沒有很多，其中 r 取距離差異最大時的表現稍稍遜色 r=1 和 r=2，我在 print 出 r=inf 的值時發現會來到三十幾，猜測是因為次方較大，數值的差異很容易被放大，在某些比較密集的點區就難做到細緻的判斷。

- Mahalanobis Distance

```python
# Mahalanobis Distance
pred = Mahalanobis_distance(train_data, test_data, k=5)
score = roc_auc_score(test_label, pred)
Mahalanobis_score.append(score)
```

```python
def Mahalanobis_distance(train_data, test_data, k=5):
    # Calculate inverse covariance matrix
    covariance_matrix = np.cov(train_data, rowvar=False)
    inv_cov_matrix = np.linalg.inv(covariance_matrix)
    # Calculate mahalanobis_distance
    n_samples = test_data.shape[0]
    distance_matrix = np.zeros((n_samples, n_samples))
    for i in range(n_samples):
        diff_vector = test_data[i] - test_data
        for j in range(n_samples):
            distance_matrix[i, j] = (
                np.dot(np.dot(diff_vector[j], inv_cov_matrix), diff_vector[j].T) ** (1 / 2))
    # Find anomaly
    kth_nearest = np.sort(distance_matrix)[:, k]  # omit the first, namely itself
    sorted_index = np.argsort(kth_nearest)[::-1]  # descending  order
    # Pick top n % as anomaly
    n = 10
    top_n = int(test_data.shape[0] * n * 0.01)
    pred = np.zeros(test_data.shape[0], dtype=int)
    anomaly_index = sorted_index[:top_n]
    pred[anomaly_index] = 1

    return pred
```

Mahalanobis distance 首先需要以 train_data 計算 covariance matrix，代表各個 feature 之間的關聯性，隨後使用 np.linalg.inv 對 covariance matrix 取倒數，得到倒數矩陣後，便使用兩層 for 迴圈對每個 test_data 的點去做距離的計算，此處先取出各個點在 feature 上的差異，再根據公式做點積和開根號，最後全部計算完便是 distance matrix。在得到 distance matrix 後，同樣取 Top n% large 的點做為 anomaly。

```
Mahalanobis Distance with k=5 score: 0.9185515982275735
```

Mahalanobis Distance 是我實作的所有方式裡分數最高的，猜測可能是因為有先對於 feature 做 covariance matrix，對於 feature 的關聯性處理最細緻，所以在後續計算距離時，可以很好的抓出 feature 間的距離差異，便能比較好的判斷出 anomaly point。

- Local Outlier Factor

```python
# Density based
lof_score = Local_Outlier_Factor(test_data, k=5)
score = roc_auc_score(test_label, lof_score)
LOF_score.append(score)
if i == 0:
    test_data_0 = test_data
    test_label_0 = test_label
    lof_score_0 = lof_score
```

此處 i==0 所記錄下來的資訊是用於後續 visualize

```python
def Local_Outlier_Factor(test_data, k=5):
    # Calculate rechability distance
    distances_matrix = pairwise_distances(test_data, n_jobs=-1)
    k_nearest_index = np.argsort(distances_matrix, axis=1)[:, 1 : k + 1] # omit the first, namely itself
    k_nearest = np.sort(distances_matrix)[:, 1 : k + 1] # omit the first, namely itself
    kth_nearest = k_nearest[:, -1]
    reach_dist_matrix = np.maximum(kth_nearest[:, np.newaxis], distances_matrix)
    # Calculate local rechability density
    n_samples = test_data.shape[0]
    lrd_matrix = np.zeros(n_samples)
    for i in range(n_samples):
        lrd_matrix[i] = 1 / (np.sum(reach_dist_matrix[i, k_nearest_index[i]]) / k)
    # Calculate local outlier factor
    lof_matrix = np.zeros(n_samples)
    for i in range(n_samples):
        lof_matrix[i] = (np.sum(lrd_matrix[k_nearest_index[i]]) / lrd_matrix[i]) / k

    return lof_matrix # return lof score directly
```

LOF 首先以 Euclidean Distance 去計算 test_data 各個點的距離，並且取出各點第 k 近的距離，並根據該 kth 近的 distance，算出 reachability distance，具體算法如課程投影片所示，取 kth 近和兩點間距離的最大值。算出 reachability distance matrix 後，便可以計算 lrd，取前 k 近點的 reachability distance 的平均的倒數，最後再計算 lof，取前 k 近點的平均 lrd 再除以該點 lrd。得到的 lof 可以直接放入 roc_auc_score 與 test_label 計算分數，函式會自動計算合適的 threshold 判斷哪些點會被歸類於 anomaly。

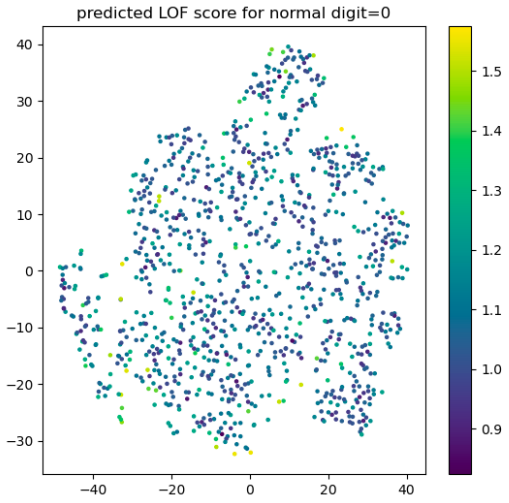
```
Local Outlier Factor with k=5 score: 0.7292456014892127
```

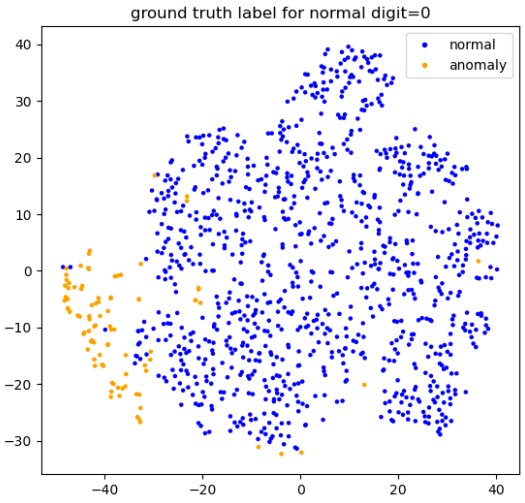
LOF 在所有方式內的分數是最低的，並且分數有一定的落差，猜測是因為數字的 feature 在經過降維過後，差異性並不會到很大，所以 density 會比較接近，從 visualize 的圖就可以看出這一點，LOF 的 range 差不多只有 0.8~1.6 之間，其實看不太出差異所在，所以在判斷 anomaly 上就會稍微遜色一點。

- Visualize

```python
def visualize_lof(test_data, test_label, lof_score):
    tsne = TSNE(n_components=2, random_state=42)
    test_data_2d = tsne.fit_transform(test_data)

    _, ax = plt.subplots(1, 2, figsize=(14, 6))

    sc1 = ax[0].scatter(
        test_data_2d[:, 0], test_data_2d[:, 1], c=lof_score, cmap="viridis", s=5
    )
    plt.colorbar(sc1, ax=ax[0])
    ax[0].set_title("predicted LOF score for normal digit=0")

    colors = ["blue" if label == 0 else "orange" for label in test_label]
    sc2 = ax[1].scatter(test_data_2d[:, 0], test_data_2d[:, 1], color=colors, s=5)
    ax[1].set_title("ground truth label for normal digit=0")

    blue_patch = plt.Line2D(
        [0],
        [0],
        marker="o",      You, yesterday • complete density based
        color="w",
        markerfacecolor="blue",
        markersize=5,
        label="normal",
    )
    orange_patch = plt.Line2D(
        [0],
        [0],
        marker="o",
        color="w",
        markerfacecolor="orange",
        markersize=5,
        label="anomaly",
    )
    ax[1].legend(handles=[blue_patch, orange_patch])

    plt.show()
```

首先使用 TSNE 進行降維成 2D，再創建一個 subplots 可以同時放下兩張圖片，隨後第一張圖繪畫出 test_data 的 lof_score，這裡使用的參數都是在計算 LOF 時所存下來的變數進行傳入。繪畫完第一張圖後再畫第二張圖，根據 test_label 的 0 為藍 1 為橘，畫出 label 分佈，並在第二張圖上標示出 maker。


predicted LOF score for normal digit=0
40
30
20
10
0
−10
−20
−30
−40
−20
0
20
40
1.5
1.4
1.3
1.2
1.1
1.0
0.9


ground truth label for normal digit=0
normal
anomaly
40
30
20
10
0
−10
−20
−30
−40
−20
0
20
40