Part 1. Implement K-means & DBSCAN algorithm using Numpy

Part 1-1. Implement K-means clustering algorithm

TODO : Fill in the blanks of the codes and write your own description of the source code in your report.

```python
class KMeans:

    def __init__(self, n_clusters=1, random_state=500):
        '''
        parameter:
        n_clusters: desired number of clusters
        random_state: random state for initializing the centroids
        '''
        assert n_clusters >= 1
        self.n_clusters = n_clusters
        self.random_state = random_state
        self.cluster_centers_ = None    # cluster centers (=centriods) will be initialized in 'fit' method


    def fit(self, X):
        '''
        parameter:
        X: data
        shape of X: (number of sample, feature dimensions)

        Compute the centroids that can achieve the minimum variance within clusters given a (training) data X.
        Clustering is performed iteratively until the centroids are not changed, i.e., until the optimum is reached.
        The centroids obtained here; "self.cluster_centers_" will be used in "predict" method.

        It returns a array of cluster labels for data X.
        For each sample in X, the cluster label means the index of the nearest center among centers in self.cluster_centers_. Thus, the cluster labels range from 0 to (n_clusters-1).

        Return:
        cluster_labels: numpy array of cluster labels for all data samples in X
        '''

        # Here, the centroids is initialized with arbitrary samples of X.
        np.random.seed(self.random_state)  # the choice of these random samples is governed by the "self.random_state".
```

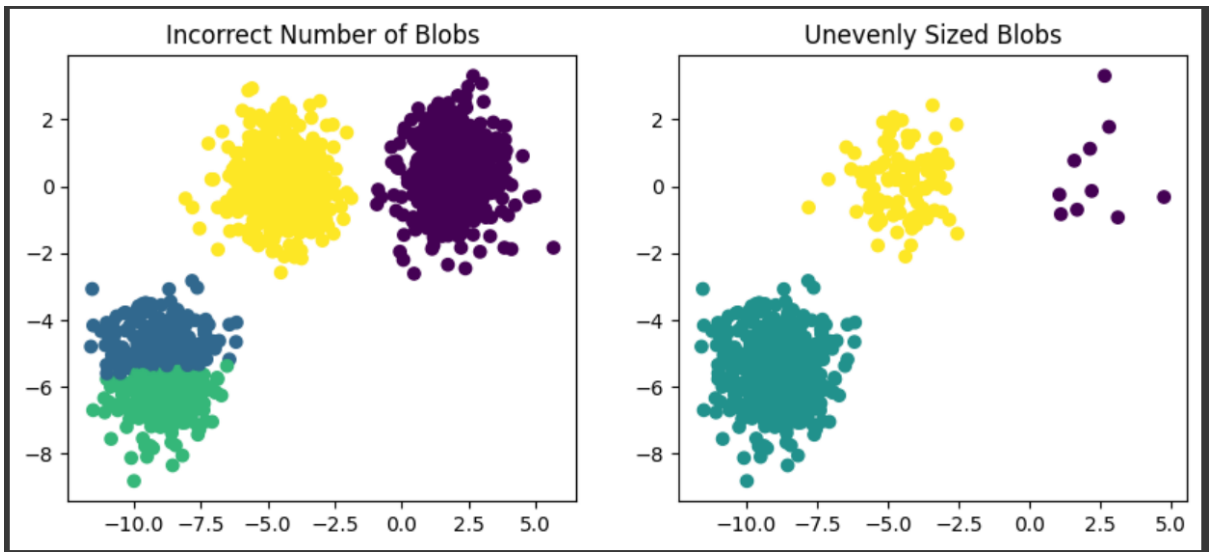
```python
        # Here, the centroids is initialized with arbitrary samples of X.
        np.random.seed(self.random_state)  # the choice of these random samples is governed by the "self.random_state".
        initial_cluster_centers_idx = np.random.choice(len(X), self.n_clusters, replace=False) # choose the 'n_clusters' number of samples in X to take them as initial centriods
        self.cluster_centers_ = X[initial_cluster_centers_idx]  # initial centroids

        # fill in the blank ------------------------------------
        n = len(X) # or X.shape[0], 전체 데이터 개수
        dim = X.shape[1]
        #print(dim)
        while(1):
        #for i in range(100):
            # 데이터 레이블링
            distance_arr = np.linalg.norm(X-self.cluster_centers_[0], 2, axis =1)    # 각 클러스터까지의 거리
            for i in range(self.n_clusters-1):
                distance_arr1 = np.linalg.norm(X-self.cluster_centers_[i+1], 2, axis =1)
                distance_arr = np.column_stack((distance_arr,distance_arr1))
            new_labels = distance_arr.argmin(axis=1)
            # var를 최소로 하는 새로운 center를 찾는다
            new_center = np.zeros([self.n_clusters,dim]) #initialize
            num_of_dot = np.zeros(self.n_clusters)

            for k,item in enumerate(new_labels):
                new_center[item]+=X[k]
                num_of_dot[item]+=1
            for j in range(self.n_clusters):
                new_center[j] = new_center[j]/num_of_dot[j]
            # 기존 center랑 비교한다
            if (new_center == self.cluster_centers_).all():
                break
            else:
                self.cluster_centers_ = new_center
            # 같으면 out, 다르면 반복
            # ------------------------------------------------------


        cluster_labels = new_labels
        return cluster_labels
```

```
    def predict(self, X):
        '''
        parameter:
        X: data
        shape of X: (number of sample, feature dimensions)

        For each sample in the data X (might be testset), predict the cluster using the centroids obtained from "fit" method.
        It returns an array of cluster labels for data X.

        Return:
        cluster_labels: numpy array of cluster labels for all data samples in X (testset)
        '''
        assert self.cluster_centers_ is not None

        # fill in the blank ------------------------------------
        dim = X.shape[1] #데이터의 차원 수
        distance_arr = np.linalg.norm(X-self.cluster_centers_[0], 2, axis =1)    # 각 클러스터까지의 거리 np.array
        for i in range(dim-1):
            distance_arr1 = np.linalg.norm(X-self.cluster_centers_[i+1], 2, axis =1)
            distance_arr = np.column_stack((distance_arr,distance_arr1))
        cluster_labels = distance_arr.argmin(axis=1)
        # ------------------------------------------------------

        return cluster_labels
```

def fit 함수는 X(data)를 input으로 받아 cluster_labels(모든X에 대한 numpy array of cluster labels)을 반환한다. 먼저 전체 데이터 중 랜덤하게 n개(클러스터의 개수)를 골라 초기 클러스터의 중심점으로 잡는다. 그 이후 모든 데이터 X에 대해 각 클러스터까지의 거리를 distance_arr = np.linalg.norm(X-self.cluster_centers_[0], 2, axis =1) 로 구한 후 argmin 함수를 통해 거리가 가장 가까운 중심점으로 라벨링해준다. 그 이후에는 variance를 최소로 하는 새로운 중심점을 찾아주기 위해 같은 label을 가진 데이터들의 중심을 계산하고, 해당 점을 새로운 중심점으로 정의한다. 새로운 중심점과 기존 중심점을 비교해 값이 같으면 labeling이 끝난 것으로 간주하고 while문을 빠져나온다. 그렇지 않으면 while문을 통해 이 과정을 계속해서 반복한다.

predict함수는 위의 fit함수에서 각 클러스터까지의 거리를 구한 후 argmin함수를 통해 가장 가까운 중심점을 찾아주는 코드를 그대로 써주었다.

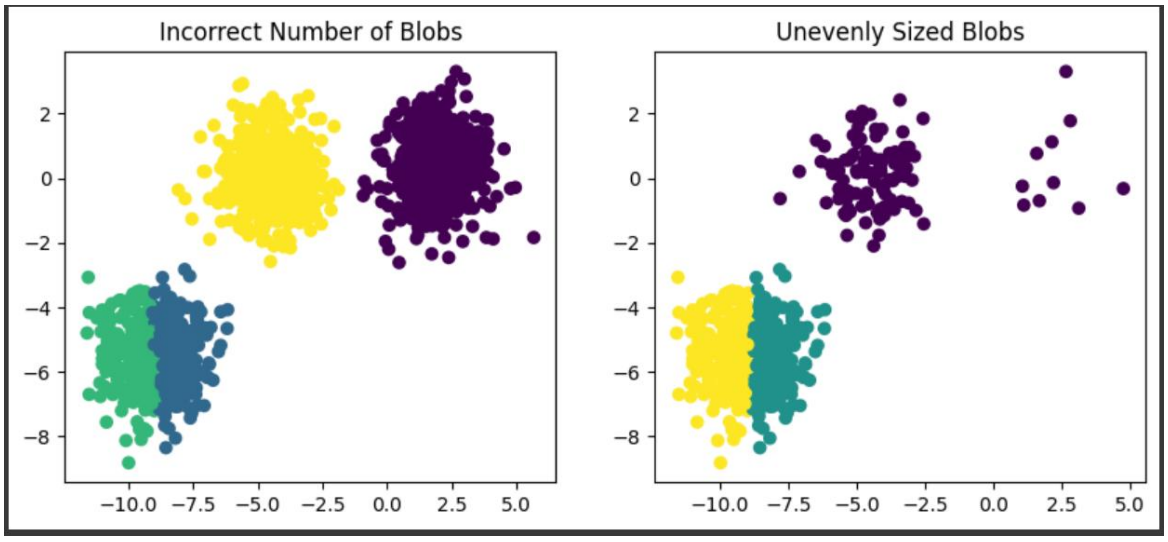
TODO : Run the test code for blobs dataset below and attach the results in the report.



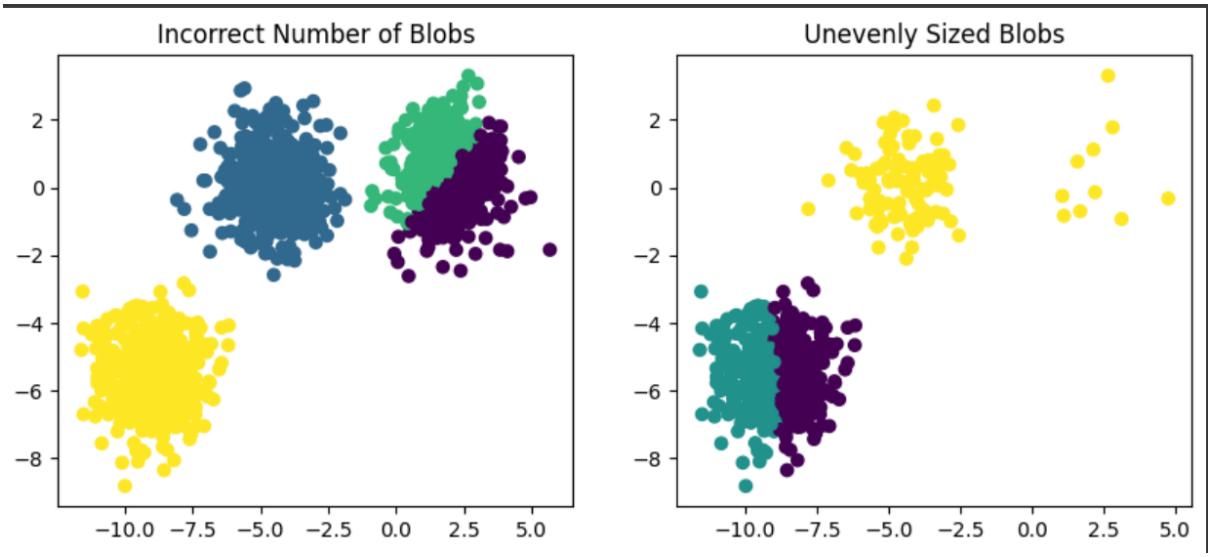
(random state = 500)

TODO : Try to run K-means algorithm with various random_state values in initializing centroids and attach the results in the report. Describe your observations as well as the reason why this happens.

(TODO : 중심점 초기화 시 다양한 random_state 값으로 K-means 알고리즘을 실행하여 결과를 보고서에 첨부합니다. 관찰 결과와 이러한 현상이 발생하는 이유를 설명합니다.)



(random state = 100)



(random state = 300)

Incorrect Number of Blobs의 경우에는 random state에 따라 결과는 조금 다르지만 클러스터링 자체는 잘 되었다고 볼 수 있다. Unevenly Sized Blobs의 경우에는 random state가 100, 300으로 달라지니 클러스터링이 잘못되었다.

이 부분이 K-mean 알고리즘의 취약한 부분이라 할 수 있다. 초기 클러스터의 중심점을 어떻게 지정하는가에 따라 결과가 바뀔 수 있다는 것이다. 이러한 현상(Unevenly Sized Blobs의 경우에 왼쪽 아래의 클러스터(A라고 하겠다.)가 하나로 묶이지 못하고 두 클러스터로 분리되어 있는, 잘못된 클러스터링이 발생한) 이유는 초기 클러스터 중심을 선정 할 때 3개의 중심점 중 2개 이상이 영역 A에 포함되어서 클러스터링이 잘못되었다고 유추할 수 있다.

Part 1-2. Implement DBSCAN algorithm

TODO : Fill in the blanks of the codes and write your own description of the source code in your report.

DBSCAN class에서 fit_predict함수는 모든 데이터 X에 대해 라벨링이 완료될 때까지 작동한다. 임의로 arbitrary_point를 뽑아 해당 점이 core이면 current_lable에 1을 더한 후 visit_all_successive_neighbors 함수를 실행시킨다. Core가 아니면 noise로 간주하고 -1로 라벨링한다.

pick_arbitrary_point 함수는 아직 라벨링이 되지 않은 점들 중 임의로 한 점을 뽑는 함수이므로 np.argwhere(self.cluster_labels==0)를 통해 라벨링이 되지 않은(즉, label이 0인) 점들의 np.array를 구한 후 random.choice함수를 통해 그 중에서 임의로 한 점을 선택해주었다.

```
    def pick_arbitrary_point(self):
        '''
        Pick arbitrary point among points that are not visited so far (for next successive visiting).
        It returns an "index" of point("p_idx"), not a data point itself.
        '''
        # fill in the blank ------------------------------------
        no_visit = np.argwhere(self.cluster_labels==0)
        p_idx = int(random.choice(no_visit))
        # ------------------------------------------------------
        assert self.cluster_labels[p_idx] == 0              # for sanity check
        return p_idx
```

is_core_sample 함수는 (eps 이내의) 주변 점들을 구하는 get_neighbors 함수를 이용해 구현하였다. len함수를 통해 주변 점들의 개수를 구해 그 값에 1을 더한 값이 min_samples 이상이면 Ture를, 그렇지 않으면 False를 반환한다.

```
    def is_core_sample(self, p_idx):
        '''
        parameter:
        p_idx: index of point

        Check whether the "p_idx" is a core sample or not.
        If it is, return True. Otherwise, return False.
        You can use "get_neighbors" method, which is defined below.
        You can define a core sample that has greater than or equal to min_samples points in its neighbor, where the point itself is also included in its neighbor.
        '''
        # fill in the blank ------------------------------------
        num_of_neighbors = len(self.get_neighbors(p_idx)) + 1
        if num_of_neighbors >= self.min_samples:
            self.core_sample_indices_.append(p_idx)
            return True
        return False

        # ------------------------------------------------------
```

visit_all_successive_neighbors 함수는 앞에서 설명했듯이 core점인 경우에 대해서 실행되어지는 함수이다. 이 함수는 해당 함수 주변의 점에 대해 방문(visit)하여 라벨링하는 것 뿐만 아니라 주변 점이 core point라면 그 core point의 주변 점 또한 방문하는 함수이다. 따라서 set을 이용해주었다. 가장 처음에는 all_neighbors_indices에 input으로 받은 p_idx 점만 존재한다. 그 이후에는 집합 all_neighbors_indices에서 임의로 점을 뽑아 해당 점이 core sample이면 주변 점들을 all_neighbors_indices에 더해준다(get_neighbors 함수 이용). 그러나 이 때 이미 labeling이 된 점들(self.labeled_indices)에 대해서는 다시 방문할 필요가 없으므로 차집합 연산을 해준다. 그리고 또한 기본적으로 label이 0과 -1인 점들(특정 클러스터에 포함되지 않은 점들)에 대해서는 현재 label값을 지정해주고 labeled_indices 집합에 해당 점을 포함시킨다.

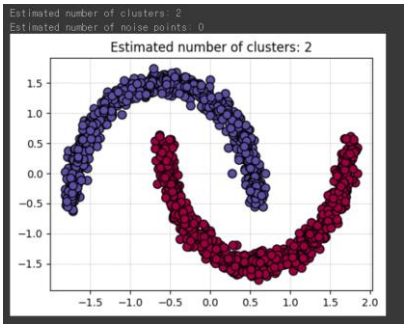
```
68    def visit_all_successive_neighbors(self, p_idx):
69        '''
70        parameter:
71        p_idx: index of point
72
73        Visit all the neighbors of "p_idx" as well as the neighbors of all the visited points if they are the core points themselves.
74        Assign current cluster label everytime you visited. But you don't need to relabel them if they are already allocated to a specific cl
75        It returns nothing but modifies "self.cluster_labels" in-place when labeling.
76        '''
77
78        all_neighbors_indices = {p_idx}
79
80        while all_neighbors_indices:
81            point = all_neighbors_indices.pop()
82            if self.is_core_sample(point):
83                all_neighbors_indices = all_neighbors_indices | self.get_neighbors(point)
84                all_neighbors_indices = all_neighbors_indices - self.labeled_indices
85            if self.cluster_labels[point] == 0 or self.cluster_labels[point] == -1:
86                self.cluster_labels[point] = self.current_label
87                self.labeled_indices.add(point)
```

get_neighbors 함수는 모든 점들과의 거리를 np.linalg.norm함수를 통해 계산하여 그 값이 eps보다 작으면 neighbors로 간주하고 집합에 포함시킨다. 마지막에 p_idx점 자체는 집합에서 제거해준다.

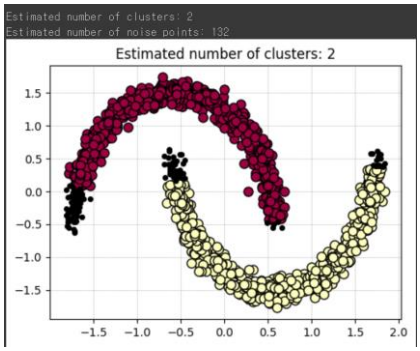
```
89    def get_neighbors(self, p_idx):
90        '''
91        parameter:
92        p_idx: index of point
93
94        It returns a "set of indices" of neighbors of "p_idx" point.
95        l2 norm will be considered for computing distances.
96        '''
97        # fill in the blank ------------------------------------
98        neighbors = set()
99        for i in range(len(X)):
100           if np.linalg.norm(X[i]-X[p_idx], 2) <= self.eps:
101               neighbors.add(i)
102       neighbors.remove(p_idx)
103       return neighbors
104       # ------------------------------------------------------
```

TODO : Try three different (eps, MinPts) combinations(depeding on your choice) for each dataset and attach the results in the report. Describe how the results change with varying eps and MinPts.

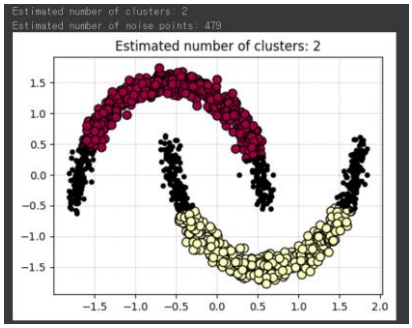
TODO: 각 데이터 세트에 대해 (선택에 따라) 세 가지 다른 (eps, MinPts) 조합을 시도하고 결과를 보고서에 첨부합니다. 다양한 eps 및 MinPts에 따라 결과가 어떻게 변화하는지 설명합니다.



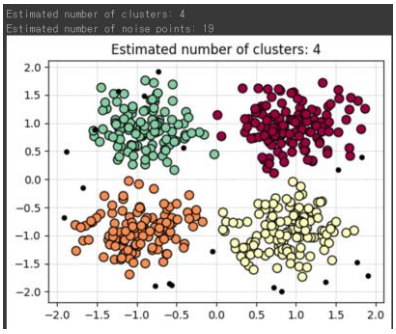
eps : 0.5, MinPts : 100



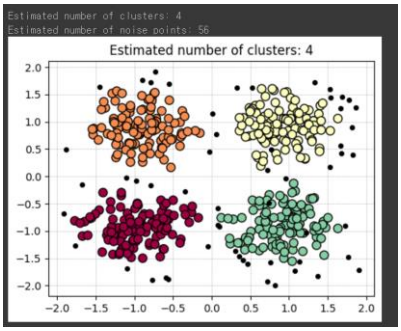
eps : 0.4, MinPts : 100



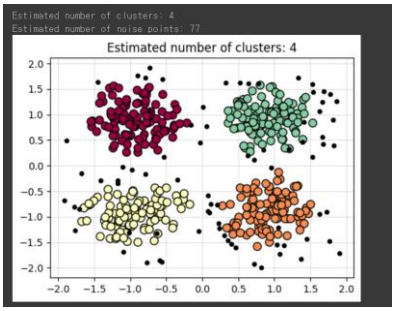
eps : 0.5, MinPts : 150



eps: 0.5, MinPts: 40



eps: 0.5, MinPts: 60



eps: 0.4, MinPts: 40

eps가 작아질수록, MinPts가 커질수록 noise의 개수가 많아지는 것을 확인할 수 있었다. 또한 극단적인 케이스일 경우에는 클러스터가 제대로 이루어지지 않았다(ex. MinPts가 너무 크면 대부분의 점이 noise 처리, eps가 너무 크면 모든 점이 하나의 클러스터로 정의됨 등등). 따라서 DBSCAN 알고리즘을 통해 데이터를 분류할 때에는 적절한 eps값과 MinPts값을 적용시키는 것이 중요하다.

TODO : Answer the following questions (cleary write down your own explanation on your answer to get a full credit)

Q1. What is the main difference between noise and border point?

Q2. Do you think that the clustering results are deterministic even if the visiting order is changed? If you do not think so, find the values of eps and MinPts that produce different clustering results depending on the visiting order for the given dataset below, and describe why the results are different.

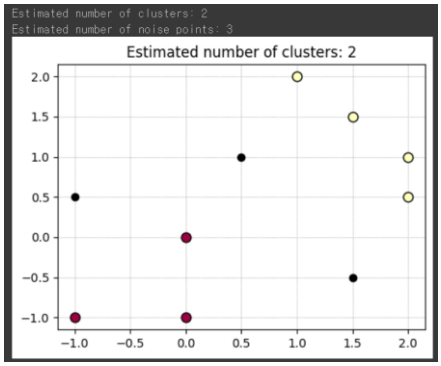
Q1. 소음과 경계점의 주요 차이점은 무엇입니까?

Q2. 방문순서가 바뀌어도 클러스터링 결과가 결정적이라고 생각하십니까? 그렇지 않은 경우 아래 주어진 데이터 집합의 방문 순서에 따라 서로 다른 클러스터링 결과를 생성하는 eps 및 MinPts

값을 찾고 결과가 다른 이유를 설명합니다.

A1. Noise와 border point의 주요한 차이점은 주변에 core point가 존재하는지 여부이다. Core point가 아닌 점은 noise 혹은 boder point일텐데, 해당 점의 주변(neighbor, eps 이내의 점)에 core point가 있다면 boder point로서 label을 할당받을 것이고 core point가 없다면 noise 처리될 것이다.

A2. eps:1.1, MinPts:3으로 고정한 후 random seed값을 바꿔가며 코드를 실행시켜주었다. 결과는 아래와 같다.

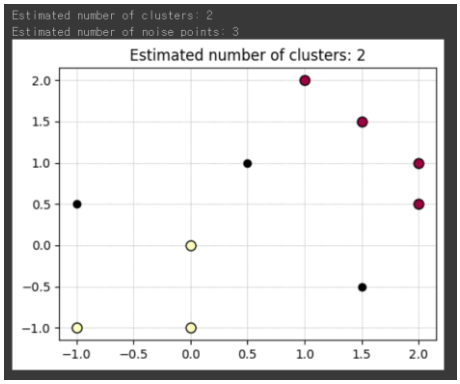


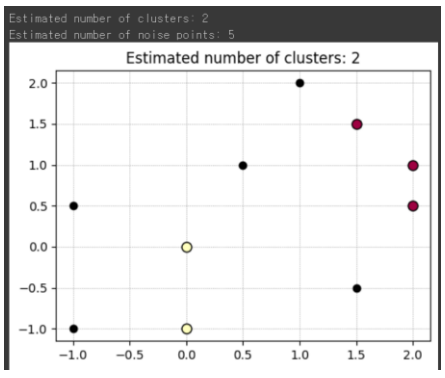
클러스터링 결과는 항상 같게 나왔다.

Q3. Let us slightly modify the algorithm (in fit_predict method) as follows:

Then, what is the possible range of number of noise samples after the clustering is completed? (write down in general form and express it using the number of differnt types of points(e.g. core, noise) which are assumed to be obtained in orignal DBSCAN. The modified version runs with the same and fixed eps, MinPts that the original one used)

그렇다면 클러스터링이 완료된 후 가능한 노이즈 샘플의 수는 얼마나 됩니까? (일반적인 형태로 기록하고 원본 DBSCAN에서 얻은 것으로 가정한 여러 유형의 포인트(예: 코어, 노이즈)를 사용하여 표현합니다. 수정된 버전은 원래 버전을 사용한 것과 동일하고 고정된 eps, MinPts로 실행됩니다.)





eps:1.1, MinPts:3 (random seed =12) eps:1.1, MinPts:3 (random seed =13)

앞에서와 달리 방문 순서에 따라 서로 다른 클러스터링 결과가 도출될 수 있음을 확인할 수 있다. 결과가 다른 이유는 border point이지만 주변의 core point보다 먼저 이 점을 방문하게 될 경우 core point가 아니므로 -1이 labeling되어 noise로 확정되기 때문이다. 위 결과에서도 (-1,-1)은 borer point로 원래는 클러스터에 포함되는 것이 맞으나 random seed =13의 경우 noise로 처리되어있다. 즉 (–1,-1) 점을 먼저 방문했다는 의미이다.

이렇게 알고리즘을 구현할 경우 core point를 하나도 방문하지 않은 채 모든 noise 및 border point를 먼저 방문하게 된다면 이론상 최대 noise + border point의 수(또는 모든 point수 – core point의 수)만큼 noise로 표현될 것이다. 따라서 가능한 noise 샘플의 수는 3개~7개 사이이다. (original DBSCAN을 적용시켰을 시에 core point 개수=3, border point 개수= 4, noise point 개수 = 3)

Part 2. Numpy Implementation of autograd, torch-like Tensor and Module

Task 1. Implment backward of the following operations:

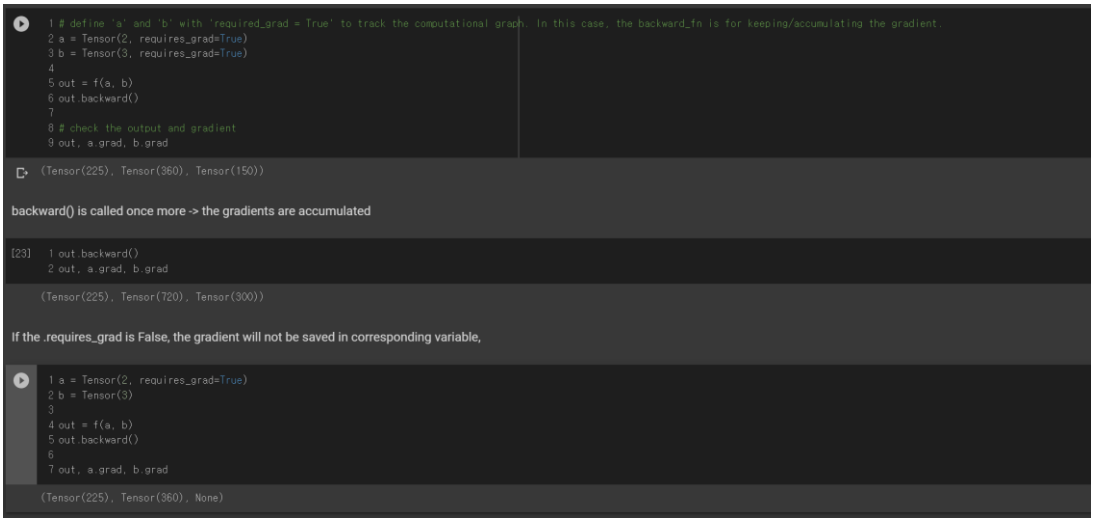
TODO1 : Fill in the blanks of the codes for Task 1 and write your own description of the source code in your report.

```python
# It computes the output using only "handcrafted" basic level operations!
def f(a, b):
    # (a^2 + 1)^2 x b^2 = (a^2 x b + b)^2
    c = mul(a, a)
    d = mul(c, b)
    e = add(d, b)
    output = mul(e, e)
    return output


# handcrafted basic level operations
# addition
def add(a, b):
    # fill in the blank ------------------------------------
    output = a+b
    # ------------------------------------------------------

    def backward(grad_output=1):
        # fill in the blank ------------------------------------
        grad_input_a = np.ones_like(a) * grad_output
        grad_input_b = np.ones_like(b) * grad_output
        a.backward_fn(grad_input_a)
        b.backward_fn(grad_input_b)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output


# multiplication
def mul(a, b):
    # fill in the blank ------------------------------------
    output = a*b
    # ------------------------------------------------------

    def backward(grad_output=1):
        # fill in the blank ------------------------------------
        grad_input_a = b * grad_output
        grad_input_b = a * grad_output
        a.backward_fn(grad_input_a)
        b.backward_fn(grad_input_b)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output
```

add함수의 출력값(output)은 a+b이고, 이것의 gradient는 1(벡터)에 grad_output을 곱한 값이다. 따라서 np.ones_like 함수를 통해 정의할 수 있다.

mul 함수의 경우 출력값은 a * b이다. 이 함수의 gradient는, a의 경우 b * gard_output이고 b의 경우 a * grad_output이다.

TODO2 : Attach the results in your report.



TODO3 : Consider and implement the following function f(x,y) . Get the gradient of f with respect to x,y when (x,y)=(1,2) . If there is any operation you want to use, feel free to define it just like how we defined add() and mul() functions above.

먼저 f의 정의에 맞게 f의 연산을 기본 연산으로 쪼개서 나타낸다.

```
def g(x, y):
    # fill in the blank ------------------------------------
    #output = np.sin(x)*(x+y)*(x**3+1)
    # ------------------------------------------------------
    a = add(x,y)
    b = mul(x,x)
    c = mul(x,b)
    d = add(c,a)
    e = add(d,a)
    output = mul_sin(x,e)

    return output

```

Add, mul 함수의 경우 위에서 사용한 코드를 그대로 사용하였다. mul_sin함수는 매개변수 x,y를 받아 sin(x) * y 연산을 실행하는 함수이다. 따라서 output은 np.sin(x) * y이다. 각 매개변수의

gradient는 x의 경우 np.cos(x)*y, y의 경우 np.sin(x) * grad_output으로 나타내어진다.

```
def mul_sin(x,y):
    output = np.sin(x)*y

    def backward(grad_output=1):
        grad_input_x = np.cos(x) * y
        grad_input_y = np.sin(x) * grad_output
        x.backward_fn(grad_input_x)
        y.backward_fn(grad_input_y)

    output.backward_fn = backward
    return output
# ------------------------------------------------------------
```

아래는 x=1, y=2에 대한 코드 실행 결과이다.

```
# define 'x' and 'y' with 'required_grad = True' to track the computational graph.
x = Tensor(1, requires_grad=True)
y = Tensor(2, requires_grad=True)

out = g(x, y)
out.backward()

# check the output and gradient
out, x.grad, y.grad
```

```
(Tensor(5.89029689), Tensor(7.98947107), Tensor(1.68294197))
```

Task 2. Implment zero_grad() and update() methods

TODO1 : Fill in the blanks of the codes for Task 2 and write your own description of the source code in your report.

Zero_grad는 매개 변수의 모든 gradient를 0으로 만든다. 먼저, 모든 파라미터를 self.parameters()로 받은 후에 해당 원소가 Tensor이면 grad.fill(0)를 통해 0으로 만들고, 그렇지 않으면 np.zeros_like함수를 통해 0으로 만든다.

Update는 각 매개변수를 lr의 학습 속도로 업데이트해준다. 마찬가지로 self.parameters()로 모든 파라미터를 받은 후 각 원소의 값을 -lr * (i.grad) 만큼 조정하여 업데이트한다.

.

```python
    def zero_grad(self):
        # fill in the blank ------------------------------------
        pm = self.parameters()
        for i in pm:
            if isinstance(i.grad, Tensor):
                i.grad.fill(0)
            else:
                i.grad = np.zeros_like(i)
        # ------------------------------------------------------

    def update(self, lr):
        # fill in the blank ------------------------------------
        pm = self.parameters()
        for i in pm:
            i -= lr * (i.grad)
        # ------------------------------------------------------

```

TODO2 : Attach the results in your report.

```python
class TestModel(Module):
    def __init__(self, init_param):
        super(TestModel, self).__init__()
        self.some_param = Tensor(init_param, requires_grad=True)

    def forward(self, x):
        return f(self.some_param, x)  ## f is defined above in Part1-Task1

test_model = TestModel(12.)
out = test_model(Tensor(0.3))
out.backward()

print('gradient of some_param before zero_grad : ', test_model.some_param.grad)
print('some_param before update : ', test_model.some_param)
test_model.update(lr=0.01)
print('some_param after update : ', test_model.some_param)
test_model.zero_grad()
print('gradient of some_param after zero_grad : ', test_model.some_param.grad)
```

```
gradient of some_param before zero_grad :  626.3999999999999
some_param before update :  12.0
some_param after update :  5.7360000000000015
gradient of some_param after zero_grad :  0.0
```

Task 3. Implement the ReLU, Linear modules(with backward)

Task 4. Implement CrossEntropyLoss(with backward) to define appropriate criterion before training

TODO1 : Fill in the blanks of the codes for Task 3, 4 and write your own description of the source code in your report.

ReLU 함수는 0을 기점으로 0보다 작으면 출력값은 0이고(y=0), 0보다 크면 input 그대로 output이 된다(y=x). 따라서 output = np.maximum(0, input)으로 표현할 수 있다. gradient도 마찬가지로 0을 기점으로 나뉘는데 0보다 작으면 grad가 0이되고, 0보다 크면 grad는 grad_output * 1이다. 이를 해결하기 위해 np.sign 함수를 사용하여 grad를 정의해주었다.

```
class ReLU(Module):
    def __init__(self):
        super(ReLU, self).__init__()

    def forward(self, x):
        return relu_function(x)


def relu_function(input):
    # fill in the blank ------------------------------------
    output = np.maximum(0, input)
    # ----------------------------------------------------
    def backward(grad_output):
        # fill in the blank ------------------------------------
        grad_input_input = grad_output * ((np.sign(input) + 1) / 2)
        input.backward_fn(grad_input_input)
        # ----------------------------------------------------
    output.backward_fn = backward
    return output
```

Linear 함수는 wx+b를 반환하는 함수이므로 output은 np.matmul(input,W) + b이다. 각 파라미터에 대한 gradient는 이전 lab 세션때 배웠듯이 transpose에 유의하여 계산해준다.

```python
class Linear(Module):
    def __init__(self, in_features, out_features):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.init_parameters()

    def init_parameters(self):
        W = np.random.randn(self.in_features, self.out_features) * math.sqrt(2 / (6 * self.in_features))
        bound = 1 / math.sqrt(self.in_features)
        b = np.random.uniform(-bound, bound, size=(self.out_features))
        self.W = Tensor(W, requires_grad=True)
        self.b = Tensor(b, requires_grad=True)

    def forward(self, x):
        return wx_plus_b(self.W, self.b, x)


def wx_plus_b(W, b, input):
    # fill in the blank ------------------------------------
    output = np.matmul(input, W) + b
    # ------------------------------------------------------

    def backward(grad_output):
        # fill in the blank ------------------------------------
        grad_input_W = np.matmul(input.T, grad_output)
        grad_input_b = np.matmul(np.ones(len(input)), grad_output)
        grad_input_input = np.matmul(grad_output, W.T)
        W.backward_fn(grad_input_W)
        b.backward_fn(grad_input_b)
        input.backward_fn(grad_input_input)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output
```

CrossEntropyLoss를 정의에 맞게 계산해주면 다음과 같다.

$$\mathbf{loss} = -\log(q_y) = -\log\left(\frac{e^{\sim y}}{\sum_j e^{z_j}}\right) = -Z_y + log\sum_j e^{z_j}$$

이것의 gradient를 구하는 과정은 조금 복잡한데, 자세한 과정은 아래 수식으로 적어두었다. 이때 유의해야 하는 점은 grad를 구하는 과정에서 시그마 부분을 미분해줄 때 i=j일 때와 i!=j일 때를 고려하여 계산해주어야 한다.

$$
\begin{aligned}
\nabla_{z_i} Loss &= \nabla_{z_i}(-z_y + \log \sum_j e^{z_j}) \\
&= \nabla_{z_i} \log \sum_j e^{z_j} - \nabla_{z_i} z_y \\
&= \frac{1}{\sum_j e^{z_j}} \nabla_{z_i} \sum_j e^{z_j} - \nabla_{z_i} z_y \qquad \text{from} \quad \frac{d}{dx} ln[f(x)] = \frac{1}{f(x)} \frac{d}{dx} f(x) \\
&= \frac{e^{z_i}}{\sum_j e^{z_j}} - \nabla_{z_i} z_y \\
&= q_i - \nabla_{z_i} z_y \\
&= q_i - \mathbb{1}(y = i)
\end{aligned}
$$

$$
\text{where} \quad \mathbb{1}(y = i) = \begin{cases} 1 & \text{if y=i} \\ 0 & \text{otherwise} \end{cases}
$$

```python
class CrossEntropyLoss:
    def __call__(self, input, target):
        return compute_cross_entropy_loss(input, target)


def compute_cross_entropy_loss(input, target):
    # fill in the blank ------------------------------------
    output = 0
    length = len(input)
    for i in range(length):
        tgt = target[i]
        ipt = input[i]
        output += (np.log(np.sum(np.exp(ipt))) - ipt[tgt])
    output = Tensor((output/length), requires_grad = True)
    # ------------------------------------------------------

    def backward():
        # fill in the blank ------------------------------------
        grad_input_input = np.exp(input)
        summ = np.sum(grad_input_input, axis = 1)
        for i in range(len(grad_input_input)):
            grad_input_input[i] /= summ[i]
            grad_input_input[i][target[i]] -= 1
        grad_input_input /= len(input)
        input.backward_fn(grad_input_input)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output
```

TODO2 : Attach the results in your report.

```
  5%|▌         | 1/20 [00:07<02:22,  7.50s/it]Loss = 0.5976743977352439
 10%|▉         | 2/20 [00:15<02:24,  8.02s/it]Loss = 0.23216241042671143
 15%|█▍        | 3/20 [00:24<02:20,  8.25s/it]Loss = 0.16636484746362962
 20%|█▉        | 4/20 [00:31<02:06,  7.88s/it]Loss = 0.1268566726499497
 25%|██▍       | 5/20 [00:41<02:08,  8.57s/it]Loss = 0.10148149223319261
 30%|██▉       | 6/20 [00:50<01:59,  8.55s/it]Loss = 0.083579199833649
 35%|███▍      | 7/20 [00:58<01:48,  8.37s/it]Loss = 0.06984142115224735
 40%|███▉      | 8/20 [01:05<01:37,  8.17s/it]Loss = 0.05945913935479037
 45%|████▍     | 9/20 [01:14<01:30,  8.22s/it]Loss = 0.05101464342146573
 50%|████▉     | 10/20 [01:21<01:20,  8.01s/it]Loss = 0.043736275121405654
 55%|█████▍    | 11/20 [01:30<01:13,  8.14s/it]Loss = 0.03853626425128526
 60%|█████▉    | 12/20 [01:38<01:06,  8.35s/it]Loss = 0.03293278904273675
 65%|██████▌   | 13/20 [01:46<00:56,  8.10s/it]Loss = 0.02836972512710277
 70%|██████▉   | 14/20 [01:54<00:48,  8.16s/it]Loss = 0.02437575441870696
 75%|███████▌  | 15/20 [02:03<00:41,  8.26s/it]Loss = 0.02112507069193561
 80%|████████  | 16/20 [02:10<00:31,  7.99s/it]Loss = 0.018087014489237628
 85%|████████▌ | 17/20 [02:19<00:24,  8.19s/it]Loss = 0.015290462560477147
 90%|█████████ | 18/20 [02:27<00:16,  8.26s/it]Loss = 0.013133806469908119
 95%|█████████▌| 19/20 [02:34<00:07,  7.97s/it]Loss = 0.01143620774238661
100%|██████████| 20/20 [02:43<00:00,  8.16s/it]Loss = 0.009546901877568377
Training Finished
```

```
[23]  1 prediction = model(test_images)
      2 prediction_label = np.argmax(prediction, axis=1)
      3 test_acc = np.sum((prediction_label == test_labels))/len(test_labels)
      4 print('Test Accuracy = {:.2f}'.format(100*test_acc))

      Test Accuracy = 97.95
```

Training on MNIST Dataset

```
  3%|▎         | 1/30 [00:13<06:42, 13.90s/it]Loss = 1.2501769376149994
  7%|▋         | 2/30 [00:26<06:10, 13.24s/it]Loss = 1.0862850909627255
 10%|▉         | 3/30 [00:39<05:49, 12.94s/it]Loss = 1.0049301720904598
 13%|█▎        | 4/30 [00:51<05:34, 12.86s/it]Loss = 0.958731983690068
 17%|█▋        | 5/30 [01:04<05:19, 12.78s/it]Loss = 0.915227189887724
 20%|██        | 6/30 [01:17<05:05, 12.75s/it]Loss = 0.8848178772743067
 23%|██▎       | 7/30 [01:30<04:53, 12.75s/it]Loss = 0.857701375251709
 27%|██▋       | 8/30 [01:42<04:40, 12.76s/it]Loss = 0.8273987409935036
 30%|███       | 9/30 [01:55<04:26, 12.71s/it]Loss = 0.8141616321745703
 33%|███▎      | 10/30 [02:08<04:13, 12.67s/it]Loss = 0.7982399799587785
 37%|███▋      | 11/30 [02:20<03:59, 12.60s/it]Loss = 0.7739239559279664
 40%|████      | 12/30 [02:33<03:47, 12.63s/it]Loss = 0.7558615322478633
 43%|████▎     | 13/30 [02:45<03:34, 12.63s/it]Loss = 0.7426419413849787
 47%|████▋     | 14/30 [02:58<03:22, 12.66s/it]Loss = 0.723870224748847
 50%|█████     | 15/30 [03:13<03:18, 13.21s/it]Loss = 0.7083480561703793
 53%|█████▎    | 16/30 [03:25<03:03, 13.14s/it]Loss = 0.6916700329117275
 57%|█████▋    | 17/30 [03:38<02:49, 13.03s/it]Loss = 0.6777476989397124
 60%|██████    | 18/30 [03:51<02:35, 12.95s/it]Loss = 0.669508563986773
 63%|██████▎   | 19/30 [04:04<02:21, 12.90s/it]Loss = 0.6558168276207672
 67%|██████▋   | 20/30 [04:16<02:08, 12.82s/it]Loss = 0.6390514231481427
 70%|███████   | 21/30 [04:29<01:54, 12.78s/it]Loss = 0.6276787011966038
 73%|███████▎  | 22/30 [04:42<01:42, 12.83s/it]Loss = 0.6317610089621198
 77%|███████▋  | 23/30 [04:55<01:30, 12.86s/it]Loss = 0.6037212862613907
 80%|████████  | 24/30 [05:08<01:17, 12.91s/it]Loss = 0.5890756478380667
 83%|████████▎ | 25/30 [05:21<01:04, 12.89s/it]Loss = 0.5795452528782566
 87%|████████▋ | 26/30 [05:34<00:51, 12.89s/it]Loss = 0.5717600703309638
 90%|█████████ | 27/30 [05:47<00:38, 12.89s/it]Loss = 0.559497171412907
 93%|█████████▎| 28/30 [05:59<00:25, 12.87s/it]Loss = 0.5433470543067526
 97%|█████████▋| 29/30 [06:12<00:12, 12.80s/it]Loss = 0.5405198060483031
100%|██████████| 30/30 [06:25<00:00, 12.85s/it]Loss = 0.5173837133636979
Training Finished
```

```
 1 prediction = model(test_images)
 2 prediction_label = np.argmax(prediction, axis=1)
 3 test_acc = np.sum((prediction_label == test_labels))/len(test_labels)
 4 print('Test Accuracy = {:.2f}'.format(100*test_acc))

Test Accuracy = 72.97
```

Trained on CIFAR-10 Dataset