Part 1: K-Means Clustering

Import Modules

In [33]:
```
import numpy as np
import sklearn
import matplotlib.pyplot as plt
import scipy.stats as stats
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import torch
import torch.nn as nn
```

Helper Functions

In [34]:
```
'''
Helper function given by teaching team to load data from data2D.npy.
'''
def load_data():
    X =np.load('data2D.npy')
    valid_batch = int(len(X) / 4.0)
    np.random.seed(45689)
    rnd_idx = np.arange(len(X))
    np.random.shuffle(rnd_idx)
    val_data = X[rnd_idx[:valid_batch]]
    train_data = X[rnd_idx[valid_batch:]]

    return train_data, val_data
```

Training Helpers

In [35]:
```
'''
This function helps us train the kmeans algorithm using given
value for k, for given number of epochs.
It returns the final training loss value and cluster centroids.
'''
def train_kmean_torch(train_data, k = 5, lr=0.1, epoch=150):
    # List of k random cluster centers containing d=2 coordinates
    m = torch.rand((k, train_data.shape[1]),
                   requires_grad=True,
                   dtype=torch.float64)

    # Convert training data to tensor
    X_train = torch.from_numpy(train_data)
    # Define optimizer as ADAM optimizer with iterable list of centroids
    optimizer = torch.optim.AdamW([m], lr=lr)

    # For each epoch, do the follwing
    for e in range(epoch):
        list_mse = []

        # For kth centroid, calculate the squared differences
        # between the two coordinates of centroid and each
        # training example. Then, take the mean of the two
        # squared differences to get the mse
        for i in range(k):
            differences = nn.functional.mse_loss(X_train,
                                                 m[i].expand_as(X_train),
                                                 reduction='none')
            list_mse.append(torch.sum(differences, dim=1,
                                      dtype=torch.float64))

        # Stack the list of lists as a k x N tensor
        list_mse_torch = torch.stack(list_mse, dim=0)

        # Calculate the minimum mse along the column of each example
        # Signifies the centroid to which given example is closest
        # We now get a N sized tensor
        list_mse_torch_min,_ = torch.min(list_mse_torch, dim=0)

        # Take mean of tensor to get loss function of kmeans algo
        L_train = torch.mean(list_mse_torch_min, dtype=torch.float64)

        # Perform backward propagation
        optimizer.zero_grad()
        L_train.backward()
        optimizer.step()

    # After all epochs are done, detach the gradient of loss and centroids
    L_train = L_train.detach().numpy()
    m = m.detach().numpy()

    return L_train, m
```

In [36]:
```
'''
Given a dataset to test and list of centroids, it calculates average loss
'''
def evaluate(test_data, m):
    # Initialize number of examples and loss value
    num_examples = len(test_data)
    loss = 0.0

    # Perform the following for every example in the test dataset:
    # Calculate mse of the point from each centroid, then from this,
    # Take the minimum and add it to the loss value
    for example in test_data:
        mse_list = [np.sum((example-centroid)**2) for centroid in m]
        loss += np.min(mse_list)

    # Finally return average loss over the entire dataset
    return loss/num_examples
```

In [37]:
```
def get_association(test_data, m):
    # Get number of centroids, examples, dimensions and initialize losses
    num_cluster = len(m)
    N, d = test_data.shape
    L_k = np.zeros((N, num_cluster))

    # For each cluster, calculate mse of each point in test dataset
    for k in range(num_cluster):
        L_k[:,k] = [np.sum((example-m[k])**2) for example in test_data]

    #Assign to the nearest cluster.
    index = np.argmin(L_k, axis = -1)
    index = index.reshape(len(index), 1)
    return index
```

Testing Functions

In [38]:
```
'''
This testing function is given by teaching team to test our
implementation of kmeans algorithm
'''
def test_pytorch(train_data, test_data, k=5):
    L,m = train_kmean_torch(train_data, k)
    index = get_association(test_data, m)
    new_X = np.concatenate((test_data, index), axis = 1)

    print ("PyTorch test score:", evaluate(test_data, m))

    color_list = ['g', 'b', 'm', 'y', 'c']
    for i in range(len(m)):
        tmp = new_X[new_X[...,-1] == i]
        plt.scatter(tmp[:,0], tmp[:,1], c=color_list[i])
```

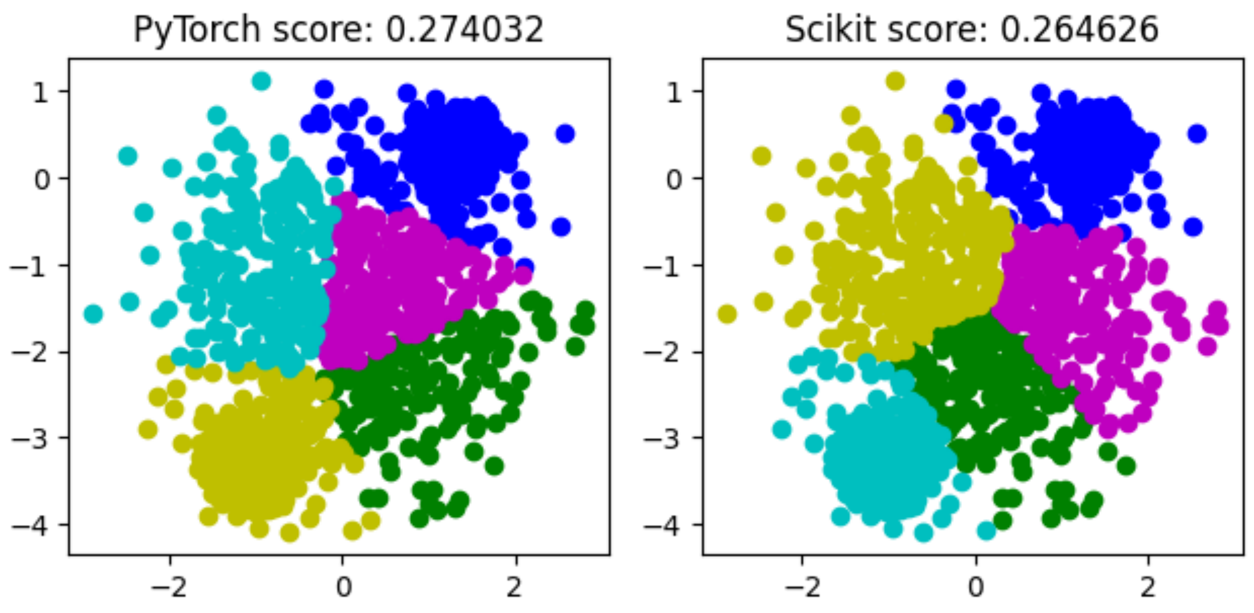
In [39]:
```
'''
This testing function is given by teaching team to scikit implementation
of kmeans algorithm
'''
def test_sckitlearn(train_data, test_data, k=5):
    kmeans = KMeans(n_clusters=k, max_iter=5000,
                    algorithm='lloyd', n_init=10)
    kmeans = kmeans.fit(train_data)

    index = kmeans.predict(test_data)
    index = index.reshape(len(index), 1)
    new_X = np.concatenate((test_data, index), axis = 1)

    print ("Scikit-learn test score:",
           evaluate(test_data, kmeans.cluster_centers_))

    color_list = ['g', 'b', 'm', 'y', 'c']
    for i in range(len(kmeans.cluster_centers_)):
        tmp = new_X[new_X[...,-1] == i]
        plt.scatter(tmp[:,0], tmp[:,1], c=color_list[i])
```

Test Results

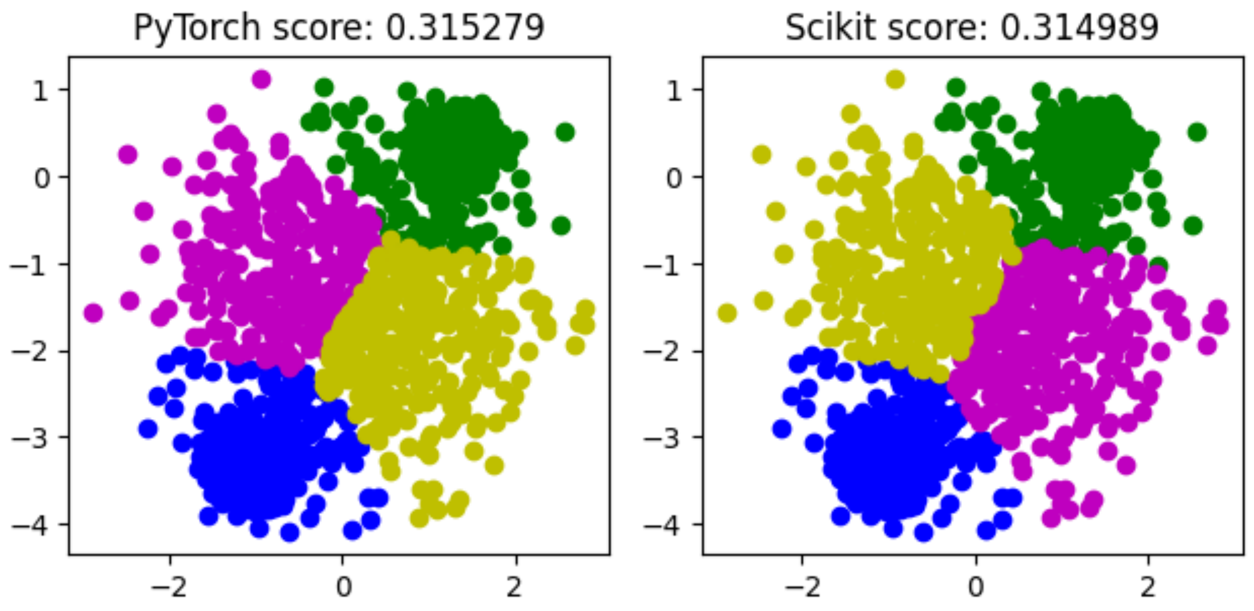
In [40]:
```
train_data, test_data = load_data()
```

In [41]:
```
'''
Custom tester functions that are an extension of those provided by
the teaching team, that plot the two graphs side-by-side
'''
def custom_test_pytorch(train_data, test_data, k=5):
    L,m = train_kmean_torch(train_data, k)
    index = get_association(test_data, m)
    new_X = np.concatenate((test_data, index), axis = 1)

    score = evaluate(test_data, m)
    plt.title("PyTorch score: %lf" % (score))

    color_list = ['g', 'b', 'm', 'y', 'c']
    for i in range(len(m)):
        tmp = new_X[new_X[...,-1] == i]
        plt.scatter(tmp[:,0], tmp[:,1], c=color_list[i])

def custom_test_sckitlearn(train_data, test_data, k=5):
    kmeans = KMeans(n_clusters=k, max_iter=5000,
                    algorithm='lloyd', n_init=10)
    kmeans = kmeans.fit(train_data)

    index = kmeans.predict(test_data)
    index = index.reshape(len(index), 1)
    new_X = np.concatenate((test_data, index), axis = 1)

    score = evaluate(test_data, kmeans.cluster_centers_)
    plt.title("Scikit score: %lf" % (score))

    color_list = ['g', 'b', 'm', 'y', 'c']
    for i in range(len(kmeans.cluster_centers_)):
        tmp = new_X[new_X[...,-1] == i]
        plt.scatter(tmp[:,0], tmp[:,1], c=color_list[i])

def custom_kmeans_tester(train_data, test_data, k=5):
    plt.subplot(1, 2, 1, aspect=1)
    custom_test_pytorch(train_data, test_data, k=k)
    plt.subplot(1, 2, 2, aspect=1)
    custom_test_sckitlearn(train_data, test_data, k=k)
    plt.tight_layout()
    plt.show()
```

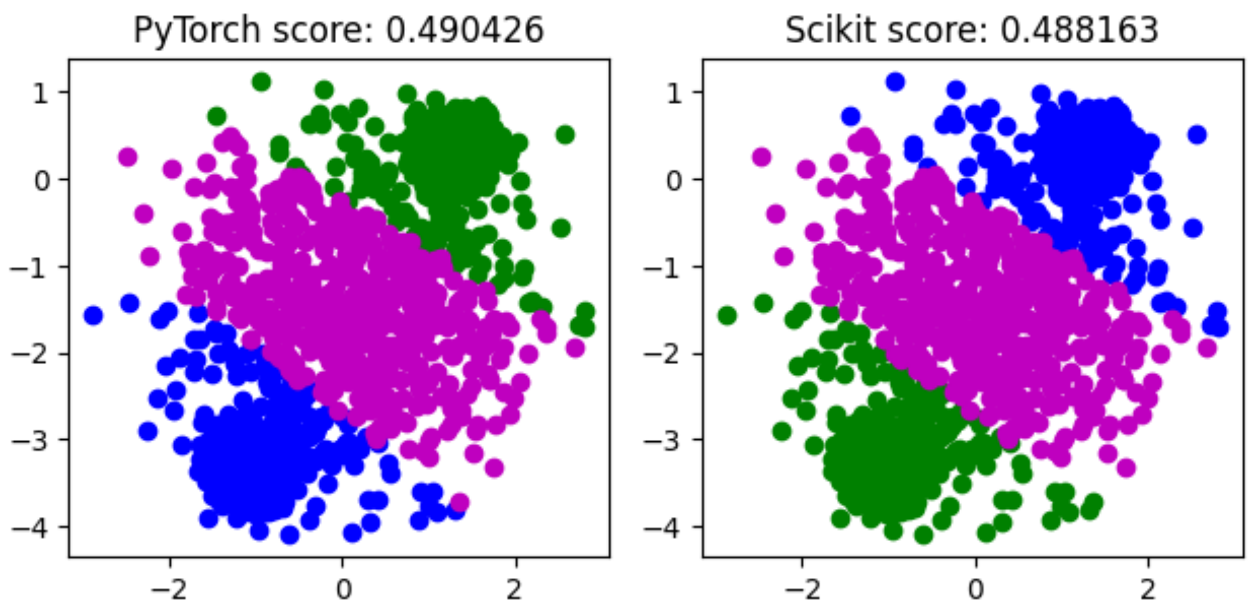
In [42]:
```
'''
Result of custom tester for k=5 clusters.
Most of the times, our score exceeds the scikit-score.
Sometimes, our pytorch implementation yields a better score.
The arrangement of regions in scikit implementation stays similar
    through several runs, but our implementation changes very often
'''
custom_kmeans_tester(train_data, test_data, k=5)
```



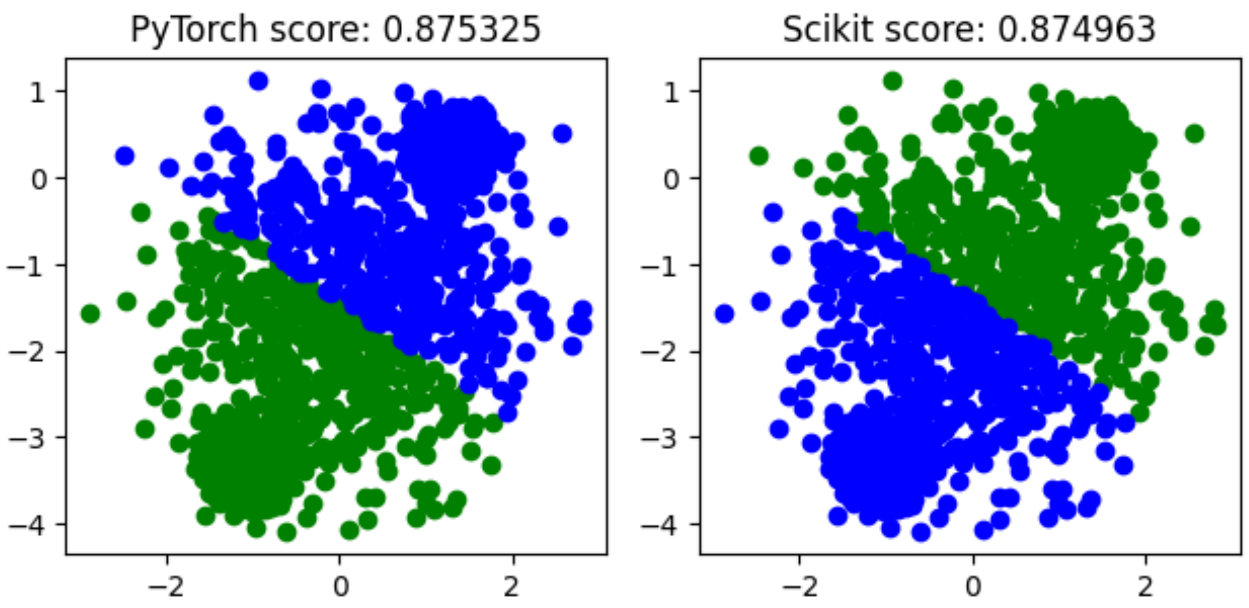
In [43]:
```
'''
Result of custom tester for k=4 clusters.
Trends between scores very similar to those for k=5
The arrangement of regions seems to be the same for both
    implementations throughout.
'''
custom_kmeans_tester(train_data, test_data, k=4)
```



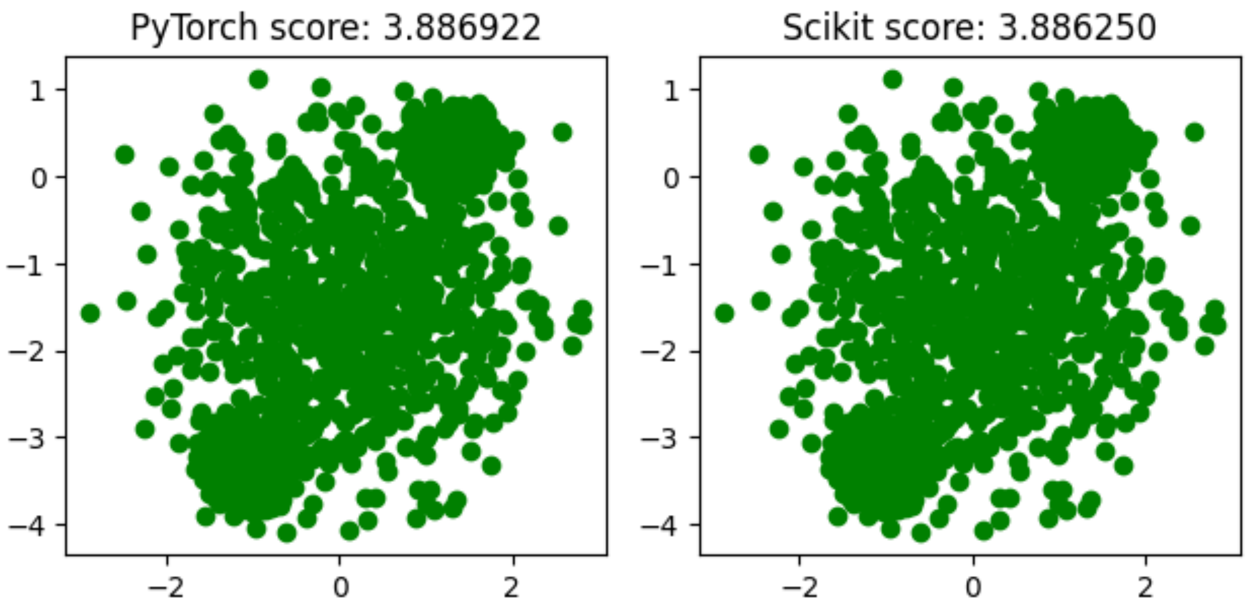
In [44]:
```
'''
Result of custom tester for k=3 clusters.
Scores for scikit are almost always better than ours.
The arrangement of regions seems to be the same for both
    implementations throughout.
'''
custom_kmeans_tester(train_data, test_data, k=3)
```



In [45]:
```
'''
Result of custom tester for k=2 clusters.
Scores for scikit are almost always better than ours.
The arrangement of regions seems to be the same for both
    implementations throughout.
'''
custom_kmeans_tester(train_data, test_data, k=2)
```



In [46]:
```
'''
Result of custom tester for k=1 clusters.
Scores for scikit are almost always better than ours.
The arrangement of regions seems to be the same for both
    implementations throughout.
'''
custom_kmeans_tester(train_data, test_data, k=1)
```



Part 2: Mixture of Gaussians

Import Modules

In [47]:
```
import numpy as np
import sklearn
import matplotlib.pyplot as plt
import scipy.stats as stats
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.distributions as D
from scipy.stats import truncnorm
```

Helper Functions

In [48]:
```
'''
Helper function given by teaching team to load data from data2D.npy.
'''
def load_data():
    X =np.load('data2D.npy')
    valid_batch = int(len(X) / 3.0)
    np.random.seed(45689)
    rnd_idx = np.arange(len(X))
    np.random.shuffle(rnd_idx)
    val_data = X[rnd_idx[:valid_batch]]
    data = X[rnd_idx[valid_batch:]]

    return data, val_data
```

In [49]:
```
'''
This function generates a truncated normal distribution between -ve and
+ve threshold value
'''
def truncated_normal(size, threshold=1):
    values = truncnorm.rvs(-threshold, threshold, size=size)
    return values
```

In [50]:
```
'''
The inputs to the function are:
1] X: N examples within the dataset, stored as a tensor of d=2 coordinates
2] MU: Tensor of k centroids, each element having d=2 coordinates

This function calculates the distance between each example in X and each
centroid in MU, after unsqueeze happens, and returns these pair-wise
distances as a tensor of N x k dimension
'''
def distanceFunc(X, MU):
    # Add an extra dimension at the end to make the shape N x d x 1
    X1 = torch.unsqueeze(X, -1)

    # Add extra dim in the start of transpose to make the shape 1 x d x k
    MU1 = torch.unsqueeze(MU.T, 0)

    # Calculate the squared euclidean-distance between all combinations
    # of input example and cluster center
    pair_dist = torch.sum((X1 - MU1)**2, 1)
    return pair_dist
```

In [51]:
```
'''
The inputs to the function are:
1] X: N examples within the dataset, stored as a tensor of
      d=2 coordinates
2] mu: Tensor of k centroids, each element having d=2 coordinates
3] sigma: Tensor of k values, each representing standard deviation
          of the kth cluster

This function calculates the log of guassian distributions's pdf
for each possible pair of centroid and example in the dataset,
and returns it as a tensor of dimensions N x k
'''
def log_GaussPDF(X, mu, sigma):
    # Get the number of dimensions, here dim will be 2
    dim = X.shape[-1]

    # Create tensor containing pi value
    Pi = torch.tensor(float(np.pi))

    # Squares each value of standard deviation and then transposes
    # to make the shape d x k
    sigma_2 = (torch.square(sigma)).T

    # Get pairwise distance of each centroid-example pair
    diff = distanceFunc(X, mu)

    # The following lines emulate taking the logarithm
    # of gaussian distribution's pdf
    log_PDF = diff / sigma_2                # (x-m)^2 / sigma^2
    log_PDF += dim * torch.log(2 * Pi)      # + log(2*pi)
    log_PDF += dim * torch.log(sigma_2)     # + log(sigma^2)
    log_PDF *= -0.5                         # Finally, * for sqrt and 1/2
    return log_PDF
```

In [52]:
```
'''
The inputs to the function are:
1] log_PDF: N x k tensor with log of gaussian pdfs of all
            example-centroid pairs
2] log_pi: Tensor of k values containing probabilities assigned
           to each cluster

This function calculates and returns two things:
1] The log of joint probability of each example being assigned
to each cluster as a tensor of dimensions N x k
2] The log of marginal probability for each of the N examples
i.e. sum of joint probabilities along each column
for that particular example. It has N elements.
'''
def log_posterior(log_PDF, log_pi):
    # Calculate log of joint pdf of each point w.r.t. each component
    # of shape N x k
    log_joint = log_PDF + log_pi.T

    # Calculate log of marginal pdf for each point
    # of length N
    log_marginal = torch.logsumexp(log_joint,dim=1)

    return log_joint, log_marginal
```

Training Helpers

In [57]:
```
def train_gmm(train_data, test_data, k = 5,
              epoch=1000, init_kmeans=False):
    # Load the data
    X_train = torch.from_numpy(train_data)
    X_test = torch.from_numpy(test_data)

    # Initialize logits depending on value of init_kmeans flag
    # If true, use kmeans to get better than random cluster centroids
    if init_kmeans:
        logits = torch.ones(k, requires_grad=True)
        kmeans = KMeans(n_clusters=k, max_iter=5000, n_init=10)
        kmeans = kmeans.fit(train_data)
        mu = torch.tensor(kmeans.cluster_centers_, requires_grad=True)
        lr = 0.005
    else:
        logits = torch.rand(k, requires_grad=True)
        mu = torch.randn((k,X_train.shape[1]), requires_grad=True)
        lr = 0.005

    # Initialize standard deviations
    sigma = np.abs(truncated_normal((k,1), threshold=1))
    sigma = torch.tensor(sigma,requires_grad=True)
    optimizer = torch.optim.Adam([logits, mu, sigma],
                                 lr=lr,
                                    betas=(0.9, 0.99),
                                    eps=1e-5)

    # Train the model
    for i in range(epoch):
        logpi = F.log_softmax(logits, dim=0)

        # Get log of gaussian pdfs and then get the marginal pdfs
        log_PDF = log_GaussPDF(X_train, mu, sigma)
        _, log_marginal = log_posterior(log_PDF, logpi)

        # Compute the marginal mean as loss value.
        loss = -log_marginal.mean()

        # Update parameters using back-propagation on the loss fn
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Check performance on Test data
    logpi = F.log_softmax(logits, dim=0)

    # log_GaussPDF and log_posterior
    log_PDF = log_GaussPDF(X_test, mu, sigma)
    log_joint_test, log_marginal = log_posterior(log_PDF, logpi)
    test_loss = -log_marginal.mean()

    # Detach all gradient functions and return final trained values
    test_loss = test_loss.detach().numpy()
    log_joint_test = log_joint_test.detach().numpy()
    pi = torch.exp(logpi).detach().numpy()
    mu = mu.detach().numpy()
    sigma = sigma.detach().numpy()

    return test_loss, log_joint_test, pi, mu, sigma
```

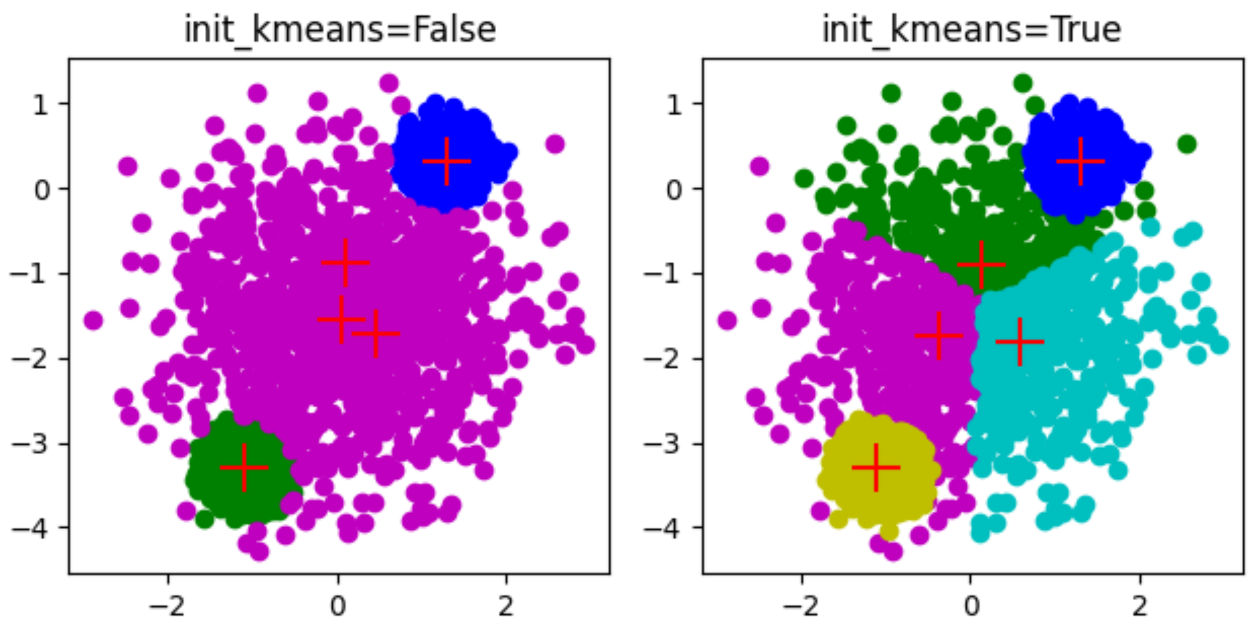
Testing Functions

In [54]:
```
def test_GMM(k = 5, init_kmeans=False):
    train_data, test_data = load_data()
    test_loss, log_joint_test, pi, mu, sigma = train_gmm(train_data,
                                                          test_data,
                                                          k,
                                                          init_kmeans=
                                                          init_kmeans)

    index = log_joint_test.argmax(axis=1)
    index = index.reshape(len(index), 1)
    new_X = np.concatenate((test_data, index), axis = 1)

    color_list = ['g', 'b', 'm', 'y', 'c']
    for i in range(len(mu)):
        tmp = new_X[new_X[...,-1] == i]
        plt.scatter(tmp[:,0], tmp[:,1], c=color_list[i])
    plt.scatter(mu[:,0], mu[:,1], s=300, c='r', marker = '+')
```

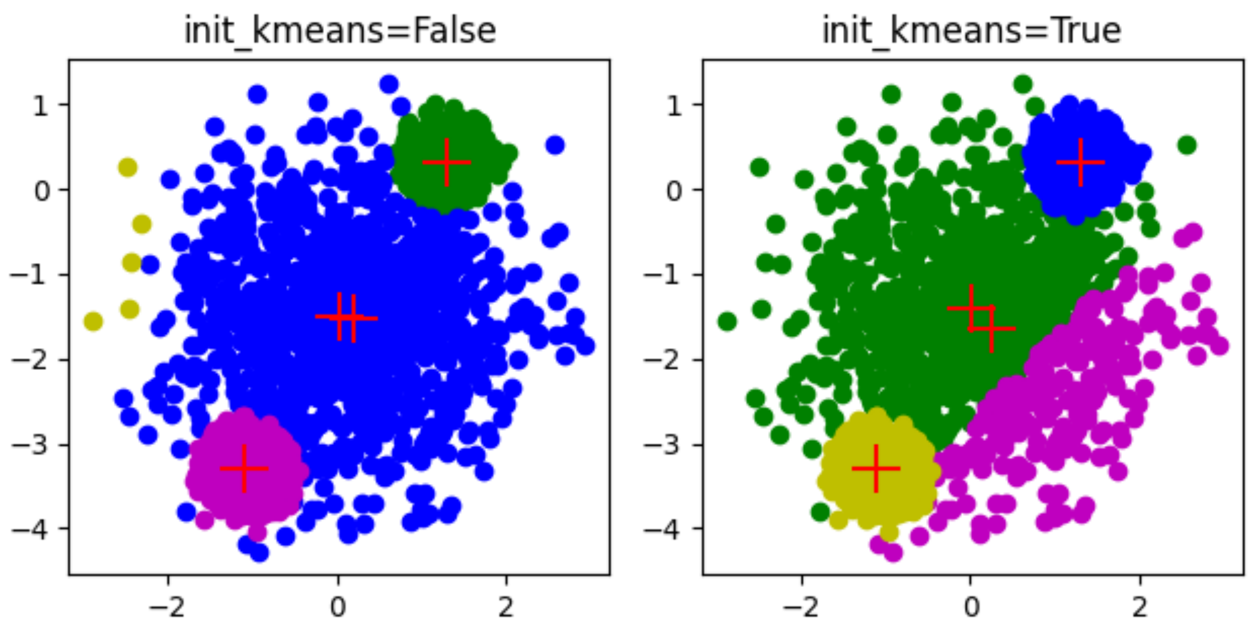
Test Results

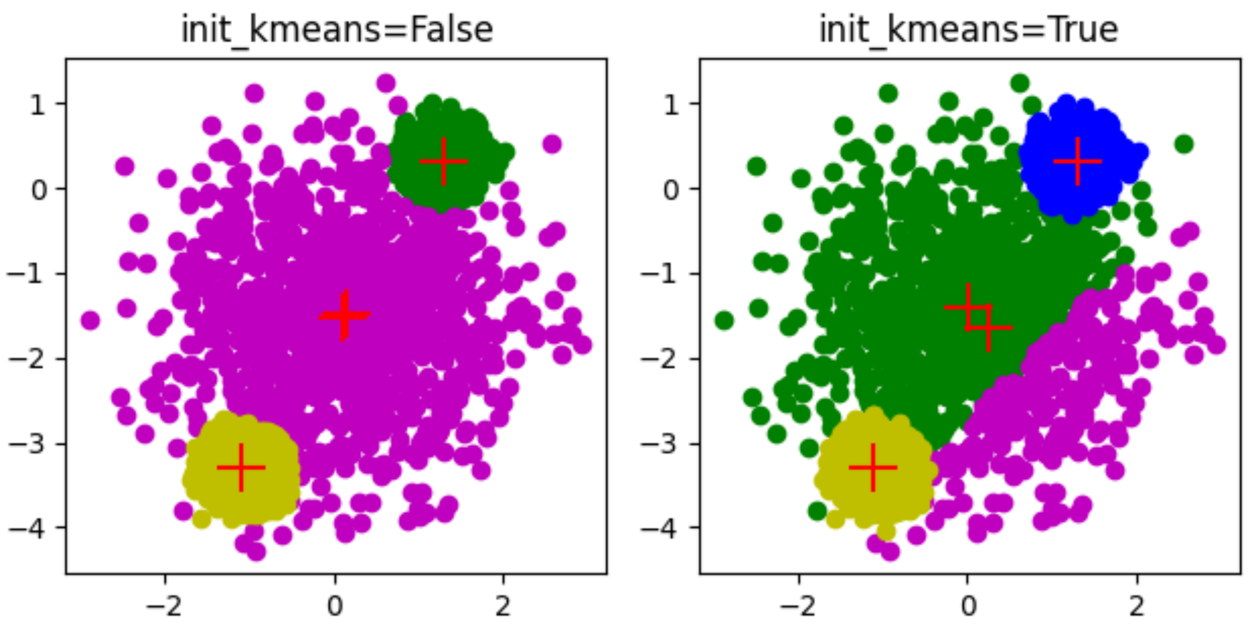
In [55]:
```
'''
Custom tester functions that are an extension of those provided
by the teaching team, that plot the two graphs side-by-side
'''
def custom_test_GMM(k = 5, init_kmeans=False):
    train_data, test_data = load_data()
    test_loss, log_joint_test, pi, mu, sigma = train_gmm(train_data,
                                                          test_data,
                                                          k,
                                                          init_kmeans=
                                                          init_kmeans)

    index = log_joint_test.argmax(axis=1)
    index = index.reshape(len(index), 1)
    new_X = np.concatenate((test_data, index), axis = 1)

    if init_kmeans: plt.title("init_kmeans=True")
    else: plt.title("init_kmeans=False")

    color_list = ['g', 'b', 'm', 'y', 'c']
    for i in range(len(mu)):
        tmp = new_X[new_X[...,-1] == i]
        plt.scatter(tmp[:,0], tmp[:,1], c=color_list[i])
    plt.scatter(mu[:,0], mu[:,1], s=300, c='r', marker = '+')

def custom_gmm_tester(k=5):
    plt.subplot(1, 2, 1, aspect=1)
    custom_test_GMM(k=k, init_kmeans=False)
    plt.subplot(1, 2, 2, aspect=1)
    custom_test_GMM(k=k, init_kmeans=True)
    plt.tight_layout()
    plt.show()
```

In [58]:
```
'''
Result of custom tester for k=5 clusters.
The one without init_kmeans is unable to find a good clustering,
whereas that with init_kmeans=True makes well-defined clusters.
Therefore, the right one is better
'''
custom_gmm_tester(5)
```

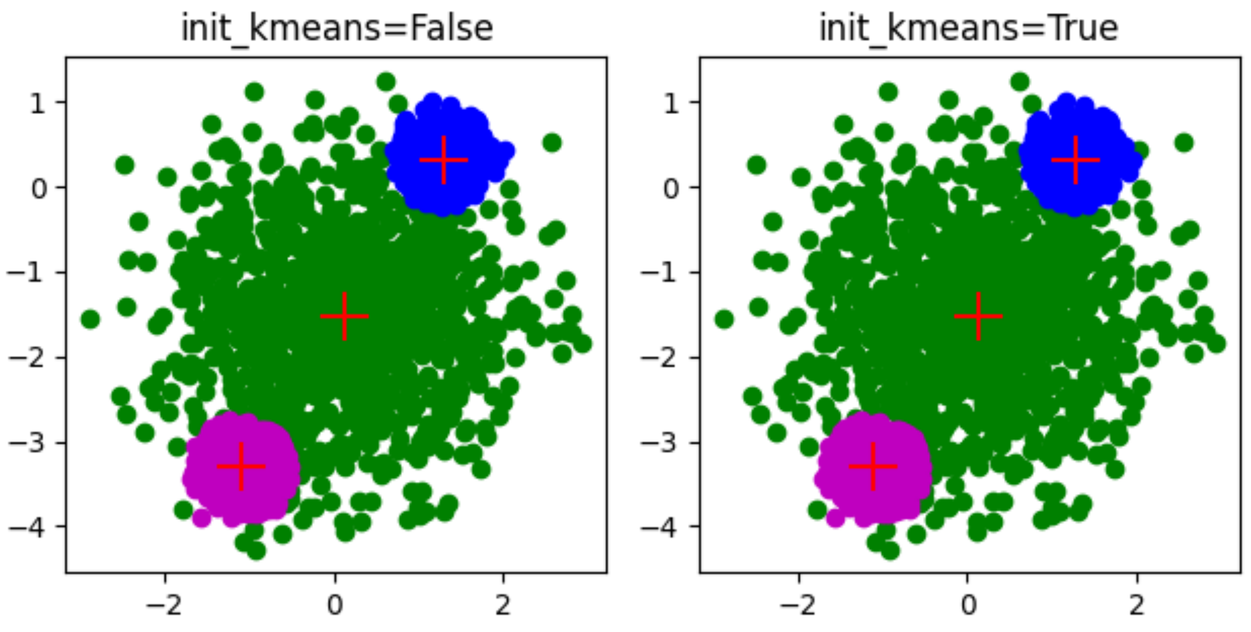


```
In [66]: '''
         Result of custom tester for k=4 clusters.
         The one without init_kmeans is barely able to make 4 clusters,
         whereas that with init_kmeans=True makes well-defined clusters,
         although the cluster centers are not perfect.
         Looks like k=5 is the ideal number of clusters.
         '''
         custom_gmm_tester(4)
```

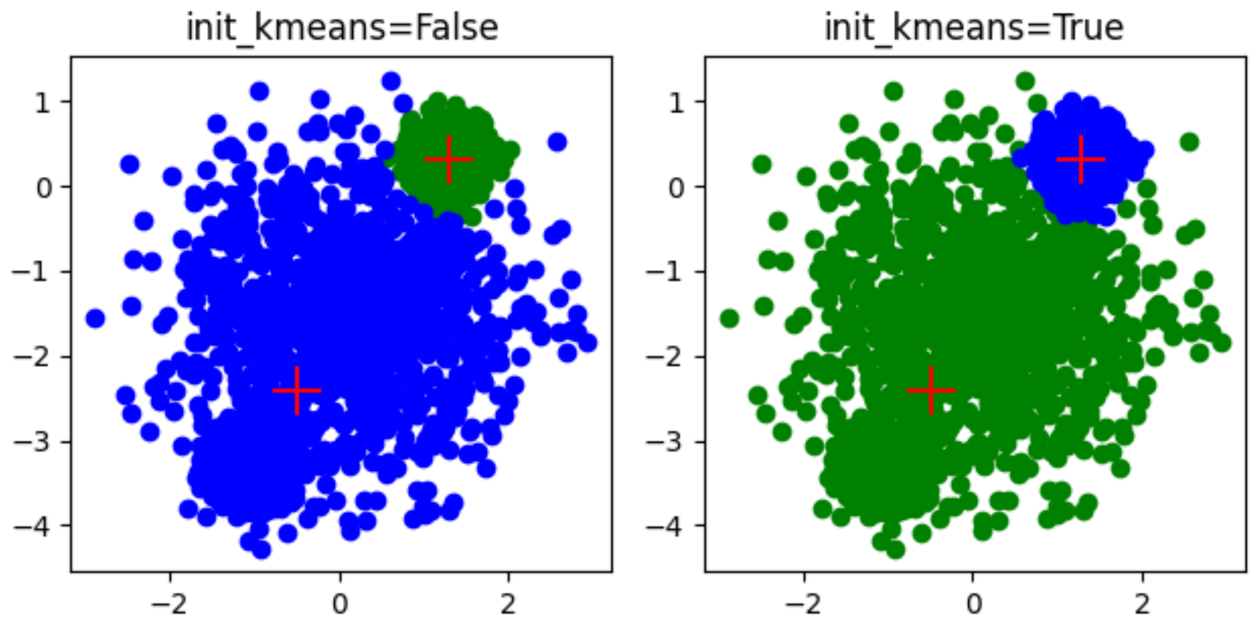




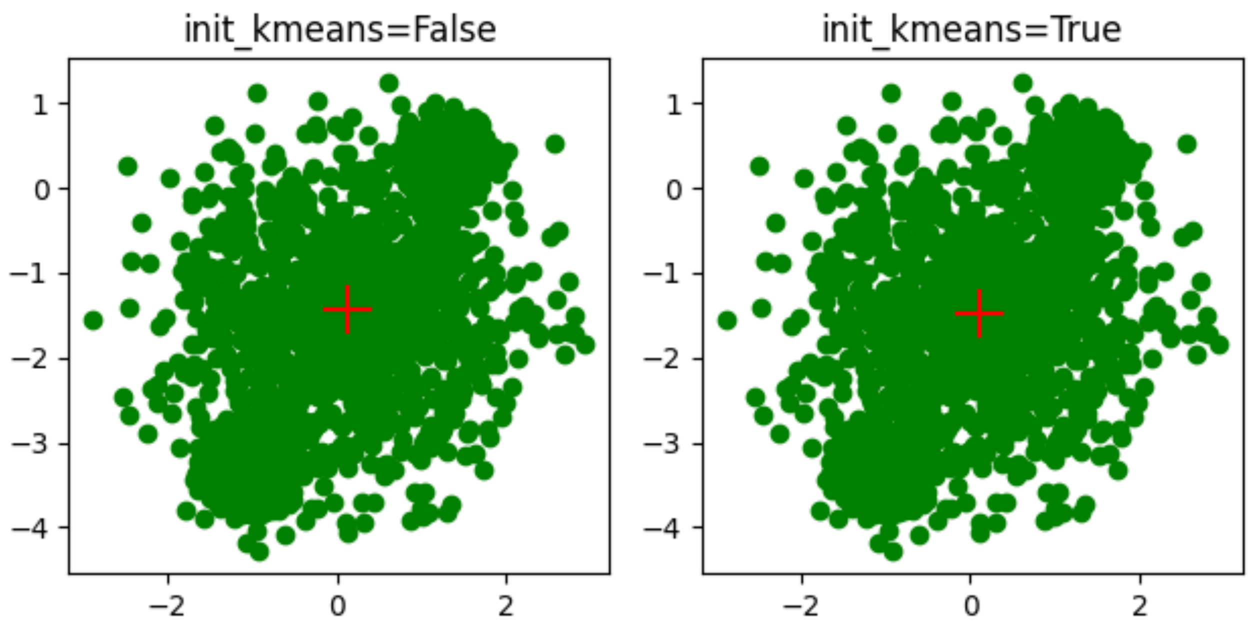
In [60]:
```
'''
Result of custom tester for k=3 clusters.
Both graphs are very similar and initializing through kmeans
does not give any visible advantage.
'''
custom_gmm_tester(3)
```



In [61]:
```
'''
Result of custom tester for k=2 clusters.
Both graphs are very similar and initializing through kmeans
does not give any visible advantage.
'''
custom_gmm_tester(2)
```



In [62]:
```
'''
Result of custom tester for k=1 clusters.
Both graphs are very similar and initializing through kmeans
does not give any visible advantage.
'''
custom_gmm_tester(1)
```


init_kmeans=False
init_kmeans=True
1
0
−1
−2
−3
−4
−2
0
2