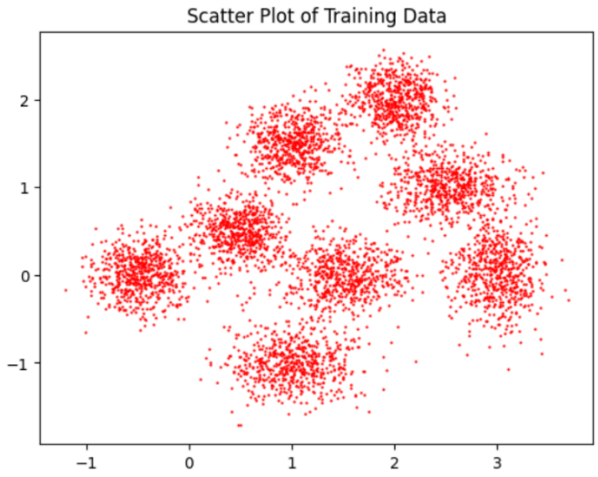
Visualization of the Training Set



It looks like having 8 “groups”, so I will set the K to 8 in the following programming.

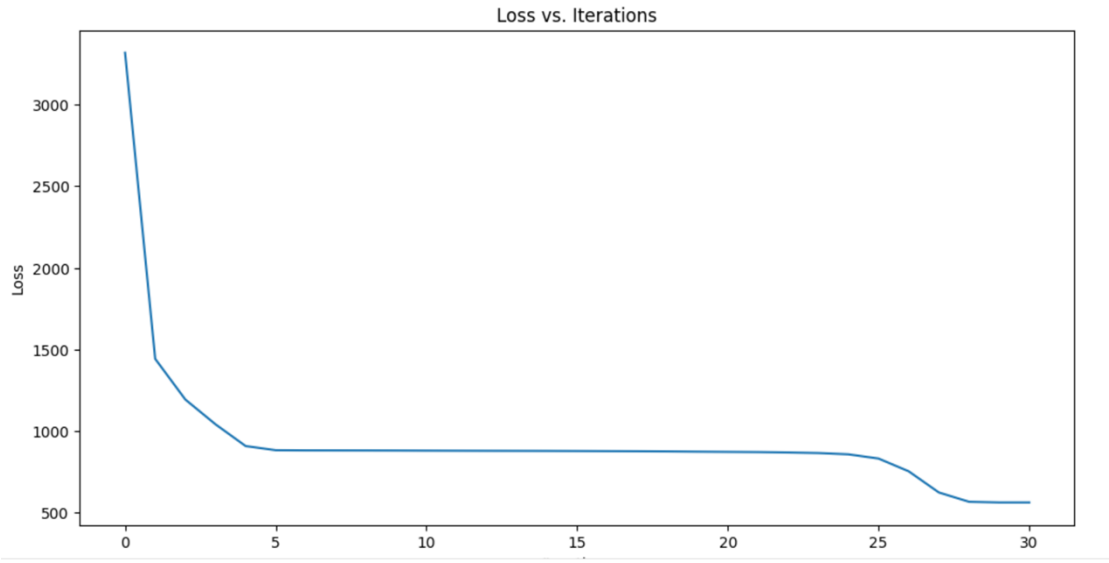
Task 1:

Subtask1:

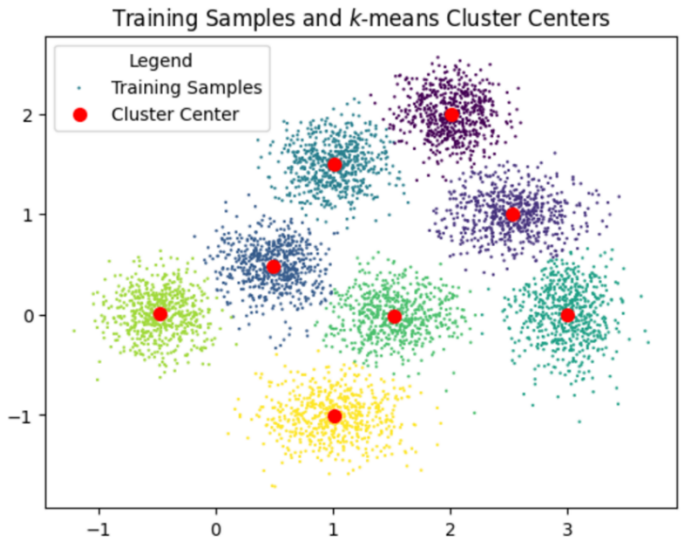
```python
def solve_k_means(
        x: ndarray, c: ndarray, *, max_step: int=10000
) -> Tuple[ndarray, ndarray, list]:
        """K-means solver. Given K initial cluster centers, update c iteratively.
        Note:
                - You may change the function signature.
                - Please try your best to write vectorized code (i.e. avoid for loops over indices).
                - Design some criterion for stopping the iteration.
        Args:
                x (ndarray[float]): shape [N, D], storing N data samples. D is the feature dimension.
                c (ndarray[float]): shape [K, D], storing K initial cluster centers.
                max_step (int): Maximum number of steps in $k$-means iteration.
        Returns:
                c (ndarray[float]): shape [K, D], updated K cluster centers after iterations.
                index (ndarray[int]): shape [N], the index of the nearest cluster center of
                        each sample, in range {0, ..., K - 1}.
        """
        N, D = x.shape
        K = c.shape[0]
        losses = []
        index = np.zeros(N, dtype=int)
        prev_c = np.zeros_like(c)
        for step in range(max_step):
                distances = np.linalg.norm(x[:, np.newaxis] - c, axis=2)    # Shape (N, K)
                new_index = np.argmin(distances, axis=1)
                if np.array_equal(index, new_index):
                        break
                index = new_index
                for k in range(K):
                        c[k, :] = np.mean(x[index == k], axis=0)
                loss = np.sum((x - c[index])**2)
                losses.append(loss)
        return c, index, losses
```

First assign the variables and then update accordingly. Also, keep track of the loss history for convenience.

Subtask2:



Subtask3:



Subtask4:

It runs at least 10-15times until it converges to a global minimum.

It converges seldomly.

Task2:

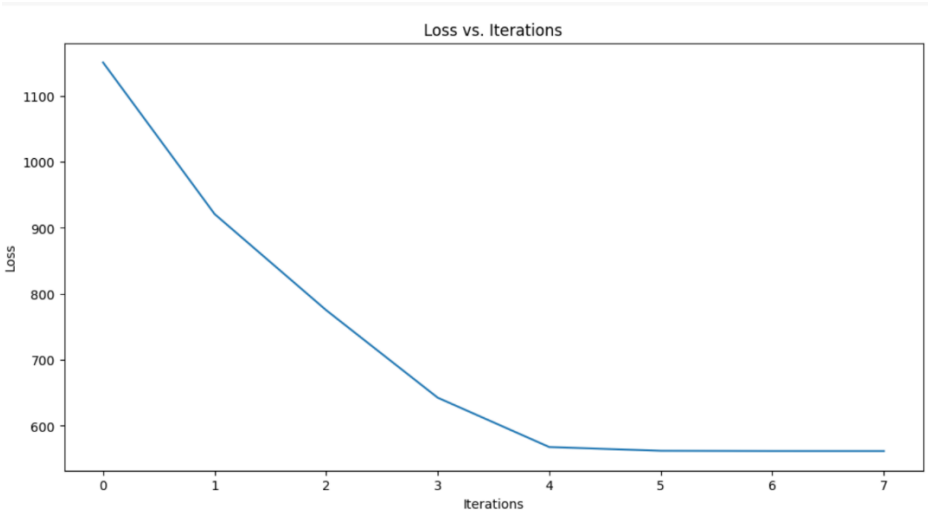
Subtask1:

```
import random
def k_means_pp_initialization(x: ndarray, K: int) -> ndarray:
    """K-means++ initialization method.
    Args:
        x (ndarray): shape [N, D], storing N data samples. D is the feature dimension.
        K (int): Number of cluster centers.
    Returns:
        c (ndarray): shape [K, D], K initial cluster centers generated by K-means++.
    """
    N, D = x.shape
    centers = np.zeros((K, D))

    idx = np.random.choice(N)
    centers[0] = x[idx]
    losses = []

    sq_distances = np.linalg.norm(x - centers[0], axis=1) ** 2

    for k in range(1, K):
        probabilities = sq_distances / np.sum(sq_distances)
        idx = np.random.choice(N, p=probabilities)
        centers[k] = x[idx]

        new_sq_distances = np.linalg.norm(x - centers[k], axis=1) ** 2
        sq_distances = np.minimum(sq_distances, new_sq_distances)

    return centers
```

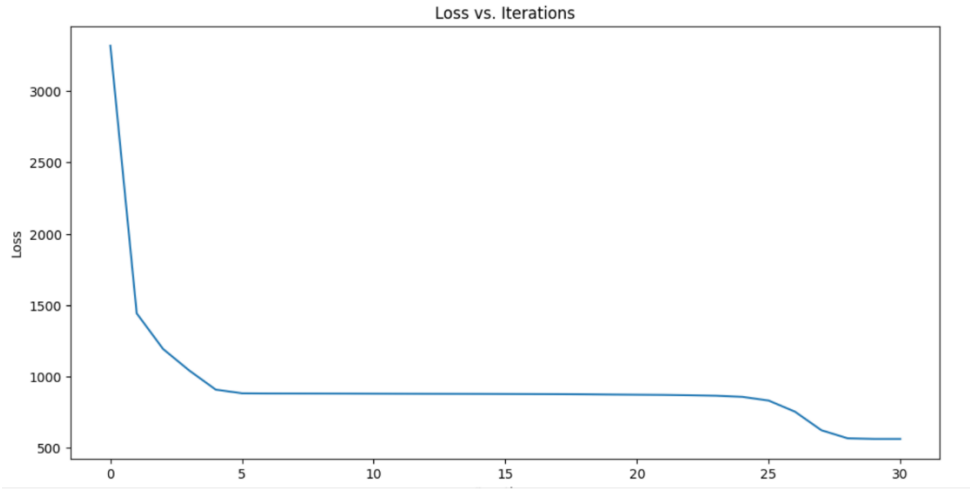
Randomly choose the first center, then choose the next center with a probability proportional to squared distance and update squared distances for each point.

Subtask2:

k -means++ initialization:



standard normal initialization:



Subtask3:

k -means++ initialization converges more fastly.

Task3:

Subtask1:

```
def e_step(x, weights, means, covariances):
    n = x.shape[0]
    K = len(weights)
    responsibilities = np.zeros((n, K))
    for k in range(K):
        responsibilities[:, k] = weights[k] * multivariate_normal(means[k], covariances[k]).pdf(x)
    responsibilities /= responsibilities.sum(axis=1, keepdims=True)
    return responsibilities
```

Compute responsibilities for each component of the GMM.

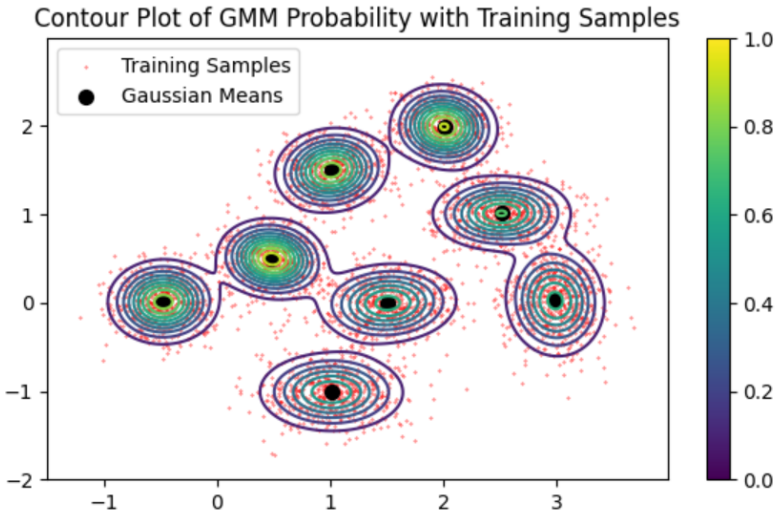
```python
def m_step(x, responsibilities):
    n, d = x.shape
    K = responsibilities.shape[1]
    n_k = responsibilities.sum(axis=0)
    weights = n_k / n
    means = np.dot(responsibilities.T, x) / n_k[:, None]
    covariances = np.zeros((K, d, d))
    for k in range(K):
        x_centered = x - means[k]
        covariances[k] = np.dot(responsibilities[:, k] * x_centered.T, x_centered) / n_k[k]
    return weights, means, covariances
```

Update the parameters of the GMM

```python
def expectation_maximization(x, K, max_iter=100, tol=1e-4):
    weights, means, covariances = initialize_parameters(x, K)
    log_likelihood_history = []

    for _ in range(max_iter):
        responsibilities = e_step(x, weights, means, covariances)
        weights, means, covariances = m_step(x, responsibilities)
        ll = log_likelihood(x, weights, means, covariances)
        if log_likelihood_history and np.abs(ll - log_likelihood_history[-1]) < tol:
            break
        log_likelihood_history.append(ll)

    return means, covariances, weights, log_likelihood_history[-1]
```

Run the EM algorithm to fit a Gaussian Mixture Model.

Subtask2:



Subtask3:

```
Log-likelihood on the training set: -9693.226261958467
Log-likelihood on the development set: -1678.018134965429
```