Q 1.1 Expectation Maximisation

For **complete data**: 1) Here assume documents use $\{d_1, d_x\}$ and their corresponding latent variables are $\{z_1, \ldots z_N\}$.

$$P(d_1, z_1, \ldots, d_v, z_N) = \prod_{n=1}^{N} \prod_{K=1}^{K} \left(Qk_n \prod_{w \in A} u_{K_{n,w}}^{c(w,d)_{n_k}^{Z}} \right)$$

where $z_n = (z_{n_1}, \ldots, z_{n_k}) \rightarrow$ The log-likelihood is:

$$\ln P(d_1, z_1, \ldots . dd_N, z_n) = \\ \sum_{n=1}^{N} \sum_{k=1}^{K} z_{n,k} \left(\ln \Phi_{k_n} + \sum_{\omega \in R} ((\omega, d) \ln \mu_{kn}) \right.$$

$\rightarrow$ So after maximizing loy-likelihood the parametar

$$Q_k = \frac{N_k}{N} \text{ where } N_k = \sum_{n=1}^{N} Z_{nk}$$

The word proportion parameters for each cluster:

$$\mu_{k,w} = \frac{\sum_{n=1}^{N} z_{n,k} C(w, d_n)}{\sum_{w \in A} \sum_{n=1}^{\infty} z_{n,k} C(w', d_n)}$$

$\rightarrow$ For **incomplete date**: Here, document clusters are not given, so Zn is latent, so the probability of observed document is

$$\begin{aligned} p(d_1, \ldots d_N) &= \prod_{n=1}^{N} p(d_n) \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} p(z_{n,k} = 1, d_n) \\ &= \prod_{n=1}^{N} \sum_{k=1}^{K} \left(a_k \prod_{w \in A} \mu_{k,w}^{c(w,d_n)} \right) \end{aligned}$$

- So the log-likelihood is:

$$\text{So the log-likelihood is:}$$

$$\begin{aligned} \ln p(d_1, \ldots d_N) &= \sum_{n=1}^{N} \ln p(d_n) \\ &= \sum_{n=1}^{N} \ln \sum_{k=1}^{K} p(Z_{n,k} = 1, d_n) \\ &= \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \left(Q_k \prod_{w \in A} \mu_{k,w}^{c(w,d_n)} \right) \end{aligned}$$

$\rightarrow$ Where, Q is probability vector of size K, μ_k is word proportion $\sum_{w \in A} \mu_{k,w} = 1$ is word proportion vector, $C(w, d)$ is number of occurences of word dW in document d.

$\rightarrow$ Without class labels, direct optimization is hard because the likelihood function involves sums inside the algorithms ($\log$ of sums), which is difficult to optimize directly.

2) The Expectation - Maximization (EM) algorithms is a two-step iterative method used to estimate model parameters when data includes unobserved variables:

- Expectation-step (E-step): calculate the probarbilities of the latent variables based on the observed data and current parameter estimates
- Maximization-step: update the model parameter to maximise the likelihood of the date given the probabilities estimated in the E-step.
- This process repeats untill the parameters converge, effectively handling incomplete date.

Q 1.2 Expectation and Maximization steps of the (soft)-EM algorithm for Document Clustering

Q-1.2

The loy-likelihood function of document cluster probability is:

$$\ln p\left(d_1, \ldots dn\right)=\sum_{n=1}^N \ln p\left(d_n\right)$$
$$=\sum_{n=1}^N \ln \sum_{k=1}^k p\left(z_{n,k}=1, d_n\right)$$
$$=\sum_{n=1}^N \ln \sum_{k=1}^k\left(\phi_k \prod_{w \in A} \mu_{k,w}^{c(w,d)}\right)$$

Now, Q function for EM algorithm;

$$Q\left(\theta, \theta^{\text{old}}\right)=\sum_{n=1}^N \sum_{k=1}^k p\left(z_{n,k}=1 / d_n, \theta^{\text{old}}\right) \ln p\left(z_{n,k}=1, d_n, \theta\right)$$
$$=\sum_{n=1}^N \sum_{k=1}^k p\left(z_{n,k}=1 \mid d_n, \theta^{\text{old}}\right)$$
$$\left(\ln \Phi_k+\sum_{w \in A} c\left(w, d_d \ln \mu_k, w\right)\right.$$
$$=\sum_{n=1}^N \sum_{k=1}^k r\left(z_{n,k}\right)\left(\ln \phi_k+\sum_{w \in A} c\left(w, d_n\right) \ln \mu_k, w\right)\right)$$

where $\theta=\left(0, \mu_1, \ldots \mu_k\right)$ is collection of model parameter.

\documentclass{article}

\begin{document}

$$r(Z_{n,k}) = P(Z_{m,k} = 1 | d_m, \theta^{\text{old}}) \text{ (responsibility factors)}$$

- Here, the maximising components:

$$\Phi_k = \frac{N_k}{N} \quad \text{where } N_k = \sum_{n=1}^{N} r(Z_{n,k})$$

- The word proportion parameters for each cluster:

$$\mu_{k,w} = \frac{\sum_{n=1}^{N} r(Z_{n,k}) \cdot c(w, d_n)}{\sum_{w' \in A} \sum_{n=1}^{N} r(Z_{n,k}) \cdot c(w', d_n)}$$

- Choose the initial setting of parameters. While the convergence is not met:

$$\theta^{\text{old}} = (\Phi^{\text{old}}, \mu^{\text{old}}, N_k^{\text{old}})$$

  E-step: Set $\forall n, \forall k : r(Z_{n,k})$ based on θ^{old}

  M-step: Set θ^{new} based on $\forall n, \forall k : r(Z_{n,k})$

$$\theta^{\text{old}} \leftarrow \theta^{\text{new}}$$

- E-step: Posterior probability of the cluster assignment:

$$r(Z_{n,k}) = \frac{\Phi_k \prod_{w \in d} \mu_{k,w}^{c(w,d)}}{\sum_{j=1}^{K} \Phi_j \prod_{w \in d} \mu_{j,w}^{c(w,d)}}$$

\end{document}

M:-step: $\rightarrow$ update the parameters:

$$\Phi_k=\frac{\sum_{d \in D} \gamma\left(z_{d,k}\right)}{\sum d_{d \in D} \gamma\left(z_{d,k}\right) \sum_{w' \in V} c\left(w', d\right)}$$

$D=$ total number of Document. $V=$ vocubulury.

```
# from google.colab import drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Q 1.3 Load the data

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.preprocessing import normalize

with open("/content/drive/MyDrive/Task2A.txt", 'r') as file:
    text = file.readlines()
all([length == 2 for length in [len(line.split('\t')) for line in text]])
labels, articles = [line.split('\t')[0].strip() for line in text], [line.split('\t')[1].strip() for line in text]
docs = pd.DataFrame(data = zip(labels,articles), columns=['label', 'article'])
docs.label = docs.label.astype('category')
docs.head()
```

	label	article
0	sci.crypt	ripem frequently asked questions archive name ...
1	sci.crypt	ripem frequently asked questions archive name ...
2	sci.crypt	ripem frequently noted vulnerabilities archive...
3	sci.crypt	certifying authority question answered if you ...
4	sci.crypt	rubber hose cryptanalysis some sick part of me ...

Next steps: Generate code with docs View recommended plots New interactive sheet

Q 1.4 Implement the EM algorithm

```
# Create a DataFrame
docs = pd.DataFrame(data=zip(labels, articles), columns=['label', 'article'])
docs['label'] = docs['label'].astype('category')

# Convert the articles into TF-IDF vectors
vectorizer = TfidfVectorizer(stop_words='english', max_features=500)
X = vectorizer.fit_transform(docs['article']).toarray()

# Normalize the data
X = normalize(X)


import numpy as np

class SoftDocumentClustering:
    def __init__(self, K, tau_max=200, epsilon=0.01, random_state=None):
        self.K = K
        self.tau_max = tau_max
        self.epsilon = epsilon
        self.random_state = random_state
        np.random.seed(self.random_state)

    def log_sum_exp(self, x):
        c = np.max(x)
        return c + np.log(np.sum(np.exp(x - c)))

    def fit(self, X, verbose=False):
        N, D = X.shape
        # Normalize X to prevent numerical instability in dot product
        X = X / np.linalg.norm(X, axis=1, keepdims=True)

        self.Psi_hat_ = np.ones(self.K) / self.K
        self.Mu_hat_ = X[np.random.choice(N, self.K, replace=False)]
        r = np.zeros((N, self.K))
        tau = 0
        Mu_hat_old = self.Mu_hat_.copy()

        while tau < self.tau_max:
            if verbose:
                print(f"Iteration {tau}")

            for k in range(self.K):
                r[:, k] = np.dot(X, self.Mu_hat_[k])

            # Fix potential divide by zero by adding a small epsilon
            r[r == 0] = np.finfo(float).eps

            for n in range(N):
                r[n, :] = np.exp(np.log(r[n, :]) - self.log_sum_exp(np.log(r[n, :])))
```

```
            Nk_hat_ = r.sum(axis=0)
            self.Psi_hat_ = Nk_hat_ / N
            self.Mu_hat_ = (r.T @ X) / Nk_hat_.reshape(-1, 1)

            tau += 1
            if np.allclose(self.Mu_hat_, Mu_hat_old, rtol=self.epsilon):
                break
            Mu_hat_old = self.Mu_hat_.copy()

        if verbose:
            print(f"Converged in {tau} iterations")

        return self


    def predict_proba(self, X):
        # Predict the soft membership probabilities of each document for each cluster
        N = X.shape[0]
        r = np.zeros((N, self.K))
        for k in range(self.K):
            r[:, k] = np.dot(X, self.Mu_hat_[k])  # Cosine similarity with the centroids

        # Normalize with log-sum-exp trick
        for n in range(N):
            r[n, :] = np.exp(np.log(r[n, :]) - self.log_sum_exp(np.log(r[n, :])))

        return r

    def predict(self, X):
        r = self.predict_proba(X)
        return np.argmax(r, axis=1)


class HardDocumentClustering:
    def __init__(self, K, tau_max=200, epsilon=0.01, random_state=None):
        self.K = K  # Number of clusters
        self.tau_max = tau_max  # Maximum number of iterations
        self.epsilon = epsilon  # Convergence threshold
        self.random_state = random_state
        np.random.seed(self.random_state)

    def fit(self, X, verbose=False):
        N, D = X.shape
        # Initialize cluster memberships and centroids
        self.Psi_hat_ = np.ones(self.K) / self.K  # Equal cluster probabilities
        self.Mu_hat_ = X[np.random.choice(N, self.K, replace=False)]  # Randomly choose centroids
        z = np.zeros((N, self.K))  # Hard assignment matrix

        tau = 0
        Mu_hat_old = self.Mu_hat_.copy()

        while tau < self.tau_max:
            if verbose:
                print(f"Iteration {tau}")

            # E-step: Assign each document to the closest centroid (hard assignment)
            distances = np.dot(X, self.Mu_hat_.T)  # Using cosine similarity for assignment
            z = np.zeros_like(distances)
            z[np.arange(N), np.argmax(distances, axis=1)] = 1  # Hard assignment

            # M-step: Update cluster centroids
            Nk_hat_ = z.sum(axis=0)  # Sum of assignments for each cluster
            self.Mu_hat_ = (z.T @ X) / Nk_hat_.reshape(-1, 1)  # Update centroids

            tau += 1
            if np.allclose(self.Mu_hat_, Mu_hat_old, rtol=self.epsilon):
                break
            Mu_hat_old = self.Mu_hat_.copy()

        if verbose:
            print(f"Converged in {tau} iterations")

        return self
```

```
    def predict(self, X):
        # Predict the cluster assignments for each document
        distances = np.dot(X, self.Mu_hat_.T)  # Cosine similarity
        return np.argmax(distances, axis=1)  # Assign to closest centroid
```

Q 1.5 clusters K=4, and run the hard and soft clustering

```
# Set number of clusters
K = 4

# Run Soft Clustering with K=4
soft_clustering = SoftDocumentClustering(K=K, tau_max=100, epsilon=0.01, random_state=42)
soft_clustering.fit(X, verbose=True)
soft_clusters = soft_clustering.predict(X)
print("Soft Clustering Assignments (K=4):")
print(soft_clusters)

# Run Hard Clustering with K=4
hard_clustering = HardDocumentClustering(K=K, tau_max=100, epsilon=0.01, random_state=42)
hard_clustering.fit(X, verbose=True)
hard_clusters = hard_clustering.predict(X)
print("Hard Clustering Assignments (K=4):")
print(hard_clusters)
```

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Converged in 6 iterations
Soft Clustering Assignments (K=4):
[2 2 2 ... 3 3 3]
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Converged in 11 iterations
Hard Clustering Assignments (K=4):
[2 2 2 ... 3 3 3]
```

Q 1.6 PCA on the clusterings

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Check soft clustering assignments
print("Soft Clustering Assignments Sample: ", soft_clusters[:10])

# Check hard clustering assignments
print("Hard Clustering Assignments Sample: ", hard_clusters[:10])

# Step 1: Perform PCA on the TF-IDF vectors (X)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Step 2: Plotting the clusters from soft clustering
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Soft Clustering (K=4) - PCA Projection")
for cluster in range(K):
    mask = soft_clusters == cluster
    plt.scatter(X_pca[mask, 0], X_pca[mask, 1], label=f"Cluster {cluster}")
```

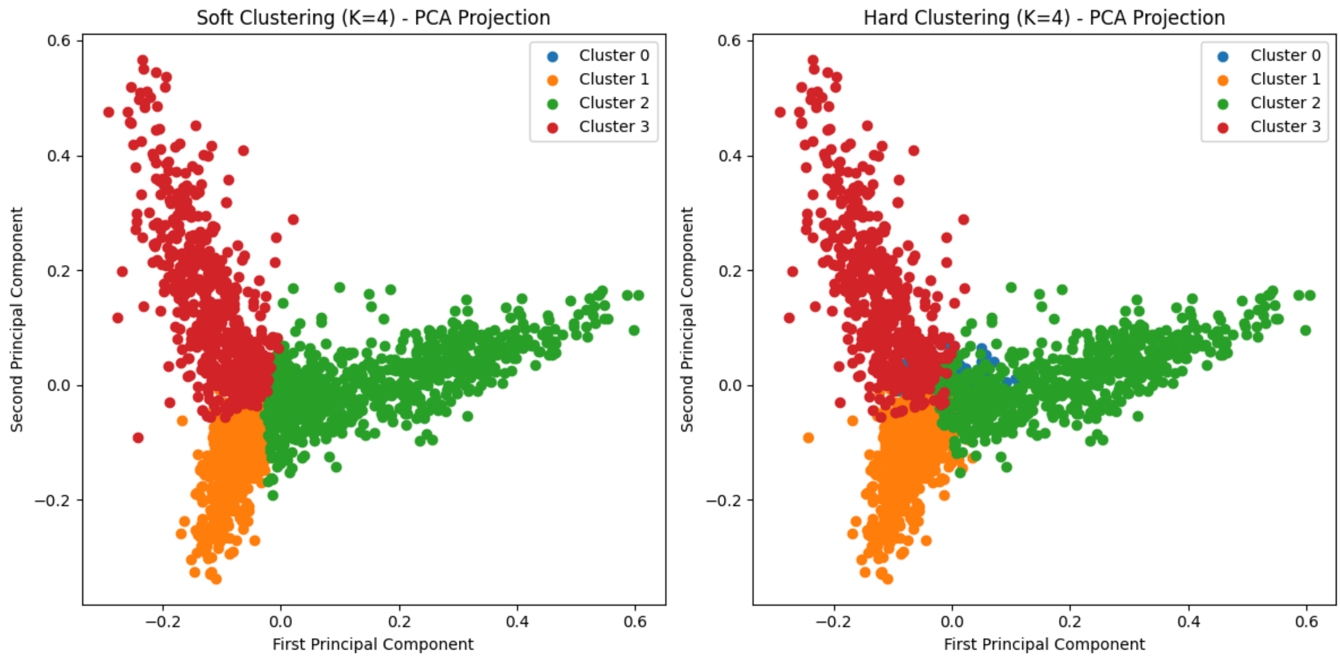
```
plt.xlabel("First Principal Component")
plt.ylabel("Second Principal Component")
plt.legend()

# Step 2: Plotting the clusters from hard clustering
plt.subplot(1, 2, 2)
plt.title("Hard Clustering (K=4) - PCA Projection")
for cluster in range(K):
    mask = hard_clusters == cluster
    plt.scatter(X_pca[mask, 0], X_pca[mask, 1], label=f"Cluster {cluster}")

plt.xlabel("First Principal Component")
plt.ylabel("Second Principal Component")
plt.legend()

plt.tight_layout()
plt.show()
```

```
Soft Clustering Assignments Sample:  [2 2 2 2 1 2 2 2 2 2]
Hard Clustering Assignments Sample:  [2 2 2 2 1 2 2 2 2 2]
```



Discussion: Differences between Soft and Hard Clustering

In the plots above, we visualize the clusters obtained from both soft and hard clustering using PCA. Here's a comparison based on the visualizations:

Soft Clustering:

Soft clustering allows documents to have probabilities of belonging to multiple clusters. As a result, the boundaries between clusters may appear more overlapping or fuzzy. In the plot, we observe that points belonging to different clusters may be closer to one another, especially near the cluster boundaries, reflecting the fact that the soft EM algorithm doesn't assign hard boundaries to clusters. This flexibility can be beneficial when there are documents that could logically belong to multiple clusters (e.g., documents that discuss multiple topics).

Hard Clustering:

Hard clustering assigns each document to exactly one cluster. As a result, the clusters appear more distinct and separated. In the plot, the points within each cluster tend to be more tightly grouped, with clearer boundaries between clusters. Hard clustering is useful when the data naturally divides into distinct groups or when there is a need for clear cluster membership. Overall, soft clustering provides more flexibility and is more suited for cases where documents can belong to multiple categories, whereas hard clustering enforces stricter groupings, which can be useful for clearly divided data.