A2_Clustering

December 21, 2021

1 Programming Component (Clustering)

[ ]:
```
"""
CSCC11 - Introduction to Machine Learning, Fall 2021, Assignment 2
M. Ataei
"""
import _pickle as pickle
import numpy as np
```

[ ]:
```
def load_pickle_dataset(file_path):
    """ This function loads a pickle file given a file path.

    Args:
    - file_path (str): The path of the pickle file

    Output:
    - (dict): A dictionary consisting the dataset content.
    """
    return pickle.load(open(file_path, "rb"))

dataset_path = f"../data/BBC_data.pkl"
BBC_data = load_pickle_dataset(dataset_path)

data = BBC_data['data']
terms = BBC_data['terms']
labels = BBC_data['labels']

print('data shape: ', data.shape)
print('terms shape: ', terms.shape)
```

```
data shape:  (2225, 9635)
terms shape:  (9635, 1)
```

1.1 Inspecting the BBC dataset

[ ]:
```
# Print on average, the number of entries that are 0 for each vector
def averageNumZeros(data):
    numZero = np.count_nonzero(data == 0)
    print('Average number of entries equal to 0 per document:', numZero / data.
 ↪shape[0])

averageNumZeros(data=data)
print('')

# Print the 10 most common and least common terms
def tenMostAndLeastCommon(data, terms):
    numTermOccurences = np.sum(data, axis=0)
    sortedInd = np.argsort(numTermOccurences)
    tenMost = sortedInd[-10:][::-1]
    tenLeast = sortedInd[:10]

    print('Ten Most Common Terms:')
    for term in tenMost:
        print(terms[term][0][0].rstrip('\n'), '\t| Num Occurences:',␣
 ↪numTermOccurences[term])

    print('')
    print('Ten Least Common Terms:')
    for term in tenLeast:
        print(terms[term][0][0].rstrip('\n'), '\t| Num Occurences:',␣
 ↪numTermOccurences[term])

tenMostAndLeastCommon(data, terms)
print('')

# print the average value for word-frequencies
def averageWordFreq(data):
    totalOccurences = np.sum(data)
    numNonZeroOccurences = np.count_nonzero(data != 0)
    print('Average word-frequency:', totalOccurences / numNonZeroOccurences)

averageWordFreq(data=data)
```

```
Average number of entries equal to 0 per document: 9506.112808988764

Ten Most Common Terms:
year    | Num Occurences: 2830
peopl   | Num Occurences: 2044
on      | Num Occurences: 1838
game    | Num Occurences: 1640
time    | Num Occurences: 1487
```

```
first   | Num Occurences: 1283
govern  | Num Occurences: 1246
go      | Num Occurences: 1222
world   | Num Occurences: 1214
get     | Num Occurences: 1196

Ten Least Common Terms:
chagrin         | Num Occurences: 3
bse     | Num Occurences: 3
angelina        | Num Occurences: 3
revolt  | Num Occurences: 3
Â£117m  | Num Occurences: 3
culprit         | Num Occurences: 3
blister         | Num Occurences: 3
horizont        | Num Occurences: 3
julio   | Num Occurences: 3
chill   | Num Occurences: 3

Average word-frequency: 1.4849044892493741
```

1.2 A2 Pg.7 Step 0: Questions 1-3

1). On average, each document vector has 9506 entries set to 0. Considering the fact that each document vector has a total 9635 entries, that means the average document has most of its entries set to 0. So in conclusion, the document term vectors are sparse.

2).

```
The 10 most common terms are:
year    | Num Occurences: 2830
peopl   | Num Occurences: 2044
on  | Num Occurences: 1838
game    | Num Occurences: 1640
time    | Num Occurences: 1487
first   | Num Occurences: 1283
govern  | Num Occurences: 1246
go  | Num Occurences: 1222
world   | Num Occurences: 1214
get     | Num Occurences: 1196

The 10 least common terms are:
chagrin     | Num Occurences: 3
bse         | Num Occurences: 3
angelina    | Num Occurences: 3
revolt      | Num Occurences: 3
Â£117m      | Num Occurences: 3
culprit     | Num Occurences: 3
blister     | Num Occurences: 3
horizont    | Num Occurences: 3
```

```
julio       | Num Occurences: 3
chill       | Num Occurences: 3
```

3). The average value for word-frequencies is roughly 1.5

1.3 K-Means Implementation

[ ]:
```python
#kmeans.py
import numpy as np

class KMeans:
    def __init__(self, init_centers):
        """ This class represents the K-means model.

        TODO: You will need to implement the methods of this class:
        - train: ndarray, int -> ndarray

        Implementation description will be provided under each method.

        For the following:
        - N: Number of samples.
        - D: Dimension of input features.
        - K: Number of centers.
             NOTE: K > 1

        Args:
        - init_centers (ndarray (shape: (K, D))): A KxD matrix consisting K␣
↪D-dimensional centers.
        """

        assert len(init_centers.shape) == 2, f"init_centers should be a KxD␣
↪matrix. Got: {init_centers.shape}"
        (self.K, self.D) = init_centers.shape
        assert self.K > 1, f"There must be at least 2 clusters. Got: {self.K}"

        # Shape: K x D
        self.centers = np.copy(init_centers)

    def train(self, train_X, max_iterations=1000):
        """ This method trains the K-means model.

        NOTE: This method updates self.centers

        The algorithm is the following:
        - Assigns data points to the closest cluster center.
        - Re-computes cluster centers based on the data points assigned to them.
        - Update the labels array to contain the index of the cluster center␣
↪each point is assigned to.
```

```
        - Loop ends when the labels do not change from one iteration to the␣
↪next.

        Args:
        - train_X (ndarray (shape: (N, D))): A NxD matrix consisting N␣
↪D-dimensional input data.
        - max_iterations (int): Maximum number of iterations.

        Output:
        - labels (ndarray (shape: (N, 1))): A N-column vector consisting N␣
↪labels of input data.
        """
        assert len(train_X.shape) == 2 and train_X.shape[1] == self.D,␣
↪f"train_X should be a NxD matrix. Got: {train_X.shape}"
        assert max_iterations > 0, f"max_iterations must be positive. Got:␣
↪{max_iterations}"
        N = train_X.shape[0]

        labels = np.empty(shape=(N, 1), dtype=np.long)
        distances = np.empty(shape=(N, self.K))
        for _ in range(max_iterations):
            old_labels = labels

            # ========================================================
            # TODO: Implement your solution within the box

            labels = np.empty(shape=(N, 1), dtype=np.long)

            # Calculate the distance between each training input and each␣
↪cluster center
            # Assign each training input to the closest cluster center
            for i in range(N):
                for j in range(self.K):
                    dif = train_X[i] - self.centers[j]
                    distances[i, j] = np.dot(dif.T, dif)

                labels[i] = np.argmin(distances[i])

            # Update each cluster center based on the inputs assigned to it
            for j in range(self.K):
                newCenter = np.zeros(self.D)
                numAssigned = 0
                for i in range(N):
                    if labels[i] == j:
                        newCenter += train_X[i]
                        numAssigned += 1
```

```
                self.centers[j] = (newCenter / numAssigned)


            # ================================================

            # Check convergence
            if np.allclose(old_labels, labels):
                break

        return labels
```

1.4 A2 Pg.7 Step 1: Questions 1-3

1). I cannot really figure out the topics the clusters represent. Every cluster has the same few words repeated many times such as: time, on, govern, tax, player. etc. A rough guess would be sports and politics.

2). Some factors that make clustering difficult are:
- Determining the optimal hyper-parameters for example, it is difficult to determine how many clusters a dataset should have.
- There is no guarantee that the algorithm will get close to the global optimum. This algorithm could very likely get trapped in a poor local minima.
- Initialization can heavily impact the results of clustering and poor initialization can lead to poor results - Problems with the input data such as outliers - With this BBC dataset, the inputs have a lot of features which means we have to deal with very high dimensional data - The input data as mentioned previously is also sparse

3). If we have a lucky guess at the cluster centers then yes, we should expect better results since this algorithm is sensitive to initialization. Each step of the optimization will lower the objective function so if we start near the global optimum, we will converge near the global optimum.

1.5 A2 Pg.8 Step 2: Questions 1-3

1). The error of the objective function is much lower (around 600000 in step 1 down to 250).

2). Compared to step 1, the clusters have a much bigger variety of words. From observing each cluster, I would roughly say that some topics are: - Soccer (because of words like: match, win, defeat, final, names of soccer teams, midfield, etc) - Politics (because of words like: elect, nation, campaign, govern, name of many countries etc.) - Film Industry (because of words like: film, critic, award, director, hollywood, cinema, etc)

3). I would consider this result better than Step 1 since there was more variety of words in each cluster and I could even slightly notice a difference between some clusters. As opposed to in Step 1 where seemingly every cluster repeated the same few words many, many times.

1.6 A2 Pg.8 Step 3: Questions 1-3

1). From my observation the topics are: - Technology (because of words like: media, microsoft, apple, software, etc.) - Politics specifically in Europe (because of words like: govern, chancellor, tax, elect, names of many European countries, etc.) - Sports, more specifically European soccer

(because of words like: team, win, defeat, championship, captain, names of European soccer clubs, etc.) - Economy (because of words like: market, growth, finance, company, account, economist, etc.) - Film industry (because of words like: film, director, hollywood, award, actor, grammy, names of famous actors, etc.)

2). After pre-processing the documents by performing random-walk diffusion, the documents in a cluster have a stronger sense of similarity with eachother. This results in the K-Means algorithm producing more meaningful clusters that have very similar data points. This is evident as the objective error has been reduced to roughly 3.5

3). In order to produce meaningful results/clusters, high-dimensional sparse data should be pre-processed. Otherwise, the results will very likely turn out very poor and meaningless.

1.7 GMM Implementation

[ ]:
```
#gmm.py
import numpy as np
from numpy.lib.function_base import cov
from scipy.stats import multivariate_normal
from functools import partial

class GMM:
    def __init__(self, init_centers):
        """ This class represents the GMM model.

        TODO: You will need to implement the methods of this class:
        - _e_step: ndarray, ndarray -> ndarray
        - _m_step: ndarray, ndarray -> None

        Implementation description will be provided under each method.

        For the following:
        - N: Number of samples.
        - D: Dimension of input features.
        - K: Number of Gaussians.
             NOTE: K > 1

        Args:
        - init_centers (ndarray (shape: (K, D))): A KxD matrix consisting K␣
↪D-dimensional centers, each for a Gaussian.
        """
        assert len(
            init_centers.shape) == 2, f"init_centers should be a KxD matrix.␣
↪Got: {init_centers.shape}"
        (self.K, self.D) = init_centers.shape
        assert self.K > 1, f"There must be at least 2 clusters. Got: {self.K}"

        # Shape: K x D
```

```
        self.centers = np.copy(init_centers)

        # Shape: K x D x D
        self.covariances = np.tile(np.eye(self.D), reps=(self.K, 1, 1))

        # Shape: K x 1
        self.mixture_proportions = np.ones(shape=(self.K, 1)) / self.K

    def _e_step(self, train_X):
        """ This method performs the E-step of the EM algorithm.

        Args:
        - train_X (ndarray (shape: (N, D))): A NxD matrix consisting N␣
↪D-dimensional input data.

        Output:
        - probability_matrix_updated (ndarray (shape: (N, K))): A NxK matrix␣
↪consisting N conditional probabilities of p(z_k|x_i) (i.e. the␣
↪responsibilities).
        """
        (N, D) = train_X.shape
        probability_matrix = np.empty(shape=(N, self.K))

        # ===================================================
        # TODO: Implement your solution within the box

        # Compute the responsibilities
        for i in range(N):
            for j in range(self.K):
                numer = self.mixture_proportions[j] * multivariate_normal.
↪pdf(train_X[i],
                                                                        ␣
↪mean=self.centers[j], cov=self.covariances[j])
                denom = 0
                for k in range(self.K):
                    denom += self.mixture_proportions[k] * multivariate_normal.
↪pdf(train_X[i],
                                                                        ␣
↪   mean=self.centers[k], cov=self.covariances[k])
                probability_matrix[i, j] = numer / denom

        # ===================================================

        assert probability_matrix.shape == (
            train_X.shape[0], self.K), f"probability_matrix shape mismatch.␣
↪Expected: {(train_X.shape[0], self.K)}. Got: {probability_matrix.shape}"
```

```
        return probability_matrix

    def _m_step(self, train_X, probability_matrix):
        """ This method performs the M-step of the EM algorithm.

        NOTE: This method updates self.centers, self.covariances, and self.
↪mixture_proportions

        Args:
        - train_X (ndarray (shape: (N, D))): A NxD matrix consisting N␣
↪D-dimensional input data.
        - probability_matrix (ndarray (shape: (N, K))): A NxK matrix consisting␣
↪N conditional probabilities of p(z_k|x_i) (i.e. the responsibilities).

        Output:
        - centers (ndarray (shape: (K, D))): A KxD matrix consisting K␣
↪D-dimensional means for each Gaussian component.
        - covariances (ndarray (shape: (K, D, D))): A KxDxD tensor consisting K␣
↪DxD covariance matrix for each Gaussian component.
        - mixture_proportions (ndarray (shape: (K, 1))): A K-column vector␣
↪consistent the mixture proportion for each Gaussian component.
        """
        (N, D) = train_X.shape

        centers = np.empty(shape=(self.K, self.D))
        covariances = np.empty(shape=(self.K, self.D, self.D))
        mixture_proportions = np.empty(shape=(self.K, 1))
        # ========================================================
        # TODO: Implement your solution within the box

        for j in range(self.K):
            sumProb = np.sum(probability_matrix[:, j])

            # compute the mixture probability
            mixture_proportions[j] = sumProb / N

            # compute the mean for the Guassian distribution
            centers[j] = np.sum(
                train_X * probability_matrix[:, j].reshape(len(train_X), 1),␣
↪axis=0) / sumProb

            # compute the covariance matrix
            cov = np.zeros(shape=(D, D))
            for i in range(N):
                dif = np.reshape(train_X[i] - centers[j], (D, 1))
```

```
                cov += probability_matrix[i, j] * np.dot(dif, dif.T)
            covariances[j] = cov / sumProb

        #
        # ================================================

        assert centers.shape == (
            self.K, self.D), f"centers shape mismatch. Expected: {(self.K, self.
↪D)}. Got: {centers.shape}"
        assert covariances.shape == (
            self.K, self.D, self.D), f"covariances shape mismatch. Expected:␣
↪{(self.K, self.D, self.D)}. Got: {covariances.shape}"
        assert mixture_proportions.shape == (
            self.K, 1), f"mixture_proportions shape mismatch. Expected: {(self.
↪K, 1)}. Got: {mixture_proportions.shape}"

        return centers, covariances, mixture_proportions

    def train(self, train_X, max_iterations=1000):
        """ This method trains the GMM model using EM algorithm.

        NOTE: This method updates self.centers, self.covariances, and self.
↪mixture_proportions

        Args:
        - train_X (ndarray (shape: (N, D))): A NxD matrix consisting N␣
↪D-dimensional input data.
        - max_iterations (int): Maximum number of iterations.

        Output:
        - labels (ndarray (shape: (N, 1))): A N-column vector consisting N␣
↪labels of input data.
        """
        assert len(
            train_X.shape) == 2 and train_X.shape[1] == self.D, f"train_X␣
↪should be a NxD matrix. Got: {train_X.shape}"
        assert max_iterations > 0, f"max_iterations must be positive. Got:␣
↪{max_iterations}"
        N = train_X.shape[0]

        e_step = partial(self._e_step, train_X=train_X)
        m_step = partial(self._m_step, train_X=train_X)

        labels = np.empty(shape=(N, 1), dtype=np.long)
        for _ in range(max_iterations):
            old_labels = labels
```

```
            # E-Step
            probability_matrix = e_step()

            # Reassign labels
            labels = np.argmax(probability_matrix, axis=1).reshape((N, 1))

            # Check convergence
            if np.allclose(old_labels, labels):
                break

            # M-Step
            self.centers, self.covariances, self.mixture_proportions = m_step(
                probability_matrix=probability_matrix)

        return labels
```

1.8 GMM Questions

For some of the datasets, GMM returns overlapping clusters whereas K-Means always returns strictly partitioned (non-overlapping) clusters. Another difference is that all the K-Means clusters have more or less a circular shape where as GMM clusters can vary with circular and elliptical shapes.

The reason for this is because K-Means is a distance based model and it focuses on minimizing the Euclidean distance between the cluster centers and their respectively assigned data points. Hence, it is clear why the K-Means clusters do not overlap because if there was overlap, then the distance between between cluster centers and their assigned data points are clearly not minimized.
However, in GMM we are focusing on figuring out the probability of a data point belonging to a cluster. We assume the data points to be sampled from K Guassian distributions and we are trying to estimate the parameters of these Guassian components. GMM also allows us to express some prior beliefs of the fraction of data assigned to each Guassian distribution. Thus GMM clusters can adapt to return overlapping and elliptical shaped clusters. The extra adaptability is why GMM seems to do better on some of these datasets.

Regarding the task of document clustering, because the BBC dataset has such high dimensional data, we could perform pre-processing on the data in order to run GMM for this task. For example, we could perform dimensionality reduction on the data by avoiding unecessary dimensions since as we saw previously, the BBC dataset has sparse data.