test

December 9, 2020

1 University of Exeter

1.1 College of Engineering, Mathematics and Physical Sciences

1.1.1 ECM3420 - Learning From Data

Coursework 2 - Clutering

1.1.2 Enter your candidate number here:

1.2 Task 1

[2]:
```
import numpy as np
from math import sqrt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

class Centroid:
    """Data structure for holding a Centroid.

    Initialise with a unique label and a numpy array as a value.

    Attributes:
        label: The label assigned to the centroid.
        initial_value: Numpy array representing the centroid of all data points␣
↪associated with this centroid
    """


    def __init__(self, label, initial_value):
        self.value = initial_value
        self.label = label


    def __str__(self):
        return f"Centroid({self.label} of value {self.value})"


    def calculate_value(self, data_points):
```

```python
        """Calculate the value of a centroid based on all datapoints.

        Loops through all data points to find ones associated with current␣
↪centroid.
        Finds and sets the average centroid based on assigned data points.

        Args:
            data_points: List of all data points being analysed

        Returns:
            Numpy array representing the new value of a centroid
        """
        # Initialise new value with same dimensions as current
        new_value = np.zeros_like(self.value)

        data_point_count = 0

        # Sum all datapoints
        for data_point in data_points:

            if data_point.centroid is self:
                new_value += data_point.value
                data_point_count += 1

        # Average new value by number of datapoints
        new_value /= data_point_count

        self.value = new_value

        return new_value


class DataPoint:
    """Data structure for a datapoint with a centroid association.

    Attributes:
        value: Numpy array holding the datapoint's value
        centroid: Centroid instance representing the datapoint's currently␣
↪associated centroid
    """

    def __init__(self, value):
        self.value = value
        self.centroid = None


    def __str__(self):
```

```python
        return f"DataPoint({self.value} of centroid {self.centroid})"


    def assign_to_nearest_centroid(self, centroids):
        """Assigns the datapoint to its nearest centroid from a list of given␣
 ↪centroids.

        Args:
            centroids: List of centroids

        Returns:
            (old_centroid, new_centroid)
                old_centroid: The previous centroid associated with the value.␣
 ↪Note that for the initial iteration, this will be None
                new_centroid: The newly assigned centroid. Note that as␣
 ↪centroids converge, this could be the same as old_centroid
        """
        flag_first = True

        min_distance = 0
        min_centroid = None

        old_centroid = self.centroid

        for centroid in centroids:
            distance = euclidian_distance(self.value, centroid.value)
            # Initialise minimum value
            if flag_first:
                flag_first = False
                min_distance = distance
                min_centroid = centroid
                pass

            # Update minimum distance and centroid
            if distance < min_distance:
                min_distance = distance
                min_centroid = centroid

        self.centroid = min_centroid

        return old_centroid, self.centroid

# ------------------------------------ #
#                UTILS                 #
# ------------------------------------ #
def euclidian_distance(point1, point2):
    """Calculate the euclidian distance between two points.
```

```
    Args:
        point1: Numpy array representing the first point
        point2: Numpy array representing the second point

    Returns:
        Float value of the euclidian distance between point1 and point2
    """
    return sqrt(sum(np.square(point2 - point1)))


def initialise(x, k):
    """Initialise DataPoints and Centroids for K-Means-Clustering

    Create a list of DataPoint objects based on the points in x.
    Initialise k Clusters, each with a random point as a starting value.

    Args:
        x: A list of datapoints
        k: The number of centroids to initialise

    Returns:
        centroids: A list of initialised Centroids
        data_points: A list of initialised DataPoints
    """
    # Create a DataPoint instance for each datapoint in x then form a list of␣
 ↪these
    data_points = [DataPoint(dp) for dp in x]

    # Randomly choose k points to initialise as clusters
    random_points_indices = np.random.choice(len(data_points), size=k)

    # Fetch the DataPoint instances that correspond to the random indices
    initial_centroid_points = [data_points[index] for index in␣
 ↪random_points_indices]

    # Create a list of centroids initialised with the DataPoint values above.␣
 ↪Assign a unique label to each
    centroids = [Centroid(label=i, initial_value=c.value) for i, c in␣
 ↪enumerate(initial_centroid_points)]

    return centroids, data_points


def incremental_kmeans(x, k, max_itr=100, random_state=None):
    """Run incremental K-Means Clustering on a dataset
```

```
    Runs incremental K-Means Clustering on a dataset. Returns a list of␣
↪datapoint cluster labels
    and the number of iterations taken.

    Args:
        x: A list consisting of numpy arrays as datapoints
        k: The number of clusters to find
        max_itr: The maximum number of iterations to run before ending␣
↪iteration. Default=100
        random_state: Integer value used to seed the randomness for␣
↪deterministic behaviour. Default=None

    Returns:
        cluster_labels: A list consisting of each datapoint's cluster␣
↪association
        iter_count: The number of iterations the K-Means Clustering ran.
    """
    # Seed the randomness if a value is provided.
    if random_state is not None:
        np.random.seed(random_state)

    # Initialise centroids and data points from x and k
    centroids, data_points = initialise(x, k)

    # Initial iteration, assigning points to their initial centroids
    for point in data_points:
        point.assign_to_nearest_centroid(centroids)


    flag_stop = False
    iter_count = 0

    while not flag_stop:
        iter_count += 1
        flag_stop = True
        for point in data_points:
            old_centroid, new_centroid = point.
↪assign_to_nearest_centroid(centroids)

            if old_centroid is not None:
                old_centroid.calculate_value(data_points)

            new_centroid.calculate_value(data_points)

            # A centroid has changed so we should continue iterating
            if old_centroid is not new_centroid:
                flag_stop = False
```

```
        if iter_count == max_itr:
            print("Ended having reached maximum iterations")
            flag_stop = True

    cluster_labels = [data_point.centroid.label for data_point in data_points]

    return cluster_labels, iter_count
```

1.3 Task 2

[3]:
```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import time

dataset = load_iris().data

inc_times = {2: [], 3: [], 4: [], 5: []}
inc_iters = {2: [], 3: [], 4: [], 5: []}

std_times = {2: [], 3: [], 4: [], 5: []}
std_iters = {2: [], 3: [], 4: [], 5: []}


for m in range(5):
    for k in range(2, 6):
        time_start = time.time()
        iters = incremental_kmeans(dataset, k)[1]
        time_end = time.time()

        inc_times[k].append(time_end-time_start)
        inc_iters[k].append(iters)

        time_start = time.time()
        kmeans = KMeans(n_clusters=k).fit(dataset)
        time_end = time.time()

        std_times[k].append(time_end-time_start)
        std_iters[k].append(kmeans.n_iter_)
```

[4]:
```
from tabulate import tabulate
from IPython.display import HTML, display

def avg_results(results):
    averages = [(sum(row)/len(row)) for row in results.values()]
```

```
    return averages

def format_table(time1, iter1, time2, iter2):
    row1 = ["", "", "K=2", "K=3", "K=4", "K=5"]
    row2 = ["Inc K-Means", "Average Time"] + avg_results(time1)
    row3 = ["", "Average Iterations"] + avg_results(iter1)

    row4 = ["Std K-Means", "Average Time"] + avg_results(time2)
    row5 = ["", "Average Iterations"] + avg_results(iter2)
    return [row1, row2, row3, row4, row5]

# avg_results(inc_times)
display(HTML(tabulate(format_table(inc_times, inc_iters, std_times, std_iters),␣
↪tablefmt="html")))
```

```
<IPython.core.display.HTML object>
```

[25]:
```
import seaborn as sns
seaborn.boxplot(x=inc_times.keys(), y=inc_times.values())
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-193dcee84904> in <module>
      1 import seaborn as sns
----> 2 seaborn.boxplot(x=inc_times.keys(), y=inc_times.values())

NameError: name 'seaborn' is not defined
```

1.4 Task 3

[ ]:
```
# your code here
```

[ ]:
```

```

[ ]:
```

```

1.5 Task 4

[ ]:
```
# your code here
```

[ ]:
```

```

[ ]:
```

```