CS 512 Project
Implementation of STREAM LSEARCH
A Stream Clustering Algorithm:

Ishani Ghose
Kaushik Vakadkar
Chaitanya Vallabhaneni

15 December 2020

1 Introduction

The project involves the implementation of the Stream Lsearch and evaluation of its performance on sensor data against a hierarchical clustering algorithm run on the entire data set.

1.1 Data

Weather sensor data downloaded from this source has been used. The input is in the form of a log file with each data point having the following attributes: data, time, temperature(R), humidity(R), light(R), voltage(R), epoch(I) and moteid(I), where R = Real number, I = Integer.
The size of the data set is 150 MB. While the data is statically stored, a stream of data can be generated from it to analyze the performance of the implemented algorithms.
The clusters generated can be used to answer the following questions:

1. What are the weather conditions that are observed?
2. Are the clusters generated representative of the entire data?
3. Is there any correlation between the different attributes? - This can be answered from the representatives of each cluster.

2 Pseudocode

1. **STREAM LSEARCH**
   The implementation of the clustering subroutine (LSEARCH) used for a given set of points is available in C++. We'll implement this in Python. This algorithm takes an upper and lower bound on the number of clusters that the algorithm can create and then performs a local search on it. The paper explaining this algorithm can be found here.

   Each data point is converted to the following list:

   ([coordinates of the data],
   centre it is mapped to - initialized to 0,

cost - distance from the centre
Weight - initialized to 1,
original index in the data - used to maintain the true values collected by the sensor)

The *kmedians* algorithm used in LSEARCH is shown below:

Algorithm 1 kmedians(kmax, kmin)

$z = 0$
$z_{max} = \sum_{x \in N}(\text{distance}(x, x_0))$ where x_0 is chosen as the first point in the chunk
$z_{min} = 0$
$z = (z_{max} + z_{min})/2$
(I, a) = Estimate an initial solution by calling function speedy2
if number of centres identified $< k_{max}$ **then**
  return solution
end if
Shuffle data
(I, a) = Estimate a solution again by calling function speedy
if number of centres identified $< k_{min}$ **then**
  (I, a) = Keep calling speedy to estimate a solution until number of centres is $\geq k_{min}$
end if
if k > number of points points **then**
  Assign each point to itself as the centre and return
end if
Shuffle data
Randomly pick θ(k_min∗log(k_min)) points as median
while k is outside the range(k_{min}, k_{max}) and $z_{max} - z_{min} > 0.0001$ **do**
  Let (F, g) be the current solution
  Run $FL(N, d, , (F, g))$ to obtain a new solution (F_0, g_0)
  if $|F_0|$ is "about" k **then**
    $(F_0, g_0) \leftarrow FL(N, d, 0, (F_0, g_0))$
    if $|F_0| > k$ **then**
      $z_{min} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2$
    else if $|F_0| > k$ **then**
      $z_{max} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2$
    end if
  end if
end while
To simulate a continuous space call *contcenters*, move each cluster center to the center-of-mass for its cluster
return (F_0, g_0)

If we are given a set N of n objects in some space with distance function d, then k-Means essentially minimizes the following expression for each chunk of data:

$$\sum_{i=1}^{k} \sum_{x \in N_i} d(x, c_i)$$

where, $N_i = \{x \in N | \forall j : d(c_j, x) \geq d(c_i, x)\}$. The value of k is determined by LSEARCH in the real space between k_{min} and k_{max} passed to *kmedians*. While performing LSEARCH, *kmedians*

performs binary search on the value of z to look for the optimal cost of opening a facility centre. The implementation of LSERACH in Python is similar to the implementation of LSEARCH in C++ and can additionally be called on stream data. If the number of centres exceeds a pre-specified limit, *kmedian* is called on the list of centres to reduce the amount of memory required to $O(k\_max)$ again.

Algorithm 2 speedy(z, k)

```
for each point ∈ chunk do
  if distance of the point from the closest centre is large then
    open a new facility centre at the point
  else
    assign the point to the closest centre and the increment the total cost by z
  end if
end for
```

Algorithm 3 speedy2(z, k, k_max)

```
for each point ∈ chunk do
  if distance of the point from the closest centre is large then
    open a new facility centre at the point
  else
    assign the point to the closest centre and the increment the total cost by z
  end if
  if number of facility centres opened till now exceeds k_max then
    return cost, k
  end if
end for
```

In the *speedy* and *speedy*2 algorithms, the distance of the point from the closest centre is determined probabilistically.
The probability of opening a centre while executing *speedy*2 is much higher than while executing speedy as the cost of opening a facility centre while running *speedy*2 is only 0.001 and $(z\_max + z\_min)/2$ while running *speedy*. Speedy2 gives a quick solution when the data points are extremely similar.

The pseudocode for the function $selectFeasible$ is given below:

Algorithm 4 selectFeasible(k_min))

```
if number of points in chunk < kmax then
  make every point a feasible facility centre
end if
Select 3*k_min*log(k_min) points randomly as feasible facility centres
return num of feasible centres, list of feasible centres
```

The pseudocode for the function FL is given below:

Algorithm 5 FL(feasible, numfeasible, z, k, cost, iter, e)

while improvement in cost $\leq$ -0.1 and number of tries to improve the cost is within a specified bound **do**
  Shuffle the feasible centres
  try: iter=3*k_max*$log(k\_max)$ times
    Choose a facility centre = x randomly and check for a negative change in the cost on opening the centre at x by calling the function *gain*
    Estimate improvement in cost from the change
end while

Description of function $gain(x, z, k)$ to estimate the gain in overall cost by opening a centre at x which is passed to the function:

(a) Check if the point x is already a centre.
- If it isn't a centre, cost of opening a facility is initialized to z
- Else the additional cost of opening is initialized to 0

(b) To evaluate how much each median would save by shutting down it's centre and moving all it's points to x, use an array [lower] and initialize it to z for all existing centres.

(c) For each point in the sample, evaluate the distance to its current centres and distance to x.
- If the point is closer to x, decrement the cost of opening the centre at x by the gain made which is the difference in the distance values. Also switch the membership of the point to x.
- If the point is closer to the current centre, the savings of shutting down it's current centre is reduced by the difference in the distance values.

(d) The number of centres that can be closed is computed from the centres that have lower[index] as positive i.e. there are positive savings in closing the centre. The the total cost of opening a centre at x is reduced by the savings from shutting down the centre and the number of centres to close is incremented by 1.

(e) Now if the cost of opening a centre at x is negative, we move all points which have been marked in $switch\_membership$ to x and update the cost of these points to the distance from x and increment the number of centres by 1, decrease the number of centres by number of centres to close.

(f) If the cost of opening the centre is 0 or positive, we do nothing and return 0 as the cost, else (-1 $\times$ the cost of opening the centre at x) is returned.

Description of function $contcenters()$:

(a) For each point in the data, update the coordinates or dimensions of the centres it is mapped to, moving towards the "center-of-mass" of the cluster.

3 Analysis of STREAM LSEARCH

1. **Time Complexity**
The running time of *kmedians* is $O(nm + nk\log k)$ where m is the number of facility centres opened during *speedy*2, n is the size of the chunk and k is the maximum number of medians. Across chunks of 1000 data points, the number of centres opened by *speedy*2 was 12 on an average.

2. **Space Complexity**
   If the number of centres being maintained across all the chunks of data exceeds a pre-specified limit, $max\_centres$, $kmedian$ is called on the list of centres to reduce the amount of memory required to $O(k_{max})$ again. However, the amount of space required to store the centres in addition to the space required to run the algorithm is $O(max\_centres)$.
   The space required to run the algorithm is $O(chunk\_size)$.

3. **Data**

   The LSEARCH algorithm is run over sensor data by simulating a stream over 50,000 data points by taking chunks of 1000 data points running $kmedians$ over it to search for k centres. The last 4 attributes of the raw data: $Temperature, Humidity, Light$ and $Voltage$ were used to form the clusters. As the different attributes lied in extremely different range, each value was normalized by the following formula:

$$norm\_val = (val - min)/(max - min)$$

   The ranges of the 4 features are:

   (a) Temperature: (-5,100)(degree celsius)
   (b) Humidity: (0,100)
   (c) Light: (0,100000)(flux)
   (d) Voltage: (2,3)

   The centres are the representatives of the data and can be used to estimate the following statistics of the data:

   - Mean and standard deviation of each attribute
   - If enough representatives have been collected, they can be clustered to find the sum of (SSQ) measure on a stream of data points

   Stream Lsearch was performed on the data with kmin = 3 and kmax = 10. These number were determined from the SSQ of the clusters from the first chunk of data.
   Standard deviations of the attributes from 20000 data points using only centres:
   [1.258e+01, 7.179e+00, 1.776e+02, 9.189e-02]

   True std_dev using all the data:
   [3.298e+00, 4.937e+00, 1.771e+02, 5.172e-02]

   Means of the attributes from representatives of the 20000 data points:
   [23.443, 35.917, 165.642, 2.644]

   True mean from all the data of 20000 data points:
   [21.925, 36.895, 156.644, 2.654]

   SSQ on stream of 20000 data points broken into chunks of 1000 data points averaged over 10 runs:

   - k=62 centres identified from 20000 data points using Stream LSEARCH
     Total SSQ = 1.97

- Kmeans run over the static dataset of 20000
  k=62, 20000, ssq = 1.48

4 Clustering algorithm: CLIQUE

CLIQUE is a density-based and grid-based subspace clustering algorithm. The paper explaining this algorithm can be found. An attempt was made in trying to apply the CLIQUE algorithm to the sensor dataset, but after several attempts, we concluded that CLIQUE does not work well with data streams. It is better suited for static data clustering. The quality of the results crucially depends on the appropriate choice of the number and width of the partitions and grid cells. This is one of the limitations of this algorithm.here.

- Grid-based: It discretizes the data space through a grid and estimates the density by counting the number of points in a grid cell
- Density-based: A cluster is a maximal set of connected dense units in a subspace

Algorithm 6 clique(data, xi, tau)

```
data_size = get_len(data)
num_features = get_width(data)
1d = []
for feature in get_features(data) do
  for unit in feature do
    if get_data_points(unit) > data_size * tau then
      1d.add(unit)
    end if
  end for
  for (k in range(num_features)) do
    candidates = self_join(elements=D(k-1), condition=(share (k-2) dimensions))
    ddu = candidates[having all (k-1) projections in D(k-1)]
  end for
  for F in feature set {F_1, F_2, .., F_k} do
    G = identity_matrix(n = num_dense_units(F))
    for i, j running [0-n] do
      if i = j then
        G[i, j] = 1
      else if is_connected(get_unit(F, i), get_unit(F, j) then
        G[i, j] = 1
      else
        G[i, j] = 0; clusters = get_connected_comp(G)
      end if
    end for
  end for
end for
```