Haskell Interim Test

Imperial College London

Department of Computing

k-means Clustering

Tuesday 09 November 2021
14:00 to 16:00
TWO HOURS
(including 10 minutes planning time)

- The maximum total is **20 marks**. There is one bonus question; any marks awarded for this will be added to your total.
- Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.
- **Important:** TEN MARKS will be deducted from solutions that do not compile in the test environment, which will be the same as the lab machines. Comment out any code that does not compile before you submit.
- **WARNING:** The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.
- The extracted files should be **directly inside** your gitlab repository. You can extract the archives by adjusting the following command: `7z x filename.7z`
- Push the final version of your code to `gitlab` before the deadline, and then go to `LabTS`, find the final/correct commit and submit it to `CATe`.

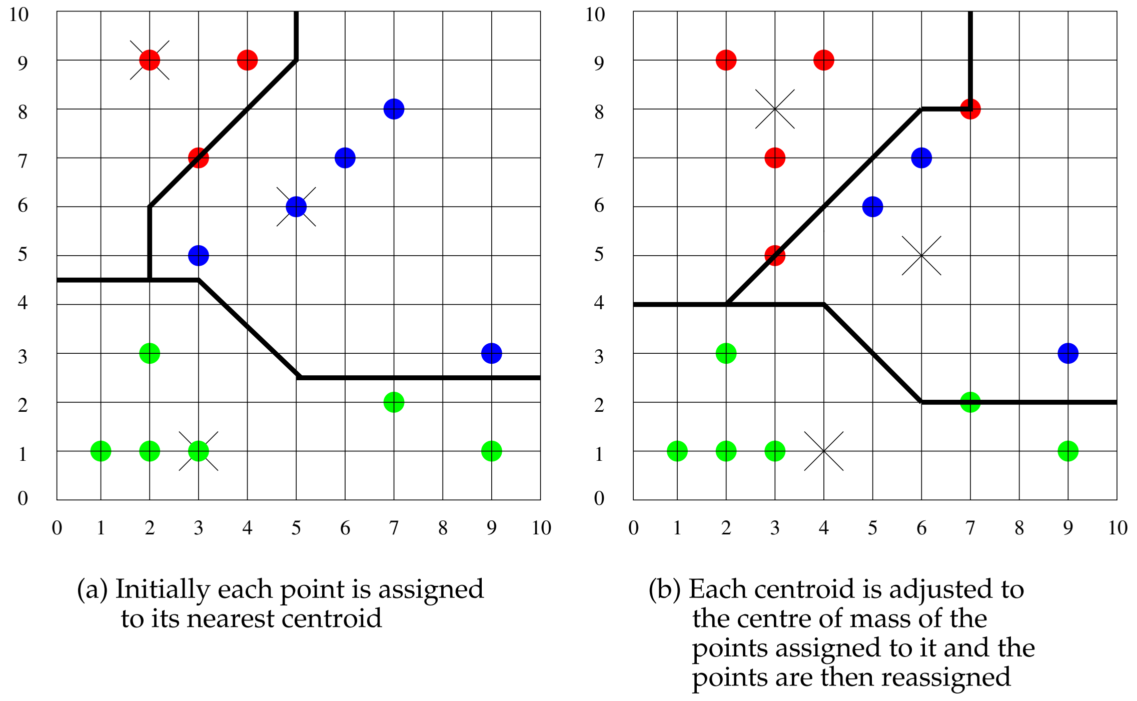

Figure 1: One iteration of the clustering algorithm

The problem

A ubiquitous problem in data analysis is that of *clustering* data into groups so that the data items in each group are in some sense "similar". For example, a photograph represented using RGB (Red, Green, Blue) encoding has a colour "palette" of $256 \times 256 \times 256 = 16777216$ different RGB settings, assuming 8-bits are used to record the intensity of each of the three colours within a given pixel. The number of palette colours used to represent the photograph can be reduced to some much smaller number k by clustering the observed RGB values into k groups, each representing "similar" RGB values. This enables the photograph to be compressed at the expense of some small loss in image fidelity. There are many other applications, e.g. in machine learning.

k-means clustering

A simple algorithm for clustering multidimensional data[1] is *k-means clustering*. This algorithm groups the data into exactly k clusters and in each iteration of the algorithm each data item is assigned to the cluster whose "centroid" is nearest to it. In general, each data item and each cluster's centroid is an n-dimensional vector (a Haskell tuple), for some $n > 0$; the centroid is the average location of the data items in n-dimensional space.

The algorithm begins by picking k initial centroids arbitrarily and it terminates when there is no change to the assignment and to the centroids, i.e. when the next iteration leads to exactly the same result as the previous one.

[1]The data in the above example is three-dimensional because each pixel value corresponds to separate R, G and B intensities

Example

Figure 1 shows an example of one iteration of the algorithm for a two-dimensional problem in which the data items are points in the 2D (x, y) plane. The initial configuration of points is shown in Figure 1(a) along with three initial centroids, marked with a '$\times$'; these centroids happen to coincide with three of the points, but in general they could be anywhere on the grid. All points and centroids are located at grid points, i.e. coordinates are integer pairs.

At the start of each step of the algorithm, each point is associated with (i.e. assigned to) its nearest centroid and this is shown in Figure 1(a). Notice how this partitions the set of points into three clusters, one for each centroid. The thick lines show the boundaries of these three partitions, but they are for illustration only: the boundaries do not feature explicitly in the algorithm. Each iteration of the algorithm involves adjusting the locations of the centroids based on the current assignment and then re-assigning the points to these new centroids. Figure 1(b) shows the result of one step of the algorithm, starting from Figure 1(a). Notice how the centroids have been relocated to the centroid of the points initially assigned to it and how the points have been reassigned. Two of them have moved from cluster 3 (blue) to cluster 2 (red). The process repeats until there is no change to the cluster assignment and to the centroids.

Centroid calculation

The centroid, c, of a set of points $p_1, p_2, ..., p_N$ $(N > 0)$, where p_i is a point (vector) in an $n-$dimensional space, is simply the average of the $p_i, 1 \leq i \leq N$:

$$c = \frac{p_1 + p_2 + ... + p_N}{N}$$

The summation here (+) is defined over $n-$dimensional *vectors* rather than numbers in a fairly obvious way. For example, in two dimensions:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

1 The problem

In this exercise you're going to implement the k-means clustering algorithm in two dimensions. You will be given N data items, each of the form (x, y) where x and y are *integers*, i.e. the data items correspond to points on a 2D grid[2]. The distance between two points is defined to be the smallest number of (integer) grid steps required to get from one to the other. This is sometimes called the "Manhattan distance" or the "1-norm distance":

$$\text{distance}((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

Each centroid is also defined to lie on a grid point, so the centroid of N points $(x_1, y_1), ..., (x_N, y_N)$ is defined to be:

$$\left(\left\lfloor \frac{\sum_{i=1}^{N} x_i}{N} \right\rfloor, \left\lfloor \frac{\sum_{i=1}^{N} y_i}{N} \right\rfloor \right)$$

where $\lfloor a/b \rfloor$ denotes the integer division (`div` in Haskell) of a by b. Notice that there is no floating-point calculation involved at all in this problem.

The following Haskell type synonyms are defined in the template provided:

[2]Normally, k-means clustering operates in continuous space, but we work with integers here in order to simplify both the problem and testing.

```
type Point = (Int, Int)

type Centroid = Point

type ClusterIndex = Int

type Assignment = (Point, ClusterIndex)

type Clustering = ([Assignment], [Centroid])

type Colouring = ([(Point, Colour)], [Centroid])

data Colour = Black | Red | Green | Blue | Mauve
            deriving (Show, Enum, Eq)
```

The data items to be clustered will be in the form of a `[Point]` and the initial cluster centroids will be taken to be the first k data items in that list; a precondition is therefore that the list contains at least k data items. k is a parameter of the clustering function and the centroids are numbered from 1 to k. The constant `points` in the template file corresponds to the points shown in Figure 1, so centroids `1`, `2` and `3` are initially assumed to be located at points `(3,1)`, `(2,9)` and `(5,6)` respectively, as shown.

```
points :: [Point]
points = [(3,1),(2,9),(5,6),(7,8),(2,3),(6,7),(3,5),(4,9),(9,3),
          (1,1), (2,1),(9,1),(7,2),(3,7)]
```

2 What to do

The template contains the type signature and a default definition for each function you are required to write. Replace the default definition (which simply returns `undefined` in each case) with the required function as you get to it.

Several lists of points, together with the output from the clustering algorithm, have been defined in the template for testing purposes, including `points` above. For example, the final result produced by the algorithm, after iterating to convergence, is defined in `result`:

```
-- Result of kmeans points 3
result :: Clustering
result
  = ([((3,1),1),((2,9),2),((5,6),3),((7,8),3),((2,3),1),((6,7),3),
      ((3,5),2),((4,9),2),((9,3),3),((1,1),1),((2,1),1),((9,1),1),
      ((7,2),1),((3,7),2)],
     [(4,1),(3,7),(6,6)])
```

The final locations of the centroids are `[(4,1),(3,7),(6,6)]`, as shown. Note that `result` does not correspond to Figure 1(b) as the figure only shows one step of the algorithm.

1. Define a polymorphic function `lookupAll :: Eq a => a -> [(b, a)] -> [b]` that, given a search key and a list of (value, key) pairs returns all values whose key matches the search key. For example,

```
*KMeans> lookupAll 3 [(4,3),(2,1),(2,3)]
[4,2]
*KMeans> lookupAll 1 fig1aAssignment
[(3,1),(2,3),(1,1),(2,1),(9,1),(7,2)]
```

[2 Marks]

2. Define a function `distance :: Point -> Point -> Int` that will return the distance between two points (an `Int`), where the distance is defined as above. For example,

   ```
   *KMeans> distance (2,5) (1,6)
   2
   ```

   [1 Mark]

3. Define a function `sumPoints :: Point -> Point -> Point` that will sum two points by summing their respective x and y coordinates. For example,

   ```
   *KMeans> sumPoints (4,2) (5,3)
   (9,5)
   ```

   [1 Mark]

4. Define a function `centroid :: [Point] -> Centroid` that will compute the centroid of a given list of `Point`s. For example,

   ```
   *KMeans> centroid points
   (4,4)
   ```

   [3 Marks]

5. Define a function `assign :: Point -> [Centroid] -> Assignment` that, given a point `p` and a list of centroids will return the pair `(p, m)` where `m` is the index of the centroid that is nearest to `p`. Recall that the centroids are numbered from 1. For example,

   ```
   *KMeans> assign (2,3) [(9,3), (1,1), (4,1)]
   ((2,3),2)
   ```

   i.e. centroid 2 (i.e. `(1,1)`) is the nearest to the point `(2,3)`.

   Note: If a point is equidistant from two centroids then it should be assigned to the centroid with the smallest index. Hint: use `minimum`.

   [4 Marks]

6. Using `assign`, define a function `assignAll :: [Point] -> [Centroid] -> [Assignment]` that will associate each point in the given list of points to the cluster whose centroid is nearest to it. The order of the points in the assignment should be preserved. For example,

   ```
   *KMeans> assignAll [(10,10), (50,70)] [(12,15), (39,51)]
   [((10,10),1),((50,70),2)]
   *KMeans> assignAll points fig1bAdjustedCentroids == fig1bAssignment
   True
   ```

   [1 Mark]

7. Define a function `adjustCentroids :: [Assignment] -> Int -> [Centroid]` that will compute the centroids for a given assignment of points to clusters; the `Int` argument is the number of clusters (k). For example,

```
*KMeans> adjustCentroids fig1aAssignment 3
[(4,1),(3,8),(6,5)]
*KMeans> adjustCentroids fig1aAssignment 3 == fig1bAdjustedCentroids
True
```

   Hint: use the function `lookupAll` defined above to extract the list of points assigned to a given cluster. Then use `centroid` to find the centroid of these points. You might find a list comprehension useful here. Recall that the centroids are numbered from 1.

   [2 Marks]

8. Define a function `cluster :: [Assignment] -> [Centroid] -> Int -> Clustering` that, given an assignment of points to centroids, a list of centroids and integer `k`, will iterate the algorithm to convergence. To do this you need to use `adjustCentroids` to adjust the given centroids using the given assignment and then (re)assign the points to the adjusted centroids. If the new assignment and adjusted centroids are *both* the same as those provided as arguments, then the function should terminate (base case); the result should be the (final) list of assignments and centroids of each cluster, i.e. a pair of type `Clustering`. You can use the next function to test your solution.

   Hint: use `assignAll`.

   [4 Marks]

9. Using `cluster` define a top-level function `kmeans :: [Point] -> Int -> Clustering` that, given a list of points and the required number of clusters, `k`, will cluster the given points into `k` clusters using the k-means algorithm. Recall that the initial set of centroids should be taken to be the locations of the first `k` points in the given list of points. A precondition of `cluster` is therefore that the list of points has at least k elements. For example,

```
*KMeans> kmeans points 3 == result
True
```

   [2 Marks]

10. (Bonus question) If we wish to render the clusters graphically we might assign a different colour to each cluster in an assignment, by mapping cluster number i to the i^{th} colour in the `Colour` data type. To get the bonus mark use a list comprehension, or higher-order function, together with the `toEnum` function from the `Enum` class, to define a function `applyColours :: Clustering -> Colouring`. For example,

```
*KMeans> applyColours (kmeans points 3)
([((3,1),Black),((2,9),Red),((5,6),Green),((7,8),Green),
((2,3),Black),((6,7),Green),((3,5),Red),((4,9),Red),
((9,3),Green),((1,1),Black),((2,1),Black),((9,1),Black),
((7,2),Black),((3,7),Red)],[(4,1),(3,7),(6,6)])
```

   [1 Mark]