1. Introduction.

Implementing the Fuzzy c-Means Algorithm

by Gagarine Yaikhom

A clustering algorithm organises items into groups based on a similarity criteria. The Fuzzy c-Means algorithm is a clustering algorithm where each item may belong to more than one group (hence the word *fuzzy*), where the degree of membership for each item is given by a probability distribution over the clusters.

2. Fuzzy c-Means Algorithm. The fuzzy c-means (FCM) algorithm is a clustering algorithm developed by Dunn, and later on improved by Bezdek. It is useful when the required number of clusters are pre-determined; thus, the algorithm tries to put each of the data points to one of the clusters. What makes FCM different is that it does not decide the absolute membership of a data point to a given cluster; instead, it calculates the likelihood (i.e., the degree of membership) that a data point will belong to that cluster. Hence, depending on the accuracy of the clustering that is required in practice, appropriate tolerance measures can be put in place. Since the absolute membership is not calculated, FCM can be extremely fast because the number of iterations required to achieve a specific clustering exercise corresponds to the required accuracy.

3. Iterations. In each iteration of the FCM algorithm, the following objective function J is minimised:

$$J = \sum_{i=1}^{N} \sum_{j=1}^{C} \delta_{ij} \parallel x_i - c_j \parallel^2 \tag{1}$$

Here, N is the number of data points, C is the number of clusters required, c_j is the centre vector for cluster j, and δ_{ij} is the degree of membership for the ith data point x_i in cluster j. The norm, $\parallel x_i - c_j \parallel$ measures the similarity (or closeness) of the data point x_i to the centre vector c_j of cluster j. Note that, in each iteration, the algorithm maintains a centre vector for each of the clusters. These data-points are calculated as the weighted average of the data-points, where the weights are given by the degrees of membership.

4. Degree of membership. For a given data point x_i, the degree of its membership to cluster j is calculated as follows:

$$\delta_{ij} = \frac{1}{\sum_{k=1}^{C} \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}} \tag{2}$$

where, m is the fuzziness coefficient and the centre vector c_j is calculated as follows:

$$c_j = \frac{\sum_{i=1}^{N} \delta_{ij} . x_i}{\sum_{i=1}^{N} \delta_{ij}} \tag{3}$$

In equation (3) above, δ_{ij} is the value of the degree of membership calculated in the previous iteration. Note that at the start of the algorithm, the degree of membership for data point i to cluster j is initialised with a random value θ_{ij}, $0 \le \theta_{ij} \le 1$, such that $\sum_j^C \delta_{ij} = 1$.

5. Fuzziness coefficient. In equation (2) the fuzziness coefficient m, where $1 \le m < \infty$, measures the tolerance of the required clustering. This value determines how much the clusters can overlap with one another. The higher the value of m, the larger the overlap between clusters. In other words, the higher the fuzziness coefficient the algorithm uses, a larger number of data points will fall inside a *fuzzy* band where the degree of membership is neither 0 nor 1, but somewhere in between.

6. Termination condition. The required accuracy of the degree of membership determines the number of iterations completed by the FCM algorithm. This measure of accuracy is calculated using the degree of membership from one iteration to the next, taking the largest of these values across all data points considering all of the clusters. If we represent the measure of accuracy between iteration k and $k+1$ with ϵ, we calculate its value as follows:

$$\epsilon = \Delta_i^N \Delta_j^C |\delta_{ij}^{k+1} - \delta_{ij}^k| \tag{4}$$

where, δ_{ij}^k and δ_{ij}^{k+1} are respectively the degree of membership at iteration k and $k+1$, and the operator Δ, when supplied a vector of values, returns the largest value in that vector.

7. Acknowledgement. Inspired by Matteo Matteucci's description of the algorithm.

8. FCM Program. Program `fcm` expects an input file, which contains the parameters for running the FCM algorithm. This input can be easily generated with the related program **`gen`**.

```
⟨ Include system libraries 18 ⟩;
⟨ Declare global variables 9 ⟩;
⟨ Define global functions 17 ⟩;
int main(int argc, char **argv)
{
  ⟨ Print usage information 43 ⟩;
  fcm(argv[1]);    /* run fuzzy c-means algorithm */
  ⟨ Print post processing information 44 ⟩;
  return 0;
}
```

9. Global variables and constants. The `fcm` program, upon execution, uses several global variables. These variables are the parameters required by the algorithm, and are normally initialised with values from the input data file generated by the **`gen`** program.

Variable *num_data_points* is initialised with the number of data points to be clustered; whereas, variable *num_clusters* is initialised with the number of clusters required. The FCM algorithm discovers clusters according to a preset number of clusters, in contrast to discovering an arbitrary number of clusters.

```
#define MAX_DATA_POINTS 10000    /* maximum number of data points */
#define MAX_CLUSTER 100    /* maximum number of clusters */
```

```
⟨ Declare global variables 9 ⟩ ≡
  int num_data_points;
  int num_clusters;
```

See also sections 10, 11, 12, 13, 14, 15, and 16.

This code is used in section 8.

10. Variable *num_dimensions* is initialised to the number of dimensions each data point has. For instance, if we are clustering Cartesian points in the two-dimensional Euclidean plane, we will have $num\_dimensions \equiv 2$.

```
#define MAX_DATA_DIMENSION 5    /* maximum number of data-point dimensions */
```

```
⟨ Declare global variables 9 ⟩ +≡
  int num_dimensions;
```

11. For every data point, the value that a dimension can take is bounded. We use a two dimensional matrix, *low_high*, to store the low and high values for each dimension. Every data point must fall within this value range.

```
⟨ Declare global variables 9 ⟩ +≡
  double low_high[MAX_DATA_DIMENSION][2];
```

12. A two dimensional array, *degree_of_memb*, stores the degree of membership for each of the data points to each of the clusters. Thus, for instance, *degree_of_memb*[i][j] gives a measure of how likely the data point i will belong to cluster j. This matrix is updated dynamically during the iterations.

```
⟨ Declare global variables 9 ⟩ +≡
  double degree_of_memb[MAX_DATA_POINTS][MAX_CLUSTER];
```

13. The clustering algorithm uses a measure of accuracy, *epsilon*, to judge whether a clustering exercise can be judged satisfactory. This eventually determines the number of iterations that must be executed before the algorithm terminates.

```
⟨ Declare global variables 9 ⟩ +≡
  double epsilon;    /* termination criterion */
```

14. The degree of fuzziness, *fuzziness*, gives a measure of tolerance while carrying out the clustering exercise, and therefore, affects the degree of membership calculations.

```
⟨ Declare global variables 9 ⟩ +≡
  double fuzziness;
```

15. Every data point has a coordinate $(x_1, x_2, ..., x_d)$ given by an ordered set of values corresponding to each of the dimensions (where d gives the highest dimension for a data point). This coordinate depends on the number of dimensions which each data point must have. For instance, for points on a real line, the number of dimensions is 1. For points on the plane, the number of dimensions is 2, where each data point is represented by their 2D-coordinate, e.g., (x, y). For points inside a volume, the number of dimensions is 3, where each data point is represented by their 3D-coordinate, e.g. (x, y, z). This implementation of the FCM algorithm works with any number of dimensions less than `MAX_DATA_DIMENSION`.

```
⟨ Declare global variables 9 ⟩ +≡
  double data_point[MAX_DATA_POINTS][MAX_DATA_DIMENSION];
```

16. For each of the clusters, a temporary centre point is maintained during each of the iterations. The centre points for all of the clusters are maintained as a two dimensional array (each row representing a vector in the same coordinate system as the data points). The values stored in this array are updated during the iterations depending on the degree of membership of the data points to each of the clusters.

```
⟨ Declare global variables 9 ⟩ +≡
  double cluster_centre[MAX_CLUSTER][MAX_DATA_DIMENSION];
```

17. Initialisation. Function *init* initialises the execution parameters by parsing the input data file. It takes as input argument the name of the input data file, *fname*, and returns `0` if the initialisation was successful. Otherwise, `-1` is returned.

```
⟨ Define global functions 17 ⟩ ≡
  int init(char *fname)
  {
    int i, j, r, rval;
    FILE *f;
    double t, s;
    ⟨ Open input file 28 ⟩;
    ⟨ Read number of data points, clusters and dimensions 29 ⟩;
    ⟨ Read fuzziness coefficients 30 ⟩;
    ⟨ Read required accuracy 31 ⟩;
    ⟨ Initialise data points 32 ⟩;
    ⟨ Initialise degree of membership matrix 19 ⟩;
    fclose(f);
    return 0;
  failure: fclose(f);
    exit(1);
  }
```

See also sections 20, 23, 24, 25, 26, 36, 45, and 47.

This code is used in section 8.

18. Randomise degree-of-membership matrix. We use a random number generator to initialise the degree of membership before applying the FCM algorithm. To do this, we use the function `rand()` from the standard `math` library.

```
⟨ Include system libraries 18 ⟩ ≡
#include <math.h>
```

See also section 49.

This code is used in section 8.

19. For every data point, in relation to one of the clusters, we assign a random probability that the point will belong to that cluster. The value is a real number between `0.0` and `1.0` inclusive. Since the sum of the probabilities for all of the clusters for a given data point should be `1.0`, we adjust the probability of the first cluster after we have assigned probabilities for all of the remaining clusters.

```
⟨ Initialise degree of membership matrix 19 ⟩ ≡
  for (i = 0; i < num_data_points; i++) {
    s = 0.0;    /* probability sum */
    r = 100;    /* remaining probability */
    for (j = 1; j < num_clusters; j++) {
      rval = rand() % (r + 1);
      r −= rval;
      degree_of_memb[i][j] = rval/100.0;
      s += degree_of_memb[i][j];
    }
    degree_of_memb[i][0] = 1.0 − s;
  }
```

This code is used in section 17.

20. Calculation of centre vectors. Function *calculate_centre_vectors* updates the centre vectors for each of the clusters. The aim of the algorithm is to continue updating this centre of vectors until it is close to the actual value. The closeness to the actual value is determined by the termination criterion (epsilon).

```
⟨ Define global functions 17 ⟩ +≡
  int calculate_centre_vectors()
  {
    int i, j, k;
    double numerator, denominator;
    double t[MAX_DATA_POINTS][MAX_CLUSTER];
    ⟨ Precompute powers of degree of membership 21 ⟩;
    ⟨ Calculate centre vectors 22 ⟩;
    return 0;
  }
```

21. Precompute powers of degree-of-membership. The values calculated here are used in two parts of the following calculation of centre of vectors. We have moved the computation of powers here to reuse the values, so that we don't have to calculate the powers again. Calculation of powers are expensive.

```
⟨ Precompute powers of degree of membership 21 ⟩ ≡
  for (i = 0; i < num_data_points; i++) {
    for (j = 0; j < num_clusters; j++) {
      t[i][j] = pow(degree_of_memb[i][j], fuzziness);
    }
  }
```

This code is used in section 20.

22. ⟨ Calculate centre vectors 22 ⟩ ≡

```
for (j = 0; j < num_clusters; j++) {
   for (k = 0; k < num_dimensions; k++) {
      numerator = 0.0;
      denominator = 0.0;
      for (i = 0; i < num_data_points; i++) {
         numerator += t[i][j] * data_point[i][k];
         denominator += t[i][j];
      }
      cluster_centre[j][k] = numerator / denominator;
   }
}
```

This code is used in section 20.

23. Calculation of norm. Function *get_norm* is one of the most important calculations of the FCM algorithm. It determines the closeness of a data point to the centre of vectors for a given cluster. The calculation of the norm depends on the number of dimensions the data points have.

⟨ Define global functions 17 ⟩ +≡

```
double get_norm(int i, int j)
{
   int k;
   double sum = 0.0;

   for (k = 0; k < num_dimensions; k++) {
      sum += pow(data_point[i][k] − cluster_centre[j][k], 2);
   }
   return sqrt(sum);
}
```

24. Update degree of membership. Function *get_new_value* calculates the new degree of membership for the data point i in cluster j.

⟨ Define global functions 17 ⟩ +≡

```
double get_new_value(int i, int j)
{
   int k;
   double t, p, sum;

   sum = 0.0;
   p = 2/(fuzziness − 1);
   for (k = 0; k < num_clusters; k++) {
      t = get_norm(i, j)/get_norm(i, k);
      t = pow(t, p);
      sum += t;
   }
   return 1.0/sum;
}
```

25. Function *update_degree_of_membership* updates the degree-of-membership values for all of the data points. Since we want to stop the iteration when all of the points are close enough to one of the centre of vectors, this function returns the maximum difference between the old value and the new value, so that it can be checked against the termination condition.

```
⟨ Define global functions 17 ⟩ +≡
  double update_degree_of_membership( )
  {
    int i, j;
    double new_uij;
    double max_diff = 0.0, diff;
    for (j = 0; j < num_clusters; j++) {
      for (i = 0; i < num_data_points; i++) {
        new_uij = get_new_value(i, j);
        diff = new_uij − degree_of_memb[i][j];
        if (diff > max_diff) max_diff = diff;
        degree_of_memb[i][j] = new_uij;
      }
    }
    return max_diff;
  }
```

26. The FCM algorithm. Function *fcm* encapsulates the main phases of the FCM algorithm. It contains the skeleton of the algorithm, which corresponds to the mathematical description in the introduction.

```
⟨ Define global functions 17 ⟩ +≡
  int fcm(char *fname)
  {
    double max_diff;
    init(fname);
    do {
      calculate_centre_vectors( );
      max_diff = update_degree_of_membership( );
    } while (max_diff > epsilon);
    return 0;
  }
```

27. Input. This sections lists the input functions and code segments that are used for reading initialisation values, parameters from the input data file.

28. ⟨ Open input file 28 ⟩ ≡

```
  if ((f = fopen(fname, "r")) ≡ Λ) {
    printf("Failed␣to␣open␣input␣file.");
    return −1;
  }
```

This code is used in section 17.

29. ⟨ Read number of data points, clusters and dimensions 29 ⟩ ≡

```
fscanf(f, "%d␣%d␣%d", &num_data_points, &num_clusters, &num_dimensions);
if (num_clusters > MAX_CLUSTER) {
  printf("Number␣of␣clusters␣should␣be␣<␣%d\n", MAX_CLUSTER);
  goto failure;
}
if (num_data_points > MAX_DATA_POINTS) {
  printf("Number␣of␣data␣points␣should␣be␣<␣%d\n", MAX_DATA_POINTS);
  goto failure;
}
if (num_dimensions > MAX_DATA_DIMENSION) {
  printf("Number␣of␣dimensions␣should␣be␣>=␣1.0␣and␣<␣%d\n", MAX_DATA_DIMENSION);
  goto failure;
}
```

This code is used in section 17.

30. ⟨ Read fuzziness coefficients 30 ⟩ ≡

```
fscanf(f, "%lf", &fuzziness);
if (fuzziness ≤ 1.0) {
  printf("Fuzzyness␣coefficient␣should␣be␣>␣1.0\n");
  goto failure;
}
```

This code is used in section 17.

31. ⟨ Read required accuracy 31 ⟩ ≡

```
fscanf(f, "%lf", &epsilon);
if (epsilon ≤ 0.0 ∨ epsilon > 1.0) {
  printf("Termination␣criterion␣should␣be␣>␣0.0␣and␣<=␣1.0\n");
  goto failure;
}
```

This code is used in section 17.

32. ⟨ Initialise data points 32 ⟩ ≡

```
for (i = 0; i < num_data_points; i++) {
  for (j = 0; j < num_dimensions; j++) {
    fscanf(f, "%lf", &data_point[i][j]);
    if (data_point[i][j] < low_high[j][0]) low_high[j][0] = data_point[i][j];
    if (data_point[i][j] > low_high[j][1]) low_high[j][1] = data_point[i][j];
  }
}
```

This code is used in section 17.

33. Output. This sections lists the output functions and code segments that are used when printing output, error messages, etc. either to the standard output stream, of an output file.

34. ⟨ Open output file 34 ⟩ ≡

```
if (fname ≡ Λ) f = stdout;
else if ((f = fopen(fname, "w")) ≡ Λ) {
  printf("Cannot␣create␣output␣file.\n");
  exit(1);
}
```

This code is used in sections 45 and 47.

35. ⟨ Print execution parameters 35 ⟩ ≡

```
printf("Number␣of␣data␣points:␣%d\n", num_data_points);
printf("Number␣of␣clusters:␣%d\n", num_clusters);
printf("Number␣of␣data-point␣dimensions:␣%d\n", num_dimensions);
printf("Accuracy␣margin:␣%lf\n", epsilon);
```

This code is used in section 44.

36. This function outputs the current membership matrix so that it can be plotted using Gnuplot. For information on Gnuplot, Google gnuplot. NOTE: This will only work when the number of dimensions is 2. That is the points are on a plane. This may be extended for higher dimensions.

⟨ Define global functions 17 ⟩ +≡

```
int gnuplot_membership_matrix()
{
  int i, j, cluster;
  char fname[100];
  double highest;
  FILE *f[MAX_CLUSTER];
  ⟨ Check number of dimensions is 2 37 ⟩;
  ⟨ Create data file for each cluster 38 ⟩;
  ⟨ Send data points to respective cluster data file 40 ⟩;
  ⟨ Close the cluster data files 41 ⟩;
  ⟨ Write the gnuplot script 42 ⟩;
  return 0;
}
```

37. ⟨ Check number of dimensions is 2 37 ⟩ ≡

```
if (num_dimensions ≠ 2) {
  printf("Plotting␣the␣cluster␣only␣works␣when␣the\n");
  printf("number␣of␣dimensions␣is␣two.␣This␣will␣create\n");
  printf("a␣two-dimensional␣plot␣of␣the␣cluster␣points.\n");
  exit(1);
}
```

This code is used in section 36.

38. We create a separate file for each of the clusters. Each file contains data points that has the highest degree of membership.

⟨ Create data file for each cluster 38 ⟩ ≡

```
for (j = 0; j < num_clusters; j++) {
  sprintf(fname, "cluster.%d", j);
  if ((f[j] = fopen(fname, "w")) ≡ Λ) {
    printf("Could␣not␣create␣%s\n", fname);
    ⟨ Cleanup cluster data files and return 39 ⟩;
  }
  fprintf(f[j], "#Data␣points␣for␣cluster:␣%d\n", j);
}
```

This code is used in section 36.

39. ⟨ Cleanup cluster data files and return 39 ⟩ ≡

```
for (i = 0; i < j; i++) {
   fclose(f[i]);
   sprintf(fname, "cluster.%d", i);
   remove(fname);
}
return −1;
```

This code is used in sections 38 and 42.

40. ⟨ Send data points to respective cluster data file 40 ⟩ ≡

```
for (i = 0; i < num_data_points; i++) {
   cluster = 0;
   highest = 0.0;
   for (j = 0; j < num_clusters; j++) {
      if (degree_of_memb[i][j] > highest) {
         highest = degree_of_memb[i][j];
         cluster = j;
      }
   }
   fprintf(f[cluster], "%lf␣%lf\n", data_point[i][0], data_point[i][1]);
}
```

This code is used in section 36.

41. ⟨ Close the cluster data files 41 ⟩ ≡

```
for (j = 0; j < num_clusters; j++) {
   fclose(f[j]);
}
```

This code is used in section 36.

42. ⟨ Write the gnuplot script 42 ⟩ ≡

```
if ((f[0] = fopen("gnuplot.script", "w")) ≡ Λ) {
   printf("Could␣not␣create␣gnuplot.script.\n");
   ⟨ Cleanup cluster data files and return 39 ⟩;
}
fprintf(f[0], "set␣terminal␣png␣medium\n");
fprintf(f[0], "set␣output␣\"cluster_plot.png\"\n");
fprintf(f[0], "set␣title␣\"FCM␣clustering\"\n");
fprintf(f[0], "set␣xlabel␣\"x-coordinate\"\n");
fprintf(f[0], "set␣ylabel␣\"y-coordinate\"\n");
fprintf(f[0], "set␣xrange␣[%lf␣:␣%lf]\n", low_high[0][0], low_high[0][1]);
fprintf(f[0], "set␣yrange␣[%lf␣:␣%lf]\n", low_high[1][0], low_high[1][1]);
fprintf(f[0], "plot␣'cluster.0'␣using␣1:2␣with␣points␣pt␣7␣ps␣1␣lc␣1␣notitle");
for (j = 1; j < num_clusters; j++) {
   sprintf(fname, "cluster.%d", j);
   fprintf(f[0], ",\\\n'%s'␣using␣1:2␣with␣points␣␣pt␣7␣ps␣1␣lc␣%d␣notitle", fname, j + 1);
}
fprintf(f[0], "\n");
fclose(f[0]);
```

This code is used in section 36.

43. ⟨ Print usage information 43 ⟩ ≡

```
printf ("-------------------------------------------------------------------------\n");
if (argc ≠ 2) {
   printf ("USAGE:␣fcm␣<input␣file>\n");
   exit (1);
}
```

This code is used in section 8.

44. ⟨ Print post processing information 44 ⟩ ≡

```
⟨ Print execution parameters 35 ⟩;
print_membership_matrix ("membership.matrix");
gnuplot_membership_matrix ( );
printf ("-------------------------------------------------------------------------\n");
printf ("The␣program␣run␣was␣successful...\n");
printf ("Storing␣membership␣matrix␣in␣file␣'membership.matrix'\n\n");
printf ("If␣the␣points␣are␣on␣a␣plane␣(2␣dimensions)\n");
printf ("the␣gnuplot␣script␣was␣generated␣in␣file␣'gnuplot.script',␣and\n");
printf ("the␣gnuplot␣data␣in␣files␣cluster.[0]...␣\n\n");
printf ("Process␣'gnuplot.script'␣to␣generate␣graph:␣'cluster_plot.png'\n\n");
printf ("NOTE:␣While␣generating␣the␣gnuplot␣data,␣for␣each␣of␣the␣data␣points\n");
printf ("the␣corresponding␣cluster␣is␣the␣one␣which␣has␣the␣highest\n");
printf ("degree-of-membership␣as␣found␣in␣'membership.matrix'.\n");
printf ("-------------------------------------------------------------------------\n");
```

This code is used in section 8.

45. Procedure *print_data_points* prints the data points.

⟨ Define global functions 17 ⟩ +≡

```
void print_data_points (char *fname)
{
   int i, j;
   FILE *f;
   ⟨ Open output file 34 ⟩;
   fprintf (f, "Data␣points:\n");
   ⟨ Print data points 46 ⟩;
   if (fname ≡ Λ) fclose (f);
}
```

46. ⟨ Print data points 46 ⟩ ≡

```
for (i = 0; i < num_data_points; i++) {
   printf ("Data[%d]:␣", i);
   for (j = 0; j < num_dimensions; j++) {
      printf ("%.5lf␣", data_point[i][j]);
   }
   printf ("\n");
}
```

This code is used in section 45.

47. Procedure *print_membership_matrix* prints the current membership matrix. It take as input the name of the output file. If this is `NULL` the output is directed to `stdout`.

```
⟨ Define global functions 17 ⟩ +≡
  void print_membership_matrix(char *fname)
  {
    int i, j;
    FILE *f;
    ⟨ Open output file 34 ⟩;
    fprintf(f, "Membership␣matrix:\n");
    ⟨ Print the membership matrix 48 ⟩;
    if (fname ≡ Λ) fclose(f);
  }
```

48. ⟨ Print the membership matrix 48 ⟩ ≡

```
  for (i = 0; i < num_data_points; i++) {
    fprintf(f, "Data[%d]:␣", i);
    for (j = 0; j < num_clusters; j++) {
      fprintf(f, "%lf␣", degree_of_memb[i][j]);
    }
    fprintf(f, "\n");
  }
```

This code is used in section 47.

49. Includes system libraries that defines constants and function prototypes.

```
⟨ Include system libraries 18 ⟩ +≡
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

50. Index.

FCM