Database Internals Software Project Report: A Spatial Dataset Clustering Tool

Kevin Pulo, `kev@cs.usyd.edu.au`
Garrick Welsh, `gaz@cs.usyd.edu.au`

May 2000

1 Introduction

As many people are aware there is a growing need for new methods in the area of data mining. One such area in which there are huge amounts of data which need to be processed is GIS, or geographical information systems.

To this end we decided to look at one of the ways to get data associated with other pieces of data, called clustering. Clustering can have an extremely bad time complexity for naive algorithms. For this reason we have looked at new ways of clustering such data. We found a new clustering algorithm for k-MEDOIDS which has a sub-quadratic time complexity.

2 Software

We wrote a few tools to test the algorithm's ability to cluster, compared to some other algorithms.

The tools that we have made are:

`generate` Generates random datasets to test the algorithms on.

`cluster` Runs the various clustering algorithms on given datasets.

`display` Displays a clustered dataset graphically.

2.1 Design

All of the tools read their input (if any) from standard input (stdin), and write their output (if any) to standard output (stdout).

The programs are written in ANSI C and C++, using the LEDA library. They could easily be extended to be part of some larger data mining application or set of shell tools.

Points are specified as two real valued numbers, that is, a point in the real plane. Clusters are specified by the non-negative integers, the cluster -1 denotes the set of *outlier* points. Lastly, the points in a dataset may have "annotation", which is simply arbitrary data associated with the point.

2.1.1 Data Format

The data format was chosen to be simple ASCII formats which are easily parsed by both programs and humans. There are two data formats used, an unclustered dataset and a clustered dataset.

The unclustered dataset simply specifies points followed by their annotations, one per line. Each line has the format:

```
(<real>,<real>) <string>
```

The two real (decimal) numbers specify the location of the point in the plane, and the string is a string representation of the point's annotation.

The clustered dataset is basically the same as the unclustered dataset, except that each point is prefixed with the cluster number it belongs to. That is,

```
<int> (<real>,<real>) <string>
```

The integer at the start of the line denotes the cluster the point belongs to, or -1 if the point has been designated as an outlier.

The one exception to this is where the annotation is "`representative`". This indicates that the point is actually the representative point of the cluster.

2.1.2 Data structures

We defined a single class, Cluster. It contains a list of points and a representative point, and provides operations to add and remove points and calculate the mean point of the points in the cluster.

A LEDA point_set is used to store the dataset and compute its Delaunay triangulation. The set of clusters is stored as a LEDA array of Cluster objects. A separate special Cluster object is used to store the outlier points (the representative of this Cluster is never used, nor does it have any meaning in this context). Annotations are stored in a LEDA dictionary which maps points to the string representation of their annotation data.

2.1.3 Adding a new algorithm

The following steps show how to add a clustering algorithm called foo to the `cluster` program.

1. Create a pair of files `cluster_foo.cc` and `cluster_foo.hh`.
2. In `cluster_foo.hh`, `#include "cluster.hh"` and provide a function prototype `extern void cluster_foo(int k);`, as is done in the various existing `cluster_*.hh` files.
3. In `cluster_foo.cc`, `#include "cluster_foo.hh"`, and then put the clustering algorithm into a function `void cluster_foo(int k)`. All other functions and data should be marked as `static`.
4. In `Makefile`, add `cluster_foo.o` to the end of the "`cluster: ...`" line.
5. In `Makefile`, add `cluster_foo.hh` to the end of the "`algorithms.o: ...`" line.
6. In `Makefile`, add the line "`cluster_foo.o: cluster.hh`".
7. In algorithms.cc, `#include "cluster_foo.hh"` and then add to the end of the `registerAlgorithms()` function "`algorithms.insert("foo", cluster_foo);`".

2.2 Compiling

Following are details of the steps required to compile the clustering tools.

2.2.1 System requirements

The clustering tools have so far only been properly tested on Intel based Linux machines. There has been some small amount of testing on Sparc Solaris as well. A Pentium should be fine on small datasets, though nearly all testing was done on Celeron or Pentium III machines.

2.2.2 Dependant libraries

Besides the standard ANSI C/C++ libraries we use an algorithmic and computational geometry library called LEDA (Library of Efficient Data structures and Algorithms). This can be found at http://www.mpi-sb.mpg.de/LEDA.

2.2.3 Required programs

- An ANSI C++ compiler (preferably `g++`). Note that you should compile the clustering tools with the same compiled as your LEDA libraries were compiled with.
- GNU `make` version 3.76 or higher.

2.2.4 Step by step compilation instructions

1. Download and install LEDA.
2. The Makefile is designed for dynamic library linking. You can either change your `LD_LIBRARY_PATH` variable or your `/etc/ld.so.conf` file to include the LEDA installation directory. Also, be sure to make the shared LEDA libraries. This can be done by typing `make shared` in the LEDA installation directory.
3. If you wish to make a static binary add the `-static` flag to the `LDFLAGS` and `LEDA_LIBS` variables in the Makefile.
4. Change the `LEDA_ROOT` variable in the Makefile to the LEDA installation directory.
5. Type `make` in the source directory to compile and link the clustering tools.

2.3 Usage

2.3.1 General usage

How do we use these programs that we've compiled? We run the programs in the order of `generate`, `cluster` then `display`. A small blurb earlier describes what each program basically does. We'll go into greater detail here. To get a quick demonstration of the program run

```
./generate | ./cluster | ./display
```

How does that work? `generate` creates a data set. The generated data set has noise points (ie points not in a cluster) and cluster points. It then outputs this data. `cluster` then reads in the data sent from `generate`. The `cluster` program then uses the default clustering algorithm (k-MEANS) on the data the default algorithm is random, where we randomly assign a cluster to each point. The data output from cluster is then read in by `display`. This is then reinterpreted and displayed on the screen.

2.3.2 `generate`

The `generate` program creates a random data set in the following way. It creates N random noise points. It then creates k clusters by choosing k points which will be the centres of circles of radius r. We then randomly choose a cluster and a point within the circle for each of the n cluster points being creating.

No input is required on standard input, and the unclustered dataset is written to standard output.

The annotation of each point is the cluster number it belongs to, or -1 for outliers. This aids greatly in evaluating how well a given clustering algorithm performs on the dataset.

Usage:
`generate` *[OPTIONS]*

`-help` Outputs a brief usage message.

`-r` *<real>* Specifies the radius of the clusters (default 0.01).

`-k` *<int>* Specifies the number of clusters (default 4).

`-n` *<int>* Specifies the number of clustered points (default 250).

`-N` *<int>* Specifies the number of noise points (default 50).

2.3.3 `cluster`

The `cluster` program reads an unclustered dataset from standard input, such as one created by the `generate` program. It then runs the specified clustering algorithm on this dataset, and outputs the resultant clustered dataset to standard output. The representatives of clusters are also output, and the annotations of the dataset points are preserved.

The algorithms themselves will be detailed later.

Usage:
`cluster` *[OPTIONS]*

`--help` Outputs a brief usage message.

`--debug` Output debugging information.

`--algorithm` *algorithmname [k]* Specifies the clustering algorithm to use, and optionally a suggestion for the number of clusters (k) for those algorithms which need or benefit from it. *algorithm_name* defaults to "kmeans", k defaults to 4.

Available algorithms:

kmeans The standard k-Means algorithm, Section 3.3.

tb The Teitz-Bart k-Medoids heuristic, Section 3.4.

ech The ECH variant of the TB k-Medoids heuristic, Section 3.5.

random Random clustering (equal sized clusters), Section 3.2.

cheat Cheating clustering, using the annotation, Section 3.1.3.

ascending Puts each point into its own cluster, Section 3.1.2.

single Puts all points into a single cluster, Section 3.1.1.

2.3.4 `display`

The `display` program reads a clustered dataset from standard input and displays it graphically in an X window, showing the points of each cluster in a different colour. It can also show the Delaunay diagram, the representatives of the clustering and the edges from the points to their representatives (the "$M(C)$" edges).

Usage:
  `display` *`[OPTIONS]`*

`--help` Outputs a brief usage message.

`--debug` Output debugging information.

`--nodesize` *`<int>`* This takes an integer as a parameter, which specifies the radius of the displayed nodes.

`--delaunay` Toggles showing the delaunay triangulation of the points (default off).

`--representatives` Toggles showing the representatives points (default on).

`--mcedges` Toggles showing the $M(C)$ edges (default on).

Interactive commands available in the `display` window:

<ESCAPE>, q Quits the program.

d Toggles the drawing of the Delaunay triangulation.

r Toggles the drawing of the representatives.

m Toggles the drawing of the $M(C)$ edges.

3 Algorithms Implemented

3.1 Testing and debugging

3.1.1 Single

This algorithm simply puts every point into a single cluster, and is only useful for testing and debugging.

The cluster's representative is set to the mean of the other points.

3.1.2 Ascending

This algorithm simply puts each point into its own cluster, and is only useful for testing and debugging.

The representatives of the clusters are set to the single point in the cluster.

3.1.3 Cheat

The generate program outputs the cluster number as the annotation of each point. The cheat algorithm uses this information to simply put the points into their original clusters, as given by the generate program. Thus, it is a "perfect" clustering algorithm, because it uses the cluster number used by the `generate` program when it creates the clusters.

For each of the clusters, the representative is set to the mean of the points in that cluster.

3.2 Random

The random clustering algorithm randomly distributes the given points into k clusters of equal size.

The representatives are set to the mean points of the clusters. The value of k must be given on the command line.

This is useful for both testing and debugging, and is also utilized by the k-MEANS algorithm in finding an initial clustering.

3.3 k-MEANS

The k-MEANS clustering algorithm implements the standard k-MEANS algorithm, as detailed in Section 2 of [1]. This takes an iterative improvement over an initial random clustering, which is provided by the above random clustering algorithm. Each iteration computes the representatives of the cluster as the mean of the points in the cluster, and then using these representatives computes a clustering, where each point is assigned to the cluster of its closest representative.

It runs in $O(n)$ time, however, often doesn't produce good quality clusterings. It requires the desired number of clusters, k, to be specified on the command line.

3.4 TB (k-MEDOIDS)

The TB clustering algorithm implements the TB heuristic to the k-MEDOIDS (or p-median) problem, as detailed in Section 3 of [1], originally proposed in [2]. It is similar to k-MEANS, except that it has the restriction that the cluster representatives must be points of the dataset.

It runs in $\Omega(n^2)$ time, but gives substantially better quality clusterings than k-MEANS. It requires the desired number of clusters, k, to be specified on the command line.

3.5 ECH (Delaunay variant of TB)

The ECH clustering algorithm implements the variant of the TB heuristic, which is described in [1]. It uses the Delaunay triangulation of the dataset to find an efficient approximation to the quantity which TB minimizes.

It runs in $O(n \log n)$ time, and gives clusterings of approximately the same quality as the TB heuristic. Further, it does not need to be given any value for k — it will ignore any value which is given on the command line, and instead find the most appropriate value of k from the data itself.

Unfortunately, due to the complicated nature of the algorithm, time constraints have prevented our implementation of the ECH algorithm from being completed. Whilst it has been fully implemented, when its output is viewed it is clear that it is flawed in some way.

References

[1] V. Estivill-Castro and M. E. Houle. Robust Distance-Based Clustering with Applications to Spatial Data Mining. *Algorithmica (Special Issue: Algorithms for Geographical Information).* To appear in 2000. `http://www.cs.usyd.edu.au/~meh/papers/gisclust.ps.gz`

[2] M. B. Teitz and P. Bart. Heuristic methods for Estimating the generalized vertex median of a weighted graph. *Operations Research*, 16:955–961, 1968.