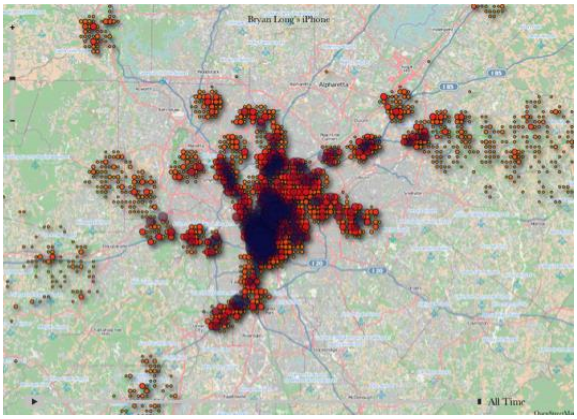
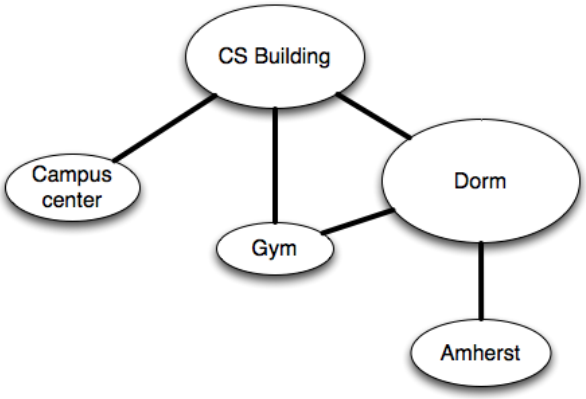
Chapter 8: GPS Clustering and Analytics

Location information is crucial for analyzing sensor data and health inferences from mobile and wearable devices. For example, let us say you monitored your stress levels throughout the day and wanted to visualize the data. Location and time is crucial to analyzing the data --- for example, you might detect that stress is highest at work rather than home, or that stress is highest in evenings. But how do you take location data, and visualize it in a way that makes it possible for you to draw such conclusions? If you took all the points where you took measurements and plotted it on a map (such as in the figure on the left), it wouldn't be particularly meaningful. It would be quite useless to tell someone, "You were at





33.93885N, 84.33697W at 5pm on 3/17/2014".

Clearly, we need a more logical way to find points that an individual might consider significant is to look at where the individual spends her time. For example, the figure on the right shows a logical representation of the location data, where locations have been clustered into logical places. Here, the size of the clusters show how much time you spent in a particular logical place -- you spent a lot of time in the CS building and at the Dorm. The lines between the clusters show how you typically moved between places -- you typically go from your Dorm to Amherst downtown rather than from the CS department to Amherst downtown. Once you have such a representation, you can overlay the data with information about other parameters like heart rate (higher at the gym, presumably), and so on. But how do we go from the raw data plotted on the left to the logical place representation shown on the right? In this chapter, we provide some ideas on how to cluster raw GPS data into meaningful places.

Clustering location data

Before we launch into the algorithm, let us start by looking at the challenges that one faces while clustering GPS data.

- **Noise**: GPS data is noisy. You might be standing at the same location, but GPS readings can be off by several meters. The GPS error depends on numerous factors including the number of satellites visible to the device, indoor vs outdoor environment, tall buildings in the vicinity, weather, and GPS device characteristics. In clustering GPS data to determine the significant places, care needs to be taken to filter this data.
- **Meaningful clusters**: One of the major challenges is identifying which GPS co-ordinates correspond to "meaningful" clusters. For example, there may be GPS points recorded while you

are driving on the freeway, but you are less likely to be interested in freeway than work or home. So, we need a way to identify meaningful clusters, and discard irrelevant ones.

- **Semantic location**: While clustering is useful, the ultimate goal is to get semantic location i.e. Home, Work, Coffee Shop, etc rather than Cluster 1 and Cluster 2. So, another challenge is how we convert from clusters to places.

GPS Clustering

We are going to address these problems in three phases. First, we pre-process data to filter out noisy readings, to reduce the number of readings, and to remove readings that correspond to meaningless locations. Second, we are going to use one of the most common clustering algorithms, referred to as k-means, to cluster the remaining data points. Third, we are going to convert from clusters to semantic locations.

Phase 1: Pre-processing (removing noise and outliers)

The pre-processing step has the following goals: a) remove noisy data, b) remove meaningless points where you did not spend sufficient time, c) reduce the amount of GPS data that a clustering algorithm (dbscan or k-means) has to process in-order to speed it up.

1. **Remove low density points**: We look for points that have few neighbors --- in other words, it doesn't look like you spent substantial enough time in that location to be worthwhile to process. For example, you could choose a radius of 20 meters, and threshold of at least 50 neighbors within the 20 meter threshold. Assuming that you were collecting GPS data once a minute, that means that you want to have spent at least 50 minutes in that location, else it is not something that you care about. This approach also filters out noisy readings that are far out since these points will have few neighbors.
2. **Remove points with movement**. GPS returns speed in addition to co-ordinates. If speed > 0 (or something small, like 0.1), then discard the GPS point. This makes sure that you don't process data while you are walking or driving, which we are not interested in. (Note that if you are interested in travel patterns, you may want to keep data involving movement, but this is not important for identifying semantic locations, which is our objective)
3. **Reduce data for stationary locations**. When you are stationary (e.g. sitting at a chair), your GPS location will not change for a long time. This will result in too many redundant points that you need to process, which will slow down any clustering algorithm (e.g. k-means). You can address this by not storing points where the change from the previously stored location is less than a small threshold (e.g. 1 meter). *Note that you may not want to remove too many such samples, since otherwise it can change the k-means clustering result too much. Try running k-means without this step, and only use this method if you find that it is too slow!*

Phase II: Clustering

There are many methods for clustering GPS points into semantic locations. We will describe two such approaches in this document - k-means clustering and DBSCAN clustering.

K-means Clustering

The key parameter that you have to select for k-means is **k**, the number of clusters. You may typically choose **k** based on the number of clusters you expect in the data, perhaps you expect about 10 clusters as the places where you typically stay in a day.

Given **k**, the k-means algorithm consists of an iterative algorithm with four steps.

1. Select *K* initial centroids at random from the points.
2. **repeat**
    a. Assign each object to the cluster with the nearest centroid (in terms of distance).
    b. Re-compute each centroid as the mean of the objects assigned to it.
3. **until** centroids do not change.

While this algorithm sometimes produces suboptimal clusterings, it is fast and really easy to to implement.

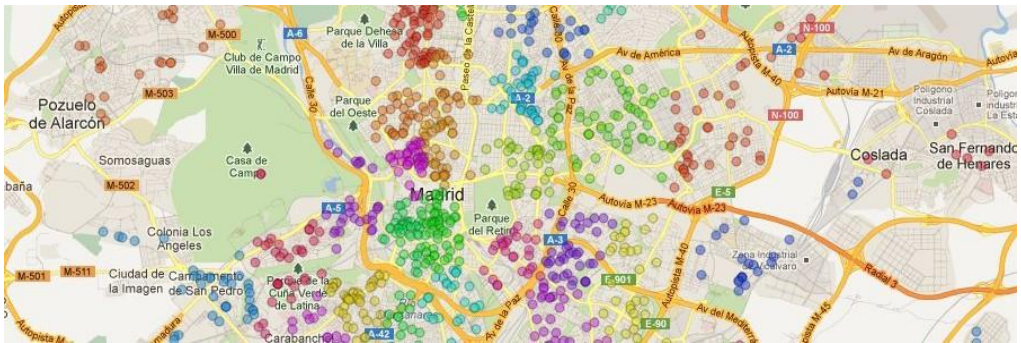
Lets look at a simple SQL implementation (from Joni Salonen [2]). Suppose we have some location data, already geocoded into latitude-longitude pairs, and we want to find clusters of locations that lie close to each other. We'll use two tables, `gps_data` to store the data and the cluster assigned to each point, and `gps_clusters` for the cluster centers:

```
create table gps_data (id int primary key, cluster_id int, lat double, lng double);
create table gps_clusters (id int auto_increment primary key, lat double, lng
double);
```

The K-means algorithm can now be implemented with the following procedure.

```
DELIMITER //
CREATE PROCEDURE kmeans(v_K int)
BEGIN
TRUNCATE gps_clusters;
-- initialize cluster centers
INSERT INTO gps_clusters (lat, lng) SELECT lat, lng FROM gps_data LIMIT v_K;
REPEAT
    -- assign clusters to data points
    UPDATE gps_data d SET cluster_id = (SELECT id FROM gps_clusters c
        ORDER BY POW(d.lat-c.lat,2)+POW(d.lng-c.lng,2) ASC LIMIT 1);
    -- calculate new cluster center
    UPDATE gps_clusters C, (SELECT cluster_id,
        AVG(lat) AS lat, AVG(lng) AS lng
        FROM gps_data GROUP BY cluster_id) D
    SET C.lat=D.lat, C.lng=D.lng WHERE C.id=D.cluster_id;
UNTIL ROW_COUNT() = 0 END REPEAT;
END//
```

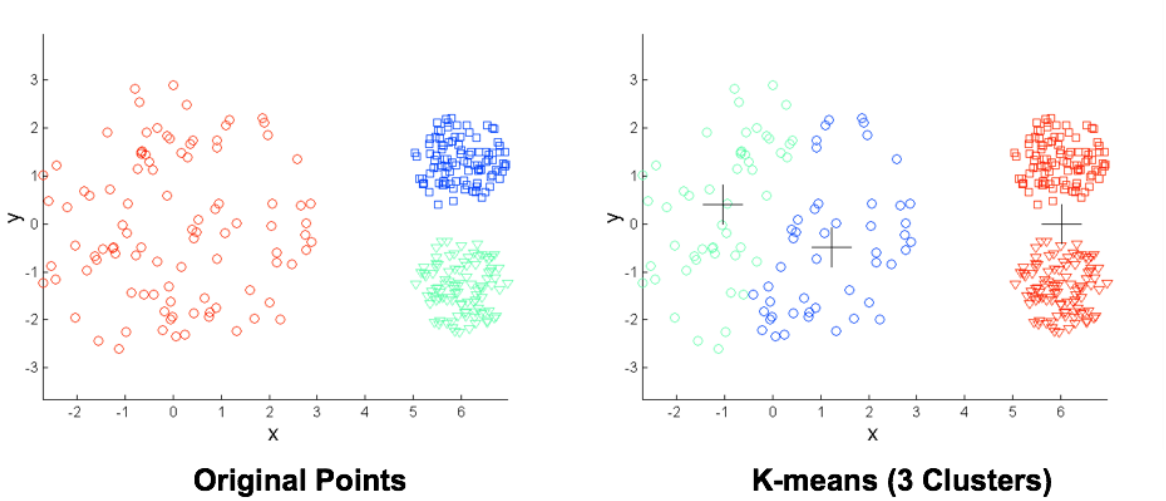
The above code should be quite useful for your assignment as well…



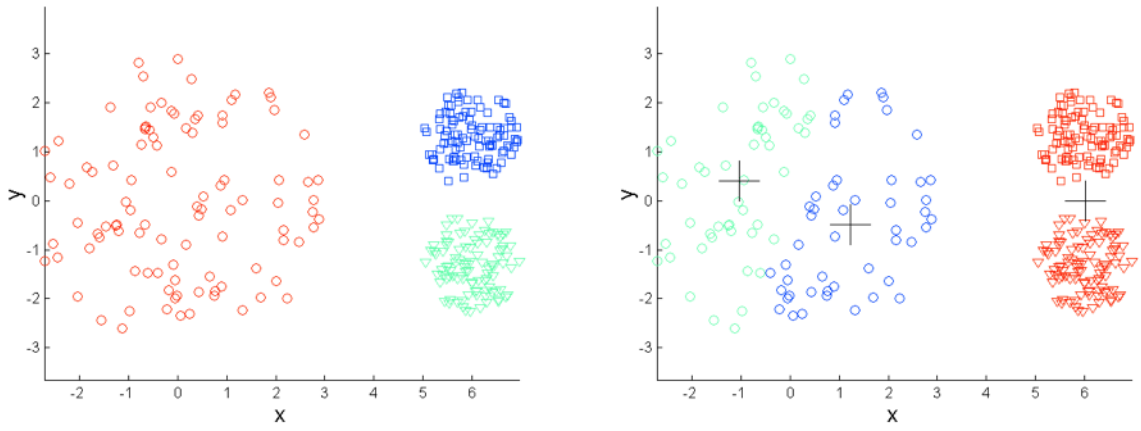
To the right is a sample of the output, based on address data:

While K-means is simple and effective, it has limitations that one needs to be aware of.

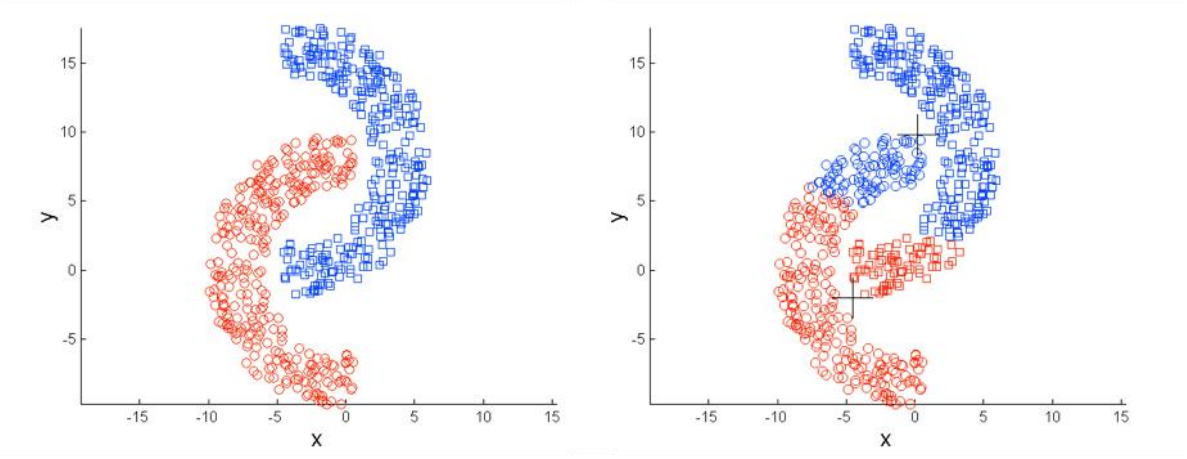
Differing sizes: K-means clustering assumes that cluster are roughly similarly sized, so if you have clusters where you expect variable sizes, K-means may not do so well in separating them. An example is shown below. The original points separate into three clusters of unequal size, but the clustered points on the right have a more even distribution.



Differing density: If the clusters have very different density in points, it can cause an issue for K-means clustering, which relies on the centroid of the points in-order to separate into clusters. The example below shows an instance where the original points (left) have one low-density cluster and two high-density clusters. This is clearly visible to the naked eye, but when K-means clusters points, it splits the low-density points to two clusters and combines the high-density clusters into one. This is a limitation of the use of the centroid of the points as the clustering metric.



Non-globular shapes: K-means clustering cannot deal with shapes that are irregular and skewed like the example below. This also stems from the fact that the centroid of a cluster is calculated as the mean over all the points in that cluster, which assumes a globular cluster shape.



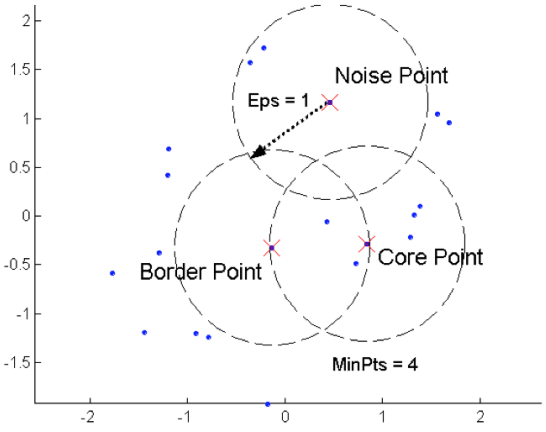
DBSCAN Clustering

Another approach to clustering is the use of DBSCAN, which does not assume that the cluster has a pre-defined shape (unlike k-means), but assumes that a cluster is a connected regions where the points are relatively dense.

DBSCAN requires two parameters: ε (eps) and the minimum number of points required to form a dense region (minPts). Given these parameters, points can be separated into three classes:

- A point is a *core point* if it has more than a specified number of points (minPts) within ε (these are points that are at the interior of a cluster)
- A *border point* has fewer than minPts within ε, but is in the neighborhood of a core point
- A *noise point* is any point that is not a core point or a border point.

On the right is an example with eps=1 and minPts=4. The core point (bottom right) has more than four neighbors; the border point (bottom left) has three neighbors, so it is not core, but one of the neighbors is a core point, so it is classified as a border point. The noise point on the top has two neighbors, but neither of these neighbors are core points, so it is classified a noise point.

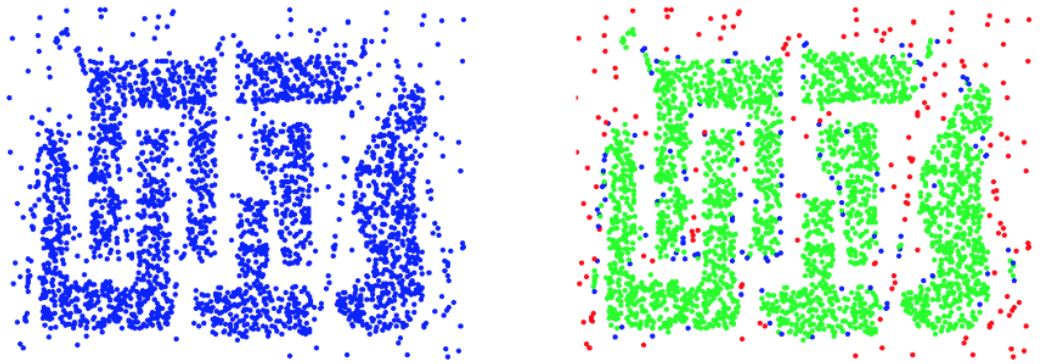


Once the points are divided into core, border and noise points, the DBScan algorithm first removes all the noise points since they are not needed for clustering. After this point, it performs clustering on the remaining points in an iterative manner by using the algorithm described below.

```
current_cluster_label ← 1
for all core points do
  if the core point has no cluster label then
    current_cluster_label ← current_cluster_label + 1
    Label the current core point with cluster label current_cluster_label
  end if
  for all points in the Eps-neighborhood, except i^th the point itself do
    if the point does not have a cluster label then
      Label the point with cluster label current_cluster_label
    end if
  end for
end for
```

The algorithm is incremental. First, you take a core point that is unlabeled and label it (say as cluster ID *i*). Then you look for all neighbors of this core point, and label them with the same cluster ID i. This iterates until you finally have no remaining points that need to be labeled.

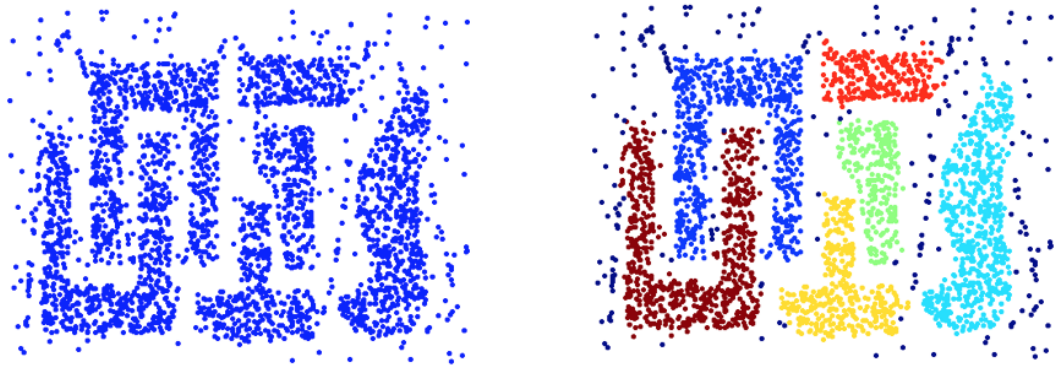
An example is shown below. On the left is the original points, and on the right is the points separated into core, border and noise points.



Original Points **Point types: core, border and noise**

Eps = 10, MinPts = 4

Once the points are clusters, you get the clusters shown on the right.



Original Points **Clusters**

For more details, look at the detailed description of DBSCAN on Wikipedia. The Wikipedia page also describes the algorithm in pseudocode, which is what you will use for your assignment (reproduced below).

```
DBSCAN(D, eps, MinPts) {
   C = 0
   for each point P in dataset D {
      if P is visited
         continue next point
      mark P as visited
      NeighborPts = regionQuery(P, eps)
      if sizeof(NeighborPts) < MinPts
         mark P as NOISE
      else {
         C = next cluster
         expandCluster(P, NeighborPts, C, eps, MinPts)
      }
   }
}

expandCluster(P, NeighborPts, C, eps, MinPts) {
   add P to cluster C
   for each point P' in NeighborPts {
      if P' is not visited {
         mark P' as visited
         NeighborPts' = regionQuery(P', eps)
         if sizeof(NeighborPts') >= MinPts
            NeighborPts = NeighborPts joined with NeighborPts'
      }
      if P' is not yet member of any cluster
         add P' to cluster C
   }
}

regionQuery(P, eps)
   return all points within P's eps-neighborhood (including P)
```

Phase III: Clusters to Semantic Locations

Lets now look at how to convert from clusters to semantic locations. This approach takes advantage of the fact that you have already reduced the data from a huge number of GPS locations to a small number of clusters (**k**).

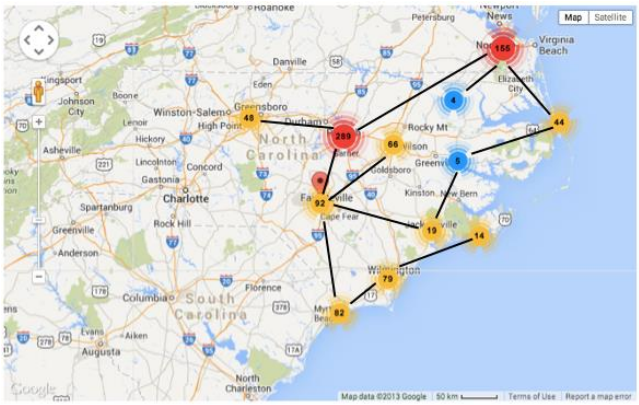
There are a few options to labeling the clusters with semantic location. One approach is to use reverse geo-coding on the cluster centroids, which refers to the process of going from GPS coordinates of the centroid to the name of the place (e.g. street address, place type, etc). There are many reverse geo-coding APIs that you can use to get this mapping (e.g. Google or Foursquare API).

But such APIs are typically limited to mapping from co-ordinates to address or eateries, and not so good for mapping to entities like gym or CS building, etc. These labels will need to be manually provided, which is not difficult given that you are only dealing with a small number of clusters.

Conclusion

In conclusion, we have given you some ideas on how to go about processing GPS location data that you might collect over many days. The techniques that we have suggested are first steps, and more tuning of either the pre-processing step or clustering step will need to be done to get it to work well for your data. But our goal was to get you to a point where you could start exploring on your own, and we hope to have accomplished this.

While our discussion is limited to GPS clustering, the problem of visualizing spatio-temporal data on health is broad and difficult to cover in a single chapter. One can imagine visualizing the data in many different ways to make it possible for a user to find useful insights from the data. For example, the figure below shows how you might show GPS clusters, and if a user clicked on a cluster, popup a window that shows a breakdown of social interactions with other individuals within that cluster. There are many such methods to explore on your own.



References

Discovering Personal Gazetteers: An Interactive Clustering Approach, GIS 2004.
K-means Clustering in MySQL - Joni Salonen
Personal Visualisation of Spatiotemporal and Social Data - Bo Stendal Sørensen