Extracting Patterns from Location History

Andrew Kirmse
Google Inc
Mountain View, California
akirmse@google.com

Tushar Udeshi
Google Inc
Boulder, Colorado
tudeshi@google.com

Pablo Bellver
Google Inc
Mountain View, California
pablob@google.com

Jim Shuma
Google Inc
Mountain View, California
jshuma@google.com

ABSTRACT

In this paper, we describe how a user's location history (recorded by tracking the user's mobile device location with his permission) is used to extract the user's location patterns. We describe how we compute the user's commonly visited places (including home and work), and commute patterns. The analysis is displayed on the Google Latitude history dashboard [7] which is only accessible to the user.

Categories and Subject Descriptors

D.0 [**General**]: Location based services.

General Terms

Algorithms.

Keywords

Location history analysis, commute analysis.

1. INTRODUCTION

Location-based services have been gaining in popularity. Most services[4,5] utilize a "check-in" model where a user takes some action on the phone to announce that he has reached a particular place. He can then advertise this to his friends and also to the business owner who might give him some loyalty points. Google Latitude [6] utilizes a more passive model. The mobile device periodically sends his location to a server which shares it with his registered friends. The user also has the option of opting into latitude location history. This allows Google to store the user's location history. This history is analyzed and displayed for the user on a dashboard [7].

A user's location history can be used to provide several useful services. We can cluster the points to determine where he frequents and how much time he spends at each place. We can determine the common routes the user drives on, for instance, his daily commute to work. This analysis can be used to provide useful services to the user. For instance, one can use real-time traffic services to alert the user when there is traffic on the route he is expected to take and suggest an alternate route.

We expect many more useful services to arise from location history. It is important to note that a user's location history is stored only if he explicitly opts into this feature. However, once signed in, he can get several useful services without any additional work on his part (like checking in).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGSPATIAL GIS '11, November 1-4, 2011. Chicago, IL, USA
Copyright © 2011 ACM ISBN 978-1-4503-1031-4/11/11...$10.00

2. PREVIOUS WORK

Much previous work assumes clean location data sampled at very high frequency. Ashbrook and Starner [2] cluster a user's significant locations from GPS traces by identifying locations where the GPS signal reappears after an absence of 10 minutes or longer. This approach is unable to identify important outdoor places and is also susceptible to spurious GPS signal loss (e.g. in urban canyons or when the recording device is off). In addition they use a Markov model to predict where the user is likely to go next from where he is. Liao, et al [11] attempt to segment a user's day into everyday activities such as "working", "sleeping" etc. using a hierarchical activity model. Both these papers obtain one GPS reading per second. This is impractical with today's mobile devices due to battery usage. Kang et al [10] use time-based clustering of locations obtained using a "Place Lab Client" to infer the user's important locations. The "Place lab client" infers locations by listening to RF-emissions from known wi-fi access points. This requires less power than GPS. However, their clustering algorithm assumes a continuous trace of one sample per second. Real-world data is not so reliable and often has missing and noisy data as illustrated in Section 3.2.

Ananthanarayanan et al [1] describe a technique to infer a user's driving route. They also match users having similar routes to suggest carpool partners. Liao et al [12] use a hierarchical Markov Model to infer a user's transportation patterns including different modes of transportation (e.g. bus, on foot, car etc.). Both these papers use clean regularly-sampled GPS traces as input.

3. LOCATION ANALYSIS

3.1 Input Data

For every user, we have a list of timestamped points. Each point has a geolocation (latitude and longitude), an accuracy radius and an input source: 17% of our data points are from GPS and these have an accuracy in the 10 meter range. Points derived from wifi signatures have an accuracy in the 100 meter range and represent 57% of our data. The remaining 26% of our points are derived

from cell tower triangulation and these have an accuracy in the 1000 meter range.

We have a test database of location history of around a thousand users. We used this database to generate the data in this paper.

3.2 Location Filtering

The raw geolocations reported from mobile devices may contain errors beyond the measurable uncertainties inherent in the collection method. Hardware or software bugs in the mobile device may induce spurious readings, or variations in signal strength or terrain may cause a phone to connect to a cell tower that is not the one physically closest to the device. Given a stream of input locations, we apply the following filters to account for these errors:

1. Reject any points that fall outside the boundaries of international time zones over land. While this discards some legitimate points over water (presumably collected via GPS), in practice it removes many more false readings.
2. Reject any points with timestamps before the known public launch of the collection software.
3. Identify cases of "jitter", where the reported location jumps to a distant point and soon returns. As shown in Figure 1, this is surprisingly common. We look for a sequence of consecutive locations $\{P_1, P_2, \ldots, P_n\}$ where the following conditions hold:
   - P_1 and P_n are within a small distance threshold D of each other.
   - P_1 and P_n have timestamps within a few hours of each other.
   - P_1 and P_n have high reported accuracy.
   - $P_2, \ldots, P_{n-1}$ have low reported accuracy.
   - $P_2, \ldots, P_{n-1}$ are farther than D from P_1.

   In such a case, we conclude that the points $P_2, \ldots, P_{n-1}$ are due to jitter, and discard them.
4. If a pair of consecutive points implies a non-physical velocity, reject the later one.

Any points that are filtered are discarded, and are not used in the remaining algorithms described in this paper.

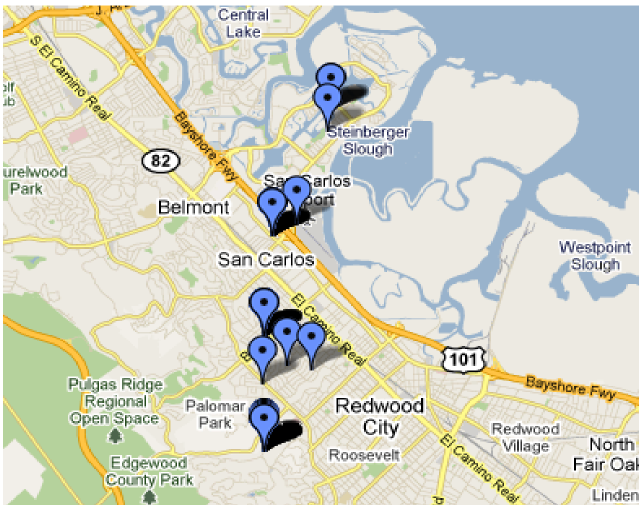

Figure 1. A set of reported locations exhibiting "jitter". One of the authors was actually stationary during the time interval represented by these points.

3.3 Computing Frequently Visited Places

In this section, we describe the algorithms we use to compute places frequented by a user from his location history. We first filter out the points for which the user is stationary i.e. moving at a very low velocity. These stationary points need to be clustered to extract interesting locations.

3.3.1 Clustering Stationary Points

We use two different algorithms for clustering stationary points.

3.3.1.1 Leader-based Clustering

For every point, we determine if it belongs to any of the already generated clusters by computing its distance to the cluster leader's location. If this is below a threshold radius, the point is added to the cluster. Psuedocode is in Figure 2. This algorithm is simple and efficient. It runs in O(NC) where N is the number of points and C is the number of clusters. However, the output clusters are dependent on the order of the input points. For example, consider 3 points P_1, P_2 and P_3 which lie on a straight line with a distance of radius between them as shown in Figure 3. If the input points are ordered $\{P_1, P_2, P_3\}$, we would get 2 clusters: $\{P_1\}$ and $\{P_2, P_3\}$. But if they are ordered $\{P_2, P_1, P_3\}$ we would get only 1 cluster containing all 3 points.

```
Let points = input points.
Let clusters = []
foreach p in points:
  foreach c in clusters:
    if distance(c.leader(), p) < radius:
       Add p to c
       break
    else:
       Create a new cluster c with p as leader
       clusters.add(c)
```

Figure 2. Leader-based Clustering Algorithm.

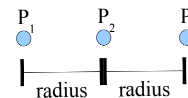

Figure 3. Three equidistant points on a line.

3.3.1.2 Mean Shift Clustering

Mean shift [3] is an iterative procedure that moves every point to the average of the data points in its neighborhood. The iterations are stopped when all points move less than a threshold. This algorithm is guaranteed to converge. We use a weighted average to compute the move position where the weights are inversely proportional to the accuracy. This causes the points to gravitate towards high accuracy points. Once the iterations converge, the moved locations of the points are chosen as cluster centers. All input points within a threshold radius to a cluster center are added to its cluster. We revert back to leader-based clustering if the iterations do not converge or some of the input points remain unclustered. Psuedocode is shown in Figure 4.

This algorithm does not suffer from the input-order dependency of leader-based clustering. For the input point set of Figure 3, it will always return a cluster comprising all 3 points. The algorithm generates a smaller number of better located clusters compared to leader-based clustering. For example consider 4 points on the vertices of a square with a diagonal of 2*radius as shown in Figure 5. Leader-based clustering would generate 4 clusters, 1 per

point. Mean-shift clustering would return only 1 cluster whose centroid is at the center of the square.

```
Let points = input points
Let clusters = []
foreach p in points:
  Compute Weight(p)
Let cluster_centers = points
while all cluster_centers move < shift_threshold:
  foreach p in cluster_centers:
    Find all points np which are within a threshold distance of p
```

$$p = \frac{\sum Weight(np_i) \times np_i}{\sum Weight(np_i)}$$

```
while not cluster_centers.empty():
  Choose p with highest accuracy from cluster_centers
  Find all points, say rp, in points which are within radius of p
  Create a new cluster c with rp
  clusters.add(c)
  points = points – rp
  cluster_centers = cluster_centers – Moved(rp)
Cluster remaining points with Leader-based clustering.
```

Figure 4. Mean-Shift Clustering Algorithm

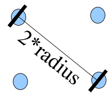

Figure 5. Four points on a square.

The iterative nature of this algorithm makes it expensive. We therefore limit the maximum number of iterations and revert to leader-based clustering if the algorithm does not converge quickly enough.

When we ran Mean-shift clustering on our test database, the algorithm converged in 2.4 iterations on an average. 3% of the input points could not be clustered (i.e. we had to revert to leader-based for them). However, it did not cause a significant reduction in the number of computed clusters (< 1%). We concluded that the marginal improvement in quality did not justify the increased computational cost.

3.3.1.3 Adaptive Radius Clustering

The two clustering algorithms described above return clusters of the input points. One possibility would be to deem clusters larger than a threshold as interesting locaitons. However, this is not ideal since the input points have varying accuracy. For instance, if there are three stationary GPS points within close proximity of each other, we have high confidence that the user visited that place as opposed to three stationary cell tower points. We run the clustering algorithms multiple times, increasing the radius as well as the minimum cluster size after every iteration. When a cluster is generated, we check to see if it overlaps an already computed cluster (generated from a smaller radius). If that is the case, we merge it into the larger cluster. Note that adaptive radius clustering can be used in conjunction with any clustering algorithm.

From our test database, we found that adaptively increasing the clustering radius from 20 meters to 500 meters and the minimum cluster size from 2 to 4, increased the number of computed visited places by 81% as compared to clustering with a fixed radius of 500 meters and a minimum cluster size of 4. We also surveyed users and found that the majority of the new visited places generated were correct and useful enough to display on the Latitude history dashboard [7].

3.3.2 Computing Home and Work Locations

We use a simple heuristic to determine the user's home and work locations. A user is likely to be at home at night. We filter out the user's points which occur at night and cluster them. The largest cluster is deemed the user's home location.

Similarly, work location is derived by clustering points which occur on weekdays in the middle of the day and clustering them.

Note that this heuristic will not work for users with non-standard schedules (e.g. work at night or work in multiple locations). Such users have the option of correcting their home and work location on the Latitude history dashboard [7]. These updated locations will be used for other analyses (e.g. commute analysis described in Section 3.4).

3.3.3 Computing Visited Places

We do some additional filtering of the input points before clustering for visited places:

1. We remove points which are within a threshold distance of home and work locations.
2. We remove points which are on the user's commute between home and work. These points are determined using the algorithm described in Section 3.4.1.
3. We remove points near airports since these are reported as flights as described in Section 3.5.

Without these filters, we get spurious visited places. Even when a user is stationary at home or work, the location reported can jump around, as described in Section 3.2. Without the first filter, we would get multiple visited places near home and work. If a user regularly stops at a long traffic signal on his commute to work, it has a good chance of being clustered to a visited place. This is why we need the second filter.

3.4 Commute Analysis

We can deduce a user's driving commute patterns from his location history. The main challenge here is that points are reported infrequently and we have to derive the path the user has taken in between these points. Also, the accuracy of the points can be very low and so one needs to snap the points to the road he is likely to be on. The commutes are analyzed in three steps: (1) Extract sets of commute points from the input. (2) Fit the commute points to a road path. (3) Cluster paths together spatially and (optionally) temporally to generate the most common commutes taken by the user.

3.4.1 Extracting Commute Points

Given a source and destination location (e.g. home and work), we extract the points from the user's location history which likely occurred on the user's driving commute from source to destination. We first filter out all points with low accuracy from the input set. We then find pairs of source-destination points. All points between a pair are candidate points for a single commute. The input points are noisy and therefore we do some sanity checks on the commute candidate points: (1) The commute distance should be reasonable. (2) The commute duration should be reasonable. (3) The commute should be at reasonable driving velocity.

3.4.2 *Fitting a Road Path to Commute Points*

We use the routing engine used to compute driving directions in Google Maps [8] to fit a path to the commute point. For the rest of this paper, we will refer to this routing engine as "Pathfinder". This is an iterative algorithm. We first query Pathfinder for the route between source and destination. If all the commute points are within the accuracy threshold distance (used in Section 3.4.1), we terminate and return this path. If not, we add the point which is furthest away from the path as a waypoint and query Pathfinder again. If Pathfinder fails, we assume that the waypoint is not valid (for example it might be in water) and drop it. We continue iterating until all commute points are within the accuracy distance threshold. Psuedocode is shown in Figure 6. Two iterations of this algorithm are shown in Figure 7. The small blue markers are the points from the user's history. The large green markers are the Pathfinder waypoints used to generate the path. After the second iteration all the user points are within the accuracy threshold.

```
Let points = input points
Let waypoints = [source, destination]
Let current_path = Pathfinder route for waypoints
while points.size() > 2:
  Let p be the point in points farthest away from current_path
  if distance(p, current_path) < threshold:
    Record current_path as a commute.
    break
  Add p to waypoints in the correct position
  current_path = Pathfinder route for waypoints
  if Pathfinder fails:
    Erase p from waypoints
  Erase p from points
```

Figure 6. Algorithm to fit a path to commute

3.4.3 *Clustering Commutes*

Given a bag of commute paths and the time intervals the commutes occurred, we cluster them to determine the most frequent commutes. Two commutes are deemed temporally close if they start and end within some threshold of each other on the same day of the week. Two commutes are deemed spatially close if their Hausdorff distance [9] is within a threshold. We use a variant of leader-based clustering (described in Section 3.3.1.1) to generate commute clusters.

The largest commute cluster on a particular day of week is the most common route taken by the user. This can be used for traffic alerts as described in Section 4.1.

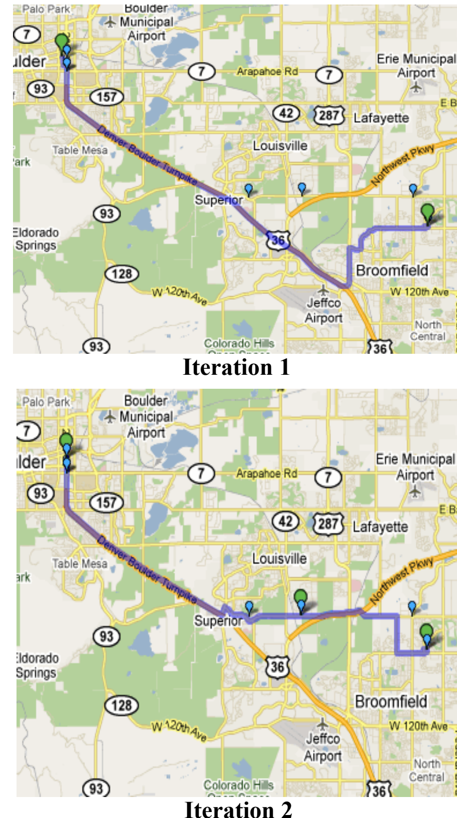

Figure 7. Two iterations of path fitting algorithm. The small markers are the user's points. The large marker are Pathfinder waypoints used to generate the path.

4. ACKNOWLEDGMENTS

Our thanks to Matthieu Devin, Max Braun, Jesse Rosenstock, Will Robinson, Dale Hawkins, Jim Guggemos, and Baris Gultekin for contributing to this project.

5. REFERENCES

[1] Ananthanarayanan, G., Haridasan , M., Mohomed , I., Terry, D., Thekkath , C. A. 2009. *StarTrack: A Framework for Enabling Track-Based Applications.* Proceedings of the 7th international conference on Mobile systems, applications, and services.

[2] Ashbrook, D., Starner, T. 2003. *Using GPS to Learn Significant Locations and Predict Movement across Multiple Users.* Personal and Ubiquitous Computing, Vol. 7, Issue 5.

[3] Cheng, Y. C. 1995. *Mean Shift, Mode Seeking, and Clustering.* IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 17, No. 8.

[4] Facebook places http://www.facebook.com/places.

[5] Foursquare http://www.foursquare.com.

[6] Google Latitude http://www.google.com/latitude.

[7] Google Latitude History Dashboard. http://www.google.com/latitude/history/dashboard

[8] Google Maps http://maps.google.com.

[9] Huttenlocher D. P., Klanderman, G. A., and Rucklidge W. J. 1993. *Comparing Images using the Hausdorff Distance.* IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 15, No 9.

[10] Kang, J.H., Welbourne, W., Stewart, B. and Borriello, G. 2005. *Extracting Places from Traces of Locations.* SIGMOBILE Mob. Comput. Commun. Rev. 9, 3.

[11] Liao, L., Fox, D., and Kautz, H. 2007. *Extracting Places and Activities from GPS traces using Hierarchical Conditional Random Fields.* International Journal of Robotics Research.

[12] Liao, L., Patterson D. J., Fox, D., and Kautz, H. 2007, *Learning and Inferring Transportation Routines.* Artificial Intelligence. Vol. 171, Issues 5-6.