GeoFriends

Ricardo Abreu
ricardo.abreu@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2017

Abstract

Location-based Social Networks are getting increasingly more popular due to the fact that nowadays it is extremely easy for someone to record and share his location. This type of geographical data can be used to link the physical and digital worlds and as such, location-based services can enhance the real world experience of their users. In this thesis we move towards this direction and aim to find similar users from their location traces, while also letting them interact with each other via geographical annotations. A system called GeoFriends is presented which main objective is to retrieve information from location data, and use it to recommend both users and nearby content. In addition to the traditional approach of comparing locations' coordinates, GeoFriends takes into account factors such as the sequence and the time of the day the users attended a given region. GeoFriends tackles some scalability issues such as how to deal with the possible constant location updates from its users. We evaluated such scenarios and found that GeoFriends scales much better than a straight forward approach. On the other hand, using a publicly available dataset, we found that GeoFriends was able to retrieve approximately 55% of any user's friends using only location data.
Keywords: Location-Based Social Networks; Content Sharing; Data Mining; User Similarity

1. Introduction

Cycle to shop initiatives such as TRACE [1] encourage people to ride their bicycles so they can earn virtual points which can later be traded for real life rewards (e.g., shop discounts, products). This way, people are encouraged to ride their bicycles more often instead of their cars. Consider an individual who likes to ride his bike but knows nobody to ride with. Based on his travel routes and places he visits, one should be able to suggest people who also attend these same regions so they can potentially become friends and ride together. This way they would both be able to meet new people as well as potentially earn more points together. Additionally, consider that some rider has a flat tire somewhere on his trips. He should be able to ask for help and get it from someone nearby, even without knowing or being friends with them.

The advances in localization techniques have enhanced traditional social networking services by enabling users to share their locations and location-related content. Nowadays there is a huge number of widely spread location-aware devices which produce very rich contextual information like a user's past locations (also called location history) [1]. Location data bridges the gap between physical and digital worlds, and as such, we aim to use it so we can suggest new friends and geo-related information to the user.

Friend recommendations have been extensively studied in the context of traditional social networks. The traditional approach relies on user interaction patterns (like visiting each others profiles), but in a location-based social network (LBSN) there is valuable context from which to extract better recommendations.

The goal of GeoFriends is to use location histories to recommend new friends and to exploit the user's current location to feed him relevant information nearby. We consider a user's location history to be the set of individual geographical points where the user shared his location (e.g., latitude-longitude), we call these points check-ins. We define an event to be the information some user shared in a given geographical region. In GeoFriends we aim to provide recommendations of both friends and events requiring minimal user intervention and relying only on the location data gathered by the system.

The contributions of this paper are the following:

- We design a system capable of recommending new friends and geo-related information based on the user's location history and current location, respectively;

[1] http://h2020-trace.eu/trace-tools/cycle-to-shop-initiative/

- We propose an approach to find and recommend similar users based on the locations they visited. We do so by considering the check-ins' information and additional factors such as the sequence, hierarchy and time of the day of the check ins;

- We recommend user-created geographically-dependent events in a scalable way in terms of communication and battery costs;

- We aim to exploit the benefits of having both users and events recommendations to improve each other. This way, GeoFriends can compare users whose location histories do not overlap.

- We evaluate both the quality of suggestions and the scalability of GeoFriends using a real world publicly available dataset from Gowalla

The rest of this paper is organized as follows. In Section 2 we describe existing relevant recommendation systems and compare some GPS mining approaches which aim to find similar users. In section 3 the architecture of GeoFriends is proposed while in Section 4 we define some implementation aspects. Finally, in Section 5 we describe how we evaluated our system, and in Section 7 we draw some conclusions.

2. Related Work

In this Section we analyze some existing relevant systems and compare them to GeoFriends based on the objectives and requisites we proposed. Additionally, we examine some GPS mining approaches to understand their advantages and limitations when trying to find similar users.

2.1. Recommendation Systems

Some systems allow users to create content and aim at suggesting such content taking into account the user's current location.

GeoFeed [2] is a location-aware news feed system that provides its users with spatially related messages from either their friends or favorite news sources. GeoNotes [4] is a location-based system that allows users to post and retrieve content associated with a given geographic position. Lastly, GeoPages [3] is a user content sharing system for a large-scale of both users and information. The main focus of this system is to minimize mobile communication and server processing costs. In terms of recommendations, GeoFriends must be able to:

- use past location data to provide friend recommendations;

- use current location data to provide event recommendations;

- suggest geographically-related events in a scalable way in terms of communication and processing costs.

None of the systems above mentioned fulfill both GeoFriends' main requisites, suggest new friends and allow the users to share content between themselves. There is a clear distinction between systems that aim to find similar users and systems which focus on sharing user content. Usually, systems that focus on the latter make use of the user's current position to suggest relevant items, while systems whose goal is to suggest new people make use of the user's location history. In GeoFriends we make use of both types of location data, the location histories and the users' current locations.

All the systems analyzed make use of location data to accomplish their objectives. However, they only consider individual locations disregarding aspects such as the sequence by which the locations were visited. This is an important aspect when comparing users because GeoFriends is incorporated in a cycle to shop initiative and as such we want to encourage people to ride together. More details about GeoFriends' approach to find similar users are given in Section 2.2.

Finally, GeoPages [3] is the only system which tackles the scalability issues of a context-aware recommendation system. In GeoFriends we will make use of the notions of capable zone, bounding box and residential domain presented in that system. Additionally, we will explore the notion of GeoHash when recommending nearby events so that we can reduce the search time when fetching the K closest events.

2.2. Mining GPS Data

One of the main goals of GeoFriends is to find similar users based on the locations they visit so they can ride together to earn some points. In this section we analyze some existing GPS mining approaches which aim to extract this knowledge from a set of users' locations. In addition to comparing coordinates GeoFriends takes into account factors such as the sequence of the locations shared and the travel time users spent traveling between regions.

Hierarchical-graph-based similarity measurement (HGSM) [5] is a user similarity framework that uses a user's visit to a given location as his implicit rating of that location. Our approach is very similar to this one because it takes into account not only the sequence by which the locations were visited but also their hierarchical property.

Xiao et al. take a different approach by modeling a user's trajectories with a semantic location history (SLH) [7], e.g., shopping malls, restaurants, cinemas - Figure 1 illustrates an example of this approach. Users sharing a longer sequence of seman-

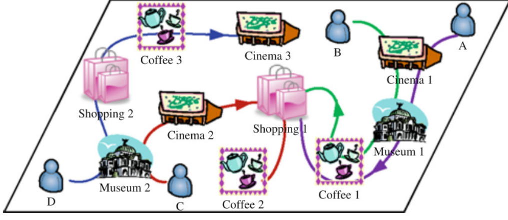

Figure 1: Example of Semantic Location Histories. Figure taken from Semantic Location History [7]

tic locations are more similar than those sharing a shorter one. For example, people sharing a sequence "museum - restaurant" would be more similar than those who visit these two categories separately.
In GeoFriends we calculate the similarity between users whose location histories do not overlap by mining the events he created. Additionally, in GeoFriends there is no external Database of Interest Points because the user similarity is calculated by comparing the locations' coordinates and taking into account additional factors such as their sequence.

GeoFriends uses the sequence of the positions shared as a way to find riders with similar trajectories instead of single point locations. Additionally, GeoFriends also consideres the time spent between locations to find users who, in addition to attending the same regions, also do so in the same time span.

From the users' location histories, GeoFriends must be able to:

- suggest friends considering the coordinates of the locations shared by the users but also additional factors such as the sequence, hierarchy and chronology of locations;
- suggest new friends even when the user arrives in a new region and does not share a meaningful amount of locations with the locals.

To better understand the similarity between users, different approaches have taken into account different factors besides the coordinates where users shared their location. For example, some techniques such as HGSM [5] take into account the intrinsic hierarchical property of locations to compare users on different scales. In GeoFriends we want to find similar people so they can become friends and ride their bikes together. Taking into account their trajectories, i.e. the sequence between locations, is crucial. This way, we take into account the sequence and time spent traveling between locations as well as the hierarchical property of the locations to infer what users are more similar to each other.

3. Architecture

GeoFriends is a system that recommends new friends to a user based on the locations he visited. Additionally, users can also create events. An event is constituted of three parts: its content (i.e., the text the user wants to share), the place where it was created (i.e., the events' coordinates) and a category (i.e., a keyword related to the events' content). For example, if a shop owner is having a sale, he can create an event with a category of "shopping", at his shop location, describing what products he is selling. We aim at using location data to provide recommendations of both friends and events.

3.1. Friend Recommendations

In GeoFriends we aim to find similar users based on location data, more specifically their check-ins and trajectories. Since the users locations can be in the scale of millions and have minor negligible differences it is not feasible to compare every GeoFriends' users' locations with each other. In GeoFriends, locations are in the latitude/longitude format and we chose to first cluster them in groups using KMeans. We use different values for the number of clusters (k) so that we have different levels of hierarchy in which to compare the users. We start by associating every user with a list of all his check ins which is then used to build a structure called *LocationInfo*. The user's *LocationInfo* holds all the information regarding his locations: the coordinates, the arrival and leaving times, and also the sequence between locations. This way, the user's *LocationInfo* is responsible to store the user's activity levels on the different clusters, and the sequences of locations visited. We assign each check-in to its closest cluster using the haversine formula to calculate the great-circle distance between two latitude-longitude points. When all the check ins have been mapped to their closest cluster, we calculate the percentage of user activity in each cluster. We also compute sequences of visited clusters so we find users who, in addition to visiting the same locations, also visit them by the same sequence. This way, a sequence is defined as a sublist of the user's visited clusters having two properties:

1. If cluster c1 and c2 belong to the same sequence, then c1 and c2 are clusters corresponding to check ins that happened in the same day;
2. There are, at least, 2 different clusters in the sequence;

Rule (1) is used so that we can capture the users daily routines, while rule (2) is used to filter out meaningless sequences. A sequence consisting of only one cluster can mean that the user is very active in that geographic zone, but that information is already available to us when we calculate the percentage of user activity on each cluster. This way, we are only interested in sequences that traverse

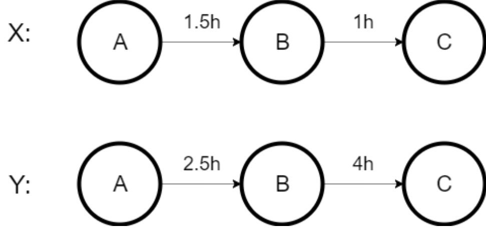

Figure 2: Example of two potential user sequences

more than one cluster and happened in the same day.

Besides finding users who attend the same regions (clusters) we are also interested in finding users with similar travel times between such regions. Consider the two hypothetical sequences described in Figure 2: a circle denotes the cluster visited while the caption above the arrows defines the time a user took traveling between clusters. The travel time is calculated using the time stamp associated with the user's check ins. As mentioned before, this information can be found on the user's *LocationInfo*. Both users go from cluster A to cluster B and finally to cluster C. However, user Y takes significantly more time to travel from cluster B to cluster C. We define a transition time threshold (*transTimeThresh*) that consists of the maximum acceptable difference in travel times, between the same clusters. If we set this threshold to 2 hours, we will get $< A, B >$ as the maximum similar sequence between users X and Y, since (2,5 - 1,5 $<$*transTimeThresh*) but (4 - 1 $>$*transTimeThresh*). For this example, if the threshold was set to be greater or equal to 3, we would get $< A, B, C >$ as the maximum similar sequence. To find similar sequences any two users might share, GeoFriends relies on a sequence matching algorithm that finds the maximum-length common sequences shared by the two users being compared. The implemented algorithm is an adaptation of the sequence matching algorithm presented in HGSM [5] and we now explain it in more detail. The intuition behind this algorithm is the following: given two cluster sequences Seq1 and Seq2, we start by finding the minimal common sequence shared by Seq1 and Seq2 (usually, a 1-length similar sequence is computed). Then, we try to expand it into a 2-length similar sequence and so forth. This process of extension is repeated until a maximum common sequence length threshold is reached or none of the common sequences can be extended any further - Algorithm 1. The bigger the current common subsequence is, the less likely it is for it to be extended, and as such, the maximum common sequence length threshold (*maxSeqThresh*) is responsible for limiting the maximum number of clusters in a possible common subsequence. This threshold is motivated by the fact that the authors of HGSM [5], found that instead of searching for all the similar sequences, finding sequences under a certain length can achieve comparable performance. This way, the *maxSeqThresh* is set to 5, as this was the value proposed in HGSM [5]. After reaching this threshold or not being able to extend any of the common sequences any further, GeoFriends has a set of common sequences from which it only keeps the maximum length ones.
The process of extending a sequence is described in Algorithm 2; it is the core of the sequence matching algorithm. A similar sequence is a group of clusters that appear in both original sequences. By using the clusters' sequences, arrival and leaving times of each user's *LocationInfo*, GeoFriends can match users who travel the same places on a similar time span.
In Figure 3 we have two original sequences from users X and Y and below them we have the common sequences computed after the first step of the extend sequence algorithm (thus only being a 1-length similar sequence). Each matched cluster has two indexes that represent the positions in which this cluster appeared in the two original sequences. For example, $A01$ describes a cluster that appeared in the first position in sequence X and the second position in sequence Y. These indexes help to differentiate the same cluster being visited in different points in time. Moreover, using these indexes we can infer the order by which the clusters were visited and so, extend the current similar sequence. This is what the *consequent()* function does: it checks if the cluster we are considering to extend the sequence happened in a point in time after the last cluster of the current similar sequence (line 4 of Algorithm 2). After checking if the clusters are consecutive in time, we can simply use the respective arrival and leaving times to check if the two clusters being considered satisfy the transition time threshold (lines 5 and 6). If they do, the sequence is extended from a k-length sequence, to a (k+1)-length similar sequence (line 7).
For example, in the second iteration of our algorithm, with a *transTimeThresh* of 3 hours, c1 (line 2) will take the value of $< A01 >$ and c2 (line 3) will be $< B12 >$. The consequent function will return true since both indexes of cluster $B12$ are bigger than the indexes of $A01$. Then, the difference in travel times is calculated, and since it is smaller than 3 hours, the common sequence is extended from $A01$ to $< A01, B12 >$. Using the same logic, GeoFriends also extends a common sequence when c1 and c2 take the values of $B12$ and $C23$, respectively; creating a $< B12, C23 >$ common sequence. When c1 is equal to $< A01 >$ and c2 to $< C20 >$ the consequent function will return false. This is be-

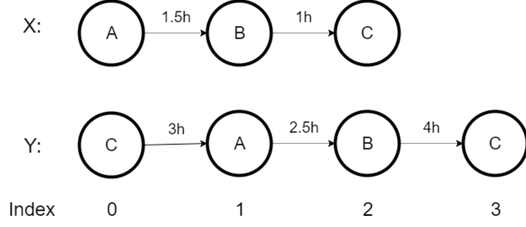

Figure 3: Extend Sequence algorithm after one iteration and the corresponding similar sequences found

cause the index in which the cluster $C20$ happened in sequence Y (index 0) is smaller than the current index in cluster $B12$ (index 2), and as such we must not consider it. This happens for every combination of c1 and c2 which has c2 equal to $C20$. There is only one combination of c1 and c2 values left to analyze: c1 = $A01$, and c2 = $C23$. Here the consequent function returns true but the difference in travel times is greater than the set threshold (6,5 - 2,5 >3). Thus, $< A01, B12 >$ and $< B12, C23 >$ are the only 2-length common sequences possible. In the third iteration both c1 and c2 will have the value of $< B12 >$ and as such, c2 will take the value of the second cluster (lines 4 and 5). Now c1 is equal to $< B12 >$ and c2 to $< C23 >$. Since the indexes are consecutive the difference in travel times is calculated. Once again, this difference respects the *transTimeThresh* and as such the $< A01, B12 >$ sequence is extended to $< A01, B12, C23 >$. In the next iteration it will not be possible to extend the sequence any further and as such, the maximum length common sequence between users X and Y is set to $< A01, B12, C23 >$.

Finally, GeoFriends can now assign a *similarityScore* to each user pair. This score is based on two factors: the users' activity levels on different clusters, and on the common sequences they may share. A user's activity level is a vector that represents the percentage of check ins he has in each cluster, for example, [0.7,0.2,0.1] for clusters 0, 1 and 2. Using this information we can assign an *activityScore* for the two users by computing the cosine similarity of their respective activity levels. GeoFriends uses the cosine similarity because it measures the cosine of the angle between any two vectors, in this case the two users' activity vectors. This way, if the users have exactly the same level of activity in all clusters their *activityScore* is maximized (i.e. is equal to 1); while a bigger difference in activity levels will result in a wider angle between the two vectors which in turn decreases the *activityScore*, as desired. For two activity levels x and y we have $activityScore = cos(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{y}||}$. The cosine similarity is calculated by the dot product of two numeric vectors, and it is normalized by the product of the vector lengths, so that output values close to 1 indicate high similarity. For example, if users X and Y have an activity level of [0.3,0.5,0.2] and [0.5,0.2,0.3], respectively, their *activityScore* is equal to 0.82.

When the process of extending common sequences ends, GeoFriends has a set of common sequences between the 2 users being considered. Then, by comparing the length of these resulting common sequences, GeoFriends only keeps the ones that have the maximum length. Using these sequences we define a *sequenceScore* as

$$sequenceScore = \sum_{1}^{n} (l * 2^l)$$

where l stands for the (maximum) length of the common sequence found and n is the number of maximum length common sequences found. The 2^l is a multiplying factor to give longer sequences more significance.

After calculating the activity and sequence scores GeoFriends can define the final *similarityScore* for the user pair. We can set different weights for the activity and sequence scores to achieve higher quality recommendations, for example: $similarityScore = 0,30 actScore + 0,70 seqScore$. In the end, we suggest users who have at least a *similarity score* of 0.5 with each other. In Section 5 we demonstrate how we experimented and evaluated GeoFriends with different *activityScore* and *sequenceScore* weights.

Algorithm 1: Sequence Matching Algorithm(SeqA,SeqB,maxSeqT, transTimeT)

```
sequenceSet =
  add1lengthSequences(SeqA,SeqB);
while step ≤ maxSeqT do
  | foreach Sequence seq in sequenceSet do
  |   extendSequence(sequenceSet,seq,transTimeT);
  | step++;
end
```

3.2. Event Recommendation

Users running the GeoFriends' client application are able to read any information (i.e., events) created by other users when they reach the region where that event was created.

When a client first connects to the server it computes a domain which is a geographic region that

Algorithm 2: Extend Sequence(sequenceSet, seq, transTimeT)

```
foreach Sequence aux in sequenceSet do
    c1 = seq.getLastCluster();
    c2 = aux.getFirstCluster();
    if c1.equals(c2) then
        c2 = aux.getSecondCluster();
    end
    if consequent(c1,c2) then
        delta =
         calculateDifferenceTravelTimes(c1,c2);
        if delta < transTimeT then
            seq.addCluster(c2);
        end
    end
end
```

contains its current position and nearby events. By checking its position against those of the nearby events received, the client can infer if it is near an event. This is useful because by doing so the client does not need to continuously contact the server every time a new location is available. The client is also responsible for monitoring its current (probably changing) position and contacting the server when it moves out of its assigned domain so it can calculate a new one.

To achieve this, we define two background processes: the location tracking and the events nearby processes.

GeoFriends' location tracking service listens for location updates via native sensors (e.g., GPS or WiFi providers). This service is responsible for detecting the user's check-ins and also for sending them to the database when an aggregation threshold is met - Algorithm 3.

Algorithm 3: Location Tracking Algorithm

```
lastKnownLocation = null;
pastLocations = [];

upon event < locationChanged(l) > do
lastKnownLocation = l;
pastLocations.add(l);
if pastLocations.size > threshold then
    sendLocationsToServer(pastLocations);
end
emit event
  < newLocation(lastKnownLocation) >
```

The events nearby process is responsible for fetching events from the database which are close to the user's current position. It uses the last known location from the location tracking process to center its search. Two extra user-defined factors are used: the furthest event and preferred workload thresholds. The first sets a limit on how far a suggested event can be, while the latter defines the preferred number of events a user wants to see near his location.

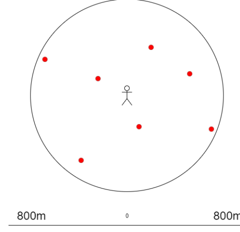

Figure 4: Fetching all events inside the furthest event threshold

To find the nearest events to a location, GeoFriends uses a geocoding system called Geohash which transforms a two-dimensional search into a one-dimensional one.

This way, GeoFriends' approach for the nearby events service is:

1. Transform the user's last known location into a GeoHash.

2. Calculate the maximum and minimum GeoHash values that an event nearby can take, considering the furthest event threshold.

3. Perform a ranged query starting and ending at the minimum and maximum Geohash values calculated, respectively.

Consider a scenario in which there are 7 events in a 800m radius around a 4-event workload user (Figure 4). In such a situation, knowing which events are inside the 800m radius is not enough because we want to be able to fetch the closer 4 events out of the 7. To do so, we must modify step (2) described above. A straight forward approach would be to simply take the first 4 events that the range query returns. This does not work because we are starting the range query at the minimum Geohash value instead of the users current Geohash. If there are enough events near that minimum, those will be the first to be returned while not necessarily being the closest to our user. To address this issue, instead of pulling every event which is inside the furthest event threshold once, we perform incrementally wider ranged queries around the user's location. Algorith 4 describes the process of fetching nearby events in GeoFriends. We start with a default minimum radius (line 5 in Algorithm 4) and keep incrementing it around the user's location until the preferred workload or the furthest event

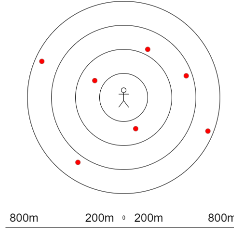

Figure 5: Incremental ranged queries to fetch closer events first

thresholds are reached (see Figure 5 and lines 12 to 14).

Once any threshold is reached and the client has the K nearest events (lines 16 and 17), we define a spatial region called residential domain. This region is a squared area, centered at the position where the client queried the database, which contains the K fetched events. The side of the square is defined in terms of the last ranged query's radius, since this was the query to reach some threshold. For instance, in the scenario described in Figure 5, the residential domain side would be 600m, the diameter of the last ranged query. The residential domain defines an area inside which the client does not need to continuously check for nearby events. The client is responsible for monitoring its current position (gathered from the location tracking service), and checking it against the residential domain's bounds (line 21). Once outside its bounds, the client must repeat the process of incremental ranged queries (line 22) to infer which events are now closer to its new location and as such compute a new residential domain.

4. Implementation

In this chapter we will describe the implementation decisions made to go from the architecture design and planning to the final GeoFriends system, with respect to the client and server applications. GeoFriends is built using Google's Firebase [2], more specifically its Realtime Database [3] and Authentication [4] features. The authentication feature allows users to log in using their existing accounts from popular sites (such as Google and Facebook). The Realtime Database provides storage, access and synchronization of the data created in our system. It is a cloud-hosted database where the data is stored as JSON and directly accessible by client applications - eliminating the need for an intermediate server which parses and relays the client's requests.

Since it is necessary to perform expensive and possibly long computations to infer which users are more

Algorithm 4: Events Nearby Algorithm

```
workload = getPreferredWorkload();
furthest = getFurthestThreshold();
nearbyEvents = [];
resiDomain = [];
radius = MIN_RADIUS;
monitoring = false;
loc = LocationTrackingProcess.getLastKnowLocation();

getEventsFromServer(loc,radius);

upon event < newEventFromServer(e) > do
nearbyEvents.add(e);
if nearbyEvents.size < workload && radius < furthest then
    radius = radius * k;
    getEventsFromServer(loc,radius);
else
    resiDomain = calculateResiDomain(radius);
    monitoring = true;
end

upon event < newLocation(loc) > do
if monitoring && isOutsideResiDomain(loc) then
    getEventsFromServer(loc,MIN_RADIUS);
end
```

similar to each other, we still need a server process to handle this intensive work outside the client application. When this process finishes its job it communicates with the clients by updating the database with its results, in this case, the suggested friends for each user. The client application is built using Android since most modern smart phones support wireless communication and location inference. The database provides a REST API which makes the server application language independent. All we need to do is append .json to the end of the database endpoint URL and send a request from any HTTPS client. This way, the Firebase Realtime Database sits between the server and the clients - Figure 6.

4.1. Client

As mentioned above, the client application is implemented using the Android framework. Firebase provides an Android SDK with helper functions to interact with their services (e.g., their realtime database) [5]. This way, when a client needs to access any information stored in the database, it does not need to contact the server. Instead, the client makes use of the Firebase's SDK functions to

[2] https://firebase.google.com/
[3] https://firebase.google.com/docs/database/
[4] https://firebase.google.com/docs/auth/
[5] https://firebase.google.com/docs/android/setup

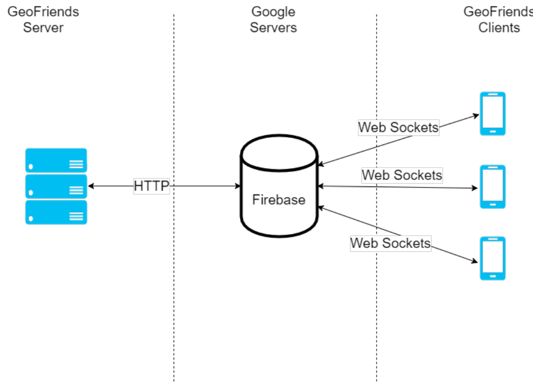

Figure 6: GeoFriends' client and server applications and their relation with Firebase. Figure taken from Firebase Blog [6]

directly read, write and delete data in the database. Thus, all the logic to recommend nearby events can run exclusively on the android application without any client-server communication. We created two background services: the location tracking service and the events nearby service. These two services are directly mapped to the processes presented in Section 3.
Firebase's Realtime Database supports a type of query that retrieves all data where some value is between an upper and lower boundary. These are called ranged queries, and are available in the form of "startAt" and "endAt" primitives. These primitives do prefix and sufix matching before returning the data to the client, and in conjunction with Geohashes help us retrieve the correct nearby events. As demonstrated in algorithm 4, GeoFriends fetches the events from the server based on a location and a radius (getEventsFromServer function). Using the location and these sufix and prefix matching functions, we can translate this call into a ranged query like so: $('locations').startAt(minHash).endAt(maxHash)$. This query will give us every location that matches the $startAt$ and $endAt$ criteria. The $minHash$ and $maxHash$ values are computed in terms of the radius and location received.

4.2. Server

All the logic of comparing users' movement patterns and suggesting new friends takes place at the server. Additionally, the server process is also responsible for extra auxiliar work flows such as data cleaning and clustering.
As mentioned before, the server process is implemented in Java mainly due to language familiarity but also due to the fact that there are helper libraries such as SPMF [6] and firebase4j [7] which provide very useful functionality for our system. The first is an open source data mining library from which we use the KMeans clustering algorithm implementation. The latter, is an open source client library written in Java which helps interacting with the Firebase's REST API. The clustering algorithm is used when extracting locations and sequences for each user (see Section 3). The firebase4j library is used any time we want to read or write data to the Database - e.g. insert the list of suggested friends for the users.

[6] http://www.philippe-fournier-viger.com/spmf/
[7] https://github.com/bane73/firebase4j

5. Evaluation

In this Section we show how we are going to evaluate GeoFriends. First, we want to evaluate GeoFriends' capability of recommending new friends to the user based on the locations he visited. This way, we will use a publicly available dataset of user's check-ins and friends from Gowalla to simulate a group of users' moving patterns and friends, respectively. By doing so, we aim to evaluate the feasibility of recommending new friends to a user based on location data. More details about the friend recommendation evaluation are given in Section 6.
Additionally, we want to evaluate the scalability of our system in terms of communication costs. To do this, we calculate the amount of bytes spent while varying the number of clients and their degree of activity.

6. Friend Recommendation

In order to analyze the quality of our friends suggestions we define the notion of relevant friend. A user U1 is considered a relevant friend of user U2 if there is a connection between U1 and U2 in Gowallas friends' dataset. This way, by using both the check ins as well as the friends datasets, we can evaluate the effectiveness of suggesting new friends based on location histories.

We start by calculating the overall precision and recall of our friend recommendations:

- Precision is the ratio of the number of relevant friends suggested relative to the total number of friends suggested;
- Recall is the ratio of the number of relevant friends suggested relative to the total number of relevant friends

In other words, for a number of suggested friends S and a number of relevant friends R, precision tells us the percentage of the relevant friends R that are in S, while recall shows the percentage of relevant friends who were suggested. These two metrics are used to infer the feasibility of our system and they both depend on the similarity score assigned to each user pair. As mentioned in Section 3, the final similarity score of each user pair depends on two

partial scores, the activity score and the sequence score. This way, we will evaluate the previously mentioned metrics while varying the weight of each partial score - Figure 7.

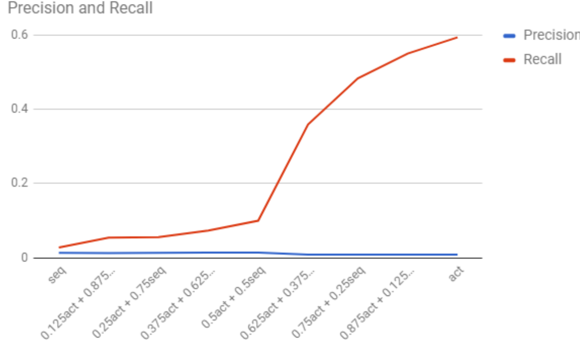

Figure 7: Users Recommendation Precision and Recall while varying the weights of the activity and sequence scores

We start by just considering the *sequenceScore* and then, gradually increase the weight of the *activityScore* by a factor of 0,125. E.g. $simScore = seqScore$; $simScore = 0,125actScore + 0,875seqScore$; $simScore = 0,25actScore + 0,75seqScore$, etc.. By doing this, we hope to tune the system in order to achieve the best balance between the two partial scores. In GeoFriends we want to find new people for a user to ride his bike with, and as such, a low *precision* score (i.e., suggesting many non-relevant friends) is not a problem. With respect to *recall*, a score closer to 1.0 indicates that GeoFriends was able to find many relevant friends by using only location data. This way, in GeoFriends, we are more interested in maximizing the *recall* score than in maximizing *precision*.
In Figure 7 we can see that around the $0.5act + 0.5seq$ mark there is an increase in the overall *recall* of relevant friends. This happens because with this configuration of the final score, there is an increase of recommended users by GeoFriends; thus, more non-relevant and relevant friends are suggested. Although suggesting more non-relevant friends hurts the overall *precision* of our system, the gain in overall *recall* is bigger. The *precision* score goes from 0.0134 to 0.008, while the *recall* increases from 0.0995 to 0.3587. As previously mentioned, maximizing the *recall* score is more important than maximizing precision, since we want to infer the best partial scores' weights for GeoFriends to find real life friends using only location data.

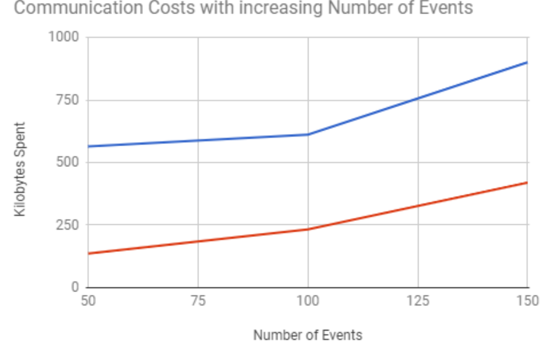

Figure 8: Scalability evaluation with respect to number of events and bytes spent

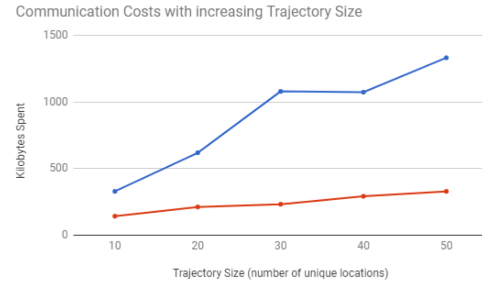

Figure 9: Scalability evaluation with respect to trajectory size and bytes spent

6.1. Event Recommendation

In this Section we evaluate GeoFriends' ability to scale with a growing number of events and with different levels of user activity. In this context, the user activity is directly related to his trajectory, for example, a user with a trajectory consisting of 20 check-ins has a higher level of activity than one with a trajectory size of 5. We evaluate GeoFriends by analyzing communication costs, in the form of bytes sent/received, incurring from users' location updates. We compare our approach with a straight forward one where clients query the database every time a new location is available, to check if there are any new events of interest nearby. This way, we perform the following experiments:

- measure the average bytes received/spent while varying the clients' trajectory size (Figure 9).
- measure the average bytes received/sent while varying the number of events (Figure 8);

Figure 9 illustrates the scalability evaluation with respect to the user's trajectory size. In this experiment, we define 5 degrees of user activity (i.e. a user having a trajectory consisting of 10, 20, 30, 40 and 50 check-ins) to simulate an increasingly more active user. We performed 10 measures for each trajectory size to avoid bias on a singular route.

We observed that after the 10th measurement for the same trajectory size the variation in bytes exchanged was minimal. The plotted value for each trajectory size is the mean of the 10 measurements. As we can see in Figure 9 GeoFriends outperforms the straight-forward approach as expected. Each time the user moves, i.e., has a new check-in, the straight forward approach contacts the server while GeoFriends only does so if the client crosses his assigned domain. We can also observe that in GeoFriends the downloaded and uploaded bytes stay relatively stable in contrast with the straight forward approach where they increase proportionally to the trajectory size. Again, this is due to the fact that GeoFriends does not need to perform a wireless call every time a new check in is available. This way, the bytes downloaded consists mainly of the events' data while the uploaded bytes are directly related to the number of calls the system makes to the database. Figure 8 also demonstrates the better scalability of GeoFriends, this time in terms of an increasing number of events. As previously mentioned, we defined 3 event's loads (50, 100 and 150 events) to simulate a low, medium and high event-density in the test area. When the test area was populated with 50 events, GeoFriends spent 76% less in communication costs. At 100 events 62% less and at 150 events 53% less. The higher the number of events the more likely it is for a GeoFriends' user to cross his domain's boundaries and thus, more server calls are performed. This explains why the amount of bytes exchanged increases with the increase in the number of events. As in Figure 9 the uploaded bytes are related mainly to the processing costs of calling the database, while the downloaded bytes correspond to the events' data.

7. Conclusions

In this paper, we started by describing a real world scenario where location data could be used to enhance everyday life, more specifically in a cycle-to-shop environment. It is possible to suggest new friends to a user by finding people who visit the same locations he does, this way, people can make new connections with individuals nearby. We described some challenges incurring from the use of location data in recommendation systems and presented GeoFriends. In Section 2, we analyzed and discussed some existing recommendation systems and GPS mining techniques, taking into account the objectives we proposed. We concluded that there is a distinct separation between systems that recommend friends and systems that recommend user content.

GeoFriends is a system capable of recommending new friends while allowing users to share geo-related content, thus, we use location data to support both of these types of recommendations. To find similar users GeoFriends considers the percentage of check ins each user had in each previously defined region, and additionally, it uses a sequence matching algorithm which aims to find the maximum length common sequence shared by the users being considered. This approach made possible for GeoFriends to find approximately 55% of any user's friends using only location data - Section 6. On the other hand, the notion of residential domain and incremental ranged queries helped GeoFriends achieve better scalability when dealing with nearby events' recommendations - Section 6.1.

Overall GeoFriends is unique in its ability to suggest both users and content using only location data while simultaneously offering good scalability compromises in terms of communication costs and battery usage.

References

[1] Bao, Jie, Zheng, Yu, Wilkie, David, Mokbel, and Mohamed. Recommendations in location-based social networks: a survey. *GeoInformatica*, 19(3):525–565, 2015.

[2] J. Bao, M. F. Mokbel, and C.-y. Chow. GeoFeed: A Location-Aware News Feed System. *Proceedings of the 28th IEEE International Conference on Data Engineering*, pages 54–65, 2012.

[3] Y. Cai and T. Xu. Design, analysis, and implementation of a large-scale real-time location-based information sharing system. *Proceeding of the 6th international conference on Mobile systems, applications, and services - MobiSys '08*, page 106, 2008.

[4] F. Espinoza, P. Persson, A. Sandin, H. Nyström, E. Cacciatore, and M. Bylund. GeoNotes: social and navigational aspects of location-based information systems. *3rd International Conference on Ubiquitous Computing (UbiComp)*, pages 2–17, 2001.

[5] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-y. Ma. Mining User Similarity Based on Location History. (c), 2008.

[6] A. Narayanan. https://firebase.googleblog.com/2013/03/where-does-firebase-fit-in-your-app.html. 2013.

[7] X. Xiao, Y. Zheng, Q. Luo, and X. Xie. Inferring social ties between users with human location history. *Journal of Ambient Intelligence and Humanized Computing*, 5(1), 2014.