Final Report: Distributed Lifestyle Management System

Anurag Dubey Ishan Shah Tanay Potnis Virag Doshi
Reed Anderson Yi Hou

May 1, 2017

Abstract

Thousands of people who are suffering from chronic health conditions need access to support from others who're facing the same condition. We propose a distributed application to motivate people with chronic health conditions to log their health data. Our application anonymizes this data and uses it to match people with similar conditions together. Our design can utilize a Hybrid P2P [Yan01] approach to enable the peers to discover each other through data correlation techniques. These users can share their health statistics for anonymous comparisons. We also allow them to form chats and groups to allow messaging or group broadcasts.

1 Problem Definition

Autoimmune diseases are a class of disease where in the immune system attacks healthy body tissue instead of external bacteria and viruses. These type of diseases are typically not curable and patients have to depend on immune suppressing drugs to manage their symptoms indefinitely. According to the American Autoimmune Related Disease Association (AARDA), approximately 1 in 5 people suffer from some type of autoimmune disease. The major hurdle to living with autoimmune disease is that the quantity and frequency of steroidal and immune-suppressing drugs often has to be tailored to each individual. In addition, a host of lifestyle changes are also needed to manage daily symptoms which again are not standardized by any national or international medical agency. Non-tangible factors like diet, stress management and even sleep quality can make a difference in alleviating symptoms. In view of these major hurdles, most patients are left to experiment by trial and error as to what course of treatment and action works best. To support such endeavors, due to the advent of social media, a number of support groups and communities have come up on online platforms that connect patients with similar health conditions to share personal experiences and advice. However, often this information shared is anecdotal and typically not targeted. We would like to come up with a solution that enables the same experience but in a more targeted way through matching of users based on similarity and correlation of health parameters. For our analysis, we chose health parameters related to autoimmune thyroid and adrenal disease. Namely, Hypothyroidism (Hashimoto's disease) and Adrenal Insufficiency. This is because both diseases have been known to have a correlation to similar dietary, stress and nutritional deficiencies and are often correlated with each other. Our system collect characteristic values from a user and store and share them in a distributed fashion. System will keep a reliable and fault tolerant data pool that will be used for finding correlation between datasets collected from multiple users. Nearest neighbor algorithm will be used to find users with highest correlation and then that will be used to produce a list of such users. This list will be used to discover and talk to other users.

2 Related Work

There are several applications designed specifically for people to share fitness data. Platforms such as fitbit's and Google fit's provide users a way to share their daily data and fitness achievements with their friends and have a healthy goal oriented competition or ranking system. But these are centralized approaches that only limit sharing to a minimum. These approaches don't provide anonymity either. The approach that we have taken would provide users the ability to share data anonymously with a significantly larger user-pool and get better feedback and comparisons due to the data-processing on the user side. There are other platforms which allow users to get feedback from trainers and get customized advice however, these options are expensive. While the application we are brainstorming would allow users with similar profiles to be matched anonymously and share insight and other advice they may have for a better lifestyle.

2.1 Mobile Fitness Applications

There are several applications designed specifically for people to share fitness data. Platforms such as Fitbit and Google Fit provide users a way to share their daily data and fitness achievements with their friends and have a healthy goal oriented competition or ranking system. But these are centralized approaches that only limit sharing to a minimum. These approaches don't provide anonymity either. The approach that we have taken would provide users the ability to share data anonymously with a significantly larger user-pool and get better feedback and comparisons due to the data-processing on the user side. There are other platforms which allow users to get feedback from trainers and get customized advice however, these options are expensive. While the application we are brainstorming would allow users with similar profiles to be matched anonymously and share insight and other advice they may have for a better lifestyle.

2.2 Anonymous chat

For anonumous chatting between clients, a few P2P communication methodologies were explored such as such as [gnu] and [Tor]. However due to the unreliability of the methodologies used for bypassing NATs and firewalls such as UDP hole punching, STUN [STU], TURN [TUR] and ICE [ICE] servers, and the fact that all of these methods need the users to be aware of the IP address of the other users, a centralized approach to the chat client was used. Redis [Red] was used to implement the chat due to its ease of use, low latency and support for various data structures.

2.3 Feature Vector Matching

Intuitively, users with similar features will be closer to each other if they were visualized in a Euclidean plane; similar users will cluster together. Our matching of users is therefore inspired by the very simple K-nearest neighbors (KNN) classification method, where Euclidean distance between continuous variable feature vectors is calculated over a multidimensional feature space.[NK16] Instead of classifying users, we see how much they cluster and connect users by nearest distance within the user matrix. We stored the multidimensional feature space with the Hadoop Filesystem (HDFS). This data was accessible for processing distances by Apache Spark and Apache Storm in implementation. Feature scaling was used at the server, before comparison of distance, for the continuous variables representing health values.[KR09]

3 Design Description

The distributed lifestyle management system is made up of three parts, client apps, index replication system and data pool. Client apps have a control of personal health information update, and can send requests to index replication system to seek groups to join in. Index system replicates all users' index value on each member in the distributed network and provide fast response for clients' requests. The data pool is to store users' various documents. Whenever a user modifies his information, the client will send a request to the index replication system which will put updates in the data pool.

3.1 Index Replication System

Each registered user has an index value, The index does not store the user's real information, which is just similar to a card catalog in a library. The card catalog contains metadata about the book and a location where the book can be found. Similarly, the index value consists of user's characteristic matrix, IP address, and meta data of user's information in the data pool. When a user is the first time to register in lifestyle management system, the client will create an index value for this user and send registration request to the index replication system. When the server receives a registration request, it will replicated this entry in the other members in the distributed

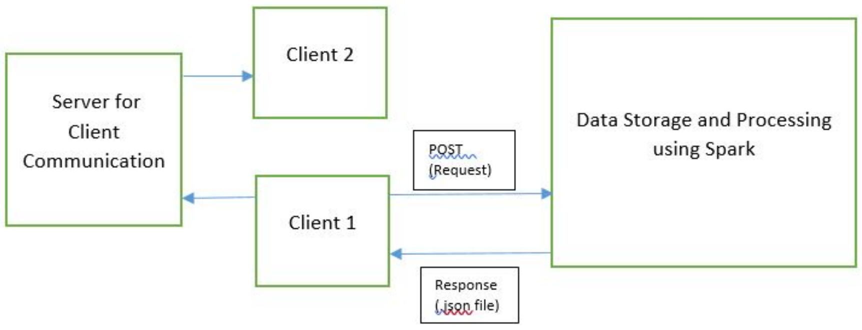
Server for Client Communication
Client 2
Client 1
POST (Request)
Response (.json file)
Data Storage and Processing using Spark

Figure 1: Overall architecture.

network. So each server in the index replication system will have the same copy of all users' index values. If a user does some updates or modifications on his or her health data, the update request is sent to one of members in the index replication system. Then the update request is forwarded to and executed on the data pool. When the execution is completed on the data pool, it sends back responses to update user's index value, which is also replicated on all members in the index replication system. Another case is when a user moves to a different place and changed IP address, the user needs to update his or her new IP address to the index value.

3.2 Data Pool

The data pool, or multidimensional feature space, is the place where users store their health information. The data pool is also distributed to provide reliability and scalability. But it is different from the index replication system, because all health records of each user is not required to replicate in the distributed system. Replicating all health data to every member in the distributed system would be bandwidth intensive, wasteful on network resources and pose data throughput concerns. So all users' data would be stored off the index replication system in the data pool. This makes data pool highly scalable and be able to store a wide variety of data, from images to documents. Data from cheap mobile devices and wearable sensors is growing at an exponential rate. The distributed architecture of data pool provides cost efficient high scalability. As more health data is added, the cost efficient commodity hardware can be easily added to handle the increased load. For reliability in addition to scalability, we implemented the distributed file system on HDFS, which provides built-in fault tolerance and disaster recovery.

3.3 Data Pool Processing

We ultimately choose Apache Spark Streaming for data processing. Apache Spark is a cluster computing framework using immutable resilient distributed datasets (RDDs) using shared memory over a distributed cluster. Spark proved to meet the needs for a system that: computed a one user against all for batch processing comparison against a potentially large data pool, preformed the same calculations for each query, and awaited streaming data after initialization for comparison against the HDFS stored matrix. Streaming data was sent to the running Spark instance as a TCP stream.

Our Spark algorithm compared an updated RDD of the matrix M joined with a streaming RDD representing the querying user's vector. The calculations performed used the Pythagorean formula:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

Where p is the user's vector, and q is one vector m in matrix M. The map phase used the index i as a key for p_i and q_i values to send to a reduce phase, where the values were subtracted and squared. The resulting value was then mapped by key q to a reduce function that sums all values and takes the square root. These distances between vector m and the target p were finally reduced to find the k neighbors with minimum distance.

Additionally, we first used Apache Storm to stream process queries against the data pool, specifically for user matching. Apache Storm uses a graph structure to separate map and reduce logic for distributed, stream processing in graph nodes. Ultimately, Spark Streaming had much higher performance than Storm, which we discuss in our implementation results. Our Storm topography is below:

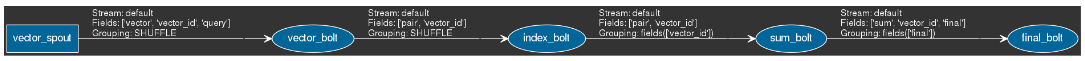
vector_spout
Stream: default
Fields: ['vector', 'vector_id', 'query']
Grouping: SHUFFLE
vector_bolt
Stream: default
Fields: ['pair', 'vector_id']
Grouping: SHUFFLE
index_bolt
Stream: default
Fields: ['pair', 'vector_id']
Grouping: fields(['vector_id'])
sum_bolt
Stream: default
Fields: ['sum', 'vector_id', 'final']
Grouping: fields(['final'])
final_bolt

Figure 2: Apache Storm Topology.

3.4 Access Control

The user has full access to his or her data and control over how his or her data is shared. When the user becomes a member of chat group, the user assigns a set of access permissions to the other group members. After a group member is granted access to a user's health information, he or she can query the index replication system to fetch the desired data from the data pool. At any given time the user may alter the set of permissions. This allows the user to make all decisions about what data is collected and how the data can be shared. Access control permissions provide each with an environment of transparency and hide private information from untrusted visitors.

3.5 Chat Protocol

Each user will be provided with a unique ID number that will be used to identify the user. The user has the option to any user as long as the user's user ID us known. The client maintains a chat history of all the user chats on the local machine.
The chat system utilizes a central approach. The chat server is implemented using a distributed queue on server side. The database server maintains a queue for each user. Whenever a new message is sent to that user, the sender's user ID is pushed onto the queue along with the message. To receive the message, the item has to be popped from the queue. This way, a simple queue data structure is utilized to maintain order between messages.

3.5.1 Sending messages

Sends are done by performing a HTTP POST to the server with the sender's and receiver's user IDs along with the message. In addition to the HTTP POST, the client also appends the message to the chat log. In response to the HTTP POST, the server pushes the message to the queue of the receiver, and returns the status.

3.5.2 Receiving messages

The client runs a receive thread that polls the server for any new messages. Receives are done by performing a HTTP GET request to the server with the receiver's user ID. In response server pops all the items in the receiver's queue and returns them to the client in a JSON format. The client then separates the messages according to the sender and adds them to their respective logs.

4 Implementation

Our project continued to follow the outline for a client application, index replication system, and data pool. For the Index Replication System, we created a single server, multi-client implementation with communication happening in form of HTTP requests and responses. Continuing to the data pool, the server and Spark master ran on the same machine, allowing the server to directly message Spark with a local TCP port and for the server to quickly retrieve output from the Spark cluster in the form of a text file within the local file system. An extension to this design would have a separate Spark master machine receive messages in a queue from the server, and a separate server retrieving output from HDFS. Finally, our client allows users to interface with the server and communicate with other users.

This code is available on github and can be accessed online [Col].

4.1 Client

Client program runs on every user's device that is using our system. Every client is provided a client ID which is used to identify the client. We used Python to implement the client. In the initial setup, our code uses default values for other client parameters like user health statistics and personal data.

The client implementation collects and quantifies the following parameters from its user:

(1) Age

(2) Gender

(3) Avg Heart Rate (Daily)

(4) Exercise Hours (Daily)

(5) Sleep Hours (Daily)

(6) Salt Level

(7) Water Intake (Daily in Liters)

(8) Disease Type (Types mapped to numbers for comparison)

(9) Disease Duration (Months)

(10) Other Data

This data is converted into a vector in the client object. This completes initialization. Then the client sends a *register* request to the server along with its ID. This is followed by login where the client application provides credentials it used at the registration time. This identifies the client. Now client sends its local vector to server in another request and the server returns back with a list of closest user IDs. The client can now initiate chat with any of the users in this list. The chat protocol uses central server in our initial code because of NAT limitations in p2p networks.

4.2 Server

The server was also running on Python and used Flask framework for generating communication endpoints. The central server is responsible for maintaining all metadata that is needed to process client requests. For now, all the server's data is stored in local memory.
We used redis database as a messaging store as it supports queue and distributed modes of operation. After the server is contacted by a client with a "match" request, it launches a real-time compute job.

The job is responsible for running Euclidean distance algorithm on its dataset to figure out user IDs which are *closest* to client's vector in terms of vector values. This is done by launching a Spark job with a TCP request containing the client's vector and indentifying index as input. The server returns a JSON to the client containing IP addresses and IDs of the other clients it thinks are most similar to the client. The client uses that list to discover and talk to other peers.

To ensure security, we make sure that every client is registered and logged into the server before it is allowed to perform any updates or send its data. The User ID is Anonymised and converted into a 10 character ID which is sent to the server. This means that at the server level, the data is handled and tagged in terms of anonymous strings. We plan to encrypt all the connections in future so that MITM attacks cannot happen.

Table 1: Comparison of Apache Storm and Spark (in seconds)

	6x t2.micro 10,000 x 20	**6x t2.micro 100,000 x 20**	**6x t2.micro 1,000,000 x 20**
Apache Storm	59.31	480.00	Incomplete
Apache Spark	0.01	0.23	1.19

4.3 Data Processing Implementation with Apache Storm and Apache Spark

Our running application used Apache Spark Streaming for matching users by feature vectors, but we also implemented Apache Storm for performance comparison. Table 1 reflects a comparison of data quantity and performance scalability between the two applications. Data was represented as a 64 bit floating point matrix of size 10,000 by 20, where 10,000 represents a hypothetical number of users, and size 100,000 by 20, with 100,000 hypothetical users. Clusters of six t2.micro instances were used. For Storm, these t2.micro instances excluded the nimbus, user interface, and Zookeeper servers. For Spark, the master is excluded from this count.

Apache Storm worked poorly for our application because, while Storm topography could initialize and await new data and always run the same logic on new data, a bottleneck was introduced at the spout phase of the topography, where the batch matrix data was continually loaded and sent to processing nodes, and required keys for matrix rows to be consistent and named by only one spout. This could have been mitigated by partitioning the matrix and manually assigning new bolts; but, ultimately, Storm was designed for other applications, particularly ones with inconsistent, heterogeneous data that bolts would quickly process from multiple spouts.

5 Conclusions and Future Work

Our final evaluation was focused on emulating the client experience and chat at the client side. At the server end we compared performance of our data processing systems Spark and Storm, and found Spark to be an ideal match for our data processing needs. As far as future development ideas go, there are several ways the application can be improved upon. Instead of using a centralized server approach for chat, an encrypted P2P approach could be implemented. Machine learning and deep learning algorithms could be leveraged for achieving better matching of users with similar conditions. Algorithms could even be used to use the data for prediction of future health issues and warn users. Matched users could be added to a forum kind of group chats.

References

[Col] Github repo for this report. https://github.com/ReedAnders/distributed_health_app.

[gnu] Gnu talk peer to peer chat system. http://gnutalk.sourceforge.net/.

[ICE] Ice: Interactive connectivity establishment. https://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment.

[KR09] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[NK16] Henry Neeb and Christopher Kurrus. Distributed k-nearest neighbors. 2016.

[Red] Redis: An open source (bsd licensed), in-memory data structure store. https://redis.io/.

[STU] Stun: Session traversal utilities for nat. https://en.wikipedia.org/wiki/STUN.

[Tor] Torchat peer to peer chat system. https://github.com/prof7bit/TorChat/wiki.

[TUR] Turn: Transversal using relays around nat. https://en.wikipedia.org/wiki/Traversal_Using_Relays_around_NAT.

[Yan01] Hector Yang, Beverly; Garcia-Molina. Comparing hybrid peer-to-peer systems. 2001.