PeopleWeb:
A System to Extract Social Data from Interpersonal Communication

Senior Research Project

Dana Spiegel
April 30, 1999

1. ABSTRACT

PeopleWeb is a system to extract, from communications between people, data about each person's interests and connections to other people. A modular framework is presented that simplifies the process of adding new modules that augment the functionality of system. The system is evaluated based on an initial test data set and is found to produce useful and novel data about the connections between people, as well as their interests. Future directions for the project are discussed, and scenarios for its introduction into the workplace are considered.

2. INTRODUCTION

Humans are evolutionary creatures. They are not static—they move about intellectually, socially, and physically. Even the most learned of physicists has trouble at times with rudimentary mechanical analysis: his knowledge of physics has evolved such that he may not immediately remember basic analysis. It is only natural that over time a person forgets one thing, while at the same time, learns another. (This is not to be confused with the view of the human brain as a limited-size storage device: our tendency to lose knowledge is not due to a need for "space" to store other knowledge, but rather is a function of our neuronal chemistry, a subject beyond this paper.) Likewise, it is natural to learn new things over time: a person who is a novice programmer will, over time and with practice, become more proficient and knowledgeable in his field.

Social structure is also dynamic: a best friend drifts away, a colleague changes jobs. We grapple every day with our network of contacts, maintaining a relationship that is worthwhile or informative, and falling out of touch with someone who, for some reason, is no longer involved in complementary work. Keeping track of these relationships takes time and effort, and as our contact networks grows, we tend to lose track of some of these relationships. It is not

unusual to *not* keep in touch with someone who could be a rich source of information, and instead to try to acquire that crucial information through inefficient (albeit better known) channels.

Furthermore, for the businessperson just starting out, building a network of contacts is a difficult and time-consuming—perhaps even daunting—task. This task is made even more difficult by a lack of knowledge about each possible contact's expertise. Even for an established organization, with defined lines of information flow, knowledge of communication lines and their content is opaque not only for the new employee, but for the experienced worker as well—especially if that experienced worker is involved in only certain aspects of a company's workflow. There are also those lines of communication that arise from random contact between people within an organization, or from interaction that is not directly related to a person's job. (Perhaps two people from different areas of an organization share a common hobby that is not part of their work, but *is* part of a third person's job.) These links between people are known only to a small group that shares a work environment with the participants, or perhaps known only to the participants themselves.

Information about the social network within a community is lost because we—as individuals and organizations—don't keep track of it. Groups that try to explicitly maintain a knowledge base about each person's work and expertise miss most of this information because it is not part of a job description and is often not reported by each employee. The organizations that attempt to keep track of people's contacts, and that make this information available to other workers, miss the single most important aspect of those contacts: their *temporal evolution.* A contact's importance changes over time depending on each person's job (which also evolves over time) and the changing status of each person's other contacts.

The ideal way to keep track of contact and expertise information is to build an "intelligent" system that understands the evolution of such information. Today's systems are static; they don't understand that every bit of information they contain is constantly evolving, and they don't allow for the cascading effect a single change in one piece of information can have on every other piece of information. As a result, information databases quickly become stale and outdated. An "intelligent" system would work on its own, constantly monitoring and mining both explicit and implicit information from communications streams to keep its data current, without relying on individual user input to build its databases. Web crawlers are good examples of such automatic data mining systems, but even they only go so far. Web crawlers do little to "mine" the information; instead, they "gather" and index textual information, providing a raw database that can then be manually searched for information, leaving the user to figure out what to do with the results.

The *PeopleWeb* system presented below is just this kind of "intelligent" system. Its basic purpose is twofold: to keep track of each person's interests, and to keep track of the links between individuals within a community. (A *community* is any group of people who interact with each other on a regular basis.) *PeopleWeb* is built with the previously stated ideals in mind: it automatically gathers information and it understands the volatile quality of the information it maintains.

PeopleWeb goes a few steps further, however. Since communication between people is not bound to any particular medium, *PeopleWeb* provides an open architecture into which any communications medium can be inserted: email, web pages, Usenet, chat, pagers, telephones. As long as information can be extracted in textual form from a particular medium, that medium can be included as a source of information for *PeopleWeb*.

The information maintained by *PeopleWeb* is useless without some intuitive way to access it. *PeopleWeb* provides a client application that presents its knowledge in multiple ways. No single interface to information is universally ideal, so the *PeopleWeb* system provides not only a graphical view of its database of people and their connections and interests (which is generally suitable for direct individual interaction), but also provides table and list views that can be used when appropriate. Since user interface design is constantly evolving, and since each person has their own personal display preferences, *PeopleWeb* provides an open architecture for data visualization.

PeopleWeb collects and makes available information about people and their connections to other people that otherwise would be lost. Using *PeopleWeb*, a new employee can see the information flow between people in his new community. He can instantly access information about the expertise, explicitly stated or not, of every member of his community. And an experienced worker can see those connections, between himself and others, that are rich in information flow. He can find the most direct path to the person who is most knowledgeable about a subject in which he is interested. *PeopleWeb* can reveal productive but unrecognized connections, and can make this important information available to everyone. It can reveal individual expertise that would otherwise have gone unnoticed, and can inform others of such knowledge.

As companies grow, information about the informal structure of their employees becomes even more important, and managing this information becomes exponentially more difficult. *PeopleWeb* provides a way to manage this information, and makes this information available in an easily understandable way so that a community can function most efficiently, taking full advantage of all of its members.

3. DESIGN

The *PeopleWeb* system comprises five modules, each representing a step in the process of converting textual data into a usable graphic representation. The modules, termed *DataAdapter*, *DataMiner*, *PersonPublisher*, *PeopleWebReader*, and *PeopleWebRenderer*, each are integral parts of the *PeopleWeb* system, and together implement the entire process.

Each module is built as a Java "Bean." This allows any of the modules to be replaced or augmented with a different one of the same kind, making the system as a whole adaptable to any environment. Interfaces for each of the modules are provided, so that a user may write plugable modules in order to augment or modify the functionality of the basic system. At least one module of each type is provided in the *PeopleWeb* package.

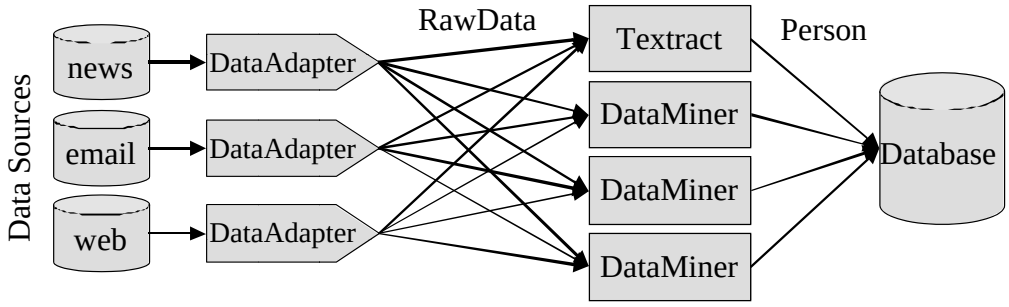

Figure 1. Framework for the *PeopleWeb* system. Data sources introduce data into the system via *DataAdapter*s. *DataAdapter*s convert this data into *RawData* objects, which are processed by *DataMiner*s. *DataMiner*s then extract *Person*s from the data and insert them into the database.

3.1. DataAdapter

The *DataAdapter* module implements the data input part of the *PeopleWeb* system. A *DataAdapter* wraps a particular form of data (for example, a Usenet server), and provides data to the *PeopleWeb* system in a standard form. All data provided to the system is packaged in a *RawData* object, which holds the data and meta-data to be processed by a *DataMiner*.

The *RawData* object is defined by the interface in `dspiegel.PeopleWeb.RawDataInterface`. A user can create his own storage class for data, and so long as it implements the *RawDataInterface*, the class can be used by the *PeopleWeb* system. The methods in the *RawDataInterface* are:

```
public Hashtable getMetaData();
public String getBody();
public Date getTime();
public String getOwner();
public String getGroupID();
public String getUID();
```

The *RawDataInterface* abstracts a communication into an object with six pieces: `Metadata`, a `Body`, a `Time`, an `Owner`, a `GroupID`, and a `UID`.

`Metadata`, which is provided in the form of a hashtable, is any information that describes the circumstances of the communication. In the case of an email, this information is the email header. For a web page, this information may be `<META>` tags embedded in the HTML. To allow maximum freedom in defining the type of information held in the hashtable, the data is assumed to be representable as a freeform string. The only requirement is that this information be provided by the storage class in the form of key/value pairs. For example, the header from a Usenet news group, which has the following form:

```
From: dspiegel@us.ibm.com (Dana Spiegel)
Subject: PeopleWeb implementation
Newsgroups: comp.sys.mac.programmer.misc
Date: Tue Aug 17 14:41:44 EDT 1998
Organization: IBM
Lines: 2
Mime-Version: 1.0
Content-Type: text/plain
Content-Transfer-Encoding: 8bit
```

is easily translated into a hashtable: the keys are the identifier strings before the colon; the values, the strings after the colon. Generally speaking, a hashtable key should always be a categorization for the value it encodes.

The `Body` of a *RawData* object is the text of the communication. For an email message, this is the body; for a Web page, it is the actual text of the HTML document. The data must be provided as a string (regardless of the *RawData* object's internal form for this data), which is then used as the central data on which the *DataMiner* will work. Whereas all other fields of the *RawData* object describe the context of the communication, the body is where the actual communication should reside. It is suggested that this text be stripped of supplemental text (such as HTML tags), so that the *DataMiner* does not process this extraneous data as part of the communication.

The `Time` is the time when the communication occurred, or if that is unavailable, the time when the data was first read. This data is used by the *DataMiner* to gauge the timeframe for the validity of the information mined from the *RawData* object. For a Usenet article (see above), this should be set to the `Date`.

The `Owner` is the person who is doing the communicating. In a Usenet article or email, this is the person represented by the `From` tag. Though it is not necessary, specifying the owner of a communication aids the *DataMiner* in identifying the person to whom unclaimed key words should be assigned. For

example, in an email, topics discussed which are not tied to other people mentioned in the communication, are tied to the writer of the email.

The `UID` is a unique identifier for a particular *RawData* object. It can be any object with a string representation. Since most *RawData* objects encapsulate a complete communication, such as an email or Web page, it is suggested that the `UID` for that *RawData* object refer back to the communication which is being encapsulated. For an Usenet article, this is the `Message-Id`; for a Web page, this is the Uniform Resource Locator (URL) for that page. Currently, it is required that a *RawData* object provide some form of unique id, so that each *RawData* object can be uniquely identified by a *DataMiner* when processed as part of a group.

The `GroupID` is the identifier for the group to which a particular *RawData* object belongs. Certain *DataMiner* modules will require a *DataAdapter* to provide a group of related *RawData* objects so that each individual *RawData* object can be analyzed in context with related *RawData* objects. In the case of a Usenet article, the `GroupID` is the newsgroup from which the article was retrieved. For a *DataMiner* that requires a group of *RawData* objects, all articles from that particular Usenet group are encapsulated and sent to the *DataMiner*.

A *DataAdapter* must implement the following methods:

```
public void nextRawData();
public boolean moreRawData();
public RawData getRawData();
```

These methods provide the entire functionality of a *DataAdapter*. `moreRawData` checks to see if there are any *RawData* objects to be retrieved by the calling object. `getRawData` does the actual retrieval of a *RawData* object. `nextRawData` readies the next *RawData* object for retrieval. In this way, a *DataAdapter* can provide a conduit for both a static set of data and a continuous feed of data.

The current process for using a *DataAdapter* is to instantiate a new *DataAdapter*, and then retrieve *RawData* objects from it, one at a time, while there are *RawData* objects available for retrieval. The following code shows an example:

```
String theGroupID = new String();
Vector myRawDataVector = new Vector(100, 20);
DataAdapter myAdapter = new NNTPAdapter();

while (myAdapter.moreRawData()) {
    RawData myRawData = myAdapter.getRawData();
    if (theGroupID.length() == 0) {
        theGroupID = myRawData.getGroupID();
        myRawDataVector.addElement(myRawData);
        myAdapter.nextRawData();
    }
    else {
        if (theGroupID.equals(myRawData.getGroupID())) {
            myRawDataVector.addElement(myRawData);
            myAdapter.nextRawData();
        }
        else {
            break;
        }
    }
}
```

This code retrieves from a *DataAdapter* a group of *RawData* objects. A *String* is created to keep track of the current `GroupID`, a *Vector* is created to hold the *RawData* objects, and the *DataAdapter* is created. While there are *RawData* objects to be retrieved, a *RawData* object is retrieved from the *DataAdapter*. The *RawData* object's `GroupID` is checked. The current *RawData* object is added to the *RawData Vector*, and the *DataAdapter* is incremented.

On subsequent loops through this code, *RawData* objects are retrieved, and their `GroupID`s are compared to the stored `GroupID`. If there is a match, then that *RawData* object is added to the *RawData Vector* and the loop runs again. If the `GroupID`s do not match, then the loop is stopped, and further processing of the *Vector* of *RawData* objects can proceed.

This code sample assumes that the *DataAdapter* provides *RawData* objects

from the same group sequentially, so that if a new group is encountered, there are no further objects from that group to be retrieved. This is not always the case: a *DataAdapter* can provide *RawData* objects in any order, so that *RawData* objects from many groups are mixed together. Code should assume that a *RawData* object could be retrieved at any time that belongs to a group that has already been encountered.

3.2. DataMiner

The *DataMiner* module implements the basic data processing part of the *PeopleWeb* system. A *DataMiner* wraps a particular data-mining program or sequence of programs, taking as input a *RawData* object or group of *RawData* objects, and produces a *Person* object or series of *Person* objects.

A *DataMiner* must implement the following methods:

```
public boolean takesGroups();
public Vector processRawData(Vector theRawDataVector)
      throws Exception;
public Vector processRawData(RawData theRawData) throws
      Exception;
```

`TakesGroups` checks if the *DataMiner* takes *RawData* objects individually or as a group. `ProcessRawData` takes either a *RawData* object or a group of *RawData* objects, mines the data contained in the *RawData* object(s), and returns a *Vector* of *Person* objects, filled with the data that has been extracted from the *RawData*.

A *Person* object describes a person by a name, a unique id, a set of *Connections*, a set of *Interests*, and a list of other names by which the *Person* can be referred. Each unique *Person* has a unique id. When the *Person* Database encounters two *Person* objects that have the same unique id, it attempts to merge the data that is contained within each of them. If data is not in conflict, it is aggregated. If it is in conflict—for example, if one John object `ASKS` Matt a

question about Java, and later the John object ANSWERS a question from Matt about Java—the *Connection* is then upgraded to TALKS.

3.3. PersonPublisher

The *PersonPublisher* module implements a basic database publishing service utilizing Java Servlets so that applications can access the stored *Person* information via a native Java object. In this way, an application programmer who wishes to utilize *PeopleWeb* data has merely to talk to a *PersonPublisher* using a simple API to extract data about a *Person* or group of *Person*s (as opposed to publishing the raw data to be accessed with complex queries using Java Database Connectivity or JDBC). This method of publishing *PeopleWeb* data also allows clients within the *PeopleWeb* system to work only with Java-native *Person* objects and not have to translate data into *Person* objects.

At present, the *PersonPublisher* interface has neither been defined nor implemented.

3.4. PeopleWebReader

The *PeopleWebReader* module makes programmatically interacting with a *PersonPublisher* database simple and transparent for a *PeopleWeb* client developer. By utilizing a *PeopleWebReader*, a client can request information about a *Person* through the *PeopleWebReader*, and the *PeopleWebReader* will contact remote *PersonPublisher* databases to retrieve the necessary information. Further, the *PeopleWebReader* provides a temporary local store for *Person* objects, which can function both as a cache for already retrieved *Person* objects and as a proxy for updating *Person* object information to be submitted to a *PersonPublisher*.

The functionality of the *PeopleWebReader* as a cache allows a *PeopleWeb* client to operate over a low-bandwidth or intermittent connection to the network. A *PeopleWebReader* can query a *PersonPublisher*, download and store needed *Person* information, and then disconnect from the *PersonPublisher*, allowing the client to function without needing to talk to the *PersonPublisher* constantly. This is a useful feature for use on a laptop: a businessperson, working while in transit, can successfully use a *PeopleWeb* client and extract meaningful information from the *PeopleWeb* system.

The *PeopleWebReader*'s functionality as a proxy for updating a *PersonPublisher*'s data is similarly important to a *PeopleWeb* client's functionality. A *PeopleWebReader*, when receiving data from a user, will update the local copy of the *Person* object. The *PeopleWebReader* contacts the *PersonPublisher* and updates the information in that database either when a set amount of time passes, a network becomes available, or there are a significant number of changes to be propagated to the *PersonPublisher*. This method allows for a more efficient communications scheme between *PeopleWeb* clients and servers: since there is no continuous connection to the *PersonPublisher*, the *PeopleWebReader* can operate offline. Also, since a group of updates are sent at once, instead of one update at a time, the updates made to the *PersonPublisher* database can be optimized.

At this time, the *PeopleWebReader* component to the *PeopleWeb* system has not been built.

3.5. PeopleWebRenderer

A *PeopleWebRender* is an object that displays data about a *Person* or a group of *Person*s to a user. A client application that wants to utilize *PeopleWeb* data can pass *Person* objects to an instantiation of a *PersonWebRenderer*, which will render on screen that information. In this way, a client application can

seamlessly utilize the *PeopleWeb* system and provide access to *PeopleWeb* data to a user.

Though no *PeopleWebRenderer* have been implemented, the *PeopleWeb* system should contain a set of standard displays for this data: tree format, grid format, and graphical format. The tree and grid formats are implementations of standard Java Tree and Grid information display classes, as shown in Figures 2a and b. The graphical format is a proposed display, specific to *PeopleWeb*, that provides a graphically intuitive display for *Person* connection and interest information. A sample of this display is shown in Figure 2c.

Interests			
Person	**Topic**	**Rating**	**Modified**
Dana	Java	Expert	April 24
Matt	Java	Novice	April 10
Eric	Java	Novice	March 3

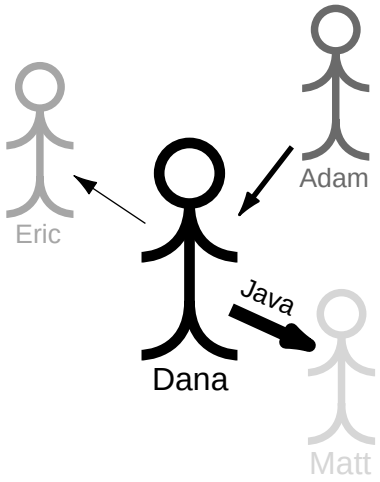

Figure 2. a) A tree view of the *Person* object Dana. By clicking on the triangles, the nested information is shown. b) A list view of *Interest* data. c) A graphical view of four *Person* objects. The arrows show the flow of information, and the brightness of the connected *People* signifies the age of the connection. The thickness of the arrows represents the intensity of the connection.

4. IMPLEMENTATION & ANALYSIS

Currently, the only modules of *PeopleWeb* to have been implemented are the *DataAdapter* and the *DataMiner*. Thus far, the system can retrieve data from a Usenet news server, package the data into *RawData* objects, send these *RawData* objects to the Textract *DataMiner*, and store the resulting information in a DB2 database. Though the system is not yet complete, interesting results have already been obtained by running operational components of *PeopleWeb*. Discussing the functionality of the Textract *DataMiner* is also worthwhile, since it is perhaps the most integral part of the *PeopleWeb* system.

Textract is a toolkit that textually analyzes a set of documents to extract both information about the "named" objects present in the text, as well as relationship information about those objects. A "named" object can be thought of as a subject or object in a normal English sentence. For example, Textract would categorize "Dana Spiegel" as a "Person," while "MIT" might be categorized as a "place" or an "organization," depending on its context. Textract also can *canonicalize* a name: "MIT" and "Massachusetts Institute of Technology" both refer to the same underlying object within a sentence. Textract might deem "Massachusetts Institute of Technology" the canonical form of that object, and "MIT" a variant form of that same object.

Using this capability, *PeopleWeb* can extract the objects within a communications transcript, and track their use over the course of a conversation. *PeopleWeb* can, for example, see that "John" and "John Doe" are used interchangeably within a communication. Using Textract, *PeopleWeb* can refer to the single object John Doe when analyzing the text that surrounds that object, without needing to always connect the particular instantiation of the person to Textract's underlying representation of that person.

Textract also finds relationships between objects. Consider the following sentence: "Charles Vest, the President of MIT, today announced he will begin jogging with students on a daily basis." Reading this sentence, a person would conclude that Charles Vest is the president of MIT. Textract can also find this relationship, reporting that `Charles Vest : is-a : President` and `Charles Vest : President : MIT`. With this deduction capability, *PeopleWeb* can extract associations between people to include in the *Person* database.

Taking a bit of *RawData* and feeding it into Textract allows *PeopleWeb* to extract from the text people and their relationships to other people and things. This is only part of the process that a *DataMiner* must perform, however. Since the people found by Textract are at best represented only by names, *PeopleWeb* must connect these names with an online directory that contains identifying information—name, email address, unique id—for each known person. This way *PeopleWeb* can link the data that Textract finds about a person to a particular, known person in the *PeopleWeb* system. Otherwise, *PeopleWeb* may confuse John Doe and Jack Doe, and the stored information about each person will get mixed.

The job of validating people falls to the *PersonFinder* object that resides within the Textract *DataMiner*. The *PersonFinder* takes all of the people that Textract has found and matches them to an online directory. This is no small task, however: there might be multiple Johns or, indeed, multiple John Does in our text or in our database. We have a set of algorithms to sort out this confusion.

The simplest case arises when there is a one-to-one match: one John Doe in the text matches the one John Doe in the online directory. In this case, the *PersonFinder* merely uses the unique id of the John Doe in the online directory to store information in the *Person* database.

More complicated cases arise when there is more than one John Doe in the online directory. In this case, the *PersonFinder* looks at the context in which the John Doe arises in the text and tries to match as many interests or connections as possible to each John Doe in the directory. The directory entry that is most similar wins, and is used to store data in the *Person* database. In the case that there is more than one equal match, the PersonFinder just picks one of the directory entries. A similar process ensues should there be more John Does in the text than there are in the directory: closest matches, based on context, win. Any extra John Does from the text are discarded, since we cannot validly identify them.

This algorithm should be modified to support a qualitative evaluation of the match that is made. This will allow *PeopleWeb* to evaluate the accuracy of the new data. For example, if a person is closely matched, then *PeopleWeb* can infer that the new data is a good description of that person. If a person is only partially matched, *PeopleWeb* can still store the new data; however, it will wait until similar, validating data is encountered before making inferences based on the possibly inaccurate data.

The last part of the analysis is the qualitative evaluation of the type of communication. Once the people are identified, the text is searched for keywords and particular grammatical constructs. For example, if there is a communication from John to Matt about Java, and there are question marks within the text, then the *DataMiner* categorizes this communication as a question, and stores `JOHN ASKS MATT ABOUT JAVA`.

All information that is inserted into the *Person* database is dated. This allows the system to treat old information differently from new information. For example, if John Doe was interested in Java a few months ago, but no recent communication from or to him deals with Java, *PeopleWeb* can deduce that he

isn't very interested in Java now. More interestingly, suppose there are two John Does, one that has recently become interested in squash, and one that was at one point interested in it, but is no longer. A *DataMiner* encounters a John Doe that is interested in squash. *PeopleWeb* can infer that the John Doe encountered is more likely to be the John Doe that the system knows as recently interested in squash, and less likely to be the John Doe that has lost interest in the topic. *PeopleWeb* can interact with volatile data based not just on its categorization, but its age as well, and can more intelligently process and portray data about people.

The *PeopleWeb* system was tested using as input articles from a Usenet newsgroup. The social data that was extracted by *PeopleWeb* shows that useful information was indeed extracted using the *PeopleWeb* framework for data analysis. Connection information, though not complete, was present, and each *Person* had associated with them a partial set of interests.

When looking at a few *People* objects, it is easy to see the type of connections they have to each other. If, however, the *Person* database is expanded to contain thousands of objects, each with at least a few dozen associated connections and interests, understanding the raw data becomes impossible. For this reason, intuitive visualizations such as those shown in Figure 2 are required to view this data, as are simple query systems for filtering the data. It is unfortunate that the implementation of the data-collection pieces of the *PeopleWeb* project required such extensive development before real results were seen, since the intuitive displays for this information required more time than was available. Further development of this project should include similarly extensive development of visualizations for this data.

Also, a more complete analysis of the accuracy of the algorithms for data extraction discussed above was not done due to time constraints. The *PeopleWeb*

system must be run using a large amount of input data and a complete online directory before conclusive evidence can be found for its successful implementation of the ideals stated in the introduction. Because of the design of the system, the entire infrastructure of the data-acquisition part of *PeopleWeb* had to be developed and implemented before the system could be tested. This requirement of a complete infrastructure is one of the tradeoffs of having such a modular system. However, now that this infrastructure is in place, adding new data sources and new data mining applications is fast and easy. Further development of the *PeopleWeb* project should incorporate these new modules.

It is difficult to evaluate whether *PeopleWeb* extracts the proper amount of information from communications text. Since some of the algorithms used in *PeopleWeb* are simple, it may be possible to extract more data from the text. Still, *PeopleWeb* can extract a significant amount of information, and it is unclear that implementing more complex textual analysis algorithms will increase the amount of data that can be functionally extracted from the text. It is more likely that increasing the complexity of the algorithms will increase the *quality* of the data.

The issue of security and privacy of information has not been addressed. The *PeopleWeb* system is intended for non-malicious gathering and extracting of connection information; however, there are data sources (such as email) that are private to those who own them. A security and privacy policy must be implemented so that these rich data sources can be tapped.

5. CONCLUSION

The *PeopleWeb* project can be considered a partial success: the goals set out in the introduction—to mine interpersonal communications to extract information about a person's interests and the connections they have to other people—have been met in part. A complete framework for the project was developed and an application was written that takes advantage of this framework to extract this interest and connection data from Usenet postings and store this data in a database. Pertinent information about the people participating in this Usenet group was collected. A larger test run of this application, as well as testing within a real-world setting, is needed to conclusively claim that *PeopleWeb* is a worthwhile system for extracting and collecting social information. However, preliminary results indicate that *PeopleWeb* is a useful and novel system.

Since the system is incomplete without an intuitive user interface, further development of the *PeopleWeb* framework and the applications that are based upon this framework are necessary if the project is to be considered a complete success. This further development is much easier because of the *PeopleWeb* framework. Furthermore, the system design principles of the *PeopleWeb* framework make augmenting the system, either by the developers or by third parties, an easy task.