WhatsHere

Geoffrey Hayes [hayesgm@seas.upenn.edu]
Professor Jianbo Shi
University of Pennsylvania
April 18, 2008

Abstract

WhatsHere is a web application allowing users to find events in their community. The system uses a crawler to mine data from newspapers and websites across the Internet. Based on a collaborative recommender system utilizing context information, the WhatsHere recommender system takes a novel approach by focusing strongly on source similarity and adaptive application-specific context curves for individual users. Our algorithms provide linear-time queries to ensure scalable real-time results.

1 Introduction

There exists an emerging gap in the use of the Internet to approach general location-based event searches for web users. While many search engines and geographical service websites allow users to lookup points of interest by location (e.g. Citysearch, Google Maps), these fail to address the issues of locality, punctuality, and user-preference. WhatsHere allows a community of users to search for and contribute to localized events. A simplistic yet effective frontend web display limits input from the user while interactively engaging his or her interests.

We have developed a search algorithm where a web user may search events by simply specifying a lookup location. Combining this address with the user’s history and preferences, the system delivers a sample of results to the client. The algorithm then uses the interactions of the web user in order to drill-down on the intent of the user for future searches. Having observed trends that a single source (web site, newspaper, ...) often hosts by-and-large similar events, WhatsHere categorizes events by their originating source.

2 Related Work

WhatsHere relies on two primary pillars for dynamic event delivery, which when taken in conjunction provide a proof-of-concept to the WhatsHere approach: event crawling, and associated relationships of data.
The most basic related work involving event crawling is Zvents.com, an event search engine which likewise utilizes the Heritrix open-source web crawler. The Zvents engine is built on a two-index crawl. The first index is a compilation of event-bearing websites, discovered through both a general crawl of the web and user-submitted websites. Once friendly-content websites are identified, the second crawler then scans the contents of the first index repeatedly, attempting to identify events based on proprietary algorithms, the result of which is a massive catalog of event data, further reinforced by the ability of users to manually enter individual events on top of this list.

The second related work, essential to our user experience, is the concept of related data points, or the thumbs up thumbs down approach best implemented by Pandora.com. Pandora takes into account, when delivering content, the underlying characteristics of the data space, users acceptance or rejection of characteristics of data, and finally similarities to other users preferences, all of which are represented by the WhatsHere approach.

3 Approach

This project requires integration from many sources and different servers (see figure 1). Geographical data is obtained from deCarta web services. A database server is required for data mining and serves an integral component of the search and user profiling systems. An application server provides an HTML/XML front-end for the end-user browser. Finally, a web crawler application fetches, identifies features, and stores events from an unlimited number of discovered web sites.

deCarta is a company which offers a geospatial software platform to perform many functions necessary to WhatsHere. They offer a professional web service API which we access through Java, performing Remote Procedure Calls (RPC). These calls are performed by our application server running Apache Tomcat. We use deCarta’s geocoding service to transform data from postal addresses to latitude/longitude coordinates. Their JavaScript API is used to fetch map data in the form of GIF images which are presented to the user. deCarta provides routing information, to obtain directions from one location to another and to present this path to the user. Finally, deCarta offers a point-of-interest service which can be used to retrieve static lookups in an area not included in our database store.

Data from the web crawler and the user community are stored using MySQL’s database application and server. A general schema is provided in figure 2. The user table contains information specific to each user. The source table specifies a list of all event sources which we have discovered, populated primarily by our web spider and identified by a URI hash. The event table contains all available information on specific events (e.g. its location and timespan). The host table specifies which sources are hosting which events (a single source may host multiple events). Finally, the alike table qualifies the similarity between two sources for our recommender algorithms. The database contains a large set of
Figure 1: The flow of data between servers
Figure 2: The database schema

Stored Procedures to ensure the speed and consistency.

The Apache Tomcat application server supports JavaServer Page (JSP) technology and Java Servlets. JSP pages are used for delivering integrated content in the form of standard HTML to the end-user browser. These pages in turn will make Javascript calls with AJAX functionality. AJAX allows a single retrieved web page to dynamically update its content without sending an entirely new page to the user. Java Servlets are used to interpret and respond these XML messages.

We have developed a web crawler (spider) to discover events throughout the Internet. This spider runs as its own process with multiple threads. Each thread interprets a web site, discovers URLs and adds each to a cache of URLs to search if it has not already been seen. This lookup is done by storing each URL in a database keyed on the URI hash. When interpreting each page discovered, the spider also looks for features which fit our criteria as event specifications. If found, the spider adds this URL to a list of possible sources. A separate set of worker threads in the spider application retrieve and drill-down this list of URLs to discover event features and mark the address for future reference. This URI, based on a simplification of the original URL (e.g. www.philly.com from www.philly.com/events), is added as a source to our source database if it does not already exist. A third set of worker threads will periodically poll this source for updated event data.

Finally, we have created a high quality recommender system which, given past user ratings, dynamically recommends events to a querying user, described in section 4. Section 5 describes the algorithms used to implement the various components of the system. Section 6 details the running-time of these algorithms. We evaluate our methods in section 7, and section 8 describes our conclusions. The appendix details the functions referenced in section 5.1.
4 Recommender System

4.1 Related Recommender Work

As the amount and variety of information on the Internet continues to grow at a rapid pace, recommender systems have become an invaluable tool for users to drill-down on desired material among the masses. These technologies represent an important area of research as currently many commercial applications employ such systems to increase business activity, such as Netflix Prize and Amazon. The purpose of these systems is to present the user a novel set of recommended items or products, given information of his past purchases and rating history. Methods generally present the user an ordered list of recommended items by predicting the ratings of every item and sorting the items by rating [2]. The general problem is, given evidence $e$ collected by the system (rating history, item and user feature sets, ...), to present the user a set of items which maximizes the utility experienced by the user [1]. The difficulty is that the evidence is often very sparse and so the system must therefore make a best-effort recommendation about complicated client decisions. The research has typically categorized systems into one of three approaches: content-based, collaborative, or hybrid.

Content-based systems estimate the similarity between different items in the system and use this information to present users items which resemble other items that the user has rated highly in the past. This is based on the principle that if the user likes a certain item, it is probably he will like similar items. These systems, however, tend to over-specify the preferences of the user, only recommending items often too similar to previous purchases. Further, content analysis of the items is a non-trivial problem (most systems use a bag-of-words approach for text documents). Also, when the system has little information on the previous actions of the user, it is unable to make informed decisions (the new-user problem) [1]. Finally, these systems have been shown to scale unreasonably with an increase in items and perform poorly in real-time [2].

Collaborative systems, alternatively, attempt to estimate the similarity between users in the system and then use this information to choose items which resemble the items which similar users (peers) have rated positively in the past. These systems have been found to be generally more successful than content-based systems [2]. Two methods have emerged to implement collaborative systems: memory-based and model-based. Memory-based systems treat users as a feature vector, generally based on their previous ratings, and often perform a cosine similarity measure between users. Model-based systems build a graph with nodes as rating values (e.g. 1-5) and attempt to determine the conditional probability that a certain item would ascertain a certain rating for a user. Simple collaborative systems have even been shown to be more accurate than highly detailed content-based systems [1], and collaborative systems can recommend items which are dissimilar to the user's previous selections. Collaborative methods have difficulty in providing recommendations for items which have been recently added to the system (the new-item problem) or to users...
which are new to the system (the new-user problem). It has also been suggested that instead of rating the similarity of users by collaboration, it may be better to rate the similarity of the items by user-rating patterns\cite{2}\cite{4}. This is called item-based or content-based collaboration\cite{4}.

Finally, hybrid methods attempt to combine the results from content-based and collaborative systems. Research has shown that hybrid methods can produce much richer results than content-based or collaborative systems alone \cite{11}.

The preceding approaches include only a fraction of the available data in their decision-making processes. Examining the context of the purchases or ratings in the system can allow much crisper results\cite{10}. For example, one may determine that academic users generally buy textbooks in the fall and novels during summer. Social context may also exhibit certain discernible patterns, such as the influence by trusted users upon his social connections. As emotions are crucial in real-world decision making, Gonzlez et al. suggest using emotional detection techniques to better inform existing algorithms of the emotional context of a client’s decision\cite{5}. Physical context data, such as a user’s GPS location can improve current techniques. Horozov et al. suggest that users who live in close proximity have higher agreement in restaurant ratings\cite{7}. These forms of additional information in the query may substantially improve results in many different recommender systems\cite{10}.

Further research suggests several techniques which may be employed in advent systems, such as multi-criterion analysis, non-intrusiveness, semantically-enhanced queries, and flexibility in query structure\cite{1}\cite{9}. Herlocker et al. suggest that serendipity, or a measure of novelty in returned results, may be an important metric for future analysis of systems\cite{6}.

### 4.2 WhatsHere Recommender System

WhatsHere, as stated in Section 2, is a dynamic web application which seeks to present users a well-tuned set of events from a plethora of data-mined sources. As we present the user successive sets of results and ask him to evaluate sources during each phase, we quickly develop the need for a robust recommender system to drill-down on the user’s desires. We treat the source of an event (e.g. The Philadelphia Inquirer) as an item in the traditional sense of recommender systems. In the terms of the previous section, WhatsHere uses an item-based collaborative recommender system implementing context support with emphasis on location and serendipity.

The goal of our recommender system is to present the client a robust set of 10 sources from a query based solely on a location. We let $C$ be the set of all users, $S$ be the set of all sources, and $u$ be the utility function for the user seeing a given event ($u : C \times S \mapsto R$), where $R$ is an ordered set. Of $Q$, the result set from a given lookup, we seek to maximize:

$$\text{aggr} \sum_{q \in Q} u(q)$$
Figure 3: A graphical example of a network with 4 sources ($\alpha, \beta, \pi,$ and $\omega$) where $\alpha$ and $\beta$ are highly probably to occur in conjunction, $\alpha$ and $\omega$ are highly unlikely to occur in conjunction, and we know very little about the concurrence of $\alpha$ and $\pi$.

The aggregate function may be either an $argmax$ or a summation. We note that while most systems are evaluated by correctly predicting a withheld user’s rating[6], we find that our goal is to present users with optimal but diversified results.

Much literature has shown the successful implementations of collaborative recommenders[2]. This method allows users to be given suggestions which may vary from the user’s discrete rating history, but still be of high utility. This outcome is optimal for our system.

As suggested by Deshpande et al., our system determines the similarity between items (sources) and not directly between users [2]. This has been shown to produce systems which scale much more flexibly (especially as we anticipate a significantly larger base of users than sources) [4]. We determine the similarity between the users using a novel approach described in the following section.

### 4.2.1 Source Similarity by Markov Network

There have been many approaches measuring the similarity between users (and items) in collaborative recommender systems. We use a model-based approach measuring the conditional probability of a user liking one source based on aggregate analysis of users who have stated interest in that and other similar sources.

We create a simplified Markov network graph $\Xi = (N, E)$ with a node for each source $s \in S$ (see [3, Introduction to Statistical Relational Learning]). The edge weights ($W$) of the graph ($W : e \in E \mapsto [-1.0, 1.0]$) correspond roughly to the conditional probability of a generic user’s likelihood of being interested in both of the sources (high weight $\approx 1.0$) or only one of the sources (low weight $\approx -1.0$). An edge weight near to 0 implies that the sources have no discerned correlation. It is also worth noting the simplifying assumptions that all edge weights in this graph are symmetric ($W(e_1, e_2) = W(e_2, e_1)$) and assumed to be 0 if otherwise undefined. Figure 3 presents a graphical example of such a network.

Inference on a Markov network generally follow the form $P(Y|E = e)$ where $Y$ is a query of random variables in the network and $e$ is an instantiation of
evidence (a known partial-assignment of random variables). The query we seek to ask in WhatsHere is, given the user’s past rating history, what is the conditional probability of the user liking each other source in the network. Let $R_u$ be the set of all sources user $u$ has rated paired with his rating of that source, and let $R_u(x)$ denote the rating of source $x$ by user $u$. Then, for a given user $u$, we let evidence $e_u = R_u$. Let the value of each node $n$ in $Ξ$ (denoted $\text{val}(n)$) be considered the rating which user $u$ has given the corresponding source (and $∅$ otherwise). We seek to ascertain the expected utility of each node. Let $J(n) = \{x : x \in N \text{ and } \text{val}(x) ≠ ∅ \text{ and } W(n,x) ≠ 0\}$, i.e. the set of all properly valued neighbors of $n$. We estimate the expected utility of a node $n$ to be as follows:

$$E(u(n)) ≈ \text{val}(n) + k \times \sum_{x \in J(n)} \text{val}(x) \times W(n,x) \times δ$$

where delta is a limiting factor ($\approx 0.1$) and $k$ is inverse of the magnitude of set $J(n)$. In this way, we simplify our operation on the Markov network to produce a weighted sum of probable influences. In Section 6 we show that this assumption allows linear worst-case running-time for our algorithm.

We use information gleaned from collaboration of users to build and learn our Markov network graph. If we consider the weight of an edge between two nodes to be the flow between those nodes (interpreted as the conditional probability of a random user liking both of the adjacent nodes), we aim to increase the flow if a user positively rates a node adjacent to a node which we previously know he likes, and similarly reduce flow if a user positively rates a node which we previously know he doesn’t like. Let $M_r = \{|s| : (r,s) \in R\}$, where $R$ is the set of all sources paired with every rating given to it. If user $u$ rates source $s$ positively, we update as follows:

```plaintext
for each $r \in R_u$ do
    $r ← W(r,s) + (1 - W(r,s) \times \frac{R_u(r)}{\arg\max M_r})$
end for
```

In this way, our algorithm increases the flow between like sources (and limits the flow between unlike sources) with an exponentially-limited curve. Further, it computes the link’s weight by discerning how the user has rated each connected source versus the overall highest magnitude of rating for the source. We have found this technique appropriate to limit the influence of novel users.

### 4.3 Adaptive Context Curves

Our algorithm uses further event-specific metrics to inflate or deflate the estimated utility function based on contextual factors specific to our application. Several examples are listed below.

- Proximity: is the event close enough to the user?
Figure 4: Two context graphs. Graph (a) depicts the learning effect adapted to context curves. The curve shifts as we learn the traditional habits of the user. In Graph (b), we depict a static curve which weighs the importance of explicit user reviews, requiring an event to have both a high average and review count to achieve a high score.

- Punctuality: is the event soon enough for the user? Is the event too long away for the user?
- Time of Day: does the user enjoy events at this time of day?

We have found that these and other factors are extremely important in a user’s decision about specific events. We take a novel approach in recommender systems for evaluating the importance of these metrics. For each context $T$ based on factors $T_f = \{f_0, f_1, \ldots\}$, we have modeled a general user’s behavior with an adaptive curve $G$ (where $G : f_0 \times f_1 \times \ldots \mapsto [0 - 100]$). For example, the relative location of the event from the user, based on factors: the GPS position of the user, $(\text{lat}_u, \text{lon}_u)$, and the GPS position of the event, $(\text{lat}_e, \text{lon}_e)$, attains the curve:

$$G_{\text{location}} = \frac{k}{l - \sqrt{(\text{lat}_e - \text{lat}_u)^2 + (\text{lon}_e - \text{lon}_u)^2}}$$

Where $k$ and $l$ are generic coefficients. We pick default values for $k$ and $l$ for each user. As we learn the user’s habits, we adjust $k$ and $l$ to better fit the user (see Figure 4 (a)). This approach allows us (if we choose accurate curves for a given context) to overcome the new-user difficulty (as we have basic habits built in), and simultaneously drills-down on specific user behaviors.

WhatsHere utilizes many different types of context curves. Our system, for instance, allows users to write explicit reviews for events (beyond the simple thumbs-up/thumbs-down). In Figure 4(b), we show the importance of the factors (total review count and average rating) and their composite effect on the context score. When we combine the scores from many context curves, we achieve an apt modifier to the recommender system’s rating.
4.4 Serendipity

One of our founding goals is to present each user a unique experience by selecting events which vary from those he or she has attended in the past and yet still pique his or her interests. To achieve this end, we decided not to choose the top-n highest scoring events from the previous sections, but instead to use the score as a probability indicator. That is, let $E$ be the set of all events and $S(e)$ be the score attained by an event $e$ from the recommender and context systems. $P(e)$ is the probability that $e$ will be chosen by the WhatsHere system.

$$P(e) = \frac{S(e)}{\sum_{e \in E} S(e)}$$

After choosing an event $e$ by this probability scheme, the system removes $e$ from $E$ and repeats 9 times (or until $E$ is empty). By design, events that are dissimilar from the user’s taste will still (infrequently) be presented to the user to allow him to fine-tune his taste to these suggestions. We have found that this effective to add interest to our system.

5 Algorithm

Our search algorithm combines information from many inputs in order to provide a robust but randomized set of results to the querying user. For a new user, we use his or her location as the most important data point for the query. As the user tells us his or her interests by rating events, we begin to profile the user on the sources in which he or she may be most interested. Combining this profile with adaptive context metrics, we strive to allow the serendipitous discovery of events which a user might not even have known he or she would like.

Our approach is meant to be simple and intuitive. After obtaining an address from the user, we present the user with ten events in his or her area from a random sample space of local events. The user may examine these events and choose to rate one. As the user does this, we retrieve ten fresh results based on the updated user profile. As the user continues to rate sources, we are able to drill-down on the users preferences while providing a list of events to the user. The user may choose to view the specifics for any of these events and route a path from his or her address to the event.

A key component of our system is that we implicitly build a graph measuring similarity between sources (see section 4.2.1). Suppose we have a scenario where most users who like source $\alpha$ simultaneously like source $\beta$. If user $X$ who we know to strongly like source $\alpha$ submits a query, we would like with high probability to return an event hosted by source $\beta$. This is on the assumption that source $\alpha$ and source $\beta$ are, in this case, highly similar. Any user who likes one of these sources, given no other information, has a high probability of liking the other. The equations which these inferences are drawn on are as follows:
Let $\alpha \oplus \beta$ imply source $\alpha$ is similar to source $\beta$
Let $\alpha \ominus \beta$ imply source $\alpha$ is not similar to source $\beta$
Let $A + \alpha$ imply User A shows preference to source $\alpha$
Let $B - \alpha$ imply User A shows dislike to source $\alpha$
Let $\alpha \oplus \beta$ imply source $\alpha$ is very similar to source $\beta$ (and similar for other operators)

\[ P(\alpha \oplus \beta) \approx P(A + \beta | A + \alpha) \] (1)
\[ P(\alpha \oplus \beta) \approx P(A - \beta | A - \alpha) \] (2)
\[ P(\alpha \oplus \beta) \approx P(A + \beta | A + +\alpha) \] (3)

Suppose that user $X$ has often positively rated source $\alpha$ and negatively rated source $\beta$. If this user then positively rates source $\gamma$, then this contributes to the thesis that source $\alpha$ and $\gamma$ are alike, while sources $\beta$ and $\gamma$ are dislike. In this case, the system marginally strengthens the connectivity of $\alpha$ and $\gamma$ in the alike graph, while marginally weakening the connectivity of $\beta$ and $\gamma$. The amount to which the algorithm increases or decreases these connections is based on an exponentially-limited curve of how strongly user $X$ likes source $\alpha$ or dislikes source $\beta$ considered from the entire pool of people to have expressed interest in those sources. We prefer the user with the most experience with rating a source to have the largest effect on the likeness of that source. The case follows analogously in the reverse when a user negatively rates a source (the likeness is now strengthened with sources that user does not like).

The second feature of our algorithm is called fetch. When a user submits a query based on a location, we perform fetch which returns 10 results which we have determined may be of interest to the user. The user may then rate these sources (as above) and call fetch again with updated results. Fetch takes several factors into account: the proximity of the event to the user (with an absolute threshold), the previous ratings of sources by the user, and other metrics based on past user experience (such as the punctuality of the event).

To implement the crawler, we extract information from websites and other sources related to localized events. Data compilation is completed through the use of regular expressions (summarized in table 1). Framed to perform location-based lookup, the expressions capture fields such as postal addresses from HTML documents. To differentiate effectively several event listings presented on a single page, the crawler treats the HTML document as a malformed XML document and uses the Document Object Model (DOM) to discover nodes which contain single event listings. We perform a depth-first search on the tree-view of the document and find the highest node in the tree which fully contains a single address. This is performed by the searchTreeEvents algorithm and demonstrated in figure 5.
Figure 5: A sample webpage viewed as a DOM tree. The crawler algorithm performs a DFS to discover the highest elements which have exactly one descendant which contains an address. In this example, those such elements are marked with asterisks.

![DOM tree diagram]

Table 1: Regular Expressions used in Crawler

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example Data</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>238 S. 40th Street</td>
<td>(B(\text{\d+})s([\text{\w}\text{\d}.] + \text{s}) + (\text{Alley</td>
</tr>
<tr>
<td>city</td>
<td>Philadelphia, PA</td>
<td>([\text{\w}] + \text{,} (\text{s})? [\text{A-Z}] [\text{A-Z}])</td>
</tr>
<tr>
<td>url</td>
<td><a href="http://www.yahoo.com/">http://www.yahoo.com/</a></td>
<td>((\text{http:} / /</td>
</tr>
<tr>
<td>time</td>
<td>8p / 08:34 PM</td>
<td>(((0)? [0-9])(1 [0-2]))(0-5) [0-9] /</td>
</tr>
<tr>
<td>page division</td>
<td>&lt;div&gt; / &lt;span&gt;</td>
<td>((t) &lt; \text{s} /*</td>
</tr>
</tbody>
</table>
5.1 Functions

The following functions implement the algorithms described above. Complete
descriptions of these functions are included in the appendix. searchEventTree
performs a search of nodes in order to discover the highest nodes in a tree con-
taining a single address-node as a descendant. This function is used by the
crawler to find single event information on a web page which may contain sev-
eral event listings. The function uses descendantAddressCount which recurses
to count the number of address-nodes contained under each node in the tree.
searchEventTree then performs a depth-first search (DFS) to choose the high-
est nodes in the tree with only one address-node descendant and add it to our
result set.

ratePositive, simpleFetch, metricFetch, and quickFetch perform the rec-
ommender system’s main functions. ratePositive updates the user’s individual
rating for a source, as well as adjusting the likeness of relevant sources in the
alike table. Each of the fetch functions determine a score for each discovered
event and then select events at random based on these scores. metricFetch
and quickFetch perform identically to simpleFetch but adjust how the score is
determined. The remaining functions are helper functions, generally performing
database or deCarta operations.

6 Analysis

A worst-case running time analysis of searchEventTree would be exponential,
but as we memoize the results from descendantAddressCount, the aggregate
running time is in the order of n, the number of nodes in the tree. ratePositive
varies on input s, the set of sources the user has previously rated. The running
time of the function is $O(\|s\|)$ as the for-loop on line 5 will execute that many
times. For simpleFetch we have to consider three factors: the running time of
lookup, called lk, the set of sources the user has rated, s, and the set of events
returned by lookup, e. The running time of simpleFetch is thus $O(lk + \|s\| \cdot \|e\|)$. This is since lookup is called once and the for-loop on line 10 will run $\|s\| \cdot \|e\|$ times. Finally, the running time of quickFetch is similarly $O(lk + \|e\|)$, since
we limit the inner for-loop to running 3 times.

7 Evaluations

Herlocker et al. examined the many techniques which have been employed to
evaluate recommender systems[6]. The field has explored a rich set of methods
without consensus. Herlocker suggests each recommender system should focus
on a specific recommendation result. WhatsHere aims to Find Good Items,
while allowing each user to Express Self and Help Others. As an event-focused
recommender system, we have found no suitable data set available on which to
perform off-line analysis. We have therefore determined to synthesize a data set
in order to evaluate our system. As data sets such as the one explained below
are generally used in primary analysis, we expect to perform further research on controlled live user experiments in the future.

7.1 Synthetic Users and Data Set

To test our system, we created a data set which we hope is similar to real usage of our system. We designed our system and thus our data set on the following assumptions:

(1) There are many more users than sources

(2) Certain sources are inherently similar, as observed by many users

(3) Users generally rate such sources consistently

Keeping with these assumptions, we created 500 test users and 50 sources (each with a single event). Each source was placed into one of five categories, and each user was given a binary preference for each category. That is, User 0 may, for instance, like categories 1, 2 and 5 and dislike categories 3 and 4. Each of the 500 users then rated sources chosen at random from the system. If the source was from a preferred category, the user rated it positively 90% of the time (and negatively 10% of the time). Similarly, a user disliking the category would rate a source from it negatively 90% of the time (and positively 10% of the time). While this data set fits our assumptions, we note that it is tailored unrealistically and will differ from real world data. Our users may entirely disagree on the similarity of two sources or may rate sources rather arbitrarily. We seek to explore this in further experimentation.

7.2 Analytic Method

Herlocker et al. discuss using a Receiver Operating Characteristic (ROC) curve to evaluate the effectiveness of recommender systems. Based on signal detection theory, a ROC curve shows the probabilistic ability of a system to detect signal from noise. Applied to recommender systems, the curve exposes the system’s ability to choose relevant items over non-relevant items, which bears similarity to precision and recall methods. The area under a ROC curve (Swet’s A Measure or auc) is related to the system’s ability to determine a relevant item from a non-relevant item. A random predictor is expected to produce a linear ROC curve from the origin to the upper-right corner (auc=0.5), while a perfect predictor should create a curve which goes strait upward from the origin then straight right (bowed out, auc ≈ 1.0).

In our data set, we simulated each user rating 30 sources chosen at random. We also examined the benchmarks where users had each rated 5 sources, 10 sources and 15 sources. We performed the first step of fetch, but instead of choosing events at random, we ranked the events by sorting the scores (see simpleFetch, line 11). An item on the list was considered relevant if the user preferred the category of the source and irrelevant otherwise. Figure 6 (a) shows
Figure 6: (a) Receiver Operating Characteristic (ROC) curves shown for a single user, when each user in the system had rated 5 sources, 10 sources, 15 sources, or 30 sources, respectively. For the 5-rates curve, the recommender system performed worse than random. By 30-rates, the system had very high accuracy. The fact that the system performed worse at 15-rates than 10-rates shows early instability. (b) The ROC curve for a novel user to the 30 rates/user system who, himself, had only rated 5 sources (strategically chosen).

ROC curves for a random user in our system for each of the 4 benchmarks. We found several interesting facts from these curves. First, the system at 5 rates/user performed worse than chance. Second, the system performed better at 10 rates/user than 15 rates/user. Finally, it was highly effective and stable by 30 rates/user. Together, this shows that the system suffers instability until enough users have contributed to the system.

As mentioned in Section 2, many recommender systems suffer from the new-user problem[1]. That is, when a novel user enters the system, it is difficult to determine his or her preferences until he or she has submitted many rating indications. We explored our system’s ability to minimize this problem by adding a new user to the 30 rates/user system. This user was given a discrete preference for each categories and he rated one source at random from each category according to these predilections. The user’s ROC curve after these specific 5 rates is presented in figure 6 (b). The auc of this curve was .97 showing very high accuracy. We thus minimized the novel user issue in our system.

8 Conclusions

Efficiently returning quality results from our detailed recommender system and web crawler, WhatsHere delivers a robust system which fits user needs. When data is sparse, the system relies on contextual information, which is then fine-tuned as we gather input from the user. Further, every user in our collaborative scheme helps to focus the results for others, allowing novel users to quickly receive accurate recommendations. WhatsHere provides an integrated system which demonstrates how to effectively delivery quality results to clients from our ever-expanding access to knowledge and data.
Appendix: Functions and Algorithms

Algorithm 1  function descendantAddressCount
1: function descendantAddressCount(WebsiteNode n)
2:   if cache.contains(n) // check for cached value then
3:     return cache.get(n)
4:   end if
5:   if RegularExpAddress.matches(n) // n is an address then
6:     count ← 1
7:   else
8:     count ← 0
9:   end if
10: for each child c of n do
11:   // recurse on each child
12:     count ← count + descendantAddressCount(c)
13: end for
14: cache.add(n,count) // store result
15: return count

Algorithm 2  function searchEventTree
1: function searchEventTree(WebsiteNode n)
2: count ← descendantAddressCount(n)
3: if count = 0 then
4:   // no addresses, do nothing
5: else if count = 1 then
6:   resultSet.add(n)
7: else
8:   // count > 1
9: for each child c of n do
10:   searchEventTree(c)
11: end for
12: end if

// returns likeness of s_1 and s_2
function likeness(source s_1, source s_2)
if alike contains \{s_1, s_2\} then
  return alike \{s_1, s_2\}
else
  return 0
end if
function lookup(location l)

// returns all sources rated by u
function getUserRates(user u)

// adjusts user rating by delta
function updateRating(user u, source s, int delta)

// add node to interval [min, max]
function addNode(object o, int min, int max)

// retrieves object containing interval
function lookupNode(int point)

Algorithm 3 function ratePositive
1: function ratePositive(user u, event e)
2: s ← e.source
3: call updateRating(u,e,1)
4: rates ← getUserRates(u)
5: for each source sx in rates do
6:    like = likeness(s,sx)
7:    value = rates[sx].value
8:    max = getMax(sx)
9:    likeness’ = like + (1 - like)×(value/max)
10:   call updateUserRate(u,sx,likeness’)
11: end for
Algorithm 4 function simplefetch

1: // retrieves 10 weighted results
2: // $\delta$ is a constant about 0.1
3: function simplefetch(user $u$, location $l$)
4:   events ← lookup($l$)
5:   rates ← getUserRates($u$)
6:   running ← 0
7: // determine a score for each discovered event’s source
8: for each source $s$ in events do
9:   sum[$s$] ← 0
10: for each rate $r$ in rates do
11:   sum[$s$] ← sum[$s$] + $r$.val $\times$ likeness($r$.source, $s$) $\times$ $\delta$
12:   end for
13: call addNode($s$, running, running + sum − 1)
14: running ← running + sum
15: end for
16: // choose random events based on sum score
17: List results ← new List()
18: for $i = 1$ to 10 do
19:   rand ← random(max) // random in interval [0, max(sum))
20:   results.add(lookupNode(rand))
21: end for
22: return results
Algorithm 5 function metricfetch
1: // retrieves 10 metric weighted results
2: // $\delta$ is a constant about 0.1
3: function metricfetch(user u, location l)
4: events ← lookup(l)
5: rates ← getUserRates(u)
6: running ← 0
7: eventcountset ← new Set()
8: for each event e in events do
9: s ← e.source
10: sum[s] ← 0
11: for each rate r in rates do
12: sum[s] ← sum[s] + r.val $\times$ likeness(r.source, s) $\times$ $\delta$
13: end for
14: score ← adjustSscoreByMetric(metricFunction(), sum)
15: score ← adjustSscoreByMetric(metricFunction(), score) // ...
16: call addNode(s, running, running + score - 1)
17: running ← running + sum
18: end for
19: List results ← new List()
20: for i = 1 to 10 do
21: rand ← random(max) // random in interval [0, max)
22: results.add(lookupNode(rand))
23: end for
24: return results
Algorithm 6 function quickfetch

1: // retrieves 10 metric weighted results
2: // $\delta$ is a constant about 0.1
3: function quickfetch(user $u$, location $l$)
4: events $\leftarrow$ lookup($l$)
5: rates $\leftarrow$ getUserRates($u$)
6: running $\leftarrow$ 0
7: eventcountset $\leftarrow$ new Set()
8: for each event $e$ in events do
9:     $s \leftarrow e.source$
10:     $sum[s] \leftarrow 0$
11:     // limit to three highest rated sources
12:     for $i$ from 1 to 3 do
13:         $r \leftarrow \max(rates)$
14:         $sum[s] \leftarrow sum[s] + r.val \times \text{likeness}(r.source,s) \times \delta$
15:         rates.remove($r$)
16:     end for
17:     score $\leftarrow \text{adjustSscoreByMetric}(\text{metricFunction}(), sum )$
18:     score $\leftarrow \text{adjustSscoreByMetric}(\text{metricFunction}(), score ) // ...$
19:     call addNode($s$, running, running + score + 1 )
20:     running $\leftarrow$ running + sum
21: end for
22: List results $\leftarrow$ new List()
23: for $i$ = 1 to 10 do
24:     rand $\leftarrow \text{random}($max$)$ // random in interval $[0, \text{max})$
25:     results.add( lookupNode ( rand ) )
26: end for
27: return results

References


