Web Monitoring for Light-Weight Devices

Dissertation

submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Tejaswi. N

(Roll no. 04329016)

under the guidance of

Prof. Soumen Chakrabarti

Kanwal Rekhi School of Information Technology

Indian Institute of Technology Bombay

2006

Abstract

In this report, we consider the problem of multiple lightweight devices monitoring unstructured data on the web. To do this effectively, we propose a 3-tiered architecture for a consolidated system that monitors changes in unstructured web data to cater to these clients (lightweight devices). Typically, our clients work under constrained resources like memory, network bandwidth, processing power, etc. This limits their ability to run applications that require timely and relevant web-data: applications like news readers, offline/online search engines, directories, or general web-page monitoring tools. The proposed system can deliver relevant and timely data to its clients by adapting to their profiles and usage statistics, which it uses to crawl and monitor the web effectively. Our layered architecture includes modules to manage novelty detection, workload, monitoring and scheduling, client profiles, client usage statistics, data packaging and delivery, client connections, etc. In this report, we show how relevant ideas from related literature apply to the monitoring and crawling side of our system. We develop a new way of delivering content to lightwieght devices from the middle layer of our system under bandwidth constraints based on novelty of the content. We investigate various novelty detection algorithms, and how they are used in strategies to choose the best documents from the server.

Contents

List of Figures

Chapter 1

Motivation

Lightweight devices like PDAs, as a part of their design, have been synchronizing data with home computers for some time now. Many such devices are also equipped with technology that enable them to connect to the Internet and download web data. Desktop Internet usage has shown that, simple surfing of web pages is not enough for non-trivial information needs. Tools like search engines, continuous query monitoring tools, etc. are needed to help a user get timely and relevant information. To tailor these tools to lightweight devices, we need to consider their intermittent connectivity with limited data transfer, processing power, memory capacity, and other constraints that are inherent to these devices. To harness the full power of the Internet even under these constraints, we provide these clients with an intermediate server to which they can connect using their on/off Internet connectivity. This server itself, through a "fat" pipe, is connected to the Internet[1] from where it crawls and monitors different sources including news articles, homepages, responses from search engines to continuous queries, RSS feeds, etc. The server then uses its internal architecture to serve each client with relevant pages as per the client's profile and application requirement.

Consider a few motivating scenarios:

- The client's application is an offline search engine, and the user is known to search mostly for cricket statistics in the search engine. In this case, we need to refresh the cricket part of the search engine's index more often so that latest statistics are in the search space.

- Whenever she connects to the Internet, the client's user wants to track breaking

[1]In this report, we use Internet to mean the browsable World Wide Web, web-services, search engines, etc. Other meanings are disambiguated by the context.

news stories only, with some specific news topics given higher priority. In this case, we need to provide novel news pages to the client and filter out other pages that talk about news items she has already seen, or are irrelevant to her. Topical bias can be factored in by serving “favorite” news items first, as soon as connection is established.

- The client specifies a set of pages *a priori*, and wants to track novel changes in these pages as and when they happen. Changes that correct typos, grammatical mistakes, etc. are not considered novel. In this case, we need to monitor the set of pages effectively, detect changes, identify novelty, and as and when it connects, update the client with these changes.

For the above and related scenarios, a dedicated intermediate server with client profiles, usage statistics, processing power to run various IR based algorithms, higher network bandwidth on the Internet side, etc. will add tremendous overall value to all of its clients. If such a server’s features are standardized, client applications can be designed with such servers in mind to harness connectivity and processing powers of both the client and the server effectively.

Chapter 2

Constraints and Costs

In this section, we informally discuss the main components that go into our architecture, what constraints they operate under, and what their cost models are likely to be. This discussion will lead to a more formal architecture that addresses most of these constraints and cost requirements.

Data sources that represent the Internet to us include plain web pages, search engines, directories, news sites, RSS feeds, blogs, etc. Most plain web pages are crawled, sometimes adaptively [1, 2], or with a dedicated theme in mind [3]. Web pages designated *a priori* as important are polled efficiently [4, 5]. Search engines provide web services (like Google API), which can be programmed to. News sites, blogs, and other similar sources provide RSS feeds which can be read using RSS readers. Our system accesses the Internet through crawling, polling, programming to web services, or using RSS readers.

We assume that our server is connected to the Internet through broadband access, and the cost incurred on the Internet end is C_I. C_I is money units per KB, and can also include setup and rental charges. The client side cost, which we denote as C_{C_i}, of the i^{th} client, is also in money units per data bits. Let the consolidated client side cost as seen by the server be C_C. C_C is more sensitive than C_I for the following reasons:

- C_C might include charges made to multiple clients that received same data. This data's C_I component is included only once in the cost analysis, while C_C component, multiple times. Keeping C_C low would counter this phenomenon to some extent.

- C_C and C_Is are being charged for low-bandwidth and high-bandwidth Internet connectivities respectively. With high-bandwidth connectivity, its possible to download significantly more data than with low-bandwidth connectivity. Some of this mismatch in data capacity is offset by the capabilities of the intermediate server which filters out unnecessary data before delivering it to clients. But some of it also *has* to be offset by packaging the client side data intelligently and using connection time effectively, which is C_C.

On the client side, we could have devices like Internet enabled smart-phones, PDAs (Personal Digital Assistant), other desktops which connect to the server, etc. In reality, each of these types of clients has its own cost, connectivity, processing, and storage models based on their service provider, technologies used to build them, etc. But, we model them as abstract clients which follow a single model that includes intermittent connectivity, cost, storage, and processing. With their resources, these clients are capable of running applications that need timely and relevant data from the web. With these applications in mind, clients subscribe to services from the server. These services might involve the client storing its profile and personalization parameters in the server.

In the main server component, which is the core of our system, we will have modules that handle client personalization, usage statistics, workload management, monitoring, adaptive/focused crawling, novelty detection, packaging, connection management, etc. Some of these tasks require high-bandwidth data connectivity to the Internet, which we have already discussed. Some of them also require reasonably high-end processing, memory, and secondary storage. Our server can be hosted on a powerful workstation which can handle such hardware requirements. To scale up the number of clients, we could transparently use a cluster of servers to handle the workload. Clustering servers to handle such load is an area of future work.

Chapter 3

System Architecture

The system has a 3-tier architecture with the Internet, server, and clients making up the 3 tiers, as shown in the following block diagram.

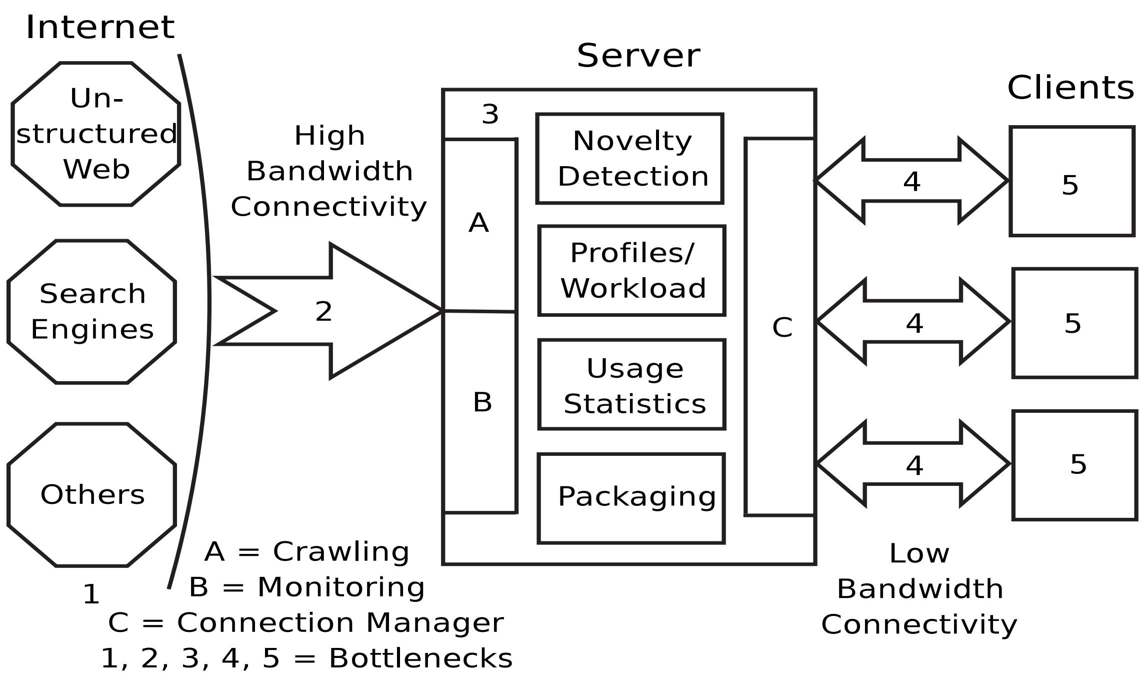

Figure 3.1: System Architecture

For the server architecture, we will consider Tier 1 and Tier 3 as components of the server that handle the Internet and clients respectively. Tier 2 is also resident in the server, but does not directly interact with either the Internet or clients.

3.1 Tier 1 - Internet Modules

Tier 1 is responsible for getting all the relevant data from the Internet in a timely fashion. Data relevancy is as defined by Tier 2. As the server connects to the Internet in multiple ways, we need many "horizontals" in Tier 1: RSS Readers, search engine web services consumers, adaptive/focused crawlers, and monitoring tools.

3.1.1 RSS Readers

News sites, blogs, discussion forums, and many other sources publish their content as RSS feeds over the RSS protocol conforming to the RSS 2.0 specification. The single important advantage of having RSS feeds from fast changing sites over repeatedly crawling/polling them lies in parsing. RSS feeds make it trivial to extract updated content, time, etc. The same extraction task can get non-trivial for crawlers which handle HTML pages. The meta-data that comes with RSS feeds can be used to identify authors, titles, and other details which are hard to find in HTML. Crawlers use heuristics to find update times as they have no way of knowing exact update times. RSS feeds come with explicit update times, and with stored previous update times, its easy to check for new updates without comparing contents.

Unless RSS feed publishers use the "ping" mechanism, the RSS reader needs to check each RSS feed for updates frequently. Though identifying the syntactic update is easy with all the RSS meta-data, polling the RSS feed to minimize time delay between updates and reads is not trivial. This time delay optimization problem is also faced by the unstructured web monitoring tools, and we will discuss it in detail there. Also, though RSS feeds have meta-data to help readers identify components in the update, the update's semantics are hard to identify. A typo correction is a trivial update, whereas an update in the survivor statistics of a natural disaster is an important update. Tier 2 tries to separate the important from the trivial updates that are seen at Tier 1.

3.1.2 Search Engine Web Services

Search engines expose APIs which enable any consumer to tap into their features programmatically. If there are Tier 2 requirements for results to specific static queries, Tier 1 will use these APIs to get the relevant results. Search engines also provide APIs to other services which can be accessed by Tier 1, and whose contents can be used by Tier 2. One important drawback of using these APIs is that search engines have limits on the number of times an API call can be made, and this contributes to bottleneck 1 (identified as “1” in the architecture figure).

3.1.3 Monitoring and Crawling

If Tier 2 needs to monitor updates to specific web pages which are identified *a priori*, we use techniques from the web crawling, monitoring, and continuous query literature. A detailed literature survey of these techniques is given in Chapter 4.

Most of the methods in current monitoring literature try to predict page change probabilities, use it in conjunction with some other metric like embarrassment, obsoleteness, freshness, missed updates, etc. to form an objective function, and then try to optimise it under some hardware/network resource constraints. We conjecture that these predictions can be improved by speculating on *some* page choices during monitoring. The percentage of pages to speculate on, the speculation strategy itself, and its effectiveness need to be studied from both theoretical and experimental settings. Bandit Problems [6] try to model the speculative exploration theory. The limitation of applying Bandit Problems to our monitoring context is the sheer number of bandits we have (web pages to monitor), which overwhelms typical exact Bandit Problem solutions . There is little work done on approximating solutions to scaled up versions of Bandit Problems. This is also left as future work.

Most Tier 2 requirements can be met by the above mentioned “horizontals,” and typically, crawlers are not needed. But under some circumstances they might be needed. In our motivation scenario 1, our client used an offline search feature which required a searchable index on the client device. Though refreshing the index can be solved by a combination of RSS feed reading and monitoring the web, building the initial index might take a focused crawler [3]. Crawlers are Tier 1’s least used “horizontal.” Also, most web servers have politeness requirements which crawlers or monitoring tools’ polling components cannot violate. This makes monitoring and crawling also face bottleneck 1.

We notice that all Tier 1 “horizontals” connect to the Internet using high bandwidth connections. Their cost is relatively easy to calculate and optimize as they are always connected through a single connection. Therefore, bottleneck 2, which appears (refer architecture diagram) in this connectivity, is not as significant as other bottlenecks. In case the server decides to "lazily" use Tier 1's services, the cost model of C_I might become involved. Also, Tier 1 has to handle bottleneck 1. Though bottleneck 1 in itself is not obtrusive, it plays a role in polling scheduling algorithms. This has to be precisely formalized and is left as future work.

3.2 Tier 2 - Internals

Tier 2 is driven by what clients want. Modules inside Tier 2 are motivated towards optimizing what is downloaded from the Internet and what is delivered to the client. Currently, our design contains a profile manager that also handles workloads and personalization, textual novelty detection, a data processing module which works on client logs to generate user statistics, and a packaging module.

Clients initially subscribe to services by registering themselves with the server. Each client has its own profile which contains identification data like hardware details, owner details, importance weight of the client, etc. The key detail stored in the profile is the nature of service the client wants. Services include offline search, news, tracking of specific pages, etc. Along with the feature, the profile also stores the required data needed to service the profile. In case of offline search or news pertaining to a specific topic, the topic keyword/ID needs to be given. In case of monitoring specific web pages, their URLs need to be stored with the profile. The client might issue just a query, and these pertinent URLs can be obtained by search results through Tier 1. This profile information is further augmented by the system with other information as more user statistics is gathered. The user profile can be abstracted into a simple vector of words that she prefers, and these can capture keywords, interested sites, search engine results, etc. effectively.

We also believe that a user's preference changes over the time of a day, and also through the days of a week. Preference for content will also change during flash events that take center stage in news coverage. Regularly though, we believe that a user profile is better modeled by considering the temporal change in preferneces across the day, and across the week. To verify if this notion is valid, we collected 7.5 million records of user logs from Rediff (http://www.rediff.com) corresponding to 150,000 different browsers accessing 30,000 unique URLs on the site for a week's time. We were able to extract the type of

URL accessed, the time of access, and browser from which the access happened. Figure 3.2 gives an indication of topics and access percentages drifting over different time slots in a day. Figure 3.3 gives an idea of how the same happens in weekends over weekdays.

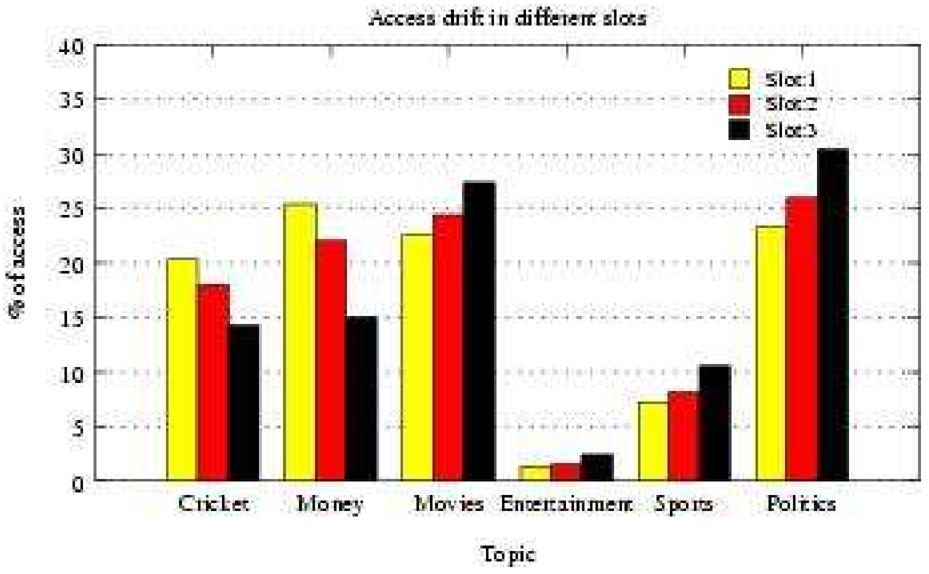

Figure 3.2: Access Drift in Single Day

This leads us to believe that a single profile vector is not good enough to capture a user's behaviour through the day, or through a week. Having 7X3 profile vectors for each slot in each day is a simple solution that would be effective.

It is quite likely that multiple clients might express interest in monitoring common web pages or news topics. A uniform workload of plain web pages to be monitored, news or other feeds to be tracked, etc. need to be built using client profiles. Say a profile wants to monitor result pages for a particular continuous query Q, and another profile wants to get news updates on a related topic R. Q and R are bound to have common pages/feeds which need to be monitored for both profiles. Consolidation of these to-be-monitored pages/feeds based on profiles is a complex problem to solve, as these pages keep changing, user preferences keep evolving, and there is a constant user churn. A quick look up index is needed to store all the pages/feeds being monitored so that exact duplicates can be eliminated quickly.

Semantic duplicate elimination is a tougher problem, which needs to be solved using novelty detection, natural language processing, etc. The page/feed list that needs to be monitored is stored where Tier 1 can access it for its own polling scheduling. These

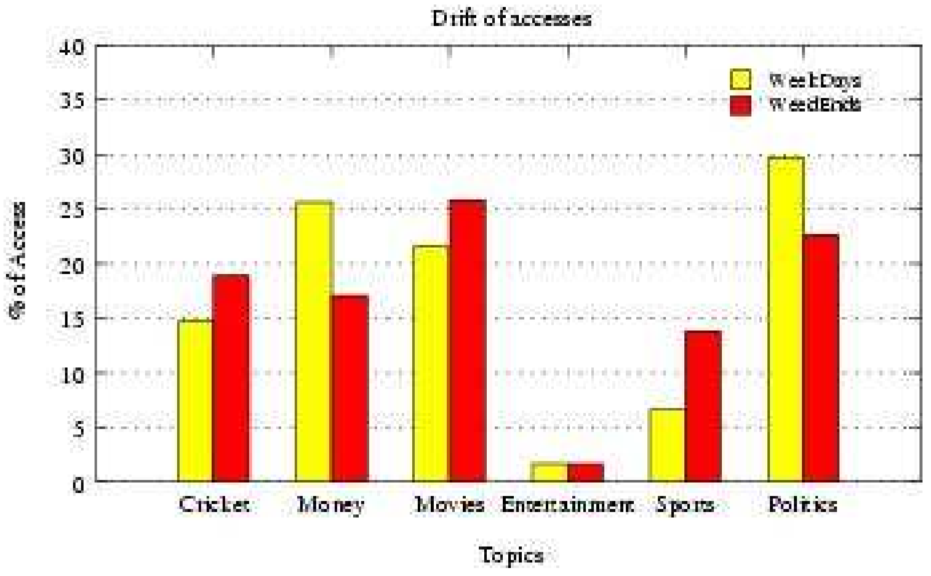

Figure 3.3: Access Drift in Weekends

scheduling algorithms depend on a static list of pages/feeds to monitor. In our case, this list is evolving dynamically as clients are churned in and out of the system, evolving user statistics, pertinent new pages being discovered, etc. This evolving list scheduling problem needs to be theoretically studied further. This is left as future work. For a practical implementation, we can fix specific time quanta only during which the site/feed list is updated. The balancing line is between our desire to have the most recently discovered relevant sites monitored as opposed to efficient monitoring of relevant sites from the past.

Novelty detection is defined as the problem of detecting novel or new content in a time ordered sequence of information. Eg. In a news article sequence describing intermediate results of counting during an election, that article that carries the final results is deemed novel. In the textual domain, James Allan, et al. [7, 8, 9, 10] have contributed a large body of work in this area. In one of their approaches [8], they compare incoming articles to all the old articles and find overlap of named entities and topic-terms (non-named entities), and the maximum overlap values are considered as features. These values are then given as input to a classifier to decide whether the new article is novel or not. The idea is to use novelty detection to filter out "redundant" articles from novel ones, and package and deliver only the novel ones to the client. In case of the offline search index's updation, if the novelty detection module identifies some of the "new" updates to the index's content as not-novel, they need not be updated. Overall, novelty detection at the Tier 2 will ensure that a large amount of unnecessary data is not delivered to the clients, under all possible application scenarios. For each feature offered by Tier 2 to Tier 3, novelty detection needs to be applied by ensuring that novelty is measured only in the client's and the application's context, so that relevant data is not flagged off as not-novel.

In Chapter 5, we design various algorithms to solve the $M : N$ novelty detection problem. We assume that Tier 2 has M documents relevant to a client's profile, and Tier 3 has N documents, and we have to find the best k documents to send over to Tier 3 from Tier 2.

In case the entire client server package suite is being built together, knowing each other's requirements, we can have the clients collect usage data (logs) and report it back to the server whenever it connects. This client to server communication for logs transfer will have some priority in the overall client server communication hierarchy. Client side logs are then processed by the server's user statistics module to find relevant latent information about the client's preferences and satisfaction. Statistics to be analyzed are time spent on each result/article, active feedback, other client activity on results (like forwarding some result to other users), etc. Result quality feedback can be objectively captured using partial ordering between presented results, usage logs, absolute scores given by users, etc. and the overall result generating algorithm can be given some rating based on these user statistics. Also, latent preferences can be discovered by doing simple association rule mining on results that satisfied the client. These preferences also go into client profiles.

Once relevant pages to be sent to each client are identified, they are packaged into chunks whose size and importance are based on each client's connectivity models and importance. The cost and connectivity models of the clients have a heavy bearing on the size of these data chunks.

Bottleneck 3 in Tier 2 is due to the heavy processing required there. Novelty detection, log processing, workload management, etc. are all heavy on storage and processing. Time lost here also needs to be minimized using better indexing techniques, smarter workload merging algorithms, quicker data processing/statistics collection algorithms, etc.

3.3 Tier 3 - Client Modules

As shown in the architecture diagram and alluded to before, Tier 3 is a connection manager. It manages client side connections so that Tier 2 can deliver its content to each client effectively. The challenges faced by the connection manager are mainly due to the intermittent connectivity of clients. These include non-trivial session management, fault tolerance, scheduling of data transfer, etc. Currently, our design does not best handle all the above mentioned factors. Instead, we will be handling them using simple techniques so that we have a complete design ready. Client server session management is handled using traditional session based techniques. Fault tolerance, a major part of the connection manager, is ignored for now. Scheduling of data transfer is based on importance of clients, round-robin, or some other simple scheduling strategy.

Bottleneck 4, which seemingly appears in this tier, is actually a part of Tier 2. The connection manager has to deliver all data that Tier 2 deems client-worthy. Therefore, it is Tier 2's responsibility to minimize the amount of delivered data so that client-server connection cost is minimized. Intermittent connections might cause data to be transfered multiple times to the clients, and this might increase cost in bottleneck 4 in spite of Tier 2 having nothing to do with it. Fault tolerance mechanisms of this sort (repeated sending), have a great impact on the overall cost of the system. Unless fault tolerant schemes are formally specified, we assume that bottleneck 4 is tackled by Tier 2, and not Tier 3. Fault tolerance at Tier 3 is a major area of future work.

Bottleneck 5, which appears in the client devices, is due to clients' internal resource constraints like storage, processing power, etc. These have some bearing on our system. Eg.: the server might overwhelm its clients with data which clients cannot store, or log collection at the client end. Bottleneck 5 is application and client dependent, and has to be handled by both clients and the server. The server sends minimal data to the clients, and this should handle bottleneck 5 from the server end to some extent.

Chapter 4

Upstream to Server - A Literature Survey

Continuous Adaptive Monitoring (CAM) [4] was one of the first monitoring strategies proposed to cater to user specified continuous queries. In this method, a set of pages is identified as being relevant to a particular query. Limited hardware and network resources form the constraints that hold back ideal monitoring of these web-pages. Given these constraints, the problem is to monitor these web-pages so that the least number of their updates are missed. The cost of each missed update is equal to the query specific relevance weight of the page.

CAM has a training (tracking) phase, which is used to build a statistical model of the change dynamics of the pages in question. Training is done in n epochs of length T. During this stage, pages are downloaded and compared against their previous versions, and a Markov chain is built to predict change behaviour of pages in future epochs. Given such behaviour, a monitoring strategy is built for future epochs by solving a discrete, separable, and convex optimisation problem. The objective function of this optimisation problem is the total weighted importance of updates that are *not* reported to the user, and this has to be minimised. The constraints on the problem are the resources of the system.

Formally, it is required to minimise:

$$\sum_{i \in P} (W_i, E_i)$$

where W_i and E_i denote the importance and the expected number of lost changes of the

i^{th} page respectively. P is the total number of pages. E_i can be represented as:

$$E_i = \sum_{j \in U_i} \rho_{i,j}(1 - y_{i,j})$$

Here, $\rho_{i,j}$ denotes the change probability of the i^{th} page in the j^{th} training epoch. $y_{i,j}$ is a boolean value that indicates whether to monitor the i^{th} page in the j^{th} monitoring (testing) epoch or not. Resource constraint is given by:

$$\sum_{i \in P} \sum_{j \in U_i} y_{i,j} = C$$

where C is the total number of monitoring tasks. The full formulation of the optimisation problem is given in [4].

CAM's performance has been measured against the classical *Uniform* and *Proportional* monitoring policies proposed by Cho and Garcia-Molina [2] under varying change probabilities. In each case, CAM is said to perform better. This can mostly be attributed to the aggressive nature in which CAM allocates monitoring resources. The other difference between CAM and [2] is in the objective function being optimised. [2] optimises average freshness of the repository whereas CAM optimises the number of missed updates across epochs. CAM does better because each missed update is penalised, whereas [2], due to its objective function being an average value, loses out on the granularity of updates.

J. L. Wolf et. al. [11] propose a related technique where they minimise the sum of weighted time-average staleness of pages, across all pages. The constraint is the total number of monitoring tasks R. Suppose a page i is monitored x_i number of times during an epoch of length T, the time-average staleness of i is represented as: $A_i(x_i) = a_i(t_{i,1}, \ldots, t_{i,x_i})$ The weight of each page i is obtained based on the *embarrassment* metric. This metric is very specific to the search engine context. A page's weight is proportional to the level of embarrassment a search engine suffers when a user clicks on the page's URL returned during a search query and finds the result not relevant to her query. This has to have happened because the search engine did not have the most recent copy of the page in its repository. It is given by: $w_i = d_i \sum_j \sum_k c_{j,k} b_{i,j,k}$ where $b_{i,j,k}$ denotes the probability that the search engine returns page i in position j on result page number k; $c_{j,k}$ denotes the frequency that a user clicks on a returned page in position j of query result page k; d_i denotes the probability that inspection of a stale page yields an incorrect response w.r.to the query. The overall task now is to minimise the objective function:

$$\sum_{i=1}^{N} w_i A_i(x_i)$$

subjected to the constraints:

$$\sum_{i=1}^{N} x_i = R \tag{4.1}$$

$$x_i \in \{0, \ldots, R\} \tag{4.2}$$

The time-average staleness of a page is obtained in [11] using an expression involving the probability distribution governing page updates. They consider exponential, general, and quasi-deterministic distributions to come up with three different expressions for the time-average staleness of pages. In the quasi-deterministic model, the page update distribution is based on observations of previous crawls. All the three versions of the optimisation problem can be framed as integer programming problems with exact, but slow solutions. Instead, the authors of [11] also propose to use a fast greedy heuristic that solves the problem within a constant bound of the exact solution. For details, refer to [11]. This method is different from CAM in the weight function used. Also, CAM minimises the number of missed updates while [11] minimises the time-average staleness of pages.

Pandey et. al. [5] propose WIC (Web Information Collector), an online algorithm that decides the monitoring schedule based on change probabilities of web-pages, application specific *urgency* functions, and a *life()* parameter that captures probability of changes to web-pages staying over time. Unlike CAM, WIC's uses an *urgency* parameter in its objective function that can be made application specific. WIC also uses a $life_i(j, k)$ parameter which denotes the probability that information made available by a change at time T_j to page P_i remains available at time T_k.

WIC's objective function can be formulated as a non-serial constrained optimisation problem and can be solved using non-serial dynamic programming. In large applications, this turns out to be expensive. WIC's authors propose a greedy algorithm that serves as a 2-approximation and runs in time linear in the number of decision variables. Additionally, the *urgency* parameter can be used to tune timeliness vs. completeness.

Edwards et. al. [1] propose an Adaptive Crawling Strategy that does not associate prior probabilities with page updates. Instead, they use existing crawl statistics to come up with an estimate for the number of obsolete pages in the repository. Initially, all pages being crawled are divided into a set of pre-determined buckets. Given a bucket, prior number of obsolete pages in that bucket, number of pages in that bucket that get crawled in a particular crawl, and the proportion of pages in that bucket that change during that crawl, an estimate of the number of obsolete pages per bucket can be done. The objective function is the number of obsolete pages across all buckets, and the constraints include the network bandwidth, number of new pages added to the repository, number of pages per bucket that get updated, and other integrity conditions. See [1] for details. The optimisation problem ends up being highly nonlinear, and they use large scale nonlinear programming methods to solve it.

Brewington and Cybenko [12] propose a simple and intuitive approach to estimate the change probability distribution of a set of pages. They define *lifetime* of a page to be the time between successive updates, and *age* of a page as the time elapsed between the last update and the next poll/crawl. They assume that the lifetime distribution of a set of pages to be stationary, and we can intuitively see that the two distributions are closely related. The act of observing that age is t units is the same as knowing that the lifetime is no smaller than t units. This indicates that the age PDF $g(t)$ should be proportional to the probability $1 - F(t)$ of a given lifetime exceeding t units, where $F(t)$ is the CDF corresponding to the lifetime PDF: $f(t)$. It can be shown that:

$$g(t) = \frac{1 - F(t)}{\int_0^\infty [1 - F(t)]dt}$$

We can now get the change update probabilities by just having the age distribution, which is easy to get with simple crawl data. Given this expression, and an assumption that page updates follow a memoryless exponential distribution, only page age observations are enough to obtain the full change distribution. [12] also gives methods to estimate change probabilities based on observed lifetimes of pages. They also define and give methods to evaluate a metric of search engine freshness called (α, β)*currency*. A repository is set to be (α, β)*current* if there is an α probability that a randomly selected page from the repository is current with a grace period of β days. Details can be found in the paper.

Gupta et. al. [13] propose monitoring techniques to answer Continuous Multi-data Incoherency bounded Queries (COMIQ). Typically, these queries need to be answered by reading data from multiple dynamically changing sources. To answer them with certain incoherency guarantees, these dynamically changing sources need to be monitored for changes in values that affect the query in question. In [13], the authors create a Discrete

Time Markov Chain for each of the data items involved in answering a query. Transition probabilities between various states of the model are determined using past observations of changes to data values. As these pages are observed more, these transition probabilities are modified to take into account previous transition probabilities and current observations. In this way, the change probabilities of the monitored pages is learnt using a DTMC.

An additional component called *self correction* is applied to the change probabilities to take external factors into account. Assuming j^{th} pull of the i^{th} data item occurred at time t_k and $(j+1)^{th}$ pull at t_l. Suppose at t_l, predicted change was $PC_{i,l}$, and the actual change was $AC_{i,l}$, they calculate the self correction term as:

$$dX_{i,j+1} = \frac{L(AC_{i,l} - PC_{i,l})}{t_l - t_k} + (1-L)dX_{i,j}$$

Here, L is the smoothing constant. $dX_{i,j}$ is added to the predicted change probability of the i^{th} data item.

The optimisation problem solved in [13] differs slightly from the ones in standard monitoring problems. It has an objective function that is based on the incoherency bound and the change probabilities of the data items. Secondly, the data items being monitored, which contribute to the incoherency, and eventually get polled for again, are numeric values.

Pandey et. al. [14] propose User Centric Web Crawling, a method where crawl strategy is directed towards maximising repository *quality*. Informally, quality of a repository is proportional to the expected average usefulness of the search engine. Usefulness of the search engine is estimated using a query load, individual query frequencies, likelihood of user clicks on specific links on search result pages, and the result ranking function. Formal definitions of usefulness and quality can be found in the paper [14]. Intuitively, downloading the latest copy of a web-page should affect the quality of the repository w.r.to a consistent query load and user behaviour. Given standard monitoring resource constraints, the objective is to download/refresh pages in such a way as to maximise repository quality. Under constraints, a page is downloaded based on priorities which are assigned proportional to the difference in repository quality that this particular page causes. Checking for repository quality change per page download/refresh is very expensive as it involves going through the entire query workload and its result set for each page. Heuristics with good approximation guarantees are provided in [14] to handle this expensive estimation.

Chapter 5

Server to Downstream - Novelty Detection

In this chapter, we study the $M : N$ novelty detection problem. We assume that Tier 2 has M documents relavent to the client, and Tier 3 has N documents already sent over to the client. Given such a situation, we come up with strategies based on novelty detection to find the best k documents that can be sent from Tier 2 to Tier 3.

We study the problem in an abstract setting by having a download limit on the number of documents to account for the connectivity and bandwidth restrictions between Tiers 2 and 3. It is also assumed that at Tier 2, the profile manager would have filtered M documents relavent to the client. Tier 2 is also expected to the N documents which this particular client has already seen from the last download iteration.

5.1 Server Document Space Search

Given this setting, ideally, we must be considering all possible subsets of size k generated from the server side document set S, and compare their novelty based ”goodness” to the N documents in the client side document set C. Checking such ”goodness” of all subsets is computationally harder than say, applying a greedy strategy, or searching through the subset-space using more sophisticated techniques.

5.1.1 Exhaustive Search Strategy

We outline an exhaustive search algorithm that enumerates all possible subsets of the server document set S of size k, and attempts to find the best subset that can be downloaded to the client side. Below, the full algorithm is outlined.

```
Input: Server Document Set $S$, Client Document Set $C$, and download limit $k$.
Set of Subsets $SET$ = set of all possible subsets of size $k$ from $S$
foreach set $s$ in $SET$ do
    $novelty = 0$
    foreach document $d$ in $s$ do
        $novelty$ += Novelty($d$, $C$)
        $novelty$ += Novelty($d$, $s \setminus d$)
    end for
    download $s$ with maximum $novelty$ value
end for
```

Figure 5.1: Exhaustive Search Strategy

This algorithm can be made slightly more efficient by caching some of the novelty values, and avoiding some computations. We notice that subsets overlap considerably on the serverside, and novelty values of server documents with respect to client documents are calculted repeatedly for different server side subsets. This can be avoided by caching values that are already calculated.

5.1.2 Greedy Search Strategy

Below, we outline a greedy strategy.

```
Input: Server Document Set $S$, Client Document Set $C$, and download limit $k$.
Repeat $k$ times
    foreach document $d$ in $S$ do
        $novelty$ = Novelty($d$, $C$)
    end for
    download $d$ with maximum $novelty$ value ($C = C \cup d_{maximum\ novelty}$)
end for
```

Figure 5.2: Greedy Search Strategy

Exhaustively going through all subsets of the server document set is computationally expensive. To avoid this, we employ a greedy strategy that attempts to find the most novel document from the server side with respect to the client documents, and downloads that document into the client document set, thereby increasing its size by one. This procedure is repeated k times.

Even the greedy strategy can be made slightly more efficient by caching incremental novelty values of server documents with respect to the current client document set. After the client document set gets the greedily chosen server document, during the next iteration, some computation is avoided due to the earlier caching.

The method Novelty(d, C) used in both algorithms is from the traditional $1 : N$ novelty detection problem. In this report, we investigate the TF-IDF Consine Dissimilarity method and a few language model based methods to find novelty in this $1 : N$ setting.

5.2 TF-IDF Cosine Dissimilarity

In the vector space model, documents are represented as vectors in a multidimensional Euclidean speace. Each axis in this space corresponds to a term. The coordinate of document d in the direction corresponding to term t is determined by two quantities:

Term Frequency TF(d, t): TF of a word is defined as the number of times term t occurs in document d. TF might be scaled to normalize document length by using sum of all frequencies or maximum of all frequencies.

Inverse Document Frequency IDF(t): IDF seeks to scale down the coordinates of terms that occur more freqently across the document set (corpus). In our case, IDF is the log of the ratio of total number of documents to the number of documents that contrain the given term.

TF and IDF are used together in the vector space model to give weights to each coordinate (term) in each vector (document) by using $TF * IDF$ as each term's weight. TF-IDF weighted word vectors of the server document d and each of client documents are generated. Dissimilarity between d and each of C are calculated by taking an inverse measure of their cosine similarity. We pick the maximum value of dissimilarity between d and each of C as the novelty of d with respect to C.

5.3 Language Models

A language model (or more precisely, a statistical language model) is a probabilistic technique for generating text. In the past several years, there has been significant interst in the usage of language modeling methods for a variety of information retrieval and natural language processing tasks. In many cases, like information retrieval, there is often very little training data (short queries), and many competing hypotheses (documents) which are to be sifted through to find the most likely hypothesis. To quote from [15], one of the pioneering papers in language modeling litearture:

> When designing a statistical model for language processing tasks, often the most natural route is to apply a generative model which builds up the output step-by-step. Yet to be effective, such models need to liberally distribute probability mass over a huge space of possible outcomes. This probability can be difficult to control, making an accurate direct model of the distribution of interest difficult to construct. The source channel perspective suggests a different approach: turn the search problem around to predict the input. Far more than a simple application of Bayes' law, there are compelling reasons why reformulating the problem in this way should be rewarding. In speech recognition, natural language processing, and machine translation, researchers have time and again found that predicting what is already known (i.e., the query) from competing hypotheses can be easier than directly predicting all of the hypotheses.

We tailor the language modeling approach to novelty detection with the following idea: We know the client side documents C, and can build a language model around these client side documents. If this model adequately generates a server side document, we deem the server side document as not-novel. So, our approach would look for that document on the server side that is least likely to be probabilistically generated from the language model which is derived from the client side.

All of our language modeling approaches depend on the idea of generating one document from another document's language model. We model the source document using a unigram multinomial term distribution θ_d. This multinomial now gives the generative probability of observing a given vector of term counts, where the probability of generating a term t $p(t|d)$ is subject to $\sum_{t \in d} p(t|d) = 1$ and $\forall t, p(t|d) > 0$. This distribution θ_d can be found by maximum liklihood estimation to be:

$$p(t|d) = \frac{TF(t_i, d)}{\sum_{t_j \in d} TF(t_j, d)} \tag{5.1}$$

The problem with using this approach is that if a word never occurs in the source document, but is still there in the corpus, it's generative probability from that document will be zero, and this will force the entire product of probabilities to go to zero. We use Bayesian smoothing using Dirichlet priors to fix these zero probabilities of unseen words. If the language model being used is multinomial, for which the conjugate prior for Bayesian analysis is the Dirichlet distribution with parameters $(\mu p(t_1|C), \mu p(t_2|C), \ldots, \mu p(t_n|C))$. Thus, the model is given by:

$$p(t|d) = \frac{TF(t_i, d) + \mu p(t_i)}{\sum_{t_j \in d} (TF(t_j|d) + \mu)} \tag{5.2}$$

In our experiments, we set $\mu = 0.5$. Using this multinomial model with smoothing, we define the document generation probability $p(d_i|d_j)$, ie. the probability of generating d_i given d_j:

$$p(d_i|d_j) = \frac{n!}{\prod\limits_{t \in d_i} TF(t, d_i)!} \prod_{t \in d_i} p(t|d_j)^{TF(t,d_i)} \tag{5.3}$$

where $n = \sum\limits_t TF(t, d)$. This represents the core of our language modeling approaches.

5.3.1 Collective Document Generative Model

Novelty of a server document d with respect to all clients documents C is given by how less likely they are to have generated the server document.

$$Novelty(d, C) \propto inverse\left(\frac{p(d|C)}{p(d)}\right) \tag{5.4}$$

Using simple Bayesian reasoning, we get:

$$Novelty(d, C) \propto inverse\left(\prod_{c \in C} p(c|s)\right) \tag{5.5}$$

where $p(c|s)$ can be calculated using Equation 5.3.

5.3.2 Individual Maximum Document Generative Model

In the collective model, we observed that all client documents being considered together in the product. So, if one client document is very similar and not-novel with respect to the server document as compared to other client documents, this effect will be not discerned in the product. To work around this, we define novelty as the probability with which the most novel document on the client side generates the server document. This is calculated by:

$$Novlty(d|C) \propto argmax_{c \in C}(-p(c|d)) \tag{5.6}$$

5.3.3 Average Document Generative Model

Assuming that each client document is equaly likely to have generated the server document, we find the generational probabilities of the server document from each of the client documents and take its average as our novelty value.

$$Novelty(d, C) = -\frac{1}{|C|} \sum_{c \in C} p(d|c) \tag{5.7}$$

5.3.4 Clustering Client Side Documents

Intuitively, we can see that some documents collectively make other documents redundant, and thus, novel documents on the server side are novel not just compared to one document (or all documents) on the client side, but to selected groups of client side documents. We also conjecture that such groups can be identified by clustering the client side documents together on terms which are common to them, and clusters formed thus would be better candidates to check novelty with.

We cluster documents using simple hierarchical clustering using cosine similarity of TFIDF weighted document vectors in the corpus vocaabulary vector space. After such clustering is done, we consider each cluster to be one single client document. These new client documents (formed out of old client document clusters) form our new client document set. Language modeling techniques mentioned above can now be applied in this new client document setting.

5.4 Experimental Results

The two search strategies coupled with various novelty detection approaches were tried out on two data sets. The first data set is TREC based and was created by the authors of [16], and the second is Google News based, which we created by crawling Google News's website over the first few days of June 2006.

5.4.1 TREC data set

Zhang et. al. [16] had human editors mark 3500 TREC documents from the Wall Street Journal and Associated Press feeds as redundant: each of them made redundant by one or more documents from the same data set. Each such redundancy judgement was associated with a human reviewer. Each of the 3500 redundancy tuples was of the format: Reviewer, Redundant Document, List of Redundant Maker Documents. Each tuple had exactly one redundant document, and typically, around one or two redundant maker documents.

To fit this data set for novelty judgements, we assumed that if a document is redundant with respect to other documents, it's not novel. Consider one reviewer's tuples. If there are n tuples, there are n redundant documents and more than n redundant maker documents. We randomly split all these documents into two parts: server and client document sets. We ensure that all the redundant documents are a part of the server document set. Redundant maker documents could be in both client and server part. The rationale behind such a split is that when we run our novelty detection algorithms on the client server split, we want none of the redundant documents from the server side to slip through in case that redundant document's redundant maker documents are already on the client side. If such a document slips through, we penalize it, and lower the accuracy of the download strategy.

In this data set, we cannot estimate *recall* values because we are penalizing redundant documents sent, and not novel documents held back. Instead, we calculate precision at differnet download limits. Like recall, with increasing values of download limit, precision drops.

5.4.2 Google News data set

Google News (*http* : //*news.google.co.in*) collects news articles from more than 4,500 news sources and presents them in easy to download RSS feed format. Every instance of the main Google News page has around 20 to 40 news stories, and each story has around 200-400 clustered news articles under it, collected from various sources. 20 such instances of the Google News page was crawled and all stories at the cluster level were downloaded and stripped of stop words. Around 29,000 documents were downloaded for this data set.

To fit this data set for novelty judgments, we assume that every news story's cluster has just one original story, and the rest of the cluster documents are redundant. This gives us around 1,000 novel documents and their corresponding 28,000 redundant documents. We start off with a mixture of novel documents and redundant documents from multiple clusters on the server side, and run our algorithms on such a client server configuration.

5.4.3 Evaluation

To evaluate our algorithms on the above data sets, we adapt traditional measures of *precision* and *recall* to our setting. Precision is defined as the ratio between the number of novel documents sent to the overall number of documents sent (which include novel and redundant documents). Recall is defined as the ratio of the number of novel documents sent to the number of novel documents present on the server side to begin with.

5.4.4 Results

In figures 5.3 and 5.4, we show experimental results of running greedy strategy with TFIDF weighted vector space cosine difference, cluster, and non-cluster based language model based novelty detection methods. We can clearly observe that the cosine dissimilarity method outperforms the language model based methods in both data sets. This prompted us to run the computationally expensive exhaustive search approach only on the cosine dissimilarity model. In figure 5.5, we show the result of running exhaustive search on the TREC data set, averaged over multiple topics (reviewers).

These results are unexpected and disappointing. We had expected the assymetric, more sophisticated language model based methods to perform better than the symmetric naive cosine dissimiarity method. This result concurs with both [16] and [17]. We believe that this is due to the way novelty is defined in our data sets. In the TREC data set, we used human judged redundancy as the opposite of novelty, and in the Google data set, two or more elements from the same cluster as being redundant with respect to each

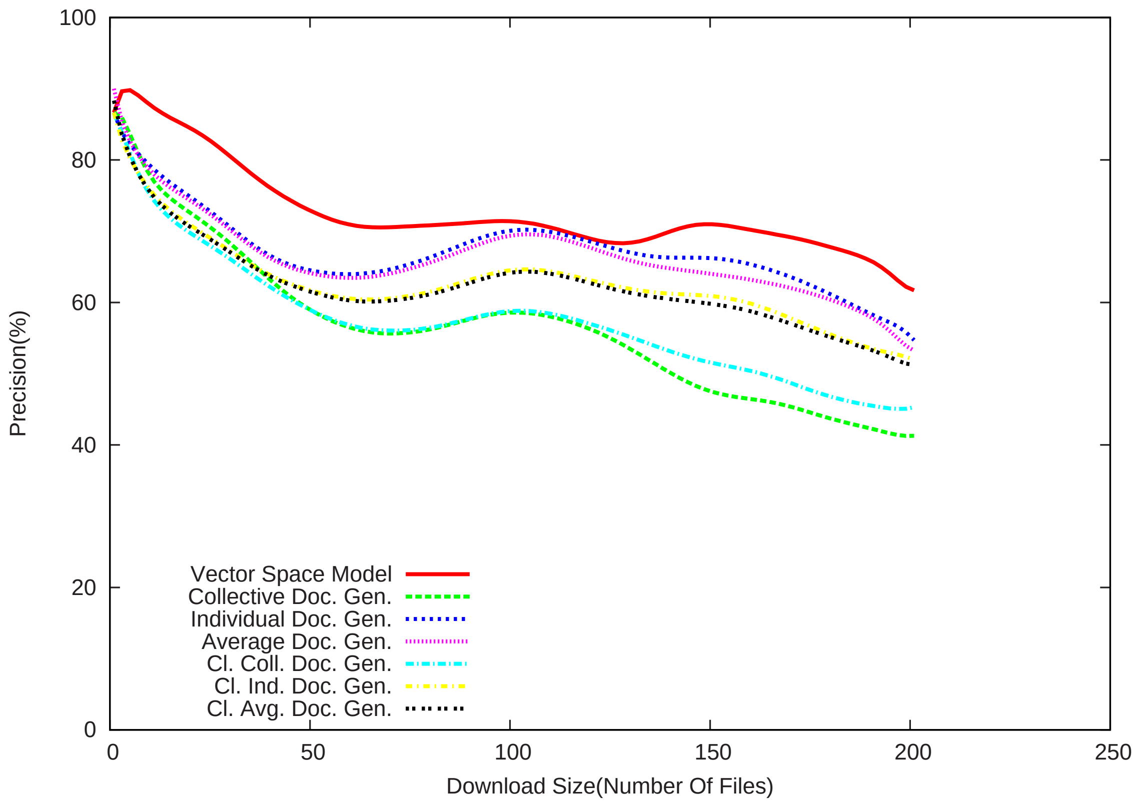

Figure 5.3: Performance on TREC Data (Greedy)

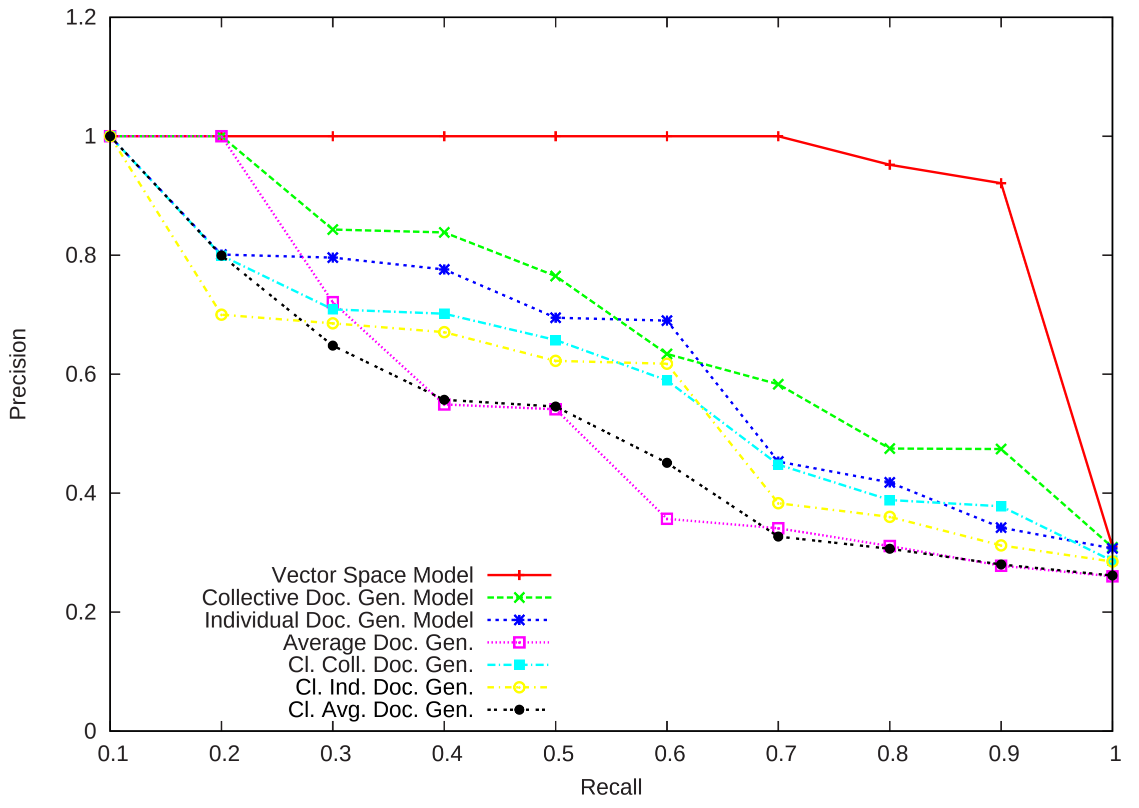

Figure 5.4: Performance on Google News Data (Greedy)

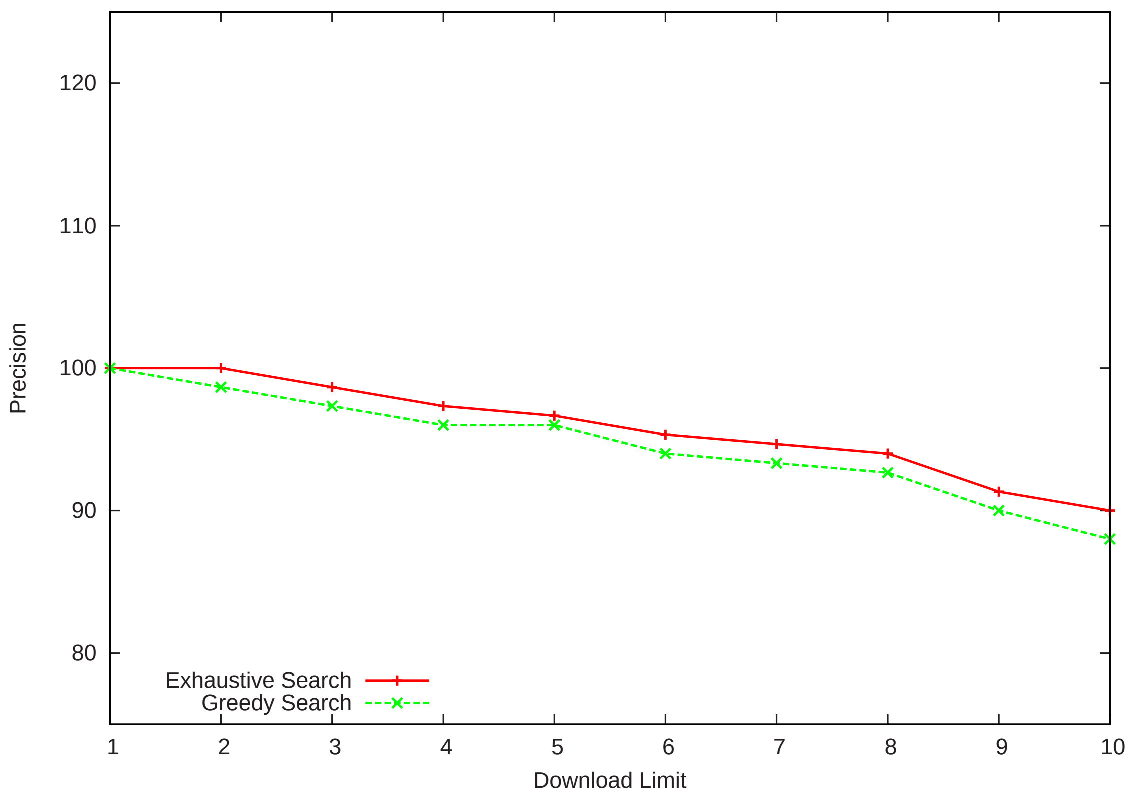

Figure 5.5: Performance on TREC data (Exhaustive, Vector Cosine Diff.

other as the opposite of novelty. These assumptions, though intuitively appealing, might not be correct. We leave further microscopic investigation into this as future work.

Also, we notice considerable difference between the performances of the various language model based methods themselves. This is mostly due to the way documents get combined on the client side in each of the methods. Novelty in the TREC data set, for example, as judged by humans, is based on multiple documents rendering one document redundant. These multiple documents need not contribute in their entirety to the novelty judgement. Our language modeling methods, though, consider all these documents to be atomic units (made up of terms). A more detailed investigation into the language modeling approaches is warranted.

In fig 5.5, we notice that, under the vector dissimilarity method, the greedy search and exhaustive search methods have very similar performances. Based on these initial results, given that exhaustive search is not doing considerably better than greedy search, we shelved our efforts to find a more sophisticated search strategy (A-star, branch and bound) in the server set space.

Chapter 6

Future Work and Conclusion

Future work on this project can be split into two parts: Short and long term goals.

6.1 Short Term Goals

In novelty detection, we did not go beyond looking at documents as atomic units of information. Segmentation of documents into chunks of coherent information and merging of documents based on more sophisticated techniques are both areas of future work that might result in better performance.

Also, in our language model approaches to novelty detection, we have just used Bayesian smoothing with Dirichlet priors. In their study of various smoothing approaches, Zhai and Lafferty [18] elaborate on various other techniques that are unexplored under our problem setting and data sets. Trying out various other smoothing techniques is left as an area of future work that might result in better performance.

Other than greedy and exhaustive search strategies, there are other sophisticated search techniques like A-Star, and other Branch and Bound based algorithms that might give us the accuracy and computational performance comparable to exhaustive and greedy search strategies respectively. This is also left as an area of future work.

6.2 Long Term Goals

We want to get a sound theoretical understanding of the following concepts, implement them, and test them on real life data sets.

- In the Novelty Detection scenario, using overlaps in repositories and incoming online pages to build efficient ND models.

- In the Bandit Problem scenario, getting approximate solutions with large number of bandits as opposed to exact solutions with a small number of bandits.
- With evolving preference/article/webpage/feed lists, designing optimal/approximate scheduling and monitoring algorithms.
- Factoring in web-server and search engine politeness requirements into scheduling and monitoring algorithms.
- Fault tolerance models in an intermittent client-server connection scenario.
- Formalizing user satisfaction fully in our problem/system context.
- Clustering of servers in case the number of clients and workload scales up.

6.3 Conclusion

In this report, we have elaborated on the design of a system that allows lightweight devices to subscribe to it and harness the information content of the Internet effectively. We have given a detailed literature survey on existing techniques that can be employed in Tiers 1-2 transition of our architecture. We have also conceptualized and implemented techniques to handle Tiers 2-3 transition. If such a system is implemented fully, lightweight devices, even under their resource constraints, can provide their users a large window into the Internet.

Bibliography

[1] Jenny Edwards, Kevin McCurley, and John Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 106–113, New York, NY, USA, 2001. ACM Press.

[2] Junghoo Cho and Hector Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 2003.

[3] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific Web resource discovery. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1623–1640, 1999.

[4] Sandeep Pandey, Krithi Ramamritham, and Soumen Chakrabarti. Monitoring the dynamic web to respond to continuous queries. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[5] Sandeep Pandey, Kedar Dhamdhere, and Chris Olston. Wic: A general-purpose algorithm for monitoring web information sources. In *Proceedings of the 30th International Conference on Very Large Data Bases*, 2004.

[6] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[7] James Allan, Ron Papka, and Victor Lavrenko. On-line new event detection and tracking. In *Research and Development in Information Retrieval*, pages 37–45, 1998.

[8] Giridhar Kumaran, James Allan, and Andrew McCallu. Classification models for new event detection. Technical report, Center for Intelligent Information Retrieval, University of Massachusetts.

[9] Giridhar Kumaran and James Allan. Text classification and named entities for new event detection. ACM SIGIR, 2004.

[10] Giridhar Kumaran and James Allan. Using names and topics for new event detection. Conference on Empirical Methods in Natural Language Processing, 2005.

[11] J. L. Wolf, M. S. Squillante, P. S. Yu, J. Sethuraman, and L. Ozsen. Optimal crawling strategies for web-search engines. In *Proceedings of the World Wide Web Conference*, 2002.

[12] Brian E. Brewington and George Cybenko. How dynamic is the Web? *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):257–276, 2000.

[13] Rajeev Gupta, Ashish Puri, and Krithi Ramamritham. Executing incoherency bounded continuous queries at web data aggregators. In *Proceedings of the World Wide Web Conference*, 2005.

[14] Sandeep Pandey and Chris Olston. User-centric web crawling. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[15] Adam Berger and John D. Lafferty. Information retrieval as statistical translation. In *Research and Development in Information Retrieval*, pages 222–229, 1999.

[16] Y. Zhang, J. Callan, and T. Minka. Novelty and redundancy detection in adaptive filtering. 2002.

[17] J. Allan, C. Wade, and A. Bolivar. Retrieval and novelty detection at the sentence level. 2003.

[18] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.

Acknowledgements

Prof. Soumen Chakrabarti, my guide, had great ideas, valuable insights & technical guidance, and constructive criticism through every stage of this project. This is invaluable to the project itself, and more so, my personal growth as a budding researcher. My sincerest thanks to him. I hope I get to work with him sometime in the future.

Prof. Krithi Ramamritham, my co-guide on this project brought along his wealth of experience and research insight into the project. His suggestions and advice proved very valuable through the project.

Also, I would like to thank Ahmed Shareef for the relentless coding and thinking efforts that he put in with me. I would also like to thank Vivekananda Tadala for his suggestions and help through the project.

My close friends: the h5-76 gang (Jatin, Nishant, Chotu, Amar, Ani, Kantupatil, and Begum) and Kriti made my stay in IIT Bombay incredible fun. I could not have done without them.

Tejaswi N
I. I. T. Bombay
July 9th, 2006