Chapter 4

Web Request Prediction

4.1 Introduction

Modeling user activities on the Web has value for both content providers and consumers. Consumers may appreciate better responsiveness as a result of precalculating and preloading of content in advance of their requests. Additionally, consumers may find adaptive and personalized Web sites that can make suggestions and improve navigation useful. Likewise, the content provider will appreciate the insights that modeling can provide and the potential financial implications of a happier consumer that gets the information desired even faster.

Sequence prediction (that is, predicting the next item in a sequence) is fundamental to prefetching, whether for memory references, or disk accesses, or Web requests. Without accurate prediction, prefetching is not worthwhile. Therefore, it is useful to consider the various methods for prediction, even before embedding them within a prefetching system. For this reason we have introduced the simple IPAM prediction method in Chapter 3, and examine the topic in more depth here. Later, in Chapters 8 and 9 we will embed these prediction mechanisms within a simulator to estimate user-perceived response times, rather than just predictive accuracy.

Sequence prediction in general is a well-studied problem, particularly within the data compression field (e.g., [BCW90, CKV93, WMB99]). Unfortunately, the Web community has re-discovered many of these techniques, leading to islands of similar work with dissimilar vocabulary. Here we will both re-examine these techniques as well as offer modifications motivated by the Web domain. This chapter will describe, implement, and experimentally evaluate a number of methods to model usage and predict Web requests. Our focus will be on Markov-based and Markov-like probabilistic

techniques, both for their predictive power, but also their popularity in Web modeling and other domains.

We are interested in applying prediction to various types of Web workloads – those seen by clients, proxies, and servers. Each location provides a different view of Web activities, and the context in which they occur. As a result, different levels of performance will be possible.

This chapter will provide:

- an enumeration and discussion of the many aspects of how, what, and why to evaluate Web sequence prediction algorithms,
- a consistent description of various algorithms (often independently proposed) as just variations and simplifications of each other,
- the implementation of a prediction program sufficiently generalized to implement a variety of techniques,
- a comparison of multiple algorithms on a single set of traces to facilitate comparison,
- evidence of some "concept drift" within Web traces (similar to what we found in Chapter 3), and
- three Web domain-inspired improvements to existing algorithms.

In the next section, we will detail the many concerns and approaches we will take to the empirical evaluation of various Web request prediction algorithms. In Section 4.3 we describe the various algorithms and their extensions as we have implemented them. In Section 4.4 we describe the workloads used and present experimental results in Section 4.5. Finally, Sections 4.6 and 4.7 discuss and summarize the findings of this chapter.

4.2 Evaluation concerns and approaches

In this section we discuss the various aspects of how and what to evaluate when we compare Web request prediction algorithms. Two high-level concerns that are addressed repeatedly are the questions of 1) whether to modify typical evaluation approaches to better fit the domain, and 2) whether to modify predictions to better fit the domain, or both.

4.2.1 Type of Web logs used

One important aspect of any experimental work is the data sets used in the experiments. While we will introduce the Web workloads that we will use in Section 4.4, the type of workload is an evaluation concern. At a high level, we are simply concerned with methods that learn models of typical Web usage. However, at a lower level, those models are often simply identifying co-occurrences among resources — with the ultimate goal to make accurate predictions for resources that might be requested given a particular context.

However, there are multiple types of relationships between Web resources that might cause recognizable co-occurrences in Web logs [Bes96, CJ97, Cun97]. One possible relationship is that of an embedded object and its referring page — an object, such as an image, audio, or Java applet that is automatically retrieved by the browser when the referring page is rendered. Another relationship is that of traversal — when a user clicks on a link from the referring page to another page. The first (embedding) is solely an aspect of how the content was prepared. The second (traversal), while likewise existing because of a link placed by the content creator, is also a function of how users navigate through the Web hypertext.

Many researchers (see, for example [JK97, JK98, NZA98, Pal98, ZAN99, SYLZ00, SH02, YZL01]) distinguish between such relationships and choose not to make predictions for embedded resources. They are concerned with "click-stream" analysis — just the sequence of requests made by the user and not the set of automatically requested additional resources. Sometimes the data can provide the distinction for us — a Web

server often records the referrer in its access logs, which captures the traversal relationship. But the HTTP referrer header is not a required field, and is thus not always available. In other cases, analysis of the data is required to label the kind of request — for example, requests for embedded links are usually highly concentrated in time near the referring page. Unfortunately, many of the publicly available access logs do not provide sufficient detail to allow us to make such distinctions authoritatively. In addition, methods that use click-stream data exclusively will require a more complex implementation, as they assume that the embedded resources will be prefetched automatically, which requires parsing and may miss resource retrievals that are not easily parsed (such as those that result from JavaScript or Java execution). As a result, in this chapter we have chosen to disregard the type of content and the question of whether it was an embedded resource or not, and instead use all logged requests as data for training and prediction. This approach is also taken by Bestavros *et al.* [Bes96, Bes95], and Fan *et al.* [FJCL99].

4.2.2 Per-user or per-request averaging

As described in Chapter 3, one can calculate average performance on a per-user (macroaverage) or per-request (microaverage) basis. Even though predictions are made on a per-user basis (i.e., on the basis of that user's previous action which is not necessarily the most recent action in the system), we don't always build per-user models. In our experiments, we will build single models (for the typical user) for Web servers and proxies, and only consider per-user models when making predictions at the client. Thus for comparison, we will generally report only per-request averages.

4.2.3 User request sessions

A session is a period of sustained Web activity by a user. In most traces, users have request activities that could be broken into sessions. In fact, during the month of December 1999, the median number of sessions per user was four (where a session boundary is defined by two or more hours without Web access), and has been growing at a rate of 1.4% per month, according to data from 20,000 Internet users collected by

Jupiter Media Metrix [MF01]. In practice, it may be helpful to mark session boundaries to learn when not to prefetch, but alternately, it may be desirable to prefetch the first page that the user will request at the beginning of the next session. We have not analyzed the data sets for sessions — for the purposes of this chapter, each is treated like a set of per-user strings of tokens. Thus, even though a Web log contains interleaved requests by many clients, our algorithms will consider each prediction for a client solely in the context of the requests made by the same client. In addition, it does not matter how much time has passed since the previous request by the same client, nor what the actual request was. If the actual request made matches the predicted request, it is considered a success.

4.2.4 Batch versus online evaluation

The traditional approach to machine learning [Mit97] evaluation is the batch approach, in which data sets are separated into distinct training and test sets. The algorithm attempts to determine the appropriate concept by learning from the training set. The model is then used statically on the test set to evaluate its performance. This approach is used in a number of Web prediction papers (e.g., [ZAN99, AZN99, NZA98, Sar00, SYLZ00, YZL01, ZY01]). While we can certainly do the same (and will do so in one case for comparison), our normal approach will be to apply the predictive algorithms incrementally, in which each action serves to update the current user model and assist in making a prediction for the next action. This matches the approach taken in other Web prediction papers (e.g., [Pad95, PM96, FJCL99, Pal98, PM99]). This model is arguably more realistic in that it matches the expected implementation — a system that learns from the past to improve its predictions in the future. Similarly, under this approach we can test the code against all actions (not just a fraction assigned to be a test set), with the caveat that performance is likely to be poor initially before the user model has acquired much knowledge (although some use an initial warming phase in which evaluation is not performed to alleviate this effect).

When using the prediction system in a simulated or actual system, note that the predictions may cause the user's actual or perceived behavior to change. In prefetching,

the user may request the next document faster if there was no delay in fetching the first (because it was preloaded into a cache). Likewise, a proxy- or server-based model that sends hints about what to prefetch or content to the browser will change the reference stream that comes out of the browser, since the contents of the cache will have changed. Thus, the predictive system may have to adjust itself to the change in activity that was caused by its operation. Alternatively, with appropriate HTTP extensions, the browser could tell the server about requests served by the cache (as suggested in [FJCL99, Duc99]). In future chapters we will incorporate caching effects, and so we will limit ourselves here to models that are built from the same requests as those which are used for evaluation. This model is appropriate for prefetching clients and proxies, as long as they don't depend on server hints (built with additional data).

4.2.5 Selecting evaluation data

Even when using an online per-user predictive model, it will be impossible for a proxy to know when to "predict" the first request, since the client had not connected previously. Note that if the predictions are used for server-side optimization, then a generic prefetch of the most likely first request for any user may be helpful and feasible, and similarly for clients a generic prefetch of the likely first request can be helpful. Likewise, we cannot test predictions made after the user's last request in our sample trace. Thus, the question of which data to use for evaluation comes up. While we will track performance along many of these metrics, we will generally plot performance on the broadest metric — the number of correct predictions out of the total number of actions. This is potentially an underestimate of performance, as the first requests and the unique requests (that is, those requests that are never repeated) are counted, and may become less of a factor over time. If we were to break the data into per-user sessions, this would be a larger factor as there would be more first and last requests that must be handled.

4.2.6 Confidence and support

In most real-world implementation scenarios, there is some cost for each prediction made. For example, the cost can be cognitive if the predictions generate increased

cognitive load in a user interface. Or the cost can be financial, as in prefetching when there is a cost per byte retrieved. Thus, we may wish to consider exactly when we wish to make a prediction — in other words, to not take a guess at every opportunity.

We consider two mechanisms to reduce or limit the likelihood of making a false prediction. They are:

- Thresholds on confidence. Confidence is loosely defined as the probability that the predicted action will occur, and is typically based on the fraction of the number of times that the predicted action occurred in this context in the past. Our methods use probabilistic predictors, and thus each possible prediction has what can be considered an associated probability. By enforcing a minimum threshold, we can restrict the predictions to those that have high expected probability of being correct. Thresholds of this type have been used previously (e.g., [Pad95, PM96, LBO99, DK01]).

- Thresholds on support. Support is strictly the number of times that the predicted action has occurred in this context. Even when a prediction probability (i.e., confidence) is high, that value could be based on only a small number of examples. By providing a minimum support, we can limit predictions to those that have had sufficient experience to warrant a good prediction [JK97, JK98, SKS98, LBO99, FJCL99, PP99b, SYLZ00, DK01, YZL01].[1]

In fielded systems, these two factors may be combined with some function. However, for our studies it is useful to examine them separately.

4.2.7 Calculating precision

Given the ability to place minimum thresholds on confidence and support, the system may choose not to make a prediction at all. Thus, in addition to overall accuracy (correct predictions / all actions), we will also calculate precision — the accuracy of the predictions when predictions are made. However, the exact selection of the denominator

[1] In fact, Su *et al.* [SYLZ00] go further, requiring thresholds not only of page popularity (i.e., support), but also ignoring sequences below some minimum length.

can be uncertain. We note at least two choices: those actions for which a prediction was attempted, and those actions against which a prediction was compared. The former includes predictions for requests that were never received (i.e., made beyond the last request received from a client). Since we cannot judge such predictions, when reporting precision, we will use the latter definition.

4.2.8 Top-n predictions

All of the variations we have described above provide probabilities to determine a prediction. Typically there are multiple predictions possible, with varying confidence and support. Since it may be useful (and feasible) to prefetch more than one object simultaneously, we will explore the costs and benefits of various sets of predictions. As long as additional predictions still exceed minimum confidence and support values, we will generate them, up to some maximum prediction list length of n. The top-n predictions can all be used for prediction and prefetching, and, depending on the evaluation metric, it may not matter which one is successful, as long as one of the n predictions is chosen by the user.

In some systems (e.g., [FJCL99]), there are no direct limits to the number of predictions made. Instead, effective limits are achieved by thresholds on confidence or support, or by available transmission time when embedded in a prefetching system. In this chapter we do not directly limit the number of predictions, but instead consider various threshold values. Limits are typically needed to trade-off resources expended versus benefits gained, and that trade-off depends on the environment within which the predictions are being made. Later, in Chapters 8 and 9 we will embed this prediction system within a simulated environment and will then need to specifically limit the predictions (prefetches) performed because of resource constraints.

4.3 Prediction Techniques

n-grams

Typically the term *sequence* is used to describe an ordered set of actions (Web requests, in this case). Another name, from statistical natural language processing, for the same ordered set is an n-gram. Thus, an n-gram is a sequence of n items. For example, the ordered pair (A, B) is an example 2-gram (or bigram) in which A appeared first, followed by B.

For prediction, we would match the complete prefix of length $n-1$ (i.e., the current context) to an n-gram, and use the n-gram to predict the last item contained in it. Since there may be multiple n-grams with the same prefix of $n-1$ actions, a mechanism is needed to decide which n-gram should be used for prediction. Markov models provide that means, by tracking the likelihood of each n-gram, but using a slightly different perspective — that of a state space encoding the past. In this approach, we explicitly make the Markov assumption which says that the next action is a function strictly of the current state. In a k-step Markov model, then, each state represents the sequence of k previous actions (the context), and has probabilities on the transitions to each of the next possible states. Each state corresponds to the sequence of $k = n - 1$ actions in a particular n-gram. There are at most $|a|^k$ states in such a system (where $|a|$ is the number of possible actions).

Typically k is fixed and is often small in practice to limit the space cost of representing the states. In our system we allow k to be of arbitrary size, but in practice we will typically use relatively small values (primarily because long sequences are infrequently repeated on the Web).

Markov Trees

To implement various sequence prediction methods, a highly-parameterized prediction system was implemented. We use a data structure that works for the more complex algorithms, and just ignore some aspects of it for the simpler ones.

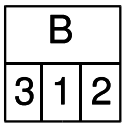
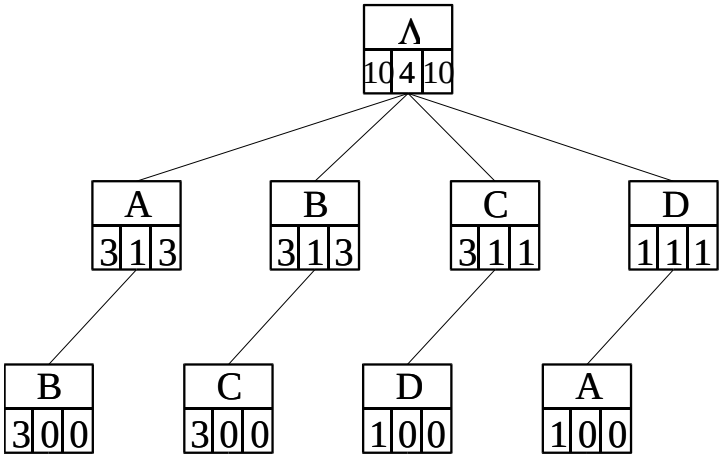


Figure 4.1: A sample node in a Markov tree.

In particular, we build a Markov tree [SSM87, LS94, FX00] — in which the transitions from the root node to its children represent the probabilities in a zero-th order Markov model, the transitions to their children correspond to a first order model, and so on. The tree itself thus stores sequences in the form of a trie — a data structure that stores elements in a tree, where the path from the root to the leaf is described by the key (the sequence, in this case). A description of an individual node is shown in Figure 4.1.

Figure 4.2 depicts an example Markov tree of depth two with the information we record after having two visitors with the given request sequences. The root node corresponds to a sequence without context (that is, when nothing has come before). Thus, a zero-th order Markov model interpretation would find the naive probability of an item in the sequence to be .3, .3, .3, and .1, for items A, B, C, and D, respectively. Given a context of B, the probability of a C following in this model is 1. These probabilities are calculated from the node's *SelfCount* divided by the parent's *ChildCounts*.

To make this process clear, we provide pseudocode to build a Markov tree in Figure 4.3, and Figure 4.4 illustrates one step in that process. Given the sequence (A, B, A), we describe the steps taken to update the tree in 4.4a to get the tree 4.4b. All of the suffixes of this sequence will be used, starting with the empty sequence. Given a sequence of length zero, we go to the root and increment *SelfCount*. Given next the sequence of length one, we start at the root and update its *ChildCounts*, and since there is already a child A, update that node's *SelfCount*. Given next the sequence of length two, we start again from the root, travel to child B, and update it's *SelfCount* and find that we need to add a new child. Thus we also update B's *NumChildren*, and its *ChildCounts* as we add the new node. Assuming we are limiting this tree to depth



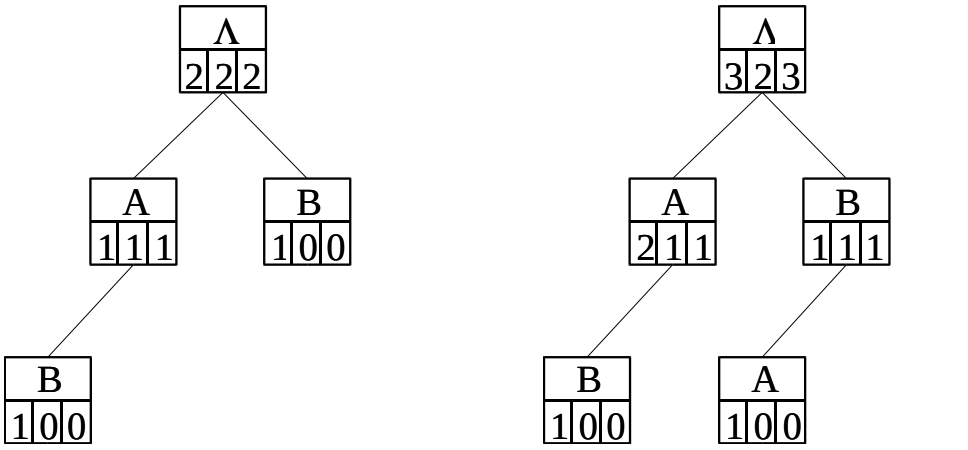
Sequences from two different sessions: (A, B, C, D, A, B, C) followed by (A, B, C).

Figure 4.2: A sample trace and simple Markov tree of depth two built from it.

Given a sequence s, a MaxTreeDepth, and an initial tree (possibly just a root node) t:

```
for i from 0 to min(|s|,MaxTreeDepth)
    let ss be the subsequence containing the last i items from s
    let p be a pointer to t
    if —ss— = 0
      increment p.SelfCount
    else
      for j from first(ss) to last(ss)
         increment p.ChildCounts
         if not-exists-child(p,j)
           increment p.NumChildren
           add a new node for j to the list of p's children
         end if
         let p point to child j
         if j = last(ss)
           increment p.SelfCount
         end if
      end for
    end if
end for
```

Figure 4.3: Pseudocode to build a Markov tree.



(a) Markov tree from sequence (A, B).

(b) Markov tree from sequence (A, B, A).

Figure 4.4: Before and after incrementally updating a simple Markov tree.

two, we have finished the process, and have (b).

In effect, we have built essentially what Laird and Saul [Lai92, LS94] call a TDAG (Transition Directed Acyclic Graph). Like them, we limit the depth of the tree explicitly and limit the number of predictions we make at once. In contrast, they additionally have a mechanism to limit the expansion of a tree by eliminating nodes in the graph that are rarely visited.

Path and point profiles

Thus, after building a Markov tree of sufficient depth, we can use it to match sequences for prediction. Schechter *et al.* [SKS98] describes a predictive approach which is effectively a Markov tree similar to what we have described above. The authors call this approach using *path profiles*, a name borrowed from techniques used in compiler optimization, as contrasted with *point profiles* which are simply bigrams (first order Markov models). The longest path profile matching the current context is used for prediction, with frequency of occurrence used to select from equal-length profiles.

k-th order Markov models

Our Markov tree can in general be used to find k-th order Markov probabilities by traversing the tree from the root in the order of the k items in the current context. Many researchers (e.g., [Pad95, PM96, Bes96, Bes95, JK97, JK98, NZA98, LBO99, Duc99, SR00, FS00, ZY01]) have used models equivalent to first order Markov models (corresponding to trees like that depicted in 4.2) for Web request prediction, but they are also used in many other domains (e.g., the UNIX command prediction from Chapter 3 and hardware-based memory address prefetching [JG99]). Others have found that second order Markov models give better predictive accuracy [SH02, ZAN99], and some, even higher order models (e.g., fourth-order [PP99a]).

During the use of a non-trivial Markov model of a particular order, it is likely that there will be instances in which the current context is not found in the model. Examples of this include a context shorter than the order of the model, or contexts that have introduced a new item into the known alphabet (that is, the set of actions seen so far). Earlier we mentioned the use of the longest matching sequence (path profile) for prediction. The same approach can be taken with Markov models. Given enough data, Markov models of high order typically provide high accuracy, and so using the largest one with a matching context is commonly the approach taken. Both Pitkow and Pirolli [PP99b] and Deshpande and Karypis [DK01] take this route, but also consider variations that prune the tree to reduce space and time needed for prediction (e.g., to implement thresholds on confidence and support, testing on a validation set, and minimum differences in confidence between first and second most likely predictions). Our code allows this approach, and in general provides for a minimum and maximum n-gram length for prediction. Su *et al.* also combine multiple higher-order n-gram models in a similar manner. Interestingly, Li *et al.* [LYW01] argue in contrast that the longest match is not always the best and provide a pessimistic selection method (based on Quinlan's pessimistic error estimate [Qui93]) to choose the context with the highest pessimistic confidence of all applicable contexts, regardless of context length. They show that this approach improves precision as the context gets larger.

PPM

There are, of course, other ways to incorporate context into the prediction mode. PPM, or prediction by partial matching [BCW90, WMB99], is typically used as a powerful method for data compression. It works similarly to the simple Markov model-based approach above, using the largest permissible context to encode the probabilities of the next item. However, it also tracks the probability of the next item to be something that has never been seen before in this context (called the "escape" symbol when used for compression), and thus explicitly says to use the next smaller matching context instead. There are multiple versions of PPM that correspond to different ways of calculating the escape probability: PPM-A, PPM-C, PPM-D, as well as others that we do not consider. Since the next item in the sequence could (and is) given some probability at each of the levels below the longest matching context, we have to examine all matching contexts to sum the probabilities for each candidate prediction (appropriately weighted by the preceding level's escape probability). We do this by merging the current set of predictions with those from the shorter context by multiplying those probabilities from the shorter context by the escape symbol confidence (e) in the longer context, and multiplying those in the longer context by (1 - e). We can calculate various escape probabilities using the counts stored in our Markov tree. A few researchers have used PPM-based models for Web prediction (e.g., [FJCL99, Pal98, PM99]). Actually Fan *et al.* go further, building Markov trees with larger contexts. Instead of using large contexts (e.g., order n) for prediction, they use a context smaller than n (say, m), and use the remaining portion of the tree to make predictions of requests up to $n - m$ steps in advance.

Recency

As we discussed in Chapter 3, the patterns of activity of a user can change over time. The same is true of Web users. As the user's interests change, the activity recorded may also change — there is no need to revisit a site if the user remembers the content, or if the user now has a different concern. Thus, the users themselves provide a source

of change over time in the patterns found in usage logs. There is a second source of change that may be even more significant — changes in content. A user's actions (e.g., visiting a page) depend upon the content of those pages. If the pages change content or links, or are removed, then the request stream generated by the user will also change. Thus we speculate that as long as Web access is primarily an information gathering process, server-side changes will drive changes in user access patterns.

We can attempt to quantify the changes found in some Web logs by using a variant of the IPAM algorithm described in Chapter 3. Recall that it used an *alpha* parameter to emphasize recent activity. However, for processing efficiency, we cannot update all transition probabilities for a given state on each visit. Instead, here we use a simplification: before incrementing the count for a new action, on every $\frac{1}{(1-\text{alpha})}$ visits, we halve the counts of all actions that have been recorded from the previous action. So for example, an alpha of .99 would halve the counts on every 100th visit. Thus we still maintain the property of emphasizing recent events, and in addition, we can bound the maximum count value to $\frac{2}{(1-\text{alpha})}$. This is helpful as it allows us to limit the storage needed to represent the values. Other approaches have been used to emphasize recent activity. Duchamp [Duc99], for example, keeps track of the next page requested by the most recent fifty visitors to a page, thus limiting the effect that a no-longer-popular page sequence can have, but unrealistically requiring cooperation from browsers to self-report link selections.

Additional parameters

In addition to the varieties described above, the prediction system can also vary a number of other parameters that are motivated by the Web domain:

- **Increasing the prediction window** (that is, the window of actions against which the prediction is tested). Typically evaluation is performed by measuring the accuracy of predicting the next request. Instead of only predicting the very next request, we can measure the accuracy when the prediction can match any of the next n requests. This may be useful in matching the utility of preloading a cache as the resource may be useful later.

- **De-emphasizing objects likely to be already cached.** Since the browser has a cache, it is unlikely to make requests for recently requested resources. Thus, the system can reduce the likelihood of predicting recently requested resources.

- **Retaining past predictions.** Likewise, we know that users may go back to pages in their cache, and click on a different item. Our system can retain predictions made in the past (that is, a set of possible next requests and their probabilities) and combine them with current predictions to get (hopefully) a more accurate set of predictions.

- **Considering inexact sequence matching.** Web request sequences are potentially noisy. In particular, users may change the order in which some pages are visited, or skip some resources entirely and still have a successful experience. Thus, our system has been expanded to be able to consider some kinds of inexact sequence matching for building a prediction model.

More detail on these approaches will be provided when we describe experiments using them.

Finally, we should note that while we would like to find or build an IOLA as we described in Chapter 3, for the purposes of these tests we will focus on potential predictive performance (e.g., accuracy) and ignore certain aspects of implementation efficiency. In particular, our codes are designed for generality, and not necessarily for efficiency (especially in terms of space — to make more experiments tractable, we have had to pay some attention to time).

4.4 Prediction Workloads

There are three primary types of Web workloads, corresponding to three viewpoints on the traffic. Here we characterize each of the three and describe the datasets of each type that we will use. A summary of the datasets used can be found in Table 4.1.

Trace name	Described	Requests	Clients	Servers	Time
EPA-HTTP	[Bot95]	47748	2333	1	1 day
Music Machines	[PE98, PE00]	530873	46816	1	2 months
SSDC	[DL00]	187774	9532	1	8 months
UCB-12days	[Gri97, GB97]	5.95M	7726	41836	12 days
UCB-20-clients	Section 4.4.2	163677	20	3755	12 days
UCB-20-servers	Section 4.4.3	746711	6620	20	12 days

Table 4.1: Traces used for prediction and their characteristics.

4.4.1 Proxy

Typically sitting between a set of users and all Web servers, a proxy sees a more limited user base than origin Web servers, but in some cases a proxy may group together users with some overlapping interests (e.g., users of a workgroup or corporate proxy may be more likely to view the same content). It typically records all requests not served by browser caches, but logs may contain overlapping user requests from other proxies or from different users that are assigned the same IP address at different times.

The models built by proxies can vary — from the typical user in a single model to highly personalized models for each individual user. In any case, the predictive model can be used to prefetch directly into its cache, or to provide hints to a client cache for prefetching.

We will use one proxy trace in our experiments. The UC Berkeley Home IP HTTP Traces [Gri97] are a record of Web traffic collected by Steve Gribble as a graduate student in November 1996. Gribble used a snooping proxy to record traffic generated by the UC Berkeley Home IP dialup and wireless users (2.4Kbps, 14.4Kbps, and 28.8Kbps land-line modems, and 20-30Kbps bandwidth for the wireless modems). This is a large trace, from which we have selected the first 12 out of 18 days (for comparison with Fan *et al.* [FJCL99]), for a total of close to six million requests.

4.4.2 Client

The client is one of the two necessary participants in a Web transaction. A few researchers have recorded transactions from within the client browser [CP95, CBC95,

CB97, TG97] making it possible to see exactly what the user does — clicking on links, typing URLs, using navigational aids such as Back and Forward buttons and Bookmarks. It is also typically necessary to log at this level to capture activity that is served by the browser cache, although one potential alternative is to use a cache-busting proxy [Kel01].

Using an individual client history to build a model of the client provides the opportunity to make predictions that are highly personalized, and thus reflect the behavior patterns of the individual user. Unfortunately, logs from augmented browsers (such as those captured by Tauscher and Greenberg [TG97]) are rare. Instead, we will use a subset of requests captured by an upstream proxy (from the UCB dataset) with the understanding that such traces do not reflect all user actions — just those that were not served from the browser cache.

We have extracted individual request histories from the UCB proxy trace. However, not all "clients" identified from proxy traces with unique IP addresses are really individual users. Since proxies can be configured into a hierarchy of proxy caches, we have to be concerned with the possibility that proxy traces could have "clients" which are really proxy caches themselves, with multiple users (or even proxies!) behind them. Likewise, even when a client corresponds to a particular non-proxy system, it may correspond to a mechanized process that repeatedly fetches one resource (or a small set of resources). Since the latter correspond to highly regular request patterns, and the former correspond to overlapping request patterns, we will attempt to avoid the worst of both in the concern for fairness in evaluation. We have ranked the clients by total numbers of requests, and ignored the top twenty, and instead selected the second twenty as the representative set of active users.

The potential performance improvement by prefetching into the client cache is, of course, one of the primary motivations of this dissertation. Building good user models can also help in the development of intelligent interfaces and appropriate history mechanisms.

4.4.3 Server

Servers provide a complementary view on Web usage from that of clients. Instead of seeing all requests made by a user, they see all requests made to one or more servers. Since they only know of requests for particular servers, such logs are unlikely to contain information about transitions to other systems. Technically, the HTTP Referrer header provides information on transitions into a particular server, but these are rarely provided in publicly available logs.

When learning a predictive model, the server could build an individual model for each user. This would be useful to personalize the content on the Web site for the individual user, or to provide hints to the client browser on what resources would be useful to prefetch. One difficulty is the relative scarcity of information unless the user visits repeatedly, providing more data than the typical site visit which commonly contains requests for only a handful of resources. An alternative is to build a single model of the typical user — providing directions that may say that most users request resource B after fetching resource A. When trends are common, this approach finds them. A single model can also provide information that could be used in site re-design for better navigation [PE97, PE00]. In between the two extremes lies the potential for a collaborative filtering approach in which individual models from many users can contribute to suggest actions useful to the current user, as pointed out by Zukerman *et al.* [ZA01]. For the experiments in this dissertation, we generally build a single model based on the traffic seen where the model is stored. Thus, a server would build a single model, which while likely not corresponding to any particular user, would effectively model the typical behavior seen.

In addition to those already described, predictive Web server usage models can also be used to improve server performance through in-memory caching, reduced disk load, and reduced loads on back-end database servers. Similarly, they could be used for prefetching into a separate server-specific reverse proxy cache (with similar benefits).

Server logs are widely available (compared to other kinds of logs), but they have some limitations, as they do not record requests to non-server resources and do not see

responses served by downstream caches (whether browser or proxy).

This chapter will use traces from three servers. The first is the EPA-HTTP server logs [Bot95] which contain close to 48,000 requests corresponding to 24 hours of service at the end of August 1995. The second is two months of usage from the Music Machines website [PE98, PE00], collected in September and October of 1997. Unlike most Web traces, the Music Machines website was specifically configured to prevent caching, so the log represents all requests (not just the browser cache misses). The third (SSDC) is a trace of approximately eight months of non-local usage of a website for a small software development company. This site was hosted behind a dedicated modem, and was collected over 1997 and 1998. Additionally, we have extracted the twenty-most-popular servers from the UCB proxy trace.

4.5 Experimental Results

In this section we will examine the effect of changes to various model parameters on predictive performance. In this way we can determine the sensitivity of the model (or data sets) to small and large changes in parameter values, and to find useful settings of those parameters for the tested data sets.

4.5.1 Increasing number of predictions

In order to help validate our prediction codes, we replicated (to the extent possible) Sarukkai's HTTP server request prediction experiment [Sar00]. This experiment used the EPA-HTTP data set, in which the first 40,000 requests were used as training data, and the remainder for testing.

We set up our tests identically, and configured our prediction codes to use a first order Markov model (i.e., an n-gram size of 2, with no minimum support or confidence needed to predict). Thus unlike the remainder of the experiments presented in this chapter, this experiment builds a model using the initial training data, and freezes it for use on the test data. This is to facilitate comparison with Sarukkai's results. The Static Predictions line in Figure 4.5 corresponds closely to the performance of the

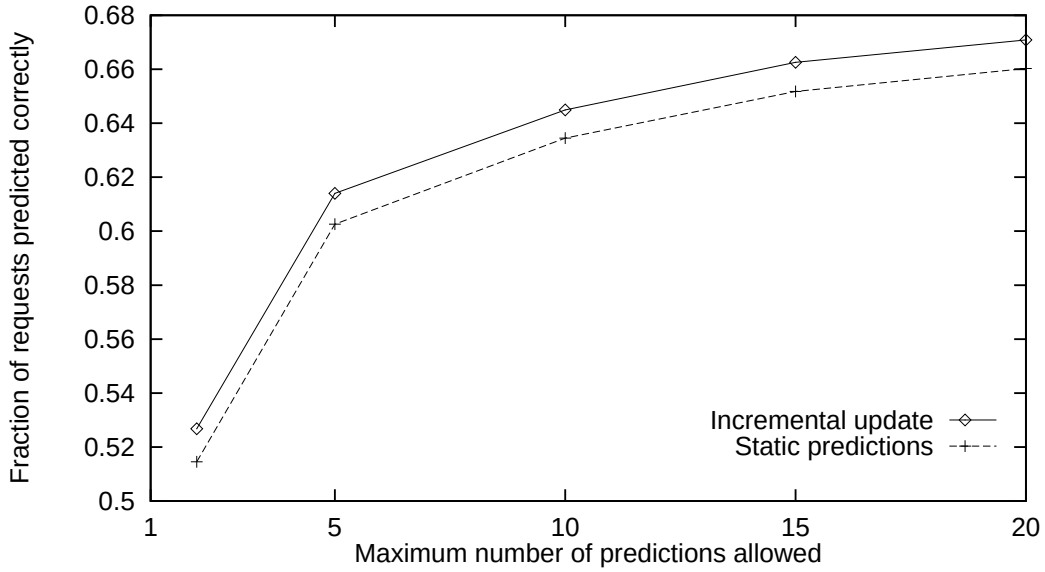


Figure 4.5: Predictive accuracy for the EPA-HTTP data set with varying numbers of allowed predictions.

first test of Markov chains reported in Figure 5 of [Sar00]. Figure 4.5 also shows the approximately 1% absolute increase in predictive accuracy of the same system when it is allowed to incrementally update its model as it moves through the test set.

The EPA-HTTP logs, however, are rather short (especially the test set), and so we consider the performance of prediction for other server logs in subsequent tests and figures. Since they provide a more realistic measure of performance, we will use incremental predictive accuracy throughout the rest of this dissertation.

In Figure 4.6, we examine incremental predictive performance while varying the same parameter (number of predictions permitted) over four other traces (SSDC, UCB servers, Music Machines, and UCB clients). For the initial case of one allowed prediction, we find that performance ranges from slightly over 10% to close to 40% accuracy. As the number of predictions allowed increases to 20, predictive performance increases significantly — a relative improvement of between 45% and 167%. The server-based traces show marked performance increases, demonstrating that the traces do indeed contain sufficient data to include most of the choices that a user might make. The performance of the client-based trace, conversely, remains low, demonstrating that the experience of an individual user is insufficient to include the activities that the user

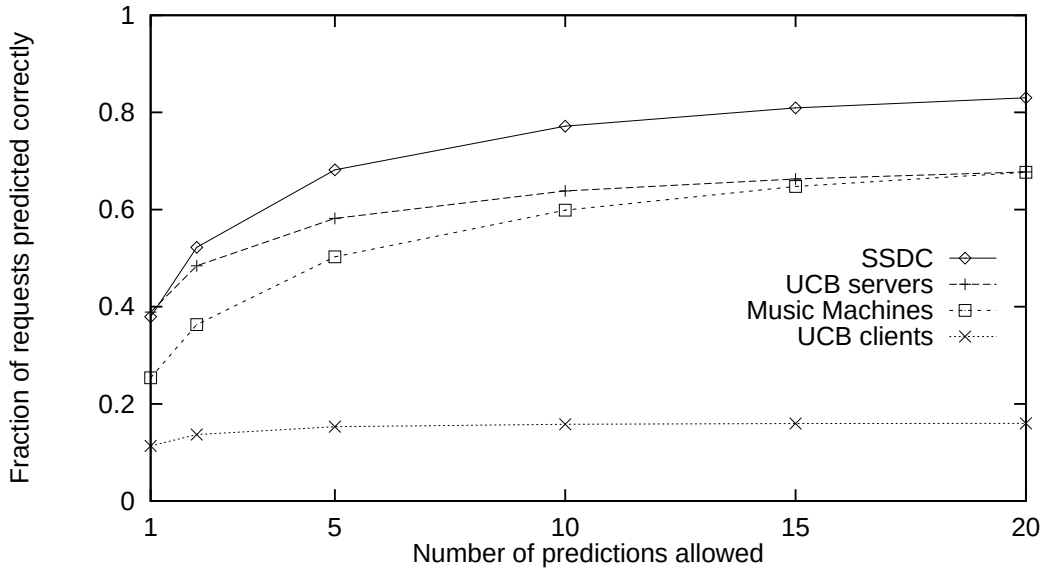


Figure 4.6: Predictive accuracy for various data sets with varying numbers of allowed predictions.

will perform in the future. Finally, note that to achieve such high levels of predictive performance, systems will need to make many predictions, which usually come with some resource cost, such as time, bandwidth, and CPU usage.

4.5.2 Increasing n-gram size

One potential way to improve accuracy is to consider n-grams larger than two. This increase in context allows the model to learn more specific patterns. Figure 4.7 shows the relative improvement (compared to n=2) in incremental predictive accuracy for multiple traces when making just one prediction at each step. Thus, if the accuracy at n=2 were .2, an accuracy of .3 would be a 50% relative improvement, and all traces are shown to have zero improvement at n=2. In each case, the longest n-gram is used for prediction (ties are broken by selecting the higher probability n-gram), and 1-grams are permitted (i.e., corresponding to predictions of overall request popularity) if nothing else matches. The graph shows that adding longer sequences does help predictive accuracy, but improvement peaks and then wanes as longer but rarer (and

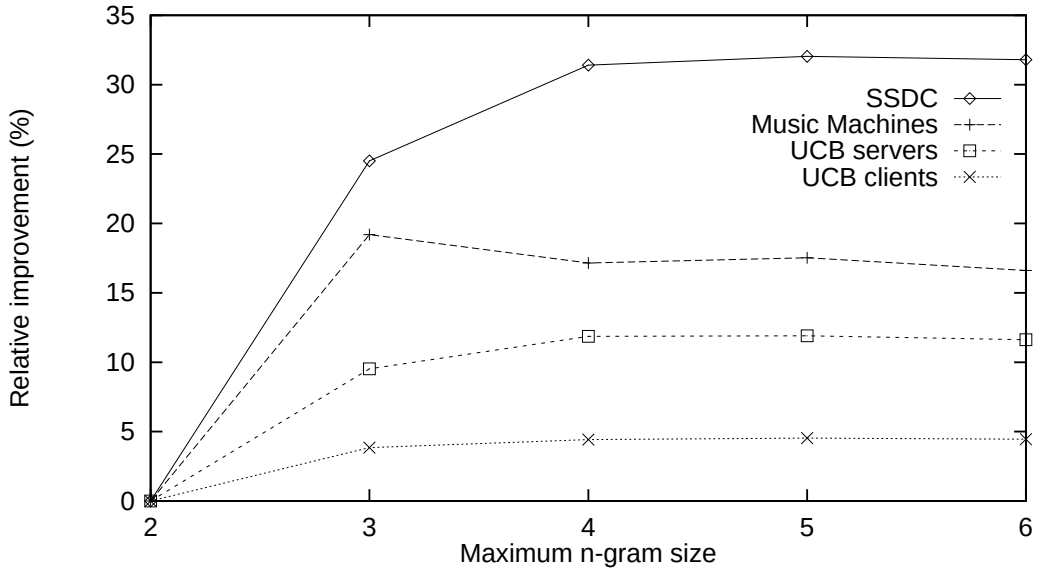


Figure 4.7: Relative improvement in predictive accuracy for multiple data sets as maximum context length grows (as compared to a context length of two).

less accurate) sequences are used for prediction.[2] Thus, the figure shows that larger n-grams themselves can be useful, but Figure 4.8 demonstrates that using the largest n-grams alone for predictions is insufficient. This performance is explained by the fact that longer n-grams match fewer cases than shorter n-grams, and thus able to make fewer correct predictions.

4.5.3 Incorporating shorter contexts

Prediction by partial match provides an automatic way to use contexts shorter than the longest-matching one. In our experiments, we find that the PPM variations are able to perform slightly better than the comparable longest n-gram match. Figure 4.9 plots the performance improvement for three versions of PPM (described in Section 4.3) over the default n-gram approach. As Figure 4.9(a) shows, in the best case, PPM-C gives a close to 3% improvement when only the best prediction is used. When multiple predictions are permitted (as in Figure 4.9(b)), the performance improvement is even less.

[2]As a result, in subsequent tests we will often use an n-gram limit of 4, as the inclusion of larger n-grams typically does not improve performance.

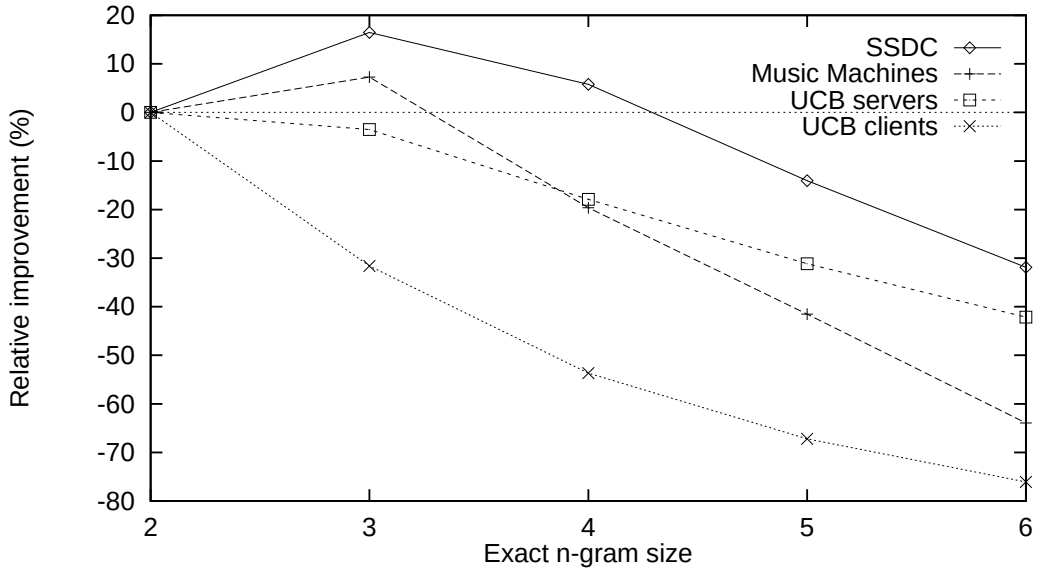


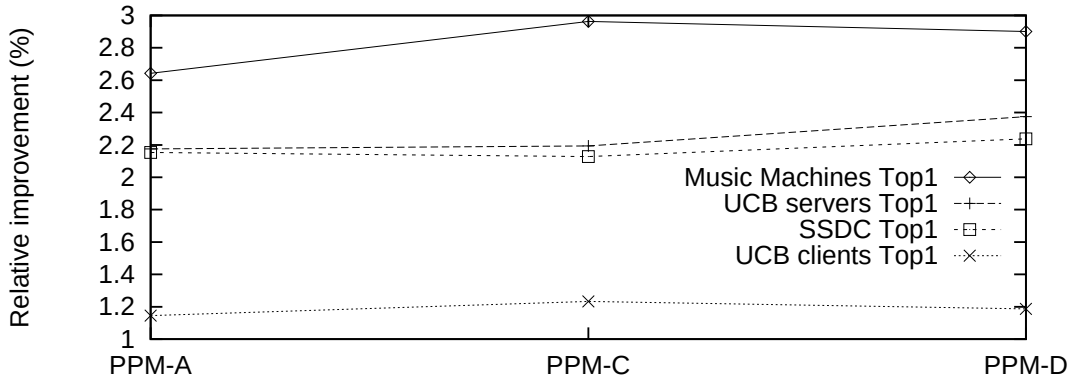
Figure 4.8: Relative improvement in predictive accuracy for multiple data sets using only maximal length sequences (as compared to a context length of two).

However, we can also endow n-grams with the ability to incorporate shorter n-grams. In this version of our system, the n-grams always merge with the results of prediction at the shorter context, with a fixed weight (as opposed to the weight from the dynamically calculated escape probability in the PPM model). Figures 4.10(a) and (b) shows the predictive accuracy for various weightings when using a maximum n-gram of size 6, and minimum of 2. The n-gram model combined with the sample best (out of the five tested) weighted shorter context provides comparable performance (less than .5% absolute difference) to the best PPM model.

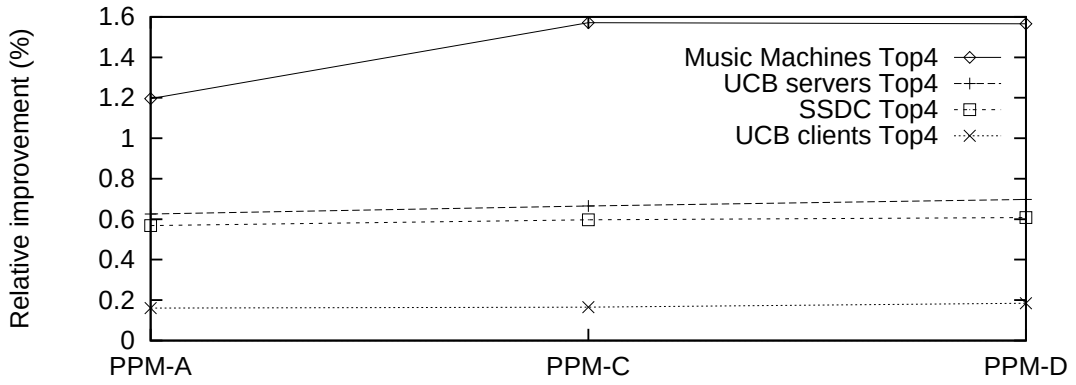
4.5.4 Increasing prediction window

Another way to improve reported accuracy is to allow predictions to match more than just the very next action. In a real Web system, prefetched content would be cached and thus available to satisfy user requests in the future. One can argue that when a prediction does not match the next request, it is incorrect. However, if the prediction instead matches and can be applied to some future request, we should count it as correct. Thus, in this test we apply a Web-specific heuristic for measuring performance.

In Figure 4.11 we graph the relative performance improvement that results when



(a) Top-1 predictions

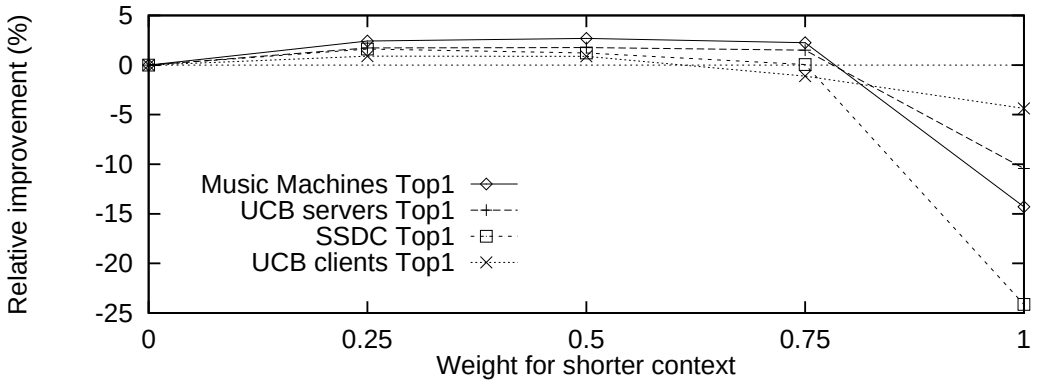


(b) Top-4 predictions

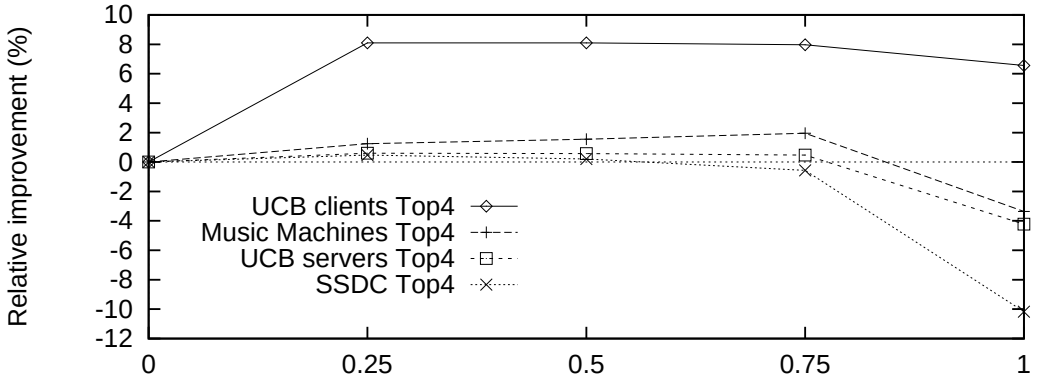
Figure 4.9: Relative improvement in predictive accuracy when using three PPM-based prediction models instead of n-grams.

we allow predictions to match the next 1, 2, 3, 5, 10 and 20 subsequent requests. While at 20 requests performance continues to increase, the rate of growth is tapering. The apparent boost in potential performance suggests that even when the next request is not predicted perfectly, the predicted requests are potentially executed in the near future.

Predictive accuracy, even when using the mechanisms described here, is limited at least by the recurrence rates of the sequences being examined. That is, in this chapter, we only consider predicting from history which means that every unique action cannot be predicted when it is first introduced. Thus, if we were to plot predictive performance on a scale of what is possible, the graphs would be 2-6% absolute percentage points



(a) Top-1 predictions



(b) Top-4 predictions

Figure 4.10: Relative improvement in predictive accuracy when using shorter contexts in n-gram prediction.

higher.

4.5.5 De-emphasizing likely cached objects

One drawback to the above approach is that it breaks the typical *modus operandi* of "predict the next request" with which most machine learning or data compression researchers are familiar. Here we retain the traditional evaluation model, but change instead the prediction based on a Web domain-specific heuristic. Since recently requested objects are likely to be in a user's cache, it is unlikely that the user will request them again, even when such an object has high probability from the model. This will

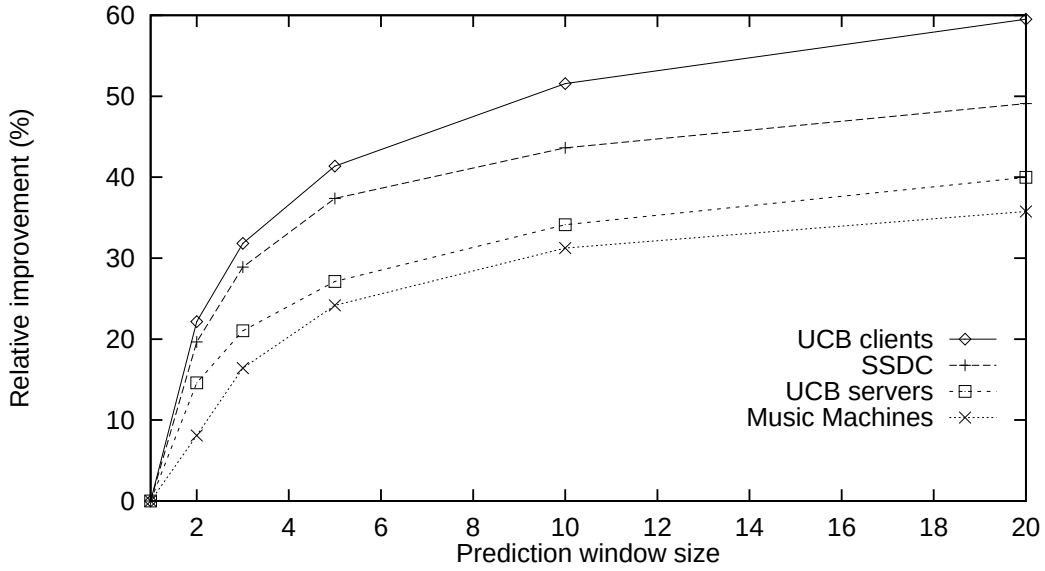


Figure 4.11: Relative improvement in predictive accuracy as the window of actions against which each prediction is tested grows.

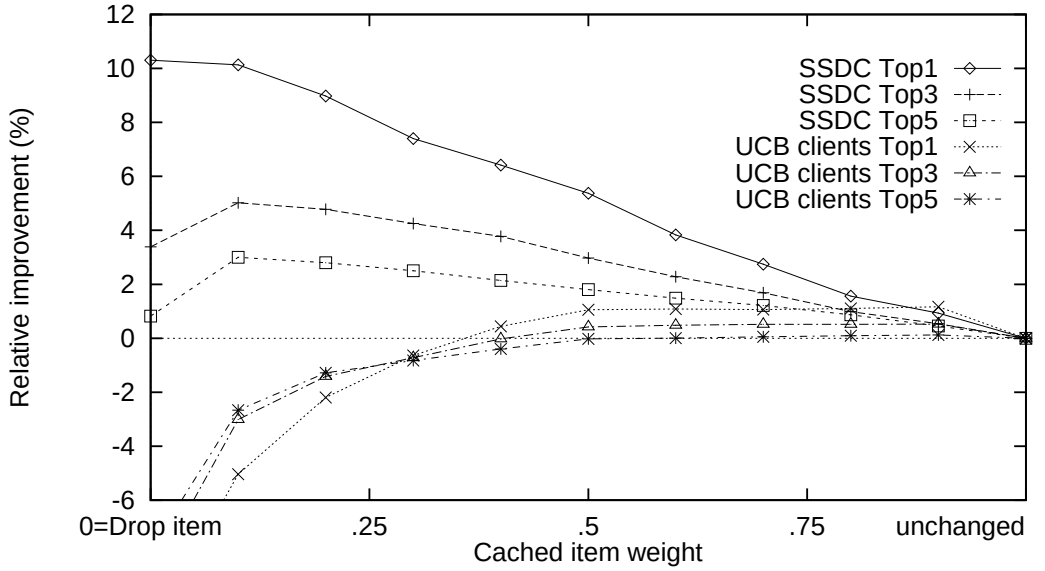


Figure 4.12: Relative changes in predictive accuracy as cached requests are de-emphasized.

be the case for most embedded resources (like images), that are often highly cacheable, but is even possible for HTML pages, when a user takes an atypical route through a Web site. Since it will not be always true (some objects are uncacheable, and so the user will need to retrieve them repeatedly), we are unlikely to want to prevent such

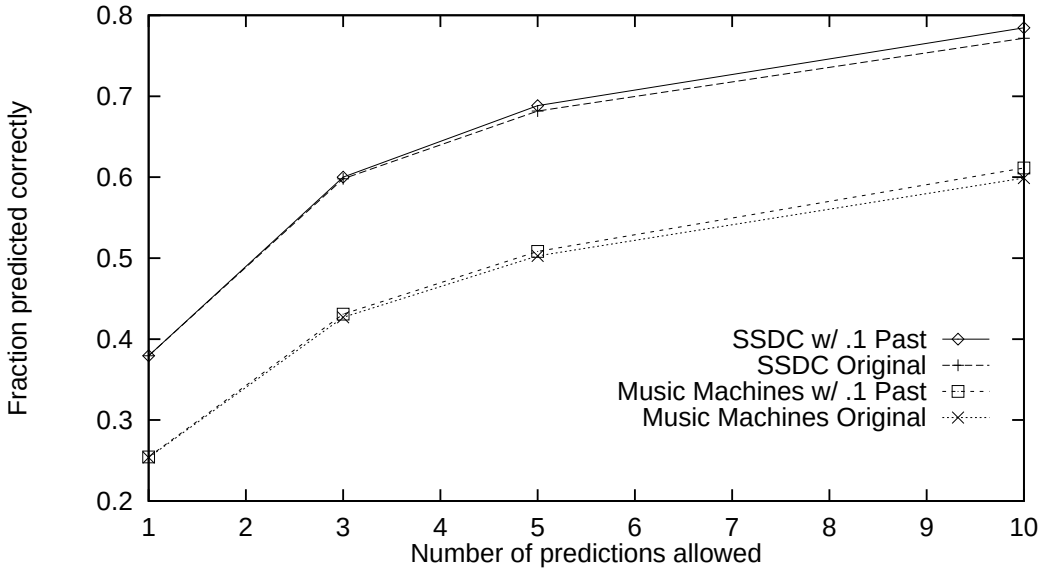


Figure 4.13: Predictive accuracy as past predictions are retained.

near-term repetitions, but we do want to de-emphasize them.

Our system keeps a window of the most recent n requests made per client. We compare a potential prediction to those in that window, and examine the appropriate emphasis given to those predictions by varying the weight given to them. Figure 4.12 shows the performance possible using this technique on two traces and three values for the number of predictions allowed (all within a first order Markov model). It shows that for the SSDC trace, it is beneficial to reduce the likelihood of re-predicting objects that have been recently requested (and thus, likely still in cache). Since the Music Machines trace was generated from a Web site that set all objects to be uncacheable, a test using that Web log would never show improvements (thus, we do not plot it here). The UCB clients trace likely contain references with a smaller likelihood of being cacheable, and thus only show slight improvements for small decreases in weights. The UCB server traces (not shown for clarity) have similar or slightly worse performance than the UCB clients.

4.5.6 Retaining past predictions

Another Web domain-specific improvement can be realized by considering how users often browse the Web. When a particular page (e.g., B) is found to be uninteresting, or has no further value, the user will often use the back button to select a new link (e.g., link C, from page A) of potential value. The act of going back to a previous page is of interest in two ways. First, the revisiting of page A is not recorded in external request logs because the page is typically cached. Second, the links from the earlier page (A) may be selected, even when that page was not the most recently requested one. Thus, since the links of page A may be valuable to the user in later contexts, so might the predictions of next pages from A. We implemented a mechanism to attempt to capture this effect by merging the previous prediction with the current one, subject to a weighting.

In our experiments, we found this to be only minimally useful, and even then only for cases in which multiple predictions are allowed. Figure 4.13 shows that the improvement in predictive performance increases slightly as the number of allowed predictions increases.

4.5.7 Considering inexact sequence matching

In many domains, small variations in the sequences are not significant. HTTP request traces are one of those domains in which some variation is expected — the particular request ordering of a page and its embedded resources is a function of the page HTML source, the browser rendering speed and enabled options (such as Java and JavaScript), and network performance. Additionally, on some visits, a resource may be retrieved, where in other visits, the resource may be cached and thus missing from the request trace.

Therefore, we considered a mechanism that allowed for such variation. In our standard implementation (i.e., for exact sequences) with maximum n-gram size of 4, the arrival of D in the *real* sequence (A, B, C, D) will trigger model updates for all suffixes:

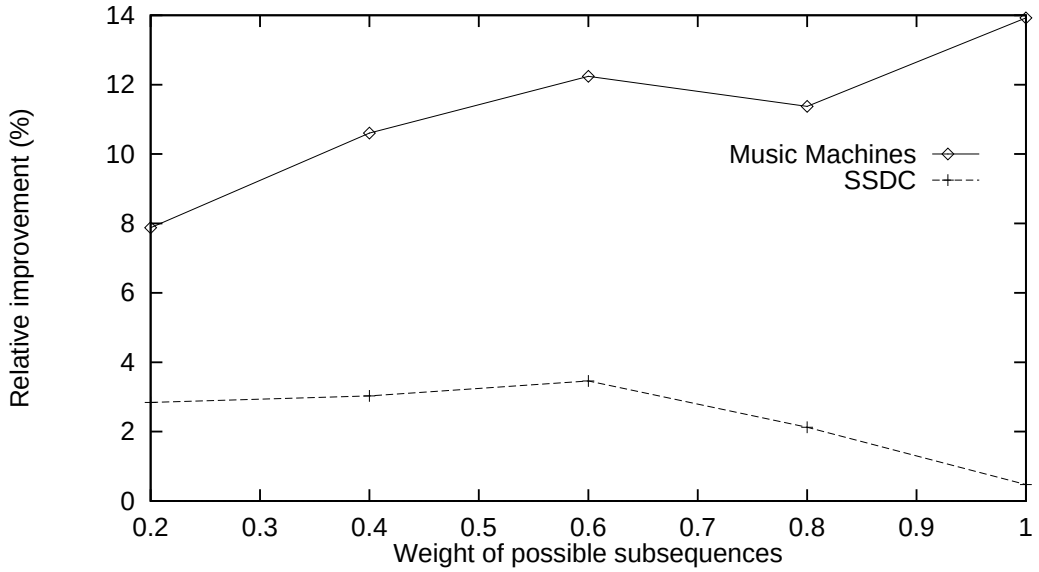


Figure 4.14: Relative improvement in predictive accuracy for n-grams as the weight for non-exact matches is varied.

(D), (C, D), (B, C, D), and (A, B, C, D). In this way, we track how often each subsequence was seen. To allow inexact matching, we generalized the model to incorporate updates corresponding to *unreal* sequences, and continue to use exact matching at the point of prediction.

In our modified implementation, the arrival of D in the real sequence (A, B, C, D) will again trigger model updates for all real suffixes, but also all unreal suffixes that include the last event: (A, D), (B, D), (A, C, D), (A, B, D). The counts for these unreal sequences are updated by adding a weight (typically less than 1).

Figure 4.14 shows relative improvement as the unreal update weight is varied. The scenario plotted is for a fixed-size Markov model (order 4), the top five predictions are used, no confidence or support thresholds, and performance is tested across various prediction window sizes. This approach is particularly beneficial in cases like these, but is not effective (or can even be detrimental) for scenarios that predict from multiple contexts (e.g., PPMs) and are focused on evaluation based strictly on predicting the next action. We should also point out a practical consideration — this approach is slower and requires a much larger prediction model as many of the unreal sequences that are now captured were sequences that did not occur naturally. This factor, combined with

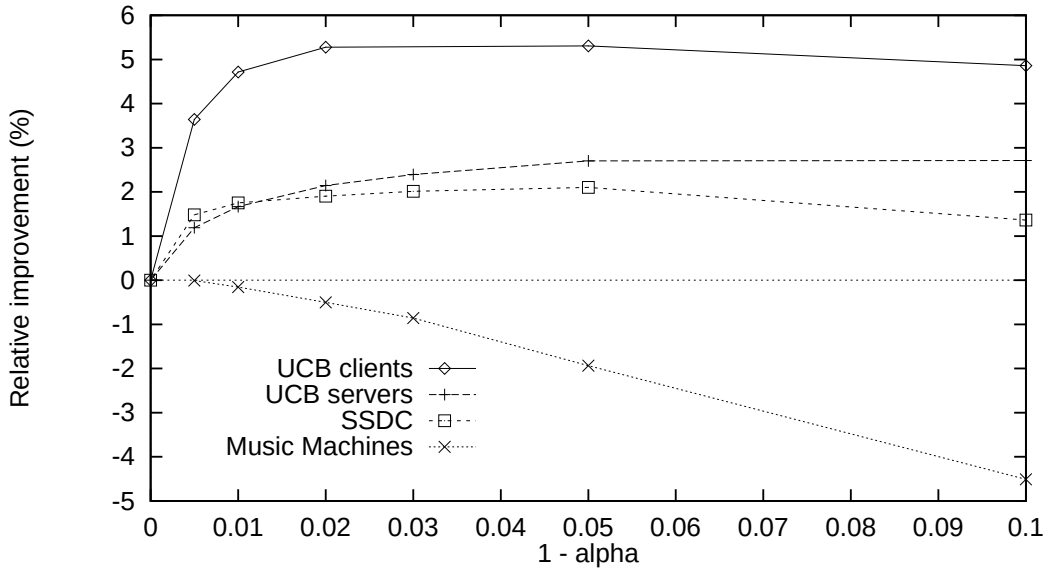


Figure 4.15: Relative change in predictive accuracy as alpha (emphasis on recency) is varied.

improvements of just 3-16%, suggest that this approach is not likely to be useful in practice.

4.5.8 Tracking changing usage patterns

As we mentioned in the discussion of predictive methods in Section 4.3, we can emphasize recent activity by changing the value of alpha. Figure 4.15 shows the effect of alpha on predictive accuracy. In these experiments, we set the maximum and minimum n-gram size to be 4 and 1, respectively. Note that an alpha of 1 means no emphasis on recency, and an alpha of 0 means to use the most recent action as the prediction of the action coming next. Unlike most of the traces in the figure, which show small improvements with an appropriate *alpha*, the Music Machines trace declined slightly in performance as *alpha* decreased. We speculate that concept drift in server logs are predominantly from server changes and not changes in user patterns (from changing user interests). If the Music Machines content did not change over the (two month) time period of the trace, then an emphasis on recent activity is not warranted.

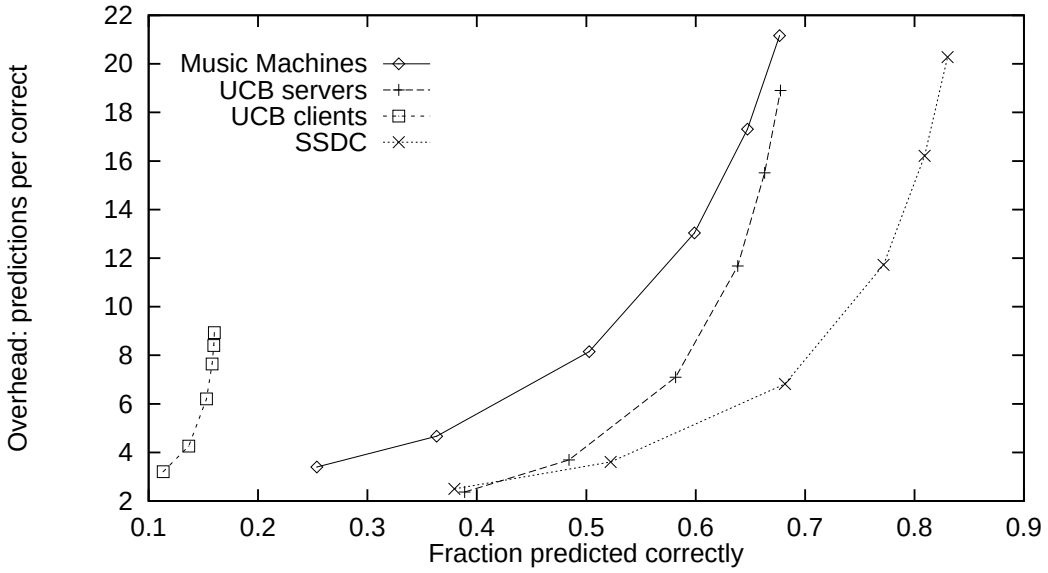


Figure 4.16: Ratio of number of predictions made to those predicted correctly.

4.5.9 Considering mistake costs

Accuracy alone is typically only important to researchers. In the real world, there are costs for mistakes. Figure 4.16 provides one measure of the cost of allowing for additional predictions per action. As the number of allowed predictions per action increases (1, 2, 5, 10, 15, 20) the fraction predicted correctly grows, but the number of predictions needed to get that accuracy also increases significantly. (Note that this example configuration, the same as in Figure 4.6, does not limit predictions to any particular confidence or support, and so likely overestimates the number of predictions made in practice.) In a prefetching scenario in which mistakes are not kept for Music Machines or SSDC, the average per-request bandwidth could be up to twenty times the non-prefetching bandwidth. Note however, that this analysis does not consider the positive and extensive effects of caching, which we will examine in future chapters. The simple point is that there is always a tradeoff — in this case we can see that for increasing coverage, we will require additional bandwidth usage.

Therefore, it is worthwhile to consider how to reduce or limit the likelihood of making a false prediction, as discussed in Section 4.2.6. Figure 4.17 demonstrates the increased precision possible (with lower overall accuracy) as the threshold is increased

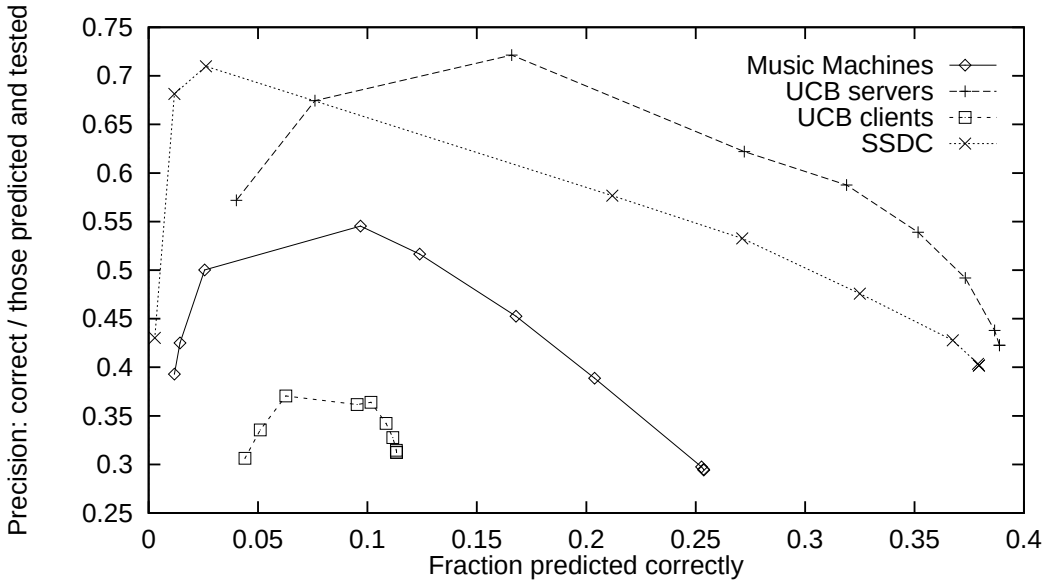


Figure 4.17: Ratio of precision to overall accuracy for varying confidence.

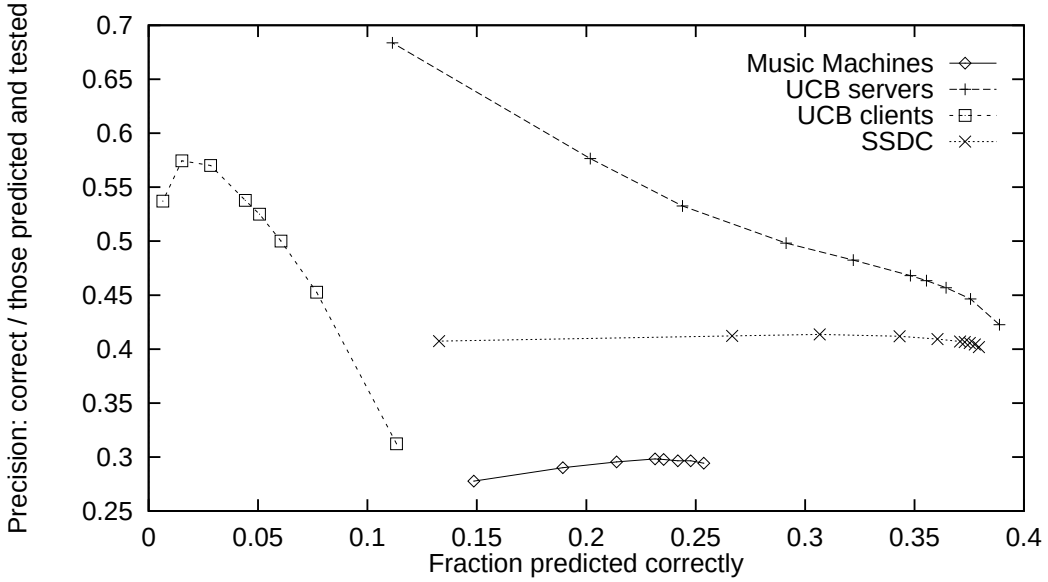


Figure 4.18: Ratio of precision to overall accuracy for varying support.

for an otherwise traditional 1-step Markov model. For each trace, we plot the precision as the minimum confidence threshold varies from .01 to .95. Low threshold values result in higher fractions predicted correctly. The figure demonstrates that while much larger values of precision are possible, they come at a cost of the number of requests predicted correctly. On the other hand, high precision minimizes the rate of mis-predictions,

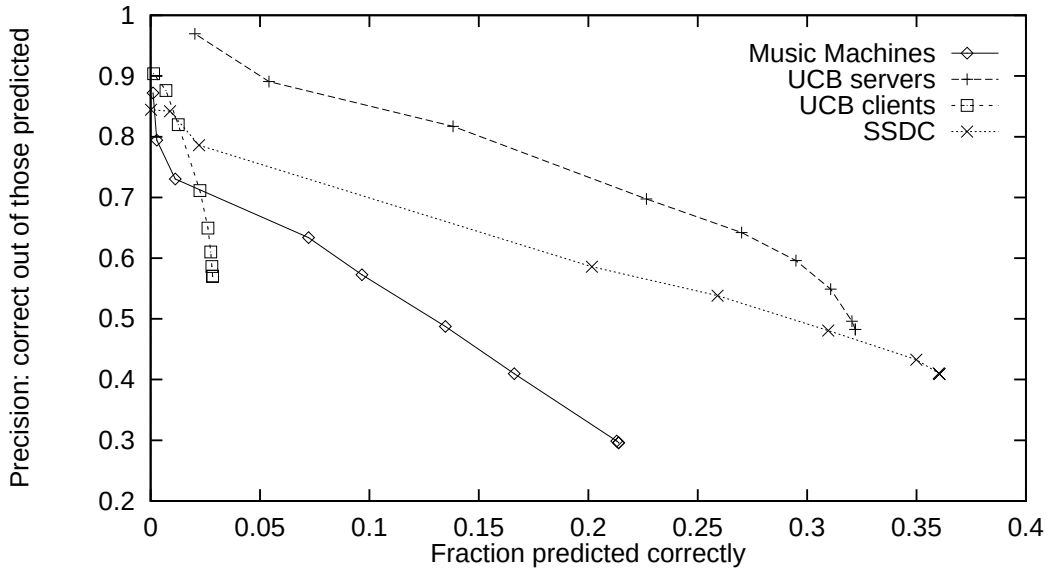


Figure 4.19: Ratio of precision to overall accuracy for a minimum support of 10 with varying confidence.

reducing the costs associated with each correct prediction.

Figure 4.18 shows the same scenario except using a support threshold (ranging from 1 to 50 instances) instead. The effect on increased precision is more variable, as it depends on the number of times a each pattern is found in a trace. Thus, the UCB clients have relatively few repetitions per client, so the curve is shifted further to the left. Neither SSDC nor Music Machines achieve significant growth in precision, but UCB servers with a threshold of 50 instances gets a precision close to .7.

Figure 4.19 provides one combined view — a minimum support of 10 instances combined with a varying confidence threshold. As can be seen, with this combination it is possible to achieve precision that exceeds that possible from either alone. The lesson here is that high accuracy predictions are quite achievable, but are typically applicable to a much smaller fraction of the trace.

4.6 Discussion

We've demonstrated various techniques to improve predictive accuracy on logs of Web requests. Most of these methods can be combined, leading to significant improvements.

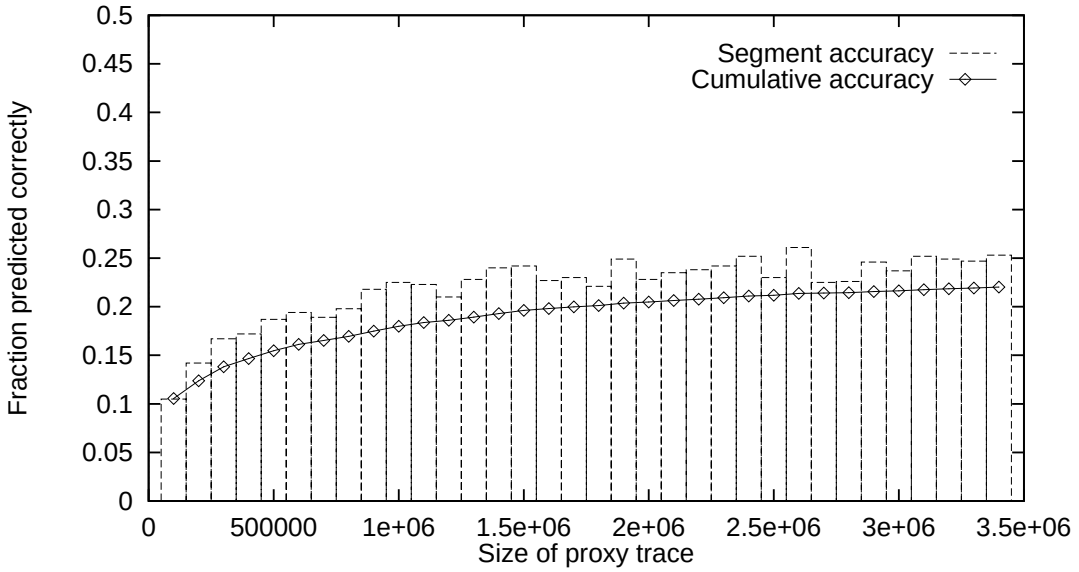


Figure 4.20: Predictive accuracy for a first-order Markov model on UCB proxy data as the data set size varies.

A first-order Markov model is able to get approximately 38% top-1 predictive accuracy on the SSDC trace. A top-5 version can increase that by another 30% to have 68% predictive accuracy. After a few more adjustments (including using PPM, version C, with maximum n-gram 6, reducing predictions likely to be in cache down to 20%, and including past predictions with weight .025), accuracies of 75% are possible. Put another way, this constitutes a 20% reduction in error. And if we consider only those predictions that were actually tested (i.e., ignoring those for which the client made no more requests), we get a prediction accuracy of over 79%. This performance, of course, comes at the cost of a large number of guesses. In this case, we are making five guesses almost all of the time. Alternatively, lower overall accuracy but at higher precision is possible with some modifications (such as the use of confidence and support thresholds).

Some investigators have focused on this issue. Although they use a simplistic model of co-occurrence, Jiang and Kleinrock [JK97, JK98] develop a complex cost-based metric (delay cost, server resource cost) to determine a threshold for prefetching, and use it to consider system load, capacity, and costs. Likewise Zukerman *et al.* [ZA01, AZN99, ZAN99] as well as others (e.g., [SH02, Hor98]) develop decision-theoretic utility models to balance cost and benefit and for use in evaluation.

Finally, the astute reader may have noticed that while we mentioned proxy workloads for completeness back in Section 4.4, we have not included them in the figures shown, as they do not provide significant additional information. While running the UCB client workloads and calculating the client-side predictive accuracy (which is what we report), we also calculated the accuracy that a proxy-based predictor would have achieved. In general, its accuracy was 5-10% higher, relative to the client accuracy (i.e., .1-2% in absolute terms).

For reference, we provide one graph of performance over one large proxy trace. Figure 4.20 shows the short– and long-term trends for predictive accuracy for the first 3.4M requests of the UCB trace. This is for a baseline predictor: a first-order Markov model allowing just one guess without thresholds. It shows that the model continues to learn throughout the trace (as the per 100k segment accuracies are always above the cumulative accuracy). Note that even with millions of requests, the displayed trace only corresponds to about eight days of traffic, and thus is understandable why its performance would not have stabilized yet. However, it is also apparent that the overall mean predictive accuracy will likely converge to an asymptote somewhere less than 25%.

4.7 Summary

In this chapter we have examined in detail the problem of predicting Web requests, focusing on Markov and Markov-like algorithms. We have discussed the problems of evaluation, and have provided a consistent description of algorithms used by many researchers in this domain. We built generalized codes that implement the various algorithms described. The algorithms were tested on the same set of Web traces, facilitating performance comparisons.

We demonstrated the potential for using probabilistic Markov and Markov-like methods for Web sequence prediction. By exploring various parameters, we showed that larger models incorporating multiple contexts could outperform simpler models. We also provided evidence for limited "concept drift" in some Web workloads by testing

a variation that emphasized recent activity. Finally, we demonstrated the performance changes that result when using Web-inspired algorithm changes. Based on these results, we recommend the use of a multi-context predictor (such as PPM, or the multi-context n-gram approach described in this chapter). However, for the Web, it appears that relatively few contexts are needed — n-grams don't need to be much more than 3 or 4. Incorporating some caching-related heuristics may also help, such as de-emphasizing predictions likely to be cached. Finally, the utilization of thresholds on prediction can significantly reduce the likelihood of erroneous predictions.

In this chapter we hinted at some of the effects that client caching can have on performance. This issue will be explored further in later chapters, once we have better mechanisms for evaluating the effects of prefetching techniques.