Honours Year Project Report

Predicting Web 2.0 Thread Updates

By

Shawn Tan

Department of Computer Science

School of Computing

National University of Singapore

2012

Honours Year Project Report

Predicting Web 2.0 Thread Updates

By

Shawn Tan

Department of Computer Science

School of Computing

National University of Singapore

2012

Project No: H079830
Advisor: A/P Kan Min-Yen
Deliverables:
Report: 1 Volume

Abstract

In this report, we study a problem and design an efficient algorithm to solve the problem. We implemented the algorithm and evaluated its performance againts previous proposed algorithms that solves the same problem. Our results show that our algorithm runs faster.

Subject Descriptors:
C5 Computer System Implementation
G2.2 Graph Algorithms

Keywords:
Problem, algorithm, implementation

Implementation Software and Hardware:
python, bash

Acknowledgement

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

With the advent of Web 2.0, sites with forums, or similar thread-based discussion features are increasingly common. In this project,our goal is to predict updates in such threads.

Table **??** shows us that many of the popular Web 2.0 sites have comment features. This suggests that content on the web is increasingly being created by users alongside content providers. While mining structured, curated content from sites like Amazon, for data like prices is easy and effective, data that can be obtained from user-generated content are of a different nature. One may be able to infer public sentiment about a given product that would not be readily available from an e-commerce site. In some cases, news may travel more quickly through such online community discussion than through traditional media. Users also typically discuss purchased products bought online via these forums, and companies that want to get timely feedback about their product should turn to data mined from such sites.

A naive way of getting timely updates is to aggressively hit the pages repeatedly downloading the pages at a very frequent rate. However, the number of pages in a forum site are far too large to perform this efficiently on every forum. One way to minimise this cost would be to look at the time differences between previous posts to estimate the arrival of the next one. We believe that the content of the thread has information that can give a better estimate of the time interval between the last post and a new one.

For example, a thread in a technical forum about a Linux distribution may start out as a question. Subsequent questions that attempt to either clarify or expand on the original question

	T	FB L	FB S	G +1	L	DL	C	PV	Follows
http://www.lifehacker.com	1	1	1				1	1	
http://digg.com/	1	1			1	1	1	1	
http://9gag.com/	1	1	1	1	1		1	1	
http://www.flickr.com/					1		1	1	
http://news.ycombinator.com/					1		1		
http://stackoverflow.com/					1		1	1	
http://www.youtube.com/					1	1	1	1	
http://www.reddit.com/					1	1	1		
http://www.stumbleupon.com/					1		1	1	
http://delicious.com/	1	1					1	1	1

Table 1.1: Features of popular Web 2.0 sites

T = Twitter mentions

FB L = Facebook Likes

FB S = Facebook Shares

G +1 = Google +1

L = Likes (Local)

DL = Dislikes (Local)

C = Comments

PV = Page Views

Follows = Site-local feature for keeping track of user's activities

may then be posted, resulting in a quick flurry of messages. Eventually, a more technically savvy user of the forum may come up with a solution, and the thread may eventually slow down after a series of messages thanking the problem solver. Suppose 10 days later, someone with a slight variation of the same problem posts on the thread again. A crawler that estimates update rates solely on the age of the thread to determine its download rate of the thread may not update itself with the thread.

Let us define all such thread-based discussion styled sites as forums. Ideally, an incremental crawler of such user-generated content should be able to maintain a fresh and complete database of content of the forum that it is monitoring. However, doing so with the previously mentioned naive method would (1) incur excessive costs when downloading un-updated pages, and (2) raise the possibility of the web master blocking the requester's IP address.

This year-long project proposes to use content-based features of a given thread to predict its next update time. We argue, that the content within the posts of the thread should be important in predicting the thread updates, and propose our approach to solving the problem.

Chapter 2

Related Work

2.1 Refresh policies for incremental crawlers

In order to devise such a strategy, we need to predict how often any user may update a page. Some work has been done to try to predict how often page content is updated, with the aim of scheduling download times in order to keep a local database fresh.

We will discuss the *timeliness* of our crawler to maintain the freshness of the local database, which refers to how new the extracted information is. Web crawlers can be used to crawl sites for user comments and threads for postprocessing later. Web crawlers which maintain the freshness of a database of crawled content are known as incremental crawlers. Two trade-offs these crawlers face cited by Yang, Cai, Wang, and Huang (2009) are *completeness* and *timeliness*. *Completeness* refers to the extent which the crawler fetches all the pages, without missing any pages. *Timeliness* refers to the efficiency with which the crawler discovers and downloads newly created content. We focus mainly on timeliness in this project, as we believe that timely updates of active threads are more important than complete archival of all threads in the forum site.

Many such works have used the Poisson distribution to model page updates. Coffman and Liu (1997) analysed the theoretical aspects of doing this, showing that if the page change process is governed by a Poisson process $\frac{\lambda^k e^{-\lambda\mu}}{k!}$, then accessing the page at intervals proportional to λ is optimal.

Cho and Garcia-Molina trace the change history of 720,000 web pages collected over 4 months, and showed empirically that the Poisson process model closely matches the update processes found in web pages (Cho, 1999). They then proposed different revisiting or refresh policies (Cho & Garcia-Molina, 2003; Cho & Garcia-molina, 2003) that attempt to maintain the freshness of the database.

The Poisson distribution were also used in Tan, Zhuang, and Mitra (2007) and Wolf, Squillante, and Yu (2002). However, the Poisson distribution is memoryless, and in experimental results due to Brewington and Cybenko (2000), the behaviour of site updates are not. Moreover, these studies were not performed specifically on online threads, where the behaviour of page updates may be very different from that of static pages.

Yang et al. (2009), attempted to resolve this by using the list structure of forum sites to infer a sitemap. With this, they reconstruct the full thread, and then use a linear-regression model to predict when the next update to the thread will arrive.

Online forums and bulletins have a logical, hierarchical structure in their layout, which typically alerts the user to thread updates by putting threads with new replies at the very top of the thread index. Yang's work exploits this as well as their linear model to achieve a predicton of when to retrieve the pages. However, this is not so for comments found on blog sites or discussion threads in an e-commerce site about a certain product and the lack of these pieces of information may result in a poorer estimate, or no estimate at all.

Our perspective is that the available content on the thread at the time of the retrieval should also be factored into the model used to predict the page updates. Next, we look at some of the related work pertaining to thread content.

2.2 Thread content analysis

While there is little existing work using content to predict page updates, we will review some existing work related to analysing thread-based pages which we think will aid us in our efforts to do content-based prediction.

Wang and McCarthy (2011) did work in finding out linkages between forum posts using

lexical chaining. They proposed a method to link posts using the tokens in the posts called $Chainer_{SV}$. While they do analyse the content of the individual posts, the paper does not make any prediction with regards to newer posts. The methods used to produce a numeric similarities between posts may be used as a feature to describe a thread in its current state, but incorporating this into our model is non-trivial.

With these related work in mind, we next propose our modelling of a thread as a Markov chain, and our approach to solving the problem.

2.3 Evaluation metrics

Yang et al. (2009) have a metric known as the T-score which gives the average time a post is made and when the post is received. The lower the T-score, the better the model. However, the metric does not account for visits which retrieve nothing new from the thread. As such, a crawler that repeatedly crawls the site at a frequent rate would do very well.

In Georgescul, Clark, and Armstrong (2009), the authors propose a new scheme for evaluating segmentation algorithms, Pr_{error}. This metric is the weighted sum of two probability counts Pr_{fa} which is the probability that a false alarm segmentation is made, and Pr_{miss} which is the probability that a segmentation is not made when there should be one. Unfortunately for our purposes, the metrics are calculated using the number of ground truths and segmentations given a window. As such, it does not account for the "distance" between the ground truths and the segmentation. It also does not allow for the predictions to appear after the ground truths, all requirements needed for a metric to evaluate timeliness of a model.

2.4 Predicting events in social media

There has been some work done recently in predicting events in social media, and in particular, tweets. Wang, Chen, and Kan (2012) dealt with predicting the retweetability of tweets using content. They applied two levels of classification, the first level categorising tweets into 6 different types: Opinion, Update, Interaction, Fact, Deals and Others. This was done using

similar techniques as Sriram and Fuhry (2010) and Naaman, Boase, and Lai (2010). The Opinion and Update categories are then further categorised into another three and two sub-categories each. The authors performed this categorisation using labeled Latent Dirichlet Allocation

The task in this work, was to predict which of three predefined classes a tweet will fall into no retweets, low and high. Our task is slightly more challenging, since we are trying to minimise the time from which a post is made to when the page is revisited. However, the feature sets used in these works should prove useful in our task.

2.5 Forecasting and Machine Learning

Since the purpose of our envisioned model would be to create a crawler that can estimate the best times to revisit a page, a proper approach to modeling this as a time-series model would be through machine learning techniques.

An interesting approach to forecasting stock prices was presented in Cao and Tay (2003). The technique involved tweaking conventional SVMs to weigh recent training instances more heavily than older instances. This is a particularly useful idea, since we face the same issue in our task: Recent posts are more descriptive of the current state of the thread, and hence should be more useful in predicting the next post.

Chapter 3

Method

We aim to predict the amount of time between the arrival of the next post and the time the last post in the thread was made. The information available to us are the previously made posts that we observe when first visiting the thread. The assumption made here is that the thread is not paginated in any way, and a single visit to the thread gives us the latest posts without having to traverse through the links to the latest page. This is because in practice, we would be able to keep track of where the last visited page of the thread was, and reading the new posts would incur a few more requests to the thread. This, in comparison to constantly hitting the page for updates, would be negligible.

More formally, what we are trying to do is to estimate a function f such that given a feature vector $\mathbf{X}$ representative of a window $\rho_{t-w+1}, \rho_{t-w+2}, \cdots, \rho_t$, where ρ_t represents the t-th post in the thread, we can approximate Δ_t with $f(\mathbf{X})$. In the following sections, we will discuss various methods for estimating f. Various notations will be used, a quick reference is provided in Table 3.1.

In the following sections, we describe the various methods we have to approximate f.

3.1 Baselines

A simple way of estimating the revisit rate would be to use the average time differences given the observed posts, or a training set. In previous work, we have seen that if page updates follow

Notation	Description
ρ	A post
t	Index of a post in a thread
w	Number of posts in a window
ρ_t	The t-th post in the thread
$\mathbf{v}_t$	The frequency count vector of the posts used in the t-th post
Δ_t	Time difference between a post at position t and a post at position $t+1$
$\mathbf{t}_\Delta$	Vector of Δ_ts in a given window
$\mathbf{t}_{\text{ctx}}$	Bit vector representing the day of week, and the hour of day
$\mathbf{X}$	Feature vector extracted from a window

Table 3.1: Notation reference

a Poisson distribution, then revisiting at the Poisson mean would be an optimal revisit policy.

In our baseline revisit policy, we took into account the last made post whenever we make a visit to the thread, and calculate our next revisit time based on the average post intervals added to the timestamp of the previous post. This is in contrast to an even simpler revisit policy that just revisits at a constant, fixed rate, independent of the posts being made to the thread.

One other way of predicting using average post intervals would be to use the concept of a *window*. Averaging out the time differences between the posts would intuitively work, because it captures the context of the situation: A series of posts with short intervals should mean that the next post would come at around the same interval as the few that came before.

In terms of using content for prediction, windows also make sense: Forum users view content as paginated posts, so time differences between posts do not affect their decision to post. Rather, reading a number of posts together affect whether or not the user chooses to reply.

An example of a window ($w = 2$) can be seen in Figure 3.1.

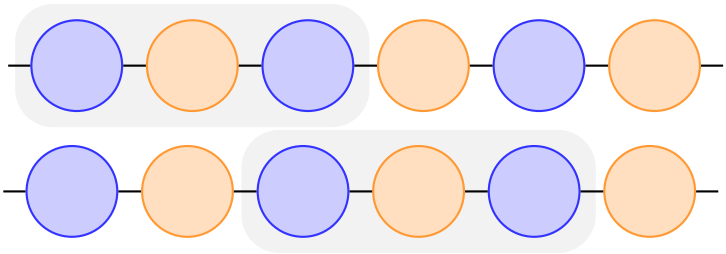


Figure 3.1: A series of events, posts (blue) and visits (orange). The diagram demonstrates the concept of a window of $w = 2$.

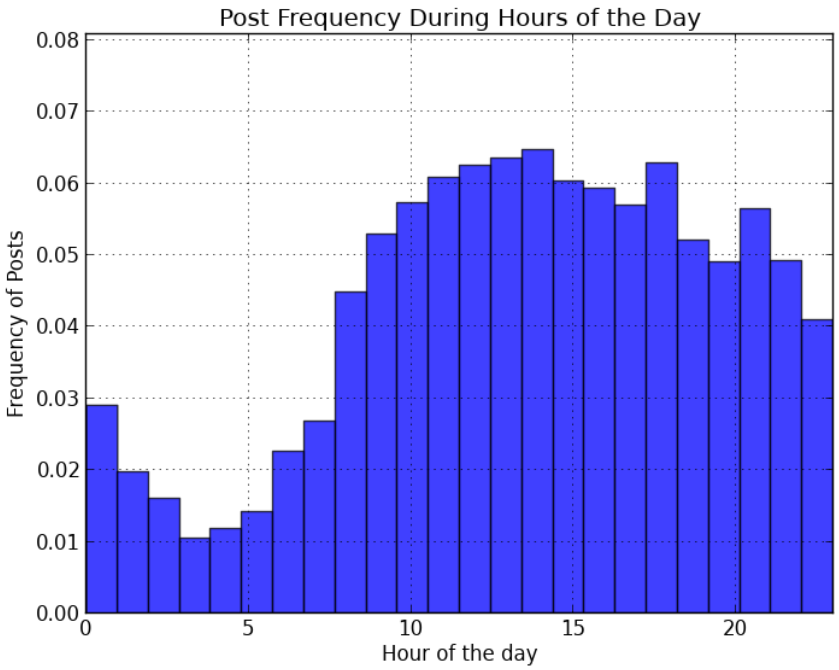
3.2 Performing regression on windows

Previous work has used linear regression on a number of different features extracted from forums (Yang et al., 2009). In their paper, the regressed function was used as a scoring function rather than a predictive function. In site of this, we attempted to implement the same model, but this resulted in evaluations worse than that of the baseline.

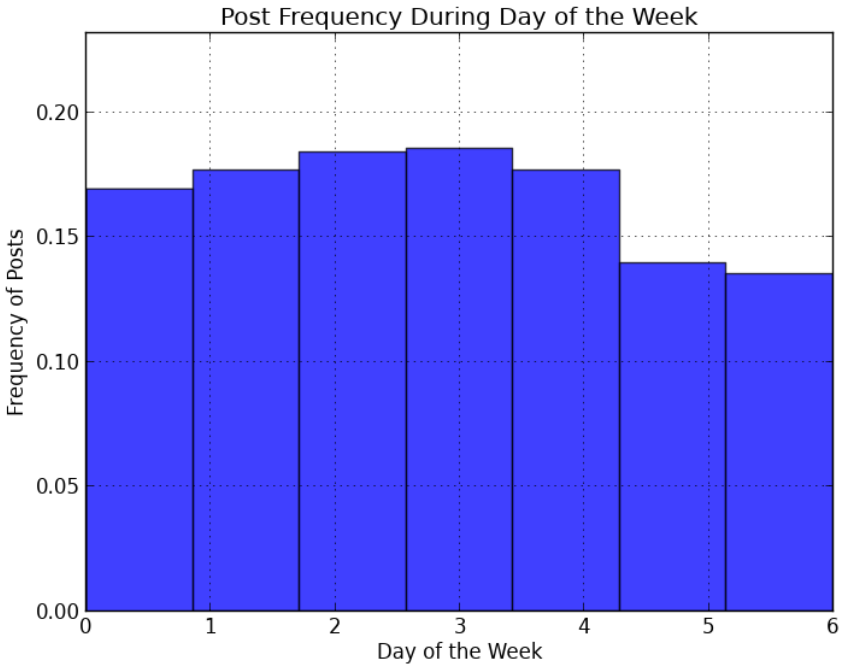
We did use some of the features mentioned in the paper: Window posts time differences and time context features (bit-vector representations of the day of the week and hour of the day). In our own statistics we took from the `avsforum.com` threads, we have also found that the time of the day the day of the week matters when dealing with threads. An example of such a thread can be seen in Figure 3.2a and Figure 3.2b, where it can be seen that activity on the board is highest at 2 PM, and drops slightly, suggesting some type of lunch period, and then goes up again during the early evening and at 9 PM, before dropping to its lowest at 3 AM. The weekly graph also shows a pattern, showing lower posting frequencies during the weekends, and its highest during Thursdays.

This suggests that the *time context* of which the posts were made are important when trying to determine the rate of posts. As such, we factor in the day and hour information into our feature sets as well.

However, this time in stead of linear regression, we used a regression method known as Support Vector Regression (SVR), using a radial basis function kernel. This method allows the using of different kernels, allowing for better estimation of the target function.



(a) The hourly post frequency for the hours during the day.



(b) The daily post frequency for the hours during the week. (Monday is 0)

	MAPE	T-score	Visit/Post
$K = 5, \mathbf{v}$	9.163 ± 1.446	1499.524 ± 186.353	19.347 ± 8.899
$K = 10, \mathbf{v}$	9.175 ± 1.447	1498.429 ± 186.397	19.347 ± 8.899
$K = 15, \mathbf{v}$	9.154 ± 1.446	1503.877 ± 186.800	19.347 ± 8.899
$K = 20, \mathbf{v}$	9.126 ± 1.446	1503.899 ± 186.814	19.347 ± 8.899
$K = 25, \mathbf{v}$	9.129 ± 1.448	1501.925 ± 186.800	19.348 ± 8.899
$K = 30, \mathbf{v}$	9.147 ± 1.449	1502.578 ± 186.782	19.349 ± 8.899
$K = 35, \mathbf{v}$	9.174 ± 1.448	1501.913 ± 186.811	19.349 ± 8.899
$K = 40, \mathbf{v}$	9.172 ± 1.451	1501.942 ± 186.818	19.349 ± 8.899
$K = 45, \mathbf{v}$	9.175 ± 1.450	1496.252 ± 186.440	19.350 ± 8.899
$K = 50, \mathbf{v}$	9.172 ± 1.448	1495.567 ± 186.449	19.352 ± 8.899

Table 3.2: Experiment results: Varying vocabulary size

The main focus of study in this report was to see if content helps with predicting thread updates would produce an improvement. Some of the ways that content data were extracted into feature vectors are the following: Word frequency, the tf-idf of these words, and Part-of-Speech tags.

We perform the standard preprocessing steps like removing stopwords and tokens of length less than three. We also use Porter's stemming algorithm as another preprocessing step, before performing a word frequency count. However, the use of the full vocabulary of the thread as a feature vector greatly increases the time needed to train the model. As such, we used a simple univariate regression technique for feature selection, and selected only the K best tokens for consideration. Table 3.2 shows the results of this experiment.

These feature sets are used in different combinations, with different window sizes. The results will be seen in the next chapter.

The methods in this section use features extracted from the current window. A model is then trained using these extracted features in order to make a prediction. We take a look now at a two other novel methods that we developed.

3.3 Discounted sum of previous instances

Posts made further in the history of the thread may have an effect on when the latest posts arrive. The magnitude of this effect, however, may diminish over time.

Instead of having a finite window for which all posts (in said window) are treated equally, why not try to account for all previous posts, but weigh them accordingly: the earlier they were made, the less weightage on the prediction the post should have.

Following this intuition we used a discounted sum over previous posts' word frequency vector:

$$\mathbf{X}'_t = \mathbf{X}_t + \alpha \mathbf{X}'_{t-1}$$

where $\mathbf{X}_t$ is the feature vector at post t. α is the *discount factor* and satisfies $0 \leq \alpha < 1$.

This new feature vector $\mathbf{X}'_t$ will be used in the same way as before, instances of $\mathbf{X}'$ will be regressed with their Δ_t values. As before, we will look at the results for this method in the next chapter.

Up till now, we have looked at methods that treat the model as static – once trained, the model never gets updated during run time. However, this is unrealistic due to the fact that over time, different words are popular as a direct result of different topics in the real world being popular. In this case, these fluctuations may be due to new updates to firmware being released or newer models of, say, a stereo set.

3.4 Stochastic Gradient Descent

We also used stochastic gradient descent to estimate the function f. However, during runtime, instead of using a static function, we continue to allow f to vary whenever new posts and their update times are observed.

Having already attempted using linear regression for this purpose, we have found it unsuitable for f to be estimated by a linear function. Such a linear function has often resulted in making a negative prediction, and sometimes an overly huge one, when given feature vectors that have previously never been observed. The function has to be somehow constrained such

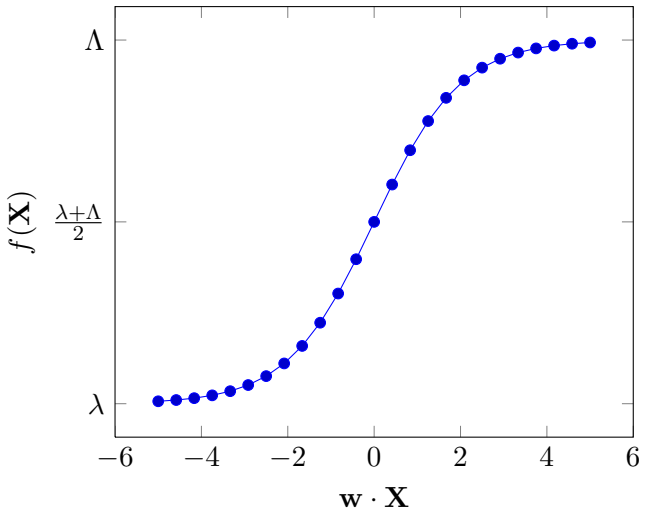


Figure 3.2: Scaled sigmoid curve

that the value returned never drops below 0, and never predicts something too huge such that many posts are missed.

Since $f(\mathbf{X}) > 0$, we used a scaled sigmoid function,

$$f(\mathbf{X}) = \frac{\Lambda - \lambda}{1 + e^{\mathbf{w}\cdot\mathbf{X}}} + \lambda$$

where Λ and λ are the scaling factors. This results in $f : \mathbb{R}^{|\mathbf{X}|} \to (\lambda, \Lambda)$. Bounding the estimation function between λ and Λ allows us to restrict the prediction from becoming negative, or, becoming exceedingly huge. For our purposes, we set $\lambda = Q_3 + 2.5(Q_3 - Q_1)$, where Q_n is the value at the n-th quartile. A visual interpretation of such a curve can be seen in Figure 3.2.

The resulting update rule for $\mathbf{w}$ is then given by,

$$\Delta\mathbf{w}_i = \eta \underbrace{\left(\widehat{\Delta_t} - \Delta_t\right)}_{\text{error term}} \underbrace{(f(\mathbf{X})(1 - f(\mathbf{X})))}_{\text{gradient}} \mathbf{X}_i$$

which is similar to the delta update rule found in artificial neural networks. We omit the scaling factor in the gradient as it is a constant and then experiment with various values of η, the learning rate.

In this chapter, we have outlined the specific task we will be attempting, to try and predict the time from the current last post in the thread to the next. We have discussed the types of

features we will be using, time context features, with a focus on content features that consists mainly of the tokens present. We have also discussed the concept of a window, and how it could help to make predictions better. Also, two novel methods were discussed, and, in the next chapter, we will look at how these methods stack up against one another.

Chapter 4

Evaluation

One of the contributions of this project was also to come up with a good metric for measuring the performance of a model that performs predictions.

Our metric has to be different from traditional methods of measuring performance. One example of such a measure is Mean Average Percentage Error (MAPE), which we use to measure the performance of the learnt model. This value is given by

$$\frac{1}{|P|}\sum_{t=1}^{|P|}\left|\frac{f(\mathbf{X}_t)-\Delta_t}{\Delta_t}\right|$$

where A_i is the actual value, and F_i is the forecasted value for the instance i. Realistically, the model would not be able to come into contact with every possible window, since chances are it will make an error that causes it to visit a thread late, causing it to miss two posts or more.

Notation	Description
P	List of posts.
V	List of visits.
T	A thread's T-score.
$T_{\max}$	A thread's maximum T-score.
$t(\rho)$	Timestamp of the post.

Table 4.1: Notation used for evaluation metrics

This value does not reflect how well the model will do in a real-time setting, but gives an idea of how far off the model is given a window.

In such measures, the performance of the model is measured on an instance by instance basis. To give a concrete example, say we are attempting to predict stock prices. Given the feature vector as input, we get an estimate of what the stock prices will be for, say, the next day. We can then measure the absolute difference between what was predicted and the actual amount, and evaluate the model based on that.

In our case, we want to know how long it takes before any post made will be retrieved by the crawler. We also want to ensure that the model does not choose to make too many requests. The rest of the chapter explains in detail how we came up with our metric, its advantages and limitations.

4.1 Potential errors

To be thorough, let us also enumerate the types of errors that a model making predictions could encounter.

The model can potentially make a prediction such that the next visit comes before the arrival of the next post. The predictions being made are the Δ_t between the posts, rather than the visitation times, hence, it is possible for the model to make a prediction that occurs before the current time. An erroneous prediction can also cause the crawler to come in before the next post (two, or more, visits, but nothing new fetched). Errors of this type waste bandwidth, since the crawler will make an unnecessary visit to the page.

Another type of error would have the prediction causing the next visit to come some time after a post. Since most predictions are almost never fully accurate, there will be some time between the post is made and the page is fetched. These errors are still relatively acceptable, but the time difference between the post arriving and the visit should be minimised. The visit could also come more than one post later. Errors of this kind incur a penalty on the freshness of the data, more so than the after one post, especially if the multiple posts are far apart time-wise.

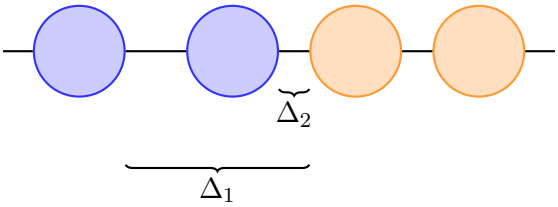


Figure 4.1: An example of a series of events used in our evaluation.

4.2 T-score, and the Visit/Post ratio

We also want to know the *timeliness* of the model's visits. Yang et al. (2009) has a metric for measuring this. Taking Δt_i as the time difference between a post i and it's download time, the timeliness of the algorithm is given by

$$T = \frac{1}{|P|} \sum_{i=1}^{|P|} \Delta t_i$$

A good algorithm would give a low T-score. However, a crawler that hits the site repeatedly performs well according to this metric. The authors account for this by setting a bandwidth (fixed number of pages per day) for each iteration of their testing. In our experimental results, we also take into account the number of page requests made in comparison to the number of posts.

4.3 Normalising the T-score and Visit/Post ratio

We normalise the T-score to get a comparable metric across all the threads. In order to do this, we consider again the thread posts and visits as a sequence of events. We then define the *lifetime*, denoted as l, of the thread as the time between the first post and the last post. Any visits that occur after the last post are ignored.

We then consider the worst case in terms of timeliness, or misses. This would be the case where the visit comes at the end, at the same time as the post. So we get a value $T_{\max}$ and

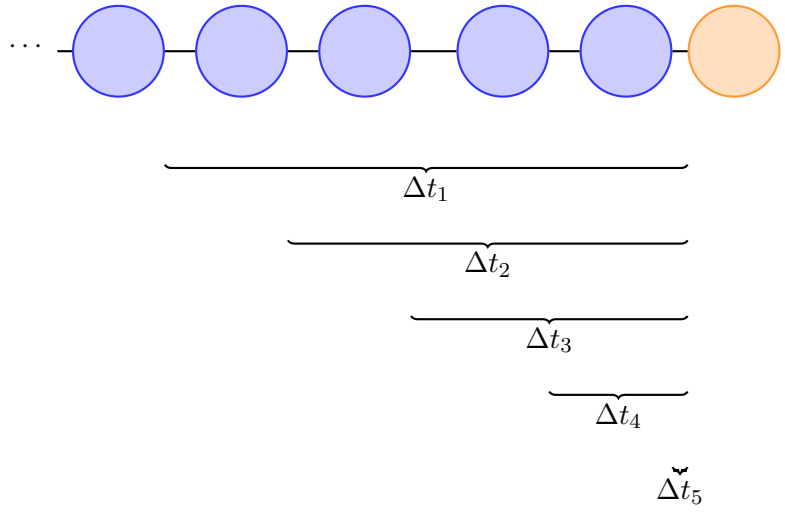


Figure 4.2: An example of calculating $T_{\max}$. A visit is assumed at the same time as the final post made, and the usual T-score metric is calculated

P_{miss} such that

$$\begin{aligned} Pr_{\text{miss}} &= \frac{T}{T_{\max}} \\ &= \frac{T}{\left(\frac{\sum_{\rho}(\max_{\rho'} t(\rho') - t(\rho))}{|P|}\right)} \\ &= \frac{|P| \cdot T}{\sum_{\rho}(\max_{\rho'} t(\rho') - t(\rho))} \end{aligned}$$

An example can be viewed in Figure 4.2. Assuming that there are no posts before ρ_1 here, we simply take the usual T-score value to get $T_{\max}$ It is difficult to consider the worst case in terms of false alarms, or visits that retrieve nothing. There could be an infinite number of visits made if we are to take the extreme case. In order to get around this, we consider discrete time frames in which a visit can occur. Since for this dataset, our time granularity is in terms of minutes, we shall use minutes as our discrete time frame. With this simplified version of our series of events, we can then imagine a worst-case performing revisit policy that visits at every single time frame. Here, we assume all quantities are measured in terms of minutes. This gives us

$$Pr_{\text{FA}} = \frac{|V|}{(\max_{\rho} t(\rho)) - |P|}$$

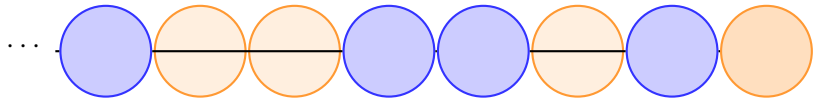


Figure 4.3: An example of calculating the maximum number of visits given a thread. The ratio between the number of visits predicted and the number of visits to the thread, and is used as Pr_{FA}

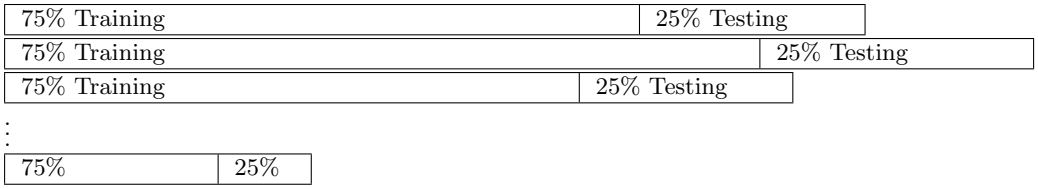


Figure 4.4: Our experiment setup

where l is in units of:our specified discrete time frame. Figure 4.3 shows an example of how Pr_{FA} is calculated.

With these two normalised forms of the original metrics, we can use the harmonic mean to give a weighted combined form of the two error rates, Pr_{error}:

$$Pr_{\text{error}} = 2 \cdot \frac{Pr_{\text{miss}} \cdot Pr_{\text{FA}}}{Pr_{\text{miss}} + Pr_{\text{FA}}}$$

In the following sections, we will discuss the results of our experiments with the various algorithms found in the previous chapter, and measure their effectiveness using their T-scores and Visit/Post ratio, and comparing them using the Pr_{error} metric.

4.4 Experiment setup

The first 75% of the thread was used as training data, while the remaining 25% was used as test data. We used Support Vector Regression for this regression task, employing a Radial Basis Function kernel as our learning algorithm.

4.4.1 Parameter Tuning

Before we begin performing experiments on the full dataset, we first tuned the machine learning algorithms using a sample of the forum threads. In the following experiments, the threads chosen

	Pr_{error}	T-score	Visit/Post
$w = 1$	0.498 ± 0.002	1576.082 ± 253.300	18.267 ± 7.290
$w = 5$	0.498 ± 0.002	1541.595 ± 232.272	17.907 ± 7.508
$w = 10$	0.499 ± 0.002	1488.688 ± 196.648	18.371 ± 7.947
$w = 15$	0.500 ± 0.002	1443.138 ± 183.408	19.234 ± 8.805
$w = 20$	0.499 ± 0.001	1584.171 ± 227.209	18.880 ± 8.602

Table 4.2: Some results

from our extracted dataset are those with a 100 to 1000 posts. This amounted to 97 threads. In each of these experiments, we run the algorithm with different parameters, and use the optimal one in our final evaluation.

Window size

Using a combination of feature sets, we experiment with different window sizes, $w = 1, 5, 10, 15$.

Performing the experiment using only the Δ_t values within the window, we obtain the results found in Table 4.2. The results show that $w = 15$ provide the best T-score. We must however, keep in mind that its Visit/Post ratio is the highest, but also has a higher standard error.

Using only the content, we perform the same experiment again. Since the size of the vocabulary is large, we select the $K = 50$ best tokens to consider using Univariate feature selection. This gives us the results in Table 4.3. The best T-score here does not do as well as that in the previous experiment. However, it is interesting to note that, again, $w = 15$ results in the best T-score.

For our final experiment for tuning the window size, we combine the various feature sets together. We also include the time-context in this experiment, and we arrive at the results found in Table 4.4. Again, $w = 15$ has the best T-score, but only with a slight improvement over our first experiment.

In any case, this suggests that $w = 15$ may be the best window size. In the following

	Pr_{error}	T-score	Visit/Post
$w = 1$	0.498 ± 0.002	1576.082 ± 253.300	18.267 ± 7.290
$w = 5$	0.498 ± 0.002	1541.595 ± 232.272	17.907 ± 7.508
$w = 10$	0.499 ± 0.002	1488.688 ± 196.648	18.371 ± 7.947
$w = 15$	0.500 ± 0.002	1443.138 ± 183.408	19.234 ± 8.805
$w = 20$	0.499 ± 0.001	1584.171 ± 227.209	18.880 ± 8.602

Table 4.3: Some results

	Pr_{error}	T-score	Visit/Post
$w = 1$	0.498 ± 0.002	1537.992 ± 251.250	18.272 ± 7.291
$w = 5$	0.498 ± 0.002	1541.587 ± 232.271	17.907 ± 7.508
$w = 10$	0.499 ± 0.002	1488.669 ± 196.646	18.371 ± 7.947
$w = 15$	0.500 ± 0.002	1443.130 ± 183.407	19.234 ± 8.805
$w = 20$	0.499 ± 0.001	1584.171 ± 227.209	18.880 ± 8.602

Table 4.4: Some results

experiments, this will be our w value.

Decay factor

In our discounted sum method, we have to tune the α parameter. We search through 0.1 to 0.9 (inclusive) with 0.1 increments to find the best possible value for α. We used the combined set of features for this experiment. The results are shown in Table 4.5.

$\alpha = 0.9$ performs the best, but its improvement over the rest of the values for α are not by much. Also, note that the T-scores do not defer much from the previous experiment, although there is a slight improvement.

	Pr_{error}	T-score	Visit/Post
$\alpha = 0.1$	0.500 ± 0.002	1443.129 ± 183.407	19.234 ± 8.805
$\alpha = 0.2$	0.500 ± 0.002	1443.127 ± 183.407	19.234 ± 8.805
$\alpha = 0.3$	0.500 ± 0.002	1443.126 ± 183.407	19.234 ± 8.805
$\alpha = 0.4$	0.500 ± 0.002	1443.124 ± 183.406	19.234 ± 8.805
$\alpha = 0.5$	0.500 ± 0.002	1443.121 ± 183.406	19.234 ± 8.805
$\alpha = 0.6$	0.500 ± 0.002	1443.119 ± 183.406	19.234 ± 8.805
$\alpha = 0.7$	0.500 ± 0.002	1443.116 ± 183.405	19.234 ± 8.805
$\alpha = 0.8$	0.500 ± 0.002	1443.112 ± 183.405	19.234 ± 8.805
$\alpha = 0.9$	0.500 ± 0.002	1443.107 ± 183.404	19.234 ± 8.805

Table 4.5: Some results

Learning rate for Stochastic Gradient Descent

Because of the scaling factors applied to the sigmoid function, a small change in the exponent of e results in huge fluctuations. As such, we need to find a small enough learning rate such that the predicted values do not end up at only the extremes, but large enough such that the model is adaptive enough to "react" to changes.

In this experiment, we find that $\eta = \eta = 5 \cdot 10^{-8}$ is the best value for the learning rate. Also note that this model produces the best results for the sample dataset.

In the following section, we will describe the experiment performed on the full dataset.

4.5 Experiments

	Pr_{error}	T-score	Visit/Post
$\eta = 5 \cdot 10^{-5}$	0.499	1595.563	19.097
$\eta = 5 \cdot 10^{-6}$	0.501	1525.705	19.122
$\eta = 5 \cdot 10^{-7}$	0.502	1440.440	19.121
$\eta = 5 \cdot 10^{-8}$	0.501	1407.172	19.108
$\eta = 5 \cdot 10^{-9}$	0.502	1416.182	19.110
$\eta = 5 \cdot 10^{-10}$	0.501	1451.729	19.106
$\eta = 5 \cdot 10^{-11}$	0.501	1482.868	19.104
$\eta = 5 \cdot 10^{-12}$	0.501	1487.555	19.104

Table 4.6: Some results

Chapter 5

Conclusion

5.1 Future Work

References

Brewington, B. E., & Cybenko, G. (2000). Keeping up with the changing web. *Computer*, , 2000, 52–58.

Cao, L., & Tay, F. (2003). Support vector machine with adaptive parameters in financial time series forecasting. *Neural Networks, IEEE Transactions on*, *14*(6), 2003, 1506–1518.

Cho, J. (1999). The evolution of the web and implications for an incremental crawler. *Science*, , 1999, 1–18.

Cho, J., & Garcia-Molina, H. (2003). Effective page refresh policies for Web crawlers. *ACM Transactions on Database Systems*, *28*(4), December, 2003, 390–426.

Cho, J., & Garcia-molina, H. (2003). Estimating Frequency of Change. (650), 2003.

Coffman, E., & Liu, Z. (1997). Optimal robot scheduling for web search engines. *Sophia*, , 1997.

Georgescul, M., Clark, A., & Armstrong, S. (2009). An analysis of quantitative aspects in the evaluation of thematic segmentation algorithms. *... of the 7th SIGdial Workshop on ...*, (July), 2009, 144–151.

Naaman, M., Boase, J., & Lai, C.-h. (2010). Is it really about me?: message content in social awareness streams. *Proceedings of the 2010 ACM conference ...*, , 2010, 189–192.

Sriram, B., & Fuhry, D. (2010). Short text classification in twitter to improve information filtering. *... in information retrieval*, , 2010, 4–5.

Tan, Q., Zhuang, Z., & Mitra, P. (2007). Designing efficient sampling techniques to detect webpage updates. *Proceedings of the 16th*, *1*(3), 2007.

Wang, A., Chen, T., & Kan, M. (2012). Re-tweeting from a Linguistic Perspective. *NAACL-HLT 2012*, , 2012.

Wang, L., & McCarthy, D. (2011). Predicting Thread Linking Structure by Lexical Chaining. *Proceedings of the*, , 2011, 76–85.

Wolf, J., Squillante, M., & Yu, P. (2002). Optimal crawling strategies for web search engines. *on World Wide Web*, , 2002.

Yang, J., Cai, R., Wang, C., & Huang, H. (2009). Incorporating site-level knowledge for incremental crawling of web forums: A list-wise strategy. *on Knowledge*, , 2009, 1375–1383.

Appendix A

Code