Efficient Processing of Time Weighted Scoring Events

Mike Koss

Created: Sept 22, 2008
Updated: October 1, 2008

Background

Diverse systems have a requirement to calculate relative "popularity" or "relevance" scores based on a sequence of time-based events. The general pattern assigns a score or weight to different events that can occur in the system, and then accumulates those scores over time to calculate a relative popularity of a given object, piece of content, user, or the like. These systems can then report on the most popular pieces of content over distinct time periods (e.g., today, this week, this month, this year).

For example, a shopping web site can record purchase events for individual products in their catalog. By accumulating these scores, they can then display the most popular products that are currently being sold, making it easier for their customers to find the most likely products to be purchased.

The current state of the art for calculating popularity scores entails tabulating individual scoring events in a database. In order to discriminate recent scores from past scores, a log is maintained that counts all events for each object and for each time period. The log can then be queried to return the sum of the counts for the time period of interest (e.g., "the last 7 days").

This solution has several inefficiencies. Data storage is required for each time interval and object of interest. Storage can be reclaimed by periodically scanning all the old data and either removing it or summarizing it into larger time intervals but at the cost and complexity and of reading and re-writing a potentially large database file. A potentially complex query is required to retrieve the data for a given time interval. For example, displaying the most popular products over the last 30 days, would require storing data for each individual day for each individual product, adding the counts for the last 30 days, and sorting those values into descending order (highest to lowest).

An alternate method entails storing time-weighted, exponentially decaying values to each object to be scored. In this embodiment, a single value, a cumulative score, is maintained for each object. The score is updated by

adding the value of new scoring events to the cumulative score. At the end of each time interval of interest the current scores are all updated by multiplying them by a constant scaling factor, k, between 0 and 1. For example, if k = 0.5, older values are depreciated by 1/2 during each time interval. In effect, the older scoring events will become less and less valuable in increasing the reported popularity over time, in favor of scoring for more recent events.

This alternate solution has the benefit of requiring only a single unit of storage for each object to be scored, but has the drawback of requiring that all the scores be updated each time interval in order to apply the depreciating scale factor. When the population of objects become large, this can represent a large data processing cost to update the entire database on a daily basis.

News sites (like Digg and Reddit) use algorithms so that recency of submission is a dominant factor. Their goal is to show only the most popular NEW stories on their front pages. The Digg algorithm appears to use a weighted average of the number of up votes (diggs) and down votes (burys) in order to determine popularity of a story. The date of submission of a story is used to filter all stories that were submitted today, this week, this month, or this year. Reddit's scoring function is based on the log of the number of votes plus a factor based on the date of the submission (log(v) + t/K). It has the advantage that each story's score can be updated in real time, stored and indexed by the underlying (Postgres) database. Because each submission is achored to a fixed submission date, the weight for all voting events for a particular story is equal, regardless of the recency of the vote.

Reference: http://www.seomoz.org/blog/reddit-stumbleupon-delicious-and-hacker-news-algorithms-exposed

Definitions

An *object* is a data representation of a physical or logical entity for the purposes of performing data processing about the entity. Objects can represent people, content items (news stories, blog posts, products, web pages, and the like), products, services, tags, keywords, and the like.

An *event* is any action, whether user generated or automatic, that occurs at particular time. Events can be differentiated by type and may be associated with one or more objects. Examples include a user clicking on a hyperlink, voting "up" or "down" a news story, accessing a web page, silencing or blocking a user or topic, marking content as a "favorite", sending content to a fiend, or commenting on content.

An *event stream* is a time-ordered sequence of events.

A *score* is a numeric value that is associated with an event. The size of the score can depend on the type of event as well as by attributes of the objects with which the event is associated.

A *cumulative score* is a weighted sum of scores in an event stream that apply to a particular object. The weights are determined by the time at which each event occurs.

A *half-life* is a time interval in which the weights of cumulative scores at the beginning of the interval are 1/2 the value of weights of cumulative scores at the end of the interval.

Summary

In order to calculate an estimate of the cumulative score over time, a time-weighted coefficient is assigned to each past score. The time-weight is based on a exponential decay function such that events that occur in the past have an exponentially decreasing coefficient, based on their age. Events that occur t units in the past, have a coefficient of 0.5^(t/h), where h is the half-life of the cumulative score. Events that occur h units in the past have a coefficient of 1/2. Events that occur 2*h units in the past have a coefficient of 1/4, etc.

The present embodiment calculates cumulative time-weighted scores from an event stream, for a collection of objects. In order to compare cumulative scores in a database, the present embodiment additionally stores a time-normalized form of the cumulative score. A starting time is chosen for the system as one which is earlier than all scoring events that will occur in the system. The weight for events that occur at the starting time are assigned as "1", and exponentially increase (double) each h units of time. So that time-weights are calculated as 2^((t-start)/h). In order to bound the size of the data storage requirement for exponentially increasing time-weighted scores, the log (base 2) of the cumulative score is stored. Since the log(x) is an increasing function of x, comparison of log(a) < log(b) has the same truth value of comparing a < b. These time-normalized values can be stored in a database table, indexed and sorted to allow for efficient retrieval of objects with the highest (and lowest) cumulative scores.

The embodiment can also compute as estimate of the rate of scoring values being accumulated by any single object over a recent time interval.

For example, a news service can assign a score of "1" to a news story whenever it is read by a subscriber. Over time, some stories accumulate higher scores than others. If the subscribers (or publishers) of the news service desire to see the most popular stories over the past "day", "week", "month", or "year", this embodiment can efficiently retrieve that list. For a given news story, the news service could display an estimate of the number of times that story has been read over the last day, week, month or year.

Only a constant amount of storage is required for each object being scored, yet, no re-writing of the database representation is required beyond the calculation of individual scoring events. Standard database indexing and querying can be used to retrieve the objects with the highest score over a desired time interval. The characteristic half-life of a scoring event can be tuned for different applications. Multiple distinct half-lives can be maintained simultaneously with only constant storage required for each object.

Definition of Variables

Variable	Definition
t	Time (e.g., units may be in days, beginning with 0).
h	The characteristic half-life of a value. This *time constant* determines how valuable a past event is compared to a present event.
v(t)	The intrinsic (non time-weighted) score of an event or action (taken at time, t).
S(t)	The time-weighted, net present sum of all actions up to and including time, t.
k	The depreciation factor over one unit of time: $(1/2)^{1/h}$ For example, when h=1, k=0.5. When h=7, k=0.906. $0 < k < 1$

Formulation

The cumulative score is calculated as:

$$S_n = \sum_{t=0}^{n} (1-k)k^{n-t}v_t$$

The time-normalized cumulative score is calculated as:

$$S'_n = \frac{S_n}{k^n} = \sum_{t=0}^{n} \frac{1-k}{k^t}v_t$$

The log (base 2) of the time-normalized cumulative score is calculated as:

$$log(S'_t) = log(\frac{S_t}{k^t}) = log(S_t) - tlog(k)$$

$$= log_2(S_t) + \frac{t}{h}$$

Calculation

To organize the calculation of log(S'), the following values are stored for each scored object (in a database or in-memory data structure):

1. S - the cumulative score at time of latest update
2. Last - time of last update
3. LogSp - log (base 2) of the time normalized cumulative score

```
// Time scale constants based on selected half-life time-constant
Score.h = 3;
Score.k = Math.pow(1/2, 1/Score.h);

Score.log2 = Math.log(2);

// Initialization
function Score()
{
   this.S = 0;
```

```
    this.Last= 0;
    this.LogSp = 1;
}

// Update the cumulative score for an event scoring value, v, at
time, t.
Score.prototype.Update = function(v, t)
{
    // Scoring event more recent than any past event
    if (t > this.Last)
        {
        this.S = (1-Score.k)*v + Math.pow(Score.k, t-this.Last) *
this.S;
        this.Last = t;
        }
    // Scoring event older than the most recent scoring event
    else
        {
        this.S += (1-Score.k) * Math.pow(Score.k, this.Last-t) * v;
        }

    this.LogSp = Math.log(Math.abs(this.S))/Score.log2 + this.Last/
Score.h;
    if (this.S < 0) this.LogSp *= -1;
}
```

Retrieving the objects with the highest valued cumulative scores is as simple as executing a standard SQL query such as:

```
SELECT * FROM Scores ORDER BY LogSp DESC LIMIT 100;
```

Variations

An embodiment may use an in-memory data structure, such as a heap, or a priority queue, to store cumulative scores. This would allow for very fast (constant time) calculation of the highest (or lowest) scoring objects. An embodiment may store cumulative scores in a SQL database (such as MySQL, or SQL Server, or the like), and create clustered indices such that querying for the highest (or lowest) scoring objects is optimized for fastest retrieval. An embodiment may store cumulative scores in a distributed database (such as Google AppEngine or AWS Simple Table).

Event scoring can be performed online (in real time) by writing directly to an online database of cumulative object scores. Alternatively, scoring events can be logged and applied by an offline (batch) process.
An embodiment may shard the cumulative scores among distributed database records to increase scalability and reduce contention. Components of the cumulative sum can be stored in distinct storage locations (possibly located on different processing units). A combined cumulative score can be calculated by the following method:

```
// Combine cumulative scores from distributed cumulative scores
Score.prototype.CombineScores(a)
{
   for (var i = 0; i < a.length; i++)
      this.Update(a.S, a.Last);
}
```

Trend Data

In addition to raw popularity, trend information can be extracted by comparing scores for different time scales. The scores at different time scales are already normalized to the average value per unit time. By storing differences or ratios of the scores, the database can be queried to find the scores that have increased (or decreased) the most in the most recent time interval, compared to the longer time-scale interval.

```
[insert formulas and code]
```

Applications

Time weighted scoring can be used in a broad variety of applications, including the following examples:

1. Ranking the popularity of news stories on a news web site.
2. Ranking the popularity of blog posts in a blog or on a collection of blogs.
3. Ranking the popularity of products on a shopping web site or collection of shopping web sites.
4. Ranking the popularity of a web site in a social bookmarking service.
5. Ranking the popularity of content created by a user in a forum.
6. Ranking the popularity of search terms on a search engine.

7. Ranking the popularity of tags used to categories bookmarks, blogs, or news stories.
8. Ranking the frequency of requests of an IP address in an network intrusion detection system.
9. Ranking the click-through and revenue rates of ads in an advertising distribution network.
10. Ranking the activity level of users in a social network or social network group.
11. Calculating the resource usage of a customer by a network service provider.
12. Ranking the response time of pages on a web site.
13. Generating analytics information about the most requested pages on a web site.