TimeScore - System for Processing Time-weighted Scores

August, 21, 2009

Background

Diverse systems have a requirement to calculate relative popularity or relevance based on sequences of events that occur throughout time over a population of entities to which these events are assigned. The general pattern assigns scores (or weights) to distinct events, and then accumulates these scores over time to derive a net score. The system can assign relative popularity or relevance to entities based on their relative net scores.

For example, a information processing system may contain a collection of news stories (entities). The events can consist of people selecting the most interesting headlines to read, rating the quality of the news story based on interesting-ness or relevance, recommending that their friends read the news story. Each event can be assigned a distinct weighted score (e.g., selecting a story to read can have a score of 1.0, while recommending story to a friend can carry a score of 3.0).

If each event is assigned a numerical score, the entities can record a net score based on the accumulated scores of all past events.

The problem with a simple adding up of all event scores, is that the element of time is not taking into account. A news story may exist in the system for a period of days, months, or years. What was interesting or "news" today, ceases to be so tomorrow. We desire a system that can adjust the event weights to take into account that more recent events should be weighted more highly than past events.

One approach is to "depreciate" all past scores stored in the system at specified time intervals (e.g., once per day). This has the disadvantage that the information system must read and modify all of the scores stored in the system each day to adjust them. Systems can reduce this workload by expiring entities from the system, but they risk loosing track of surges in popularity that may occur well after the expiration date of an entity.

Another approach is to keep a record of all recent scoring events, ordered by time. Periodically (e.g., nightly), net scores can be recalculated based on accumulating just the scores from the most recent time interval. It is expensive to process all scoring events (or even a summary of recent scoring events) until the event information can be discarded or ignored (after an system-defined maximum age).

Exponential Inflation

A more efficient approach is to define a baseline time, on which all scores will be interpreted. We then use an "inflation" model to increase the weights that are applied to individual events, so that an event occurring today, is given a higher relative weight to the same event occurring in the past.

In this approach, entities that are no longer receiving scoring events, need not be read or updated, yet their scores can be compared against more active entities to determine which are the most popular or relevant. This has the benefit that most databases have a method of efficiently retrieving values in descending value of an attribute of an entity (e.g., ISAM databases, SQL-based indexes, or Google AppEngine Model indexes).

A very desirable model for comparing time based scores is to apply an exponentially increasing weighting factor to event scores. Exponential growth valuations are common in financial calculations (compound interest and Net Present Value calculations). A baseline time is chosen (the earliest time for which all events in the system can occur), and assigned a weight of one (1). Future weights are calculated as an exponentially increasing function of time.

Exponential functions can be characterized by a *half-life*, the time over which the value of the weighting function exactly doubles. The difficulty of processing values that grow exponentially is that the required storage for values increases linearly over time (or exceeds the available space allocated to store each value, e.g., if using a floating point numeric representation).

To address this concern, the preferred embodiment stores the logarithm of the score, rather than the score itself. Since logarithms are monotonically increasing functions, a value, v1, that is greater than another, v1, will have a logarithm log(v1), that is greater than the logarithm, log(v2) of the other value, i.e.:

$$v1 > v2 \text{ iff } \log(v1) > \log(v2) \mid \text{for all values } v1, v2 > 0$$

Any database index created based on the log of the scores for each entity, will order values in the same order as an index created over the scores themselves.

Preferred Embodiment

In order to efficiently calculate, time-weighted exponentially increasing scores, this embodiment stores 3 values for each entity:

1. t: The time the last scoring event was accumulated.
2. Q: The quotient, the score, and the current time-weight
3. LogS: The log of the accumulated score, S.

In order to update the score of an entity, given a new event score, V2, at time t2, the following calculations are performed (assume our time-weight function has half-life, h).

If t2 > t:

$$Q = V_2 + \frac{1}{2}^{\frac{(t_2 - t)}{h}} * Q$$

$$t = t_2$$

else if t2 < t (update for an *older* event):

$$Q = \frac{1}{2}^{\frac{(t - t_2)}{h}} * V_2 + Q$$

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

An implementation of the TimeScore algorithm in Python:

```
from datetime import datetime, timedelta
import math

class ScoreCalc():
    """
    Base functions for calculating half-life values from a stream of scoring events.

    Time units are abstract and zero-based.  The default half-life is 1 time unit.

    The Net score is valid at a particular time, tLast.  LogS is globally comparable as it is
    based at time t = 0.
    """

    def __init__(self, tHalf=1.0, value=0.0, tLast=0.0):
        """
        The score cannot be 0 - since we use Log(S) as a ordering key.  Instead, all scores
        are based at value = 1 at time = 0.  Negative log scores would occur for score values less
        than 1 at time = 0 - these are not allowed.
        """
        self.tHalf = float(tHalf)
        self.k = 0.5 ** (1.0/self.tHalf)
        self.S = 1.0
        self.tLast = 0.0
        self.Increment(value, tLast)

    def Increment(self, value=0.0, t=0.0):
        value = float(value)

        t = float(t)

        if t > self.tLast:
            self.S =  value + (self.k ** (t - self.tLast)) * self.S
            self.tLast = t
        else:
            self.S += (self.k ** (self.tLast - t)) * value

        try:
            self.LogS = math.log(self.S)/math.log(2) + self.tLast/self.tHalf
        except:
            # On underflow - reset to minimum value - 1 at time zero
            self.S = 1.0
            self.tLast = 0.0
            self.LogS = 0.0
```

Differential TimeScores - Trending Scores

TimeScores can also be used to automatically identify *trending* scores with a score dataset. By calculating timescores over multiple half-life intervals (say, 1 day, 1 week, 1 month, and 1 year), the ratio of scores over different time scales can be used as an indicator of near-term or *trending* popularity.

Say, for example, that an entity regularly receives 10 units of score added to it each day. On a normal day, the ratio of the daily to the weekly score will roughly constant. But, if activity for this entity spikes to, say, twice it's normal level, then the ratio of daily to weekly scores will increase by a factor of 2.

$T = Sd/Sw$ (Trend defined as ratio of Daily to Weekly Score)

Since we are already storing the Log's of the time scores, we can easily also store and index our data set by the difference of two time scores to order or data set by the Trend value.

$LogT = Log(Sd) - Log(Sw)$

Base-line Normalized Trending

In statistics, deviations from the norm are often expressed as the square of the deviation. In the same way, the trending function of interest may be defined as a higher power of the trend ratio, for example, as the square of the ratio of two TimeScores:

$T2 = (Sd/Sw)^2$

This allows us to *normalize* trend data to preferentially have higher scores for entities that have highy baseline scores:

$T' = T2 * Sw$

$Log T' = 2 * Log(Sd) - Sw$

This technique allows us to highlight those entities that already have a relatively high baseline score more heavily, and yet still allow low-scoring entities rank highly if their near-term Trend value is large enough.

Claims

1. A method for processing scoring events for a plurality of records comprising:
    a. providing a data storage means which is able to store a collection of records including attributes for a last updated time, T, a current aggregate score, LS, and normalized score, Q,
    b. providing a characteristic half-life, H,
    c. providing a base, K, defined as (1/2)^(1/H),
    d. providing a query means whereby records can be retrieved from said data storage means according to the numerical ordering of an attribute of said records,

e. providing a data input queue means through which a series of event data can be retrieved comprising a record identifier, time, TR, and value, V, for each event,
f. providing processors which are connected to the said data storage means which can read and update said records, and can remove individual event data from said data input queue means and will:
    1. retrieve the record identified by the said record identifier, and
    2. calculate a new attributes as either:
        1. Q = V + K^(T-TR) * Q, and T = TR, if TR is greater than or equal to T, or
        2. Q = K^(T-TR) * V + Q if TR is less than or equal to T and,
    3. calculate new attribute LS = log(Q) + T/H, and
    4. store updated record back to said data storage means,
g. providing an output report comprising the ordered collection of records from said query means, where the query is ordered by descending order of the the aggregate score attributes of said records,

whereby said report will contain

2. Method of claim 1 where LS is used as a key to create an queryable index in a database.
3. The method of claim 1 wherein said output report if further restricted to retrieve only those records whose one or more other attributes are constrained to be equal to a specific value.
4. The method of claim 1 wherein said output report is limited to contain a fixed number of records comprising a prefix of the complete ordered list of records.

There may be a way to generalize the above claim 1 and break down this way:

1. *Weight scores and exponential function of time based on a fixed time base.*
2. method of 1 where logarithm used for compression of aggregate scores
3. method of 2 where t/Q/LS calculation method is used to calculate aggregate scores
4. method of 3 where reported scores are displayed normalized to the user's current time (virtual depreciation)
5. method of claim 3 where scores are normalized by (1-k) factor - scaling aggregate score to be equal to a steady stream of constant values in each period.

Other dependent claims:

1. Simultaneous calcuation of scores with different half-lives (day, week, month, year) to provide different time-scale reporting.
2. Calculation and indexing on Trend value defined as the ratio of a TimeScore with a smaller half life and a TimeScore with a longer half-life.
3. Calculation and indexing of Trend-Squared emphasizing larger excursions from the normal TimeScore value.
4. Calculation and indexing of Trend-Squared TIMES TimeScore baseline, highlight Trending entities, but preferentially those that already have a relatively high TimeScore baseline value a the longer time scale (half-life).
5. Storing multiple dimensions of timescore data - combining via sums and differences (for positive and negative scoring dimensions) - equivalent to products and quotients of underlying scores
6. Using tags to define partitions of dataset (but sharing same underlying timescores)
7. method of changing the (previously fixed) half-life for all future scores w/o having to recalculate original scores

Equation expressions for http://www.sitmo.com/latex/ here:

```
Q=V_2 + \frac{1}{2}^\frac{(t_2-t)}{h}*Q
t = t_2

Q=\frac{1}{2}^\frac{(t_2-t)}{h}*V_2+Q

LogS = log_2(Q) + \frac{t}{h}
```

(capture + and - scoring system)
(incuding flitering by tags)