**Popularity & Relevance Ranking**

**First Stage Algorithm Considering “Search Popularity” and “Recency”**

1: Input (1) : N documents to rank as text that can be scanned into the program from disk files.

e.g: 3,000 documents that we can scan into the program as text from disk files.

2. Input (2) : M\_d search strings performed for every day d, with d going back 100 days from today, as text.

e.g:

10000 search strings performed today (d=1) as text,

10000 search strings performed yesterday (d=2) as text,

10000 search strings performed day before yesterday (d=3) as text,

. . . up to 100 days going back to day d=100 (roughly 3 months,

as text that can be parsed into the program.

This is a first stage algorithm to give us an initial score each for ranking the N documents. We shall build on and improve this algorithm with more and more sophisticated models later.

\* Calculate the TD-IDF score vector for the words in each document, considering the full database of documents for base wordcounts.

\* Now each document has its top scoring word vector and a corresponding TD-IDF score vector.

\* For each day d ( d = 1..100) put all the day’s search strings together and consider them 1 search “document”.

\* For each day d ( d = 1..100) calculate the TD-IDF score vector for all the words in the searches for the day as in the above section.

\* For each day d ( d = 1..100) pick the top scoring X words and their TD-IDF values.

Thus for each one of 100 days we have an X length top scoring search words and its corresponding X length top scoring TD-IDF values.

\* For the third stage take each documents Word-TD/IDF vector. For each day d, pick the search-d’s words ( if they appear in the document i) and the corresponding TD/IDF scores.  
Now for each day we have an X-length search-word TD/IDF vector, and an X-length vector for the document’s TD/IDF values for the same words.

\* Now for this document, calculate the Cosine Similarity Score for each search-day.

\* Repeat for all documents.  
  
At the end of this algorithm we have 100 TD/IDF Cosine similarity scores – one for each day going back 100 days – for each one of our N documents.

For each document, combine each one of 100 days’ scores using exponential decay in time.

Score(n)

Where *Score(n)* is the nth document’s final score (out of N documents),

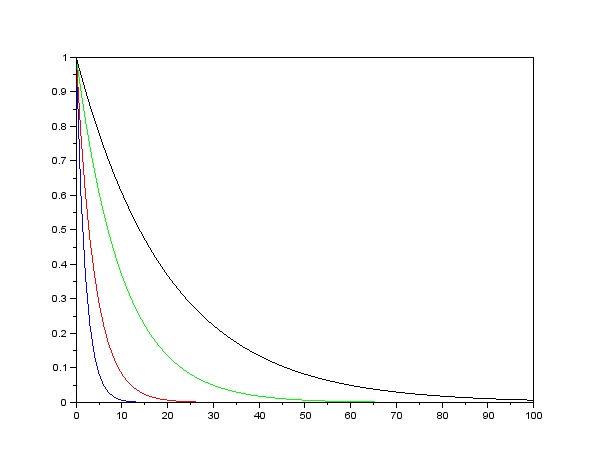
*d* is day-d out of the 100 days going back considered,

*e* is the mathematical Euler constant for exponential decay ( *e* ~= 2.7182818284), and

*CosineTdIdf(n,d)* is the similarity measure for each Document *n*, for each day *d* search vector as described above.

Above ***r*** is the exponential decay rate we will apply over the 100 days considered.

The plot below shows the exponential decay graph over 100 days along the x-axis for different decay rates for ***r***.



The decay rates are:

* Blue: r = 0.5
* Red : r = 0.25
* Green: r = 0.1
* Black: r = 0.05

Thus, by varying the decay rate ***r***, we can decrease at varying rates the impact each day’s TD/IDF Cosine score will have on the overcall score for the document ***n***.

At the end we shall index the documents in the decreasing order of their final scores.

This this is an initial algorithm for scoring and ranking that captures both

1. Search Popularity, and
2. Recency

into one score to give us an initials ranking algorithm with a strong basis in statistics and scientific method.

This also gives us the initial programming structure to adding more sophisticated models and factors to grow the Scoring algorithm in the future.

**Model Adaptation to Scoring and Ranking based on Citations & Shares**  
  
The above model was developed for search engine ranking based on search popularity.

The model can be adapted to Score and Rank on any event by the public such as:

* Shares & Citations
* Reader commenting
* Other reader actions such as LIKEs and FOLLOWs

The sharing of each article in the database in the form of its URL appearing on other articles is consistent with the model because each appearance of an article’s URL on another article can be considered a Citation or the Sharing of the first article.

Hence, the model can be easily adapted to rank on “Citation and Sharing” by way of the subject article’s URL appearing on other articles.  
  
For this adaptation the inputs need to be changed to:

* For each Subject article, the collection of other articles that Cite and Share it in the form on its URL appearing on them, and
* The date-time stamp of the articles that Cite or Share the URL of the first Subject article.

Given the above inputs the Scoring will be done as in the above formula weighted on the exponential curve on the number of days from today of the Citation or Sharing of the URL.  
  
Some regularisation will be done to remove the bias against the most recent articles which have not had the time required for citations to start appearing.

**Citation Model**

For each document, combine each one of 100 days’ scores using exponential decay in time.

Where *Score(n)* is the nth document’s (out of N documents) final score,

*t* is the hour from nor out of the 36 hours days going back articles in the database are considered,

The current hour is t=1, the past hour is t = 2;

URLs\_cited\_for\_n is the number of citations of the nth document’s URL on day d in the database,

Total\_URLs\_cited is the total number of URLs cited on day d for all documents in the database.

A is a constant to magnify the normalised fraction to significant values, e.g. A = 1000,

*e* is the mathematical Euler constant for exponential decay ( *e* ~= 2.7182818284),

r is the Decay Rate parameter as defined in the graph above. It is a real number in the range [0.00001, 0.9]. We shall vary this parameter to change the time impact on the Score.

Values for r between [0.1,0.2] provide good decay curves over 36 periods.

**Implementation Notes:**

This algorithm has now been implemented in Pickscomb, primarily in the following source code.

1. Logic/RecencyCalculator::GetRecencyCurve(uint numPeriods) method implements the Time based discounting and fits the curve based on the number of time periods we go back.

The curve is fitted in Matlab depending on the period (default 15 minute periods) over the maximum period of 5 days (120 hours) with scores after the first 36 hours heavily discounted to the last approximately 1.5 decile of the weighting.

The methods other than GetRecencyCurve(uint numPeriods) in the source file are not used and are not relevant to the model.

1. Logic/CitationStore.cs: This implements the primary Model – the Citation Based Popularity Model.  
   It implements a separate model for each Vertical (in class VerticalCitationStore) and maintains a store of the models for all the Verticals collectively (in the collective class CitationStore). Operations can be applied to the individual model for each Vertical separately or on all of the verticals together. The model is careful to expire all citations that age past the considered maximum time limits.

In method VerticalCitationStore::setTimeWeightedCitationPopularity()the model pulls together the citation scores with the time based weighting.

1. Logic/CitationPopularityRanker.cs: This executes the Model implemented in CitationStore.cs on the Pickscomb database.

(3.1) Pre Fetched Ranked Articles:

It first extracts the cited URL with their Citation Popularity scores from the Model above. These URLs are provided by the Model ranked (ordered) in the descending order of their Time Weighted Popularity Citation Scores.

The Ranker extracts all the actual Articles from the Pickscomb cloud repo for the time period in question, as set in CitationStore:MaxHoursConsidered (which defaults to 5 days = 120 hours). It then matches each URL ranked by the Model with the actual Article if it is already in the Pickscomb repository.  
  
The Ranker has an important parameter to set:

Uint CitationPopularityRanker.NPreFetched

This parameter determines how many of the ranked Articles the Ranker will pre-fetch from the Pickscomb repo and serve to the calling function. If you set this parameter to “uint.MaxValue” the Ranker will pre-fetch and serve all the Articles it finds in the Pickscomb repo within the considered time (MaxHoursConsidered = 5 days). These are served with their PopularityScore according to the Model, sorted in the descending order of this score. Thus the first Article is the most popular article with the topmost ranking. Duplicate keys are allowed: i.e. two Articles can have the same Score and, if they do, will appear sorted next to each other in the list.

The pre-fetched Articles are served to the calling function by:  
SortedList<double, Article> getTopRankedNArticles( Guid vertical )

where the key (double) is the popularity score and which is also saved in Article.PopularityScore and persisted to the repo.

(3.2) Mapped GUIDs of all Articles, scored and ranked.

If you choose to pre-fetch fewer than all the Articles found in the considered time limit, you can still get the mapped GuIDs of all the Articles in the second list served:

SortedList<double, Guid> getAllRankedArticleMetrics(Guid vertical)

Here, only the mapped Guid of the Articles and their PopularityScore are served. The calling function can then use the Guids to fetch the Articles it wants from the Pickscomb repo itself.

Once again duplicate keys are allowed: i.e. Two Articles can have the same Score and, if they do, will appear sorted next to each other in the list.

**Note:** For both (3.1) and (3.2) above CitationModel itself only serves the scored and ranked URLs that do have Citations in some Article somewhere.  
  
It is possible, however, that the Pickscomb repo contains articles within the relevant time limit that do not have any citations anywhere. The Ranker gives these Articles in the repo that have no citations a score that is **half the lowest score** of the lowest ranked Article and appends them to the bottom of the above two lists served.

(3.3) It can also be the case that the Articles corresponding to all the cited URLs are not found in the Pickscomb database. This list of Articles are served in a third list:  
Dictionary<string, RankingMetrics> getAllRankedURLNotInDB(Guid vertical)

Here the Key (string) is the URL of the popular Article not found in Pickscomb repo and the RankingMetrics contain its PopularityScore and its PopularityRanking.

This list can be served to the Crawler to find and enter popular and relevant Articles to the Pickscomb repo and facilitates Article Discovery.

(3.4) The other important statistic estimated and served by the CitationStore is the AveragePopularityScore.

It can be read directly from the static CitationStore for each vertical as:

CitationStore.citationStore.CStore( verticalGuid ).AveragePopularityScore

Or it can be extracted through the Ranker interface as:

CitationPopularityRanker. getAveragePopularityScore( Guid vertical )

(3.5) NOTE: This ranking functionality is meant to be rerun every “CitationStore.Period” minutes and it is important to rescore everything to the current time period by using “reRank()”. Otherwise the model will serve stale data stored in it that is more than a Period old and not current.

CitationPopularityRanker citationPopularity = new CitationPopularityRanker();

citationPopularity.reRank(verticalGuid);

(3.6) After the current ranking and indexing is done and the new Articles are displayed on the front page, you can reclaim the memory from the stores and the lists immediately with:

CitationPopularityRanker.clearAll()

(3.7) The number of minutes interval between consecutive runs of the Ranking and Indexing is a parameter that can be configured via “App.config”.

<appSettings>

<add key="RankingInterval" value="15" />

</appSettings>

Thus if you run the ranking and indexing every hour, set this to value 60. This is read into the “**CitationStore.Period**” property which can also be set programmatically.

The value must be in the range 15 minutes to 120 minutes, and defaults to 15 minutes in case of any errors.

**Top-N Entity Boosting**

The Top-N Entity Boosting is a process of boosting carried out on the set of Articles already scored and ranked based on Popularity or a combination of Popularity metrics.

The input to the Model is the set of Articles in the Pickscomb repository within the time frame considered each of which has been assigned a Popularity Score and ranked (ordered) in the descending order of their popularity.

The Top-N Entity Boosting module boosts or increases the score on those Articles that contain entities that have appeared within the Top N entities previously within the considered time frame.

**Not a Time Weighted Model**

NOTE: This is not a Time Weighted model because we do not have the information when, within the considered time frame, the particular Entity appeared in the TopN Entities.

To make it a Time Weighted model the Analytics module, which store the EntityCounts must store with it the time-stamp of when each particular Entity appeared, similar to how I store the timestamps in the Citation Model. With such a change that stores the timestamps of each Entity appearance, this model can also be turned into a time weighted model similar to the Citation Model.

In such a Time Weighted model the time discounting decay curve must be fitted correctly using Matlab.

**Implementation**

The model is implemented in: Logic/TopNEntityBoost.cs source file.

The N in Top-N is a parameter that is configurable via App.config as:

<appSettings>

<add key="TopN" value="19"/>

</appSettings>

and one which can also be set programmatically in the property:

int TopNEntityBoost.NumPopularEntities.

The model maintains a weighting vector topNEntityWeightings[] which first stores the Entity counts of those N entities with the highest counts. Then it normalizes the weights by dividing by the sum of these N entity counts, such that the N entity weightings sum to 1.0.

In function double getEntityBoostWeighting( Article article ) in the model takes an Article and sums up the weighting of the entities the Article contains that are among the Top N entities in the model. Thus an Article that contains all of the Top N entities will get a combined weighting of 1.0. Any Article that contains some of those entities and not all will get a weighting of a fraction between 0.0 and 1.0, with 0.0 weighting assigned to an Article with contains none of the Top N entities in it.

Clearly the more of the Top N entities an Article contains the higher the weighting score it will get; and if it contains more of the higher weighted out of the Top N entities, the higher its weighting score.

The value to be boosted by is added to each Article’s existing PopularityScore.

There are 2 ways available to boost:

1. Boosting with Popularity Bias
2. Boosting without Popularity Bias

In Popularity Biased Boosting the value added to the PopularityScore of the Article is based on its existing PopularityScore. Thus we take the Article’s own existing PopularityScore and multiply that by the Top-N weighting above to get the boosting value that will be added.

Thus in Popularity Biased Boosting an Article A1 which has the same Top-N weighting as another Article A2, but which has higher existing PopularityScoreA1 than the second (which has a lower PopularityScoreA2) will be boosted by a bigger value than the second Article. Thus this is called “Popularity Biased Boosting.”

BoostingScoreA1 = w \* PopularityScoreA1

BoostingScoreA2 = w \* PopularityScoreA2

where w is the same TopNEntityWeighting fraction,

BoostingScoreA1 > BoostingScoreA2

because PopularityScoreA1 > PopularityScoreA2

There is a further weighting parameter that determines how important the Boosting score is compared to the existing PopularityScore.

This is a parameter that is configurable via App.config as:

<appSettings>

<add key="TopNBoostPercent" value="100"/>

</appSettings>

and can also be programmatically set in the property:

double TopNEntityBoost.TopNBoostPercent

This defaults to 100% which tells us that the Boosting score is equally important compared to the Article’s existing PopularityScore.

This can be a value between 50% (which indicates that the Boosting Score added is **half as important** as the Article’s existing PopularityScore) and 200% (which indicates that the Boosting Score added is **twice as important as** the Article’s existing PopularityScore.)

This percentage value is turned into a fractional weighting between 0.5 and 2.0 (for the range 50% to 200% respectively) by multiplying by 0.01 (dividing by 100).

The input to the boosting function call is SortedList<double, Article> scoredArticles

Where the double is the existing PopularityScore and the list is sorted in the descending order of its popularity.

1. In Popularity Biased Boosting the function:

SortedList<double, Article>

boostWithPopularityBias(SortedList<double, Article> scoredArticles)

Accordingly the new score for each Article is reset by:

newScore = Article.PopularityScore

+ Article.PopularityScore \*(0.01\*TopNBoostPercent)\* getEntityBoostWeighting(Article);

The return parameter is the same list sorted in the descending order of the new Boosted PopularityScore. In the return list duplicate keys are allowed which are stored next to each other.

II. Popularity Unbiased Boosting is carried out by a second function:

SortedList<double, Article>

boostPopularityUnbiased( SortedList<double, Article> scoredArticles,

double boostingScore)

Here, instead of using each Article’s own existing PopularityScore, we enter a separate value in the second parameter to boost by. Thus this is Popularity Unbiased Boosting because 2 Articles which have the same TopNEntityBoost weighting will get added the same value:

newScore = Article.PopularityScore

+ **boostingScore** \*(0.01\*TopNBoostPercent)\* getEntityBoostWeighting(Article);

**It is very important that this second parameter entered into the function does not over dominate the Articles’ existing PopularityScores by being much larger than them in comparison**. It is not advisable for this value to be more than the existing maximum PopularityScore from Citation popularity.

In my calling function I therefore feed the average statistic from the CitationModel that I extract as:

CitationPopularityRanker.getAveragePopularityScore( Guid vertical )

into this parameter, thus ensuring that it is not much larger than the CitationPopularity of the most popular Article. Nor should it be too small in comparison to the Citation PopularityScores such that boosting becomes ineffective.

**Model Combination – Time Weighted Citation Popularity and Top N Entity Boosting**

I have carried out the combination of the 2 models’ algorithms in the source file “Ranker.cs” in:

Ranker.newRun( Guid verticalId, DateTime start, DateTime end,

CancellationTokenSource cancellationToken)

The following code section highlights how to carry out the combination:

CitationPopularityRanker citationPopularity = new CitationPopularityRanker();

citationPopularity.NPreFetched = uint.MaxValue;

var utcNow = DateTime.UtcNow;

citationPopularity.reRank(verticalId);

SortedList<double, Article> popularityRankedArticles =

citationPopularity.getTopRankedNArticles(verticalId);

double averagePopularityScore =

citationPopularity.getAveragePopularityScore(verticalId);

TopNEntityBoost booster = new TopNEntityBoost(verticalId, start, end);

booster.NumPopularEntities = 24;

SortedList<double, Article> boostedPopularityRankedArticles =

booster.boostPopularityUnbiased(popularityRankedArticles, averagePopularityScore);

OR

SortedList<double, Article> boostedPopularityRankedArticles =

booster.boostWithPopularityBias( popularityRankedArticles );

**Relatedness or Similarity Ranking**

This is the process of ranking how close the other Articles of an Article repository are to one given Article.

This is done in 3 stages:

1. First we take each Article of Document in the repository (published within the specified time frame) and break it down to a set of its content words with one unit of each word. We compile this into a vector. This vector id normally called the **Bag-of-Words model** of each Article.
2. Secondly we need to define which words in the Bag-of-Words Vector are important words that define the essence of the Document. These important words must be given higher weighting compared to less meaningful, common words like “the”, “in”, “at”, “he”, “she” etc.  
     
   To achieve this we compute the **Term Frequency, Inverse Document Frequency** **statistic** (henceforth called the **TF-IDF statistic**) for each word in the Bag-of-Words Vector for each Document.   
     
   See further details on the TF-IDF metric here: [http://en.wikipedia.org/wiki/Tf-idf](http://en.wikipedia.org/wiki/Tf%E2%80%93idf)
3. Now information in each Document in the corpus is represented by a TF-IDF weighted Bag-of-Words Vector. Then, given a particular Document in the corpus, we rank how “similar” or “related” each other Document is to the given Document by taking the **Cosine Distance between** the two Documents’ **TF-IDF weighted Bag-of-Words Vectors**.

Details of Cosine Distance is given in details here:  
<http://en.wikipedia.org/wiki/Cosine_similarity>

Details of Cosine Distance or Cosine Similarity between TF-IDF weighted Bag-of-Words Vector representations of Text Documents is given in further details here:

<http://pyevolve.sourceforge.net/wordpress/?p=2497>