**NLP Models for Popularity & Relevance Ranking**

1. **A First Stage Algorithm Considering “Search Popularity” and “Recency”**

This section details a first stage algorithm based on Time Weighted Search Popularity underlying searched article ranking used by Amazon, Google, and Yahoo! Though the approach in this section is not used in our products Pickscomb, Artemis or Optimus, it forms the basis for the NLP algorithms developed for ranking Articles based on Time Weighted Citation Popularity in the next section.

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 Citation Frequency Fraction 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 Citation Frequency Fraction 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 Citation Frequency Fraction 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 Citation Frequency Fraction values.

Thus for each one of 100 days we have an X length top scoring search words and its corresponding X length top scoring Citation Frequency Fraction values.

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

\* Repeat for all documents.  
  
At the end of this algorithm we have 100 Citation Frequency Fraction – 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)

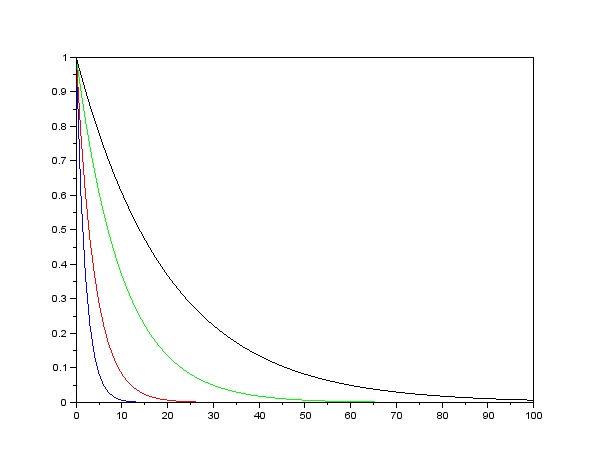
Where *Score(n)* is the nth document’s final score (out of N documents),

*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

*(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 above 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.

1. **Model Adaptation to Popularity Scoring and Ranking based on Citations & Shares Weighted in Time**

The above model was developed for search engine ranking based on search popularity. This is the core algorithm used by Amazon for ranking the popularity of its books and by Yahoo! and Google for ranking web Articles. In this section the algorithm is adapted to rank Articles based on Time Weighted Citation 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 model thus adapted for Citation Weighted Popularity Ranking is used for ranking the documents in Pickscomb, Artemis, and Optimus products as the first stage algorithm.

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.

**Adaptation for Artemis**

For Artemis the Time Weighted Citation Popularity model is adapted as follows.

* We consider citations in Articles added to the DocDB database over the past 36 hours.  
  This is defined as: Ranker.RankingHoursConsidered
* The citations can cite Articles that were published over a longer time period into the past. Currently we consider Articles added to DocDB 6 hours before the 36 hour limit. This value can be adjusted and is defined as: Ranker.RankingHoursBuffer
* We have Ranker.RankerRunPeriod which defaults to 15 minutes. This period can be set via the config file to a different value. This means that the Ranker re-ranks the article corpus and produces a new set of results every 15 minutes.
* We partition our citations time period (the 36 hours) into time slots or windows of 15 minutes each. This value can be adjusted and is defined by:   
  CitationPopularityModel.CitationWeightingPeriod

Assuming 15 minute slots, then we have T = 144 time slots in the 36 hours.

* Internal to the Ranker we use a citation concept called MatchingUrl in which we do a reduction so that the same or very similar citations will be counted together and not as separate. This is a concept used in a black-box setting entirely inside the Citation Model Ranker and has no effect on anything outside. Having them counted separately skews the model and hence this technique is used.
* For each time slot we calculate the decay rate *r* for the exponential decay curve.
* We calculate the ranking score of each cited matchingUrl *ui*  according to the following equation:
* We count the number of citations for each time slot for each citation matchingUrl *ui* and this count is the numerator.
* We count the total number of citations for each time slot and this is the denominator.
* We weight the division score by the exponential decay value for the time slot.
* Then we add up these factors for all the time slots.
* This multiply by a factor A in order that the score not be too small a fraction.
* After all the sores are calculated for all the citation URLs, we sort the vector in the descending order of the citation score, and then divide the vector by the largest citation score to normalize the score values to the range [0.0 … 1,0] with 1.0 scored URL being the most popular Article.

**Eliminations of Bottlenecks in the Popularity Ranker in Pickscomb**

The Popularity Ranker thus implemented in Pickscomb provided good results, ranking the most popular Articles with high scores and allowing them to be listed on the front page.

However, the Pickscomb architecture and implementation suffered from some serious performance bottlenecks that made the Ranker run very slowly, sometimes taking more than 40 minutes to complete.  
  
(1). In Pickscomb, the running period for the Ranker was set to 15 minutes. That means a new Ranker process was started every 15 minutes as a web job. Upon start-up the Ranker would fetch the past 36 hours of full Articles from the database, build the CitationModel from scratch every time, do the ranking, and write the results to the cache. Then the Ranking process would die.  
  
The primary bottleneck we have eliminated for Artemis is keeping the Ranker process alive instead of allowing it to die and restart every 15 minutes. Instead it starts up the first time the server is started, builds the initial model by fetching 36 hours of data, writes the results, then instead of dying, the process goes to sleep for the remainder of the time until it is time to wake up and do the next round of ranking 15 minutes from the last time it started.  
  
This means the Ranker has to fetch and build the model from 36 hours of data only the first time. The next time round it has to wake up and fetch only the last 15 minutes of Article data from the database, because the previous model is still live in memory from the last model build. The new Ranker then cleans up the expired 15 minutes of data from the Model, adds the new 15 minutes of citation data, does the calculations, writes the results, and goes to sleep.  
  
This provides us with the following performance speed-ups:

* Process wakeup from sleep is much faster than process startup.
* The Ranker does not have to fetch 36 hours of data every time. After the initial time it fetches only 15 minutes of data from the database. This eliminates a major bottleneck.
* In model building, the Ranker does not have to add 36 hours of data into the model every time. After the first time it has to add only 15 minutes of data to the model and clean up 15 minutes of outdated data from it. This speeds up the Ranking core algorithm.

In order to facilitate this, the process that starts the web job must also not die, but instead go to sleep and wake up every 15 minutes. It needs to follow the pseudocode below and only re-start the Ranker job if it has died or is frozen. Otherwise it allows the Ranker to wake up, run and sleep indefinitely.  
  
void RunRankerJob()

{

static Task rankerTask = new Task();

if ( rankerTask != null &&

rankerTask.Status.equals(TaskStatus.Running) )

{

// still running well, do nothing

}

else

{

if ( rankerTask != null && !rankerTask.IsCompleted() )

{

// Ranker in a bad state

// Cancel and kill using the CancellationToken

// then restart below

}

// start anew

rankerTask = Task.Factory.StartNew(() =>

{

// Staring Stuff

});

}

}

(2) The second and main bottleneck was the database fetch of the 36 hours of Articles into the Ranker. This alone took more than 10 minutes and often 15 minutes to complete. It was not necessary to fetch the full Articles with a lot of meta data into the Ranker. The Ranker needed only the citations, Article URL, and creation time for this part. Also the main database we were fetching the data from was a massive database of about 300 days of Articles, and selecting from this took a long time.  
  
I have reviewed the new DocDB database and it still suffers from the same problems for the Ranker. DocDB does not improve performance because I am not searching for anything other than the key and I do not need data in JSON format.   
  
Hence it is best to create 2 NoSQL Tables for the Ranker. Into the first tables named CITATIONS we shall strip the data essential to the Ranker only and store it without all the extra meta data. The Ranker will also trim this table so that data older than the hours we need is deleted.   
  
Despite storing the main corpus or Articles in DocDB it is recommended that we create a NoSQL table for the Ranker therefore.   
  
The database functions fetching from and writing to the database are separated as a black box on ArtemisRanker.RankeDbAccess.cs. Hence, it is entirely possible to fetch directly from DocDB and extract the information into the string format the Ranker is expecting without requiring any modifications to the rest of the Ranker code. However, I propose that we do implement 2 NoSQL tables for the Ranker and keep the data required in them to benefit from the optimizations.  
  
(3) In the previous architecture inter-process communications were supposed to happen through a “cache” which was part of every one of the process and attempted to update though the database. Even though the cache did not function properly because it did not write back to the database, over 6 to 7 processes attempting to access the database every few minutes clogged up database access for other processes. As I advised, the cache has been now removed and this alone should unclog and make database access faster.

(4) Communicating the results of the ranking meant updating the Score field of each Article and writing all the Articles back into the document database. The front end would then try to read back all the Articles, sort them in the descending order of the Score, and then display the top ones on the front page. These database operations through the main Document database are hugely expensive operations that are very slow because DocDB is very large.  
  
I have researched and experimented with the new DocDB database also and this is also still very slow and expensive even though the fetch at the front end might slightly be improved because we can search in the Score field.  
  
It is still recommended that we create a separate NoSQL Rankings table to write the scored articles to in purely (Article ID, Score) format, rather than do an expensive update to the main DocDB. The front end then does a very fast read of this table and fetches only the top scoring Articles it wants to display from DocDB by their IDs.  
  
It might later be possible to further improve this by using a Message Queue that the Ranker writes the Ranking results to and from with the Front End reads as a consumer.

(5) The core algorithm implementation has also been optimized for high performance.

**Implementation Notes for Artemis**  
In Artemis, the above Citation Popularity model is primarily implemented in Ranker.CitationPopularityModel.cs

The Ranker.RecencyWeightingCurve.cs implements the exponential decay curve.

* CitatationPopularityModel.cs::init() function accepts ranking data as a dictionary of string pairs where the key is the ID of the Article and the value is a string concatenating the publish datetime of the Article, the title Article’s URL, and a list of all the citations in it. This contains ranking data for Articles for the past 3 hours. This function loads the initial model with its past 36 hours or Articles’ citations.
* The RankingDataProcessor.GetCitationsFromDbString() function splits and decodes the string containing ranking data and extracts the data from it.
* Then each Article is entered into the Model. This function also builds an index that maps the title Article’s matchingURL to its unique Article ID.
* This full build of the Model from the full corpus of 36 hours of Articles happens only the first time the Ranker is started. Thereafter the Ranker sleeps, wakes up, and updates the model only with the last 15 minutes of Articles since the last iteration. This update of the model is done by: CitationPopularityModel.LoadCitationsModel()
* After the model is loaded, the calculations of the ranker according to the equation above is done by: CitationPopularityModel.SetTimeWeightedCitationPopularity()
* The results are then written into the RANKINGS NoSQL table in the (Document ID, Score) format, ordered in the descending order of the score. It is possible that later we write the results to a new Message Queue that communicates between the Ranker and the Front End.

**Serious Bottleneck – All Interprocess Communications through the main DocDB**

A main bottleneck we experienced in Pickscomb is that doing all the inter-process communications for the Ranker or for any other process through the main DocDB database. this is a design fault in Picksomb that creates and will continue to create serious bottlenecks and performance jams. We must use an alternative technique to communicate results and data, particularly between the Ranker and the FrontEnd process. Some such methods are:  
  
(1) Dedicated database tables separate from the main DocDB as proposed here.  
(2) A disk file one process locks and write to and another process then locks and reads.

(3) [Mailslots, as detailed here](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365576(v=vs.85).aspx) (follow the link).  
(4) Share [memory and pipes as detailed here](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365780(v=vs.85).aspx) (follow the link).

(5) Service Bus Queues

(6) TCP/IP Sockets communications

Hence, we must design and select an alternative mechanism without going through the main DocDB for:  
(i) Ranking Results communications between the Ranker and the Front End

(ii) Ranking data communication to the Ranker.  
  
Thus this is for inter-process communications such as passing the Ranking score results—a vector of ( DocumenrID, Ranking Score) pairs ordered by the score in the descending order—between the Ranker and FrontEnd processes.  
  
Separate NoSQL database tables is a good solution I have proposed here, but we can discuss and select an alternative technique as detailed above.

To facilitate researching and experimenting with any of the above techniques and even changing the technique, the communications processing has been abstracted and separated out as black boxes into 2 separate classes:

- RankingDbAccess.cs  
- RankingDataProcessor.cs  
  
These classes manage the inter-process communications and processing the data coming or going through the channel. As long as the data is provided to the rest of the core Ranker in the defined format, the changes in the communications technique do not affect the Ranker algorithm.

For pure Citation Popularity ranking only, the data coming into the Ranker is expected to arrive in the following format.

***IPC (Inter Process Communications) data into the Ranker:***

*Key:* a string, the unique Article ID

*Value:* a string concatenating the following

“<Article publish date timestamp>|<Title Article URL>|<Citation URL 1>|<Citation URL 2>| . . . |<Citation URL N>”

***IPC Data from the Ranker to the FrontEnd process:***  
  
This is a vector of ( Document ID, Score) pairs ordered in the descending order of the Score.

1. **Top-N Entity Boosting**

This section is for the implementation in Pickscomb, yet to be adapted for Artemis.

***Planned Adaptation for Artemis***

This technique is planned to be adapted for Artemis, but only after the Popularity Ranking, TF-IDF Similarity Ranking, and their combination have been fully implemented.

The main adaptation planned is to boost for citations by “Authority” or more popular sites.  
  
Currently we weight citations by a private tweeting user and by a site like the New York Times with equal weighting. This should not be so. We obviously value citation by the New York Times a lot more than one by a private Tweeter.  
  
Thus we need to maintain a list of Authority Sites we use in our database and weightings we assign to them according to authority level and prestige. We need to then boost citations by these Authority sites to weigh higher in our model.

***Current Implementation in Pickscomb***

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 );

**4. TF-IDF 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 is 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>

The above are the outline of the main principles. In the following section I shall outline the implementation details in the ArtemisRanker.

1. The first process in Similarity Ranking is building a vector model of each document. We do this in the class ArtemisRanker.TF\_IDFDocModel.  
     
   Much of the statistics for the core model is built straightaway by the class constructor. The input required to build the model are:   
   - The unique Article ID (string)  
   - The text content of the Article (string)
2. The first thing it does is clean up the Article text so that links, special characters etc. are all removed and only the core words that define the article are remaining. It is important to clean off everything except the core words in the text that carries the meaning of the Article. There are a couple of filtering and cleaning functions in the class that does this. These can be improved to do even better cleaning as needed.
3. ***Term Frequency (TF) statistics:*** The class contains a main data structure that collects the Term Frequency statistics for each Article.  
     
   private Dictionary<string, double> \_wordTFVec;  
     
   Term Frequency is defined as:  
     
   \mathrm {tf} (t,d)=0.5+0.5\cdot {\frac {f_{t,d}}{\max\{f_{t',d}:t'\in d\}}}

where *ft,d* is the frequency of occurrence of word *t* in the current document *d*.  
This is divided by the count of the word that occurs the maximum number of times within that document.   
  
Thus for TF statistics the constructor first counts the number of occurrences of each word in the document and divides by the maximum count it finds in the process according to the above equation for each word in the wordTFVec vector above. This concludes the calculation for the TF statistics for all words in each one of the documents.  
  
Note Term Frequency (TF) is a statistic that is calculated entirely within the same document and hence the above vector is a private member of each document’s model.

1. ***Inverse Document Frequency (IDF) statistics:*** The IDF statistic is defined by the equation below:  
     
   \mathrm {idf} (t,D)=\log {\frac {N}{|\{d\in D:t\in d\}|}}

where *N* is the total number of documents in the corpus within the full number of hours we are considering which is given by:  
Ranker.RankingHoursConsidered + Ranker.RankingHoursBuffer

This is 36 + 6 = 42 hours for the current settings.  
  
|\{d\in D:t\in d\}| : means number of documents where the word *t* appears in the full 42 hours of the corpus. To prevent division by zero, it is common to adjust the denominator to 1+|\{d\in D:t\in d\}|.

Thus we collect these statistics in the following variables in the class:

// Total number of Articles

public static ulong N = 0;

// Number of Articles containing each term (word) for this vertical

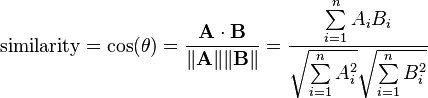
public static Dictionary<string, ulong> NWithWord =

new Dictionary<string,ulong>();

Note that these statistics are not contained within the word, but are collected for the whole corpus. Therefore these are static variables.  
  
Within the constructor of the current document we are processing, we simple increment N by 1 to count one more Article in the corpus.  
  
And for each word *w* occurring in this document we increment *NWithWord[w]* by 1.

1. ***Full TF-IDF Statistics:*** Note that we must complete the creation of a TF\_IDFDocModel for each of the documents of the full 42 hours of documents before we can calculate full TF-IDF statistics. This is because the IDF statistics are only partial and not complete until the full calculation is done for the whole 42 hours of Article models.  
     
   Hence in this TF\_IDFDocModel class we gave the function that will do the final TF-IDF calculation that must be run after the document models for all the articles in the 42 hours are built.  
     
   public void setTfIdf()  
     
   At this point all the statistics for setting IDF has been collected and this function calculates IDF and sets the full TF-IDF score for each word in the document. We store the full TF-IDF value vector for each word in the document in the vector:  
     
   private Dictionary<string, double> \_TF\_IDFVec;
2. ***The meaning of the TF-IDF statistics:*** What does the above TF-IDF statistic mean? Essentially it give very high scores to words that are unique and meaningful to the particular document and hence define what the core content of the document is about.   
     
   At the same time it assigns very low scores to words that are common and are not very meaningful such as *the, and, are, is*. It gives slightly higher scores to somewhat more meaningful but common words such as *come, go, each, sell*.  
     
   Therefore the above sefTfIDF() function orders the words in the descending order of the TF-IDF scores and takes the top M (M=40 here) words and their TF-IDF scores as the model of the document that represents its core meaning.  
     
   I have defined the number of top words to take as 40 here, but this can be adjusted to a higher value to increase the accuracy of the model at the cost of increased time for the calculations to run.  
     
   This final TF-IDF vector of top scoring words in the document and their TF-IDF scores represents a model of the important essence of the document—the important words and a measure of how important they are.
3. Sometimes we might get the same Article in the corpus by error. In such an event, because the TF-IDF counts are made in the constructor, we have the following functions to re-adjust the extra IDF statistics: public void adjustIDF()
4. The class TF\_IDFDocModel therefore builds a statistical vector model that captures the core meaning of each document in the 42 hours we are considering.   
     
   The class that uses these models to assess Similarity between Articles is: SimilarityRanker
5. The SimilarityRanker expects the following data for each Article in the 42 hours considered.  
   Article ID: unique string ID  
   Article creation or publish time  
   Article content text  
     
   The SimilarityRanker maintains 2 data structures to hold the information in estimates.  
     
   It uses the above class to build a TF-IDF vector model for each Article and holds these models mapped to the Article’s unique ID in:   
   private SortedList<string, TF\_IDFDocModel> \_ArticleTfIdfVec

Secondly, it maintains a list if the Articles in the model sorted by its creation or publish time in the index:  
private static SortedDictionary<DateTimeOffset, string> ArticleIndex

1. The Ranker uses the Init() function to build the full model from scratch the first time the Ranker is run from the full 42 hours of Articles fetched from the corpus.  
     
   In subsequent iterations the Ranker sleeps and wakes up and has to only update the model for the last 15 minutes of Articles. It uses the UpdateModel() function to do this update. This function first uses the timing index to clear out the 15 minutes of Articles that have expired from the Model using RemoveArticleFromModel() and PruneModel() functions, and then adds the new Articles from the last 15 minutes to update the model.
2. After the models of the Articles have been thus built from scratch or updated, we are ready to estimate the cosine distances between Articles.  
     
   The cosine distance is defined as on [this page](https://en.wikipedia.org/wiki/Cosine_similarity) as:  
     
   

Given 2 TF-IDF vector models of 2 Articles, the function   
private double SimilarityRanker.GetCosineDistance()   
calculates the cosine distance between them as defined above.  
  
This cosine distance will be a value between [0.0 , 1.0] where 1.0 denote perfect similarity and 0.0 shows no similarity (no important common words) at all.  
  
Note that because the 2 document models may contain different words, we first have to build a vector that is the union of words in both models and assign the respective model’s TF-IDF scores for words that appear and 0.0 for words that do not occur in that model. Then the cosine distance is a straightforward calculation.

1. Given a particular Article in the corpus, we now use this function to calculate the cosine distance from it to every other Article, order the results in the descending order of the cosine distance and from the top pick the top N most similar Articles to the one given.  
     
   This is done in the function:   
   public SortedDictionary<double, string> GetTopSimilarArticlesTo(string artid)

Note if we are considering 10,000 Articles in the corpus, this estimation is a very expensive operation. Hence, we use 3 techniques of optimizations without which this algorithm will take too long and fail.  
  
(i) Just in time calculation: We calculate the results only at the point it is required.  
  
(ii) Preventing re-calculations: Once we calculate the distance for a set of Articles, we store the result and simply retrieve it without re-calculation if it is required again.  
  
(iii) Note if there are 10,000 Articles in the corpus, we can use 10,000 x 10,000 matrix to store all of their similarity scores to each other. Consider this M matrix with row Article *r* and column article *c*. The distance between Articles r and c is the same as the distance between articles c row and r column. Hence, we need to calculate only the bottom half triangle from that divided along the diagonal, cutting the number of calculations by half.  
  
To make these optimizations we use the following data structures in the class:  
  
private double[,] \_cosineDistance; // The Cosine Distances between pairs of Articles

private bool[,] \_isCalculated; // For just-in-time calculations

Thus in the function GetTopSimilarArticlesTo() we calculate only when required, and once calculated store in the bottom half of the matrix and fetch without recalculations.

1. In fetching the most similar Articles we maintain 2 main parameters in the class.

private double MinSimilarityScore = 0.4;

public int MaxNumSimilar = 5; // fetch this many similar articles max

The cosine metric scores similarity on a scale of 0.0 to 1.0 and the first parameter holds the lowest score that we will consider for an Article to be similar.  
  
The second parameter defines the number of top similar Articles to fetch.

These filters are applied to the results fetched by the GetTopSimilarArticlesTo() function.

This completes the functionality description of the TF-IDF Similarity Ranker.

**5. Popularity and Similarity Ranking Model Combination**

For the desired results on the front page we provide two options.  
  
(1) Display of pure Popularity Ranked results.  
  
(2) Display of Popularity Ranked results with display of a cluster of most similar Articles to each popular Article. This gives us a Google like group of display or Article groups or clusters.

For this second option the results of the Popularity Ranker muse be combined with those of the Similarity Ranker. We detail how this combination is done in this section.

1. The main class for the ranker is the Ranker class. A Ranker object is initialized by the task manager giving the vertical for which it is to be run and what type of ranking is designed. Currently we offer 2 types of ranking: pure Popularity ranking and combined Popularity-Similarity ranked clusters.
2. The main entry point to the Ranker is the Ranker.Run() method.   
     
   - When the Ranker is started for the first time it first fetches the data required for building the models from the database. The data fetched is dependent on the Ranking type. The data is fetched for the past 42 hours of Articles.  
     
   - The Ranker then passes the data to the init() method of the Popularity Ranker to build and initialize the Popularity Ranker.  
     
   - If the ranking type is both Popularity-Similarity, the Ranker passes the data into the init() method of the Similarity Rnaker to build and initialize the Similarity model.  
     
   - It then performs the first iteration of ranking and writes the results to the separate database table.  
     
   - The Front End will then read the ranking results and display the popular-similar article clusters on the front page.

- In subsequent iterations the Ranker sleeps till the next runtime, wakes up, reloads both models with the last 15 minutes of data, re-ranks, writes the results to the database for the front end to read, and goes to sleep.  
  
- This iteration is repeated forever until the server is shut down.

1. In this part we shall look at how the combination of the results of the 2 types of ranking is done.

- First the Ranker uses the Popularity model to get the list of Articles ranked according to their popularity.  
  
- If we are doing pure Popularity Ranking only, then this result is written out to the database for the Front End and this iteration is complete.  
  
- If we are doing both Popularity and Similarity Ranking, the Ranker passes the topmost popular Article from the Popularity ranked list to the Similarity Ranker and we fetch the cluster of 5 most similar Articles to it. This is then our most popular cluster to display at the top of the front page.  
  
- We remove the top popular article thus listed from the Popularity list. If the similar articles fetched are on our list of popularity ordered articles, then we remove them also from the Popularity ranked list. Then what is remaining in the popular list are those most popular Articles that have not already been listed for the front page.  
  
- We then pass the topmost next popular Article from the remaining popularity list to the Similarity Ranker and fetch the next cluster of 5 most similar Articles to it. This is then out second most popular Cluster.  
  
- We repeat the process until all the Articles are thus clustered into Popularity ordered clusters and then write the results out to the database RANKINGS table for the front end.   
  
- The clusters are ranked with the most popular cluster at the top, the next popular cluster next, and so on. Each cluster has a title Article and 5 (this value is changeable) most similar Articles to it.

Thus we get a listing similar to a Google search listing on our front end.

**6. Markov Clustering Algorithms**

Research & modelling ongoing.