UNIVERSITÉ ANTONINE
Faculté d'ingénieurs en Informatique, Multimédia, Réseaux et Télécommunications

Scientific Approach of RSS Join Engine

Course: Méthodologie de recherche

Presented by: MATTA Elie et al.

Abstract

RSS feeds presents the main source of news for internet users, therefore it would be important and beneficial to join such XML-based system to improve several issues such as eliminating duplicated RSS feeds. In this paper, we will follow the 6 step scientific approach [1] to elaborate our "RSS Join Engine". We conducted our approach trying to explore this relatedness between RSS titles using a specific framework that loads user-specified RSS feeds and then joins them based on a precision value. Finally we provided a series of experiments to validate our approach.

1. Problem Statement

RSS (Really Simple Syndication) is an easy and simple way to communicate information to a subscribed user from multiple data sources (i.e. RSS feeds from CNN and BBC) that satisfies a certain condition given by the users (i.e. Category, time interval, etc.).

Wouldn't it be a waste of time and resources for the user to read the same information more than once?

2. Background research

To resolve our problem, we divided our research into 2 parts: *theoretical* and *practical*.

In the *theoretical* part, we started our search on XML similarities since RSS is an XML-based system.
First we discovered that there are several types of relations between different feeds:

1- *Disjointness*, when there is no relatedness whatsoever between the 2 feeds.
2- *Inclusion*, when a feed is totally included in another.
3- *Intersection,* when 2 feeds refer to similar and related concepts.
4- *Equality,* when 2 feeds are identical.
5- *Oppositeness,* when 2 feeds are opposite but refer to the same issue.

These relations are calculated upon certain semantic relatedness[1] using a knowledge base (i.e. WordNet).
We will detail in this paper, 2 main concepts for calculating this similarity.

In [3] Aminul Islam et al. presented three functions to calculate similarities between two sentences: *string* similarity, *word* similarity and *text* similarity.

[1] Semantic relatedness is a more general concept than similarity. Dissimilar entities may also be semantically related by lexical relations such as meronymy and antonymy[2].

The *string* similarity works only on the shape/syntax of the sentences, for example let us consider a pair of texts, T_1 and T_2:
T_1: Many consider Michael Jordan as the best player in basketball history.
T_2: Michael Jordun is one of the best basketball players.
The only difference between the two texts is the misspelled word "Jordun" in T_2.
The semantic based value between these 2 words: Jordan and Jordun, is very low if there are any, but on the other hand, if the string similarity measures are used, we obtain a good similarity score. This is the main importance of the string similarity.

The semantic *word* similarity can be based on one of many word-to-word similarity metrics like distance-oriented measures computed on semantic networks, knowledge-based (dictionary/thesaurus-based) measures, or metrics based on models of information theory (or corpus-based measures) learned from large text collections. We will focus on the corpus-based measures because of their large type coverage.
By using the *Sim()* function, they normalized the semantic word similarity, so that it provides a similarity score between 0 and 1 inclusively. For example the function returns 0.986 for words *cemetery* and *graveyard*.

The semantic *text* similarity shows its main important when 2 texts contains the common words which will lead us on how similar the order of the common-words is in the two texts (if these words appears in the same or almost the same order, or very different order).
Considering the same example T_1, T_2 and after removing the non-common words, we will obtain vector X={Michael, Jordan, the, best, player, basketball} for T_1, and vector Y={Michael, Jordan; the, best, basketball, player} for T_2. We replace X by assigning a unique index number for each token in X = {1, 2, 3, 4, 5, 6}. Based on this unique index numbers for each token in X, we also replace Y where X = Y. That is, Y = {1, 2, 3, 4, 6, 5}. The function in this case will return 0.83.

As an overall semantic similarity, our task is to derive a score between 0 and 1 inclusively that will indicate the similarity between two texts P and R at semantic level. The main idea is to find, for each word in the first sentence, the most similar matching in the second sentence. The method consists in the following six steps mentioned in [3]. For example let P = "A cemetery is a place where dead people's bodies or their ashes are buried.", R = "A graveyard is an area of land, sometimes near a church, where dead people are buried." Then S(P,R) = 0.514 where S is the function of this six step method.

On the other hand, Richard Chbeir et al. [4] created a framework for merging RSS items that consists of 4 main modules:

1. Pre-processing module: It accesses the internet and downloads corresponding RSS feeds, checks their well-formedness then returns them to the next module.
2. Relatedness module: Accepts the list of items given and computes text/element/items relatedness by accessing a knowledge base (KB).

3. Clustering module: Facilitates the merging process by using the existing clustering algorithm that groups highly intersecting news in the same cluster disregarding item relationships.
4. Merging module: Uses the output of the preceding module in order to abridge grouped element according to predefined merging rules and user preferences[2].

In this study, it's mentioned that existing clustering algorithms put together highly intersecting news in the same cluster disregarding item relationships. For instance, news items related with the *intersection* relationship and having lesser similarity score are put in different clusters according to their scores, although these items should be put together in the same cluster. As a result, a Relationship aware Single Link Level based or *RaSL*^{2} is presented.

Now that we presented the concepts of calculating the similarities, it's obvious that we need to put unbounded data streams coming from multiple data sources in a bounded memory.
This is done by using the sliding-window concept that offers us 2 types of windows: *count-based* sliding-window that contains the last T items (feeds) or a *time-based* sliding-window that contains items that have arrived in the last t time units [5].

Due to the continuous changes to the content of the windows, we found 2 possible strategies to keep the result updated with the changes. An *eager re-evaluation* strategy generates new results after each new tuple arrives but may be infeasible in situations where streams have high arrival rates. A more practical solution is the *lazy re-evaluation* is to re-execute the query periodically [5].
On each entry of a new *tuple k* a test should be done on each existing tuple having the following condition: " $\forall u \in S_1$ *and* $k.ts - T1 \leq u.ts \leq k.ts$ " / S_1 is a window, k.ts is the timestamp of the tuple k and T_1 is the time size of the S_1 time-based window. Every tuple *u* that doesn't match the condition will be exiled from the window.

And now the join process between the windows will begin using a certain join process algorithm like the NLJ (Naïve Multi-Way Join) algorithm.

In the *practical* part, our main objective is to find an application that responds to our project that is to aggregate and join RSS feeds using the conditions given by the user. As a result we found "RSS Merger" [4], a C# desktop prototype. This prototype accepts as input RSS news items, as well as Boolean input parameter allowing the user to choose whether to consider data semantics[3] or not. It measures relatedness between news items automatically after (i) stemming text values, (ii) generating vectors for each text, (iii) computing relatedness and relationships at different level of granularity[4].

[2] Every user is allowed to specify the notions of merging by associating relations between elements and merging operators.
[3] That could exploit VKB and LKB in identifying label/value neighborhoods.
[4] Granularities that could be like text, label, simple element, and item (complex element).

It clusters the RSS items based on the relatedness value and finally merge the news based on the users merging rule.

We also found two types of software, XML comparators, and RSS aggregators that could be helpful later in this project.

As for the XML comparators, we found codes that compare 2 XML documents; we also found multiple free applications that highlight the differences between 2 files. In some of them we were able to edit the XML document directly from the software, but in others this action is not possible. Here is a list of the software found:
BeyondCompare, ExamDiff, ExamXML, CompareIT, Compare and Merge…etc

As for the aggregators we found that there are many different RSS aggregators on today's market. They differ in many details like complexity, appearance, functionality etc. However all these programs have one in common—they handle RSS.
Below there are listed examples of RSS aggregators [6, 7]:

RSS aggregator	(a) Application type	(b) Database	(c) OPML/Blogroll	(d) Feed updates	(e) XML Formats	(f) Feeds	(g) Extendibility	(h) OS Integration	(i) AJAX	(j) APIs
1. Active Web Reader	2,4,5		1,2	1,3	1			1	3	
2. Amphadesk	3,4,5,6,7,8	2		1,2	1	2,3	1		1,2,3	
3. BlogLines	4,5,6,7,8	1,3,4	1,2,3	1	1,2	1,2,3		1		2
4. FeedReader	3,4	4	1,2,3	1,2	1,2	1,2		1	1,2,3	
5. FeedDemon	3		1,2		1,2			1		
6. NewsGator	3		1,2		1,2			1		4
7. SharpReader	3		1,2	1,2,3	1,2		2	1		
8. Snarfer	3	4	1,2	3	1,2	1,2	2,3		3	2
9. Gregarius	4,5,6,7,8	1,2,4	1,2,3	1	1,2	2	1,2,3		1,2	2
10. MonkeyChow	4,5,6,7,8	1,3	1,2,3	1,2,3	1,2		1			
11. NewsBeuter	3	4	1,2	1,3	1,2		1,2,3	1		2
12. Composite	3	2		1,3	1	1	1		2,3	
13. RSS bandit	3	1	1,2,3	1,2,3	1,2	1,3	1,2,3	1	1,2,3	1,3,4
14. RSS Owl	3		1,2,3	1,2,3	1,2	3	1	1	3	

Figure 1: List of RSS Aggregators

(a) 1: Outlook plug-in, 2: Browser plug-in, 3: Desktop application, 4: Hosted on a server and is controlled with a web browser; Works in the following browsers: 5: Opera, 6: Safari, 7: Firefox, 8: Internet Explorer
(b) 1: Hosted website, 2: Requires user setup, 3: Shared database, 4: Full text database search
(c) 1: OPML import, 2: OPML export, 3: Blogroll capable
(d) 1: Scheduled feed updates, 2: Individual feed update, 3: Refresh single feed
(e) 1: Reads RSS 2.0, 2: Reads ATOM
(f) 1: Feed recommendation, 2: User suggested feeds allowed, 3: Search for feeds option
(g) 1: Open source, 2: Has user plug-ins, 3: Has user themes
(h) 1: Has desktop notification system
(i) 1: Asynchronous tagging, 2: Asynchronous marking read, 3: Asynchronous retrieval items
(j) Supports 1: Dave winers aggregator, 2: Bloglines, 3: Google aggregator, 4: Newsgator, 5: Microsoft RSS platform

3. Adopted solution

As a result of the background research done earlier, we were unable to find a result to our problem, which is to join RSS feeds from several sources and return the result. We only found RSS aggregators and XML comparators. These software as their name mention, are able only to collect RSS feeds (aggregator) from given RSS sources and list them to the user, the XML comparators compares XML files and highlights the difference between them.
In the remainder of the paper, we will try to create an RSS join engine, based on the existing work while developing the function of joining the RSS feeds based on the semantic relatedness.
The main idea of the project can be presented in the following figure:

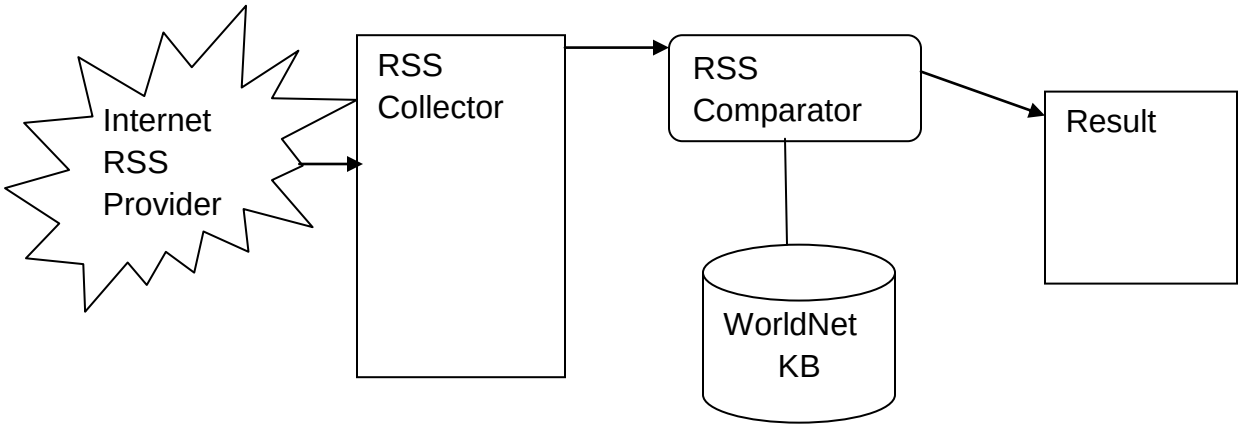

Figure 2: RSS Join Engine framework

1: The RSS Collector will connect to the internet to find the RSS providers and download the RSS title.
2: The RSS Collector will accept the given titles and will try to group them by category (Sports, Business, Economy…).
3, 4: The RSS Comparator will take the grouped titles and compare them to get the semantic relatedness using the WorldNet database, once it's done, it will give the output and the result.

Our program will be based on these functions:
function GetRSS(): this function will return the titles of the RSS from the given RSS source(URL), and list them in a listbox. It will also be automatically re-executed in a given lapse of time to keep the user updated with all the new RSS feeds.
function JoinRSS(): this is the main function of the engine, it will connect to the WorldNet KB to be able to compare each title and then tries to join existing RSS feeds in the listboxes, finally it will throw the result of the join process in another listbox.

```
function GetRSS()
        Check(URL)
        if(Check)
                Connect(URL)
                Collect(URL)
        else
                "Display Error Message"
        end If
end Function
```

function Group(RSS): this function will group the incoming titles by type (for example Sports)

function Check(URL): this function will check the availability of the given url, and if it really contains RSS feeds. It will return true if not false.

function Connect(URL) : this function will use the internet connection to connect to the given URL

function Collect(URL) : once connected, this function will collect all the titles from the source and show them to the user, while implementing in the same time the Group(RSS) function

function Comp(title1,title2)
Open an instance on WorldNet Knowledge Base
Compare each title with the other one using semantic relatedness measures (xSim,…)
return the Comparison as type (inclusion, intersection, oppositeness, disjointness or equality)

function JoinRSS()
Comp(title1,title2)
If Comp = Disjointness then
Show both titles
end If
If Comp = Equality then
Show one of the titles
end If
If Comp = Intersection then
If one of the titles is totally included in another (Inclusion) then
Show the title including the other title
else If one of the titles is intersection with another but the content is referring to opposite meaning (Oppositeness) then
Show both titles
else
Show the intersection of one of the titles
end If
end function

4. Test hypothesis

After developing the pseudo code into an executable one using C# .NET, and due to the lack of RSS Join engines, we found ourselves obliged to put our solution under multiple tests to calculate its precision and its time of response to analyze the overall performance. *These tests were held on an Intel Core 2 Duo T6400 Processor machine (with 2.0 GHz processing speed and 4GB of RAM)*. It's important to mention that the

experiments are also held locally: we saved on our hard drive two XML files that contain RSS feeds and load them using our prototype.

A) Time of response

To analyze the time of response, we executed several tests.
In the first experience, we fixed the value of the threshold to 0.2 and we varied the size of the xml files:

a) File1 size = 2.69KB (13 entries), File2 size = 3.97KB (15 entries)
b) File1size = 7.15KB (28 entries), File2 size = 10.3 KB (35 entries)
c) File1 size = 2.69KB (13 entries), File2 size = 7.15 KB (28 entries)
d) File1size = 7.15 KB (28 entries), File2 size = 3.97 KB (15 entries)

In the second step, we fixed the value of the threshold to 0.6 and we kept the files sizes as mentioned before.
In our final step, we fixed the threshold value to 0.8 and also we kept the file sizes as mentioned. The results are shown in the following figure:

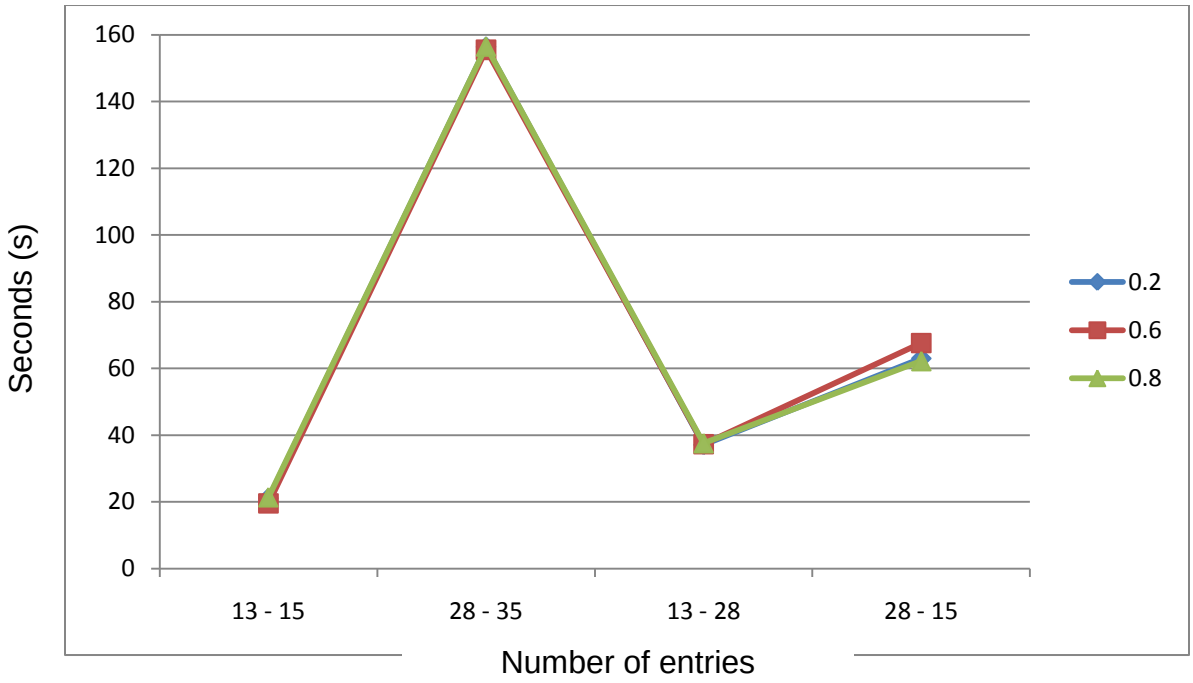

Figure 3: Time of response

B) Precision

To analyze the precision and efficiency of the join function in our project, we also did several tests.

In the second experience, we fixed the value of the threshold to 0.1 (*Disjointness*) and we varied the size of the xml files (number of entries):

a) File1 size = 2.69KB (13 entries), File2 size = 3.97KB (15 entries)
b) File1size = 7.15KB (28 entries), File2 size = 10.3 KB (35 entries)
c) File1 size = 2.69KB (13 entries), File2 size = 7.15 KB (28 entries)
d) File1size = 7.15 KB (28 entries), File2 size = 3.97 KB (15 entries)

In the second step, we fixed the value of the threshold to 0.5 (*Intersection*) and we kept the files sizes as mentioned before.
In our final step, we fixed the threshold value to 0.9 (*Equality*) and also we kept the file sizes as mentioned. The results are shown in the following figure:

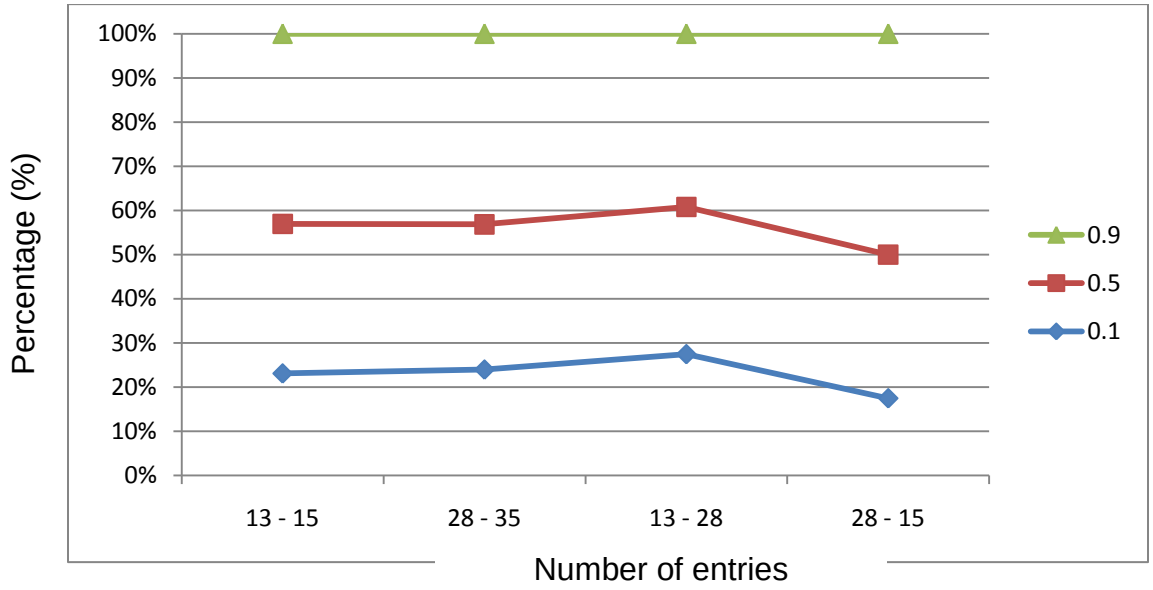

Figure 4: Precision chart

5. Conclusion

In this paper, we conducted a research of measuring semantic relatedness between RSS items. We have detected several ways for computing this relatedness such as word similarity, text similarity and string similarity to obtain one of the four existing semantic relations : disjointness, intersection, inclusion and equality which represents the basic result of our joining process prototype.
As a result of our conducted experiments based on time of response and precision, we concluded from figure 3 that no matter what was the threshold value, the time of response is identical which is abnormal because it should vary with the threshold value proportionally. But on the other hand the precision of our project was a big success

because on a threshold value equals 0.9 as shown in figure 4 we found that the precision was approximately 100% which is the output that we were expecting.
As a closure for this study, we still need to find a solution to reduce the time of response in order to make it acceptable w.r.t the human scale, and extend the join process to cover the description element of the RSS feeds; these ideas and issues will be discussed in our next paper.

References

1. Overview of the Scientific Method. http://www.sciencebuddies.org/mentoring/project_scientific_method.shtml
2. Getahun, F., Tekli, J., Chbeir, R., Viviani, M., Yétongnon, K., Relating RSS News/Items. ICWE 442-452 (2009)
3. Islam, A., Inkpen, D., Semantic text similarity using corpus-based word similarity and string similarity, ACM Transactions on Knowledge Discovery from Data (TKDD), v.2 n.2, p.1-25, (2008)
4. Getahun, F., Tekli, J., Chbeir, R., Viviani, M., Yetongnon, K., Semantic-based Merging of RSS Items, WWW: Internet and Web Information Systems Journal Special Issue: Human-Centered Web Science, Springer Netherlands, Vol. 12 (No. 11280) (2009)
5. Golab, L., Özsu, M., Processing sliding window multi-joins in continuous queries over data streams, Proceedings of the 29th international conference on Very large data bases, p.500-511, Berlin, Germany (2003)
6. Derezińska, A., Małek, T., Unified Automatic Testing of a GUI Applications' Family on an Example of RSS Aggregators, Proceedings of the International Multiconference on Computer Science and Information Technology, pp. 549 – 559, ISSN 1896-7094 (2006)
7. A directory of RSS Aggregators. http://www.aggcompare.com