Near Duplicate Document Detection for Large Information Flows

Daniele Montanari, Piera Puglisi

Near duplicate document detection for large information flows

Daniele Montanari and Piera Laura Puglisi

[1] ICT eni - Semantic Technologies
Via Arcoveggio 74/2, Bologna 40129, Italy
[2] GESP
Via Marconi 71, Bologna 40122, Italy
daniele.montanari@eni.com,pieralaura.puglisi@external.eni.com

Abstract. Near duplicate documents and their detection are studied to identify info items that convey the same (or very similar) content, possibly surrounded by diverse sets of side information like metadata, advertisements, timestamps, web presentations and navigation supports, and so on. Identification of near duplicate information allows the implementation of selection policies aiming to optimize an information corpus and therefore improve its quality.
In this paper, we introduce a new method to find near duplicate documents based on q-grams extracted from the text. The algorithm exploits three major features: a similarity measure comparing document q-gram occurrences to evaluate the syntactic similarity of the compared texts; an indexing method maintaining an inverted index of q-gram; and an efficient allocation of the bitmaps using a window size of 24 hours supporting the documents comparison process.
The proposed algorithm has been tested in a multifeed news content management system to filter out duplicated news items coming from different information channels. The experimental evaluation shows the efficiency and the accuracy of our solution compared with other existing techniques. The results on a real dataset report a F-measure of 9.53 with a similarity threshold of 0.8.

Keywords: duplicate, information flows, q-grams

1 Introduction

The information explosion due to the development of the web and its technological, organizational, and societal exploits of the last decade, resulted in growing data volumes being transmitted, processed, integrated, interpreted, and used in decision making processes, with a corresponding need for automated and accurate execution. This spurred a large body of research tackling data management from several points of view, including

– **the source**; this viewpoint is studied in fields like viral marketing[13], when issues of coverage and impression are relevant and we are interested in the

implicit and explicit diffusion mechanisms which are typical of a networked environment with distributed decision making and local, independent diffusion policies;
- **the receiver**; this is the dual point of view of the previous one, when we want to know whether we receive all and only the items we would like to receive, striving for precision and recall;
- **the structure of the network of infomediaries**; this area concentrates on the effects of network properties on the propagation of information;
- **the content and format of the payload**, namely the impact of channel format, protocols, and content types on the diffusion mechanism;
- **the distribution of affected or interested populations**, namely the positioning of the target population within the network.

This paper takes the receiver point of view and proposes an approach to remove the extra load generated from the spreading and replication mechanisms of the information through the network of intermediaries, which result in several incoming identical or nearly identical copies of the same content. Broadcast news (by news agencies, paper publications, online sites, and other sources of information and analysis) are an example scenario where the replication is quite significant; whenever an article is published by a primary source, many other infomediaries repackage the information for a specific type of readership or for distribution within a geographic area. A receiving system may then acquire the same item from a variety of different channels. The only way to reduce the incoming volume of information (while maintaining a reasonably wide coverage of sources) is to detect duplicate or nearly duplicate items upon arrival to our system, so that the copies may be stopped before entering the costly processing pipeline.

In this paper, we propose a new algorithm to find near duplicate documents based on q-grams extracted from the texts. Duplicate document detection typically consists of finding all document-pairs whose similarity is equal to or greater than a given threshold. Since similarity computation for all document-pairs requires significant resources, these methods usually apply filtering techniques to speed-up the comparison process. Each new item entering the system is normalized and cleaned of all the extra surrounding the core information constituting the payload of the message. Then, a signature of the incoming item is computed. In our case, this signature is the list of occurrences of the q-grams found in the normalized text. This signature is compared with a set of signatures coming from the analysis of other news items received earlier by the system. If a match is found, then the arriving news item is likely to be a duplicate and it is discarded. Otherwise, it is forwarded to the system for acquisition. Our approach introduces two major novelties: first, we maintain an inverted index for each q-gram in order to speed-up the comparison process among different documents and reduce the number of comparisons; second, we provide an efficient memory allocation of the inverted indexes, using a window size of 24 hours. In this paper section 2 introduces the system scenario; after reporting a brief description of the related works, section 3 describes the main algorithm; section 4 discusses the experimental evidence of this method compared to other existing techniques;

section 5 shows the architecture of the solution and section 6 reports conclusions and future work prospects.

2 A real world problem and the corresponding system scenario

The study reported in this paper stems from a general interest and an actual need to improve a real world system. In this section we provide a description of the relevant features of the latter.

The internet and a new generation of mobile devices have come to offer an unprecedented opportunity for the creation, packaging and distribution of information in real or quasi-real time. This, in turn, has prompted the rise of *infomediaries*, that is intermediaries offering packaged accesses to a huge number of worldwide info sources, the outcome of all this being the potential to collect tens or hundreds of thousands of information items per day, amounting to hundred thousands or even millions of information items per month. Even large organizations have trouble sifting through such vast amounts of information for the items which are relevant at the specific point and time of use.

The standard architecture we are dealing with consists of several concurrent pipelines taking information from the acquisition stage to the final classification and indexing. Figure 1 illustrates the components in this architecture and is briefly described below.

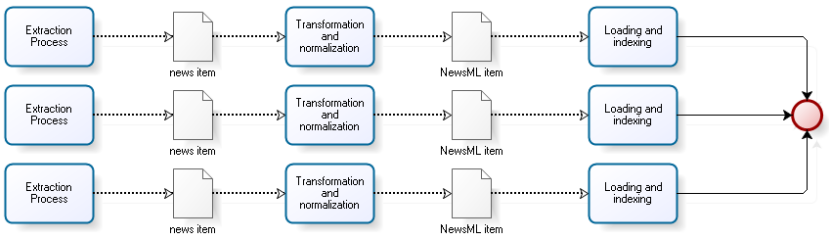

Fig. 1. Flow of news coming from different channels.

The Extraction Process includes the mechanisms and the configurations needed to access the remote source of information. In most cases this is a web crawler or a software agent with the necessary functionality for protocol agreement and remote data transfer to a local temporary repository. The outcome of this module is a set of XML or otherwise encoded files carrying structured metadata and raw content.

The Transformation and Normalization Process takes the local material acquired through Extraction and rewrites each item in a final common format

suitable for loading into the system. Metadata coming from the acquisition process are usually enhanced or normalized at this stage; normalization includes the removal of all tagging used for the original presentation and later transfer of content.

At last, the Loading and Indexing Process takes the normalized outcome of the previous step into the system, and takes care of the indexing,classifying, and extracting chores.

Due to the open and competitive structure of the info market, the same event or fact may be covered by multiple news agencies, and each article issued by any one of the covering sources may be distributed to several news channels, resulting in multiple acquisitions of the same information. Several estimates concur to suggest that about 10-15% of the acquired information is redundant due to these replication phenomenon.

Both the efficiency of the process and the quality of the resulting corpora of information demand that this redundancy be removed or minimized.

Redundancy reduction at the infomediary level is difficult to achieve, since there are no standards to refer to, resulting in uneven treatment across providers. Also, the providers are unaware of the same information being carried by others (and likely eager to propose their content anyway), hence cross-provider duplication of information cannot be removed at the source. Therefore, the best opportunity for (near) duplicate removal occurs after the acquisition and before injecting the information into the system.

In order to complete the outline of the application scenario we need to discuss what is intended by *near duplicate* information from a user point of view. Ideally, a single fact should appear exactly once in a information corpus. This is however rather exceptional, and occurring in very standardized, controlled domains only. The most common condition is to have several slightly different statements which actually concur a richer description by their presence, time of production, number, and issuing sources.

The policy defining when a pair of documents should be considered duplicate is a very difficult, possibly time- and goal-dependent one. A first level of definition of such a policy is a *similarity threshold* associated with the syntactic structure of the documents, their wording and sentences.

3 The core algorithm

3.1 Related Work

Many algorithms have been proposed to solve the duplicate document problem. There are different kinds of approaches in the state of the art: web based tools, methods based on fingerprints or shingling, algorithms that use different similarity measures. Our work is an extension of the method[10] on duplicate records detection in the context of databases that will be described in section 3.3. Some of the existing techniques that address the duplicate detection problem are listed below.

Checksum and fingerprints. Cryptography checksums like MD5 [1] and SHA1 [14] are sequences of bits used to verify that data transferred have not been altered. The checksum works well to detect any change to a file as a whole, but lacks for small modifications [2].

A fingerprint is a short sequence of bytes used to identify a document. Fingerprints are created by applying a cryptographic hash function to a sequence of tokens. In [4], the author introduced an algorithm to produce approximate fingerprints to detect similar files within a large file system. The algorithm calculates a Rabin fingerprint [5] value with a sliding window of 50 characters.

Andrei Broder's super shingle algorithm[7] converts the document into a set of fingerprints. The algorithm can cluster the near-duplicates within O(n log(n)) time[6] by creating a feature (super fingerprint) using several fingerprints.

I-Match. Another approach is represented by the I-Match algorithm [6] that uses the statistics of the entire document collection to determine which terms to include in the fingerprinting. The words with smaller IDF (Inverse Document Frequency) are filtered out since they often do not add to the semantic content to the document. After filtering, all the terms are sorted (removing duplicates) and only one fingerprint is generated for the document. Two documents would be treated as near duplicates if their fingerprint matches.

Shingles. In [7], the method tokenizes documents into a list of words or *tokens*. A*shingle* is a contiguous subsequence of w tokens contained in a document D. Identical documents contain the same set of shingles whereas near duplicate documents contain overlapping sets of shingles.

SpotSigs. In[11], the proposed algorithm extracts signatures that contribute to filter out noisy portions of Web pages. The method identifies content text of web sites combining stopwords antecedents with sequences of close words. The efficiency of the method is guaranteed by the use of an inverted index that reduces the similarity search space.

LSH. Locality-Sensitive Hashing (LSH)[12] is an algorithm used for solving the near neighbor search in high dimensional spaces. The basic idea is to hash the input items so that similar elements are mapped to the same buckets with high probability. It has also been applied to the domain of near duplicate detection.

3.2 Basic definitions

This paper represents an extension of [10], an efficient algorithm for computing duplicated records in large databases. The method computes similarity between records using two different q-grams based measures and maintains an inverted index to speed-up comparison process. We start with some basic useful definitions.

Edit distance and q-grams Let Σ be a finite alphabet of size $|\Sigma|$. Let $s_1 \in \Sigma^*$ be a string of length n made up of elements in Σ. The **q-grams** are short character substrings of length q. Given a string s_1, its **positional q-grams** are obtained by sliding a window of length q over the characters of s_1. See [10] for

a general introduction to q-grams. Note that in our work we do not use partial (length less than q) substrings.

For example, the positional q-grams of length q=2 for string *tom_smith* are: { (1,to), (2,om), (3,m_), (4,_s), (5,sm), (6,mi), (7,it), (8,th) }. The set of all positional q-grams of a string s_1 is the set of all the $|s_1|$ - q + 1 pairs constructed from all q-grams of s_1. In the example above, the number of all positional q-grams is given by: $|tom\_smith|$ - 2 + 1 = 8.

Inverted index Many approaches use filtering techniques in order to reduce storage space and computational time. Some of them use inverted indexes, i.e. data structures storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. In particular, a *word* level inverted index contains the positions of each word within a document.

Given n documents and w distinct terms, in the worst case (every word occurs in all documents) the index requires $O(nws)$ bytes, where s is the space (number of bytes) needed to save an integer.

In order to reduce space requirements, a binary vector or bitmap is used to handle an inverted index. The bitmap represents the occurrences of a term in the corpus of documents. More precisely, for each document d, the *i-th* bit of the correspondent bitmap is set if d contains the *i-th* terms. In the worst case, each term contains all set bits and the space required is $O(nw)$ bits which is an improvement with respect to the above space requirement.

3.3 DDD for duplicate document detection

The work presented here is a variation of a previous work, named DDEBIT [10], for discovering duplicate record pairs in a database. The method is based on the use of two similarity functions together with an efficient indexing technique. Each pair-wise record comparison is performed in two steps: the records are converted into strings and compared using a *lightweight* similarity measure bigram-based. If the similarity is above a *light* threshold, then strings will be compared using a more accurate similarity. We only report the definition of the *lightweight* measure since it will be extended in the present work and used as a similarity function for identifying duplicates in large sets of documents.

Definition 3.31 (**Lightweight similarity**) ***[10]****. Let R be a relation and X, Y be strings representing two tuples in R. Let q_X and q_Y be the sets of distinct q-grams of X and Y, respectively. The function $d_{light,q}$ is defined as follows:*

$$d_{light,q}(X,Y) = \frac{|q_X \bigcap q_Y|}{\max(|q_X|, |q_Y|)} \tag{1}$$

The authors introduce the concept of *bigrams array*, that is a sorted list of all distinct bigrams in the database. For each bigram, the algorithm maintains a bitmap as inverted index: the information stored in the data structure is used to find common bigrams between records.

In our present approach, we address documents in place of records and the new definition of *lightweight similarity for documents* is based on q-grams.

Given a text segment T, we remove white spaces and all the punctuation marks obtaining its *normal form* T_w. For example, if T={A rose is a flower}, then we convert T into the string T_w={Aroseisaflower}. Then, we generate its signature $T_{w,q}$, that is the set of all possible $|T_w| - q + 1$ overlapping q-grams generated from T_w. Suppose q is equal to 3, we obtain: $T_{w,3}$={Aro, ros, ose, sei, eis, isa, saf, afl, flo, low, owe, wer}.

We now state an extended definition of lightweight measure for discovering similarity between documents.

Definition 3.32 (**Document similarity measure**) . *Let C be a corpus of text documents and A and B be two documents in C. Let $A_{w,q}$ and $B_{w,q}$ be the set of all possible overlapping q-grams from A and B, respectively. The function doc_q is defined as follows:*

$$doc_q(A, B) = \frac{|A_{w,q} \bigcap B_{w,q}|}{\max(|A_{w,q}|, |B_{w,q}|)} \tag{2}$$

The measure (2) computes the similarity between two documents in terms of common q-grams between texts. This function gives emphasis to collocation of words into sentences, since q-grams are built binding sequential words. For instance, the string "the president" will contain the q-grams 'hep' and 'epr' since the words *the* and *president* are close to each other. Moreover, the denominator contributes to filter out documents similar in minimal area or having different sizes.

The formal definition for duplicated documents is stated next.

Definition 3.33 (**Duplicated documents**). *Let C be a corpus of text documents and A and B be two documents in C. Let t_d be a threshold (a value between 0 and 1) defined by the user. Then, A and B are considered duplicates if $doc_q(A, B) \geq t_d$.*

3.4 An efficient algorithm to detect duplicates

This section introduces the proposed solution for the duplicate document detection problem. The pseudo code of the basic version of the algorithm is presented in figure 2.

In line 1, the algorithm allocates the data structure for the bitmap indexes. For each q-gram, it generates a bitmap index (the total number of distinct q-grams $|\Sigma|^q$ depends on the alphabet of characters).

The dimension of the index depends also on the number of total documents in the archive. In real world systems, the number N_t of total documents in the corpus at time t is not constant, due to the asynchronous arrival of new documents and removal of old ones.

```
Duplicate_document(flowDir: f_dir, sizeOfQgrams: q
dupThreshold dup_t, alphabet Σ)
output: document pairs whose similarity is greater than dup_t

1  allocate a list of bitmaps of dimension |Σ|^q
2  for each incoming document d_{n+1} in f_dir
3      allocate a list of integers list_counters of dimension n + 1
4      for each distinct q-gram q_j extracted from d_{n+1}
5          if q_j is new (never met before in the corpus)
6              allocate a bitmap B_{q_j} for q_j of dimension n + 1
7          update B_{q_j} setting the (n + 1) − th bit
8          update list_counters //register common q-grams among d_{n+1} and d_t, 1 ≤ t < n + 1
9      for each element c_{t<n+1} > 0 in list_counters
10         compute doc_q(d_{n+1}, d_t);
11         if doc_q(d_{n+1}, d_t) == 1
12             d_{n+1} is an exact duplicate of d_t
13         if doc_q(d_{n+1}, d_t) >= dup_t
14             d_{n+1} is a near duplicate of d_t;
```

Fig. 2. The duplicate_detection procedure.

The algorithm treats the incoming flow of documents by constantly polling the input directory (line 2). After receiving the document *n+1*, where n is the current size of the archive, the algorithm generates its signature $T_{w,q}$. Moreover, for each q-gram $q \in T_{w,q}$, the method updates the correspondent bitmap setting the *(n+1)-th* bit if q occurs in document *n+1* (lines 4-7). For each new document entering the archive, the measure (2) is computed with respect to the n document of the archive. The computation is straightforward: a temporary list (*list_counters* in line 8) stores common q-grams among current document and the others in the repository. If (2) is above the threshold for at least one document in the archive, the document *n+1* will be considered a near duplicate (lines 9-14).

3.5 Computational time

Let $Q := |\Sigma|^q$ and denote by N the dimension of the corpus. The algorithm takes $O(Q * N)$ to compare a document with all the elements of the corpus. Since $Q << N$, pairwise comparisons of all the elements of the corpus takes $O(N^2)$, i.e. the algorithm has a quadratic complexity in the dimension of the corpus. The computation of the similarity measure is straightforward since the information are all contained in the temporary structures.

3.6 Temporal window and memory requirement

In this paper, we consider an in-memory version of the algorithm in order to achieve better performance in terms of execution time. A swapping version of the method may also be developed, which stores the index on secondary memory if the main memory is not enough. In this section, we focus on the treatment of the incoming flow of news on an *hourly basis*. This solution speeds up the comparison process and leads to an optimization of the bitmap allocations.

A *temporal window* is a union of time intervals, each spanning *one hour* (see figure 3). Let w be the size of the window, i.e. the number of hours comprising

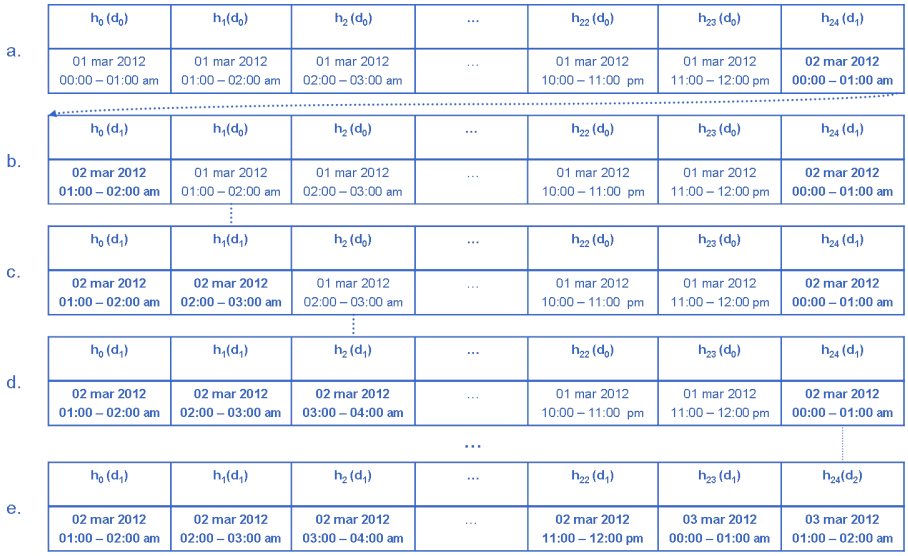

Fig. 3. Temporal sequence of documents processing using a window size of 24 hours.

the window. We set w to 24 since a preliminary analysis of the specific news channels proved that a significant percentage of duplicates usually occurs within *one day* of each other. A higher value of w would cause the creation of more indexes and a huge memory requirement without significant gains in terms of duplicates elimination. Of course, two duplicate documents arriving more than 24 hours apart will not be detected, letting in at most one duplicate every 24 hours.

For each hour h_j, the method creates a local index i_{h_j}, $j = 0..w$. Figure 3a shows the sequence of *local indexes* created by the algorithm for the first $24 + 1$ hours. In the first day of computation (d_0), the documents arrived during hour h_t will be compared with all the documents received during h_j, where $0 <= j <= t$. Moreover, the first hour of d_1: '00:00 - 01:00 of 02 mar 2012' is compared with

all the previous 24 hours starting from time interval: '00:00 - 01:00 am of 01 mar 2012'.

Figures 3b, 3c and 3d show the computation of the hourly indexes after the first 24 hours. In figure 3b, the index i_{h_0} related to d_1 (second day of computation) will overwrite the index i_{h_0} related to d_0, since '00:00 - 01:00 am of 01 mar 2012' and '01:00 - 02:00 am of 02 mar 2012' have a time difference of more than 24 hours and will be not compared anymore.

Finally, figure 3e shows the end of the second day of computation. More precisely, for each hour h_j, the index i_{h_j} will be compared with all the indexes i_{h_y}, where $0 <= y < t$ and $j < y <= w$. The computation continues using always the same window size and overwriting hourly indexes that are older than 24 hours. This implementation uses only the memory required by the actual q-grams occurrences in a specific hour interval.

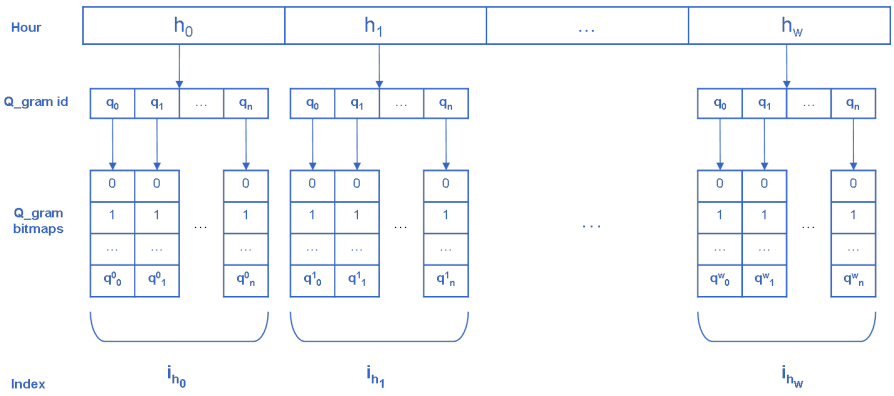

Fig. 4. Inverted index using hourly temporal indexes.

Going into more details about the construction of the index, for each hour h_i, the algorithm allocates a static array of q_n entries (see figure 4). For each q-gram q_i, a correspondent bitmap is allocated if and only if q_i is present in a document arrived during hour h_i. Let $N^t_{h_i}$ be the number of documents arrived at instant t since the beginning of hour h_i. Here, the optimization consists in allocating a bitmap of size $N^t_{h_i}$. If a q-gram appears rarely, the re-allocation of the dynamic bitmap will not be frequent; moreover, if a q-gram q_i is not found during hour h_i, the allocation of the bitmap will be not performed at all for h_i.

Let N_{h_i} be the average size of an hourly index. The memory required in the worst case is of $q_n * N_{h_i} * W$ bits.

4 Experimental analysis of the core algorithm

We tested the effectiveness of the algorithm using a corpus of 744 documents. In order to perform this experiment, we extracted this many items from a flow of real news coming from a single channel in a time interval of 24 hours. We inspected each document and found 111 correct matches.

The DDD algorithm is implemented in C++. The current implementation is about 800 lines of code and results in an executable file of about 900 KB. The experiments were performed in a SUSE Linux Enterprise Server, with a Quad-Core AMD Opteron(tm) processor 8356 and 4 GB of RAM. We performed experiments varying the size of q-grams and we found $q = 4$ as the best value. The algorithm introduces many false positives with lower values of q; the performance of DDD decreases with higher values of q since more q-grams require more memory.

We compared the effectiveness of our solution with other existing methods, i.e. SpotSig [11], Lsh [12] and Imatch [15]. We executed these algorithms using the Java implementation provided by the authors of SpotSig [15], setting the default parameters. Figure 5 reports the values of the Recall, Precision and F1-measures for several assigned threshold values. T represents all the matches, whereas $T \bigcap C$ represents the *correct* matches returned by each method. DDD shows the best value of Recall and F1 in all cases. In particular, it yields the best value of F1 with a threshold of 0.7 since the number of false positives and negatives is greatly reduced (Figure 6). The Imatch algorithm shows an optimal precision, but the recall is very low since it filters out a lot of false negatives. Spotsig registers a good behaviour in terms of both recall and precision (threshold 0.6), but Lsh performs generally better for this dataset.

The effectiveness of the algorithm has been tested by a large set of experiments. Due to space limitations we report the most significant results. Figure 7 reports the average execution time per document for different sizes of the corpus. We performed separated runs for each corpus, starting each run with an empty index. The more the hourly indexes grow, the more comparisons the algorithm performs, and the more the computation time increases, achieving the value of 1.2 seconds with size 8000. In the second experiment (see figure 8), we set the size of the corpus (500 documents) and performed 4 sequential runs of the algorithm starting from hour h_0 and ending the process at hour h_3. As in the previous test, the execution time increases as the the hourly indexes grow. In Figure 9 we show some results varying the size of the document text. We set the dimension of the corpus to 732 and saved the average time per document processing. The computation time depends on the length of the texts. In our system, the average length of the processed texts is about 5K so the execution time per document is less than 0,2 seconds.

Threshold	Method	T	T∩C	Recall	Precision	F1-measure
0,5	**DDD**	133	110	**0,991**	0,827	**0,902**
	Sigspot	99	84	0,757	0,848	0,800
	Lsh	100	85	0,766	0,850	0,806
	Imatch	64	64	0,577	**1,000**	0,731
0,6	**DDD**	114	106	**0,955**	0,930	**0,942**
	Sigspot	88	84	0,757	0,955	0,844
	Lsh	89	85	0,766	0,955	0,850
	Imatch	64	64	0,577	**1,000**	0,731
0,7	**DDD**	101	101	**0,910**	**1,000**	**0,953**
	Sigspot	82	80	0,721	0,976	0,829
	Lsh	83	81	0,730	0,976	0,835
	Imatch	64	64	0,577	**1,000**	0,731
0,8	**DDD**	101	101	**0,910**	**1,000**	**0,953**
	Sigspot	76	74	0,667	0,974	0,791
	Lsh	77	75	0,676	0,974	0,798
	Imatch	62	61	0,550	0,984	0,705
0,9	**DDD**	99	99	**0,892**	**1,000**	**0,943**
	Sigspot	76	74	0,667	0,974	0,791
	Lsh	77	75	0,676	0,974	0,798
	Imatch	62	62	0,559	**1,000**	0,717

Fig. 5. Comparisons of DDD with SpotSig, Lsh and Imatch.

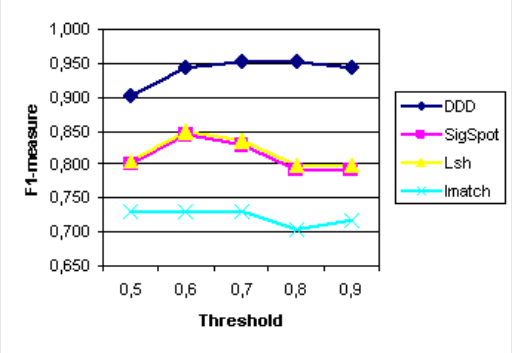

Fig. 6. F1-measure wrt threshold for all methods.

5 The overall architecture of the solution

The algorithm has been integrated in the framework of an application for the acquisition, classification, and analysis of news from heterogeneous Internet sources and providers.

In this application domain, an effective and efficient extraction of content from heterogeneous transfer and presentation artifacts is required to minimize

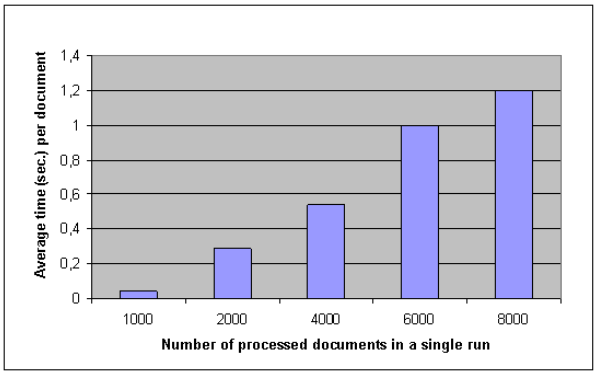

Fig. 7. Processing time versus size of the corpus.

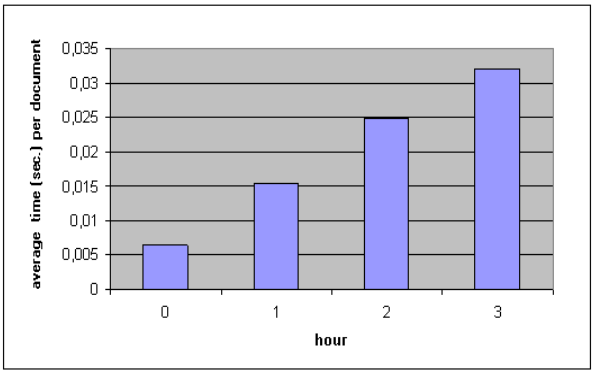

Fig. 8. Execution time with respect to different hours of processing.

the impact of overhead information on the similarity evaluation. More specifically, a single document D consists of a *core content* component C (the actual content value) and of an *accompanying content* component A, which includes all the collateral information of the artifact, like navigation and presentation code, as well as extras like advertising that may appear in a web page presenting a news article. We would like to test the similarity on the C component, ignoring the A component. If we can't remove the A component, then it may well happen that the latter outweights the C component (e.g. two news from the same source) and results in a false positive in case our similarity threshold is too low. On the other end, if our similarity threshold is too high, then D_1 and D_2 with $C_1 = C_2$ and $A_1 \neq A_2$ may result in a false negative. However, getting rid of the A component is a difficult exercise, if an automatic system must tackle many different sources of documents, since the encoding of web pages may be complex.

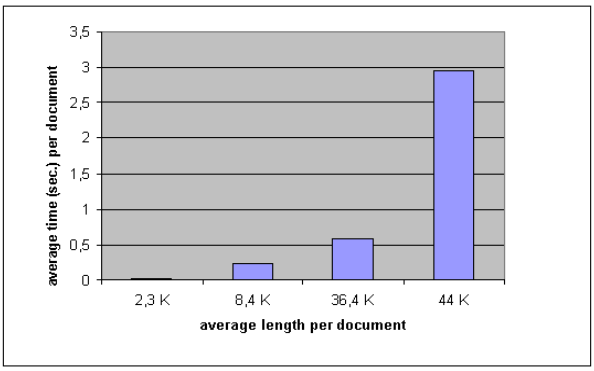

Fig. 9. Processing time varying the size of the document.

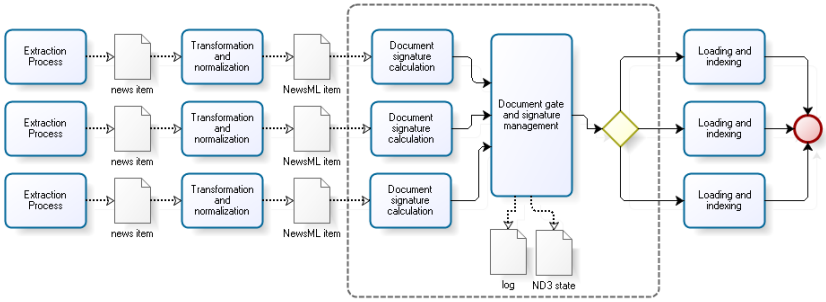

Fig. 10. Flow of the news acquisition system with duplicate document detection.

We are currently testing our system with a straightforward tag-removal filter, which however will need to be further tested and refined.

The processing of information through the overall system requires an average of approximately $t_p = 10$ seconds/document. We have also pointed out that there are about $P = 10\%$ duplicates in a typical batch of news. Therefore we have an advantage from the adoption of this duplication removal if the time saved Pt_p is more than the time t_3 spent in the extra analysis required. The case study presented here shows that the time to decide whether a single document is a duplicate of something already in the system as seen within our current horizon is less than 5 msec. Hence, we can conclude that the adoption of DDD appears to be definitely convenient.

6 Conclusions and Future Work

This study has offered an early support and validation for the proposed algorithm for near duplicate document detection. It has also helped identify and explore

a number of issues which become significant when the system is applied in a real world context with specific complexities. A number of issues have surfaced which will be focus for further studies:

- **clustering** of documents in the system to increase the performance of the comparison function;
- **semantic similarity measures**, to apply semantic analysis to the similarity assessment of a pair of documents;
- high and low **frequencies clipping**, to remove the very frequent and very rare q–grams from the comparison functions;
- **text extraction** to focus the similarity assessment to the core content of each document;
- **memory management** for online uninterrupted analysis of inbound document flows; we would like to study and implement fast memory management routines to adapt the time–sliding window of observation of the documents.

References

1. Berson, T.A.: Differential Cryptanalysis Mod 232 with Applications to MD5". EUROCRYPT. (1992) pp. 71-80. ISBN 3-540-56413-6.
2. Zhe W. et al. Clean-living: Eliminating Near-Duplicates in lifetime Personal Storage. Technical Report (September 2005).
3. J.P. Kumar et al. Duplicate and Near Duplicate Documents Detection: A Review. European Journal of Scientific Research (2009).
4. Manber Udi. Finding Similar Files in a Large File System, USENIX Winter Technical conference,January, 1994, CA.
5. Andrei Z. et al. Some applications of Rabin's fingerprinting method. Sequences II: Methods in Communications, Security, and Computer Science, Springer Verlag, 1993.
6. Abdur Chowdhury et al. Collection statistics for fast duplicate document detection. ACM Transaction on Information Systems 20(2): 171-191 2002.
7. Andrei Z. Broder. Identifying and Filtering Near-Duplicate Documents, Proceedings of COM '00.
8. L. Gravano et al. Approximate string joins in a database (almost) for free. In VLDB 2001.
9. Ilinsky et al. An efficient method to detect duplicates of Web documents with the use of inverted index.
10. Ferro et al. An efficient duplicate record detection using q-grams array inverted index. In proceedings of DAWAK 2011.
11. Theobald et al. SpotSigs: Robust and Efficient Near Duplicate Detection in Large Web Collections. In proceedings of SIGIR 2008.
12. P. Indyk et al. Approximate nearest neighbors: towards removing the curse of dimensionality. In STOC 1998.
13. http://en.wikipedia.org/wiki/Viral_marketing.
14. http://en.wikipedia.org/wiki/SHA-1.
15. A. Kolcz et al. Improved robustness of signature-based near replica detection via lexicon randomization. In KDD 2004.