Automatic Identification of Informative Sections of Web Pages

Sandip Debnath, Prasenjit Mitra, Nirmal Pal, and C. Lee Giles

Abstract—Web pages—especially dynamically generated ones—contain several items that cannot be classified as the "primary content," e.g., navigation sidebars, advertisements, copyright notices, etc. Most clients and end-users search for the primary content, and largely do not seek the noninformative content. A tool that assists an end-user or application to search and process information from Web pages automatically, must separate the "primary content sections" from the other content sections. We call these sections as "Web page blocks" or just "blocks." First, a tool must segment the Web pages into Web page blocks and, second, the tool must separate the primary content blocks from the noninformative content blocks. In this paper, we formally define Web page blocks and devise a new algorithm to partition an HTML page into constituent Web page blocks. We then propose four new algorithms, *ContentExtractor*, *FeatureExtractor*, *K-FeatureExtractor*, and *L-Extractor*. These algorithms identify primary content blocks by 1) looking for blocks that do not occur a large number of times across Web pages, by 2) looking for blocks with desired features, and by 3) using classifiers, trained with block-features, respectively. While operating on several thousand Web pages obtained from various Web sites, our algorithms outperform several existing algorithms with respect to runtime and/or accuracy. Furthermore, we show that a Web cache system that applies our algorithms to remove noninformative content blocks and to identify similar blocks across Web pages can achieve significant storage savings.

Index Terms—Data mining, feature extraction or construction, text mining, Web mining, data mining, Web page block, informative block, inverse block document frequency.

1 Introduction

Search engines crawl the World Wide Web to collect Web pages. These pages are either readily accessible without any activated account or they are restricted by username and password. Whatever be the way the crawlers access these pages, they are (in almost all cases) cached locally and indexed by the search engines.

An end-user who performs a search using a search engine is interested in the *primary informative content* of these Web pages. However, a substantial part of these Web pages—especially those that are created dynamically—is content that should not be classified as the primary informative content of the Web page. These blocks are seldom sought by the users of the Web site. We refer to such blocks as *noncontent blocks*. Noncontent blocks are very common in dynamically generated Web pages. Typically, such blocks contain advertisements, image-maps, plug-ins, logos, counters, search boxes, category information, navigational links, related links, footers and headers, and copyright information.

- *S. Debnath is with the Computer Sciences and Engineering Department, The Pennsylvania State University, 111 IST Building, University Park, PA 16802. E-mail: sandipdebnath@ieee.org.*
- *P. Mitra and C. Lee Giles are with the School of Information Sciences and Technology, The Pennsylvania State University, 311A IST Building, University Park, PA 16802. E-mail: {pmitra, giles}@ist.psu.edu.*
- *N. Pal is with eBRC Main Office, The Pensylvania State University, 401 Business Administration Building, University Park, PA 16802. E-mail: nirmalpal@psu.edu.*

Manuscript received 22 Nov. 2004; revised 26 Mar. 2005; accepted 1 Apr. 2005; published online 19 July 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDESI-0479-1104.

Before the content from a Web page can be used, it must be subdivided into smaller semantically homogeneous sections based on their content. We refer to such sections as *blocks* in the rest of the paper. A block (or Web page block) $\mathcal{B}$ is a portion of a Web page enclosed within an open-tag and its matching close-tag, where the open and close tags belong to an ordered tag-set $\mathcal{T}$ that includes tags like <TR>, <P>, <HR>, and <UL>. Fig. 1, shows a Web page obtained from CNN's Web site[1] and the blocks in that Web page.

In this paper, we address the problem of identifying the primary informative content of a Web page. From our empirical observations, we found that approximately three-fourths of the dynamically generated pages found on the Web have a table in it. An HTML table is defined using the tag <TABLE>. In a table occurring in a Web page, we consider each cell to be a block. Where tables are not available, identifying blocks involves partitioning a Web page into sections that are coherent, and that have specific functions. For example, a block with links for navigation is a navigation block. Another example is an advertising block that contains one or more advertisements that are laid out side by side. Usually, a navigation block is found on the left side of a Web page. Typically, the primary informative content block is laid out to the right of a Web page. We have designed and implemented four algorithms, *ContentExtractor*, *FeatureExtractor*, *K-FeatureExtractor*, and *L-Extractor* which identify the primary content blocks in a Web page.

An added advantage of identifying blocks in Web pages is that if the user does not require the noncontent blocks or requires only a few noncontent blocks, we can delete the rest of the blocks. This contraction is useful in situations

1. http://www.cnn.com.

Fig. 1. A Web page from CNN.com and its blocks (shown using boxes).

where large parts of the Web are crawled, indexed, and stored. Since the noncontent blocks are often a significant part of dynamically generated Web pages, eliminating them results in significant savings with respect to storage cache and indexing.

Our algorithms can identify similar blocks across different Web pages obtained from different Web sites. For example, a search on Google News on almost any topic returns several syndicated articles. Popular items like syndicated columns or news articles written by global news agencies like AP or Reuters appear in tens of newspapers. Even the top 100 results returned by Google contain only a very few unique columns related to the topic because of duplicates published at different sites. Ideally, the user wants only one of these several copies of articles. Since the different copies of the article are from different newspapers and Web sites, they differ in their noncontent blocks but have similar content blocks. By separating and indexing only the content blocks, we can easily identify that two Web pages have identical content blocks, save on storage and indexing by saving only one copy of the block, and make our search results better by returning more unique articles. Even search times improve because we have less data to search.

We propose simple yet powerful algorithms, called *ContentExtractor*, *FeatureExtractor*, *K-FeatureExtractor*, and *L-Extractor* to identify and separate content blocks from noncontent blocks. We have characterized different types of blocks based on the different features they possess. *FeatureExtractor* is based on this characterization and uses heuristics based on the occurrence of certain features to identify content blocks. *K-FeatureExtractor* is a special modification of *FeatureExtractor* which performs better in a wide variety of Web pages. *ContentExtractor* identifies noncontent blocks based on the appearance of the same block in multiple Web pages. *L-Extractor* uses various block-features and train a Support Vector (SV) based classifier to identify a informative block versus a noninformative block.

First, the algorithms partition the Web page into blocks based on heuristics. These heuristics are based on our previous study of HTML editing style over a few thousand Web pages. Lin and Ho [18] have proposed an entropy-based algorithm that partitions a Web page into blocks on the basis of HTML tables. In contrast, we not only consider HTML tables, but also other tags, combined with our heuristics to partition a Web page. Second, our algorithms classifies each block as either a content block or a noncontent block. While the algorithm decides whether a block, $\mathcal{B}$, is content or not, it also compares $\mathcal{B}$ with stored blocks to determine whether $\mathcal{B}$ is similar to a stored block. Both *(K-)FeatureExtractor* and *ContentExtractor* produce excellent precision and recall values and runtime efficiency and, above all, do not use any manual input and require no complex machine learning process. *L-Extractor* is still under experimentation, and it produces fairly good accuracy.

While operating on several thousand Web pages obtained from news and various other Web sites, our algorithms significantly outperform their nearest competitor—the Entropy-based blocking algorithm proposed by Lin and Ho [18]. We also compared *ContentExtractor* with the Shingling algorithm devised by Ramaswamy et al. [21], [22]. *ContentExtractor* achieves similar savings on storage requirements as the Shingling algorithm. However, it outperforms the Shingling algorithm significantly with respect to runtime, showing that simple heuristics can suffice to identify primary content blocks in Web pages.

The rest of the paper is organized as follows: In Section 2, we have discussed the related work. We define the concept of "blocks" and a few related terms in Section 3. We describe our algorithms in Sections 4, 5, and 7. We outline our performance evaluation plan and the data set on which we ran our experiments in Section 6. We compare our algorithms with the *LH* and Shingling algorithm in Section 6.4. We indicate our future work and conclude in Section 8.

2 Related Work

Yi and Liu [26], [19] have proposed an algorithm for identifying noncontent blocks (they refer to it as "noisy" blocks) of Web pages. Their algorithm examines several Web pages from a single Web site. If an element of a Web page has the same style across various Web pages, the element is more likely than not to be marked as a noncontent block. Their algorithm also looks at the entropy of the blocks to determine noncontent blocks. Their technique is intuitively very close to the concept of "information content" of a block. This is one of the very innovative ideas we have studied. Our algorithms only look at the inverse block document frequency (defined below) and features of blocks. In order to identify the presentation styles of elements of Web pages, Yi and Liu's algorithm constructs a "Style Tree." A "Style Tree" is a variation of the DOM substructure of Web page elements. If there are Web pages whose elements have the same style but different contents and yet are noncontent blocks, our algorithms would not be able to detect that. However, in practice, we have seen that our algorithms even in the presence of advertisement images that vary from page to page can identify them as noncontent blocks by making use of the text in the blocks that are almost the same. Since our algorithms use simple heuristics to determine noncontent blocks, it does not incur the overhead of constructing "Style Trees."

Another work that is closely related is the work by Lin and Ho [18]. The algorithm they proposed also tries to partition a Web page into blocks and identify content blocks. They used the entropy of the keywords used in a block to determine whether the block is redundant. We believe that we have a more comprehensive definition of blocks and demonstrate that we have designed and implemented an algorithm that gives better precision and recall values than their algorithm as shown below.

Cai et al. [4] have introduced a vision-based page segmentation (VIPS) algorithm. This algorithm segments a Web page based on its visual characteristics, identifying horizontal spaces, and vertical spaces delimiting blocks much as a human being would visually identify semantic blocks in a Web page. They use this algorithm to show that better page segmentation and a search algorithm based on semantic content blocks improves the performance of Web searches. Song et al. [25] have used VIPS to find blocks in Web pages. Then, they use Support Vector Machines (SVM) and Neural Networks to identify important Web pages. We observed that VIPS is significantly more expensive than our simple blocking algorithm. So, in one of our algorithms (*L-Extractor*), for the first step, we used our blocking algorithm and, in the second step, we used a SVM-based algorithm to achieve good results. However, we also show that one can use even simpler and less expensive techniques as used in *ContentExtractor* and *k-FeatureExtractor* to identify primary content blocks in Web pages.

Ramaswamy et al. [21], [22] propose a Shingling algorithm to identify fragments of Web pages and use it to show that the storage requirements of Web caching are significantly reduced. We show below that a *ContentExtractor*-based algorithm provides similar savings for Web caching, however, *ContentExtractor* is significantly less expensive than the Shingling algorithm.

Bar-Yossef and Rajagopalan [3] have proposed a method to identify frequent templates of Web pages and pagelets (identical to our blocks). Yi and Liu argue that their entropy-based method supersedes the template identification method. We show that our method produces better results than the entropy-based method.

Kushmerick [15], [16] has proposed a feature-based method that identifies Internet advertisements in a Web page. It is solely geared toward removing advertisements and does not remove other noncontent blocks. While their algorithm can be extended to remove other noncontent blocks, its efficacy for the general Web cleaning problem has not been studied. Besides, their algorithm generates rules from training examples using a manually specified procedure that states how the features to be used can be identified. This manual specification is dependent upon applications. Our algorithms do not require any manual specification or training data set (except *L-Extractor*).

There has been substantial research on the general problem of extracting information from Web pages. Information extraction or Web mining systems try to extract useful information from either structured, or semistructured documents. Since a large percentage of dynamically generated Web documents have some form of underlying templates, Wrapper [15], [16], Roadrunner [9], Softmealy [12], and other systems try to extract information by identifying and exploiting the templates. Systems like Tsimmis [5] and Araneus [2] depend on manually provided grammar rules. In Information Manifold [14], [17], Whirl [7], or Ariadne [1], the systems tried to extract information using a query system that is similar to database systems. In Wrapper systems [16], the wrappers are automatically created without the use of hand-coding. Kushmerick [15], [16] have found an inductive learning technique. Their algorithm learns a resource's wrapper by reasoning about a sample of the resource's pages. In Roadrunner [9], a subclass of regular expression grammar (UFRE or Union Free Regular Expression) is used to identify the extraction

rules by comparing Web pages of the same class and by finding similarities or dissimilarities among them. In Softmealy [12], a novel Web wrapper representation formalism has been presented. This representation is based on a finite-state transducer (FST) and contextual rules, which allow a wrapper to wrap semistructured Web pages containing missing attributes, multiple attribute values, variant attribute permutations, exceptions, and typos, the features that no previous work can handle. A SoftMealy wrapper can be learned from labeled example items using a simple induction algorithm. For other semistructured wrapper generators like Stalker [20], a hierarchical information-extraction technique converts the complexity of mining into a series of simpler extraction tasks. It is claimed that Stalker can wrap information sources that cannot be learned by existing inductive learning techniques. Most of these approaches are geared toward learning the regular expressions or grammar induction [6] of the inherent structure or the semistructure and, so, computational complexities are quite high.

The efforts mentioned above are involved in extracting information that originally came from databases. This underlying data stored in databases is very structured in nature. Our work concentrates on Web pages where the underlying information is unstructured text. The techniques used for information extraction are applied on entire Web pages, whereas they actually seek information only from the primary content blocks of the Web pages.

Using our algorithm to extract the primary content blocks of the Web pages as a preprocessing step, and then running the information extraction algorithms on the primary content blocks will reduce the complexity and increase the effectiveness of the extraction process.

Our preliminary work [10] shows great improvements in extracting the informative blocks from the Web pages. We can enhance our feature-based algorithm by using machine learning mechanisms to select the useful features that are used to identify the noncontent blocks. Our study of using Support Vector Learning approach in this context is described in Section 7.

3 Segmenting Web Pages into Blocks

In this section, we define the concept of "blocks" in Web pages and a few other related terms. Most Web pages on the Internet are still written in HTML [8]. Even dynamically generated pages are mostly written with HTML tags, complying with the SGML format. The layouts of these SGML documents follow the Document Object Model tree structure of the World Wide Web Consortium.[2] Out of all of these tags, Web authors mostly use <TABLE> to design the layouts. Our algorithm uses <TABLE> as the first tag on the basis of which it partitions a Web page. After <TABLE>, it uses <TR>, <P>, <HR>, <UL>, <DIV>, and <SPAN>, etc., as the next few partitioning tags in that order. We selected the order of the tags based on our observations of Web pages and believe that it is a natural order used by most Web page designers (according to our study of HTML editing style for a few thousand Web pages from various sources and formats).

2. W3C or http://www.w3c.org.

For example, <TABLE> comes as a first partitioning tag since we see more instances of <UL> in a table cell than <TABLE>s coming inside <LI>, an item under <UL>. Our algorithms partition a Web page based on the first tag in the list to identify the blocks, and then subpartitions the identified blocks based on the second tag and so on. It continues to partition until there is any tag left in a block in the block-set which is part of the list of tags. This ensures that the blocks are atomic in nature and no further division is possible on them. The partitioning algorithm is illustrated in Section 4.1 and this tag-set is called the partitioning tag-set.

3.1 Block Features

By definition, blocks may include other smaller blocks. But, in our implementation, as we described above we have taken all atomic blocks for computation purpose (except in a few cases in *FeatureExtractor*). Atomic blocks will have features like text, images, applets, javascript, etc. Actually, all HTML tags (Following W3C (http://w3c.org)) except the tags in partitioning tag-set are included for feature analysis. A *block-feature set* is a set of features that a block contains. Several features are associated with their respective standard tags but not all features have standard tags. For example, an image is always associated with the tag <img>, however, the text feature has no standard tag. For features that are associated with a tag, we used the W3C guidelines on HTML pages to make the full list of features. The features of a block that we have used includes, but are not limited to Text, Text-tag, List, Table, Link, Object, Frame, Form, Script, Style-Sheet, etc. The most important and nice quality of algorithm is that we can update this list as time and version of HTML pages change, without doing any fundamental changes in the algorithm.

Examples of individual features in the feature vectors constructed by our algorithms are: the number of terms (in case of text feature), the number of images (in case of <IMG> tag), the number of javascripts (in case of <SCRIPT> tag), etc. However, for text blocks, simply taking the number of terms in the block may result in falsely identifying two blocks as similar. Therefore, we augment the features by adding a binary feature for each term in the corpus of documents. If a term occurs in a block, the entry in the corresponding feature vector is a one, otherwise, it is zero. If other features are deemed important, our framework can be easily modified by adding new features and adjusting the weights of those features while computing the similarity between blocks.

3.2 Inverse Block Document Frequency and Block Similarity

ContentExtractor computes the Inverse Block Document Frequency ($IBDF$) as defined below. For example, if a block appears in multiple Web pages, say, in most of CNN's Web pages, the block will have a smaller Inverse Block Document Frequency ($IBDF$) than one that appears only in one Web page.

Let us assume $IBDF^i$ represents the $IBDF$ of a block $\mathcal{B}_i$ in a set of pages $\mathcal{S}$. Typically, the set $\mathcal{S}$ consists of similar pages from the same source. $IBDF^i$ is inversely proportional to the number of Web pages the block $\mathcal{B}_i$ occurs in.

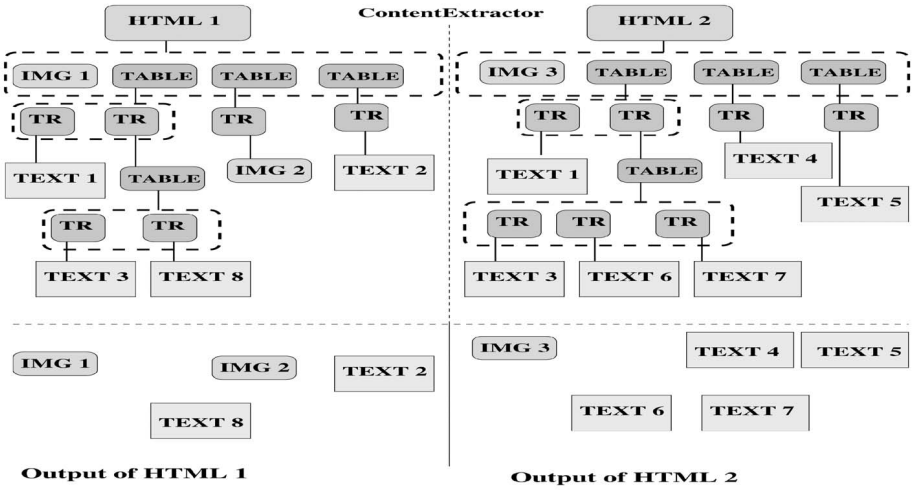

Fig. 2. Two Web pages' block structures as seen by *GetBlockSet*. The output from them is shown under the dotted line.

So, $\mathcal{S}$ is a set of Web pages of the same class, i.e., obtained from the same source. Therefore,

$$\mathcal{S} = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \ldots \mathcal{P}_M\}, \tag{1}$$

where $\mathcal{P}_i$s $(\forall i \in M)$ are individual HTML pages from that source, and

$$IBDF^i \equiv f\left(\frac{1}{|\mathcal{S}^i| + 1}\right), \tag{2}$$

where

$$\mathcal{S}^i = \cup\{\mathcal{P}_l : Sim(\mathcal{B}_i, \mathcal{B}_k) < \epsilon, \forall \mathcal{B}_k \in \mathcal{P}_l, \forall \mathcal{P}_l \in \mathcal{S}\}. \tag{3}$$

f denotes a function, usually linear or log function. The function $Sim(\mathcal{B}_i, \mathcal{B}_k)$ is a similarity measure of the two blocks. An expert provides the threshold ϵ.

There may be a question regarding whether the basis of our algorithm is a rule-based technique. Actually, we draw analogy between the TF-IDF measure in vector-space model [24] and our $IBDF$ measure. As we eliminate the commonly occurring or redundant words or phrases in a collection by applying the IDF measure of all the words and phrases, we here extend the same concept for blocks. If we consider blocks as the atomic units in a Web page, it is easy to visualize that the blocks having lower $IBDF$ values or high frequency of occurring in several Web pages will be eliminated as redundant blocks. TF-IDF measure and related algorithms are undoubtedly not rule-based algorithms. Likewise $IBDF$ measure and *ContentExtractor* should not be considered as rule-based approaches.

4 Algorithm: ContentExtractor

The input to the algorithms is a set (at least two) of Web pages belonging to a *class* of Web pages. A class is defined as a set of Web pages from the same Web site whose designs or structural contents are very similar. A set of Web pages dynamically generated from the same script is an example of a class. The output of the algorithms are the primary content blocks in the given class of Web pages.

The first step of all our algorithms is to use the *GetBlockSet* routine (described next) to partition each page into blocks.

4.1 GetBlockSet

The *GetBlockSet* routine takes an HTML page as input with the ordered tag-set.

GetBlockSet takes a tag from the tag-set one by one and calls the *GetBlocks* routine for each block belonging to the set of blocks, already generated. New subblocks created by *GetBlocks* are added to the block set and the generating main block (which was just partitioned) is removed from the set. The $First$ function gives the first element (tag) of an ordered set, and the $Next$ function gives the consecutive elements (tags) of an ordered set.

4.2 GetBlocks

GetBlocks takes a full document or a part of a document, written in HTML, and a tag as its input. It partitions the document into blocks according to the input tag. For example, in case of the <TABLE> tag given as input, it will produce the DOM tree with all the table blocks. It does a breadth-first search of the DOM tree (if any) of the HTML page. If the input tag is <TABLE> and there is no table structure available in the HTML page, it does not partition the page. In that case, the whole input page comes back as a single block. In case of other tags such as <P>, it partitions the page/block into blocks/subblocks separated by those tags. Fig. 2 shows the structure of two HTML pages. It also shows the blocks that our blocking algorithm identifies for each of these pages (under the dotted line).

4.3 Identifying Primary Content Blocks

After the blocks have been identified, the second step of the process involves identifying the primary content blocks and separating them from the noncontent blocks. All four algorithms identify the primary content blocks in the Web pages, but their methodologies are different.

4.4 ContentExtractor

We show the pseudocode for *ContentExtractor* in Algorithms 1 and 2 (Figs. 3 and 4). It calculates the $IBDF$

Algorithm 1: *ContentExtractor*

Input : Set $\mathcal{S}$ of HTML pages, Sorted tag-set $\mathcal{T}$
Output: Primary Content Blocks and their associated pages in $\mathcal{S}$
begin
  $\mathcal{M}_{BD} \longleftarrow \emptyset$
  { Here the $\mathcal{M}_{BD}$ matrix is the block-document matrix where rows represent document and columns represent block identifier.}
  for *each* $H^k \in \mathcal{S}$ **do**
    { Here B^k represents the kth row of the $\mathcal{M}_{BD}$ matrix. }
    $B^k \longleftarrow \mathbf{GetBlockSet}(H^k, \mathcal{T})$
    $\mathcal{M}_{BD}^k \longleftarrow B^k$

  for *each* $b_{ij} \in \mathcal{M}_{BD}$ **do**
    ${IBDF_{ij}}^{-1} \longleftarrow 1$
    for *each* $b_{kl} \in \mathcal{M}_{BD}$ **do**
      { Here $i \neq k$.}
      $Sim_{ijkl} \longleftarrow Sim(b_{ij}, b_{kl})$
      if $Sim_{ijkl} > \epsilon$ **then**
        ${IBDF_{ij}}^{-1} \longleftarrow Update({IBDF_{ij}}^{-1})$
        {Update Recalculates $IBDF^{-1}$}

  { If $IBDF^{-1}$ value above threshold we will produce the output }
  for *each* $b_{ij} \in \mathcal{M}_{BD}$ **do**
    if ${IBDF_i}^{-1} > \theta^{-1}$ **then**
      Output the content of the block
end

(*Function GetBlockSet :*)
Input : HTML page H, Sorted tag-set $\mathcal{T}$
Output: Set of Blocks in H
begin
  $B \longleftarrow H$; // set of blocks, initially set to H.
  $f \longleftarrow Next(\mathcal{T})$
  while $f \neq \emptyset$ **do**
    $b \longleftarrow First(B)$
    while $b \neq \emptyset$ **do**
      if b *contains* f **then**
        $B^N \longleftarrow GetBlocks(B, f)$ $B \longleftarrow (B - b) \cup B^N$
      $b \longleftarrow Next(B)$
    $f \longleftarrow Next(\mathcal{T})$
end

Fig. 3. Algorithm 1: *ContentExtractor*.

values of each block. For implementation purposes, we compute the $IBDF$ values as a counter, and compare with θ, which is the same as comparing $IBDF^{-1}$ with θ^{-1}. The algorithm used a similarity measure function Sim, to find out the similarity between two blocks.

4.4.1 The Similarity Function and Threshold

Given two blocks, Sim returns the cosine between their block feature vectors. We used a threshold value of $\epsilon = 0.9$. That is, if the similarity measure is greater than the threshold value, then the two blocks are accepted as identical. The threshold value can be changed according to the needs of the application and affects the precision and recall of the algorithm. Blocks that occur rarely across different Web pages, i.e., have low $IBDF$s, are output as the primary content blocks.

4.4.2 Complexity Measure

The computational complexity of this approach is dependent on the computation of the similarity measure between blocks and the computation of the $IBDF$s of the blocks.

Let us assume there are N blocks per page and the total number of documents in a class is M. According to the definition of a class, these pages are derived from the same script or from the same Web site. In practical cases, pages derived from the same class are of the same design. Their headers, left panels, or the footers are similar (depending on the threshold ϵ). Thus, during the comparison, the number of completely new blocks coming from the second page is pretty low. Therefore, when the algorithm compares pages P_i and P_{i+1}, we can arguably assume that the similar blocks will largely outnumber the dissimilar blocks.

Suppose the average number of new blocks in $P_i + 1$ that are not present in P_i is κ. Then, from above discussion

Algorithm 2: *Sim Function (ContentExtractor Algorithm continued)*

(*Function Sim :*)

Input : $Block_1, Block_2$

Output: Similarity Measure

begin

//FeatureVector produces a vector of all
//features as enlisted and described above
$\overline{\mathcal{F}_1} \longleftarrow FeatureVector(Block_1)$
$\overline{\mathcal{F}_2} \longleftarrow FeatureVector(Block_2)$
return $\cos(\overline{\mathcal{F}_1}, \overline{\mathcal{F}_2})$

end

Fig. 4. Algorithm 2: *Sim Function (ContentExtractor Algorithn continued).*

$\kappa \ll N$. Accordingly, after the first comparison, a $(N+\kappa) \times M$-dimensional Block-Document matrix will be formed. The computational complexity of this step is $O(N^2)$. After these pages are compared, the blocks of the third page will be compared with the combined set of blocks coming from the first two pages. When the second step of the comparison will be performed, the cost of computation will be increased. Ultimately, the total number of comparisons will be

$$\begin{aligned} N^2 + (N+\kappa) \times N + (N+2\kappa) \times N + \ldots + (N+(M-2)\kappa) \\ \times N = (M-1)N^2 + \frac{\kappa}{2}(M^2 - 3M + 2)N \\ = \left(MN^2 - N^2 + \frac{\kappa}{2}M^2N - \frac{3\kappa}{2}MN + \kappa N \right) \\ = O(M^2N) \end{aligned} \tag{4}$$

as $M >> N$ and $\kappa << N$.

The Block-Document matrix computation will be dependent on the value of M or the number of pages in the set and the average number of blocks in each individual page. In the future, we would like to explore if taking a smaller number of pages in a set is enough for identifying the irrelevant blocks. The time complexity to make the sorted block-document matrix is $O(M^3N^2log(N))$.

If all Web pages in the same class are dynamically generated from the same template, and running *ContentExtractor* for all M documents is excessively costly, in practice, we can identify the template using fewer than M documents. Then, for all M documents, the primary content block that appears at a fixed place in the template can be extracted using the template.

5 Algorithm: FeatureExtractor/ K-FeatureExtractor

We now show our second algorithm, *FeatureExtractor*. We designed *FeatureExtractor* such that any informative block (corresponding to any feature) can be identified. For example, *FeatureExtractor* invoked with the features text, image, and links, identifies the text blocks, image blocks, or navigational blocks as the primary content blocks, respectively. We show the pseudocode for *FeatureExtractor* in Algorithm 3 (Fig. 5).

5.1 Block Features

The following list describes the features of a Web page block that we have used in our implementation. A Web page block can have any or all features of an HTML page. The W3C HTML guidelines have been followed here:

- Text: The text content inside the block.
- Text-tag: The text tags, e.g., <h1>, <h2>, etc., inside the block.
- List: The lists available inside the block.
- Table: Available tables inside the block.
- Link: URLs or links inside the block.

Algorithm 3: FeatureExtractor

Input : Set of HTML pages H, Sorted Tag Set $\mathcal{T}$, Desired Feature $\mathcal{F}_I$

Output: Content Blocks of H

Feature: Feature set $\mathcal{F}_S$ used for block separation sorted according to importance taken from $\mathcal{T}$

begin

$B \longleftarrow GetBlockSet(B, \mathcal{T})$

{ $\mathcal{W}$ is the output variable that records potential output block sets}
$W \longleftarrow \emptyset$
for *each* $b \in B$ **do**
  $P_1 \longleftarrow Pr(\mathcal{F}_I|\mathcal{F})$
  $P_2 \longleftarrow Pr((\mathcal{F} - \mathcal{F}_I)|\mathcal{F})$
  if $P_1 > P_2$ **then**
    $\mathcal{W} \longleftarrow W \cup b$

{ Now depending on the condition or choice we will produce output from the set $\mathcal{W}$ }
for *each* $b \in \mathcal{W}$ **do**
  $P_b \longleftarrow Pr(\mathcal{F}_I|\mathcal{F}, \mathcal{W})$
  //$\mathcal{F}_I = \mathcal{T}_I$ in the experiment
{ Output: Sort $\mathcal{W}$ according to the Probability value P_b and (1) Produce the content of the Winner block }

end

Fig. 5. Algorithm 3: FeatureExtractor.

Algorithm 4: K-FeatureExtractor

begin

... same as *FeatureExtractor* except the last statement ...

{ Output: Sort $\mathcal{W}$ according to the probability value P_b and (2) Use k-means clustering and take high probability cluster(s). Combine the text contents from all of the blocks. }

end

Fig. 6. Algorithm 4: K-FeatureExtractor.

- Object: Image, Applet, etc., available in the block.
- Frame: Frames inside the block. Usually, it is rare to have a frame in the block, but to make the list complete it has been added.
- Form: Forms available inside the block.
- Script: Javascripts or other types of scripts written in the block.
- Style-Sheet: This is also to make the list complete and compliant to W3C guidelines. Styles are usually important for browser rendering, and usually included inside other tags, like links and tables, etc.

A question may arise here about why we are taking these features. As aforesaid, all these blocks are HTML blocks and we are trying to find out a particular block or set of blocks which can be identified by the block-property such as text-blocks or image-blocks. In FeatureExtractor, we are looking for all the text-blocks and, so, we need to compare the properties of a block against other blocks. These comparisons are only possible if we consider all the HTML tags as the feature-set of the blocks. As we mentioned earlier, we can update this list if we so desire because of changes in HTML features or because of an application's updated preferences of desirable features easily without fundamentally changing the algorithm.

Unlike the *ContentExtractor* algorithm, the *FeatureExtractor* algorithm does not depend on multiple Web pages but depends on the feature-set and the chosen feature for output. The set features are HTML features as explained before. For example, let us consider the chosen feature is text ($\mathcal{T}_I$). Now, our algorithm calculates a value for each feature in each block. Say, a block contains 1,000 words and two images and three links and an applet, and the maximum values of words, images, links, and applets contained in blocks in the data set are 2,000, 4, 50, and 3. Then, the values for the features in the given block are $1,000/2,000$, $2/4$, $3/50$, and $1/3$, respectively. After that, we put each block in the winner-basket if the sum of the feature values of the desired features is greater than the sum of the feature values of the rest of the features. From this winner-basket, we recompute the feature values for this new set of blocks, and chose the one with highest value of desired feature.

Now, according to this algorithm a block with a single word and nothing else would be the obvious winner and will be chosen. In most practical cases, this scenario did not arise. And, also, we do not consider a single row or column of a table as a block. We consider the whole table (in the highest depth of table tree) as a block. So, the chance of getting a block with a single word is distant.

5.2 K-FeatureExtractor

Though *FeatureExtractor* performs with high precision and recall for one of our data sets, it may not do so in general and can be improved. For Web pages with multiple important text blocks, a typical reader may be interested in all the sections not just one of them (winner of *FeatureExtractor*). For example, an end-user may be interested in all the reviews available from a page on Amazon.com and each review is in a separate block. General shopping sites, review sites, chat forums, etc., may all contain multiple blocks of important textual information. *FeatureExtractor* shows poor precision and recall as it produces only one text-block with highest probability, while other important blocks are not retrieved. To overcome this drawback, we revised the last part of the *FeatureExtractor* and named the new algorithm as *K-FeatureExtractor* (Algorithm 4 in Fig. 6). To handle more general Web pages of varied editing-styles, we improved the *FeatureExtractor* algorithm. Instead of taking just the winner block from the winner-basket, we apply a k-means clustring algorithm to select the best probability blocks from the basket. This helps us get high precision and recall from shopping Web sites and review Web sites and, in general, a much broader range of Web sites. The results from using the *K-FeatureExtractor* for these types of Web pages are shown in Table 3 separately. Needless to mention that *FeatureExtractor* did not do well for these Web pages. *K-FeatureExtractor* uses an adaptive K-means clustering on the winner set to retrieve multiple winners as opposed to *FeatureExtractor* that selects a single winner. The usual values of k taken are 2 or 3, and the initial centroids are chosen from the sorted list at equidistant index values. After the clustering is done, the high probability cluster(s) are taken and the corresponding text contents of all those blocks are taken as the output.

6 Experimental Evaluation

In this section, we present an empirical evaluation of our methods. We also compare our algorithms with two other major competitors.

6.1 First Comparison: With the LH Algorithm

We implemented and compared our algorithm with *LH*, the entropy-based algorithm proposed by Lin and Ho [18]. They use the terms *precision* and *recall* to refer to the metrics to evaluate their algorithm. Although, the use of these terms are somewhat different from their usual sense in the field, "Information Retrieval," in order to avoid confusion, we use the same terms (added with a "b-" for blocks) to refer to the evaluation metrics of our work.

TABLE 1
Details of the Data Set

Site	*Address*	*Category*	*Number*
ABC	http://www.abcnews.com	Main Page, USA, World, Business, Entertainment, Science/Tech, Politics,Living	415
BB	http://www.bloomsberg.com	Main Page, World, Market, US Top Stories, World Top Stories, Asian, Australia/New Zealand, Europe, The Americas	510
BBC	http://www.bbc.co.uk	Main Page, The Continents, Business, Health, Nature, Technology, Entertainment	890
CBS	http://www.cbsnews.com	Main Page, National, World, Politics, Technology, Health, Entertainment	370
CNN	http://www.cnn.com	Main Page, World, US, All Politics, Law, Tech(nology), Space (Technology), Health, Showbiz, Education, Specials	717
FOX	http://www.foxnews.com	Main Page, Top Stories, Politics, Business, Life, Views	476
FOX23	http://www.fox23news.com	Main Page, General, Local, Regional, National, World, In Depth, Sports, Business, Entertainment, Health	658
IE	http://www.indianexpress.com	Main Page, International, Sports, National Network, Business, Headlines	269
IT	http://www.indiatimes.com	Main Page, Main Stories, Top Media Headlines	454
MSNBC	http://www.msnbc.com	Main Page, Business, Sports, Technology an Science, Health, Travel	647
YAHOO	http://news.yahoo.com	Main Page, Top Stories, US (National), Business, World, Entertainment, Sports, Technology, Politics, Science	505
Shopping	http://www.shopping.com	Miscellaneous Products	100
Amazon	http://www.amazon.com	Book Pages	100
Barnes And Noble	http://www.bn.com	Book Pages	100
Epinion	http://www.epinions.com	Reviews	100

The number of pages taken from individual categories is not shown due to the enormous size of the latex table, but the interested reader can contact authors to get the details.

6.2 Metric Used

Precision is defined as the ratio of the number of relevant items (actual primary content blocks) r found and the total number of items (primary content blocks suggested by an algorithm) t found. Here, we used a block level precision and so we call it *b-Precision*:

$$b\text{-}Precision = \frac{r}{t}. \quad (5)$$

Recall has been defined as the ratio of the number of relevant items found and the desired number of relevant items. The desired number of relevant items includes the number of relevant items found and the missed relevant items m. In case of blocks, we call it as block level recall or $b\text{-}Recall$:

$$b\text{-}Recall = \frac{r}{r+m}. \quad (6)$$

Similar to the way it is defined in information retrieval literature by Van Rijsbergen [23], we can refer to the F-measure here as the *b-F-measure* and define it as:

$$b\text{-}F\text{-}measure = \frac{2 * (b\text{-}Precision) * (b\text{-}Recall)}{(b\text{-}Precision) + (b\text{-}Recall)}. \quad (7)$$

6.3 Data Set

Exactly like Lin and Ho, we chose several Web sites from the news domain. We crawled the Web for news articles and other types of Web sites to collect documents. The details (name, source, category, number) of the data set are shown in Table 1.

In total, we took 15 different Web sites including news, shopping, opinion posting Web sites, etc., whose designs and page-layouts are completely different. In Tables 2 and 3, we took 11 different news Web sites for the first comparison. Unlike Lin and Ho's data set [18] that is obtained from one fixed category of news sections (only one of them is "Miscellaneous" news from CDN), we took random news pages from every section of a particular Web site. This choice makes the data set a good mix of a wide variety of HTML layouts. This step was necessary to compare the robustness of their algorithm to ours.

6.4 Performance Comparison

We implemented all four algorithms in Perl 5.8.0 on a Pentium-based Linux platform. With the generous help from a few graduate students and professors, we calculated the b-precision and b-recall values for each Web site and layout category for text feature. These values are shown in Tables 2 and 3.

Our algorithms outperform *LH* in all news sites in all categories. The b-recall is always good since all algorithms could find most relevant blocks but the results obtained by running the *LH* algorithm were less precise than those obtained by *ContentExtractor* since the former algorithm also includes lots of other noncontent blocks.

We believe that the primary reason for the poor b-precision of *LH* is because of the greedy approach taken by their algorithm while identifying the solution. A second reason is that the *LH* algorithm works at the feature level instead of the block level. *LH* gives a high redundancy score

TABLE 2
Block Level Precision and Recall Values from *LH* Algorithm, *ContentExtractor*, and *FeatureExtractor*

Site	b-Prec of LH	b-Recall of LH	b-F-measure of LH	b-Prec of CE	b-Recall of CE	b-F-measure of CE	b-Prec of (K-) FE	b-Recall of (K-) FE	b-F-measure of (K-) FE
ABC	0.811	0.99	**0.89**	0.915	0.99	**0.95**	1.00	1.00	**1.00**
BB	0.882	0.99	**0.93**	0.997	1.00	**0.998**	1.00	1.00	**1.00**
BBC	0.834	0.99	**0.905**	0.968	1.00	**0.983**	1.00	1.00	**1.00**
CBS	0.823	1.00	**0.902**	0.972	1.00	**0.985**	0.98	0.977	**0.978**
CNN	0.856	1.00	**0.922**	0.977	1.00	**0.988**	0.98	0.98	**0.98**
FOX	0.82	1.00	**0.901**	0.967	1.00	**0.983**	1.00	0.99	**0.994**
FOX23	0.822	1.00	**0.902**	0.985	1.00	**0.992**	1.00	1.00	**1.00**
IE	0.77	0.95	**0.85**	0.911	0.993	**0.95**	0.93	0.99	**0.959**
IT	0.793	0.99	**0.878**	0.924	0.981	**0.951**	0.96	0.98	**0.969**
MSNBC	0.802	1.00	**0.89**	0.980	1.00	**0.989**	0.92	1.00	**0.95**
YAHOO	0.730	1.00	**0.84**	0.967	1.00	**0.98**	1.00	0.95	**0.974**

The second, third, and fourth columns are from LH *algorithm, the fifth, sixth, and the seventh columns are from* ContentExtractor *and the eighth, ninth, and 10th columns are from* (K-)FeatureExtractor. *We put K in parentheses to imply that these results are almost the same from* FeatureExtractor *and* K-FeatureExtractor.

TABLE 3
Block Level Precision and Recall Values from *LH* Algorithm, *ContentExtractor*, and *FeatureExtractor*.

Site	b-Prec of LH	b-Recall of LH	b-F-measure of LH	b-Prec of CE	b-Recall of CE	b-F-measure of CE	b-Prec of K-FE	b-Recall of K-FE	b-F-measure of K-FE
Shopping	0.79	1.00	**0.88**	0.971	1.00	**0.985**	1.00	0.99	**0.994**
Amazon	0.771	0.99	**0.86**	0.98	1.00	**0.989**	1.00	0.967	**0.983**
Barnes And Noble	0.81	1.00	**0.895**	0.982	0.98	**0.98**	1.00	0.968	**0.983**
Epinion	0.79	1.00	**0.88**	0.97	1.00	**0.984**	1.00	0.956	**0.977**

The second, third, and fourth columns are from LH *algorithm, the fifth, sixth, and the seventh columns are from* ContentExtractor *and the eighth, ninth, and 10th columns are from* K-FeatureExtractor. *Due to poor performance of* FeatureExtractor *for these Web pages (which we do not show here) we improved it to* K-FeatureExtarctor.

to features that occur across Web pages. The redundancy score of a block is proportional to the weighted sum of the redundancy scores of each feature it contains. Instead of looking at occurrences of features across Web pages, the *ContentExtractor* algorithm looks at occurrences of similar blocks across pages. This fundamental difference results in better b-precision obtained by our algorithm.

The *FeatureExtractor* algorithm only works well on Web pages where the primary Web pages have one dominant feature. For example, in news Web pages, text is the dominant feature. However, if we go to a domain where the primary content is a mix of multiple features, *FeatureExtractor*'s b-precision suffers. If *FeatureExtractor* has to be deployed in such a domain, it must be modified to handle multiple features and use a weighted measure of the presence of multiple features to identify the primary content pages. Due to the dependence of *FeatureExtractor* on one particular feature, we expect it to perform poorer than *ContentExtractor* in more general cases where the dominant features in a primary content block are not known.

In other cases (the last four Web sites in Table 1) where there is a single dominant feature but multiple blocks should be in the winner set (not just a single winner), *FeatureExtractor* may not perform well. And, supporting our intuition, *FetaureExtractor* resulted in poor performance for these Web sites. Because of that, we used *K-FeatureExtractor* for these Web sites. The results are shown in Table 3 compared to our *ContentExtractor* and *LH* algorithms.

6.5 b-Precision and b-Recall

Both *FeatureExtractor* and *ContentExtractor* performed better than *LH* in almost all cases. Actually, with *ContentExtractor*, there are few or almost no missing blocks because the algorithm discards only repetitive blocks and keeps the other blocks and repetitive blocks have low real information content. In the news domains, most primary content blocks were dominated by text content and, so, *FeatureExtractor* deployed with the mandate to find blocks with predominantly text content, performs well. The b-precision of *ContentExtractor* increases with the number of pages involved in $IBDF$ calculation. We compare the features of the first three algorithms in Table 4.

6.6 Execution Time

Fig. 7 shows execution time taken by the three algorithms (*LH*, *ContentExtractor*, and *FeatureExtractor*) averaged over all test Web pages. We did not include *K- FeatureExtarctor* as the time taken by it would be same and will overlap the lowermost curve to make it more cluttered. From the figure, it is clear that our algorithms outperform the *LH* algorithm

TABLE 4
A Property-Wise Comparison Table for Three Algorithms.

Property	LH	ContentExtractor	(K-)FeatureExtractor
b-Precision	*Low*	*High*	**Very High**
b-Recall	*High*	*Very High*	**Very High**
Number of pages needed	*All the pages to calculate Entropy of features*	*Very few* $(5-10)$ *pages from same class are enough to give high performance*	**A single HTML page is all that is needed**
Time of completion	*Always more than ContentExtractor*	**Less than LH (shown in figure 3)**	**Even less than ContentExtractor**

Note that (K-)FeatureExtractor *here represents both* FeatureExtractor *and* K-FeatureExtractor. *For the case of* (K-)FeatureExtractor, *we took the b-precision, b-recall, b-F-measure, and all other comparisons with respect to text feature.*

by a significant margin. We can further increase the performance of *ContentExtractor* by generating a template of a set of Web pages using five to 10 Web pages from a site and using the template to extract primary content blocks.

Here, in Table 4, we present a comparison table for the features of both algorithms. This table shows the clear difference between *LH* and our *ContentExtractor* and *(K)-FeatureExtractor* algorithms.

6.7 Second Compariosn: With Shingling Algorithm

In this section, we compare one of our algorithms with the Shingling algorithm proposed by Ramaswamy et al. [21]. Regarding the way this algorithm is designed, it is the closest to our *ContentExtractor* algorithm and, therefore, we will attempt to compare these two algorithms side-by-side. They also partition the HTML page into blocks (in their case they call them the nodes of the AF or Augmented Fragment tree). Then, they characterize each individual node with different properties, such as $SubtreeValue$, $SubtreeSize$, $SubtreeShingles$, and others. The detection of similar nodes was done by an algorithm called "Shared Fragment Detection." The Shingling algorithm was designed to save storage for Web-caches. First, we compare the storage requirements of a Web-cache using a *ContentExtractor* algorithm versus one obtained by the Shingling algorithm. Table 5 shows a comparison of the Shingling algorithm with *ContentExtractor*. To show the amount of storage-savings obtained, we show the initial storage requirement of few HTML files, when neither of our algorithms have been run. From Fig. 8, it is evident that both *ContentExtractor* and the Shingling algorithm provide substantial (and almost similar) amount of savings in caching size.

Precision and recall values using *Shingling* algorithm are very very close (from Table 6) to our those from *ContentExtractor* algorithm as we see from the Web pages we have analysed. Thus, the main advantage of *ContentExtractor* is its time of execution. Due to less complex steps and easy feature-based characterization of individual blocks in *ContentExtractor*, the generation of informative blocks is very fast. The *Shingling* algorithm depends mainly on calculating the hash values of all $(N-W+1)$ possible token-IDs for N tokens and shingles set of window length W. The computation of hash values and the computation for the comparison involved in the resemblance equation

$$Resemblance(A_i, B_j) = \frac{SubtreeShingles(A_i) \cap SubtreeShingles(B_j)}{SubtreeShingles(A_i) \cup SubtreeShingles(B_j)} \quad (8)$$

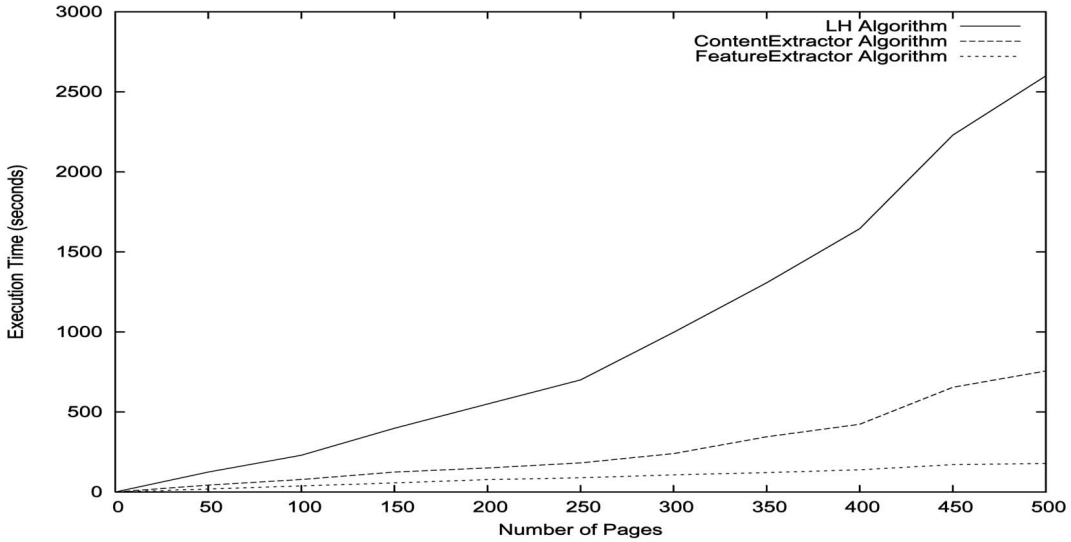

Fig. 7. Runtimes for the *LH*, *ContentExtractor*, and *FeatureExtractor* algorithms. The vertical axis represents the time of execution (in seconds) for a number of pages (plotted in the horizontal axis).

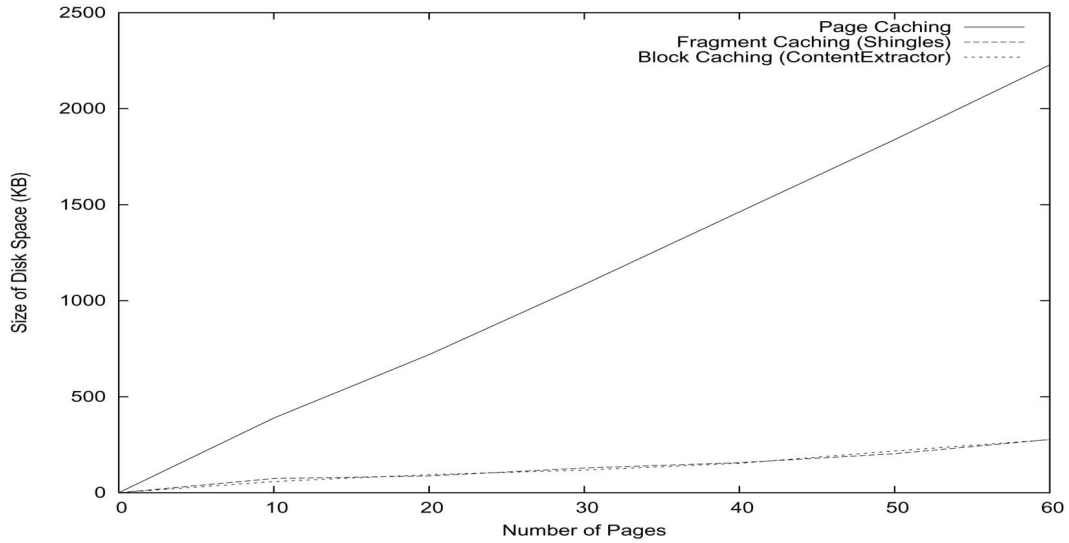

Fig. 8. Total storage requirement of Web-caches using *ContentExtractor* and Shingling algorithms.

are expensive. *ContentExtractor* does not remove any HTML tags and uses them for making the feature vector. This computation is relatively simple and inexpensive because the comparison/similarity is based on just a cosine calculation between two vectors. Therefore, *ContentExtractor* is much faster than the Shingling algorithm. Fig. 9 shows a comparison of runtimes taken by the Shingling algorithm and *ContentExtractor*. Clearly, the Shingling algorithm does not scale very well and, thus, the times for the larger number of Web pages is not reported.

7 Algorithm: L-Extractor

Song et al. [25] used VIPS to perform page segmentation and then used an SVM to identify primary content blocks in a Web page. VIPS is an expensive page segmentation algorithm. However, we hypothesized that an SVM can be very useful to identify primary content blocks. To prove our hypothesis, we applied our $GetBlockSet$ algorithm to 250 Web pages. In the next step, we created the feature-vectors for these blocks using HTML tags as described above. This set includes all HTML tags except those that are

TABLE 5
A Property-Wise Comparison Table for *Shingling* Algorithm and *ContentExtractor*

Property	Shingling Algorithm	ContentExtractor
Atomic Structure	AF Tree Node	*Block*
Basis of Similarity	Shared Fragment Measure	$IBDF$ *Measure*
Similarity Calculation	*ShareFactor*	$IBDF$ *value*
Similarity Threshold	*ShareFactor*	θ^{-1}
Matching	*MinMatchFactor*	ϵ
Atomic Property	*Complex and expensive measurement of* $(N - W + 1)$ *node-ids.*	*Simple HTML tag feature, which is perfect for measuring similarity. Inexpensive.*
Precision and Recall	**Similar to ContentExtractor**	**Similar to Shingling**
Speed	*Slow*	**Much faster**
Disk Space Requirements (Figure 4)	**Similar to ContentExtractor**	**Similar to Shingling**

TABLE 6
Block Level Precision and Recall Values from *Shingling* Algorithm and *ContentExtractor* Algorithm for 50 Web Pages

Site	b-Prec of Shingling	b-Recall of Shingling	b-F-measure of Shingling	b-Prec of CE	b-Recall of CE	b-F-measure of CE
ABC	0.92	0.99	**0.953**	0.915	0.99	**0.95**
BB	0.98	1.00	**0.989**	0.997	1.00	**0.998**
BBC	0.971	1.00	**0.985**	0.968	1.00	**0.983**
CBS	0.97	0.99	**0.979**	0.972	1.00	**0.985**
CNN	0.97	1.00	**0.984**	0.977	1.00	**0.988**

The second, third, and fourth columns are from Shingling *algorithm, the fifth, sixth, and the seventh columns are from* ContentExtractor.

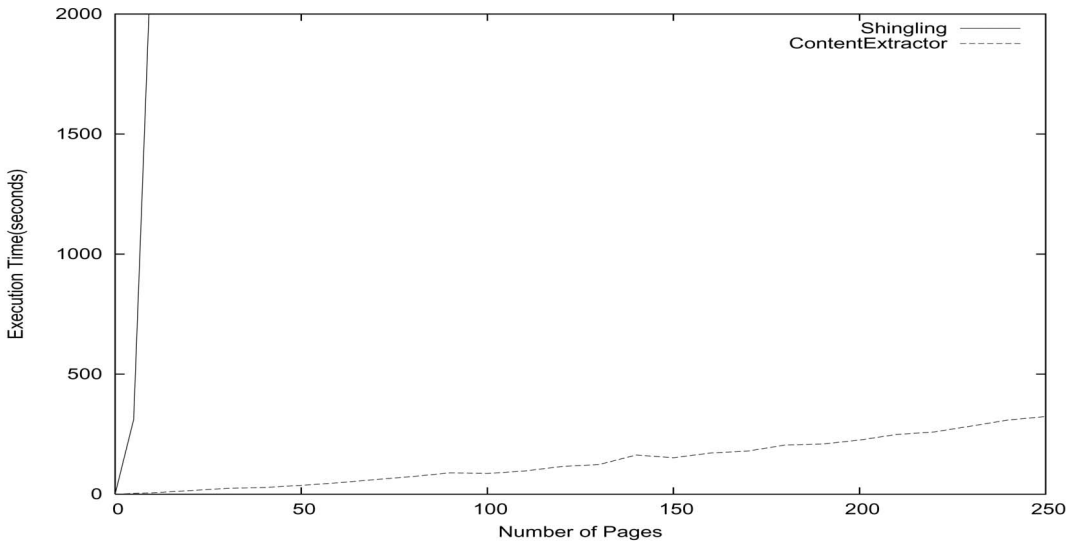

Fig. 9. Total execution time taken for *ContentExtractor* and the Singling-based algorithm.

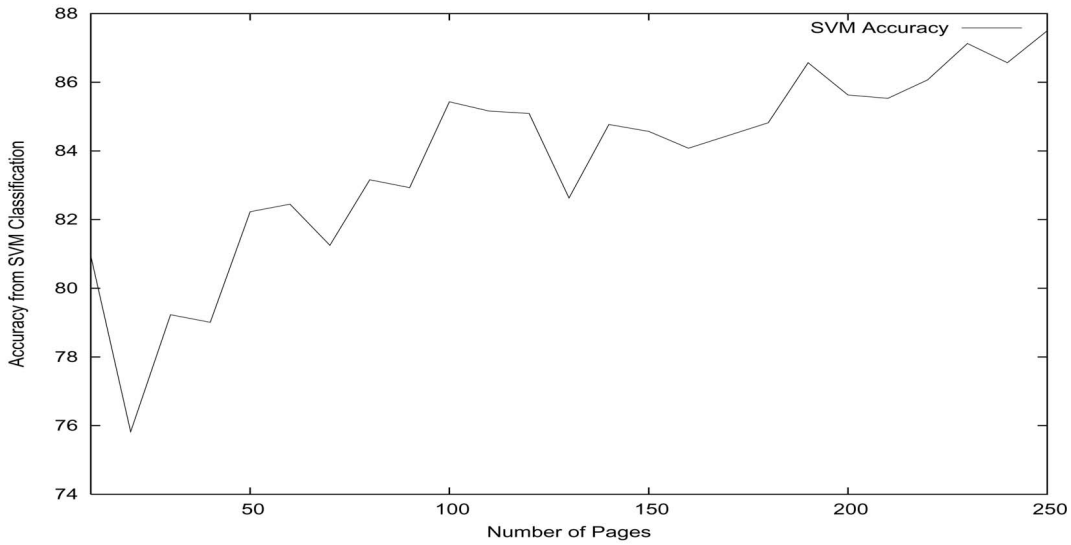

Fig. 10. Accuracy obtained by using our blocking algorithm with an SVM-based classifier based on block features to identify primary content blocks.

included in the partitioning-list of tags and including text feature. We then ran our Support Vector Learning classifier [11] (We used a linear Kernel for the Perl SV classifier with cost and weight values of two-class C-SVC algorithm both set to 1 with 5-fold cross validation following Chang and Lin [13]). Fig. 10 shows the accuracy of finding the informative blocks over increasing numbers of Web pages. From this study, we can claim that our block-partitioning algorithm combined with an SVM works with high efficiency.

8 Conclusions and Future Work

We devised simple, yet powerful, and modular algorithms, to identify primary content blocks from Web pages. Our algorithms outperformed the LH algorithm significantly, in b-precision as well as runtime, without the use of any complex learning technique. The *FeatureExtractor* algorithm, provided a feature, can identify the primary content block with respect to that feature. The *ContentExtractor* algorithm detects redundant blocks based on the occurrence of the same block across multiple Web pages. The algorithms, thereby, reduce the storage requirements, make indices smaller, and result in faster and more effective searches. Though the savings in filesize and the precision and recall values from "Shingling Algorithm" is as good as from *ContentExtractor*, *ContentExtractor* outperforms the "Shingling Algorithm" by a high margin in runtime. We intend to deploy our algorithms as a part of a system that crawls Web pages, and extracts primary content blocks from it. In the next step, we will look at the primary content and identify heuristic algorithms to identify the semantics of the content to generate markup. The storage requirement for indices, the efficiency of the markup algorithms, and the relevancy measures of documents with respect to keywords in queries should also improve (as we have shown briefly by caching size benefit) since now only the relevant parts of the documents are considered.

REFERENCES

[1] J.L. Ambite, N. Ashish, G. Barish, C.A. Knoblock, S. Minton, P.J. Modi, I. Muslea, A. Philpot, and S. Tejada, "Ariadne: A System for Constructing Mediators for Internet Sources," *Proc. SIGMOD,* pp. 561-563, 1998.

[2] P. Atzeni, G. Mecca, and P. Merialdo, "Semistructured and Structured Data in the Web: Going Back and Forth," *Proc. Workshop Management of Semistructured Data,* 1997.

[3] Z. Bar-Yossef and S. Rajagopalan, "Template Detection via Data Mining and Its Applications," *Proc. WWW 2002,* pp. 580-591, 2002.

[4] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Block Based Web Search," *Proc. 27th Ann. Int'l ACM SIGIR Conf.,* pp. 456-463, 2004.

[5] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinon, J. Ullman, and J. Widom, "The Tsimmis Project: Integration of Heterogeneous Information Sources," *Proc. 10th Meeting Information Processing Soc. of Japan,* pp. 7-18, 1994.

[6] B. Chidlovskii, J. Ragetli, and M. de Rijke, "Wrapper Generation via Grammar Induction," *Proc. Machine Learning: ECML 2000, 11th European Conf. Machine Learning,* pp. 96-108, 2000.

[7] W.W. Cohen, "A Web-Based Information System that Reasons with Structured Collections of Text," *Proc. Second Int'l Conf. Autonomous Agents (Agents '98),* K.P. Sycara and M. Wooldridge, eds., pp. 400-407, 1998.

[8] World Wide Web Consortium, World Wide Web Consortium Hypertext Markup Language.

[9] V. Crescenzi, G. Mecca, and P. Merialdo, "Roadrunner: Towards Automatic Data Extraction from Large Web Sites," *Proc. 27th Int'l Conf. Very Large Data Bases,* pp. 109-118, 2001.

[10] S. Debnath, P. Mitra, and C.L. Giles, "Automatic Extraction of Informative Blocks from Webpages," *Proc. Special Track on Web Technologies and Applications in the ACM Symp. Applied Computing,* pp. 1722-1726, 2005.

[11] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Verlag, 2003.

[12] C. Hsu, "Initial Results on Wrapping Semistructured Web Pages with Finite-State Transducers and Contextual Rules," *Proc. AAAI-98 Workshop AI and Information Integration,* pp. 66-73, 1998.

[13] C.W. Hsu, C.C. Chang, and C.J. Lin, "A Practical Guide to Support Vector Classification," *A Library of Support Vector Machines,* 2003.

[14] T. Kirk, A.Y. Levy, Y. Sagiv, and D. Srivastava, "The Information Manifold," *Proc. AAAI Spring Symp. Information Gathering from Heterogeneous Distributed Environments,* pp. 85-91, 1995.

[15] N. Kushmerick, "Wrapper Induction: Efficiency and Expressiveness," *Artificial Intelligence,* vol. 118, nos. 1-2, pp. 15-68, 2000.

[16] N. Kushmerick, D.S. Weld, and R.B. Doorenbos, "Wrapper Induction for Information Extraction," *Proc. Int'l Joint Conf. Artificial Intelligence (IJCAI),* pp. 729-737, 1997.

[17] A.Y. Levy, D. Srivastava, and T. Kirk, "Data Model and Query Evaluation in Global Information Systems," *J. Intelligent Information Systems,* special issue on networked information discovery and retrieval, vol. 5, no. 2, pp. 121-143, 1995.

[18] S.-H. Lin and J.-M. Ho, "Discovering Informative Content Blocks from Web Documents," *Proc. Eighth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 588-593, 2002.

[19] B. Liu, K. Zhao, and L. Yi, "Eliminating Noisy Information in Web Pages for Data Mining," *Proc. Ninth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 296-305, 2003.

[20] I. Muslea, S. Minton, and C.A. Knoblock, "Hierarchical Wrapper Induction for Semistructured Information Sources," *Autonomous Agents and Multi-Agent Systems,* vol. 4, nos. 1-2, pp. 93-114, 2001.

[21] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis, "Automatic Detection of Fragments in Dynamically Generated Web Pages," *Proc. 13th World Wide Web Conf.,* pp. 443-454, 2004.

[22] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis, "Automatic Fragment Detection in Dynamical Web Pages and Its Impact on Caching," *IEEE Trans. Knowledge and Data Eng.,* vol. 17, no. 5, pp. 1-16, May 2005.

[23] C.J. Van Rijsbergen, *Information Retrieval.* Butterworth-Heinemann, 1979.

[24] G. Salton, *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968

[25] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma, "Learning Block Importance Models for Web Pages," *Proc. 13th World Wide Web Conf.,* pp. 203-211, 2004.

[26] L. Yi, B. Liu, and X. Li, "Visualizing Web Site Comparisons," *Proc. 11th Int'l Conf. World Wide Web,* pp. 693-703, 2002.

Sandip Debnath is a PhD candidate at Penn State University. Prior to that, he received the BS degree in electronics and teleCommunications, MTech degree in computer science, and the MS degree in computer science. His research areas include search engines, Web crawling, data mining, machine learning, ontologies, and artificial intelligence. He has more than 20 conference and journal publications. He is also involved in design, development, and maintenance of SmealSearch at Penn State University. He has also been session chair at ICML and referee in several conferences and journals including UAI, EC, KDD, and the *IEEE Transactions on Knowledge and Data Engineering*.

Prasenjit Mitra received the PhD degree from Stanford University in 2004. Currently, he is an assistant professor at the Pennsylvania State University. His main research interests are in database systems, digital libraries, the semantic Web and data security. He has been a senior member of the technical staff at Oracle Corporation, Narus Inc., and DBWizards. He has served on the program committee of a workshop and the 2005 IEEE International Conference on Services Computing. He has also been a referee for several workshops, conferences, and journals including *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, Information Systems, DKE, and *Journal of Universal Computer Science*.

Nirmal Pal is the executive director of the eBusiness Research Center in the Smeal College of Business at the Pennsylvania State University. He is an expert on eBusiness strategy, governance, assessment, and evaluation, as well as IT strategy and planning. He previously served as the director of IBM Global Services Consulting Group, where he has been a member of the management team since its inception in 1991.

C. Lee Giles left NEC Research Institute, now NEC Labs, in 2000 to become the David Reese Professor in the School of Information Sciences and Technology. He is also a professor of computer science and engineering, professor of supply chain and information systems, and associate director of research in the eBusiness Research Center at the Pennsylvania State University, University Park. He has been associated with Princeton University, the University of Pennsylvania, Columbia University, the University of Pisa, and the University of Maryland; and has taught at all of the above.