JUICER – A DATA MINING APPROACH TO INFORMATION EXTRACTION FROM THE WWW

Irmina MASŁOWSKA*, Dawid WEISS*

Abstract. We present a novel approach to automatic text mining on the World Wide Web. Considering the fact that the enormously dynamic growth of the WWW results in a need for new, more powerful information extraction tools we designed and implemented a system, which adapts techniques originally introduced in the field of data mining. We believe that similar systems, which usually base on machine learning or natural language processing methods, can prove to be ineffective when dealing with the very large numbers of hypertext documents of different structure and subject. Moreover, such systems tend to treat HTML documents as plain texts not taking into account the additional information contained in their markup tags.

1 Introduction

The World Wide Web has indisputably become one of the largest and most diverse sources of information on virtually any subject interesting to humans. At the same time it has made it impossible to handle the information manually. The problem, known as the *information overload* on the WWW constitutes still a new challenge for researchers in natural language processing, machine learning, and related AI disciplines, also being a subject of many commercial applications [3]. Since the majority of information is embedded in hypertext documents, which are understandable only to humans, the issue of creating effective software tools for processing it automatically remains an open problem.

* Institute of Computing Science, Poznań University of Technology, Piotrowo 3A, 60-965 Poznań, Poland

Although in recent years a considerable amount of work has been done in applying machine learning and natural language processing algorithms to a variety of problems in classifying and extracting information from the text [2][4][6][8][9][10][13], many of the proposed methods lack scalability and generality, i.e. they do not take into account the overwhelming amounts of documents often needed to be processed and the cosmopolitan characteristic of the WWW, which results in the fact that the language in use does not necessarily have to be English.

In this paper we present a system designed to extract relevant, human-understandable information from large collections of hypertext documents. Our system, called Juicer, operates on documents forming thematic domains, applies a new universal method to finding representative keywords and assigning weights to them, finds potentially interesting associations between keywords, and presents them to the user making it possible to automatically discover interesting information without even skimming the documents or to identify and access only relevant documents in a fast and effective way.

To cope with the enormously large collections of documents possible to obtain from the WWW we propose a novel approach, which adapts the *Apriori* algorithm [1] originally designed to induce association rules from very large databases (VLDB).

In this paper, we first describe the functionality of Juicer. Next, we present proposed methods and algorithms comparing them to the commonly applied ones. Finally we give our conclusions on the system performance and the possible future development of methods used in Juicer.

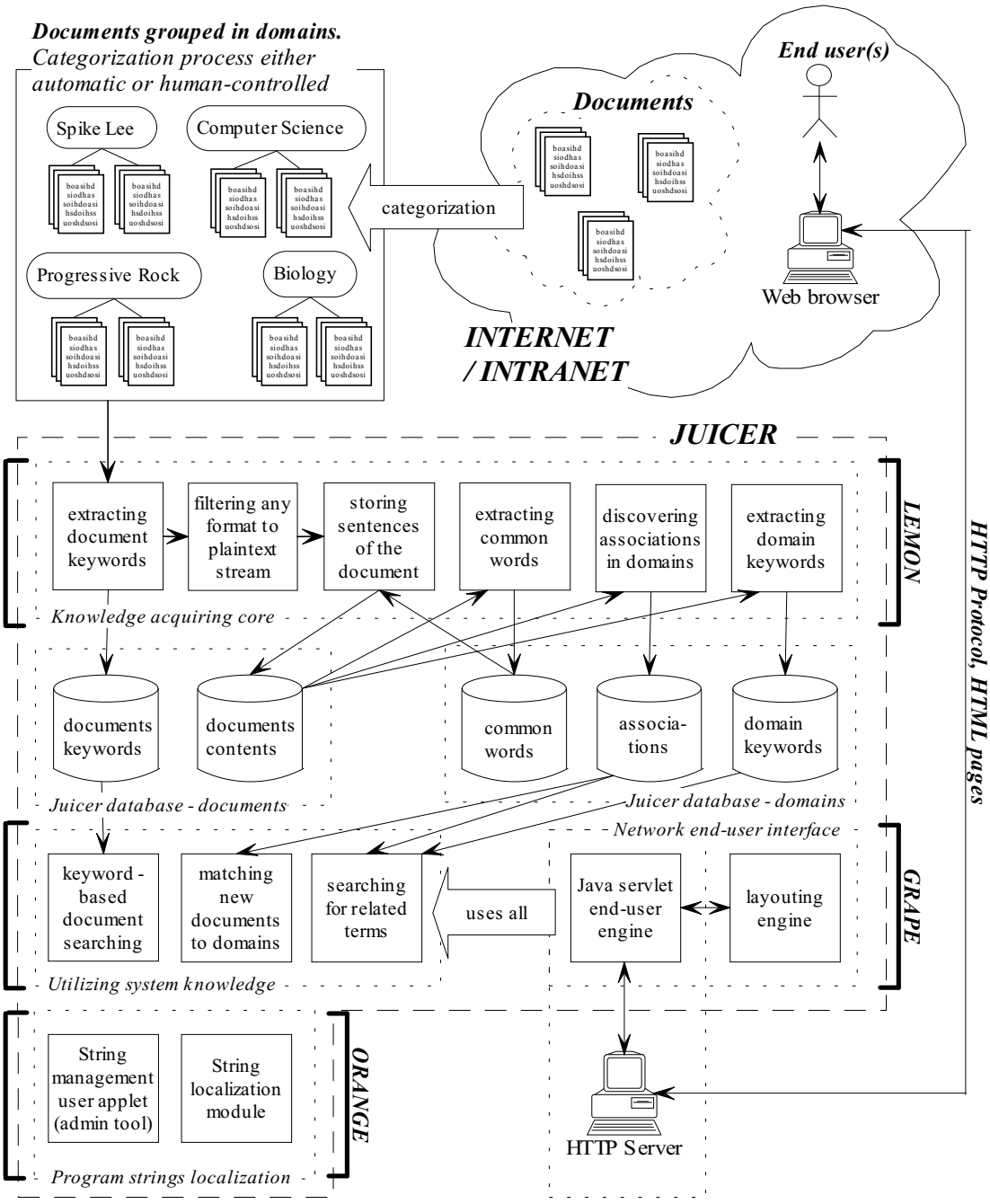
2 The Juicer system

Juicer has been developed as a B.Sc. thesis in the Institute of Computing Science, Poznań University of Technology [7].

The system is written entirely in Java, uses Oracle database and consists of two main software modules: Lemon and Grape. Lemon is an administrative tool – a stand-alone Java application. Grape is the user front-end implemented as Java servlets, accessible through any WWW browser like Netscape Navigator or Microsoft Internet Explorer. An additional application called Orange has been written, which can be used to create localized interfaces for Lemon and Grape.

An overview of Juicer's structure and functionality is given in Fig.1. Its three software modules (i.e.: Lemon, Orange, and Grape) are distinguished by brackets enclosing their functional sub-modules. The arrows that can be seen between particular system sub-modules and databases represent the data flow in the system.

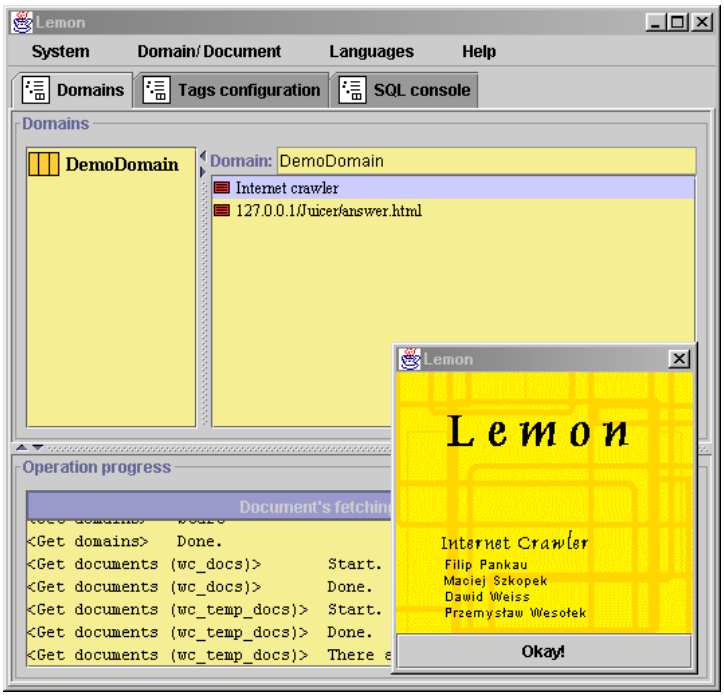
Fig.1. The overview of Juicer



Lemon wraps the core of the system – keyword and associations discovering engine. It has numerous features such as web-spider, SQL console with output formatting (for advanced system administrators) and, most of all, graphical access to many system variables and data stored in the database.

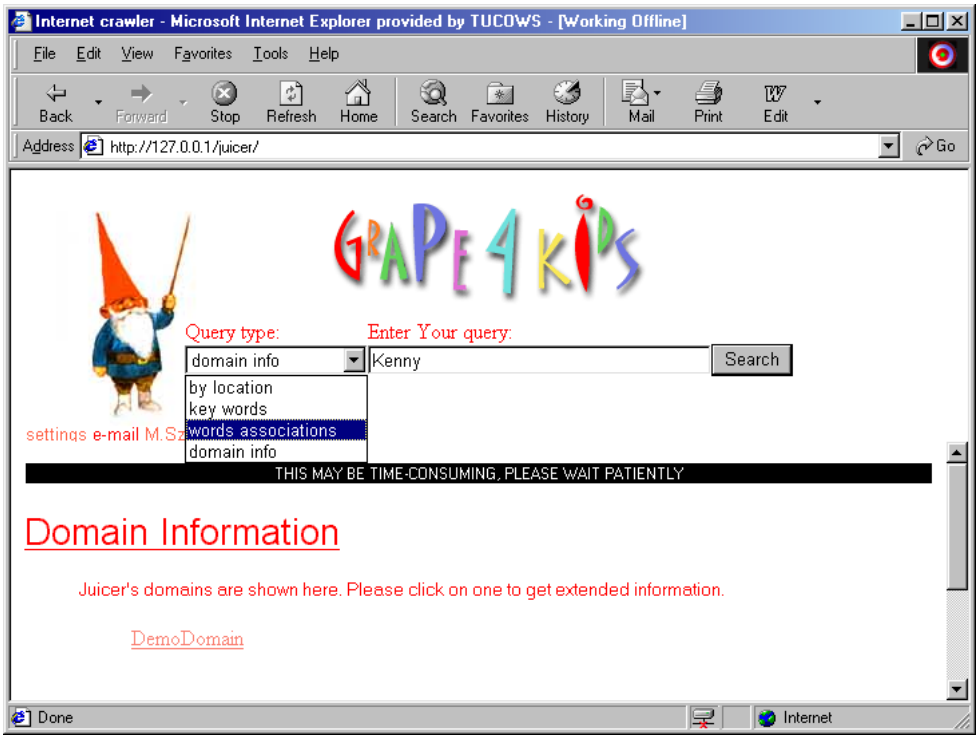
Grape is a set of HTML pages combined with servlets. It allows users to make use of four system services: displaying domains and their associations, searching for related terms, keyword-lookup of documents, and assigning new documents to existing domains.

Fig.2. Lemon – administrative application of Juicer



In Fig.3 one can notice an example of the fully customizable user interface, which is a unique feature in web-searching systems.

Fig.3. Grape – user front-end of Juicer



The *"domain info"* service displays specific information about domains stored in the system. One can see the domain's strongest associations, its keywords, the number of documents constituting the domain, etc.

The *"keywords"* service is exactly what other search engines like Altavista or Google offer – it lists documents that somehow refer to the list of keywords provided by the user. In this functional area Juicer provides the same options as any other search system.

The two remaining services make use of the association rules* – a feature which distinguishes Juicer among other information extraction systems. *"Words associations"* is a simple service that allows navigating through associations found on the data stored in the system. For instance for the word "right" in one of the domains we would get associations like "right→coordinate", "right→left" or "right→corner".

"By location" is a simple assistance service that lets the user match an HTML document, pointed by its URL address, to one of system domains. Such help is useful if we have a document of which we are not sure what it contains and we want to find out without reading it thoroughly.

3 Information extraction in Juicer

3.1 Extracting keywords from documents

Keyword extraction is what actually starts the process of discovering associations in Juicer. A good keyword is a word that is in some way *representative* of the text it was picked from. In Juicer's case it is crucial to pick out very representative and interesting words because a user wants the discovered word associations to have both of those features. If we didn't pay enough attention to this phase, we would be apt to choose words that are frequent in the text, but are useless for discriminating of particular domains the text comes from (words like "the", "that" or "this" – we refer to them as *common words* or *junk words*).

We decided that all of word associations would include relations between *domain* keywords (for the definition of a domain, refer to section 3.2). An association between occasional words cannot have a high support value because the support of an association is always smaller than all supports of words constructing it and if those words had high support value, they wouldn't be occasional! Therefore the association would be removed later on anyway because of its support value being below the support threshold (refer to 3.4).

The most common practice applied to find out which words are characteristic to the document, is to use some frequency-based method [11][12]. Sometimes it is

* In fact, we search for frequent sets (see chapter 3.4.3.) not standard association rules.

enough to calculate how many times a given word occurs in the document and divide that figure by the total number of words in it. Simple as it is, that method has many drawbacks. The ratio depends strongly on the length of the document for instance. Besides in HTML documents there is another piece of information that the above simple method doesn't count in and which seemed very tempting to use – HTML markup tags. Those tags describe the structure of a document (for example the <TITLE> tag indicates the title of the document). Many of HTML elements carry additional information about words they affect, like boldface or underline usually mean emphasis put on those words (one can deduce they must be more important than the other ones, so perhaps they are keywords).

In this section we describe the methods and formulas used to assign words their *importance* value (words having importance higher than some threshold are considered keywords).

3.1.1 Calculating words importance for an HTML document

The *keyword importance*, which we define later in this paragraph, is the very baseline for further discovering of association rules. Its calculation is not straightforward and it consists three phases.

First we calculate the weight of each single word which is a derivative of the word's context. The context we are talking about is a result of HTML tags' scope. Initially all words in a document have the same weight value: $w_0=1$. That value is then modified using so-called *rewards* and *penalties*, depending on the tags the word is within. Here is a short example of what we understand under those terms. Let's assume the following sentence comes from an HTML document:

You should take part in our **very special <u>promotion</u>**!

As one can see, part of the sentence is displayed in boldface and one word is underlined. The source HTML code for that sentence could look as shown below:

```
You should take part in our <B>very special <U>promotion</U></B>!
```

The initial weight w_0 equals 1 for each word in the sentence. The <B> tag scope will be rewarded with some higher value, e.g. 5 and the <U> tag will be given a penalty of –3 (just for the need of the example, although it is not necessarily adequate to how underlined text is usually perceived). To calculate words' weights we need one more element – the *tag influence aggregation function*. That function can be defined in a number of ways (for details, refer to [7]), we used the FIFO definition in which the sequence of tags in the file imposes the order of penalties/rewards used to modify words' weights. In our example the first tag to affect those weights would be <B> because it comes before <U>.

The FIFO function aggregating rewards and penalties for a word in context of n tags $z_1, z_2, \ldots, z_n$ is defined below. For each tag z_i a penalty or reward value of t_i is defined. That value is in range $[-w_{min}; w_{max}]$, if it's less than zero it means penalty, reward otherwise.

(1)

```
w=w0
for i=1 to n do
   if (ti>=0) then
         // ti is a reward
         w = w + ti*(wmax-w)/(wmax-wmin)
   else
         // ti is a penalty
         w = w - |ti|*(w-wmin)/(wmax-wmin)
```

Now we can calculate weights for three sample words in our example – "should", "very" and "promotion".

w_{max}= 10, w_{min}=-5, w_0=1, reward(B)=5, reward(U)=-3
"SHOULD": $w=w_0=1$
"VERY": $w_0=1$, $w=w_1=1+5*(10-1)/(10-(-5))=4$
"PROMOTION": $w_0=1$, $w_1=1+5*9/15=4$, $w=w_2=4-|-3|*9/15=2,6$

The context weighting and the FIFO procedure we described seem very suitable for modeling user perception of the displayed text. Through the customized definition of rewards and penalties for tags, the system user can achieve very simple filter through which he or she can put some more emphasis on design styles that he or she finds more important.

In some cases however assigning weights to words based on HTML tags can be misleading – let's imagine that an author forgets to close a tag. What happens then is that the tag rewards majority of words, which in fact would be unfair or meaningless (because giving a reward to all words is the same as not giving it at all). Also, it very often happens that authors overuse some tags, e.g. to make the document visually more attractive. In such case it would be unfair to reward words in such a document with high weights when it is a matter of misinterpretation of styles in HTML...

In Juicer we tried to avoid situations as mentioned above by introducing the *devaluation of commonly used tags*. It's the second phase of the process of extracting keywords from the document.

The diminishing influence of a particular, very commonly used tag, is achieved by decreasing its reward or penalty. Knowing what percentage of all words in the document the tag affects, we can recalculate its reward or penalty using the formula given below:

$$actual\_reward = initial\_reward * \sqrt{1 - freq} \tag{2}$$

Where *freq* is the ratio of all words within that tag scope to the total number of words in the document.

Naturally the actual reward or penalty value has to be known before weighting words. We swapped the two phases because it seemed more reasonable to first show how we weight words and then how we modify that process.

The last phase of keyword extraction aggregates the weights of identical words to one value, which we called *word importance*. For a thorough explanation of the used formula, please refer to [12].

$$importance(S_i) = \frac{\sum_{s_j \in S_i} w(s_j)}{\sqrt{|S_i|}} \tag{3}$$

Where S_i is the set of identical words s_j in the document, $w(s_j)$ is the weight of a single word s_j from that set. In other words, importance is a sum of weights for some particular word divided by the square root of the number of occurrences that word has in the document.

Eventually we sort and normalize the result. Words having the highest ranks are supposed to be keywords (some of them can still be common terms, which will be removed later on after common words extraction phase).

3.2 Extracting keywords from domains

All documents stored in Juicer's database are grouped in logical sets called *domains*. More formally, a domain may be specified as a set of documents, which relate to the same, or very close, subject. Also other definitions may be given, for instance a domain is a set of documents with a vocabulary from the same *field* of science, culture etc.

It seems impossible to give any strict definition of a domain. Examples of them such as: cars, computer science, biology or heavy industry are all correct but as well we could split computer science to programming languages, hardware and networking and they would still remain a perfect example of domains. Our intention was to leave that part to system administrator, who can ask other people (i.e. system users, experts) for help. There are some hints about good domains though. A domain should contain at least twenty mid-sized documents, so that the process of extracting keywords is reliable and the associations found in such a domain are representative. Also it should not be too big since "oversized" domains tend to introduce a lot of unimportant information (noise), which drastically reduces system performance and does not improve results at all. A good piece of advice is to choose documents, which are very *specific* to a domain.

The question arises if it's worth throwing all that extra work on administrator's shoulders and what are the benefits of having domains in the system. We will try to point out a few.

An important advantage of keeping the documents grouped neatly in domains is that it makes it possible to use a very simple yet effective junk-words discovering algorithm that is based on domains. There is no need of installing additional software or dictionaries for getting rid of common words like pronouns for instance. Moreover it

is possible to get rid of common words from virtually *any* language that has relatively simple inflexion. We discuss this subject in the next subsection.

Another asset of having domains is that after the knowledge base is created one can use the system to assign a document to some domain, without even reading it. The assignment is based on associations discovered in domains, as it will be explained in section 3.5.2.

3.2.1 Preparing the list of keywords for a domain

Having explained the important role of an effective keyword extraction for a document, we can proceed to the description of the corresponding process for the whole domain.

The simplest approach would be to make use of known keywords calculated for separate documents. That indeed would be very simple, however it is worth noticing that a word which is representative of a document is not necessarily representative of the document's domain. Thus, instead of introducing a method for aggregating keywords from separate documents we decided to use a probabilistic technique described below.

The general idea of the probabilistic keyword extraction is to treat every domain as a whole, i.e. one big document. For each word s_k in domain D_i we can calculate the probability that some randomly drawn word from D_i will be s_k.

$$P_{total}(s_k) = \sum_{d_{ij} \in D_i} P(s_k \left| d_{ij} \right.) * P(d_{ij}) \tag{4}$$

Where s_k is some particular word from domain D_i, d_{ij} is i-th document from that domain. $P(s_k|d_{ij})$ is the probability of drawing word s_k from document d_{ij}. $P(d_{ij})$ is the probability of choosing document d_{ij} from domain D_i.

Implementation of that formula is even less sophisticated – we divide the sum of weights (see formula 1) of all occurrences of a particular word by the sum of weights for all words in the domain. Simple as that, the resulting probability is representative and normalized for the whole domain.

The terms having highest probability are apt to become keywords if one more condition is fulfilled. Let's assume a word occurs only in a very small fraction of domain's documents, but it has very high importance in them. Such a word is indisputably not a keyword for the whole domain. To remove such "superwords", as we call them, from the keyword candidates, we added the mentioned condition that a keyword has to occur in some minimum number of documents (more specifically it must be a *frequent* term in the domain – see formula 6).

3.3 Discovering common words (junk words)

Common words are those, which build the sentence, create its structure but do not carry any information or meaning important to the domain field. For example, sentences "Amiga was one of the best computers I've had" and "The finest computer I've worked on was an Amiga" carry pretty much the same information but use a different vocabulary to achieve the goal of informing about author's favorite machine. The conclusion is that in most cases it is unnecessary to waste processing time on some words. We could get rid of them before applying some more advanced algorithms and thus speed up the system.

In the case of Juicer removing common words from the input data was a very important issue because of resource constraints (memory and time). For systems like ours the process of filtering the common words out makes the results better and processing faster. The amount of data to be processed by more advanced algorithms decreases over twice, so the gain achieved from filtering documents should be clear. In some applications, however, that approach is not possible. Like in language-specific syntax decomposition, we cannot remove anything from the input text stream, because what we get after filtering may look like babbling.

3.3.1 Automatic common words discovering in Juicer

A typical approach to the problem of filtering out the common words is to use their predefined, handcrafted dictionary [8]. An automation of the dictionary creation process has been introduced, which searches the documents for the most frequent words and treats them as junk words [8][9]. The latter method has an obvious disadvantage that it can eliminate the most representative keywords if they frequently occur in a document. Keeping that in mind but still wanting to discover common words automatically we introduced an alternative method.

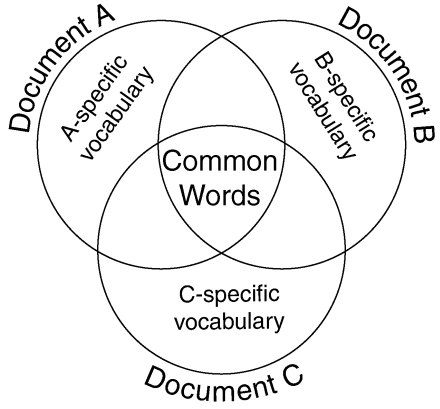
We started from analyzing several documents from different domains, trying to find the shared attribute of common words, marking them out from other terms. We finally got to an idea that they are indeed *common*, ordinary, so all documents, regardless of their contents, should contain at least a subset of those frequently used terms we're looking for. The graphical representation of this observation is presented on the picture below. All documents *share* some vocabulary that is used to structure the message into sentences (see Fig.4).

Thus the simplest algorithm for discovering common words is to take an intersection of documents' vocabulary. However several important issues still have to be solved.

The first threat is that we may take an intersection of several documents concerning the same subject. In such case the common words would include not only frequent terms, but also keywords, which, as it was explained in section 3.1, are

frequent too! To avoid this, we have to take the intersection of the whole *domains*. According to their definition, domains differ in subject and hence they must have different keywords. Therefore the intersection of them would include only common words, without those that are frequent, but possibly important to the domain[*].

Fig.4. Different documents usually share a subset of their vocabulary



Another danger is that some terms may "leak out" from the intersection set. If we used a strict intersection operator, every common word would have to exist in every domain. While this is true in most cases, it might happen that a new domain was so small that it wouldn't include some words, thus restricting them from becoming common terms. Democratic-like solution proposed by us is to demand for a common word to be present in at least some percentage of all domains.

The final set of common words is constructed using formula given below.

$$C = \left\{ s \in \bigcup_{i=1}^{n} P_i : \frac{\left| \{ P_k : s \in P_k \} \right|}{n} \geq t_{common} \right\} \tag{5}$$

Where the n is the number of domains in the system, s denotes a unique word and t_{common} is the user threshold for common words. The above formula represents the fact that a word in order to be common, has to be frequent in at least t_{common} percent of all domains (a word is frequent in D_i if it belongs to its set P_i).

The frequent set P_i for domain D_i contains all those terms that are present in at least $t_{frequent}$ percent of all documents in that domain. This condition is needed to prevent a situation in which a word, which is not frequent inside any domain, but present in all of them, becomes a common word in the system. The P_i sets are built in a fashion very similar to the one constructing the set C:

$$P_i = \left\{ \left(m = \mid D_i \mid, s \in \left\{ \bigcup_{j=1}^{m} d_j \in D_i \right\} \right) : \frac{\left| \{ d_k \in D_i : s \in d_k \} \right|}{m} \geq t_{frequent} \right\} \tag{6}$$

[*] Assuming that domains are loosely related in subject and vocabulary.

Where m is the number of documents in domain D_i, $t_{frequent}$ is the user threshold for frequent terms.

We claim that the proposed method for common words discovering is very simple yet fast and accurate. It prevents keywords from being included in the common words set without using any external dictionary, which is its main advantage. Also it is to some extent *language independent* – the algorithm remains the same for most languages, with some special inflection modifications if they are needed.

A major drawback of the method is the requirement for unrelated domains in the system. This constraint is hard to verify and requires "sensitivity" of the system administrator when selecting domains for intersecting.

A more exhaustive description of the method covering all the performance and quality factors, as well as implementation hints used in Juicer may be found in [7].

3.4 Discovering associations between words

3.4.1 Association rules and their role in data mining

In this section we present the baseline knowledge concerning association rules needed to understand further contents of this article; a reader acquainted with the subject may want to skip this point.

Let us begin with a mandatory peanut-beer sales example. In a sales department of a food store called Tey-Mart, located in Poznań, all transactions are kept in a database. A record of that database consists of unique transaction number (TID) and Boolean "true" values in columns referencing to items bought in that transaction. A partial printout of the database has been presented to the sales manager for analysis:

Tab.1. A piece of Tey-mart sales printout

TID	*items*			
	cheese	beer	peanuts	soda
1	✓			
2		✓	✓	
3	✓	✓	✓	
4	✓	✓		✓
5	✓	✓	✓	

The sales manager should notice that cheese and beer are the most commonly bought items (so maybe there is a need to order more of them). If the manager is doing a painstaking analysis, he or she can point out the fact that when a customer buys

peanuts, it will with 100% certainty go along with beer. Such information is very valuable since the store management can locate beer right next to peanuts or, on the contrary, put beer on the other side of the store so that customer has to go all the way down to it, perhaps buying something extra on his way.

The above example is humorous, but not fictional; actually most stores, and business institutions in general, support their analysis with such *associations* between data kept in their databases. Going back to our illustration of the problem, the conclusion was "customer for 100 percent buys beer when he or she buys peanuts". Such result obtained from some data set is called an *association rule* and can be written symbolically as $\{peanuts\} \rightarrow \{beer\}$. Formal definitions of an association, its *support* and *confidence* are quoted below from Agrawal's and Srikant's work [1]).

Def.1. Definition of an association rule, its support and confidence

Let $\mathcal{I}=\{i_1,i_2,...,i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction $\mathcal{T}$ is a set of items such that $\mathcal{T} \subseteq \mathcal{I}$. Associated with each transaction is a unique identifier, called its TID. We say that a transaction $\mathcal{T}$ contains $\mathcal{X}$, a set of some items in $\mathcal{I}$, if $\mathcal{X} \subseteq \mathcal{T}$. An association rule is an implication of the form $\mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X} \subset \mathcal{I}$, $\mathcal{Y} \subset \mathcal{I}$ and $\mathcal{X} \cap \mathcal{Y} = \phi$. The rule $\mathcal{X} \rightarrow \mathcal{Y}$ holds in the transaction set $\mathcal{D}$ with confidence c *if* c *percent of transactions in $\mathcal{D}$ that contain $\mathcal{X}$ also contain $\mathcal{Y}$. The rule $\mathcal{X} \rightarrow \mathcal{Y}$ has support* s *in the transaction set $\mathcal{D}$ if* s *percent of transactions in $\mathcal{D}$ contain $\mathcal{X} \cup \mathcal{Y}$.*

In other words, an association can be seen as a relation between two sets of items. Its left side is $\mathcal{X}$, its right is $\mathcal{Y}$. In our example, the support of an association rule $\{peanuts,cheese\} \rightarrow \{beer\}$ is only 40% because among all 5 transactions, only transactions 3 and 5 contain all three items (peanuts, cheese and beer). The confidence of such association is 100%, because all transactions containing both cheese and peanuts also include beer. The "reversed" association rule: $\{beer\} \rightarrow \{peanuts, cheese\}$ must have the same support value, but it differs in confidence – this time it equals only 50% because among all four transactions containing beer, only two also include peanuts and cheese.

Support tells us how often an association occurs in the data set, while confidence expresses the certainty of encountering the right side of an association when the left side has been found in a transaction. The two parameters are without a doubt important, but it seems that support for an association is somehow "primary" when judging if it is interesting or not. For a rule with small support we can never tell whether its high confidence is just coincidental or not. In our example it would correspond to "soda", when we might say "when a customer buys soda he or she will for 100% buy cheese and beer", but it is obviously not so certain since we had only one customer to prove that rule (support is only 20%).

As we have already mentioned, association rules are used in data mining. Obviously no one is able to scan manually through databases counting hundreds of thousands tuples. The problem demanded an algorithmic solution and a great number of approaches were published, most of them based on the *Apriori* algorithm,

introduced by Agrawal and Srikant [1]. For more information about the theory of mining association rules, please refer to the literature. In the following sections we discuss the way we adapted the original Apriori algorithm, thus creating a heuristics capable of mining text documents for associations between words.

3.4.2 Goals of searching for associations between words in text files

The definition of an association rule (Def.1) is no longer applicable when it comes down to spoken or written language. Undoubtedly we don't have any transactions, TIDs, or items. The *association* between words is understood as a relation, either syntactical or semantical, between two or more terms. Let's assume the word to which we are looking for associations is "dog". A syntactical association could be *{dog}→{hot}*, because these words often occur next to each other, representing an edible (some people doubt) roll with a sausage inside. Semantical relation would be *{dog}→{animal}* because dogs are animal species. In Juicer we are more interested in the former kind of associations however there are some further possibilities for discovering the latter ones as well.

A question arises, why we search for something, which seems obvious and impractical. In fact there are at least three fields where we can utilize such knowledge.

First of all, browsing such associations is a very absorbing activity. In some way it is like browsing a thesaurus, but written by hundreds of thousands of regular people, assuming the source documents were taken from the Internet resources. In some cases it is even more powerful since it can relate new terms, especially technical, to other fields of science, hence creating a "mini dictionary". For example the term "Java" in most English dictionaries is described as an island of Indonesia, while the web-search would most certainly relate it to computer science and programming languages.

This leads directly to the utilization of word associations in web-searching engines of many kinds. Let's imagine a scenario in which a user types "dog" in his or hers browser window. The "improved" search engine would not only return the list of documents containing the "dog" word, but hence the word is very commonly used, it would try to give user a hint to narrow the search. So it could lookup the associations and find out that a "dog" is associated with (a domain!) animals, also is often used with a word "shepherd", but as well with "hot" and the whole "hot dog" phrase strongly relates to food. In such fashion the user could narrow or broaden his or hers query. It could be of great help to users less experienced with Internet and its enormous resources.

The last application of associations is more computer science oriented. We think they could, to some extent, replace the keyword-vectors hegemony for describing text documents and their sets. Unlike keywords, associations carry some more information than simple word occurrence statistics, so they provide better "fingerprints" of documents.

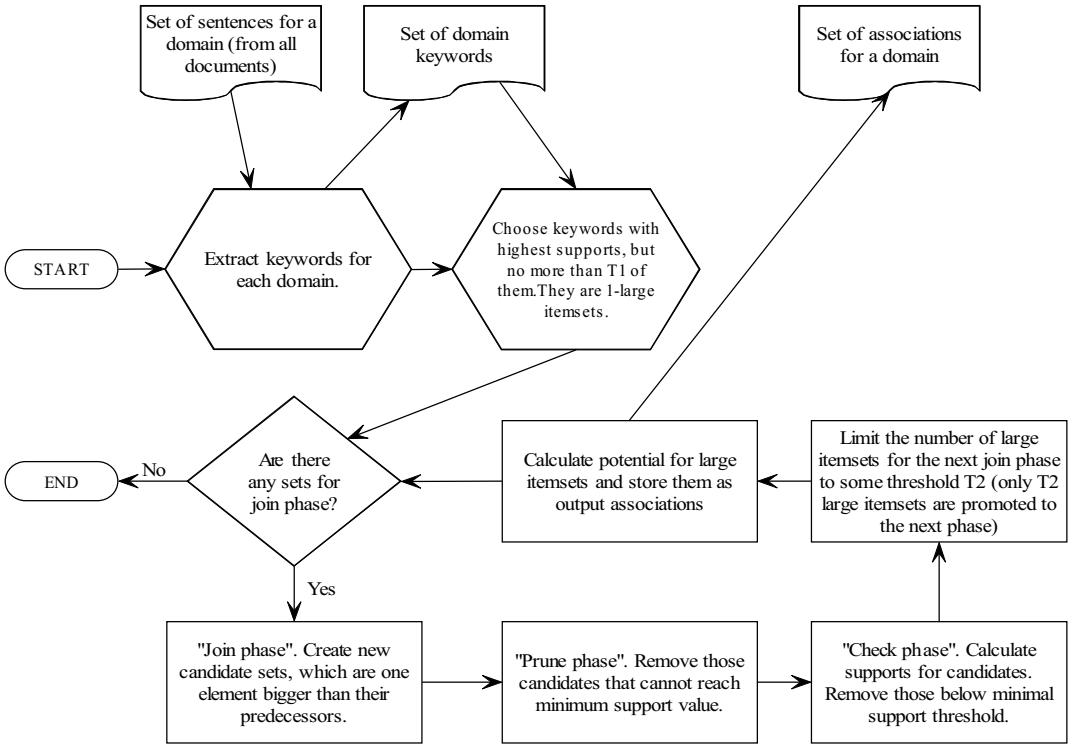
We introduced the three above functionalities in Juicer, and they will be explained further on in section 3.5.

3.4.3 Text-oriented Apriori algorithm

The need for modifying the original Apriori algorithm arose from several reasons. In this section we present those modifications together with reasons for which they were introduced.

Def.1 states clearly that all the data consist of elements, called items, which form set I. Because we are looking for relations between words, we will naturally treat words as items. However in the original algorithm we know how big that set is and what elements belong to it. When analyzing textual documents, it is not so clear – we can't distinguish any set of language terms from which we are to discover associations because it would be against the main goal of the system: we want to discover any possible association, not only those, about which we know what words they contain. Hence we need to allow all words to be items, the result is that set I remains unknown at the start of the algorithm.

Fig.5. Modified Apriori algorithm



Another problem concerns “transactions”. We decided to treat sentences like transactions – they contain words related to one another, and are to some point independent of other sentences. Of course it is a simplification of the problem, since text documents have their structure and composition from which we can conclude a great deal of information (for instance after reading an abstract we know what the whole document is about). Nevertheless it is still a big advancement comparing to the

most common Bayesian approach in which independence between all the words in a document is assumed [5].

There is one more important difference between sentences and database transactions that might have a distorting impact on results generated by the system. In a spoken or written language one might use several word many times, while in a transaction an item can occur only once. Also the meaning of a word can be determined by its position in a sentence. Language sentences are more like lists of words, while transactions can be seen as their sets. In Juicer we transform such lists to sets of words, so it does not matter how many times, or where in the sentence a word exists, it is seen only once. A more adequate transformation is a subject for further research.

Along with the problems of transforming the Apriori algorithm, we also had to face the difficulties concerning uncontrolled growth of the number of associations discovered at several steps in the algorithm, causing pitiful system performance due to memory swapping, etc.

The reason for the above-mentioned phenomenon is that in most languages there are words that are often used together. Such "friends" include collocations or phrasal verbs for instance, if the language in mind is English. Associations like these usually have high support (which means they are often used), but are rarely interesting to the user. The ones we would be interested in are multiword associations, usually of higher than average, but still relatively small support value. To discover them, we must set the support threshold below the level of "commonness" – unfortunately such decision results in a huge number of discovered associations from which it is very hard to pick the interesting ones.

One of the approaches used by us to limit the number of discovered "junk" associations is to get rid of those with common words or between common words. It helps the problem, but doesn't entirely solve it. In Juicer we defined one additional parameter describing frequent sets[*], which we called *potential*. It helped us to further limit the number of associations stored in the database because we didn't generate association rules anymore, stopping on frequent sets, which we call associations).

Let us first define potential and then illustrate how it works on a small example.

Def.2. Definition of a frequent set's potential

For a frequent set X*, consisting of N items* $\{e_1, e_2, ..., e_N\}$ *where* $\{s_{e1}, s_{e2}, ..., s_{eN}\}$ *are supports for elements* $e_1, e_2, ..., e_N$ *and* S_X *is the support of the whole set, potential is defined by the formula:*

$$p_X = \min_{i=1...N} \left\{ \frac{S_X}{s_{e_i}} \right\}$$

In other words potential is the minimum confidence of any rule of the form $X \setminus \{e\} \rightarrow \{e\}$, where X is the frequent set and e is any of elements $e_1, e_2, ..., e_N$. It is worth

[*] A *frequent set* is what we call the union of items from the left and right side of an association rule. In Agrawal and Srikant's paper it is called *k-large itemset*, where k is the number of elements in it.

noticing that so-defined potential is in fact equal to the minimum confidence of any rule of the form $X_1 \rightarrow X_2$, where $X_1, X_2 \subset X$, $X_1 \cup X_2 = X$, and $X_1 \cap X_2 = \phi$.

Let's assume we have a frequent set *{Java, language, Pascal}*, where elements supports equal *{0.8, 0.7, 0.6}* and the support of the whole set is *0.4*. The potential equals *0.5*, because that is the minimum confidence of any association rule that can be derived from that set. In this case the rule is *{language, Pascal}→{Java}*, which reflects the fact that one is least likely to encounter the "Java" word in the neighborhood of the other two words.

Even after reducing the number of discovered associations and after changing the fashion they are stored in the system, the memory load can still be significant. System administrator must have a way to prevent "stuffing" the system with uncontrolled amounts of data. We introduced such control in Juicer by adding several cutting thresholds. They limit both the parameters describing associations and the number of them at several stages of the algorithm. The final block diagram of the modified Apriori is shown in Fig.5.

It is clear that some of those modifications we introduced to the original Apriori make it a heuristics. However we must remember that, unlike in database applications, we are not particularly interested in precise values of support or confidence. Neither we are in finding all associations. What we would like to get out as a result of processing is some number of strongest relations about which we can say they are probably not coincidental.

3.5 Making use of the discovered associations in Juicer

From the very beginning it was our intention to work with associations and to use them in every possible functional area of the system. Juicer is as, we think, a pioneer of such approaches to text mining and we thought it would be interesting to gain knowledge about the possibilities of porting one of the mainstream data-mining methods to this new ground.

We exploited associations in two out of three Juicer services: searching for related terms and matching new documents to existing domains. In searching for documents with specified terms we had to use a different method – the near-neighbor search. It is so because we have associations for the whole domains, not separate documents, and comparing the associations from a new document to the resultant associations of the domains proved to be ineffective.

3.5.1 Searching for related terms

This application of associations is probably the most interesting in the light of, so popular nowadays, web searching engines like Yahoo, Altavista or Google. It allows

the user to navigate through the net of words that often occur together, thus providing a very reliable tool for narrowing the search.

In Juicer, user sends a query to system server. The query consists of one or several terms, for which associations are to be displayed. System scans its knowledge base, trying to find associations with at least one word from the set provided by the user and then sorts them due to their *matching factor* and then potential if it's not enough. The matching factor is the percent of words from the query that exist in the association found and is used to make sure that associations sharing most words with the query will be displayed at the top of the list.

We consider associations a very useful additional navigation tool in search engines (see section 3.4.2). The need for such hinting help is becoming more and more clear as thousands of new web pages appear on the Internet every day. As a result words are being used in more and more contexts (e.g. "[computer] virus", "spam [mail]", "[procedure] library"), which are often not obvious and hard to look up in printed popular dictionaries. Even for an experienced user it is often difficult to state the query in a way that allows reaching interesting documents quickly and accurately. A user to which the related terms were shown could easily either narrow the search or reject terms that he or she doesn't find interesting.

So far we just partly implemented the scenario – navigating through associations.

3.5.2 Assigning new documents to domains

The task of assigning new documents to some hierarchy of predefined classes is very popular in web-searching engines, where the hierarchy is known and usually handcrafted (for instance Yahoo hires numbers of people to do this task). The assignment can help the user in finding some other related sources of information, or simply to gain knowledge about the document's domain without reading it entirely.

Our solution to the above problem utilizes associations, which we discovered for domains. For every domain we store a certain number of associations with highest support. Those pairs, triplets or, in general, sets of words are most common and characteristic to the set of documents called a domain. The problem lies in finding a measure to estimate how "far" from such characteristic of a domain is the considered document. The process of calculating the *distance function* proposed in Juicer consists of two phases.

To calculate the distance d between document A and domain D_i we must first *fire* associations from the domain on the document. This is a relatively simple and fast process, reversed to discovering associations – we recalculate their support on sentences from the document. In second phase we calculate the distance d using the formula:

Function of distance from document A to domain D_i (7)

$$d(A,D_i)=\left(1-\sqrt{\frac{\sum_{j\in F}\left(s_j-s'_j\right)^2}{|F|}}\right)*\frac{|F|}{|N|}\quad ,F=\left\{j:s'_j>0\right\}$$

N is the number of associations stored for domain D_i, s_j is the support of j-th association in the domain, s'_j is that association support in the document, F is the set of indexes of associations fired on the document with support greater than zero.

Basically the above formula is the Euclidean distance modified by the number of associations successfully fired on the document. Such adjustment is needed to make sure that the number of successfully fired associations has the influence over the final distance. One could imagine a situation in which two domains have the same "unmodified" distance to the document. In such case it is the number of associations that should dominate and the domain from which more of them were fired should be considered "closer" to the document.

We claim that the proposed method has a major advantage over the naive Bayes classifier [5] and the TFIDF vectors [11] cosine similarity measure, most commonly used for text classification [2][6][8][9][13], in that both the popular approaches ignore dependencies between word occurrences in a document. The naive Bayes algorithm explicitly assumes that the word probabilities for one text position are independent of the words that occur in other word positions in the document. Such dependencies are obvious in real-world documents since they are natural for every human language, and they are the key issue of our association-based approach.

4 Conclusions

We have presented an alternate approach to the problem of information extraction from the WWW text documents.

The proposed algorithm for common words discovering shows the advantage of being fast, since its basic idea lies in finding the intersection of given words sets, and general – in that it can be used for non-English documents.

The specific features of HTML documents and users' individual perception of text formatting have been taken into account in the process of generating keywords.

The implementation of Juicer proved that it is possible to port some ideas originally introduced for data mining in VLDB to the ground of information retrieval and text classification.

The results of the computational experiments we have so far performed show that the system can be a very useful tool for automatic processing of WWW documents, although some implementation issues need to be given further consideration.

Specifically the phase of association discovering significantly slows down the system, since it suffers from lack of advanced VLDB techniques (e.g. indexing). Making use of such techniques would help to realize the untapped potential of the proposed approach.

5 Acknowledgements

The authors wish to acknowledge the financial support from State Committee for Scientific Research, KBN research grant. We are also very grateful to Prof. Roman Słowiński for his inspiring suggestions and to Dr. Jerzy Stefanowski for his perceptive and helpful comments. We would also like to show our gratitude to Mr. Przemysław Wesołek for his review and thorough inspection of formulas, and to Mr. Maciej Szkopek for his time spent on Juicer maintenance and documentation.

6 References

1. Agrawal R., Srikant R.: Fast algorithms for mining association rules. *Proceedings of the 20th VLDB Conference*, Santiago, 1994.
2. Craven M., DiPasquo D., Freitag D., McCalllum A., Mitchell T., Nigam K., Slattery S.: Learning to extract symbolic knowledge from the World Wide Web. *Proceedings of 15th National Conference on Artificial Intelligence* (1998)
3. Gilbert D.: *Intelligent agents: The right information at the right time*. IBM Intelligent Agent White Paper (1997)
4. Kohonen T.: *Self-Organizing Maps*. Springer-Verlag, Berlin Heidelberg, 1997.
5. Mitchell T.: *Machine Learning*. McGraw Hill, 1997.
6. Moukas A., Maes P.: Amalthaea: An evolving multi-agent information filtering and discovery system for the WWW. *Autonomous Agents and Multi-Agent Systems* **1** (1998) 59-88.
7. Pankau F., Szkopek M., Weiss D., Wesołek P.: *Automatic creation of a knowledge base from the information on the Internet*. B.Sc. Thesis, Institute of Computing Science, Poznań University of Technology, 1999.
8. Pannu A.S., Sycara K.: A learning personal agent for text filtering and notification. *Proceedings of the International Conference of Knowledge-Based Systems* (1996)
9. Pazzani M., Muramatsu J., Billsus D.: Syskill & Webert: Identifying interesting web sites. *Proceedings of the National Conference on Artificial Intelligence*. Portland (1996)

10. Riloff E.: Using Learned extraction patterns for text classification. In: Wermter S., Riloff E., and Scheler G., eds. *Connectionist, Statistical, and Symbolic Approaches to Learning for Natural Language Processing*. Springer-Verlag, Berlin (1996), 275-289.
11. Salton G.: *Automatic Text Processing*. Addison-Wesley, 1989.
12. Salton G., Buckley C.: *Text weighting approaches in automatic text retrieval*. Cornell University Technical Report 87-881, New York, 1987.
13. Slattery S., Craven M.: Combining statistical and relational methods for learning in hypertext domains. *Proceedings of the 8th International Conference on Inductive Logic Programming*, Springer-Verlag (1998)