Topic and duplicate detection in QA data

Stijn van Balen
sbalen@science.uva.nl

Wouter Bolsterlee
wbolster@science.uva.nl

Marc Bron
mbron@science.uva.nl

Meindert Hart
mhart@science.uva.nl

Nicholas Piël
npiel@science.uva.nl

Stephan Schroevers
sschroev@science.uva.nl

Pedro Silva
psilva@science.uva.nl

Wouter Suren
wsuren@science.uva.nl

ABSTRACT
In this paper, we propose a system for topic and duplicate detection in QA data extracted from the web. The topic detection method is based on a customised Naive Bayes text classifier trained on Wikipedia data and a custom topic taxonomy. The duplicate detection method is based on three variations on the Jaccard similarity measure. We present evaluation measures and promising results that show that the proposed methods perform competitively when compared to human assessments.

1. INTRODUCTION
Automatic question answering (QA) systems try to offer focused information access to textual resources. QA has been an active topic of research, especially since the introduction of a QA track at TREC[1] in 1999.

One way of presenting QA retrieval system results to a user is by using a familiar search engine interface with all results listed in decreasing order of applicability to the user's query. This approach poses limitations on the way users can navigate through the available QA resources, since it requires reformulating the query in order to expand or narrow the search. A much more natural way of finding answers is to allow users to browse through the available resources by grouping conceptually related QA pairs into so-called *topics*.

Another main problem in QA retrieval is that users should not be shown multiple identical answers to a single query. Duplicate or similar QA pairs matching the user's query should be removed or at least (visually) merged, so that similar or alternate answers to users' questions can be discovered easily.

In this article, we will present a method for automatic topic detection based on previously trained classifiers and a method for detecting duplicate and similar QA pairs, both (partly) implemented in the prototype system we developed over the course of this project.

[1]More information about the TREC QA track can be found at http://trec.nist.gov/data/qamain.html

In this research we do not focus on crawling the web and extracting QA pairs to serve as input data for our classification and duplicate detection system. We will not look into retrieval models suitable for QA systems either. Data sources used include data crawled during earlier research [3] and the deliverables of a crawling project currently under development by fellow students.[2]

2. BACKGROUND
QA systems try to address information needs in a way that is most natural to users. Previous research showed that out of millions of queries about 2% were actual questions [3]. Opposed to common text query interfaces that require users to type in any number of keywords, QA systems allow for a conversational query interface, e.g. users type in a question and the system will try to answer it.

As such, the terminology is easily understandable, since it closely resembles human conversational concepts. Firstly, questions and answers always come in pairs, referred to as *QA pairs* in this article. Since most QA data is harvested from Frequently Asked Questions (FAQ) pages on the web, QA pairs are logically grouped together into collections, e.g. a collection of FAQ pages about a specific topic or product. The logical grouping is what we call the *context* of a QA pair. It is imported to keep the context with the QA pairs, because individual questions and answers crawled from the same source do not necessarily repeat important terms all over again, e.g. product names. In such cases the topic of the QA pair is implied by the context, which has important implications for information retrieval.

QA pairs consist of a single question and a single answer. QA pairs may occur more than once in our collection, e.g. common questions or data crawled from mirrored FAQ pages. This is what we refer to as *duplicates*. However, the same question may be answered several times, not necessarily holding the same answer. Since it is not known (nor subject of our research) to discover semantic similarities or contradictions from the *meaning* of the answers, we simply refer to this as *alternative answers*.

3. SYSTEM ARCHITECTURE
Our system does not stand on its own, but is part of a larger project that aims to create an online question answering service. This means other student projects focus at crawling the web for QA pairs, creating a means of searching this data, and finally making

[2]Some of the data used for testing purposes is available from http://ilps.science.uva.nl/Resources.

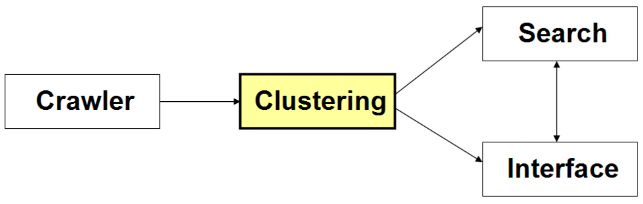

Figure 1: IR high-level Architecture

the data accessible to users through an intuitive web interface. Our topic and duplicate detection system plays a central role in the overall system architecture, which is depicted in figure 1.

As for the architecture of our subsystem, we split up all tasks into four different modules as shown in figure 2. The *classification* module does the topic detection by classifying incoming data that is learnt from Wikipedia data, as outlined in the following sections. The *grouping* module aims to remove duplicates and group similar QA pairs into clusters. The *core* module provides basic data structures used throughout the system and also handles input and output of data. Finally, the *utils* module provides helper routines used in other modules, including but not limited to text processing such as markup stripping.

The incoming and outgoing data communicated with the other subsystems of the question answering service is encoded in XML because of its well defined syntax and validation support. We created a custom Document Type Definition (DTD) to specify the exact data format accepted and output by our system.

4. TOPIC DETECTION

Topic detection is about grouping semantically related QA pairs into topics. Grouping of QA pairs may be done in several ways, such as unsupervised clustering with automatic or manual labeling of the clusters after the algorithm's halting criterion has been met. This approach looks useful at first sight, but tends to result in many 'clouds' of QA pairs that are hard to label, and even harder to navigate. The Hierarchical Agglomerative Clustering algorithm (HAC) tries to solve this issue by constructing trees instead of unconnected clouds. However, since HAC does not use backtracking, incidental incorrect merging of two 'good' clusters may result in incomprehensible topics [10].

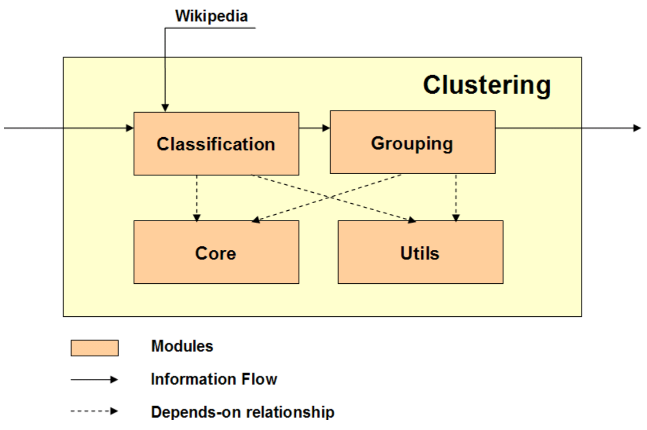

Figure 2: Project Architecture

Because typical QA systems fetch data using web crawling techniques, topic detection methods have to deal with constantly changing collections. Clustering techniques often require the whole collection to be known when the clustering algorithm is executed, or at least perform best if this is the case. This poses a problem for newly crawled, previously unseen instances that cannot be classified correctly if the training data did not contain any QA pairs resembling the new data. Misclassification (or no classification at all) of unseen instances applies to any grouping approach. No classification method performs correctly if the algorithm is not trained with a representative sample of data belonging to all possible topics.

4.1 Taxonomy

Instead of deducing a taxonomy from the available training data, we have opted to use a predefined, fixed taxonomy. This approach means that we reduce the topic detection task to a supervised classification task. This effectively works around the problem we mentioned, namely the induced grouping or hierarchy not being comprehensible to users. However, this approach poses two new problems: choosing a taxonomy and obtaining useful training data.

Regarding the first problem, an important question is whether a taxonomy can be constructed that covers the 'complete world'. Previous attempts have lead to enormous ontologies with too many classes to be useful for QA categorisation [4]. For adoption in our system, we have therefore considered two possible taxonomy candidates with a more concise structure: the Wikipedia portal hierarchy[3] and the taxonomy used by the Open Directory[4] project. Manual classification of a sample consisting of 200 QA pairs by 8 different assessors, in such a way that each QA pair in the sample was categorised twice, resulted in major classification problems when using the first two levels of the Wikipedia portal hierarchy; over 40 QA pairs were found to be about a topic that did not fit in any of the topics in the hierarchy, i.e. over 20% of the data could not be classified at all. A repeated experiment with the Open Directory hierarchy yielded much better results; after manual addition of three topics and removal of four topics all QA pairs fitted more or less in the taxonomy. The enhanced taxonomy used for our system is depicted in table 1.

4.2 Training Data

The second problem is the collection of useful training data. Even though we have opted to not use Wikipedia for our taxonomy, we do use Wikipedia article data as training data for classification. This means we have trained our classification algorithm with data that is of a completely different type than the QA pairs we will be classifying, which is considered bad practice. However, we found that this is not problematic at all, possibly because an encyclopedia is a primary source for people looking for answers. Another advantage is the very large number of the Wikipedia articles (over 1.6 million), so that most topics occurring in the real world will be covered in the training set. However, this also means a careful selection of useful articles must be made.

In order to extract articles from Wikipedia corresponding to our taxonomy, a mapping needs to be made from this taxonomy to the

[3]The complete listing can be found at http://en.wikipedia.org/wiki/Portal:List_of_portals.

[4]The Open Directory taxonomy is available at http://dmoz.org/.

Table 1: The taxonomy used for topic detection.

Topic	*Subtopics*
Recreation	Outdoors, Food, Travel
Health	Fitness, Medicine, Alternative
Society	People, Religion, Politics, Issues, Legal
News	Current events, Newspapers
Arts	Music, Movies, Television
Reference	Libraries, Education, Maps
Business	Jobs, Finance, Companies
Shopping	Clothing, Autos, Gifts
Regional	North America, Europe, Asia
Home	Homeowners, Family
Computers	Software, Hardware, Internet, Security
Kids and teens	Entertainment, School
Science	Engineering, Psychology, Biology, Physics
Sports	Soccer, Football, Basketball
Games	Role playing, Board

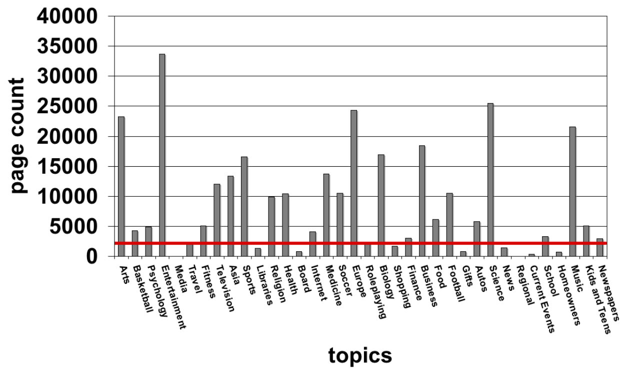

Figure 3: Distribution of Wikipedia articles over topics.

category structure of Wikipedia. Most of the times a one-to-one mapping from our taxonomy to a Wikipedia category page could be made, e.g. the *Basketball* topic in our taxonomy directly corresponds to the *Basketball* category in Wikipedia; when an exact mapping could not be made, it was not hard to find a category that suited the topic, e.g. *Jobs* is mapped to *Employment*.

A Wikipedia category page does not actually contain much textual information that can be used to train a classifier. Instead, category pages function as hubs pointing to subcategories. In order to obtain useful textual information we enrich our taxonomy: for every category in the taxonomy, the corresponding category page is expanded by following the links to the subcategories. The subcategories of the pages are aggregated and appended to the main category of the taxonomy, up to three levels deep, e.g. *Television* is expanded to include *Television genres*, which is expanded to include *Children's television*, which is in turn expanded to include *Children's television series*. We believe that by expanding the category pages the topics are better semantically defined, resulting in more accurate classifications.

Now that we have an expanded category listing for each topic in our taxonomy, the actual data from Wikipedia articles can be extracted. Since each article contains a link to the category it belongs to, we can easily extract only those articles belonging to one of the topics in our taxonomy, i.e. useful textual training data for our classifier. Initial results showed a very non-uniform distribution of articles over the topics in our taxonomy; this is depicted in figure 3.

To make sure the classifier is not biased mainly because of non-uniform class sizes, we defined a size threshold for the number of articles assigned to each topic. Most expanded categories contain at least 2300 articles, so we choose that value as the upper limit. Selecting 2300 articles from the large categories can be done in several ways, of which a random selection is the most trivial to implement. However, we believe our classifier is better trained when it is provided with high quality articles rather than a random sample articles of varying quality. We therefore need a quality assessment measure. We opted to use the PageRank algorithm [5], which computes the relevance of pages depending on their internal link structure. The PageRank of page p is based on the amount of pages that point to p. The more in-links that p has, the higher its relevance, or *prestige*. The PageRank of p is furthermore influenced by the PageRank of the pages that link to p. The higher these PageRanks, the higher the PageRank of p. Finally the final PageRank value is normalised by the total number of out-links of the pages that link to p. The higher this amount, the less 'exclusive' p is for these pages, so the lower its PageRank. For all selected articles in our Wikipedia article selection for a given topic, we calculate the PageRank score to obtain an ordered list of pages. Finally, we use the top 2300 pages from this ranking, because we believe that articles that are often linked to are more likely to contain relevant content.

We now have a suitable uniformly distributed collection of articles extracted from Wikipedia. The next step is to apply text processing routines to cleanup and enhance the data. First of all, we extract only useful article content by removing HTML and Wiki markup. The next step is lexical analysis: the text is tokenized into terms by splitting on word boundaries. Our final cleanup includes folding all characters to lowercase and removal of accents and diacritics to offer robustness to irregular spelling.

4.3 Training the Classifier

The final preparation step for our topic detection system involves choosing and training a text classifier. Many methods exist for text classification problems, amongst which Support Vector Machines (SVM) provide good performance for high-dimensional data such as textual data [9]. However, we chose a Naive Bayes classifier for our baseline, because of the relative ease of implementation, its good performance, and low training costs in terms of speed and processing power [8]. The Naive Bayes classifier is used for many text classification problems, like spam filtering [2, 1].

A Naive Bayes classifier assumes independence of the observable features (in our case the terms), which means that the order of the words as they occur in the original document is not taken into account. It is based on the Bayesian theorem, which is defined as follows for text classification purposes:

Given a set of terms t_i contained in a document D, where $D = t_1, t_2, \ldots, t_n$, the posterior probability for event c_j of a set of outcomes $C = c_1, c_2, \ldots, c_m$ is calculated as

$$\begin{aligned} P(c_j \mid D) &= P(c_j \mid t_1, t_2, \ldots, t_i) \\ &\propto P(t_1, t_2, \ldots, t_i \mid c_j) \cdot P(c_j), \end{aligned}$$

where $P(t_1, t_2, \ldots, t_i \mid c_j)$ is the posterior probability that the tokens occur given occurrence of class c_j. The Naive Bayes assumption is then defined as follows:

Algorithm 1: Tournament algorithm for topic assignment.

```
Data: Text $t$ and set of classes $C$
Result: The winner class $c_w$
begin
    $c_w \leftarrow$ pop($C$)
    foreach challenger $c \in C$ do
        $c_w \leftarrow$ naive-bayes-compare($t, c_w, c$)
    end
    return $c_w$
end
```

$$\begin{aligned} P(c_j \mid D) &= P(c_j \mid t_1, t_2, \ldots, t_i) \\ &\propto \prod_{k=1}^{i} P(t_i \mid c_j) \cdot P(c_j). \end{aligned}$$

Naive Bayes' performance can be improved by treating the multiple class problem as a binary classification problem [6]. The classification of an instance is based on the final outcome of a tournament of binary classifications. During each round two possible classes are compared; the winner of this comparison is then challenged by the next candidate class. The outcome of the tournament is decided after all candidate classes have been considered, as shown in algorithm 1.

In order to further increase classification performance, we use a χ^2 distribution to combine individual word probabilities into a combined probability [7], based on Fisher's method to combine extreme probabilities into one test statistic $\chi^2_{2k} = -2 \sum_{i=1}^{k} \log_e(p_i)$.

5. DUPLICATE DETECTION

Users do not want questions to be answered in the same way for the first couple of results when asking a question to a question answering system; they would like the system to return only those questions — or similar questions — with corresponding answers that differ from each other. This translates to a duplicate detection and alternative answer detection task. The goal is to mark duplicate questions and answers so that search interface can remove the duplicates, or at least group these together. It's beyond the scope of our research to decide whether to group or whether to remove duplicates, but enabling others to make an informed choice is within the scope of our project.

5.1 Similarity combinations for two QA pairs

When comparing two QA pairs, we define the different types of combinations of similar questions and answers as follows:

1. *Unique QA pairs:* Not similar questions, not similar answers;
2. *Alternative answers:* Similar or same questions, but not similar answers; and
3. *Duplicate QA pairs:* Similar questions and similar answers.

Our task is two detect the last two combinations, since those QA pairs that contain no similar questions are uniques and therefore not of interest for this task.

A fourth combination that has been left out are QA pairs with non-similar questions but similar answers. Depending on the answer type, these QA pairs could be classified as having semantically similar questions. If the answer is short, the chance of the two corresponding questions actually being the same is low. For example, it is hard to determine whether the corresponding questions

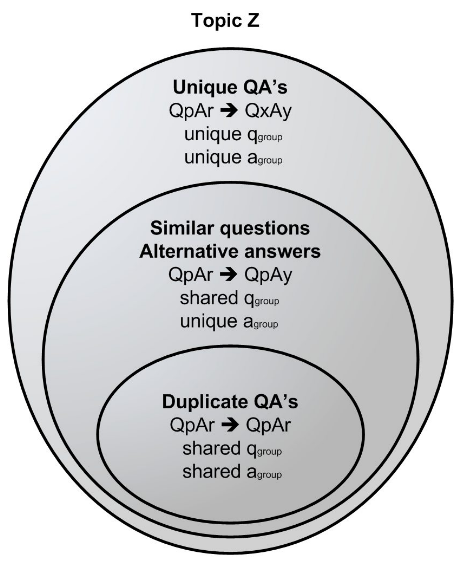

Figure 4: Overview of nested duplicate types

are semantically similar for answers such as '23' or 'In Prague'. However, if an answer is 'The coral snake sheds his skin every six months' and the questions are not marked as similar, there is a reasonable chance the questions have semantical overlap. Although this is a very interesting area of research, we have chosen not to dive into it, and stick to finding only syntactically similar question and answer pairs.

Considering the above discussion, our task can be formulated as finding subsets of QA pairs with similar questions, and within each subset finding the sub-subsets of QA pairs that have similar answers as well, resulting in the configuration depicted in figure 4.

5.2 Detection methods

We divide the large amount of QA pairs in a number of topic clusters, assigned by the topic detection part of our system. For scalability reasons we only detect duplicates within sets of QA pairs grouped into the same topic.

We could filter out exact duplicates using a hash table before starting the rest of the method. Converting the QA pairs using a hash function makes fast comparison possible, which might turn out in a performance boost. Although this is trivial to implement, we have decided not to do it, because the probability of two QA pairs being identical is extremely low. This means that even exact duplicates will be detected using slower methods.

Before we describe how we are going to detect anything, we will elaborate on what we hope to detect. There are several types of similar strings of text, which all need a slightly different way of detecting. With *exact duplicates* on top, the next level contains typographical errors or *typos*, which means that only one or a few characters differ between the two strings. Another type are strings with the same words in a different order, so-called *near duplicates*. When the meaning of two strings is same, a few different words might be used. When strings have most likely the same meaning, but they do have little word overlap, we call them *semantically similar*. Finally, we define strings as *similar* if they still use a few of the same words, even if the meaning is likely to differ.

As a measure of similarity we use an approximation of the Jaccard coefficient:

$$\text{Sim}(d_1, d_2) = \frac{|T(d_1) \cup T(d_2)|}{|T(d_1) \cap T(d_2)|}$$

,

where $T(d)$ is the set of terms in a document. The approximation algorithm, based in sketches, is presented in section 5.3. The three detectors we used are constructed from the Jaccard coefficient by varying the meaning of function $T(d)$, that is, modifying what is considering a term. For instance, to check for typos the string is converted to a list of 3-grams or character-shingles consisting of 3 characters with a certain overlap. Near duplicate detection uses word shingles as terms. Instead of shingles consisting of 3 *characters*, this method uses 3-*word* shingles with overlap on the previous and next list items. For strings with word overlap and similar strings, the full string serves as input. No conversion is applied for these types of similarity.

5.3 Implementation

We start by presenting the high-level algorithm for duplicate detection, outlined in algorithm 2. We iterate over the set of QA pairs (the first *for* loop) and for every QA pair which question has not been classified, i.e. does not have a q_{group}, we give it a newly generated q_{group} and then we iterate within every other pair which question hasn't been classified and compare them with q. If they match, i.e. *IsDuplicate* returns true, we attribute the same q_{group} to both. After this step we repeat the same method for the answers, except that we only search for duplicate answers within the duplicate question groups found in the previous step, i.e. for each $g \in$ QGroups.

When the algorithm has ended, each QA pair has a q_{group} and a a_{group} attribute, with the possible configurations as described in section 5.1. These two fields correspond to the *sim-group* and *alt-group* element attributes in our output XML, as defined in our document type definition.

The key functionality, however, is hidden in the *IsDuplicate* functions. These two functions that detect whether two questions or answers are duplicates are used as an abstraction for the various duplicate detection methods. In the final version of our prototype system *IsDuplicate* returns true if any of the individual detectors marks the pair as being a duplicate, i.e. we apply an OR-based combination. This decomposition allows us to change the specific methods we use, or the function used to compose them, with only minor changes to how the implementation as a whole works.

Because set intersection is a computationally expensive operation, we approximate the Jaccard similarity coefficient with a method that uses permutations, based in the following theorem:

$$\Pr_\pi\big(min\pi(A) = min\pi(B)\big) = \frac{|A \cap B|}{|A \cup B|}$$

The idea here is to represent every document that one wants to compare — in our case questions and answers — with sketches that consist of the application of a number of n randomly generated permutations for the given document; from each permutation we store only the minimum value. We call the various values obtained in this way a *sketch* of the document, where similar documents have similar sketches. These values are stored in the *Sketches* structure, indexed by QA pair, and keeps sketches of its respective question and answer and can be implemented in any hash-table like data type. In our implementation n is set to 25. We are now ready to write the sketches creation routine, depicted in algorithm 3.

Because we have a different function T for each of the detection methods used we actually end up with three sketches structures, for clarity omitted from the algorithm listing. At this point the two *IsDuplicate* functions become a simple matter of comparing the sketches of the questions (*IsDuplicateQuestion*) or the answers (*IsDuplicateAnswer*) of the two QA pairs against a threshold t, as shown in algorithm 4. Note that *IsDuplicateAnswer*, the function that detects duplicate answers, can be obtained from the above al-

Algorithm 2: Duplicate Detection

```
input  : Set of QA pairs
output: Set of annotated QA pairs
create-sketches()
$q_{\text{group}} \leftarrow 0$
foreach $qa \in QA$ do
    if ¬ QuestionClassified($qa$) then
        $qa.q_{\text{group}} \leftarrow q_{\text{group}} + 1$
        foreach $qa2 \in QUnclassified$ do
            if IsDuplicateQuestion($qa$, $qa_2$) then
                $qa_2.q_{\text{group}} \leftarrow qa.q_{\text{group}}$
            end
        end
    end
end
$a_{\text{group}} \leftarrow 0$
foreach $g \in QGroups$ do
    foreach $qa \in g$ do
        if ¬ AnswerClassified($qa$) then
            $qa.q_{\text{group}} \leftarrow a_{\text{group}} + 1$
            foreach $q2 \in AUnclassified$ do
                if IsDuplicateAnswer($qa$, $qa_2$) then
                    $qa_2.a_{\text{group}} \leftarrow qa.a_{\text{group}}$
                end
            end
        end
    end
end
return QA
```

Algorithm 3: Create Sketches

```
input  : QA, set of QA pairs
output: Sketches
Generate $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, set of random permutations
foreach $\pi \in \Pi$ do
    foreach $qa \in QA$ do
        $s_q = min\pi(T(qa.q))$
        $s_a = min\pi(T(qa.a))$
        Sketches.add(qa, $s_q$, $s_a$)
    end
end
return Sketches
```

Algorithm 4: Determine Duplicate Question

```
input  : $qa_1$, $qa_2$, Sketches, t
output: $true$ if $qa_1$ and $qa_2$ have duplicate questions,
$false$ otherwise
$x \leftarrow 0$
$S_a = \{s_{a1}, s_{a2}, \ldots, s_{an}\}$ = Sketches.getQuestion($qa_1$)
$S_b = \{s_{b1}, s_{b2}, \ldots, s_{bn}\}$ = Sketches.getQuestion($qa_2$)
foreach $s_{ai} \in S_a, s_{bi} \in S_b$ do
    if $s_{ai} = s_{bi}$ then
        $x \leftarrow x + 1$
    end
end
return $x/n > t$
```

gorithm by replacing *getQuestion* with *getAnswer*. Furthermore, remember that we are omitting the fact that we have multiple detectors, and consequently multiple sketches structures. Our implementation corresponds to the logical OR of instantiations of the above algorithm with the various sketches as parameters.

Duplicate detection is a computationally complex task. Every naive implementation, including ours, needs to compare items pairwise, which implies that the algorithm's memory and temporal needs are $\mathcal{O}(n^2)$, where n the number of pairs. For this reason we decided to detect duplicates only within QA pairs classified as having the same topic. The reasoning here is not only computationally inspired, but also that, if two QA pairs are not about the same topic, they cannot be duplicates. The consequence is that we subdivide the sets over which we run duplicate detection, effectively lowering the value for n. Still, this does not change the algorithmic complexity of the approach, and therefore does not solve the scalability problem by itself. Since we run duplicate detection as an off-line program execution time is not critical, but memory is. To solve this problem completely we would need to change the naive approach to duplicate detection. One possible solution involves breaking the QA set in smaller sets that we don't need to keep in memory simultaneously. This wasn't done due to lack of time, since it corresponds to a complex and error-prone programming task.

6. RESULTS AND EVALUATION

In this section, we evaluate both the topic detection and the duplicate detection methods outlined above.

6.1 Topic Detection

As stated before we created a test set consisting of 200 QA pairs with manual topic annotations by 8 different persons. We used this test set to tune the parameters for our classifier. To verify that our assumption that a threshold of 2300 articles per topic is the correct number to use for training, we did several test runs using different thresholds to train the classifier, ranging from 100 to 5000. The results showed that no improvements can be achieved by selecting more than 2300 articles, while a significant lower threshold results in higher error rates; we assume this is caused by incomplete coverage of real-world concepts in small training sets. Extremely high thresholds (over 4000 articles per topic) severely degrade performance, probably due to a bias for topics that contain many articles and lower textual quality of the pages in these topics.

To test the hypothesis that the tournament approach to Bayesian classification outperforms Naive Bayes, we tested both classification methods. Our performance measure is accuracy, defined as the percentage of instances that were correctly classified with regard to the manually annotated test set. It turned out that the tournament approach has a 55% accuracy, whereas Naive Bayes results in only 43% correctly classified instances.

These numbers seem quite low at first sight, however classification is not exact science; humans may disagree on the labels that should be assigned to an instance. Manual classification of the test data resulted in only 55–60% accuracy, which implies we cannot expect any classification method to achieve 100% accuracy. Instances can be placed into multiple categories, but our system does not take this into account. Furthermore, if an instance is classified into a certain class while our manual classification classified the same instance into its superclass, one may opt to treat this a correct classification as well. When we apply this reasoning to our test runs, accuracy of the tournament method increases from 55% to 63%. Even then manual inspection reveals that some wrongly classified instances — that is, according to our manual labelling — do make sense after all, e.g. our system classified one instance as belonging to the *Computer* $\rightarrow$ *Security* topic, while our test set placed it into the *Internet* topic, which can be considered a reasonable alternative.

6.2 Duplicate Detection

For evaluation of the duplicate detection module we manually constructed a test set of 320 questions. Starting from 80 original questions we created three types of duplicates for each. The three duplicate classes chosen are *near duplicate*, *similar* and *semantically similar*. This arrangement was chosen so that we could tune each of the detectors in a different test set, e.g. one that would favor that detector, and then evaluate the composite detector on the complete collection of 320 questions. This allows us to get an indication of how well the OR-based compositional approach performs compared to individual detectors.

6.2.1 Evaluation Metrics

One way to evaluate duplicate detection is to consider it a search task for items that match a value of a feature, in this case duplicate, over a number of input candidates and in which the number of returned results can vary. Two standard measures for evaluating performance in this case are *precision* and *recall*. Our interpretation does not differ from the traditional interpretation: precision measures the ratio of correctly assigned duplicates to assigned duplicates while recall measures the ratio of correctly assigned duplicates to golden standard duplicates:

$$\begin{aligned} P &= \frac{|\text{correct} \cap \text{returned}|}{|\text{returned}|} = P(\text{correct}|\text{returned}) \\ R &= \frac{|\text{correct} \cap \text{returned}|}{|\text{correct}|} = P(\text{returned}|\text{correct}) \end{aligned}$$

To get a measure that rewards both precision and recall, in order to allow us to optimise by tuning thresholds or other variables, we use the F_1-measure, i.e. the harmonic mean of the precision and recall values, defined as:

$$F_1 = \frac{2PR}{P+R}$$

An alternative way to do evaluation of duplicate detection is to consider it an unsupervised clustering task over duplicate groups, with a golden standard to which the results can be compared. The usual evaluation measure in this situation is *purity*, defined as the ratio of the largest class in the cluster divided by the total cluster size. When tuning the thresholds for each of the clustering techniques, the number and size of the clusters ranges from many clusters with only one item to one big cluster with all items. Obviously, the purity of a cluster with only one item is always maximal. This is a problem because not only the purity of a cluster is important but also the number of duplicates that are actually found.

To compensate for this we introduce a purity measure that is punished for degenerating into singleton clusters. We call this measure non-degenerative purity (NDP) as it tries to find pure clusters with as many items as possible. It is calculated as follows. Assuming the testset was built from creating k duplicates for each of O original questions, we then know what the optimal number of clusters is (O) and also that the maximum number of clusters is ($(k+1)O$). Assuming the clustering algorithm found C clusters, then the maximum number of clusters minus the number of clusters found results in a difference ($(k+1)O - C$). This difference is normalized by the maximum difference possible. For only singleton clusters this results in zero and for the optimal number of clusters in one. This value is multiplied by the purity to give the value for the non-degenerative purity:

$$\text{NDP}_i = \frac{\max_j(n_{ij})}{n_i} \cdot \min\left(1, \frac{(k+1)O - C}{kO}\right)$$

6.2.2 Results

We started by evaluating the individual detectors against each individual test set (and the original questions) to try to get an understanding of how they operate, by constructing a series of 9 graphs measuring the variation in the P, R and F_1 measures as the threshold t is increased.

The first remarks are that the detector that using 3-word shingles as terms performed the worst, and apparently the duplicates it found are an approximate subset of those found by the other detectors. Additionally, runs on the semantic duplicate test set provided worse results than the others, as we expected. The best performing detector was the one based on 3-grams of characters. It performed better than both others on all test sets. Figure 5 shows the results for the run on test set *SAM*, for which the detector works surprisingly well, considering it was only designed to detect typos. For the *SAM* testset the detector based on words did comparably well. However, although it was built for detecting similars, the results for the *SIM* test set are considerably worse, as can be seen in figure 6.

As mentioned, it is noticeable that character 3-grams performs better than single words. Because the graphs for single words has the same shape as those for 3-grams, but a bit lower, we assume there is significant overlap in the duplicate sets detected by these two detectors. Character n-grams for values $n \neq 3$ were also evaluated over this test set, but performance decreased rapidly.

We then tested the composite detector on a test set composed of *SAM* and *SIM* (plus the original questions) to get a measure of how the composition behaves over a union of the test sets, which can be see in figure 7. Finally, we expanded the test set further by adding *SEM* to the remaining set; see figure 8.

As can be seen performance decreases as the test set increases, although this is also a matter of the difficulty of the number of du-

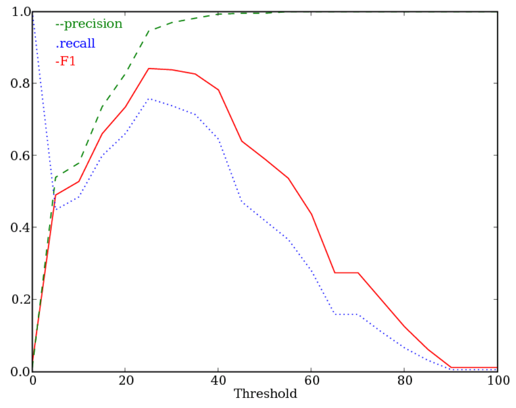

Figure 5: PRF of character shingles detector on testset SAM

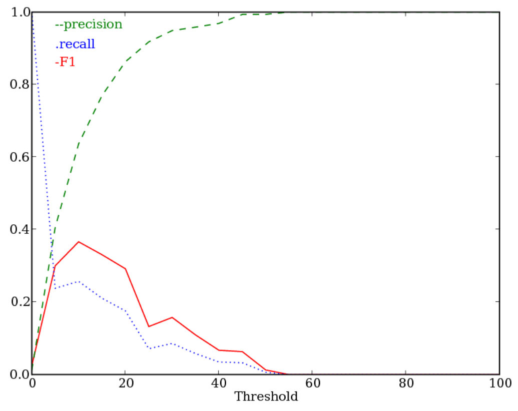

Figure 6: PRF of single word detector on testset SIM

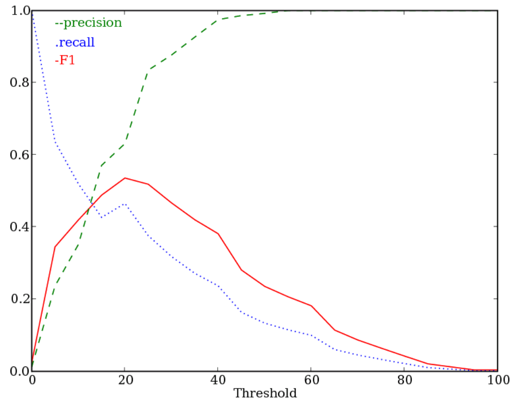

Figure 7: PRF of composite detector on testset SAM+SIM

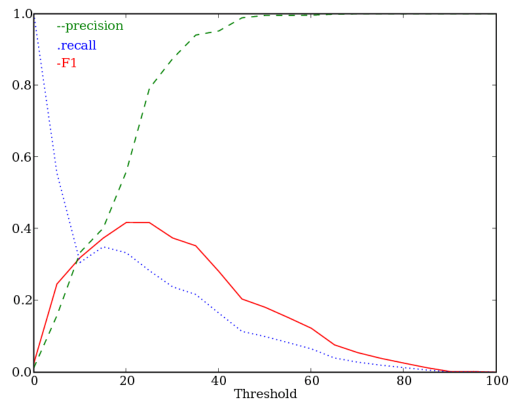

Figure 8: PRF of composite detector on testset SAM+SIM+SEM

Table 2: Optimal Tresholds for the individual detectors

Detector	*Threshold*
Words	40
Character n-grams	20
Word shingles	25

plicate types increasing. On the other hand, with a sample of 320 questions it is hard to make judgements on how results would scale to an environment where there is more word overlap between dissimilar questions. Part of the problem is that it gets harder to describe what *duplicate* means in such environments. Another part has to do with the need for manually constructed test sets, which involves either comparing documents pairwise or constructing similars from a sample; both approaches demand an extreme amount of effort to build a sample that can be treated as being web-like.

Finally, we tuned each detector's threshold independently using the F_1-measure obtained from running the OR-composition over the union of *SAM* and *SIM*. A graph for this would be hard to construct as it would require 4 dimensions, therefor we summarize the results in table 2. One reading of the values is that thresholds for word and word-shingles are higher than what their optimal values appear to be from the individual tests. This is remarkable, because it seems to indicate that in these threshold configurations the detectors are used by the composition to detect a disjoint sets of duplicates when compared to the character n-grams detector only, showing that they are in some level complementary as was conjectured.

7. CONCLUSION

Question answering systems become an increasingly popular research topic. In this paper, we presented a system to structure large collections of QA data into a taxonomy that can easily be understood and intuitively navigated by humans. Additional structuring of the data is provided by grouping duplicate and similar QA pairs. Our system can be used as a part of a question answering system that incorporates web crawling, implements methods for searching the data and provides access to the data through a web interface.

Our topic detection method uses a hand-picked topic taxonomy based on the Open Directory project. QA pairs are assigned to a topic using a customised Naive Bayesian text classifier, which we trained on Wikipedia data. Our method carefully selects Wikipedia articles by picking only those articles that correspond to the topics in our taxonomy. We further restrict the article selection by applying PageRank to pick only those articles that are most relevant. Our results show that a plain Naive Bayes classifier enhanced with a χ^2 distribution has an accuracy of 43%, which we improve to 55% by applying a tournament based approach. If we choose to treat a classification as correct if the classifier labels a QA pair as a member of a subtopic of the manually labelled topic, accuracy further improves to 63%. This is a competitive result compared to a 55–60% accuracy for manual topic assignment.

The duplicate detection method we propose is based on the Jaccard similarity measure using three different functions for generating terms from textual data. The detector based on character n-grams substantially outperforms the whole-word and word-shingle splitting methods. We also showed that an OR-based composition approach of all three detection methods with custom parameter tuning over the detector combination as a whole yielded better results from individually optimally tuned detection methods, because each method detects different subsets.

8. REFERENCES

[1] I. Androutsopoulos, J. Koutsias, K. Chandrinos, G. Paliouras, and C. Spyropoulos. An evaluation of Naive Bayesian anti-spam filtering. pages 9–17, 2000.

[2] P. Graham. A Plan for Spam. 2002.

[3] V. Jijkoun and M. de Rijke. Retrieving answers from frequently asked questions pages on the web. *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 76–83, 2005.

[4] D. Lenat. CYC: a large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing order to the web. 1998.

[6] J. Rennie. Improving Multi-class Text Classification with Naive Bayes. 2001.

[7] G. Robinson. A Statistical Approach to the Spam Problem. 2003.

[8] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian approach to filtering junk e-mail. In *AAAI-98 Workshop on Learning for Text Categorization*, pages 55–62, 1998.

[9] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *The Journal of Machine Learning Research*, 2:45–66, 2002.

[10] O. Zamir, O. Etzioni, O. Madani, and R. Karp. Fast and intuitive clustering of web documents. *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 287–290, 1997.