FP7-ICT-2011.4.2
Contract no.: 288342
www.xlike.org

Deliverable D4.3.2

Final event extraction prototype

Editor:	Gregor Leban, JSI
Author(s):	Gregor Leban, JSI, Aljaž Košmerlj, JSI, Janez Brank, JSI; Blaž Fortuna, JSI
Deliverable Nature:	P
Dissemination Level: (Confidentiality)[1]	PU
Contractual Delivery Date:	M33
Actual Delivery Date:	1.10.2014
Suggested Readers:	XLike project partners
Version:	1.0

[1] Please indicate the dissemination level using one of the following codes:

• **PU =** Public • **PP =** Restricted to other programme participants (including the Commission Services) • **RE =** Restricted to a group specified by the consortium (including the Commission Services) • **CO =** Confidential, only for members of the consortium (including the Commission Services) • **Restreint UE =** Classified with the classification level "Restreint UE" according to Commission Decision 2001/844 and amendments • **Confidentiel UE =** Classified with the mention of the classification level "Confidentiel UE" according to Commission Decision 2001/844 and amendments • **Secret UE =** Classified with the mention of the classification level "Secret UE" according to Commission Decision 2001/844 and amendments

Keywords:	event detection, topic tracking, cross-lingual document clustering, event registry, event template

Disclaimer

Full Project Title:	XLike – Cross-lingual Knowledge Extraction
Short Project Title:	XLike
Number and Title of Work package:	WP5
Document Title:	4.3.2 Final event extraction prototype
Editor (Name, Affiliation)	Gregor Leban, JSI
Work package Leader (Name, affiliation)	JSI
Estimation of PM spent on the deliverable:	10

Executive Summary

In this document we present the final pipeline for detecting events from articles collected worldwide.

The presented system starts by first collecting articles from more than 50.000 news sources. The articles and their meta-data are collected and sent to the annotation service which identifies mentions of named entities and keywords. Additional information extraction is also done, such as extraction of date mentions and identification of similar articles in other languages. Using the available information, the developed clustering algorithm identifies groups of articles that discuss the same event. Clusters are then also combined cross-lingually using a classification model that uses features such as the number of cross-lingually linked articles and concept similarity. Clusters about the same event are then stored in the Event Registry and represented with a unique event ID. For each event, the system extracts the most relevant information, such as title, summary, location, date and top concepts. All stored information is accessible using the Event Registry's web interface which provides extensive search options as well as a large number of possible visualizations of search results. The data in the Event Registry is also accessible using the API. In this document we provide details about the main API calls used to obtain information about events and articles. We also describe the Python library we developed in order to make the search easier.

The developed service is publicly available at http://eventregistry.org/.

Table of Contents

List of Tables

No table of figures entries found.

List of Figures

Abbreviations

API	Application programming interface
BIC	Bayesian information criterion
BOW	Bag-of-words
B/S	Browser/Server
CCA	Canonical correlation analysis
CLSS	Cross-Lingual Similarity Service
LOD	Linked open data
TF-IDF	Term frequency – inverse document frequency
UML	Unified modelling language
SVM	Support Vector Machine

Definitions

Article	An instance of a news report.
Entity	*Person*, *Organization* or *Location* contained in the *Title* or *Content* of an *Article*
Concept	*Entity* or a general thing that can be recognized and annotated in the articles (such as hunger strike, lawyer, military, cost, debate, etc.)
Cluster	A group of articles describing the same event
Event	An event is anything significant that is occurring in the world. An event is in Event Registry represented as one or more clusters of articles.
Event Registry	A system for managing and identifying world events

1 Introduction

There are thousands of news articles written and published every day by various news publishers across the world. There articles discuss various topics, but most of them describe situations and events that are currently occurring in the world. The goal of our work in event extraction is to identify the similarities between the articles that are being written and to represent the related groups of articles as events. For our purposes, an event is any significant happening in the world that was reported by a sufficient number of news publishers. An example of an event is shooting down of the Malaysian Airlines airplane on July 17th 2014.

There are various reasons why we are interested in identifying world events. The articles, as they are currently written, provide no structured view on the conveyed information. Although most articles about events answer the main questions about who, where, when and what, this information remains hidden in the text and requires the reader to manually extract it by reading the article. Additionally, each article only provides one point of view on the event. An alternative or more complete view can be obtained by reading other articles on the event, which is typically time consuming as it requires finding the appropriate article on another website. A global database of world events also has a great value in itself since it offers an insight into the topics and activities that our society is involved in.

In this document we will present the final version of the event extraction prototype. This work is a continuation of the work done in deliverable D4.3.1 Early event extraction prototype. Most of the components and services used in event extraction were significantly updated and extended with additional features. Some parts (like extraction of date mentions and cross-lingual article matching) of the whole pipeline, however, have not changed significantly from the first prototype. In order to present in the deliverable the complete system, we nevertheless include here also the information about those parts, even though their description is very similar to the one provided in D4.3.1.

The document is organized as follows. We start by describing the overall pipeline used by the system. Next, individual parts of the pipeline are described in details. We provide information about the data collection, followed by the pre-processing steps that are performed on each article independently. The event construction phase is presented next where we describe the clustering method and algorithms for extracting event information. We continue by describing the Event Registry frontend including its list of search and visualization options. Lastly, the API that can be used to access the data is presented together with the Python library that simplifies the data access.

2 Event detection pipeline

In order to detect and store events we use the pipeline shown in Figure 1. The pipeline contains four main parts:

1. Input data; provides input data that can be used for event extraction
2. Pre-processing steps; collected articles are individually processed by linguistic tools developed in WP2 and WP3 in order to extract available semantic information
3. Event construction; in this phase we identify groups of articles that correspond to individual events. From each identified group we extract available event information (time, location, entities, and category).
4. Event storage & maintenance; after the events are constructed they are stored in a database of events and are indexed across different fields to provide search functionalities.

In the rest of the section we will now describe in more detail individual parts of the pipeline together with the expected inputs and outputs.

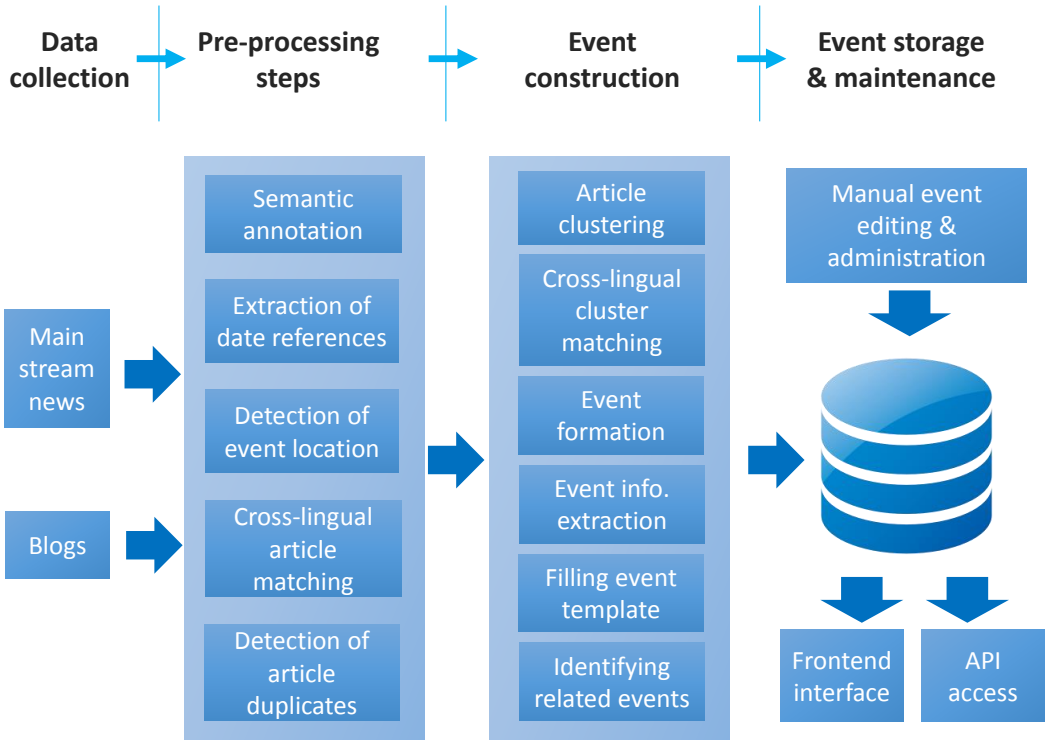

Figure 1 Pipeline used for detection and storage of events

3 Input data

The data that Event Registry uses for identification of events is obtained using the Newsfeed service (http://newsfeed.ijs.si/) [D1.3.1]. The service constantly monitors and collects articles from more than 75.000 RSS feeds. The feeds represent various worldwide news sources (majority) or blogs (minority). The collected articles are in more than 40 languages, where the majority of articles are in English, Spanish, German, French, and Chinese. Based on the availability of the text processing tools in XLike, we process and further analyse only articles in English, Spanish, German, Chinese and Slovene language. Articles in other languages are currently not imported in the Event Registry.

The Event Registry currently contains 20 million articles which were collected by Newsfeed since December 15th. On average there are about 72.000 articles added every day.

4 Pre-processing steps

In order to identify events from articles we first perform some processing steps on the individual articles. The tasks that are performed on the articles include (a) detection and disambiguation of entities and topics, (b) identification of mentions of dates, (c) detection of mention of explicit event location, (d) identification of similar articles in other languages, and (e) detection of article duplicates. Results of all the tasks are added as meta-data to individual articles and then used in the next parts of the processing pipeline.

4.1 Semantic annotation of article text

Semantic enrichment of text is provided by services developed in WP2 and WP3. For the purpose of event extraction, the most important part of enrichment is the detection of mentioned concepts [D2.1.1, D3.1.1] and the ontology based word-sense-disambiguation [D3.2.1]. When extracting information about an event, the detected concepts (named entities as well as non-entities) enable us to understand what the event is about and where it occurred.

Majority of annotations identified by the service are non-entities (around 90%-95%) such as “war”, “demonstration”, “product”, etc. The rest of the annotations are entities that represent people, locations or organizations. Because the type of the concept is not determined by the annotation service we have implemented our own type detecting method. Since the annotation service uses Wikipedia as the knowledge base we use the concept’s corresponding DBpedia resource when inferring the concept type. We identified a set of indicative relations for each concept type and use them to determine the type. For locations, for example, such relations are “populationTotal”, “timeZone”, “areaTotalKM”, and others. In cases when the concept’s DBpedia page does not exist or we are not able to infer the type, we consider the concept to be a non-entity.

Semantic annotation of articles is a resource intensive process. Based on the hardware availability our service is currently able to annotate about 80% of articles. Statistics show that on average an annotated article mentions 65 non-entities, 3.5 locations, 1.5 persons and 1.5 organizations.

4.2 Cross-lingual article matching

The Cross-Lingual Similarity Service (CLSS) (described in [D4.1.1]) is used for providing cross-lingual information related to individual articles. The service can compute an approximate similarity between English, German, Spanish and Chinese news articles. Computation of the cross-lingual similarities is based on an aligned set of basis vectors obtained by one of two methods: latent semantic indexing (LSI) and a generalized version of canonical correlation analysis (CCA) by using an aligned multi-lingual corpus. Given a newsfeed article as an input, the service returns ids of top 10 most similar articles for each language. When computing the most similar articles, the service compares the article with other articles that have been published in the last 24 hours. The limitation of a one-day time window is acceptable for event detection since an event is typically reported for a short time period. Here is an example of an output of the service in JSON format:

```
{"id":66701562,
 "similar_articles_spa": [{"id":66701512,"sim":0.1364}, …],
 "similar_articles_deu": […],
 "similar_articles_fra": […],
 …
}.
```


4.3 Extraction of date references

News articles frequently mention dates. They can represent the time when the event happened or a date that is otherwise related to the content of the article. In order to detect date mentions in the articles we implemented a module that relies on a set of regular expressions that have been defined for each language separately. Based on the article language, the appropriate set of regular expressions is selected and used to identify date occurrences. Since events can either occur at a single point in time (Bombing attack in Syria) or across a certain time period (starting and ending date, e.g. 27th July – 12th August for London Summer Olympic Games) we developed a separate group of expressions for detecting date ranges and a group for expressions for detecting single date occurrences.

An example of a single regular expression for detecting a date range from English articles is provided below:

```
\b(?P<day1>\d{1,2})(?:st|nd|rd|th)?(?:(?:\sof\s)|(?:-|–|—)|(?:\.\s?)|\s)?
(?P<month1>(?:jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)[a-z]*)\.?(?:(?:\s? (?:-|–|—
|,|')?\s?)?(?P<year1>\d{2}|\d{4}))?(?:\s?(?:to|through|until|and|(?:-|–|—))\s?)
(?P<day2>\d{1,2})(?:st|nd|rd|th)?(?:(?:\sof\s)|(?:-|–|—)|(?:\.\s?)|\s)?
(?P<month2>(?:jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)[a-z]*)\.?(?:(?:\s? (?:-|–|—
|,|')?\s?)?(?P<year2>\d{2}|\d{4}))?\b
```

The expression can identify date mentions in the form of “15th of September 2003 to 20th of November 2012” including numerous variations. Similarly, an example of a regular expression for matching a single date occurrence looks like:

```
\b(?P<month>(?:jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)[a-z]*)\.?(?:(?:\sthe\s)|(?:-|–|—
)|\s)(?P<day>0?[1-9]|[12][0-9]|3[01])(?:st|nd|rd|th)?(?:,?\s'?(?P<year>\d{2}|\d{4}))?\b
```

This expression can be used to identify date occurrences such as “September 22, 2013”.

As can be seen from both examples, both types of expressions also match incomplete date references (such as September 2012 or 15th of July). We have found that in most detected date references, the year is not specified. For this purpose we implemented a simple imputation algorithm that is able to guess the correct year based on the information when the article was published. The algorithm starts by constructing three possible complete dates from the incomplete date by using the year value as the current year, previous year and next year. It then compares the published date of the article with the three dates and chooses the one that is the closest in time. Consider the following example. The article published on Jan 15th 2014 mentions November 1st. Three constructed candidates are 2013-11-01, 2014-11-01 and 2015-11-01. By checking the time difference between 2014-01-15 and the three candidates, the algorithm would choose the 2014-11-01 as the correct date. The implicit assumption made by the algorithm is that the incomplete date reference has to mention the date that is closest in time. Based on our empirical study we found this assumption to be valid in most cases. A more accurate approach would be to analyse the sentence mentioning the date and detect if it is written in the past or future tense. The disadvantage of this approach is that it relies on the availability of the appropriate linguistic tools for each language.

Another problem that occurs when extracting date occurrences is ambiguity caused by different formats in which dates can be written (m/d/y in U.S. vs. d/m/y in Europe). When one of the digits is above 12 there is no ambiguity. In other cases we have to, however, mark the date as ambiguous since we cannot assure the correct date interpretation.

The procedure for identifying the date occurrences works in two steps. In first step we search for possible date ranges. Detected dates, if found, are marked as seen, before we run the next step in which we search for individual date mentions, which were not yet detected in the first step. All detected dates are normalized into the form YYYY-MM-DD. If year or day information is missing, the corresponding part is left empty. Information about the detected dates is then stored in an array of JSON objects. Here is an example of a JSON object for a detected date:

```
{
```

```
        "date": "2014-05-03",     // normalized form of the detected date
        "imp": false,       // did we impute the missing year?
        "amb": false,       // is the date ambiguous – could we possibly incorrectly identify day and month
        "freq": 1,          // number of times the date was detected
        "posInText": 45, // the index of the first character in text where the date is mentioned
        detectedDate: "May 3rd, 2014"    // the detected date as it appeared in text
}
```

4.4 Extraction of explicit event location

Location where the event described in the article occurred can be mentioned anywhere in the article. There is, however, a particular format that is sometimes used for specifying date and location of the event. This format is called a dateline and is a brief piece of text that is included at the beginning of the article. A typical article dateline might look like this:

IRVINE, Calif. (AP) -- President Obama will speak at the University of California, Irvine.

In this case, Irvine, California is the place where the event will occur and AP is the newswire that is the source of the story. Since locations that are identified in the dateline are crucial for determining event location we want to be able to identify them so that this information can be used in later steps when we need to determine event location from a group of articles.

To detect location in the dateline we simply apply the following procedure. First we detect if the article potentially contains a dateline. We perform this test by trying to identify a particular text pattern in the first 50 characters of the article. Datelines are separated from the rest of the articles using an *em* dash or a similar character so we use a particular regular expression to see if a dateline can be found. If it is found we check if it mentions a city. We achieve this by inspecting the annotated concepts mentioned in the dateline and identifying the ones that represent cities. To determine whether a concept represents a location we use the GeoNames database. If a city is mentioned near the beginning of the article (it might not be the first token since the newswire is sometimes mentioned first) we use it as the explicitly defined location of the mentioned event.

4.5 Detection of article duplicates

By analysing the articles from Newsfeed we found that it provides many duplicated articles – that is, articles that contain the same or almost the same content as some other article. Duplicated articles are often generated when a news source corrects or extends their own article, or by a news source that simply copies an existing article from a different publisher. Here is an example of two such articles:

Article 1:

Title: Obama sends good wishes to Jay Leno

Body: Jacquelyn Martin, File/Associated Press - FILE - This Aug. 6, 2013 file photo shows President Barack Obama talking with Jay Leno during a commercial break during the taping of his appearance on "The Tonight Show with Jay Leno" in Los Angeles…

News source: Washington Post

Article 2:

Title: Obama sends good wishes to Jay Leno

Body: FILE - This Aug. 6, 2013 file photo shows President Barack Obama talking with Jay Leno during a commercial break during the taping of his appearance on "The Tonight Show with Jay Leno" in Los Angeles…
News source: Fort Worth Star-Telegram

In principle, there is no issue in having multiple copies of the same article in the event detection system. There are however some parts of the pipeline that are negatively affected by such duplicated content, especially if the duplicates are numerous (one of our tests identified articles that had more than 100 duplicated articles). In article clustering, for example, that will be described in the next section, the articles are grouped by their similarity using a clustering method. In case a cluster contains multiple copies of the same article, this can significantly skew the cluster's properties – its centroid, variance and the medoid article. Additionally, the duplicated articles can overestimate the support for cluster merging in the cross-lingual cluster merging (Section 5.2).

In order to determine if an article is a duplicate or not we implemented the following procedure. Given a new article, we first compute bag-of-words using the article body. We then pick 10 words with highest term frequency (not including stop words). We sort these words alphabetically and compute a hash code using them. Using the hash code we can then find a list of other documents that had the same 10 most used words. These documents represent candidates that need to be checked to determine if the new article is a duplicate. Since news sources sometimes add small modifications (such as adding the title of the news source) we do not want to do a word-by-word comparison. Instead, we use the bag-of-words representations and compare the individual word frequencies. If the discrepancy between two articles is less than 5 words then we mark the article as a duplicate and further steps in the pipeline can take this information into account.

5 Event Construction

After performing different information extraction steps on the individual articles we now come to the task of identifying events from groups of articles. First we will describe the clustering algorithm that we use to identify groups of articles that describe the same event. Next we will present methods that are used to extract from these groups different information about the events.

5.1 Article clustering

Article clustering is performed using NewsCluster, a web service for microclustering a stream of news documents, based largely on the approach of C. Aggarwal et al. [AY06, AY10, AHWY03]. The service maintains a set of documents partitioned into a number of (relatively small) clusters. Old documents are periodically discarded, thereby keeping the cluster structure focused on the current state of the data stream.

Clustering is the task of arranging documents into groups based on some perceived notion of similarity, and is often an important building block of applications. The output of the clustering process can be seen as representing a "higher-level" view of the underlying data stream and is then the basis for further processing.

The fact that data comes from a stream adds a few specific requirements to the clustering task. In particular, the clustering algorithm must allow the partition of documents into clusters to be built and updated incrementally as new documents appear in the stream; it cannot make multiple passes through the entire stream, as there is only enough storage space for documents from a relatively limited time-based window. Additionally, we assume that the contents of the stream may change over time and the set of clusters should gradually be able to adapt to that, allowing the introduction of new clusters and the splitting/merging of existing clusters.

The most important consideration for a stream-based clustering algorithm, however, is that it must on average be able to process at least as many documents per unit of time as there are new documents coming from the stream in the same unit of time, otherwise it won't be able to keep up with the incoming data. If at some point the volume of incoming data grows high enough, parallelism will need to be employed. In the present version of the NewsCluster service, we focus on the scenario where the rate of incoming data is high enough that parallelism is necessary, but not so high that it would be necessary to distribute the processing over multiple computers. Thus, our aim is to present an efficient multi-threaded clustering approach that runs on a single computer but makes good use of its multi-CPU and multi-core facilities.

Architecture

Our implementation runs as a web service, listening for HTTP requests which provide new documents that arrived from the source stream and now need to be added to our clustering-related data structures. The clustering service processes such requests asynchronously; it reports any changes in the assignment of documents to clusters by eventually making HTTP requests to one or more listeners, reporting such things as assignments of documents to clusters, outcomes of cluster splits/merges, and changes in cluster medoids.

In addition to handling the requests for adding a new document into the clustering, the service also performs maintenance operations from time to time. This includes discarding old documents and clusters, as well as saving its data structures to disk. The service keeps all its data in main memory during normal operation, but saves it to disk periodically so that it can be restarted without much loss of data in case it crashes. In earlier versions of our system, cluster merging was also performed periodically as a maintenance operation, but now it's included as a post-processing step following each addition of a new document.

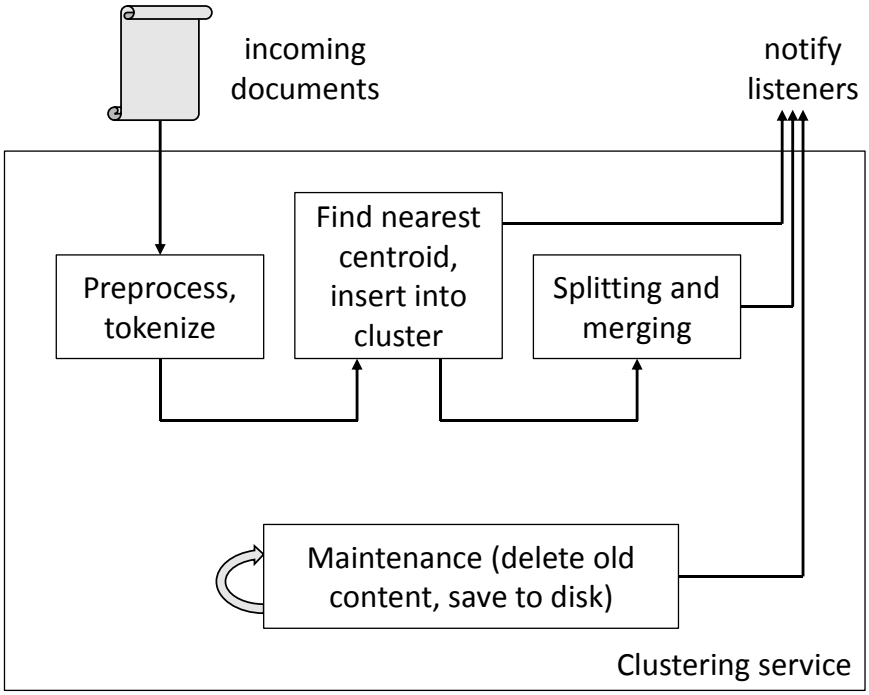

Figure 2 Architectural overview of NewsCluster web service

If we look at it more closely, the incoming HTTP request is expected to provide the following information:

- a unique document ID: this can be an arbitrary string, but if NewsCluster finds that a document with the same ID already exists in its database, the new document provided by the request is ignored;
- the full text of the document;
- optionally, a title of the document, but this is used only for diagnostic purposes and does not participate in the feature construction process;
- the language of the document: NewsCluster processes documents of each language separately from those of other languages, meaning that they will not end up in the same clusters, and the feature space (including document frequencies of features etc.) is constructed separately for each language;
- optionally, the request may also contain a set of ⟨*concept ID*, *weight*⟩ pairs; such concepts are treated as additional features in the document representation (on which see more below), on top of the features which NewsCluster obtains from the full text using the bag-of-words principle.

Document representation

For the purposes of clustering, each document is represented by a TF-IDF feature vector. First, the text of the document is split into words using a slightly adapted version of the Unicode word breaking rules [Dav12]. Next, stopwords are removed if a stopword list for the language of the document is available; currently, the NewsCluster service has stopword lists for English, German, French, Spanish, Italian, Portuguese, and Dutch.

The remaining words are used as the basis of a bag-of-words representation: the document is represented by a sparse feature vector containing one component for each word that appears anywhere in the document set so far. The value of the component corresponding to word t in the feature vector representing the document d is defined as $TFIDF(t, d) = TF(t, d) \cdot IDF(d)$, where $TF(t, d)$ is the *term frequency* (the number of occurrences of term t in the document d) and $IDF(t) = \log(N / DF(t))$ is the *inverse document*

frequency, obtained from N (the total number of documents currently maintained in the collection) and *DF*(*d*) (the *document frequency* of *t*, i.e. the number of documents in which *t* occurs at least once).

Optionally, the NewsCluster service can use *n*-grams (sequences of up to *n* adjacent words) as features, in addition to individual words.

If the incoming request contained the optional set of ⟨*concept ID*, *weight*⟩ pairs, a TF-IDF vector is also formed from these pairs. In this case, the *weight* is used as the TF, and a DF (and hence IDF) is computed for each concept just as it would be for features arising from the full text.

This results in two feature vectors, $\mathbf{x}^t$ arising from terms and $\mathbf{x}^c$ arising from concepts; they are concatenated into the final feature vector of the document: $\mathbf{x} = \langle \mathbf{x}^t, \mathbf{x}^c \rangle$. Each part is normalized separately such that after normalization, $||\mathbf{x}|| = 1$ and $||\mathbf{x}^c|| = w\ ||\mathbf{x}^t||$. Here, *w* is a user-provided parameter (*conceptWeight* on the command line) that defines the relative importance of concepts to terms in the final feature vector.

Internally, the service does not store the normalized TF-IDF vector of each document, but its TF vector (from which the TF-IDF vector can easily be generated on the fly as needed) and the original full text of the document.

The underlying clustering algorithm

Our approach is largely based on that of Aggarwal and Yu [AY06, AY10]. The main principle that we use in assigning documents to clusters is to simply assign each incoming document to the cluster whose centroid is the closest (i.e. most similar) to the document. (Note that the same idea is used by popular off-line clustering algorithms such as *k*-means, except that *k*-means then performs multiple passes through the data, recalculating centroids and reassigning documents in each pass, which we cannot afford to do in an on-line setting.) For the purposes of computing centroids, measuring similarity etc., we use the bag-of-words representation (also known as the vector space model) to represent each document with a TF-IDF feature vector, normalized to unit length. After a new document is assigned to its nearest cluster, we consider splitting that cluster or merging it with another cluster. Figure 3 presents the pseudo-code of our approach.

```
Input: d – a document to be added to the clustering
1       If d appears to be a duplicate of a document that is
        already in the clustering, stop processing d and ignore it.
2       Compute x, the TF-IDF vector representing d, normalized to unit length.
3       Find the cluster C whose centroid is the closest to x (in terms of cosine similarity).
        If the cosine similarity between x and this centroid is below a certain threshold,
        establish a new micro-cluster containing only document d (and skip the remaining steps).
4       Add d into C, updating its various aggregated statistics (such as the centroid).
5       If the splitting conditions are met, consider splitting C:
6           Find the most promising split of C into C' and C''.
7           If this split is better than the original C,
            replace C with C' and C'' in our clustering data structures.
8       Else, if the merging conditions are met, consider merging C with another cluster:
9           Find a few clusters whose centroid is closest to the centroid of C,
            in terms of cosine distance.
            For each such cluster C':
10              Let C'' = C ∪ C'. If this cluster is better than keeping C and C' then
11                  Replace C and C' by C'' in our clustering data structures and break.
```

Figure 3 Pseudo-code describing an overview of our clustering approach.

Initial phase

Initially, the service starts in a "pre-clustering" state, during which it only accumulates incoming clustering without trying to assign them to any clusters. When a certain number of documents of a given language has been accumulated (controlled by a command-line parameter, defaulting to 1000), we use a hierarchical bisecting *k*-means (i.e. 2-means) algorithm to obtain an initial partition into clusters, as shown in the following pseudo-code:

```
start by placing all documents into one cluster;
while the number of clusters is less than the maximum initial number of clusters do:
    choose the cluster C with the maximum variance from among the current clusters;
    use bisecting k-means to split it into two sub-clusters C1 and C2;
    if neither C1 nor C2 is too small, replace C with C1 and C2 in our current partition;
```

At this point, all the listeners are also informed about the initial assignments of documents to clusters. From this point onwards, the service enters its normal mode of operation, which is described in Figure 2**Error! Reference source not found.**.

After the addition of a new document, the cluster is considered for splitting. The conditions for this are that it must contain a sufficient number of documents and that sufficiently many additions must have occurred since the last time it was considered for splitting. If these conditions are met, we try to split the cluster into two sub-clusters using bisecting *k*-means. We use a variant of the Bayesian Information Criterion [PM00, MG09, LE12] to decide whether to accept the new split or not. If the cluster is split, all the listeners are notified of the new cluster memberships for all the documents affected by the split.

Background operations

In the background, the service periodically performs maintenance tasks: saving all data to disk, and removing old clusters and documents.

Removal of old documents is performed when the total number of documents for a given language exceeds a user-specified threshold. In this case the oldest few clusters (and all documents belonging to them) are deleted until the number of documents drops below the required threshold. For the purposes of this operation, the age of the cluster is defined to be half-way between the average age of all documents in the cluster and the age of the most recent document in the cluster.

Cluster statistics

Following the approach of **Error! Reference source not found.**, we maintain a set of statistics for each cluster and update them incrementally whenever the cluster changes. This includes the sum of the feature vectors of its documents, the square of this sum, per-feature variances, and a few other statistics. Our implementation also supports the option of allowing different documents to have different weights, where the weight of the document decays exponentially as the document ages, as suggested by **Error! Reference source not found.**; however, this has not been found to be useful in our applications, so we currently set all weights to 1 in practice.

The fact that we're dealing with an ever-changing stream of documents requires us to introduce a few approximations. For example, in principle, whenever a new document is added to the clustering, or an old document discarded, the document frequency (DF) of any term from that document changes; as a result, the inverse document frequencies (IDF) of such terms also change, and the value of their corresponding features change accordingly, in the TF-IDF vector of any document containing any such term, as well as in the centroid of any cluster containing any such document. So theoretically, the feature vectors of most documents and the centroids of most clusters would have to be recalculated whenever a document is added to or removed from our collection. Since the costs of such an update would be prohibitive, we introduce an approximation. First, instead of storing TF-IDF vectors of individual documents, we store TF vectors instead. The IDF can be applied on the fly whenever needed, e.g. when we wish to (re)calculate the centroid of a cluster. Secondly, when a document is added to a cluster, we only recalculate the centroid of

that cluster, but not the centroids of other clusters; those will get recalculated sooner or later when some new document is added to them.

Splitting

To save time, a cluster is only considered for splitting (line 5 of the algorithm listing) if it is sufficiently large and if sufficiently many additions to it have been made since the last time it has been considered for splitting. The main idea during splitting is to project all members of the cluster onto a line and divide them into two groups depending on whether their projection was left or right of the projection of the centroid. This is repeated several times; in the first iteration, we project onto the principal component of the original cluster; in each subsequent iteration, we project onto the line through the centroids of the two groups from the previous iteration. In line 6, the best of these splits is chosen based on minimizing the variance; in line 7, a Bayesian Information Criterion [PM00] is used to decide whether to actually accept the split.

There is in fact another type of split, not explicitly shown in the pseudo-code: each document also has a timestamp, and if the standard deviation of timestamps of documents in the cluster exceeds a given threshold, we force a split based on the timestamps (into an “older” and a “newer” sub-cluster). This is to ensure that clusters are compact in the time axis as well and that the old clusters can eventually be completely discarded.

Merging

For merging, we similarly only consider the cluster for merging (in line 8) if enough additions have been made to it since the last time it was considered for merging. Merging makes sense if two clusters are similar, e.g. as measured by the cosine similarity of their centroids. The problem here is that while the feature vectors of individual documents are sparse (i.e. they have relatively few nonzero components), the centroid of a cluster is usually fairly dense. Thus, computing a cosine between two centroids involves a dot product of two dense vectors, which is time-consuming. We resort to an approximation again: for the purposes of step 9, we temporarily make the centroid of C sparse by setting all its components to 0, except the thousand components that were highest in terms of absolute value. This substantially preserves the direction of the vector but makes the computation of dot products cheaper. In line 10, we use Lughofer’s ellipsoid-overlap criterion **Error! Reference source not found.** to decide whether to accept the merge; additionally, the merge is always accepted if the cosine between the two centroids exceeds a user-defined threshold.

The duplicate-detection step in line 1 is in practice somewhat custom-tailored to the particular document stream we’ve been using in our applications so far. This stream collects news articles from numerous websites, many of which turn out to be reprints of agency articles with few or no modifications. We declare an article to be a duplicate if an existing article has the same title (modulo capitalization and whitespace) and a sufficiently similar TF-vector. Such duplicate articles are simply discarded, rather than added into any cluster.

Microclusters

Stream clustering approaches generally establish new clusters in two ways: by occasionally splitting an existing cluster, or by simply starting a new cluster that initially contains just one document (the document that’s currently being added to the clustering), but which will hopefully grow later as further similar document appear in the stream. The idea is to start a new cluster if none of the existing clusters has a centroid that is sufficiently similar (in terms of the cosine measure) to our new document, i.e. if the cosine similarity of the nearest cluster is below a certain user-defined threshold. Such a mechanism helps keep existing clusters more pure (because they are not forced to accept a not-very-fitting new document just because all the other clusters were even less suitable for the new document), but on the other hand it poses the risk of establishing a large number of very small clusters, and many of these might be regarded by the user as basically noise.

The concept of *microclusters* is our approach to mitigate this problem. If a new document isn't close enough to any existing cluster centroids, we create a new cluster for it, but this cluster is initially flagged as a microcluster. Internally we treat it much like any other cluster; it may receive new documents and it may get involved in merges, but not in splits. However, we do not send any notifications regarding microclusters to our listeners. When the number of documents in a microcluster exceeds a user-defined threshold (e.g. 5 documents), either through a merge or through the addition of a newly-arrived document, the microcluster flag is cleared and the cluster is thenceforth treated as a regular cluster. At that point we also immediately notify the listeners of the membership and medoid of the newly-regular cluster.

The purpose of this mechanism is to avoid bombarding the listeners with noisy information about tiny single-document clusters; the listeners will only start receiving information about a document once it becomes clear that a cluster of similar documents has actually emerged around it.

Multi-threaded clustering

A single-threaded, non-parallel implementation of the approach described in the previous section is fairly straightforward. The program simply processes requests (to add a new document into the clustering) sequentially in an endless loop, finishing one request before moving on to the next one. Occasionally it can perform maintenance tasks (such as saving to disk, and discarding old clusters and documents) in between handling two requests.

Following the well-known principle that optimization should focus on those parts of the program in which the largest amount of time is spent, we timed the single-threaded implementation while performing 106 article additions. It turns out that approx. 54% of the time was spent in step 3, calculating the cosine similarity between the new document and the centroids of all existing clusters; 43% of the time was spent in step 9, calculating the cosine similarity between cluster centroids; all other steps together account for the remaining 3% of the time. Thus, it is clear that parallelization needs to focus on steps 3 and 9.

Another important consideration when designing a multithreaded solution involves the use of shared data structures. If a thread needs to modify some shared data structure, it requires exclusive access to it; that is, other threads shouldn't be using the data structure at all while it's being modified, even if they are content with read-only access to it. At the same time, we want to minimize the amount of time that threads spend waiting for some other thread to relinquish its exclusive lock on a shared data structure.

In the algorithm from Figure 3, modifications of shared data structures occur in step 4 (adding a new document to a cluster), step 7 (performing a split) and step 11 (performing a merge). Another modification of shared data, which is not as readily obvious from that pseudo-code listing, occurs when creating a feature vector **x** corresponding to the new document *d*: a shared hash table containing the document frequencies of all terms needs to be updated, and new words might need to be added into it (if *d* contains some words that have until now never been seen in our stream of documents). Similarly, the duplicate detection in step 1 uses a hash table of document titles and needs to add the new document's title to it (if it didn't turn out to be a duplicate).

A somewhat naïve idea would be to assign each newly incoming document to one of several threads, and this thread can then execute the algorithm from Figure 3. The thread could somehow lock the cluster while it's being accessed, but we can quickly see that this is unsatisfactory. Our application calls for a large number of small clusters; eventually there will be thousands, possibly tens of thousands of clusters, and we don't want to require a thread to have to acquire and release tens of thousands of locks during step 3 or step 9.

This sort of fine-grained locking also has other inconvenient aspects. It is easy to imagine a nightmare scenario in which one thread is trying to compute the cosine between the centroid of cluster *C* and a new document; another thread has already decided that it wants to insert a different new document into *C* and now wants to update its centroid; and yet another thread is trying to split *C* into two clusters, or merge it with some other cluster.

If we don't want to have to deal with cluster-level locking, we have to accept that no thread may modify clusters (which includes adding or deleting them) while some other thread is looping through them in step

3 (or step 9, for that matter). Thus, while any thread is modifying the shared data structure (i.e. performing steps 4, 7, or 11), no other thread may be reading this data (i.e. performing step 3 or 9 – but as we saw earlier, that's where our threads will be spending 97% of their time). For nearly every new document (unless it was discarded in step 1 as a duplicate), we'll eventually have to add it to a cluster (in step 4); before our thread can do so, it must wait for all other threads to reach the end of step 3 or 9, and all those threads must then stop and wait for our thread to finish step 4. (A similar consideration applies to steps 7 and 11, but those are performed more rarely.) Clearly this has the potential to lead to an undesirable amount of waiting.

We can rewrite the pseudo-code of Figure 3 in a way which emphasizes the alternation between stages which only read shared data and stages which need to modify shared data:

R1. Check if *d* is a duplicate; split it into words and prepare a TF-vector, except for any new words that aren't in our shared word table yet – these should be kept in a separate list.

M1. If R1 found *d* to be a duplicate, discard it and stop. Otherwise, add its title to the shared hash table (for future duplicate detection) and add any new words it might have contained into the shared word table; this is also the time to finalize its TF vector and update the document frequency counts in the shared word table.

R2. Compute the TF-IDF vector of our document *d* and find the cluster *C* with the nearest centroid.

M2. Insert *d* into *C*, updating *C*'s centroid and other aggregate statistics.

R3. Consider splitting *C* or merging it with other clusters, if appropriate. Do not modify any shared data structures; if the decision to split *C* is made, record what the new clusters *C*' and *C*'' would be; likewise, if the decision for a merge is made, record which cluster it would merge with and what would the resulting cluster be like.

M3. Update the shared data structures to reflect the splits or merges decided upon in step R3.

The key observation here is that there is no reason why all these steps should be performed by the same thread, as long as we maintain a small "context" data structure which helps threads keep track of the request as it passes through the stages.

Note also that stage R2 basically corresponds to step 3 of Figure 2, while step 9 is included within R3. All the M-stages are cheap, quick operations. Thus, in our multithreaded clustering implementation, we have a single main thread which performs the M-stages for all requests; the other threads are worker threads and perform R-stages. The requests pass between the main thread and the worker threads until they are complete, as shown in Figure 4.

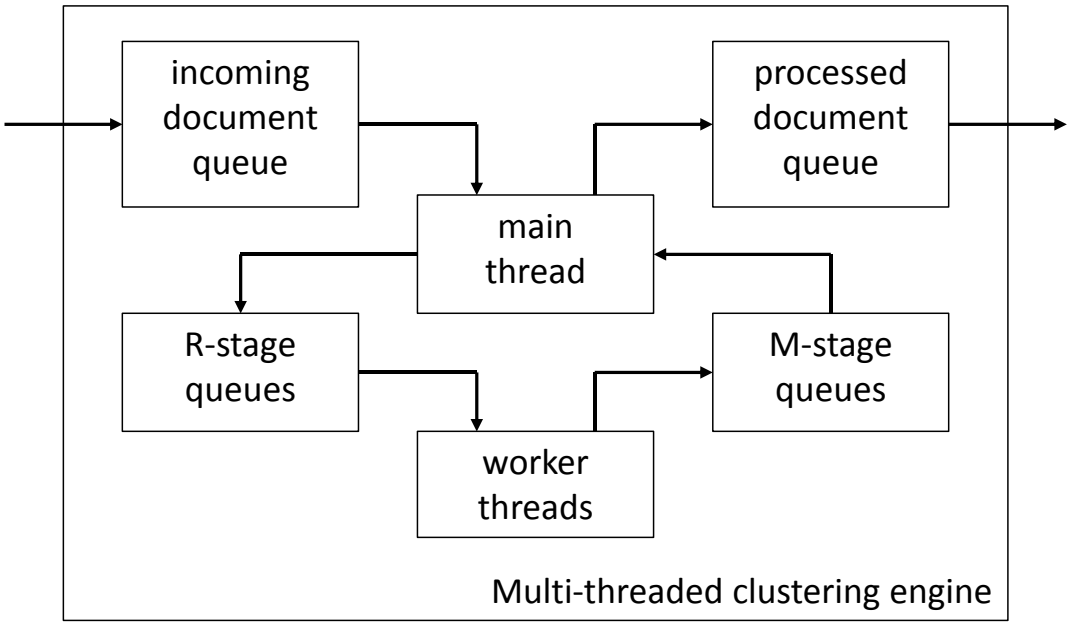

Figure 4 An overview of our multi-threaded clustering approach, showing the flow of requests through the system

Thus, each worker thread runs in an endless loop in which it takes a job from a queue, performs the next stage (which will be either R1, R2, or R3), and deposits it into a different queue. The main thread, on the other hand, assigns jobs and periodically blocks the worker threads, performs the M-stages, and restarts them. The following is a simplified pseudo-code of the main thread:

1 Wait a set amount of time (e.g. 1 second).

2 Set a flag which tells the worker threads to stop after they finish their current job.
Wait for all the worker threads to finish their current job.

3 Perform the M-stages of all the requests which are currently in progress.

4 Return to step 1.

Thus, most of the time the main thread sleeps (step 1) and lets the worker threads perform the R-stages of various requests. Every now and then, the main thread performs a barrier synchronization (step 2), stopping all the workers after their current job is done. Thus, at step 3, all workers are asleep, so the main thread can modify shared data.

Occasionally (e.g. once per hour), the main thread stops issuing any new R1-stage jobs to worker threads and waits for all partly completed requests to fully complete (i.e. all the way through M3). At this point, there are no partly completed requests in the system, so this is a good time to perform periodic maintenance tasks such as saving the data to disk. After this, normal processing can resume.

Since step 3 performs the M-stages of all currently open requests in one place, it is in a good position to coordinate their sometimes conflicting ideas as to what should be done. First, it performs the M1-stages, as these cannot conflict with other requests. Next it performs any splits and merges (M3) requested by recently completed R3 stages; while doing so, the main thread keeps track of which clusters have been split or merged, and ignores split/merge requests that involve clusters that have been affected by a previously processed split/merge request. Finally, the main thread performs M2-stages, inserting documents into the clusters requested by recently completed R2 stages; if any such cluster has been split, the main thread checks the centroids of the two sub-clusters to see which is closer to the document.

Note that this approach means that each document must pass through three iterations of the main thread before it is fully processed. Thus, if e.g. step 1 of the main thread takes 1 second, it will take at least 3 seconds before the document is processed. To use an analogy from networking, we have achieved high bandwidth at the price of also having high latency.

Future work related to clustering

The multithreaded clustering approach presented here achieves a considerable degree of parallelism, allowing it to fully utilize a typical present-day multi-CPU multi-core PC. A further form of parallelism, not mentioned above but present in our application, comes from the fact that our stream of documents is multilingual and each language is processed separately from the others, thus each language can have its own main thread and set of worker threads.

The work presented here could be extended in several directions. For example, this approach could be applied to non-textual data with only minor modifications. The key idea behind our multithreaded approach is the multi-stage processing, concentrating all modification of shared data into one thread and using barrier synchronization for the worker threads; and there is nothing text-specific in this.

The computation of cosine similarities (which is where the algorithm still spends most of its time) could be speeded up by the means of random projections [CW12] into a limited (and fixed) number of dimensions. In our preliminary experiments, projecting our feature space into 1000 random projections resulted in almost no distortion (in terms of which centroid is closest to which document). After such a projection, documents and centroids become fixed-length dense vectors, and cosines can be computed very efficiently by making use of the SIMD capabilities of modern processors (as in various numerical linear algebra libraries).

Another possible direction for further work is to replace the current flat clustering with a hierarchical one. This can be desirable for some applications, and it would also speed up the assignment of documents to clusters if this is done in a top-down fashion instead of examining all the clusters.

Finally, at some point the rate of incoming documents may well grow beyond what can be processed by a single computer, so it would be interesting to investigate clustering approaches based on distributed computing.

5.2 Cross-lingual cluster matching

The clustering service uses article text as the main feature for finding articles that discuss same event. As such, the service has to effectively run a separate instance of clustering for each analysed language. Since articles in different languages can discuss the same events we want to be able to identifying clusters regardless of their language that talk about the same event and represent them using the same event ID.

In order to identify different clusters that discuss the same event we build a classification model that can predict if two clusters are about the same event or not. As with all classification methods we first need to construct a set of features that describe the two tested clusters. These features need to be selected so that their values will be as discriminating as possible in separating between the pairs of clusters that describe the same event and the pairs of clusters that describe different events. The features that we constructed are mainly built using the following information:

1. The most similar articles as computed by the Cross-Lingual Similarity Service. The service computes for each article the list of 10 most similar articles in all languages.
2. The annotated concepts in the articles. Since the concepts are language independent we can find common concepts in the articles in a cluster and use them to compare clusters between each other.
3. Dates of the articles in a cluster.

Features based on Cross-Lingual Similarity Service (CLSS). Given two clusters c_a and c_b we use similar articles obtained using the CLSS as follows. For each article a in c_a we check how many of the a's similar articles are in cluster c_b. In the same way we compute a match count also for each article in cluster c_b. The final score is then the sum of the two match counts normalized by the number of articles in each cluster (in order to remove the importance of cluster size). This score is an indication of how similar the two clusters are according to CLSS. The intuition behind it is that if two clusters are about the same event, the articles in one cluster should point to articles in the other cluster and the computed normalized score should be high.

Additional feature that we compute using CLSS information is also the average similarity score. For each similar article, CLSS also provides a similarity score which is a weight on the [0, 1] interval. The average similarity score is computed as an average score of the articles that point from c_a to c_b and vice-versa. If this score is high this indicates that articles in once cluster are very similar to articles in the other cluster.

Features based on the concept annotations. The articles in each cluster contain annotations provided by the annotation service. These annotations, consisting of named entities as well as non-entities, are represented with URIs that are language independent. The English word "employment" will be, for example, represented with the same concept URI as Spanish word "empleo". Based on the annotations in each article in the cluster we can compute a weighted list of top annotations for each cluster. Using the two lists we can compute features such as cosine similarity between lists (which takes into account the actual weights of the annotations) as well as Jaccard index (which only computes the overlap, ignoring the actual weights). These two types of scores can also be computed separately for each concept type (people, organizations, locations and non-entities). All the computed scores are used as learning features.

Features based on the article dates. Dates of the articles are also an important feature when deciding if articles are about the same event or not. We can assume that if the time difference between the articles in the two clusters is large, they are likely about a different event. One learning feature that we therefore extract is the time difference between the average article dates in both clusters. Additionally, we also identify date references mentioned in the articles in each cluster and use the number of overlapping date references as another learning feature.

Other features. Beside the three main groups of features we also use as features other relevant properties. One such property is if both clusters agree on the location of the event described in the articles. Another feature computes the agreement in the topics (categories) that are described in the articles. Additional features are computed based on the quality of the clusters (cluster variance, average cosine similarity to centroid, etc.).

For each pair of clusters we can compute the set of features that we have just described and based on their values the classification model predicts if the given pair describes the same event or not. The learning algorithm that we used for building a classifier is a linear SVM. We've manually built a dataset of 150 learning examples containing positive and negative examples of cluster pairs. By training an SVM model we were able to achieve 88% classification accuracy using a 5-fold cross-validation. The most relevant features for classification were found to be features related to Cross-Lingual Similarity Service. We believe that the accuracy of the model will be additionally improved once the annotation service will annotate all the imported articles. Currently we found many tested clusters without any annotated articles which reduced the relevance of the features related to annotations.

5.3 Event formation and event information extraction

At this point in the pipeline we have already identified groups of articles that describe the same event as well as potentially found multiple clusters in different languages that should be represented as a single event. The next step is to create an actual event using the clusters and to extract from the articles structured information about the event.

For the purpose of representing an event we have created a data structure that points to one or more clusters, and clusters are the data structures that point to individual articles (see UML diagram on Figure 5 for details). One could argue that semantically the clusters are somewhat redundant – that articles should simply be assigned directly to an event. While this is true, we find that keeping the clusters significantly simplifies the event maintenance. Each created event is assign a unique ID. The IDs are numbers that increase monotonically with each new event and for a given event always stays the same.

It is important to note that the events are highly dynamic structures which evolve a lot – at least while new articles about an event are being written. Each time a new article is received it is assigned to a cluster. Since the clustering works in online mode, the clusters change significantly over time – they can merge or split into

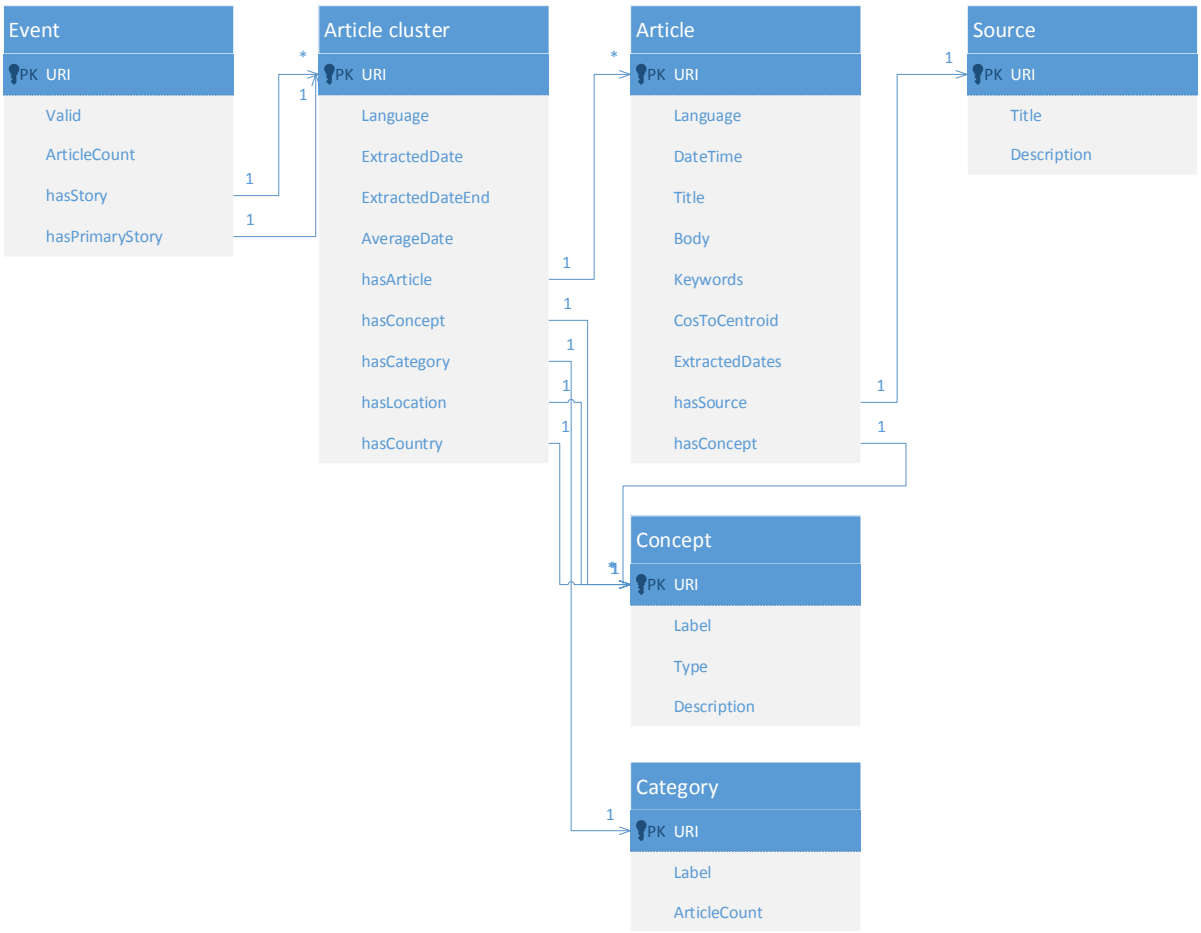

Figure 5 UML diagram of the core data structures used to represent event information

two clusters and individual articles can be reassigned to a different cluster. All of these changes need to be propagated also on the level of events. Changes and the actions they trigger are as follows:

1. A new article is added to an existing cluster. Properties of the event containing the cluster are updated.
2. An existing article is reassigned to another cluster. Properties of the source and target events are updated.
3. An existing cluster is split into two clusters. The larger cluster stays assigned to the current event. A new event is generated using the smaller cluster. Properties of both events are updated.
4. Two existing clusters are merged into one cluster. The smaller cluster is merged into the larger cluster. If the event that owned the small cluster does not contain any other clusters, the event is marked as invalid. Properties of both events are updated.

Updating of event properties requires re-examination of the articles assigned to the event and potentially changing the values of different properties. Some changes, such as article count and validity of an event, are atomic and require no additional explanation. Most event properties are however non-trivial and require a proper description.

Event title and text snippet

Event title is one of the basic properties that needs to be extracted for each event. Event title should be a short text that summarizes what the event is about. If the event is covered in different languages then the title needs to be determined for all corresponding languages. To set the title(s) of the event we used the

clusters of articles assigned to the event. For each of the clusters we identify the centroid article of the cluster and use its title as the title of the event for the particular language. The reason for using the centroid article is that the article is the closest to the centre of the cluster and therefore seems to be best fitted to describe the event in that language. The title of the article is used as the title of the event since it is short and should summarize the event content.

We also wish to identify a text snippet that would in a few sentences describe what the event is about. For this purpose we also rely on analysing individual clusters of articles and identifying the centroid article. From the centroid article we then use the first paragraph as the text snippet for the particular language. Similarly as for the event title we also plan as a part of the future work to analyse different summarization methods to identify a better way for identifying the event text snippet.

Extracting event date

Most events have a particular date that can be associated with the event. This date can be a single day or it can be a time period with a starting and ending date.

In order to determine the event date we use the information about the date references that we extracted from individual articles during the pre-processing phase. Our approach iterates over all articles assigned to an event and collects the identified date references. Next, we count the number of times each date is mentioned in the articles and choose the most frequent one. We want to make sure the date is mentioned significantly enough in order to use it as the event date so we set a threshold – the date has to occur at least in 30% of all articles assigned to an event. Although we presented the method as if we were only analysing the single day occurrences, we do in the same way analyse also the detected time periods. If the most mentioned time period is more frequent than the most mentioned single date then we use the time period as the starting and ending date of the event.

A large percentage of events does not contain any date references or the ones mentioned are not frequent enough to be considered as reliable. In such cases we decided to rely on the dates when the articles were published. We could assume that an event is only being reported after it happens so in principle we could use the date of the first article as the date of the event. Our experiments however show that setting event date based on the first article can be problematic. Clusters sometimes contain incorrectly assigned articles and taking the earliest article date can set too early date for the event. To be more conservative we decided to use the following approach. First, we sort all articles assigned to an event by date of publishing. Next we start iterating over the list. For each article we check how central it is to its cluster. If its cosine similarity to the cluster's centroid is above 0.7 we use the date of the article as the date of the event and finish the procedure. Since it's possible that the event is about a broader topic and the articles are less similar to each other we have also included another stopping criteria. When iterating over the list of articles we stop once we've checked 20% of articles assigned to the event and use the corresponding article's date as the event date. This allows us to skip the possibly incorrectly assigned articles and at the same time avoid assigning events to a later dates (which was the occasional problem of our first prototype that used the average article date).

Extracting event location

A majority of events are associated with a particular geographical location. The event location can be a city or a geographic area. Knowing the location is semantically important part of the event so we want to identify it from the articles.

In order to determine the event location we rely on the article annotations provided by the text annotation prototype [D3.1.1] and the word-sense disambiguation prototype [D3.2.1]. From the provided annotations we first need to isolate the annotations that represent the locations. We achieve this in the same way as in Section 4.4, namely using the GeoNames database. Of over eight million place names that are listed in the GeoNames database we selected those that have a Wikipedia page. Since the annotation service uses Wikipedia as the knowledge-base, these are the only locations that the service can identify in the text. From GeoNames we also used the information about the location labels in different XLike languages, as well as the

countries and continents in which the places are located. The country information is, for example, necessary if we later wish to search for events not by a particular location but by the name of the country.

Once we know which annotations represent locations, we can check all articles in the event and extract a list of mentioned locations. These locations are the candidates for the event location and we need to determine which one of them (if any) is the location where the event occurred. Our first prototype used a predefined rule based that determined the event location solely based on the number of times the location was mentioned. In our current implementation we however treat determining event location as a classification problem. As for cluster matching, we extract learning features for each detected location and use the model to decide if a particular location is the event location or not. Learning features are much simpler than in the case of cluster matching. For each candidate location we generate a set of features that consists of counts and ratios of event articles that mention the location. The second set of features contains counts and ratios of articles where the location was found in the dateline area of the article.

In order to test how accurately we can predict the event location using the described set of features we have built a dataset with 200 learning examples. Each learning example represents one location mentioned in the event articles. The class was assigned manually and determines whether the location is the event location or not. The classification algorithm that we used for building a model was a linear SVM. Using 5-fold cross-validation we were able to achieve 98% accuracy on the created dataset.

The procedure that we use for determining event location using the SVM model is as follows. First we identify all locations mentioned in the event articles. For each location we compute the learning features and classify the example using the model. If prediction is negative for all examples then no location is used as the event location. In case prediction is positive for a single location then that is used as the event location. In the rare case when prediction is positive for two or more locations we check the predicted probabilities and use the location with the highest probability.

Extracting event entities and keywords

Each event discusses one or more keywords and mentions a number of entities – people, organizations and locations. These entities and keywords (we will call them concepts) are crucial for understanding what the event is about and we wish to identify them and assign them a score of relevance for the particular event.

For extracting event concepts we rely on the annotations provided by the text annotation and disambiguation services. We start by identifying the concepts that were annotated in articles that are assigned to the event. Beside the URI, each concept is also assigned a weight that corresponds to the relevance of the concept for the article. The weight is computed based on how frequently the concept appeared in the article as well as where in the article it appeared. If, for example, Barack Obama is frequently mentioned in the article, then he will be assigned a high weight. The weights are currently integer numbers ranging from 1 to 5. By going over all articles in the event we compute the sum weight for each of the concepts identified in the articles. Concepts that occur frequently and with high weights will get a high total weight, whereas concepts that occur rarely and with lower weights will obtain a low total weight. Next, we rank the concepts by their total weight and ignore the concepts whose weight is lower than 5% of the weight of the highest ranking concept of the event. By ignoring the concepts with low weight we are removing the noise, which can occur due to incorrectly assigned articles or spurious annotations.

The concepts that are above the threshold are used as the event entities and keywords. Along with the concepts themselves we also use the total weights for the concepts since they are a good estimate of what is more/less relevant for the event. The total weights are then normalized to the interval [0, 100] and can be seen when viewing the event info. An example of a possible visualization of top concepts for event “Barack Obama promises military support for eastern European allies” is shown in Figure 6.

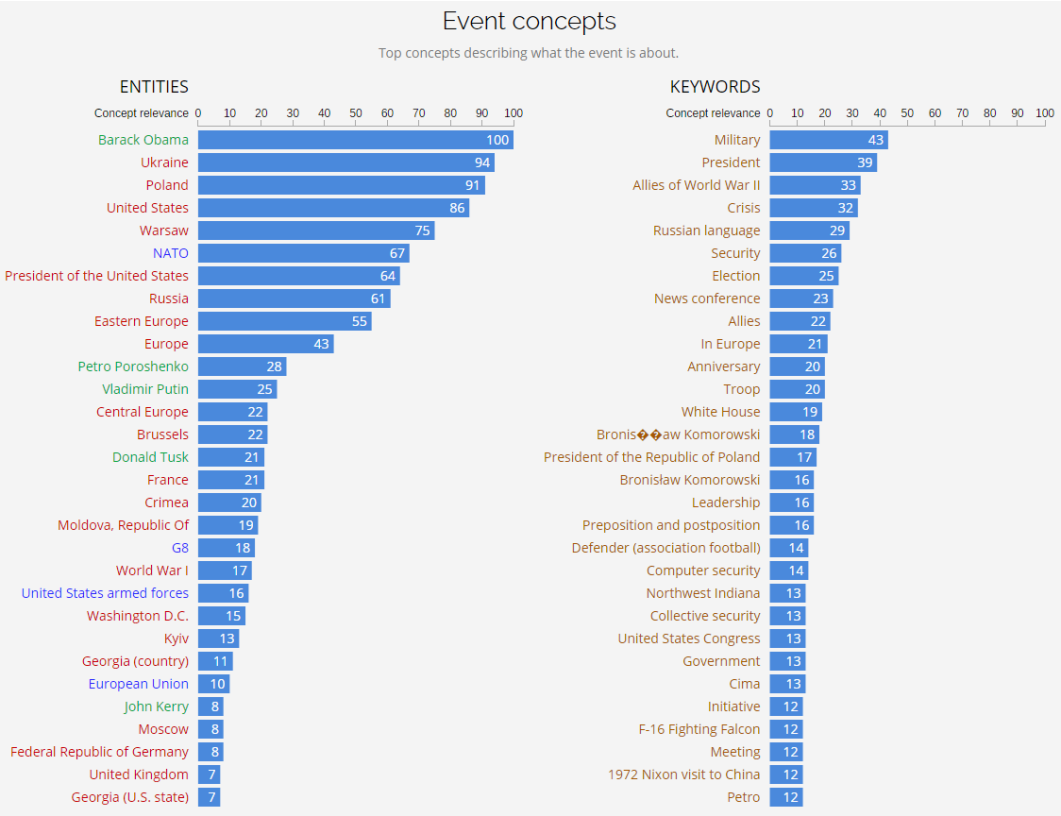

Figure 6 Example of top concepts for event "Barack Obama promises military support for eastern European allies".

Event categorization

Events can also be categorized by their types. Because topics can be very diverse we need to organize them into a taxonomy. An example of a small part of such taxonomy could be a root type "Sports", a sub-type "Athletics" with another sub-type "Triple jump". Another example could be a root type "Natural disasters" with a sub-type "Earthquake".

There organizing events into a taxonomy there is no clear choice which taxonomy to use. Most news publishers do organize their content into some form of a taxonomy, but they are considered to be a company secret and therefore not shared publicly. It is likely that taxonomies from different publishers, if they could be compared, would overlap in some areas. However, since different publishers have emphasis on different topics, different parts of the taxonomy would be more/less granular.

Since there is no public taxonomy for categorizing news we had to find a publicly accessible taxonomy that would suit our needs. The taxonomy that we chose to use is the DMoz [DMoz] taxonomy. DMoz is an open directory project that has more than 5 million web pages categorized into a taxonomy with over 1 million categories. This number of categories was too large so we decided to only use the categories up to three levels deep. In this way we obtained more than 4.600 categories which seems sufficient for our needs.

An even more important part than the taxonomy itself is for us the fact that nodes in the taxonomy also have a list of web pages that are associated with the particular category. Since the web pages are discussing the topic represented by the category we can learn from them about the characteristic features of each category. We do this by obtaining the content of the web pages, transforming them into the bag-of-words format and weighting the words using the TF-IDF weighting scheme. The pages for the category and its sub-categories can be considered as a cluster in high dimensional space and we can identify the vector representing the centre of the cluster. By identifying such vectors for each category we are able to obtain compact representations for the categories. For a new document that we would like to classify into one of the

categories we can then similarly compute the BOW format of the document, weight the words with TF-IDF weighting and use the cosine similarity to compare the document vector with the vectors of individual categories. The category with the highest cosine similarity can be chosen as the category of the document.

For categorization of events we used the DMoz classifier in the following way. For each cluster in the event we identified 5 (or less, if cluster was smaller) articles that are closest to the center of the cluster. We merged the content of these articles into a single document, computed the BOW format for the document and weighted the words with TF-IDF weighting. We identified the category with the most similar vector and assigned it to the cluster.

Our DMoz classifier currently only works on English text – determining categories for other languages in not supported. We have already implemented also a version of the classifier that relies on canonical correlation analysis, which is able to determine categories also for non-English text. The classifier is however still in testing phase so we are not yet able to report about the results.

5.4 Identifying related events

World events are not independent but highly interconnected – one event is often caused by a chain of other events. There are different criteria that we can use to identify related events. In principle we can determine event similarity/relatedness using any of the event properties:

1. Time: using a specific time window can identify very diverse events. Since their only commonality is time, the criteria by itself does not necessarily indicate relatedness.
2. Location: location can be an important feature for identifying geographically related events, especially in combination with time. It can identify events that touch various topics, which are relevant for a particular region. A good example would be finding events occurring in Ukraine in the last months.
3. Concepts (entities and non-entities): By comparing the “what” and “who” of the events we can identify events that share similar actors or topics discussed in the events. Using such criteria could identify as similar events, such as “Google releasing Android 4.4” and “Google and LG offering Nexus 5”.
4. Category: comparing events based on their type is somewhat related to the comparison using entities and keywords. A good example of using the criteria for finding similar events would be looking at an earthquake event and finding other examples of earthquakes.

Most of the described conditions offer a straightforward way for comparing the events. A condition that perhaps requires a short explanation is relatedness based on concepts. Since each event is associated with a weighted list of concepts we used the same methodology as when comparing documents. Each concept associated with the event is considered to be a document term and its weight represents the term frequency for the document. Having the bag-of-words representation of the events we also want to apply TF-IDF weighting to take into account that some concepts are more popular than others. After the vectors are weighted we can use any similarity measure (we opted for cosine similarity) to compute concept similarity between events.

5.5 Event template extraction

As described in Section 5.3 we extract time and location information from an event as well as a number of related concepts. We do not know what kind of relations connect these concepts to the event. For most events we can identify several characteristic properties other than time and location which describe it. For example, an earthquake has a magnitude, number of injured, estimated damage cost, etc. We call these properties event roles (or event slots) and a set of roles describing one event type is an event template (or event schema).

The problem of extracting event template information is two-fold. For a specific event type we must construct the template by identifying the roles that comprise it then we must have some way of populating

the roles with information from articles describing an event of that type. There are methods for automatic induction and population of event templates but state-of-the-art approaches still do not offer high enough precision and recall for our needs.

To obtain high quality event templates we have experimented using a crowdsourcing approach where human annotators both build the templates as well as populate them with instances from the articles. In order for this approach to be feasible, especially considering the volume of articles we handle, the template induction and population process must be guided in the sense that we should present the human annotator as little choice as possible while still allowing extraction of all necessary information to produce detailed templates. Only this way can we hope to achieve the necessary efficiency at a low enough cost (human annotators are not likely to work for free). We have developed a prototype of an interface which would facilitate this kind of crowdsourcing effort.

Template extraction interface

The template extraction prototype is not currently part of the Event Registry service and is available online separately at http://aidemo.ijs.si/eventAnnotation. It presents the user with events from the Event Registry service for which the user extracts event templates. Each event is presented as a set of articles that describe it obtained by the clustering method described in Section 5.1.

The interface is shown in Figure 7 and consists of a header and a body divided in three sections. The header contains a drop-down list where the user chooses an event to work on. In the body the left section contains the event articles, the middle section terms and entities extracted from the articles which are potential role-fillers and the right section a list of event types or a list of roles depending on the current stage of extraction.

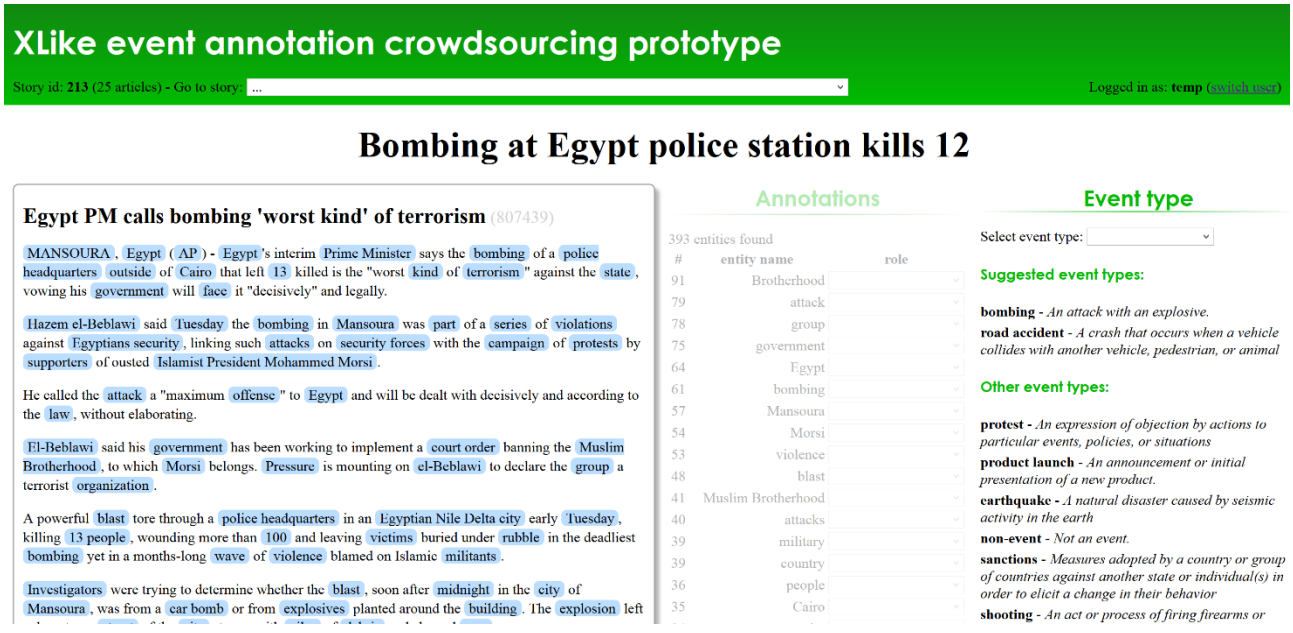

Figure 7 Template extraction interface

The template extraction work-flow consists of the following two stages:

1. **Identify event type:** The user first determines the type of event described in the articles. Using a drop-down list on the right he can choose from event types previously defined by other users or define a new event type by selecting the "…new event type…" option. Newly defined event types are added to the database for all future users to use.

   To improve efficiency of this step and help users find appropriate pre-defined event types we have developed a suggestion system which highlights two most likely event types for the event. Suggestions are generated using a SVM classifier built on 100 events hand labeled by an expert into 5 event types: bombing, product release, earthquake, road accident and protest. Events in the learning set were evenly distributed between the event types. Features used by the classifier are concepts from the event and its

articles as well as BOW features computed for event title, event summary and titles of event articles. We tested the classifier using leave-one-out testing and obtained classification accuracy of 67 %. The system suggests two event types that obtained the highest score in the classifier.

In a live setting where users would be actively adding new event type annotations into the system the suggestion classifier would also need to be updated in an online fashion or periodically rebuilt. Keeping the quality of the suggestions good with increasing number of event types is one of the challenges of deploying such a system live however that remains a matter of future work at this point.

2. **Identify and fill roles:** Once the user has identified the event type the interface replaces the event type list with the roles defined for the chosen event type. The displayed roles were again defined by other users beforehand. If the user has created a new event type the role list is empty. It also activates the middle section of the body containing a list of entities and terms extracted from the event articles that are potential role fillers. All sequences of consecutive nouns and numbers (including dates) that appear in at least two articles are highlighted as role fillers allowing for high recall while still removing those that do not seem significant enough. Entities are sorted by their frequency in the articles.

   The user can fill roles in two ways - by hovering over a highlighted entity in the article text with the cursor and selecting a role from the drop-down list that appears (Figure 8**Error! Reference source not found.**) or using the drop-down list in the entity table (Figure 9). Hovering over an entity in the table shows the first (at most) ten paragraphs from the articles that contain the entity enabling the user to quickly review the potential role the entity has in the article. New roles can be defined by the user in the similar way as event types by choosing the "…new role…" option from any drop-down list.

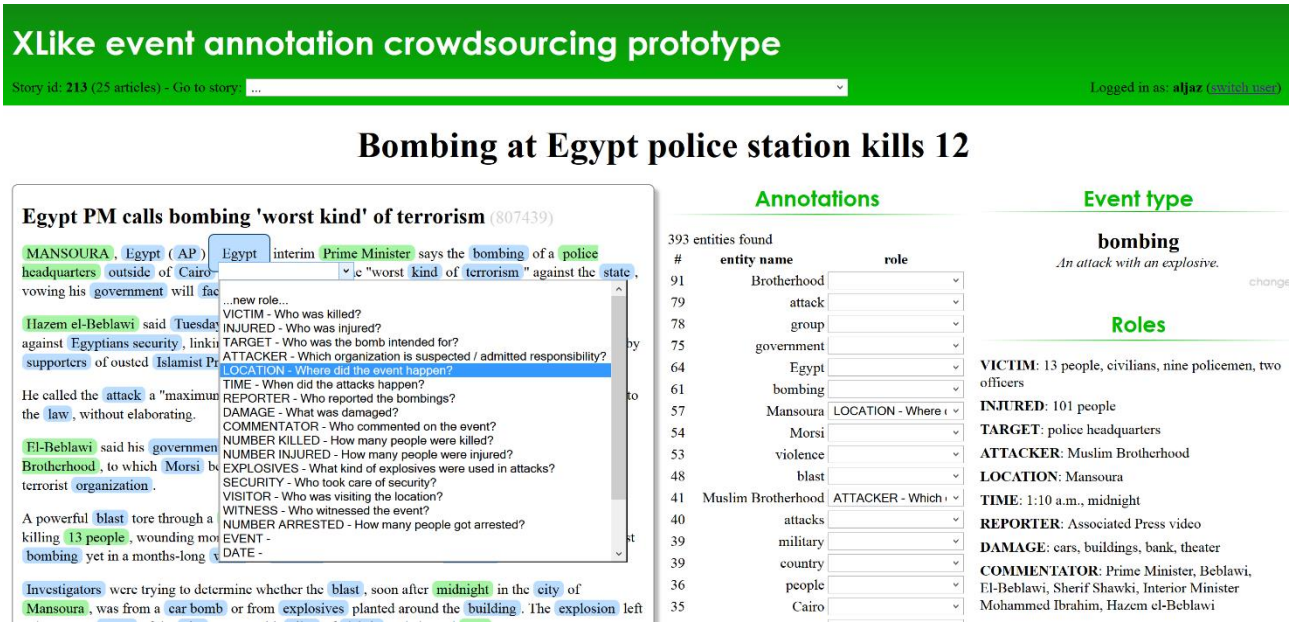

Figure 8 Choosing a role from highlighted entity the text

Entities annotated with individual roles are listed next to the role in the role list resulting in a info-box like display of structured event data.

Evaluation

We have performed a small experiment by having 11 users extract structured information about the same 10 events. We analyzed the results from two main viewpoints. Is the interface effective in supporting the user workflow and how much do users agree on the extracted templates?

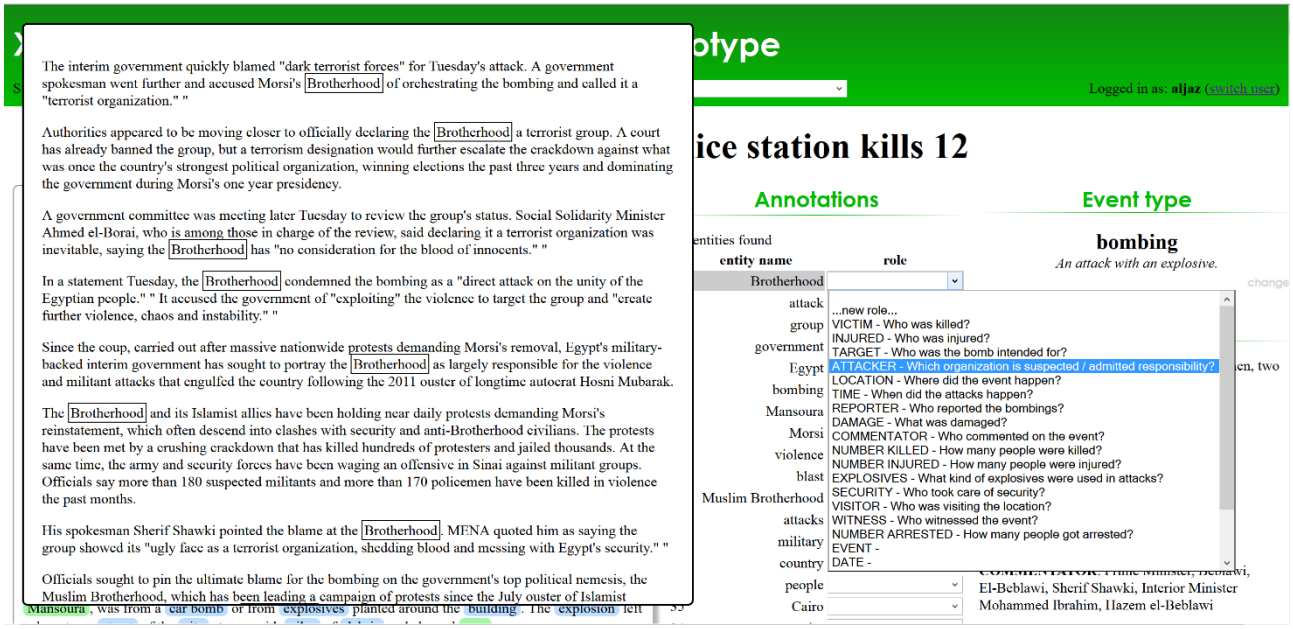

Figure 9 Choosing a role from the entity list

In order for the interface to be efficient it must be successful in guiding the user through the extraction process and reduce the number of decisions the user must make and amount of information he must process. Results show that the users on average use 12.1% ± 3.1 %[2] of all proposed entities as role fillers, meaning there is a lot of room for improvement with respect to entity proposition. However, it is challenging to reduce the number of proposed entities without affecting recall and a way to improve entity proposition remains a matter of future work. The event type suggestion system does better with users on average using the first suggested event type in 6.6 ± 1.9 events and one of the two suggested event types in 7.2 ± 2.0 events.

One way to assess the quality of the extracted templates is to see how much the users agree on them. If many users have extracted the same template for an event this raises confidence that it is a good representation of the event information. Looking at event type agreement we see that on average two users have chosen the same event type for only 5.9 ± 2.0 events. This is surprisingly low and looking at the data we see that users have quite commonly invented similar event types for some events. One option to address this is to provide a good core taxonomy of events along with an improved event type suggestion system which could notify users about existing types.

To measure agreement on roles we computed the average Jaccard index of annotated entities on pairs of events where two users selected the same event type. The index was computed as the fraction of the number of entities both users have annotated with the same role and the number of all entities annotated by both users. Its average value per event is 0.25 ± 0.09. In a more thorough analysis of role disagreement we discovered several avenues for possible improvement like term disambiguation (’12 people’ vs ‘twelve people’), use of semantic background knowledge (if ‘Paris’ is chosen as location add also ‘France’) etc. It must also be taken into account that complete agreement is likely impossible to achieve. For example in a bombing story where a police station was bombed the police station could reasonably be deemed either a target of the attack or its location.

[2] All average values are presented with their standard deviation.

6 Event Registry

Once the event is identified and processed we store all available information in the Event Registry. Data storage is provided by the QMiner[3] platform, which is a data analytics platform for storing and processing structured and unstructured data. All relevant event parameters are explicitly stored and indexed so that an efficient search can be provided across all the events. Whenever the event changes (new articles are assigned to it, a cluster splits or merges), the event properties are automatically updated in order to reflect the changes.

Event Registry provides a web application through which it is possible to access the stored information. It offers advanced search options as well as an extensive set of visualizations that enable the users to spot the patterns in the search results. Additionally it also offers a set of administrative tools for manually updating any information related to events.

6.1 Search options

Having various structured information about events enables Event Registry to offer a large set of search options. Users can specify one or more of the following conditions:

- **Concepts**. There are currently more than 1.2 million concepts that users can search for. Names of the concepts can be entered in any of the 5 languages, depending on the language selected in the Event Registry interface. If more than one concept is specified, results will include only events that are about all of them.
- **Keywords**. Uses can enter any number of plain keywords to search for. Resulting events have to contain articles that mention all of the specified keywords.
- **Event location**. Choosing a city or a country will return events that occurred in that geographic area. Specifying more locations will return events that occurred in any of those areas.
- **Publisher**. Setting a publisher will return events that have been reported about by the publisher. Entering multiple publishers will return events that have been reported by all of them.
- **Category**. The user can pick a category from the dropdown menu or enter it by name. Choosing a category will return events that are assigned to the category or any of the subcategories. Entering more categories returns events assigned to any of the categories.
- **Date**. Events can be filtered based on the date when they occurred. Entering a starting date, ending date or both will limit the resulting events to those that occurred in the specified time period.
- **Language coverage**. One can choose to only see events that at least have coverage in a particular language.
- **Event size**. Setting the event size will return events that have at least a specified number of articles that report about the event.

Besides specifying what the resulting events should contain, users can also specify what they should not. If text in the concept, location, publisher or category input box starts with "-" this will be treated as a negative condition and the events that match the condition will not be included.

An example of a populated search dialog is shown in Figure 10. The results in this example would be events about Barack Obama that are not about health care, that took place anywhere in United States between August 10th and 20th and were reported about in New York Times.

[3] https://github.com/qminer/qminer

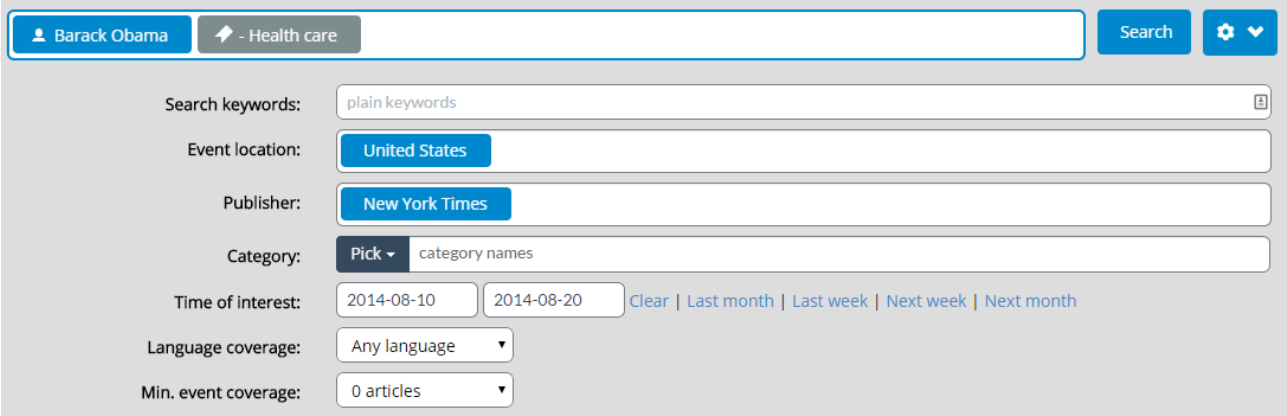

Figure 10 Search options in Event Registry

6.2 Visualizing search results

After entering any number of search conditions and pressing the Search button, the resulting events that match all the entered criteria are shown. By default, the events are shown as a list where we display for each event the main information. The events can be sorted by time, size or relevance computed based on the search conditions. If searching for Barack Obama, for example, sorting by relevance would show first the events where Obama is a very important entity for the event.

When searching for events one is often presented with thousands of events that match the search criteria. In order to obtain a better overview of results, Event Registry offers a large number of visualizations of search results. We will now briefly describe some of them. Examples of these visualizations are all shown in Figure 11.

Tag cloud. To generate a tag cloud we inspect the central articles for each resulting event and extract the most discriminative keywords. By looking at top keywords one can immediately spot top topics discussed in the events. An example of a tag cloud for events about Microsoft is shown in Figure 11.

Location. Knowing the geographic location of the events helps us to understand how the events are geographically distributed over the world. An example of an interesting map can be seen in Figure 11, where we searched for events mentioning earthquakes and the dots represent their locations. As expected we see a lot of activity in the North and South America next to the Pacific Ocean.

Concept graph and concept matrix. Different concepts are commonly related to each other. Using the concept graph one can see different entities and keywords that frequently occur together in the events. The concept matrix displays similar information but displays positive interconnectedness as well as negative.

Concept trends. The concept trends graph displays how popularity of relevant concepts in the search results changes over time. When searching for events related to Ukraine we can see in Figure 11 that Crimea was not appearing in articles until the end of February 2014.

Clustering of events. Resulting events can be about various topics. One way to find subgroups in the events is to use the clustering visualization that identifies coherent subgroups of events that are about similar topics. Clusters can be additionally split into subgroups depending on the user's interest.

Category visualization. The events are categorized into the DMoz taxonomy. Using the category visualization we can see how the events are distributed into the top three levels of this taxonomy.

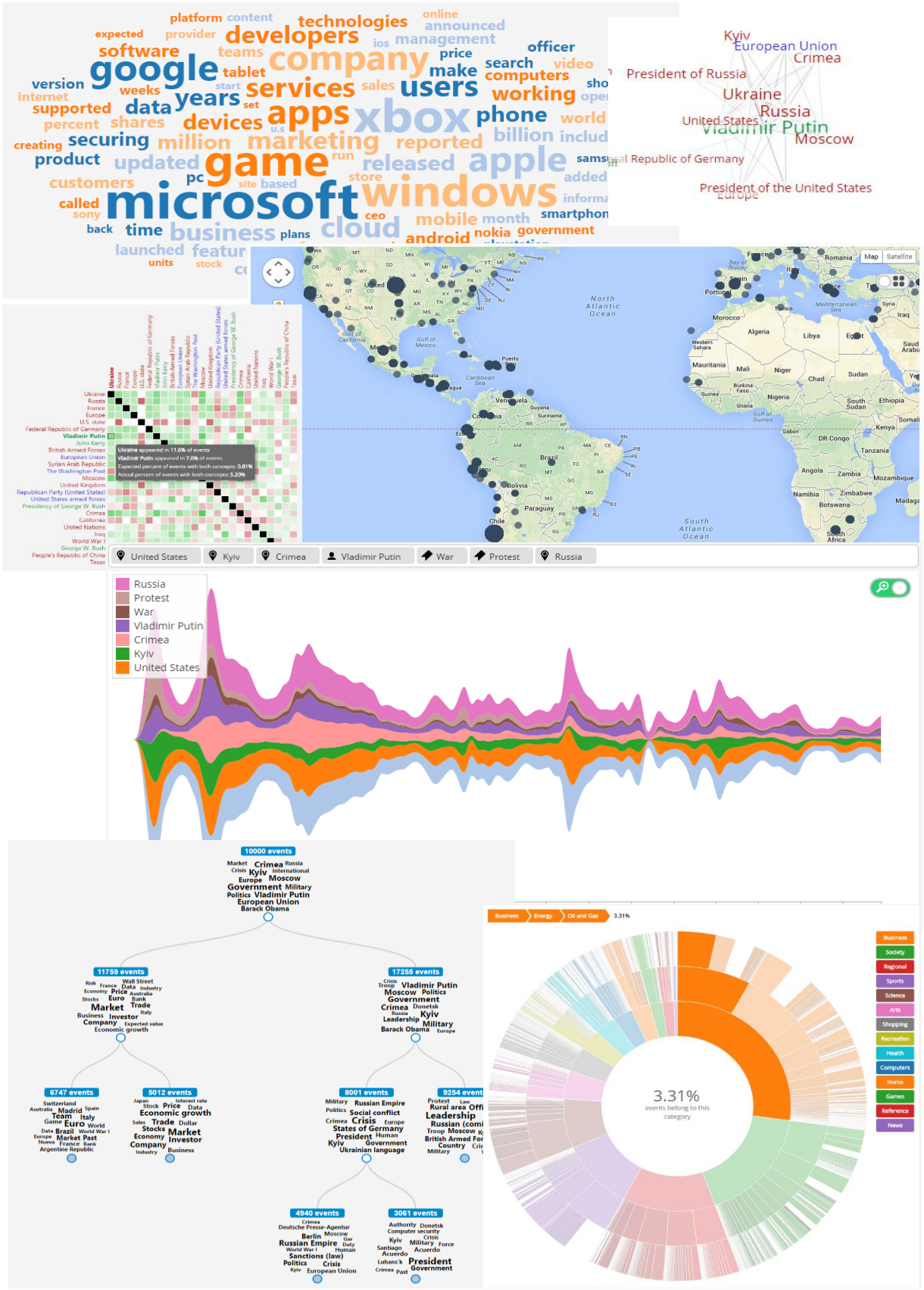

Figure 11 Examples of visualizations of search results

6.3 Displaying event information

For each event that we find using the search conditions we can display additional details. Some examples of these details will be described next and are included in Figure 12.

Event information and articles. For each event we can display its core information (title, summary, date and location) and the list of articles that describe event. Articles are grouped by language and can be ordered by date or by relevance to event. Sorting by relevance shows first the articles that are closest to the center of the cluster therefore showing first articles that are most related to the event.

Timeline of articles. In order to see what the trend of publishing articles was, one can see the timeline visualization.

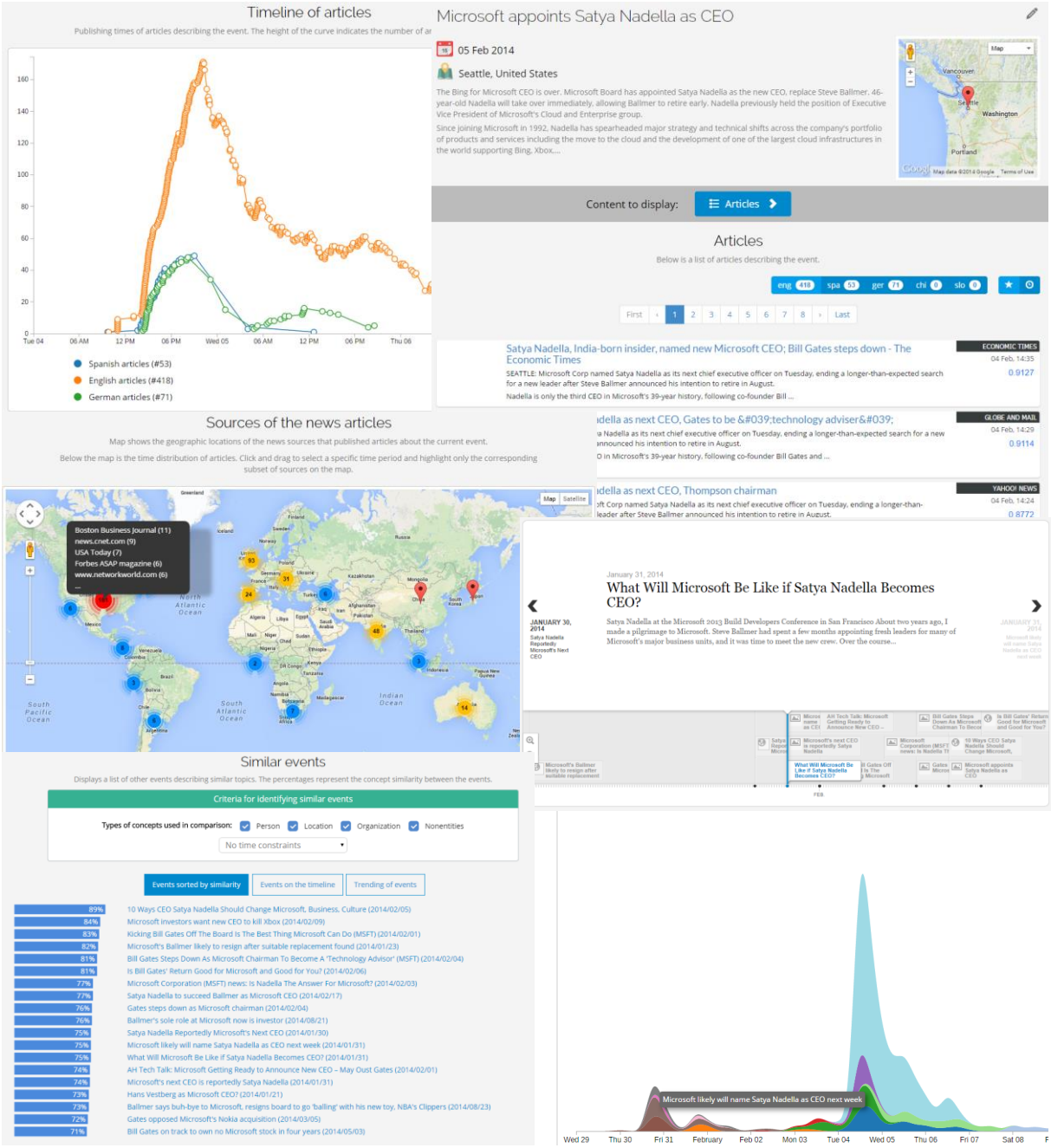

Figure 12 Information that can be displayed for each event

Sources of news articles. A world map is used to display the locations of the news sources that were reporting about the event. Using a sliding time window below the map one can even see when individual publishers published their article about the event.

Similar events. There are also three visualizations that can be used to show events similar to the once currently viewed. The first visualization simply presents a list of most similar events based on the chosen criteria (specific types of concepts or time constraints). An alternative view is to display the similar events on a timeline where one can easily see the time dimension of the related events. The third way of seeing the related events is as a trending chart where overlap between events can be spotted as well as the size of each event.

6.4 Live tracking of events

Event detection is done in an online fashion – as soon as a new article is received a corresponding event is updated or a new event created. This allows us to show web page displaying a real-time activity of articles and events. An example of such a page is shown in Figure 13. The blue markers represent new articles and are positioned based on the location of news source. The red circles are events and are positioned based on the event's location. The size of the circles represents the current number of articles discussing the event.

The real-time activity can be shown not only for the whole feed of articles and events but also for particular concepts, categories and locations. In this case articles and events are showed on the map only if they are related to the particular interest. This allows one to have a custom filter that among all the collected content shows only the specific content of interest.

Figure 13 Showing real time activity with articles and events

6.5 Administration options

Since the whole pipeline of event detection and information extraction is automatic, it is inevitable that there are mistakes in the extracted data. Most frequent examples of such mistakes are (a) articles being incorrectly assigned to an event, (b) two clusters are about the same event that are being stored as separate events and (c) incorrectly extracted date and location of the event.

Since our goal is to have a repository of events with as high accuracy as possible, we implemented a set of administrative options that can be used to fix the detected errors. These options are only available to the users that log into the Event Registry and have been assigned sufficient privileges.

The administrative options that are currently available include (see also figure below for screenshots):

- Removing from the event individual articles that have been incorrectly assigned to it.
- Updating event information. The user can change the event's location and date as well as title and summary for each language.
- Reassigning clusters to different events.
- Edit display information about concepts (labels for each language, image, description) as well as news sources (label, location, image, description).

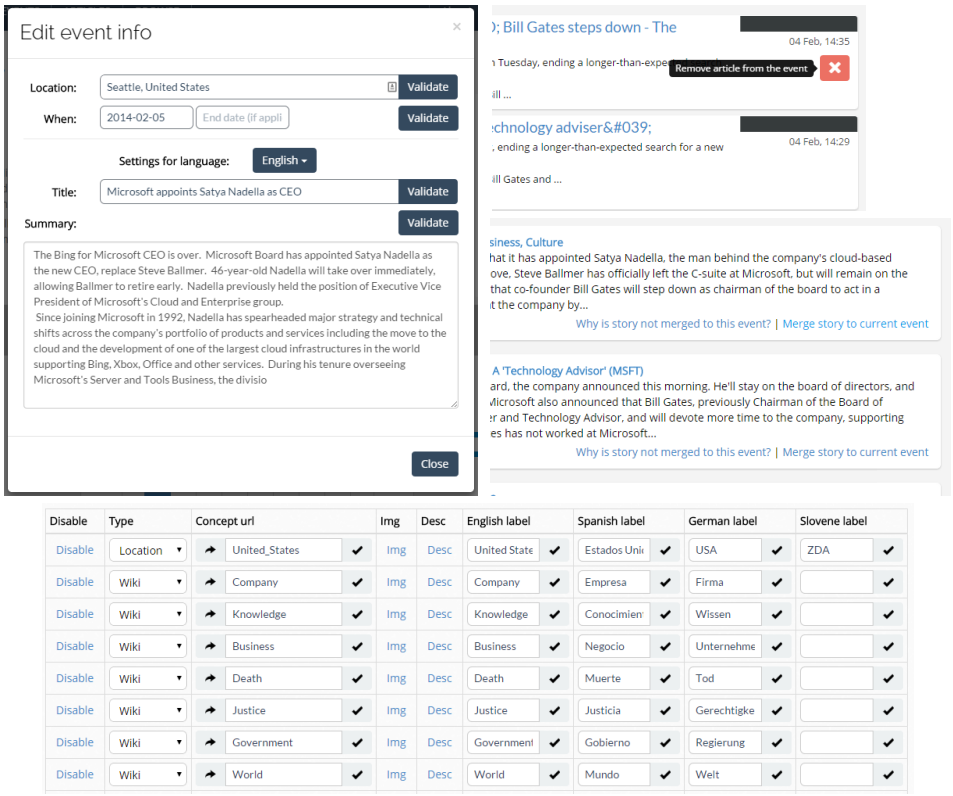

Figure 14 Administrative options for editing information in Event Registry

7 API Access to Event Registry

Event Registry is designed so that its data can be accessed not only through the available user interface but also through the provided application programming interface (API). In fact, the user interface itself is the main user of this API. All web pages in the user interface are actually just templates that are populated with content obtained by executing the appropriate API requests in JavaScript. The returned data in all API calls is in JSON format.

Most API requests that can be made on Event Registry follow the following format:

http://eventregistry.org/json/dataType?action=actionName&param1=val1&param2=val2&...

The dataType value determines the kind of data that we wish to interact with (example values include article, event, concept, source, etc.). The action parameter then determines more specifically the kind of action that we wish to execute for the chosen dataType (examples of actions include getArticle, getArticles, getEvent and getEvents). The additional parameters can be specified to determine the details of the request. Note that each parameter and its value need to be URL-encoded in order for the request to be a valid URL.

In the rest of the section we will describe the part of the API that is most relevant to obtaining information about events and articles in the Event Registry. Additionally we will also describe the Python library that we developed in order to avoid the need for manually generating web requests and thus simplify the process of interacting with Event Registry.

7.1 API calls for obtaining information about events

Using the API calls related to events, one can (1) query events that match a particular set of conditions, (2) obtain extended information about one or more specific events, and (3) modify existing event information. Modifying event information is only allowed for registered users that have sufficient privileges and does not need to be described here.

Searching for events

In order to query for events that match a particular set of conditions, one needs to make a GET web request that has the form:

http://eventregistry.org/json/event?action=getEvents&param1=val1&...

The additional arguments that are expected are:

1. One or more search conditions that determine what events are we searching for
2. One or more resultType values that determine what information about the resulting events to return. For each result type we can provide a set of possible options that specify the details of the requested data.

The search conditions can be of different types:

Parameter name	Example value	Description
conceptUri	http://en.wikipedia.org/wiki/Barack_Obama	The parameter specifies the URI that is assigned to each recognized concept in the Even Registry. It will return events where the particular concept is relevant for the event.
locationUri	http://en.wikipedia.org/wiki/Ljubljana	Specifying a value will return only events that happened at the particular location. The value can be a URI that represents a city or a country. In case a country is specified, events from all cities in that country will be returned.

publisherUri	www.bbc.co.uk	Setting a publisher URI will return only events that have been covered by the particular publisher.
categoryUri	http://www.dmoz.org/Society/Issues	Setting a category returns events that have been assigned to the particular DMoz category.
lang	eng	The language(s) in which the event should at least be covered.
keywords	new year	One or more keywords that should be found in the articles assigned to an event.
dateStart	2014-04-01	Limit the resulting events based on their date. The specified date will represent the first valid date of the resulting events.
dateEnd	2014-04-20	The specified date will represent the last valid date of the resulting events.
minArticlesInEvent	10	The minimum number of articles that an event should have in order to be returned as a result.
maxArticlesInEvent	40	The maximum number of articles that an event should have in order to be returned as a result.
dateMention	2001-09-11	The date that should be mentioned in the articles assigned to the event.
dateMentionStart	2014-02-07	The resulting events should have at least one article that mentions one of the dates in the specified date range. The parameter specifies the first valid date mention.
dateMentionEnd	2014-02-23	The last valid date that should be mentioned in the articles.
ignoreConceptUri ignoreLocationUri ignorePublisherUri ignoreCategoryUri ignoreKeywords		Users can remove from resulting events obtained using other parameters, those events, that match the ignore* parameters. Using ignoreConceptUri parameter, for example, one can hide events that are about a particular concept.
categoryIncludeSub	true | false	When a *categoryUri* parameter is set, this parameter determines whether to include also all sub-categories or not.
eventUriList	234,123,212	If we have a pre-computed list of event URIs for which we wish to obtain some information we can specify them as a comma separated list. In case this parameter is specified, no other search conditions are considered and all *resultType* results will be only computed on the list of provided event URIs.

The search conditions in the query should contain at least one of the listed parameters. If more than one parameter is specified, the matching events are first computed for each of the parameters and the final list of resulting events is then determined as an intersection of the matching events.

As it can be seen, several of the conditions expect to receive a URI as a value. It is not expected for the users to know the possible URIs in advance. Instead, the Event Registry provides a set of API calls that can return matching URIs for a given label. These API calls are described in Section 7.3.

Note also that for parameters conceptUri, locationUri, publisherUri, categoryUri and their ignore* variations we can have multiple instances of the same parameter. To find, for example, events related to Obama and Ukraine, one would have two conceptUri parameters in the request – one for each concept. In most cases, specifying multiple instances of the same parameter computes an intersection of the resulting events. An exception is the locationUri parameter. Since each event is only associated with one location, specifying multiple locationUri parameters identifies events that occurred in any of the specified locations.

Another thing that needs to be mentioned is that is not enough to specify only the ignore* parameters and none of the others. The ignore* parameters effectively just remove events from the list of events that match other conditions. If no other conditions are set then there are no evens to remove from.

Along with specifying the conditions that determine what events we are interested in we also have to specify one or more resultType parameters. They determine what information about the resulting events should be returned. For most resultType parameters there is an extensive set of additional parameters that can be set to determine the details of the returned data. The possible resultType values for events are:

- events. The core information about a subset of resulting events will be returned.
- uriList. The list of event URIs that matches the conditions.
- timeAggr. A computed time aggregate that can be used to show the distribution of resulting events over time.
- locAggr. Information about the locations of the events. For each location we return its information along with the number of events that occurred there.
- locTimeAggr. A combination of time and location aggregate. It can be used to display the number of resulting events that occurred at a location in a particular time period.
- conceptAggr. The list of most relevant concepts in the resulting events and their importance.
- keywordAggr. The list of keywords most relevant for the resulting events and their importance.
- conceptGraph. The information about top concepts and their relatedness. The returned data contains information about how frequently pairs of concepts occur in the same event.
- conceptMatrix. The information about top concepts and their co-occurrence frequency.
- trendingConcepts. The returned information contains data describing how individual concepts in the resulting events are trending over time.
- dateMentionAggr. Information about the mentioned dates in the resulting events.
- eventClusters. The resulting events are clustered based on the top concepts for each event. The 2-means clustering is used to split all events into small sub-clusters until the minimum size of the clusters is reached. The returned information can be used to display a hierarchical split of events into sub-clusters. For each identified cluster, a list of top concepts is returned.
- categoryAggr. The distribution of resulting events into the DMoz taxonomy is computed and returned.
- recentActivity. Information about the most recently modified events that match the criteria is returned. Can be used to identify recently added or updated events related to a particular concept, location, topic, etc.

Since the list of possible parameters for each resultType is very long we included it in the Appendix A. We have also included there a list of example request URLs and the format of the returned data.

Obtaining information about particular event(s)

The API for searching for events is used to obtain a list of events that match the criteria and for computing various aggregates on the results which can be then be further used as a summary or for visualization purposes. If we want to obtain more detailed information about one or more particular events we then use a different API call that has the following form:

http://eventregistry.org/json/event?action=getEvent&param1=val1&...

The additional arguments that are expected are:

1. One or more eventUri values that determines which events are we interested in

2. One or more resultType values that determine what information about the events to return. For each result type we can again provide a set of possible options that determine more specifically what should be included.

Possible values for resultType include:

- info. Returns main information about the event, such as location, date, title, summary, number of articles, etc.
- articles. Returns the list of articles describing the event along with the requested article meta-data.
- articleUris. Returns only the list of article URIs. The URIs are links to the original web pages containing the articles but they can be as well used in API calls related to articles.
- keywordAggr. The output is a list of keywords and their weights that best summarize the articles describing the event.
- sourceAggs. The provided information shows what are the news sources that are reporting about the event, where are they located and when did they report about the event.
- dateMentionAggr. The generated output provides information on the detected dates that were mentioned in the articles about the event.
- articleTrend. The information can be used to display how the intensity of reporting about the event has been changing over time.
- similarEvents. The output contains a list of events that are related to the given event. The criteria for relatedness is provided by the optional parameters.

More details about additional parameters for each individual resultType is in the Appendix A.

7.2 API calls for obtaining information about articles

Although events are the core information that is provided by the Event Registry, we also provide extensive support for querying and obtaining information related to individual articles. There are two main types of API calls related to articles – a call for finding articles according to search criteria and a call for obtaining extended information for one or more articles.

Searching for articles

To search for articles that matches a set of conditions one needs to make a GET web request that has the following form:

http://eventregistry.org/json/article?action=getArticles&param1=val1&...

The additional arguments that are expected are:

1. One or more search conditions that determine what articles are we searching for
2. One or more resultType values that determine what information about the articles to return. For each result type we can provide a set of possible options that specify the details of the requested data.

The search conditions can be of different types and are similar to conditions used when searching for events:

Parameter name	Example value	Description
conceptUri	http://en.wikipedia.org/wiki/Barack_Obama	The concept that should be annotated in the article

locationUri	http://en.wikipedia.org/wiki/Ljubljana	Specifying a value will return only articles in which we were able to identify a dateline and extract from it the exact location of the mentioned event. The value can be a URI that represents a city or a country. In case a country is specified, articles from all cities in that country will be returned.
publisherUri	www.bbc.co.uk	Setting a publisher URI will return only articles written by the particular publisher
categoryUri	http://www.dmoz.org/Society/Issues	Setting a category returns articles associated with the particular DMoz category.
lang	eng	The language in which the article should be written
keywords	new year	One or more keywords that should be found in the article
dateStart	2014-04-01	Limit the resulting articles based on their published date. The specified date will represent the first valid date of the resulting articles.
dateEnd	2014-04-20	The specified date will represent the last valid date of the resulting articles.
dateMention	2001-09-11	Sets the date that should be mentioned in the article
dateMentionStart	2014-02-07	The resulting articles should mention dates from the specified date range. The parameter specifies the first valid date mention.
dateMentionEnd	2014-02-23	The last valid date that should be mentioned in the articles
ignoreConceptUri ignoreLocationUri ignorePublisherUri ignoreCategoryUri ignoreKeywords ignoreLang		By specifying these conditions some results that match other conditions can be removed from the list of returned articles. Using ignoreConceptUri parameter, for example, one can hide articles that are about a particular concept.
categoryIncludeSub	true | false	When a categoryUri parameter is set, this parameter determines whether to include also all sub-categories or not.
articleUriList	234,123,212	If we have a pre-computed list of article URIs for which we wish to obtain some information we can specify them as a comma separated list. In case this parameter is specified, no other search conditions are considered and all resultType results will be only computed on the list of provided article URIs.

As when searching for events, the search conditions in the query should contain at least one of the listed parameters. If more than one parameter is specified, the matching articles are first computed for each of the parameters and the final list of resulting articles is then determined as an intersection of the matching articles.

For parameters conceptUri, locationUri, categoryUri and their ignore* variations it is possible to specify multiple instances of the same parameter. If multiple locationUri parameters are specified, then articles are found that occurred in any of the specified locations. In other cases, Boolean AND is assumed between the parameters so returning articles have to match all specified conditions.

Besides specifying the search conditions, the URL request also has to contain the resultType parameter to determine the desired output information. Possible values for the resultType are:

- articles. Returns the information about a subset of resulting articles.
- uriList. The list of article URIs that matches the search conditions.
- timeAggr. Returns the distribution of resulting articles over time.

- conceptAggr. The list of most relevant concepts for the resulting articles and their importance.
- keywordAggr. The list of keywords most relevant for the resulting articles and their importance.
- conceptGraph. The information about top concepts and their relatedness. The returned data contains information about how frequently pairs of concepts occur in the same article.
- conceptMatrix. The information about top concepts and their co-occurrence frequency in the resulting articles.
- trendingConcepts. The returned information contains data describing how individual concepts in the resulting events are trending over time.
- sourceAggr. Information about the news sources that are reporting about the matching articles.
- dateMentionAggr. Information about the mentioned dates in the resulting articles.
- categoryAggr. The distribution of resulting articles into the DMoz taxonomy.
- recentActivity. Information about the most recently added articles that match the criteria is returned. The information can be used to show near real-time activity related to a particular concept, location, topic, etc.

The full list of possible parameters for each resultType is provided in Appendix A. We also provide examples of generated URL requests.

Obtaining detailed information about particular article(s)

The main information about the articles can be obtained using the described API call and using articles as one of the values of the resultType parameters. There is however additional information that one can access by performing another API call. This call has the following form:

http://eventregistry.org/json/article?action=getArticle&param1=val1&...

There are two additional parameters that are expected in these requests:

- articleUri which specifies one or more articles for which we wish to obtain the information
- one or more resultType parameters that determine the desired information about the specified articles

Possible values for the resultType parameter are the following:

- info. The returned information contains the main data about the article. The level of details is specified by providing additional parameters.
- similarArticles. The returned information contains a list of articles that are found to be similar according to the canonical correlation analysis (CCA).
- duplicatedArticle. If the article has duplicated articles then this result type will return the list of articles that were found to be the duplicates. If there are no duplicates found, the list will be empty.
- originalArticle. If the specified article is a duplicate of another article, this will return information about the original article.

Details about the resultType parameters are listed in Appendix A.

7.3 Other relevant API calls

Although we have described only API calls related to events and articles, there are many other supported calls that are available. For the purpose of brevity we will describe here only the most relevant ones that are also needed in order to perform the event and article related queries.

Getting concept URIs

To find concept URIs for the specific concepts one can make a request of the form:

http://eventregistry.org/json/suggestConcepts?prefix=label&lang=eng&param1=val1&...

In this case, the concept label (or a part of the label) is required and is provided by the prefix parameter. Since Event Registry stores concept labels in different languages, one also needs to tell using the lang parameter in which language is the provided label. Currently possible values are eng, deu, spa, zho and slv. If one is interested only in obtaining concepts of particular type, this can be specified by an optional parameter source. Possible values for the parameter are person, org, loc and wiki and they correspond to people, organizations, locations and nonentities.

The result of the URL request is a JSON array which contains concept URIs as well as other information about the concepts that best match the specified prefix. An abbreviated list of results for prefix=Obama would look like this:

```
[{
        'fq': 336154.0,
        'labelEng': 'Barack Obama',
        'type': 'person',
        'uri': 'http://en.wikipedia.org/wiki/Barack_Obama'
},
{
        'fq': 22346.0,
        'labelEng': 'Michelle Obama',
        'type': 'person',
        'uri': 'http://en.wikipedia.org/wiki/Michell_Obama'
}, ...]
```

The uri value of the JSON object is the value that should be used as a value for conceptUri parameter when searching for events or articles.

Getting news source URIs

When finding a URI for a news source one can make request of form:

http://eventregistry.org/json/suggestSources?prefix=label

The value of the label can represent the (part of the) source title or part of the source URL. The result is again an array of JSON objects where the uri value of the objects should be used as the value in the publisherUri requests.

Getting location URIs

Location information can be obtained using a request of form:

http://eventregistry.org/json/suggestLocations?prefix=label&lang=eng&param1=val1

The prefix argument contains partial or full name of the city or country, for which we wish to obtain the URI. By setting the value for the lang parameter, one should specify in which language is the given label. Additionally, one can also set a source parameter to a value of city or country in case they wish to receive only the matching locations of a particular type.

Getting category URIs

The last type of URIs that users need to be able to obtain are URIs for DMoz categories. URL requests in this case should be of the following form:

http://eventregistry.org/json/suggestCategories?prefix=label

The label in this case should be a part of the DMoz category name, such as, for example, “basketball”.

7.4 Accessing Event Registry through Python

Accessing Event Registry data by manually creating URL requests by inspecting documentation would likely be an overkill for most of the potential users of the API. For this purpose we have implemented a simple Python library that contains a predefined set of classes that can be used for accessing the relevant information. The library is freely available at https://github.com/gregorleban/event-registry-python. The library is well documented and contains several examples. For this reason we will include here only a short description of its usage using short examples.

To use the library one needs to import it first:

```
>>> from EventRegistry import *
```

The main class that one needs to use is EventRegistry. It stores the location of the service and is able to make the necessary web requests. An instance can be created by calling:

```
>>> er = EventRegistry()
```

To obtain a list of URI suggestions for a label, the class provides methods such as suggestConcepts, suggestNewsSources, suggestLocations and suggestCategories. Each of the calls returns a list of python dictionaries where the “uri” key contains the URI value to use in the requests. If one is sure that the desired item will be first in the list, they can use an easier approach by calling methods getConceptUri, getLocationUri, getCategoryUri and getNewsSourceUri that all accept label as an argument and directly return the URI of the first item in the list.

There are four main class that are used for querying data – QueryEvents, QueryEvent, QueryArticles and QueryArticle. We will show now examples of possible requests using all four types.

QueryEvents example

An example of a query for events could look something like this:

```
>>> q = QueryEvents()
>>> q.addConcept(er.getConceptUri("Obama"))              # get events related to Barack Obama
>>> q.addCategory(er.getCategoryUri("society issues"))   # and are related to issues in society
>>> q.addNewsSource(er.getNewsSourceUri("bbc"))          # and have been reported by the BBC
>>> q.addRequestedResult(RequestEventsUriList())         # return uris of all events
>>> q.addRequestedResult(RequestEventsInfo(page = 0, count = 30))  # return event details for first 30 events
>>> q.addRequestedResult(RequestEventsConceptAggr())     # compute concept aggregate on the events
>>> res = er.execQuery(q)
```

We start by adding a concept, category and news source condition. By calling the addRequestedResult method we specify what the desired results that we wish to obtain using the query are. By calling the execQuery method we execute the query and return the results. The res object in our example would be a python dictionary with the following structure:

```
{
‘conceptAggr’: [
        {'labelEng': 'United States', 'score': 42.0, 'type': u'loc', 'uri': u'http://en.wikipedi…ed_States'},
        { 'labelEng': 'Barack Obama', 'score': 29.0, 'type': 'person', 'uri': u'http://en.wikipedi…ack_Obama'}, …],
```

‘events’: { ‘resultCount’: 122,
‘results’: [
{'articleCounts': {'eng': 54.0, 'total': 54.0}, 'categories': [{...}], 'concepts': [{...}, ...], 'eventDate': '2014-08-29', 'eventDateEnd': '', 'multiLingInfo': {u'eng': {...}}, 'uri': '1211229', 'wgt': 9.0}, ...
]}
‘uriList’: ['1211229', '1204045', '1195905', '1175569', ...]
}

As it can be seen from the result, each requested information has a corresponding property in the returned object and its value holds the returned results.

QueryEvent **example**

When information about a particular event is required, one can use the QueryEvent class as in the following example:

```
>>> q = QueryEvent("123");
>>> q.addRequestedResult(RequestEventInfo(["eng", "spa", "slv"]))
>>> q.addRequestedResult(RequestEventArticles(0, 10))
>>> q.addRequestedResult(RequestEventArticleTrend())
>>> q.addRequestedResult(RequestEventKeywordAggr())
>>> eventRes = er.execQuery(q);
```

In this example we have requested for information about the event with URI “123”. We have asked for event details including the title and summary in English, Spanish and Slovene language. We have also asked for 10 articles about event. Since sorting is not specified, this will return 10 articles that are closest to the center of the clusters. The article trending request will return info about the intensity of new articles at different times, whereas the keyword request will return top keywords for the event.

QueryArticles **example**

An example of the QueryArticles class use when searching for articles is as follows:

```
>>> q = QueryArticles();
>>> q.setDateLimit(datetime.date(2014, 4, 16), datetime.date(2014, 4, 28))
>>> q.addKeyword("apple")
>>> q.addKeyword("iphone")
>>> q.addRequestedResult(RequestArticlesInfo(page=0, count = 30));
>>> res = er.execQuery(q)
```

In this case we specify the time limit to include only articles between 16th and 28th April, 2014. Additionally we specify two keywords to search for – apple and iphone. The resulting information will only contain 30 top articles.

QueryArticle **example**

The last example shows the use of the QueryArticle class.

```
>>> uri = “http://www.bbc.com/news/technology-29139533”
>>> q = QueryArticle(uri);
>>> q.addRequestedResult(RequestArticleInfo())
>>> q.addRequestedResult(RequestArticleDuplicatedArticles())
>>> articleRes = er.execQuery(qa);
```

Assuming that uri is a valid URI of an article in the Event Registry, the example requests for article information as well as the list of articles that are duplicates of the article.

8 Future work

The system for event extraction that we have presented consists of a pipeline of services. Some of these service are in a stable and mature phase whereas others are more at the level of a prototype. Event template extraction in particular is an example of a prototype service that we believe can be extended and improved in several ways. First we wish to build a catalogue or a taxonomy of possible event types. Once these are identified we need a classifier that will for a given event identify the event type with high accuracy. To improve the accuracy we plan to extend the set of learning features to include also other available event information such as the categorization of articles, annotated concepts and information about news sources.

Even more importantly than just identifying the correct event type is the extraction of info boxes for events. For each event type we want to define a set of properties that should be populated for events of that type. The task in this case is twofold – an extended set of properties has to be first identified for each event type and a method then developed that will be able to extract values for instances of events.

A next step that is of high importance for us is also inclusion of support for a larger number of languages. Event Registry currently relies heavily on the annotations provided by the annotation service. This is therefore the main service that needs to be adapted in order to support other languages.

Additional area that we plan to look into is extracting more information from the articles. An example would include extraction of citations. By finding different mentions of the same citations we could analyse how they are evolving and morphing when being reported by different news publishers.

9 Conclusion

In this deliverable we provided information about the final version of the event extraction prototype. The presented work is an extension of the work done for the D4.3.1 Early event extraction prototype.

The presented system is able to identify world events by monitoring and analysing the news articles published worldwide. The service starts by collecting and annotating news articles written in 5 languages – English, Spanish, German, Chinese and Slovene. For articles in these languages we identify mentions of people, locations, organizations, keywords and dates. Using the canonical correlation analysis we also identify similar articles in other languages. Based on the article features we identify clusters that are describing the same event. Since clusters in different languages can describe the same event we used machine learning to build a model that is able to identify such clusters so that we can represent them using a single event ID. Once an event is created we developed methods for extracting from it the key information such as the title, summary, date, location and the most relevant concepts. We have also presented our current work on extracting event templates for different event types.

The data in the Event Registry can be easily accessed using the developed front-end interface. It provides a large number of options to search for events and articles of interest. The resulting information can be visualized and presented in different ways that offer an easier insight in the data. Additionally, the Event Registry also provides API calls that can be used to access the data directly. This deliverable provides the necessary details about the available parameters as well as the description of the available Python library that can be used to simplify the access to the data.

References

[AHWY03] C. C. Aggarwal, J. Han, J. Wang, P. S. Yu. A framework for clustering evolving data streams. *Proc. of the 29th VLDB Conference*, 2003.

[AY06] C. C. Aggarwal, P. S. Yu. A framework for clustering massive text and categorical data streams. *Proc. 6th SIAM Int. Conf. on Data Mining* (SDM), 2006.

[AY10] C. C. Aggarwal, P. S. Yu. On clustering massive text and categorical data streams. *Knowledge and Information Systems*, 24(2):171-196 (2010).

[CW12] Â. Cardoso, A. Wichert. Iterative random projections for high-dimensional data clustering. *Pattern Recognition Letters*, 33(12):1749–55, 2012.

[Dav12] M. Davis (ed.) *Unicode Text Segmentation*. Unicode Standard Annex #29, Unicode 6.2.0, 12 September 2012.

[D1.3.1] Early prototype of data infrastructure, XLike deliverable

[D2.1.1] Shallow linguistic processing prototype, XLike deliverable

[D3.1.1] Early text annotation prototype, XLike deliverable

[D3.2.1] Early ontological word-sense disambiguation prototype, XLike deliverable

[D4.1.1] Cross-lingual document linking prototype, XLike deliverable

[D6.2.1] Early prototype, XLike deliverable

[Dav12] M. Davis (ed.) *Unicode Text Segmentation*. Unicode Standard Annex #29, Unicode 6.2.0, 12 September 2012.

[DMoz] “DMoz, open directory project.” [Online]. Available: http://www.dmoz.org/.

[FP07] Fung, Gabriel Pui Cheong, et al. "Time-dependent event hierarchy construction." *Proceedings of the 13th ACM SIGKDD conference*. ACM, 2007.

[Geo] “GeoNames.” [Online]. Available: http://www.geonames.org/.

[JA98] Allan, James, et al. "On-line new event detection and tracking." *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1998.

[LE12] E. Lughofer. A dynamic split-and-merge approach for evolving cluster models. *Evolving Systems*, 3:135-151 (2012).

[MG09] M. Muhr, M. Granitzer. Automatic cluster number selection using a split and merge k-means approach. *DEXA Workshops* 2009, pp. 363-7.

[PM00] D. Pelleg, A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. *Proc. ICML* 2000.

Annex A Extended list of parameters for API calls

In this part of the document we describe the additional parameters that can be specified for different API requests. In the first section we describe details related to requests related to events and in the second section requests related to articles.

For each request, the user can ask for multiple result types by specifying multiple resultType parameter values (e.g. …&resultType=events&resultType=locAggr&resultType=…). Because some parameters for different resultTypes overlap (ConceptLang parameter for example, can be specified for locAggr, conceptAggr and other resultTypes) we use the following naming convention when specifying the parameters. When specifying parameter name for a particular resultType we use the name of the resultType as a prefix in the parameter name. When specifying ConceptLang for locAggr resultType, for example, we use locAggrConceptLang as the full parameter name. The same holds for other parameter names and resultType values. In all following tables we list the abbreviated names of the parameters (without their prefixes).

Result of various API requests include details about the events and articles. Depending on the use-case the user might be interested in obtaining more or less information. For this purpose we provide a set of parameters that can be used to specify which details specifically should be returned. The requests that support these parameters have in the following tables information about which prefix should be used when specifying these details.

Parameters for obtaining event details

Last part of parameter name	Default value	Description
IncludeArticleCounts	true	Should information about the number of articles for each language be included?
IncludeConcepts	true	Should the top concepts for the event be included?
IncludeMultilingInfo	true	Should the title and summary of the event for each language be included?
IncludeCategories	true	Should the categorization of the event be included?
IncludeLocation	true	Should the location details of the event be included?
IncludeStories	false	Should the individual clusters in the event be included?
IncludeImages	false	Should the images for the event be included?

Parameters for obtaining article details

Last part of parameter name	Default value	Description
IncludeBasicInfo	true	Should the basic info about the article (id, uri, language) be included?
IncludeBody	true	Should the article body be included?
IncludeTitle	true	Should the title of the article be included?
IncludeConcepts	false	Should the annotated list of concepts be included?
IncludeSourceInfo	true	Should the information about the publisher be included?
IncludeEventUri	true	Should the event URI for the article be included?
IncludeStoryUri	false	Should the cluster URI for the article be included?
IncludeDuplicateList	false	Should the list of duplicated articles be included?
IncludeCategories	false	Should the categories for the article be included?
IncludeLocation	false	Should the explicit article location be included?
IncludeImage	false	Should the image URL be included?

9.1 Queries related to events

Possible result types for getEvents action

resultType: events

Last part of parameter name	Default value	Description
Page	0	Which page of results to return.
Count	25	Number of returned events per page.
SortBy	none	[date, size, rel, none] Sorting order of events – it can be either by date (date), by number of articles reporting about the event (size), by relevance to the query (rel) or random (none).
SortByAsc	false	[true, false] Should the order be increasing or decreasing?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include for each event?
+ event details parameters (use prefix "events").		

resultType: keywordAggr

Last part of parameter name	Default value	Description
Lang	eng	[eng, deu, spa, zho, slv] In which language should be the returned keywords?

resultType: locAggr

Last part of parameter name	Default value	Description
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?

resultType: locTimeAggr

Last part of parameter name	Default value	Description
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?

resultType: conceptAggr

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?

ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	25	Number of returned concepts per concept type.

resultType: conceptGraph

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	25	Number of returned concepts.
LinkCount	50	Number of edges between concepts.
SampleSize	500	Number of events to use when computing the graph.

resultType: conceptMatrix

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	25	Number of returned concepts.
Measure	pmi	[pmi, pairTfIdf, chiSquare] The measure used when computing how much pairs of concepts are co-occuring. Possible measures are pointwise mutual information (pmi), pair frequence * IDF of individual concepts (pairTfIdf) and chi-square (chiSquare).
SampleSize	500	Number of events used when computing the matrix.

resultType: trendingConcepts

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	10	Number of returned concepts.

resultType: dateMentionAggr

Last part of parameter name	Default value	Description
MinDateMentionCount	0	Minimum number of times a date has to be identified in order to be included in the results.
MinDaysApart	0	What is the minimum difference in days between the event and the detected dates for the event?

eventClusters

Last part of parameter name	Default value	Description
KeywordCount	30	Number of keywords to include for each cluster.
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
MaxEventsToCluster	10.000	Maximum number of events to cluster. The rest of the resulting events will be assigned to the clusters based on their similarity to them.

Possible result types for getEvent action

resultType: info

Last part of parameter name	Default value	Description
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include for each event?
+ event details parameters (use prefix “info”).		

resultType: articles

Last part of parameter name	Default value	Description
Lang	eng	[eng, deu, spa, zho, slv] In which language should be the returned articles?
Page	0	The page number of returned articles.
Count	20	The number of articles to return for each page.
SortBy	cosSim	[date, id, cosSim] The order in which the articles will be returned – it can be either by date (date), by id (id) or increasing distance from the center of the cluster (cosSim).
SortByAsc	false	[true, false] Should the order be increasing or decreasing?
BodyLen	300	Maximum length of the body to return. Use -1 for whole content.
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include for each article?
+ article details parameters (use prefix “articles”).		

resultType: articleUris

Last part of parameter name	Default value	Description
Lang	eng	[eng, deu, spa, zho, slv] In which language should be the returned articles?

SortBy	cosSim	[date, id, cosSim] The order in which the articles will be returned – it can be either by date (date), by id (id) or increasing distance from the center of the cluster (cosSim).
SortByAsc	false	[true, false] Should the order be increasing or decreasing?

resultType: articleTrend

Last part of parameter name	Default value	Description
Lang	eng	[eng, deu, spa, zho, slv] In which language should be the returned articles?
MinArticleCosSim	0	[0-1] What is the minimum cosine similarity of the article to the center of the cluster for the article to be included in the trending information? Set this value to remove the articles that are likely miss-assigned.
BodyLen	300	Maximum length of the body to return. Use -1 for whole content.
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
+ article details parameters (use prefix "articleTrend")		

resultType: similarEvents

Last part of parameter name	Default value	Description
Count	20	The number of similar events to return.
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include for each similar event?
Source	concept	[concept, cca] The method used when computing the similar events – it can be either concept similarity (concept) or CCA similarity (cca).
MaxDayDiff	0	The maximum number of days that the event should be apart from the original event.
AddArticleTrendInfo	false	Should the returned information also include the trending information of articles for each event?
AggrHours	6	If trending information should be included, what is the aggregating time window?
+ event details parameters (use prefix "similarEvents")		

Examples of URL requests

The bottom request searches for events that are about Barack Obama and have been covered by BBC. The requested result includes event information (resultType=events) and top concepts (resultType=conceptAggr). Information about events includes 10 latest events (eventsSortBy=date and eventsCount=10) and does not contain event title and summary (eventsIncludeMultiLingInfo=False) and contains the list of most relevant people for each of the 10 events. The result also includes top concepts that are of type organization (conceptAggrConceptType=org) in all of the resulting events.

http://eventregistry.org/json/event?action=getEvents&resultType=events&resultType=conceptAggr&
conceptUri=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FBarack_Obama&publisherUri=www.bbc.co.uk&

eventsConceptType=person&eventsSortBy=date&eventsConceptLang=eng&eventsCount=10&eventsPage=0&eventsIncludeMultiLingInfo=False &conceptAggrConceptType=org&conceptAggrConceptLang=eng

The next example request obtains information about the event with URI "123". The requested information includes details about the event (resultType=info). The concept information should include relevant organizations and locations (infoConceptType=org and infoConceptType=loc). The labels of the concepts should be in Spanish and English language (infoConceptLang=eng and infoConceptLang=spa). The information should also contain event location (infoIncludeLocation=True).

http://eventregistry.org/json/event?action=getEvent&eventUri=123&resultType=info&infoConceptType=org&infoConceptType=loc&infoIncludeLocation=True&infoConceptLang=eng&infoConceptLang=spa

Example of event information

```
{
   "info": {
      // number of articles describing the event
      "articleCounts": { "deu": 48.0, "eng": 239.0, "spa": 124.0, "total": 411.0 },
      "concepts": [         // top concepts for the event
         {
            "labelEng": "Ukraine",
            "score": 100.0,
            "type": "loc",
            "uri": "http://en.wikipedia.org/wiki/Ukraine"
         },
         {
            "id": "191",
            "labelEng": "Russia",
            "score": 84.0,
            "type": "loc",
            "uri": "http://en.wikipedia.org/wiki/Russia"
         },
         ...
      ],
      "eventDate": "2014-07-17",        // event date
      "eventDateEnd": "",               // when did the event end (empty if on the same day)
      "location": {                     // where did the event happen
         "labelEng": "Ukraine",
         "lat": 49.0,
         "long": 32.0,
         "wikiUri": "http://en.wikipedia.org/wiki/Ukrainian_Soviet_Socialist_Republic"
      },
      "multiLingInfo": {                // title and summary of the event in different languages
         "deu": {
            "summary": "Ein Malaysisches Flugzeug mit 295 Menschen an Bord ist ...",
            "title": "Malaysisches Verkehrsflugzeug \u00fcber Ukraine abgeschossen"
         },
         "eng": {
            "summary": "A Malaysian airliner was shot down over eastern Ukraine ...",
            "title": "Malaysia airliner: Ukraine says rebels shot down Malaysian airliner, 295 dead"
         },
         "spa": {
            "summary": "EFE- La aerol\u00ednea malasia Malaysian Airlines confirm ...",
            "title": "Malaysian Airlines confirma que perdi\u00f3 el contacto con un avi\u00f3n en Ucrania"
         }
      },
      "uri": "999677"                   // URI of the event
   }
}
```

9.2 Queries related to articles

Possible result types for getArticles action

resultType: articles

Last part of parameter name	Default value	Description
Page	0	Which page of results to return.
Count	25	Number of returned articles per page.
SortBy	cosSim	[date, id, cosSim, fq] The method used for sorting the resulting articles. Sorting can be by date (date), id (id), distance to centroid of belonging cluster (cosSim) or by fitness to the query (fq).
SortByAsc	false	[true, false] Should the articles be sorted increasingly?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
BodyLen	300	Maximum length of the body to return. Use -1 for whole content.
+ article details parameters (use prefix “articles”)		

resultType: categoryAggr

Last part of parameter name	Default value	Description
SampleSize	20.000	The sample size of articles to use when computing the data.

resultType: conceptAggr

Last part of parameter name	Default value	Description
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
ConceptCount	25	The number of concepts per type to return.
SampleSize	1.000	The sample size of articles to use when computing the data.

resultType: keywordAggr

Last part of parameter name	Default value	Description
Lang	eng	[eng, deu, spa, zho, slv] The desired language of the computed keywords.

resultType: conceptGraph

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?

ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	25	The number of concepts to include in the graph.
LinkCount	50	The number of graph edges to include.
SampleSize	500	The sample size of articles to use when computing the data.

resultType: conceptMatrix

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	25	The number of concepts to return.
Measure	pmi	[pmi, pairTfIdf, chiSquare] The measure used when computing how much pairs of concepts are co-occurring. Possible measures are pointwise mutual information (pmi), pair frequence * IDF of individual concepts (pairTfIdf) and chi-square (chiSquare)
SampleSize	500	The sample size of articles to use when computing the data.

resultType: trendingConcepts

Last part of parameter name	Default value	Description
ConceptType	person,org,loc,wiki	[person, org, loc, wiki] Which types of the concepts to include?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
ConceptCount	10	The number of returned concepts

Possible result types for getArticle action

resultType: info

Last part of parameter name	Default value	Description
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
BodyLen	300	Maximum length of the body to return. Use -1 for whole content.
+ article details parameters (use prefix “info”)		

resultType: similarArticles

Last part of parameter name	Default value	Description
Lang	eng	[eng, deu, spa, zho, slv] In which language should be the similar articles?
Page	0	Which page of results to return.

Count	25	Number of returned articles per page.
SortBy	cosSim	[date, id, cosSim] The method used for ordering articles. Valid values are by date (date), by id (id), or by similarity to the centroid (cosSim).
SortByAsc	false	[true, false] Should the order be increasing?
ConceptLang	eng	[eng, deu, spa, zho, slv] In which language(s) should be the labels of the concepts?
BodyLen	300	Maximum length of the body to return. Use -1 for whole content.
+ article details parameters (use prefix "similarArticles")		

resultType: duplicatedArticles

Last part of parameter name	Default value	Description
Page	0	Which page of results to return.
Count	25	Number of returned articles per page.
SortBy	cosSim	[date, id, cosSim] The method used for ordering articles. Valid values are by date (date), by id (id), or by similarity to the centroid (cosSim).
SortByAsc	false	[true, false] Should the order be increasing?
ConceptLang	eng	[eng, deu, spa, zho, slv] in which language(s) should be the labels of the concepts
BodyLen	300	Maximum length of the body to return. Use -1 for whole content.
+ article details parameters (use prefix "duplicatedArticles")		

Examples of URL requests

The bottom request searches for articles that mention keywords apple and iphone and were published between April 16th and April 28th 2014. The returned information should contain list of last 30 articles.

http://eventregistry.org/json/article?action=getArticles&keywords=+apple+iphone&articlesCount=30&dateStart=2014-04-16&dateEnd=2014-04-28&resultType=articles&articlesBodyLen=100&articlesPage=0&articlesSortBy=date&articlesConceptLang=eng

The next request asks for information related to one particular article. The return information should have the default information about the article as well as concepts of type location and organization and source information.

http://eventregistry.org/json/article?resultType=info&action=getArticle&articleUri=http%3A%2F%2Fwww.kentucky.com%2F2014%2F07%2F17%2F3340248%2Fofficial-malaysian-plane-shot.html&infoConceptType=org&infoConceptType=loc&infoBodyLen=-1&infoIncludeImage=True&infoIncludeLocation=True&infoIncludeSourceInfo=True

Example of article information

```
{
    // title and body of the article
    "title": "Official: Malaysian plane carrying 295 people shot down over Ukraine",
    "body": "KIEV, Ukraine -- A Ukrainian official said a Malaysian passenger plane carrying 295 people...",
    "eventUri": "997019",          // event uri to which the article is assigned to
    "id": "15802495",              // internal id of the article
    "isDuplicate": false,          // is the article a duplicate of another article?
    "lang": "eng",                 // language in which the article is written
    "sim": 0.9803,                 // similarity of the article to the centroid of the event cluster
```



```
        "sourceId": "740",                  // internal id of the news source
        "sourceTitle": "WWW.SYRACUSE.COM",   // title of the news source
        "sourceUri": "www.syracuse.com",          // URL of the news source
        "date": "2014-07-17"                        // date when the article was published
        "time": "15:51:00",                         // time when the article was published
        "uri":
"http://www.syracuse.com/news/index.ssf/2014/07/official_malaysian_plane_carrying_295_people_shot_down_over_
ukraine.html",                                      // URL of the article
        "categories": [                             // DMoz categories for the article
          {
            "label": "Science/Technology/Aerospace",
            "uri": "http://www.dmoz.org/Science/Technology/Aerospace",
            "wgt": 100.0
          },
          ...
        ],
        "concepts": [                       # annotated concepts for the article
          {
            "id": "158",
            "labelEng": "Malaysia",
            "score": 3.0,
            "type": "loc",
            "uri": "http://en.wikipedia.org/wiki/Malaysia"
          },
          ...
        ]
}
```