

# XLike

## Deliverable 6.3.1

### API specification and prototype

Editor:	Blaz Fortuna, JSI
Author(s):	Blaz Fortuna, JSI; Blaz Sovdat, JSI; Esteban García Cuesta, ISOCO
Deliverable Nature:	Prototype (P)
Dissemination Level: (Confidentiality)	Public (PU)
Contractual Delivery Date:	M18
Actual Delivery Date:	29.7.2013
Suggested Readers:	Developers creating applications on top of XLike infrastructure
Version:	1.0
Keywords:	Text Analytics, JavaScript API

---

**Disclaimer**


---

This document contains material, which is the copyright of certain XLike consortium parties, and may not be reproduced or copied without permission.

All XLike consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the XLike consortium as a whole, nor a certain party of the XLike consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Full Project Title:	XLike – Cross-lingual Knowledge Extraction
Short Project Title:	XLike
Number and Title of Work package:	WP6
Document Title:	D6.3.1 - API specification and prototype
Editor (Name, Affiliation)	Blaž Fortuna, JSI
Work package Leader (Name, affiliation)	Esteban García Cuesta, ISOCO
Estimation of PM spent on the deliverable:	12 PM

**Copyright notice**

© 2012-2014 Participants in project XLike

## Executive Summary

This report documents the APIs for complex analysis of news articles, social media and other resources. The API is based on QMiner, an analytics platform for large-scale data stores and real-time streams containing structured and unstructured data. It is designed to for scaling to millions of instances on high-end commodity hardware, providing efficient storage, retrieval and analytics mechanisms with real-time response. The goal is not to replace the traditional relation databases (e.g. MS SQL Server, Oracle, MySQL), but to enable or make affordable scenarios not possible so far within traditional setups.

The functionality is exposed through a JavaScript API by using Google V8 JavaScript engine. This allows for easy entry level from programing point of view, high flexibility (since it's a scripting language) and scalability. The APIs provides support for rich querying (using custom query language), complex aggregation and text analytics on top of news articles, social media and other resources. Flexible data layer enables extension to other domains, for example internet-of-things.

The API is implemented in an open-source prototype, with the goal of establishing a go-to tool for large scale analytics over text and other types of data.



## Table of Contents

Executive Summary .....	3
Table of Contents .....	5
List of Figures.....	6
List of Tables.....	7
Abbreviations.....	8
Definitions .....	9
1 Introduction .....	10
2 QMiner .....	11
2.1 Architecture .....	11
2.2 Data Layer .....	11
2.3 Analytics Layer .....	12
2.3.1 Aggregates .....	12
2.3.2 Feature Extractors .....	13
2.3.3 Machine Learning .....	14
3 Storage and Retrieval.....	15
3.1 Data Schema .....	15
3.2 Query Language .....	16
4 JavaScript API .....	19
4.1 Overview .....	19
4.2 Object Model .....	19
4.2.1 Core Objects.....	19
4.2.2 Support Objects .....	21
4.3 Example.....	22
5 Prototype.....	23
5.1 Installation .....	23
5.2 Command-line.....	23
5.3 JavaScript Files .....	24
6 Conclusions .....	25
References.....	26
Annex A Data schema .....	27
A.1 Year 1 visualization prototype .....	27
A.2 Social Media Recommendation .....	28

---

## List of Figures

Figure 1. QMiner architecture .....	11
Figure 2. Example store .....	12
Figure 3. Typical learning pipeline in QMiner.....	13

---

## List of Tables

Table 1. Supported data types. .... 16

## Abbreviations

WP	Workpackage
API	Application Programming Interface
SVM	Support Vector Machine
BOW	Bag-of-Words
GUI	Graphical User Interface

## Definitions

JSon

JavaScript object

# 1 Introduction

This report documents the APIs which were developed within XLike project for advanced querying and processing of news articles, social media and other resources. The goal is to provide a flexible system, which can ingest datasets acquired in WP1, together with all the additional meta-data extracted by WP2, WP3 and WP4. The developed system provides an API, which can serve the visualizations in WP5 and, most importantly, provide rapid development platform for the use-case pilots.

The API is based on QMiner open-source project [1], a joint development between Jozef Stefan Institute and Quintelligence. It is an analytics platform for large-scale data stores and real-time streams containing structured and unstructured data. It is designed to for scaling to millions of instances on high-end commodity hardware, providing efficient storage, retrieval and analytics mechanisms with real-time response. The goal is not to replace the traditional relation databases (e.g. MS SQL Server, Oracle, MySQL), but to enable or make affordable scenarios not possible so far within traditional setups.

Several improvements and additions were made to the QMiner within XLike project: extension of the machine learning JavaScript API with emphasis on text data, and improved I/O JavaScript APIs (http, file system). The project is currently still under the process of documenting, and the source-code and installation will be made public in the autumn of 2013 on GitHub [1].

The report is structured as follows. Chapter two provides an overview of the architecture, and provides details on data layer, feature extraction and machine learning layers. Chapter three provides details on data layer schema definition. Chapter four provides details on JavaScript API, which is the primary way for accessing the API. Chapter five provides examples based on the developed API which were used so far in XLike project.

## 2 QMiner

### 2.1 Architecture

QMiner provides and integration of NoSQL-like storage backend with machine learning algorithms. The integration allows for sharing of resources between analytics and storage layers, reducing the redundancy in data structure. For example, free-text index and vector-space model can share the pre-processing (tokenization, normalization, etc.) and vocabulary, result in lower memory footprint and lower latency when operating on streaming data.

QMiner architecture consists of several layers, as can be seen in Figure 1. The data is located at the bottom of the architecture diagram, and is stored either externally (e.g. in a database) or internally. Data Layer accesses the data through adapters, which must expose the data sources through a predefined interface. Data Layer provides efficient access to the data by indexing the records, and providing means to sample given a distribution over the records. Analytics layer provides support to define and construct feature vectors out of records and implements several machine learning algorithms, which can be applied to them. All implemented algorithms leverage the support provided by Data Layer. The system can be accessed via JavaScript API, located in the top layer.

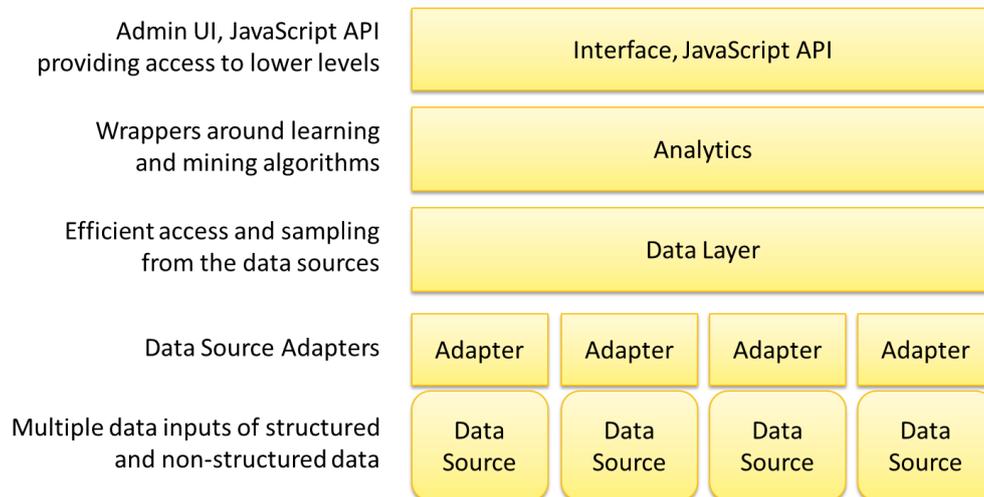


Figure 1. QMiner architecture

The system is fully developed in C++, and can run on Linux or Windows.

In the remainder of this chapter we will check in more details the functionality provided by Data Layer and Analytics layer. The following chapters will focus on the data layer schema and the functionality exposed through the JavaScript API.

### 2.2 Data Layer

The basic Data Layer abstractions are similar to databases. The data is organized around *stores* (tables). One data point inside a store corresponds to a *record* (row or instance) and one record consists of one or more *fields* (columns or features). An application can have more stores. Each record is assigned a unique 64 bit ID. The store is required to implement fast retrieval of records using the ID, preferably in constant time. Example of a store is presented in Figure 2.

The data can be stored in external database, or internally in QMiner. When data is stored externally, a C++ adapter needs to be written, which exposed in accordance with the above abstraction. When data is stored

internally, data schema needs to be defined, which declares the stores, fields and their types. Details on how schema is defined and what data types it supports are given in Chapter 3.

ID	Name	Gender
1	Davis Matthew Corbett	Male
2	Solenberger Ben	Male
3	Geisler Charlene	Female
4	Rose Stefany	Female
5	Donnelly Steve	Male
...		

**Figure 2. Example store**

Data Layer provides support for indexing and retrieving records. Currently, the system contains several indices:

- **Inverted Index** -- used to index discrete values and free text; provides support for document tokenization and word normalization.
- **Geospatial Index** – which can be used to index geographic locations presented as longitude and latitude pairs.

There are several additional indices in implementation at the moment, which will extend the system:

- **B-tree** – used to index linearly ordered data types, such as number and dates; the main benefit of tree structure is in answering range queries
- **Local Proximity Hashing** – used to answer nearest neighbour queries on high-dimensional data such as sparse vectors.

Indices can be accessed directly through a JSon-based query languages, detailed in the following Chapter. The indices are also used by Analytics layer for various tasks, such as describing and extracting features, and for sampling.

## 2.3 Analytics Layer

Analytics layer provides tools and algorithms for doing various analytics tasks, which can be composed together using JavaScript API. They can be grouped into several functionality areas: aggregates, feature extractors, and machine learning and data mining algorithms.

### 2.3.1 Aggregates

Aggregates correspond to algorithms which can take a set of records (or a stream), and produce (or maintain) some aggregate statistics of the set (or a stream). There are two types of aggregates: batch and online. Batch aggregates can be applied to a static set of records, and produce a one-time result. Online aggregates can connect to a store, and update their state as new records are added to the store.

QMiner contains the following batch aggregates:

- **Histogram** – computes a histogram over a numeric field, and provides some basic statistics (mean, median, minimum, maximum, standard deviation)
- **Pie chart** – computes a distribution over a discrete field (e.g. string, integer); it can leverage inverted index over a field when available for faster computation
- **Keywords** – computes top keywords using TFIDF weighting schema over a text field
- **Timeline** – computes distribution over a date-time field
- **Document Atlas** – computes Document Atlas[2] visualization over a text field

QMiner contains the following online aggregates, which work online on a data stream:

- **Counting** – number of records in a given time window
- **Numeric** – computes basic statistics of a numeric stream (mean, median, minimum, maximum, standard deviation)
- **Item** – counts occurrences of items (e.g. keywords or categories)
- **Exponential Moving Average** – moving average with exponential decay

### 2.3.2 Feature Extractors

Feature extractors are one of the core elements in QMiner. They can take a record, and transform it into a sparse feature vector. Feature vectors can either be consumed outside QMiner by the application, or internally using aggregators and machine learning algorithms. This abstraction enables support for both structured and unstructured data. Figure 3 shows an implementation of a typical machine learning scenario in QMiner.



**Figure 3. Typical learning pipeline in QMiner**

There are two types of feature extractors. The primary feature extractor provides feature vector directly. Secondary feature extractors can combine the outputs of several primary or secondary feature extractors.

QMiner contains the following primary feature extractors:

- **Numeric** – requires numeric field, returns its normalized value
- **Nominal** – requires discrete field (e.g. string, date), returns a sparse vector of dimensionality equal to the range of the field, with the element corresponding to the actual value of the field set to 1
- **Multinomial** – requires discrete field (e.g. string, vector of strings), returns sparse vector of dimensionality equal to the range of the field, with the elements corresponding to the actual values of the field set to 1
- **Vector-space** – require text field, returns a bag-of-words sparse vector using TFIDF weighting schema
- **Random** – returns a random number
- 

QMiner contains the following secondary feature extractors:

- **Join** – applies given feature extractor to a set of records which are obtained using a join
- **Pair** – applies a pair of feature extractors to a record, and does a Cartesian product on their outputs

### 2.3.3 Machine Learning

Machine learning algorithms take in the input feature vectors extracted with Feature Extractors. The output depends on the specific algorithm and its implementation. The implementation can learn in batch or streaming scenarios, however current implementations mostly focus on batch mode learning and application in stream.

The system contains several standard algorithms:

- **Support Vector Machine (SVM) classification, regression and ranking** [6] – State-of-the-art algorithm for text classification. Works well with high-dimensional data and can be optimized for sparse vector representation. Implementation is based on stochastic gradient descent, which can take advantage of the sampling operations supported in QMiner. Ranking can be used to learn custom ranking function, example of which is used in cross-lingual news recommendation in the Bloomberg use-case. Currently only batch learning is supported and the resulting models can be serialized to disk for later reuse.
- **K-means clustering** [7] – Standard unsupervised learning approach. The result is a collection of  $k$  sets of records, which are grouped based on the similarity of their extracted features. Currently supported measures are cosine (works well on high-dimensional data) and Euclid (works well on low-dimensional data).
- **Agglomerative clustering** [8] – Unsupervised learning approach, which returns a dendrogram of records. It has high computational complexity ( $O(n^2)$ , where  $n$  is the number of records). Current application area is to cluster geographic locations, for user-friendly display on the world map.
- **Hoeffding tress**<sup>1</sup> [9] – Decision trees learning algorithm optimized for stream processing.
- **Active learning**<sup>2</sup> [10] – Semi-supervised learning used to train classification models from unlabelled data. The implementation is based on uncertainty sampling and SVM models.

---

<sup>1</sup> Currently under development

<sup>2</sup> Currently in the process of integration

## 3 Storage and Retrieval

### 3.1 Data Schema

QMiner contains an internal data store, which can be used to store the records. It requires configuration and schema definition before it can be used.

The implementation breaks down each record into two parts, one which is always to be available in memory, and one which can be kept on hard-drive, with only the most recently used ones being cached in memory. This enables fast access to fields, which are needed for tasks which need fast access to a particular field from a large set of records. Example of this would be ranking millions of documents. Typically, ranking algorithms would require access to fields such *number of words* and *date*. However, once the records are ranked, fields such as title and content are needed only from the top 10 documents, and can be easily stored on disk.

The store supports two types of windowing:

- Time based – only records from most recent  $T$  seconds are kept,
- Record-count based – only last  $N$  records are kept.

The following commented JSON demonstrates how store schema is defined. Examples from XLike use-cases can be seen in Annex A. Table 1 lists supported data types and their names in the schema definition JSON.

```
{
  // basic store declarations
  "id" : 0, // optional, automatically assigned when not specified
  "name" : "Movies", // store name must be specified
  // list of fields
  "fields": [
    {
      "name" : "Title", // field name, obligatory
      "primary" : true, // this field corresponds to record name; default is false; can be
                        // selected only for one field in a store
      "type" : "string", // field type, obligatory, see table below for all the options
      "null" : false, // can the field be left empty, default is false
      "store" : "cache", // how is the fields stored (options: memory, cache), default is memory
      "default" : value, // default value for field; not required
      // following parameters are specific for strings (some other types as well?)
      "codebook" : true, // true if the values of a field be stored in a codebook,
                        // and use just IDs to represent them internally
      "shortstring" : true, // true if strings are no longer than 127 characters
    },
    ...
  ],
  // list of joins
  "joins": [
    {
      "name" : "Director", // join name, obligatory
      "type" : "field", // for field joins, automatically create corresponding field
      "store" : "People", // name of store joined using "Director" join
    }
  ]
}
```

```

    "inverse" : "Directed" // name of inverse join, when it exists
  },
  ...
],
// list of keys
"keys" : [
  {
    "field" : "Title", // values from which field are indexed using this key
    "type" : "text", // type of index, options: text, value, location
    "vocabulary" : "voc_01" // not necessary, provides means to share same vocabulary between
                          // multiple keys to save on memory
  }
  ...
],
// window parameters; "window" and "timeWindow" cannot be used at the same time
"window" : 1000, // keep only last 1000 records
"timeWindow" : { // keep only records within specified time window
  "duration" : 24, // duration of the time window
  "unit" : "hour", // unit the duration
  "field" : "PublicationTime" // which field to use for the record time information
                          // when non specified insertion time is used
}
}
}

```

DATA TYPE	SCHEMA NAME
Integer	int
Vector of integers	int_v
Unsigned 64 bit integer	uint64
String	string
Vector of strings	string_v
Boolean	bool
Double precision number	float
Pair of double precision numbers	float_pair
Vector of double precision numbers	float_v
Date and time	datetime
Sparse vector	num_sp_v
Bag-of-words sparse vector	bow_sp_v

**Table 1. Supported data types.**

## 3.2 Query Language

The query language is inspired by MongoDB [4] and Freebase [5] JSON-like query languages. It is based on JSON, which allows to easily construct and manipulate queries directly in JavaScript. The remainder of this section shows various constructs, which can be achieved through the queries. Examples can be seen in Section 4.3.

Simple AND queries look as follows:

```
{ $from: <store>, <field1>: <value1>, <field2>: <value2> }
```

If there are more values for same field (e.g. words for a text field), they can be passed as an array. By default, AND is assumed between the values.

```
{ $from: <store>, <field>: [<value1>, <value2>] }
```

To use OR, one must be explicit:

```
{ $from: <store>, $or : [{<field1>: <value1>}, {<field2>: <value2>}] }
```

For more possible values of single field:

```
{ $from: <store>, <field1>: { $or : [<value1>, <value2>] } }
```

Similarly, NOT operator can be specified as follows:

```
{ $from: <store>, <field1>: <value1>, $not : { <field2>: <value2> } }
```

Range operators (<, >) and non-equality (!=) are used as follows:

```
{
  $from: <store>,
  <field1>: {$ne: <value1>},
  <field2>: {$gt: <value2>},
  <field3>: {$lt: <value3>}
}
```

Location queries can be limited by radius (in meters) or by number of records:

```
{
  $from: <store>,
  <field1>: { $location: [<latitude>, <longitude>], $radius: <value_in_meters> },
  <field2>: { $location: [<latitude>, <longitude>], $limit: <value> }
}
```

Records can be mapped using any defined joins. Result is a set of records from the joined store:

```
{
  $join: {
    $name: <join_name>,
    $query: { $from: <store>, ...}
  }
}
```

We can have more joins:

```
{
  $join: [
    { $name: <join_name1>, $query: { $from: <store1>, ...} },
    { $name: <join_name2>, $query: { $from: <store2>, ...} }
  ]
}
```

Joins can be sampled by providing absolute size of the sample:

```
$join: {
  $name: <join_name>,
  $query: { $from: <store>, ...},
  $sample: <sample_size>
}
}
```

Queries can be nested. Only stores of leaf nodes need to be specified and the rest are inferred from there. In the following example, the resulting records come from joined store:

```
{
  $join: {
    $name: <join_name>
    $query: { $from: <store>, <field1>: <value1> }
  }
  <field2>: <value2>
}
```

Query can ask for specific record (by name or by id):

```
{ $from: <store>, $record: <record_name> }
```

```
{ $from: <store>, $record: <record_id> }
```

This is useful for queries with joins that start from specific record:

```
{
  $join: {
    $name: <join_name>,
    $query: { $from: <store>, $record: <record_name> }
  }
}
```

Records can be ordered already by the query engine using the following operations:

- Sorting according to the specified field and direction (1 = ascending, -1 = descending)
- Limiting number of results
- Offsetting (skipping specified number of first records, useful for paging on websites)

```
{
  $from: <store>,
  <field1>: <value1>,
  $sort: { <field2>: 1 },
  $limit: <number_of_returned_records>,
  $offset: <number_of_first_records_to_skip>
}
```

## 4 JavaScript API

### 4.1 Overview

The functionality is exposed through a JavaScript API by using Google V8 [3]. This allows for easy entry level from programming point of view, high flexibility (since it's a scripting language) and scalability (V8 is implemented in C++ making for low overheads when moving data between C++ backend and JavaScript, data-intensive objects are all implemented in C++ and very little data is actually moved between the scripting layer and the processing layer; everything is running in the same process and C++ and JavaScript objects share memory space).

### 4.2 Object Model

#### 4.2.1 Core Objects

##### 4.2.1.1 qm

This object corresponds to QMiner core, and exposes access to stores and indices. There is only one JS instance of qm object.

#### Functions:

---

<code>s = qm.store(storeName)</code>	Returns the store with name <i>storeName</i> ; null when no such store
<code>rs = qm.search(query)</code>	Executes <i>query</i> and returns results as <i>RecSet</i> object
<code>qm.featureSpace</code>	
<code>qm.featureExtractor</code>	

---

##### 4.2.1.2 Store

#### Properties:

---

<code>s.name</code>	Name of the store
<code>s.empty</code>	True when store is empty
<code>s.length</code>	Number of records in the store
<code>rs = s.recs</code>	Returns record set with all the records from the store
<code>s.fields</code>	Array of all field names
<code>s.joins</code>	Array of all join names

---

#### Functions:

---

<code>r = s[recId]</code>	Record with id <i>recId</i>
<code>r = s.rec(recName)</code>	Record with name <i>recName</i>
<code>s.add(record)</code>	Add <i>record</i> to the store
<code>rs = s.sample(sampleSize, seed)</code>	Return record set with <i>sampleSize</i> randomly selected records; default value for <i>seed</i> is 0
<code>s.addTrigger(trigger)</code>	Adds <i>trigger</i> to the store

---

### 4.2.1.3 Triggers

Triggers are call-backs which can be attached to the stores, and are executed in the following cases:

- after new record is added to the store (*onAdd*),
- after record is updated (*onUpdate*),
- before record is deleted from the store (*onDelete*).

Not all call-backs are required and get new, updated or deleted record as parameter.

#### Example:

```
qm.store("People").addTrigger({
  onAdd : function (person) { console.say("New record: " + person.Name); },
  onUpdate : function (person) { console.say("Updated record: " + person.Name); },
  onDelete : function (person) { console.say("Deleted record: " + person.Name); }
});

qm.store("Movies").addTrigger({
  onUpdate : function (movie) { console.say("Updated record: " + movie.Title); }
});
```

### 4.2.1.4 Record Set

This is a result-holding QMiner class with the following operations.

#### Properties:

---

<code>rs.store</code>	Store corresponding to records in the set
<code>rs.empty</code>	True if record set is empty
<code>rs.length</code>	The number of records in the set
<code>rs.weighted</code>	True if records are weighted
<code>r = rs[n]</code>	Returns <i>n</i> -th record from the set

---

#### Functions:

---

<code>rs = rs.join(joinName, sampleSize = -1)</code>	Returns new record set obtained by doing join <i>joinName</i>
<code>rs = trunc(recs)</code>	Keep only first <i>recs</i> records
<code>rs = r.sample(sampleSize, seed = 0)</code>	Return new record set with <i>sampleSize</i> records.
<code>rs.filterByFq(minFrequency, maxFrequency)</code>	Keep only records with frequency <i>minFrequency</i> ≤ record frequency ≤ <i>maxFrequency</i>
<code>rs.filterByIntField(field, minVal, maxVal)</code>	Keep only records with field value between <i>minVal</i> and <i>maxVal</i>

---

---

<code>rs.filterByRecId(minId, maxId)</code>	Keep only records with IDs between <i>minId</i> and <i>maxId</i>
<code>rs.reverse()</code>	Reverse the order of records
<code>rs.shuffle(seed = 0)</code>	Shuffle records inside record set
<code>rs.sortByField(field, Asc = 1)</code>	Sort records by <i>field</i>
<code>rs.sortByFq(Asc = 1)</code>	Sort records by frequency
<code>rs.sortById(Asc = 1)</code>	Sort records by ID.

---

The ``Asc`` parameter in the following operations denotes whether to sort records in ascending (``Asc = 1``) or descending (``Asc != 1``) order.

#### 4.2.1.5 Record

Record objects have the following operations exposed.

##### Properties:

---

<code>r.id</code>	Record ID
<code>r.&lt;Field&gt;</code>	Record fields are exposed as object properties

---

##### Functions:

---

<code>rs = r.join(joinName)</code>	Returns new record set obtained by doing join <i>joinName</i>
<code>rs = r.sjoin(joinName)</code>	Returns new record obtained by doing join <i>joinName</i> ; when join results in more than one record, the first one is returned; when no joined record exists, null is returned

---

#### 4.2.2 Support Objects

The API also contains several support objects, which are mainly used for handling input and output:

- **http** – serves as an HTTP server, providing easy way to define REST web services, and HTTP client, providing support for executing other REST services
- **console** – primarily used for logging progress to the console
- **fs** – access to the file system for reading and writing text file, and (de)serialization of objects (e.g. SVM models); the object only has access to the sandbox parts of file system, details on this are in Chapter 5

### 4.3 Example

The following code snippet demonstrates how the API can be used to create a machine learning model, which can classify the movie into *horror* genre, based on the title, plot, actors, director and rating.

```
// input
var positives = qm.search({'$from':'Movies', 'Genre':'horror'});
var negatives = qm.search({'$from':'Movies', '$not':{'Genre':'horror'}});
// declare feature space
var ftrSpace = qm.newFeatureSpace([
  { $type: 'text', store: 'Movies', field: 'Title' },
  { $type: 'text', store: 'Movies', field: 'Plot' },
  { $type: 'join', store: 'Movies', join: 'Actor' },
  { $type: 'join', store: 'Movies', join: 'Director' },
  { $type: 'numeric', store: 'Movies', field: 'Rating', normalize: true }
]);
// define feature space
ftrSpace.update(positives);
ftrSpace.update(negatives);
ftrSpace.finishUpdate();
// SVM training set
trainSet = svm.newTrainSet();
trainSet.add(ftrSpace, positives, 1.0, true);
trainSet.add(ftrSpace, negatives, -1.0, true);
// SVM parameters
var svmParam = { c = 1.0, j = 1.0, eps= 0.1 };
// learn
var svmCfyModel = svm.trainClassify(trainSet, svmParam);
// use
var isHorror = svmCfyModel.classify(movie);
```

## 5 Prototype

### 5.1 Installation

The developed prototype runs as a standalone server, and communicates over a port, specified in the configuration file. Typically, one instances would correspond to one project (database), and multiple instances can run simultaneously on the same computer.

Installation directory contains the following files and subdirectories:

- *qm* - QMiner executable
- *gui/* - administration GUI
- *lib/* - available JavaScript libraries (can be included using 'require')
- *watchdog/* - QMiner failover wrapper

By default, each QMiner project is assumed to follow a standard directory structure:

- *db/* - directory in which QMiner data files are stored
- *src/* - directory containing JavaScript code of core files
- *src/lib/* - directory with project specific JavaScript libraries
- *sandbox/* - host for directories available to JavaScript libraries
- *restore/archive* - last successful database dump
- *restore/journal* - entries since the last successful dump
- *qm.conf* - configuration file

### 5.2 Command-line

This section demonstrates how to initialize, start and stop QMiner instances from command-line. The following commands assume we have the directory, in which QMiner is installed, in the PATH variable, and that we have set QMINER\_HOME environment variable to point to the same directory. Commands are executed from the directory in which we want to host the project.

**Create directory structure and the basic *qm.conf* file:**

- *qm config* - setup using default parameters (port=8080, cache=1024)
- *qm config -port=8080 -cache=2048* - setup using the specified port and cache size

**Initialize empty database:**

- *qm create* - creates generic base and initializes index
- *qm create -def=schema.def* - creates generic base and empty stores defined in `schema.def`

**Start QMiner engine:**

- *qm start* - runs qminer engine using the `./qm.conf` configuration file
- *qm start -conf=custom.conf* - runs qminer engine using the custom configuration file `custom.conf`
- *qm start -rdonly* - runs qminer engine in read-only mode (default is update)

**Stop QMiner engine:**

- `qm stop` - stops qminer engine started using `./qm.conf` configuration file
- `qm stop -conf=custom.conf` - stops qminer engine started using custom configuration file

**Reload specified javascript file**

- `qm reload -name=script` - reloads script named `script`

## 5.3 JavaScript Files

On start QMiner loads and executes all the scripts located in `./src/` folder and initializes the server. If the scripts assume any start-up parameters, they can be passed to QMiner using command-line parameters. For example: `qm start -cid=12345`. The parameters are available to all scripts using `qm.argv` property.

The scripts can register custom web services as follows:

```
http.request('name', function (req, res) { ... });
```

Web services live in the namespace of individual scripts. In the above example, assuming the script was located in `script.js`, the web service URL would be `http://localhost:8080/script/name`. Any parameters and body of request are included in `req`.

Once the engine starts, it creates a server listening on the specified port, default being 8080. It serves administration interface, located at `/admin/`, which can be used to define stores and create and edit scripts.

## 6 Conclusions

This report documents the APIs for complex analysis of news articles based on QMiner system. The APIs is based on JavaScript, and provides functions for rich querying, aggregation and text analytics on top of news articles. Besides news articles, flexible data layer enables application to other similar data sets.

The developed prototype was already used to support use-cases in the first year of XLike project.

The project is currently still under the process of documenting, and the source-code and installation will be made public in the autumn of 2013 on GitHub [1].

## References

- [1] QMiner: <https://github.com/quintelligence/QMiner>
- [2] Visualization of Text Document Corpus Informatica Journal, Vol. 29, No. 4., pp. 497-502 by B. Fortuna, D. Mladenic, M. Grobelnik
- [3] Google V8 JavaScript Engine: <https://code.google.com/p/v8/>
- [4] MongoDB: <http://www.mongodb.org/>
- [5] Freebase: <http://www.freebase.com/>
- [6] Nello Cristianini and John Shawe-Taylor. 1999. An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods. Cambridge University Press, New York, NY, USA
- [7] A. K. Jain, M. N. Murty, and P. J. Flynn. 1999. Data clustering: a review. ACM Comput. Surv. 31, 3 (September 1999), 264-323
- [8] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA
- [9] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '00). ACM, New York, NY, USA, 71-80
- [10] Meng Wang and Xian-Sheng Hua. 2011. Active learning in multimedia annotation and retrieval: A survey. ACM Trans. Intell. Syst. Technol. 2, 2, Article 10 (February 2011), 21 pages

## Annex A Data schema

### A.1 Year 1 visualization prototype

```
[
  {
    "name" : "Article",
    "fields" : [
      { "name" : "URI", "type" : "string", "primary" : true },
      { "name" : "Language", "type" : "string", "codebook" : true },
      { "name" : "DateTime", "type" : "datetime" },
      { "name" : "Title", "type" : "string", "store" : "cache" },
      { "name" : "Body", "type" : "string", "store" : "cache" },
      { "name" : "Keywords", "type" : "string_v", "store" : "cache" },
      { "name" : "BodyLen", "type" : "int" },
      { "name" : "NewsfeedId", "type" : "string", "null" : true }
    ],
    "joins" : [
      { "name" : "hasSource", "type" : "field", "store" : "Source", "inverse" : "hasArticle" },
      { "name" : "hasStory", "type" : "field", "store" : "Story", "inverse" : "hasArticle" },
      { "name" : "hasConcept", "type" : "index", "store" : "Concept", "inverse" : "hasArticle" },
      { "name" : "hasImage", "type" : "index", "store" : "Concept" },
      { "name" : "similar_spa", "type" : "index", "store" : "Article" },
      { "name" : "similar_deu", "type" : "index", "store" : "Article" },
      { "name" : "similar_zho", "type" : "index", "store" : "Article" },
      { "name" : "similar_eng", "type" : "index", "store" : "Article" }
    ],
    "keys" : [
      { "field" : "Language", "type" : "value" },
      { "field" : "Title", "type" : "text", "vocabulary" : "text" },
      { "field" : "Body", "type" : "text", "vocabulary" : "text" }
    ],
    "timeWindow" : {
      "duration" : 14,
      "unit" : "day",
      "field" : "DateTime"
    }
  },
  {
    "name" : "Source",
    "fields" : [
      { "name" : "URI", "type" : "string", "primary" : true },
      { "name" : "Title", "type" : "string" },
      { "name" : "Type", "type" : "string", "codebook" : true },
      { "name" : "Location", "type" : "float_pair", "null" : true },
      { "name" : "Country", "type" : "string", "codebook" : true, "null" : true },
      { "name" : "City", "type" : "string", "codebook" : true, "null" : true },
      { "name" : "Count", "type" : "int", "default" : 0 }
    ],
    "joins" : [
      { "name" : "hasArticle", "type" : "index", "store" : "Article", "inverse" : "hasSource" }
    ],
    "keys" : [
      { "field" : "URI", "type" : "value" },
      { "field" : "Title", "type" : "value" },

```

```

        { "field" : "Type", "type" : "value" },
        { "field" : "Country", "type" : "value" },
        { "field" : "City", "type" : "value" }
    ]
},
{
    "name" : "Story",
    "fields" : [
        { "name" : "ID", "type" : "string", "primary" : true },
        { "name" : "Language", "type" : "string", "codebook" : true }
    ],
    "joins" : [
        { "name" : "hasArticle", "type" : "index", "store" : "Article", "inverse" : "hasStory"
},
        { "name" : "hasMedoid", "type" : "field", "store" : "Article" }
    ],
    "keys" : [
        { "field" : "Language", "type" : "value" }
    ]
},
{
    "name" : "Concept",
    "fields" : [
        { "name" : "URI", "type" : "string", "primary" : true },
        { "name" : "Label", "type" : "string", "null" : true },
        { "name" : "Type", "type" : "string", "null" : true, "codebook" : true },
        { "name" : "Description", "type" : "string", "null" : true, "store" : "cache" }
    ],
    "joins" : [
        { "name" : "hasArticle", "type" : "index", "store" : "Article", "inverse" : "hasConcept"
}
    ],
    "keys" : [
        { "field" : "URI", "type" : "value" },
        { "field" : "Label", "type" : "value" },
        { "field" : "Label", "name" : "Keywords", "type" : "text", "vocabulary" : "text" },
        { "field" : "Type", "type" : "value" },
        { "field" : "Description", "type" : "text", "vocabulary" : "text" }
    ]
}
]

```

## A.2 Social Media Recommendation

```

[
{
    "name" : "Article",
    "fields" : [
        { "name" : "URI", "type" : "string", "primary" : true },
        { "name" : "Language", "type" : "string", "codebook" : true },
        { "name" : "DateTime", "type" : "datetime" },
        { "name" : "Title", "type" : "string", "store" : "cache" },
        { "name" : "Body", "type" : "string", "store" : "cache" },
        { "name" : "Keywords", "type" : "string_v", "store" : "cache" },
        { "name" : "BodyLen", "type" : "int" },
        { "name" : "BlpScore", "type" : "string", "null" : true, "codebook" : true },
        { "name" : "NewsfeedId", "type" : "string", "null" : true }
    ],
}
]

```

```

"joins" : [
  { "name" : "hasSource", "type" : "field", "store" : "Source", "inverse" : "hasArticle" },
  { "name" : "hasStory", "type" : "field", "store" : "Story", "inverse" : "hasArticle" },
  { "name" : "hasConcept", "type" : "index", "store" : "Concept", "inverse" : "hasArticle" },
  { "name" : "hasImage", "type" : "index", "store" : "Concept" },
  { "name" : "similar_blb", "type" : "index", "store" : "Article" },
  { "name" : "similar_spa", "type" : "index", "store" : "Article" },
  { "name" : "similar_deu", "type" : "index", "store" : "Article" },
  { "name" : "similar_zho", "type" : "index", "store" : "Article" },
  { "name" : "similar_eng", "type" : "index", "store" : "Article" }
],
"keys" : [
  { "field" : "Language", "type" : "value" },
  { "field" : "Title", "type" : "text", "vocabulary" : "text" },
  { "field" : "Body", "type" : "text", "vocabulary" : "text" }
],
"timeWindow" : {
  "duration" : 3,
  "unit" : "day",
  "field" : "DateTime"
}
},
{
  "name" : "Source",
  "fields" : [
    { "name" : "URI", "type" : "string", "primary" : true },
    { "name" : "Title", "type" : "string" },
    { "name" : "Type", "type" : "string", "codebook" : true },
    { "name" : "Location", "type" : "float_pair", "null" : true },
    { "name" : "Country", "type" : "string", "codebook" : true, "null" : true },
    { "name" : "City", "type" : "string", "codebook" : true, "null" : true },
    { "name" : "Count", "type" : "int", "default" : 0 }
  ],
  "joins" : [
    { "name" : "hasArticle", "type" : "index", "store" : "Article", "inverse" : "hasSource" }
  ],
  "keys" : [
    { "field" : "URI", "type" : "value" },
    { "field" : "Title", "type" : "value" },
    { "field" : "Type", "type" : "value" },
    { "field" : "Country", "type" : "value" },
    { "field" : "City", "type" : "value" }
  ]
},
{
  "name" : "Story",
  "fields" : [
    { "name" : "ID", "type" : "string", "primary" : true },
    { "name" : "Language", "type" : "string", "codebook" : true }
  ],
  "joins" : [
    { "name" : "hasArticle", "type" : "index", "store" : "Article", "inverse" : "hasStory" },
    { "name" : "hasMedoid", "type" : "field", "store" : "Article" }
  ],
  "keys" : [
    { "field" : "Language", "type" : "value" }
  ]
},
{
  "name" : "Concept",

```

```

"fields" : [
  { "name" : "URI", "type" : "string", "primary" : true },
  { "name" : "Label", "type" : "string", "null" : true },
  { "name" : "Type", "type" : "string", "null" : true, "codebook" : true },
  { "name" : "Description", "type" : "string", "null" : true, "store" : "cache" }
],
"joins" : [
  { "name" : "hasArticle", "type" : "index", "store" : "Article", "inverse" : "hasConcept" }
],
"keys" : [
  { "field" : "URI", "type" : "value" },
  { "field" : "Label", "type" : "value" },
  { "field" : "Label", "name" : "Keywords", "type" : "text", "vocabulary" : "text" },
  { "field" : "Type", "type" : "value" },
  { "field" : "Description", "type" : "text", "vocabulary" : "text" }
]
},
{
  "name" : "BlpArticle",
  "fields" : [
    { "name" : "URI", "type" : "string", "primary" : true },
    { "name" : "DateTime", "type" : "datetime" },
    { "name" : "Visits", "type" : "int" },
    { "name" : "FacebookVisits", "type" : "int" },
    { "name" : "TwitterVisits", "type" : "int" },
    { "name" : "CountrySI", "type" : "int", "null" : true },
    { "name" : "CountrySIFacebook", "type" : "int", "null" : true },
    { "name" : "CountrySITwitter", "type" : "int", "null" : true },
    { "name" : "CountryDE", "type" : "int", "null" : true },
    { "name" : "CountryDEFacebook", "type" : "int", "null" : true },
    { "name" : "CountryDETwitter", "type" : "int", "null" : true },
    { "name" : "CountryFR", "type" : "int", "null" : true },
    { "name" : "CountryFRFacebook", "type" : "int", "null" : true },
    { "name" : "CountryFRTwitter", "type" : "int", "null" : true },
    { "name" : "CountryES", "type" : "int", "null" : true },
    { "name" : "CountryESFacebook", "type" : "int", "null" : true },
    { "name" : "CountryESTwitter", "type" : "int", "null" : true },
    { "name" : "CountryIT", "type" : "int", "null" : true },
    { "name" : "CountryITFacebook", "type" : "int", "null" : true },
    { "name" : "CountryITTwitter", "type" : "int", "null" : true },
    { "name" : "CountryCN", "type" : "int", "null" : true },
    { "name" : "CountryCNFacebook", "type" : "int", "null" : true },
    { "name" : "CountryCNTwitter", "type" : "int", "null" : true },
    { "name" : "CountryBR", "type" : "int", "null" : true },
    { "name" : "CountryBRFacebook", "type" : "int", "null" : true },
    { "name" : "CountryBRTwitter", "type" : "int", "null" : true },
    { "name" : "CountryIN", "type" : "int", "null" : true },
    { "name" : "CountryINFacebook", "type" : "int", "null" : true },
    { "name" : "CountryINTwitter", "type" : "int", "null" : true }
  ],
  "joins" : [ ],
  "keys" : [ ],
  "timeWindow" : {
    "duration" : 3,
    "unit" : "day",
    "field" : "DateTime"
  }
}
]

```