QMiner: Data Analytics Platform for Processing Streams of Structured and Unstructured Data

Blaz Fortuna[1,3], Jan Rupnik[1], Janez Brank[1], Carolina Fortuna[2,3], Viktor Jovanoski[1], Mario Karlovcec[1], Blaz Kazic[1], Klemen Kenda[1], Gregor Leban[1], Andrej Muhic[1], Blaz Novak[1], Jost Novljan[2], Miha Papler[1], Luis Rei[1], Blaz Sovdat[1], Luka Stopar[1], Marko Grobelnik[1], and Dunja Mladenic[1]

[1]Artificial Intelligence Laboratory, Jozef Stefan Institute, Jamova 39, Ljubljana, Slovenia.
[2]Department of Communication Systems, Jozef Stefan Institute, Jamova 39, Ljubljana, Slovenia.
[3]Internet Based Communication Networks and Services, University of Gent, Gaston Crommerlaan 8, Gent, Belgium.
{*firstname.lastname*}*@ijs.si*

Abstract

QMiner is an open source analytics platform for performing large scale data analysis written in C++ and exposed via a Javascript API. The paper presents five main design elements which focus on focus on storage, online and real-time processing as well as fast prototyping and give QMiner unique advantages as a data analytics platform for processing streams of structured and unstructured data. These design elements are incorporated in a five layer architecture represented by 1) storage and indexing, 2) stream aggregate, 3) feature extractor, 4) linear algebra and 5) analytics layers. The functionality of the platform is demonstrated by three representative usage examples containing code fragments: text classification, time series prediction and community detection in graphs.

1 Introduction

QMiner was incrementally developed within several EU Framework Programme research projects in the areas of text, web, and stream mining over the last couple of years. The main requirements for QMiner were derived from the need for rapid developed of solutions focused on user interactivity and the ability to operate on large data sets (hundreds of gigabytes) in real-time on high-end commodity hardware. These goals and constraints resulted in a unique set of features that we implemented in QMiner.

The applications are implemented in JavaScript, making it easy for novice users to get started. Using the JavaScript API, the user can easily compose complex data processing pipelines and integrate them with other systems via RESTful web services. The core system is based on a C++ library and can be included into custom C++ projects, thus providing them with stream processing and data analytics capabilities. Event Registry[**?**] is one example of a system, which was built around the QMiner storage and analytics modules.

QMiner is available as an open source project on GitHub under AGPL licence. The repository contains source code, introduction guide, complete documentation of QMiner's JavaScript APIs and examples showcasing its various uses.

The paper is organised as follows. Section 2 describes the main design elements of the platform while Section 3 focuses on the architecture of the platform. Section 4 details three representative usage examples which showcase how one can quickly build machine-learning applications using the tool. Finally, Section 5 concludes the paper.

2 Key design elements

In this section, we provide an overview of the main design elements which give QMiner unique advantages as a data analytics platform for processing streams of structured and unstructured data.

Connecting storage, indexing and analytics: QMiner stores and indexes the data in a way that makes the implemented machine learning methods more scalable. Computing feature vectors from stored records aims at reducing duplication of data in the process and relies heavily on the integrated indexing and its features (e.g. probabilistic joins) to speed up data transformations. Data schemas provide types for directly storing vectors and sparse vectors as part of records.

Multi-modal data support: QMiner provides native support for handling and learning from unstructured data such as text or graphs. Among the supported data operations are full text search, aggregation over text fields, creation and storage of bag-of-words vectors directly in the data stores, and full integration of the Stanford graph analysis library SNAP [**?**] in the C++ and JavaScript APIs. Additionally, several time series processing tools such as resampling, merging, computing moments, extrapolation, interpolation and discretization are implemented.

Streaming processing: QMiner provides building blocks for processing and learning from streams of text documents, website logs or numeric streams. For example, pipeline-able stream aggregates for maintaining aggregate statistics over the stream, and machine learning algorithms for learning from streams are available.

Probabilistic joins between tables: For some operations, computing a complete join between tables is not necessary. For example, to compute the gender distribution of visitors of a particular web page, we do not really need a complete list of visitors for that particular web page. What suffices is a statistically representative sample. QMiner supports several sampling techniques for achieving this, and integrates the support for probabilistic joins in the query language and feature extractors.

Efficiency and fast prototyping: All main components of the library are implemented in C++ and are optimized for memory usage and computational speed. This involves the data store, indexing, feature extractors, linear algebra, stream processing components, several machine learning and graph mining algorithms (the latter based on SNAP library). Most of the core functionality is exposed in the JavaScript API which enables fast prototyping and deploying as RESTful services, since QMiner can operate as an HTTP server.

3 Architecture

The core blocks of the system provide support for data storage, preprocessing and analytics and the architecture is organised in the following five layers:

Storage and indexing layer: The basic storage abstractions are similar to the ones existing in 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). The index layer provides support for indexing and retrieving records by implementing several types of indexes: (a) inverted index [**?**, section 6.5], used to index discrete values and free text, and (b) geospatial index, which can be used to index geographic locations presented as longitude and latitude pairs. There are several additional indices, currently under implementation, which will extend the system: (c) B-tree, used to index linearly ordered data types, such as number and dates, and (d) locality sensitive hashing [**?**], used to answer nearest neighbour queries on high-dimensional data such as sparse vectors. Indices can be utilized through a JSON-based query language. The indices are used by the Analytics layer for various tasks, such as describing and extracting features, and for sampling.

Stream aggregates: Aggregates are algorithms, which take a stream of records, and maintain some aggregate statistics of the stream. The aggregates connect to a store, and update their state as new records are added to the store. QMiner contains a large collection of stream aggregates, e.g. statistical moments, exponential moving average, resampling and merging of streams, interpolation and extrapolation. In addition, new stream aggregates can be implemented directly in the JavaScript layer.

Feature extractors: A feature extractor enables mapping records to linear algebra vectors (supports dense and sparse vectors). The core framework for feature extractors enables constructing and gluing several types of feature extractors and automatically performs all the necessary bookkeeping. Feature extractors include: text feature extractors (bag of words with optional stemming, stop-word removal, n-gram extraction, hashing), numeric feature extractors, element indicator vectors and set indicator vectors (multi-label). In addition, when working with the JavaScript API, custom feature extractors can be implemented as JavaScript functions.

Linear algebra: The library provides efficient representation and manipulation of linear algebra structures (dense and sparse vectors and matrices) directly in JavaScript. The library includes algorithms for solving linear systems and computing several matrix decompositions. Randomized algorithms enable solving of large scale problems, such as singular value decompositions [**?**] on large structured or sparse matrices. QMiner can optionally be compiled with optimized Basic Linear Algebra libraries such as Intel MKL, providing additional boost in performance.

Analytics: The analytics components combine all components in order to implement various popular machine learning algorithms for clustering, dimensionality reduction, classification, regression and active-learning. The implementations can be easily applied directly to data stored in the storage layer. The analytics components are either implemented purely in C++ or alternatively in JavaScript, by using core C++ modules (such as linear algebra) to retain efficiency.

4 Examples

QMiner lets you get from data to working models exposed through web service API in less than an hour. Most applications can be scripted fully in the JavaScript layer, taking advantage of the performance of components implemented in C++ and libraries such as Intel MKL. We demonstrate the system on three different tasks: text classification, time series prediction and community detection on graphs.

4.1 Text processing and classification

The example in this section demonstrates how to extract features from a movie dataset and build classification models for predicting movie genres. For a more advanced application of the platform we refer the reader to the Event Registry[**?**].

First, we import the analytics module which provides machine learning algorithms. Next, we load the data and enumerate features we would like to extract from the data set. For each feature, we need to specify its type (for example, `text`), its source (for example, `Movies`), and name (for example, `plot`). Source is the *data store* from which we want to extract the feature. We can now run a batch learner on the whole data set for training. The learner then returns the model for predicting movie genres. In the example, the model is stored in the `genreModel` variable. Finally, the model is used to predict the genre of a new, previously unseen, movie.

```
1  // Loading in the dataset.
2  qm.load.jsonFile(Movies, "./sandbox/movies/movies.json");
3  // Declare the features we will use to build classification models
4  var genreFeatures = [
5    { type: "constant", source: "Movies" },
6    { type: "text", source: "Movies", field: "Title" },
7    { type: "text", source: "Movies", field: "Plot" },
8    { type: "join", source: { store: "Movies", join: "Director"} }
9  ];
10 // Create a model for the Genres field, by training on all movies.
11 var genreModel = analytics.newBatchModel(
12    Movies.recs, genreFeatures, Movies.field("Genres"));
13 // Predict genres of a new movie
14 var newMovie = qm.store("Movies").newRec({
15   Title:"Unnatural Selection",
16   Plot:"When corpses are found with organs missing ...",
17   Genres:["Horror", "Sci-Fi"],
18   Director:{Name:"Baggs Bill", Gender:"Male"}
19 });
```

```
var predictedGenre = genreModel.predict(newMovie);
```

Code fragment 1: Example text mining: storage, feature extraction and classification.

4.2 Time series processing

The following example demonstrates how to perform time series prediction. For a more advanced example for online energy production forecasting, we refer the reader to [**?**]. The example uses stream aggregates (rounded rectangles in Figure 1) to (a) resample incoming time series into an equally spaced time series, and (b) compute two exponential moving averages (EMAs). To learn the prediction model, it constructs a feature vector containing the current numeric value of the time series, the two EMAs and the date features (time of day, day of week, month). The feature vector is sent to the incremental linear regression model. A one minute delay operator is applied to the input variables, which corresponds to a one minute prediction horizon.

The example involves two *data stores*: `Raw` and `Resampled`. The records from the `Raw` data store represent time series measurement pairs (timestamp, value) and `Resampled` stores the resampled time series, together with the results of two smoothers. Each time a record is added to the `Raw` store, the resampler is updated. The resampler may insert a record in the `Resampled` store based on the interpolation used. Each time a record is added to `Resampled` several operations are executed: computation of two smoothers, update of the delay operator, enrichment of the resampled record with the results of the smoothers, and the update of the linear regression model.

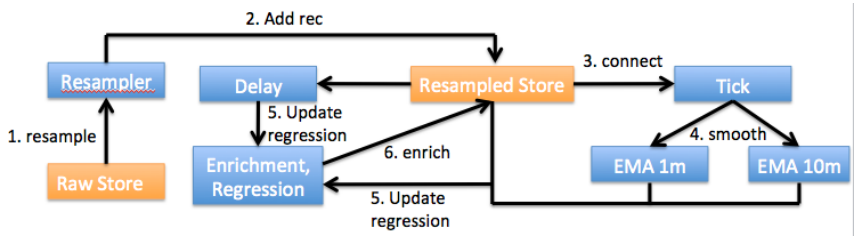

Figure 1: Example time series processing architecture

```
// Create and connect stream aggregates to the stores.
// These correspond to blue boxes in figure 1.
var sr = Raw.addStreamAggr({name: "Resamp10s", type: "resampler",
    outStore: "Resampled"/*, resampling parameters*/});
var s0 = Resampled.addStreamAggr({name: "tick", type: "timeSeriesTick",
    /*aggregate parameters*/});
var s1 = Resampled.addStreamAggr({name: "EMA1m", type: "ema", interval:
    60000, /*aggregate parameters*/});
var s2 = Resampled.addStreamAggr({name: "EMA10m", type: "ema", interval:
    600000, /*aggregate parameters*/});
var s3 = Resampled.addStreamAggr({ name: "delay", type: "recordBuffer",
    size: 6 });
// Declare features from the resampled timeseries
var ftrSpace = analytics.newFeatureSpace([
  { type: "numeric", source: "Resampled", field: "Value" },
  { type: "numeric", source: "Resampled", field: "Ema1" },
  { type: "numeric", source: "Resampled", field: "Ema2" },
  { type: "multinomial", source: "Resampled", field: "Time", datetime:
      true }
]);
// Initialize linear regression model
var linreg = analytics.newRecLinReg(/*model parameters*/);
// Add the stream aggregate that enriches the record with current
// EMAs, computes its feature vector and updates the model
var s4 = Resampled.addStreamAggr(new function() {
  onAdd: function (rec) {
    // Enrich record with latest EMAs
```

```
22      rec.Ema1 = s1.getFlt();
23      rec.Ema2 = s2.getFlt();
24      // Get the ID of the record from a minute ago.
25      var trainRecId = s3.val.last;
26      // Compute feature vector of the record
27      var ftrVec = ftrSpace.ftrVec(Resampled[trainRecId]);
28      // Update the regression model: the input is delayed
29      // by 1 minute and the output is the current value
30      linreg.learn(ftrVec, rec.Value);
31    }
32  });
```

Code fragment 2: Example time series processing

4.3 Graph analysis

QMiner supports handling of graph data by providing JavaScript API to the integrated SNAP library [**?**]. The script in Code fragment 3 is an example of loading a graph, computing communities and visualizing the results.

After importing the required modules for graph analysis and visualization, an edge-list graph file is loaded into a new undirected graph. The graph represents the co-authoring network of Slovenian researchers from biotechnical science in 2013. the nodes of the graph represent researchers while edges between nodes indicate co-authoring of at least on publication. Real networks often have community structure with several regions of internally densely connected nodes. Communities in the co-authoring network can indicate groups of strongly connected researchers from a research lab or a similar organization, or simply groups of scientist with a common research interest. The number, size and connectivity between communities reveal the cohesiveness of a network. Following the simplification of the graph by removing weakly connected nodes (with degree 3 or less), the Clauset-Newman-Moore [**?**] community detection algorithm is applied to compute the communities. In the final step, the graph is plotted using force-directed layout, with node colors corresponding to communities. The complete script in Code fragment 3 is written in JavaScript, but some of the components, such as community detection algorithm are executing in the highly optimized C++ SNAP code.

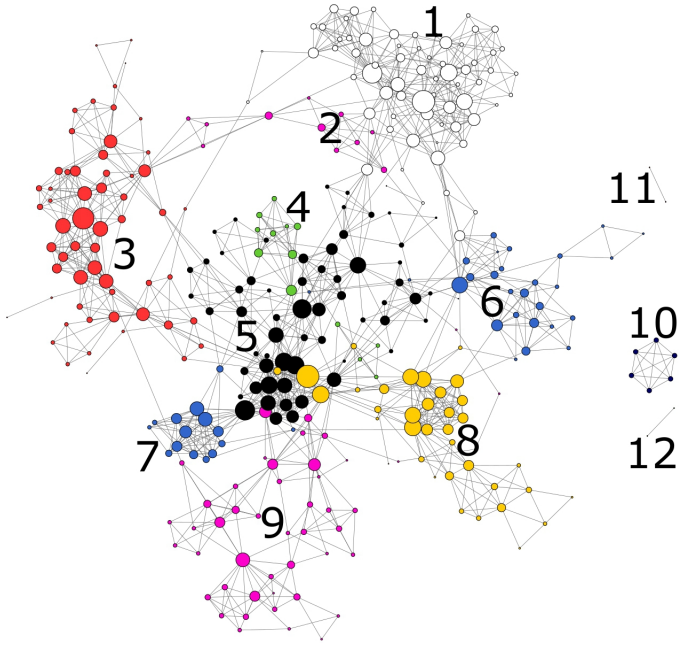

Figure 2: Communities of co-authoring graph

```
// Loading a graph file and creating a new undirected graph
var g = snap.newUGraph("biotechnology_2013.edg");
// Simplifying graph by removing all the nodes with degree <= 3
var g = snap.removeNodes(g, 3);
// computing communities using Clauset-Newman-Moore algorithm
var CmtyCNM = snap.communityDetection(g, "cnm");
// plotting the graph with colors of colors corresponding to communities
viz.drawGraph(g, "graphCNM.html", { "color": CmtyCNM });
```

Code fragment 3: Example graph analysis

The results of the analysis shows that the co-authoring graph has 12 communities labeled from 1 to 12 in Figure 2. Three smaller communities (10, 11 and 12) are completely disconnected, while the rest make a connected component. By supporting SNAP, QMiner enables rich graph analysis in an easy to use JavaScript interface, which can be especially useful when combined with the supported scalable machine learning methods.

5 Conclusion

This paper presents QMiner, an open source analytics platform for performing large scale data analysis. The system is build from five architectural modules based on five key design elements, including: connecting storage, indexing and analytics, scalability of implemented machine learning methods, multimodal data support, processing of streaming data, probabilistic joins between tables and fast as well as scalable prototyping.

The core of the platform is implemented in C++ and has a JavaScript layer on top of it which makes it more user friendly without sacrificing performance. As illustrated with the three representative examples of text classification, time series processing and graph analysis, complete workflows can be implemented with simple scripts.

Acknowledgments

The research leading to these results has received funding from several European Union Seventh Framework Programme (FP7/2007-2013) grants including Sophocles (ICT-317534-FET), X-LIKE (ICT-257790-STREP) and SYMPHONY Collaborative project (FP7-ICT-611875).

References

[1] Gregor Leban, Blaz Fortuna, Janez Brank, and Marko Grobelnik. Event registry: Learning about world events from news. *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, pages 107–110, 2014.

[2] Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.

[3] Donald E. Knuth. *Sorting and Searching*, volume 3. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, second edition, 1998.

[4] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[5] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53(2):217–288, May 2011.

[6] Aleksandra Rashkovska, Jost Novljan, Miha Smolnikar, Mihael Mohorcic, and Carolina Fortuna. Online short-term forecasting of photovoltaic energy production. *Proceedings of the Sixth IEEE Conference on Innovative Smart Grid Technologies*, pages 1–5, 2015.

[7] Aaron Clauset, M. E. J. Newman, and Christopher Moore. Finding community structure in very large networks. *Physical Review E.*, 70(6), 2004.