Plugins for PostgreSQL and Orange

D5.1 v1.0

WP5 – Visual Analytics: D5.1 Plugins for PostgreSQL and Orange

Dissemination Level: Confidential

Lead Editor: Janez Demsar

Date: 04/09/2013

Status: Final

Description from Description of Work:

T5.1. Integration between Orange and PostgreSQL: To be able to use Orange for the project, its existing infrastructure needs to be better integrated with PostgreSQL. We will provide the plugins for both Orange and PostgreSQL, a necessary scripting-level API for Orange, and the user interface to connect the existing in-memory data storage of Orange and PostgreSQL.

Contributors:

Janez Demsar, UL

Internal Reviewer(s):

BSC: Adrián Cristal, Nehir Sonmez
UNIMAN: Geoffrey Ndu

Version History

Version	Date	Authors	Sections Affected
0.1	20/08/2013	Janez Demšar, UL	Initial version
0.2	02/09/2013	Janez Demšar, UL	Changes in response to comments by review from UNIMAN
1.0	04/09/2013	Jo Dix, 2ndQ	changed version number before submission.

Table of Contents

1. Summary

This document describes the plugins for Orange that allow it to use the data stored in a database (e.g. Postgres) besides the in-memory data. This is needed for the AXLE project since storing the data in working memory is out of question for any reasonably big *big data*.

The document starts with the general description of the problem, followed by analysis of requirements, the new data storage schema in Orange and the implementation of the required classes for database access. We finish with a short description of the work that has not been done so far, but is expected to be needed in the future.

2. Analysis

The original plan for this activity was to provide plugins for connecting the PostgreSQL (or, in principle, any database) with the existing Orange storage engine (see http://orange.biolab.si/docs/latest/reference/rst/Orange.data/). Instead of storing the data in working memory, as currently in Orange, the data would be represented as an SQL query (or a data structure that is easier to manipulate but trivial to turn into an SQL query). Basic manipulation of the data, like filtering or features construction, would be implemented by changing the column selection or adding conditions to the WHERE clause in the SELECT statement. The plugins would then allow for accessing this data in the same way as the data stored in memory, thus allowing the existing algorithms in Orange to be reused without any modifications.

Upon rethinking this approach, we realized that it cannot work for big data. For instance, even the simplest of data analytic visualizations, a box plot, requires computing a few statistics (average, variance, minimal and maximal value...); these cannot be computed on the client side with only row-access to the data on a database with a few terabytes of data.

The plugins for integration between Orange and the database thus needed to go beyond purely “imitating” the in-memory data by wrapping the row access, redirecting it through an SQL query. The plugins also need to take care of all computation that requires processing large amounts of data, like computing averages, distributions and so forth. This was previously implemented within Orange itself. The corresponding methods need to be modified so that they leave the processing to the plugins (which execute it on the database server) when the data is stored in an external database (e.g. Postgres) and not in the working memory within Orange.

Parallel to this, we observed that Orange is becoming an outlier with regard to the common Python libraries due to its use of proprietary data structures. While library-specific data structures were the norm in Python when Orange was first conceived, numpy’s arrays have lately become the standard way of storing large data in Python. They are flexible enough to efficiently store and manipulate arrays of any data type; they allow for easy exchange of data between different modules and, as a consequence, there is a huge stack of modules available that work with numpy arrays. Many of them are packed into the scipy library, which also includes a powerful machine learning module scikit-learn. To keep Orange in sync with other Python libraries, we needed to reimplement the basic data structures, replacing the proprietary structures with numpy arrays.

The redesign of Orange is done as a part of other projects. The work done within AXLE helped us define a structure that allows to easily integrate classes for handling database-based big data. We have first investigated to which level we need separate functions for different data storages (in-memory vs. database, small vs. large) and then we implemented the corresponding functions for data stored in the database that are compatible with the functions for in-memory data in the new implementation of Orange.

As a consequence of the reimplementation of Orange, which is happening in parallel with AXLE, the work done within this deliverable was much larger than expected, making D5.2 much simpler as it required almost no additional work.

Required functionality

This deliverable treats Orange as a data mining library, ignoring its graphical aspects. Currently available data mining libraries of similar kind mostly provide functions that expect input as in-memory data, on which they can efficiently execute the needed operations. With the data stored on the database of such size that it cannot be transferred to the client, any operations on raw row data need to be executed on the server. To implement the planned plugins, we first needed to catalogue all such operations. We did so by analyzing the implementation of current functions in Orange - the results of our analysis are described below.

Data required by data mining methods can be roughly put into the following categories:

A. basic aggregates like mean, variance, median, minimal and maximal value,
B. distributions of discrete and continuous variables, values at percentiles,
C. contingency matrices,
D. covariance matrices,
E. filtering of rows based on various criteria, including random sampling,
F. selection of columns,
G. construction of variables from values of other variables,
H. matrices of distances (e.g. Euclidean) between all row pairs,
I. individual data rows.

Points A to D are typical examples of what cannot be done on client but can be efficiently done in the database. We therefore need to implement these operations within the data storage.

Points E to G can be implemented “lazily”, by modifying the SQL query describing the data; the query is not executed until needed, for instance to compute the data in points A to D.

Point H is difficult to implement efficiently in the common relational databases and, besides, result in a data matrix that is larger than the actual data. Any method that would require such a matrix needs to be reimplemented. An example of such a method is clustering: clustering on small data can use pre-computed matrices, while clustering of large data or data stored in databases requires specifically crafted algorithms. We thus cannot provide the data described at point H., nor do we foresee any need for it.

Point I. requires some caution with regard to how the data is retrieved and what it is used for. Row-based retrieval may be sequential (e.g. a method may be requiring rows of data, one by one, in some arbitrary or prescribed order) or random (i.e., the method requires a specific row). Second, the method that retrieves the data should not try to retrieve all data rows, for instance to find a minimum value of a certain attribute. We have implemented a few ways for accessing individual rows, which can be used in some cases but not in all. For instance, k-nearest neighbours classifier (k-NN) and discovery of association rules cannot use the aggregates such as described in points A. - D. or any similar general-type of data that can be retrieved from the server. Under this schema, they would both require access to random rows of data. However, for the efficient implementation, they would need to subsample the data (this approach would be particularly useful for k-NN) or to use algorithms implemented on the server side, such as several variations of the algorithm for discovery of association rules.

3. Implementation

This part of the report is oriented more technically. The code it refers to is available on https://github.com/biolab/orange3.

The new Orange data storage schema

While Orange is being developed outside AXLE, the above considerations influenced the design of its new data storage, so we briefly describe it here.

While the previous versions of Orange had a single class for storing data objects, `Orange.data.ExampleTable`, the new version defines an abstract class `Orange.data.Storage`. The abstract class contains no other code then definitions of optional virtual functions. Class `Orange.data.Table` provides flexible in-memory storage that uses numpy arrays of arbitrary types, including sparse matrices. Within AXLE, we developed another class `Orange.data.sql.table.SqlTable` which provides the same functionality as `Table`, except that the data is stored in the external database (e.g. Postgres).

Points A to D from the above list are handled as follows. Previously, Orange had, for instance, a class `Orange.statistics.distribution.Discrete`, with a constructor that was given a data set (an instance of `Orange.data.ExampleTable`) and a variable descriptor, and it computed the distribution of that variable by iterating through the data row by row. In the new Orange, such a class still exists and can compute the distribution from the data source in the same way as before. This is, however, inefficient, both for data stored in numpy as well as for data in the database. The new implementation of `Discrete` first tries calling the corresponding method of the data storage (`_compute_distributions`) that efficiently computes the required distribution. If this method is not implemented (both currently available storages, `Table` and `SqlTable` of course implement it), the inefficient row-by-row computation is used as a backup.

There are currently three slots, `_compute_basic_stats`, `_compute_distributions` and `_compute_contingencies`, which correspond to points A to C. The slot for point D will be defined when we first need it.

Point E, filters, are implemented in a similar way: the abstract storage class defines optional abstract methods for filtering. These are used by the classes for filtering, which also provide inefficient backups for any methods that are not implemented in the storage.

Currently defined filters match those in the old version of Orange: `_filter_is_defined`, `_filter_has_class`, `_filter_random`, `_filter_same_value` and `_filter_values`.

Points F to G are handled by domain descriptors. A data set in Orange has an associated domain descriptor that contains a list of variables, their types and other meta data, such as symbolic representation for values of discrete variables. Database storage uses the domain descriptors to define the columns in the SELECT statement. Therefore, all operations defined under F. and G. can be achieved by modifying the domain descriptor.

Row access, point I, both random and sequential, are implemented as operators defined in the classes derived from `Storage`.

Code

The code specific to SQL resides in modules `Orange.data.sql.table` with classes `SqlTable` and `SqlRowInstance`, and `Orange.data.sql.filter` with classes that extend filters in module `Orange.data.filter`.

Class `SqlTable` takes care of

- constructing variable descriptions based on column descriptions from the database;

- copying a table; copying is lazy as it requires just constructing a new class with the same attributes;
- providing random and sequential access to the rows in the database table by implementing special Python methods `__getitem__` and `__iter__`;
- implementing filters by copying the table and adding the appropriate terms to the SELECT clause;
- implementing the methods for computation of basic statistics and distributions by calling the backend.

`SqlRowInstance` extends `Orange.data.table.RowInstance` by handling meta attributes in the constructor.

Filters in `Orange.data.sql.filter` add a method `to_sql` which returns a condition for the WHERE clause that corresponds to the filter. The same effect could be achieved by monkey patching instead of deriving new classes; this is, however, not a common practice in Python.

Documentation and testing

All public functions and methods are documented using the standard Python docstrings in the rst format. Documentation is compiled to html using Sphynx.

The code is tested using a battery of unit tests (https://github.com/biolab/orange3/tree/master/Orange/tests/sql) with 82% coverage of the code.

Dependencies

Besides the other libraries used in Orange (numpy, scipy, bottleneck, scikit-learn required by Orange; nose and mock required for testing; Jinja2 and Sphinx required for building documentation), code developed within this project requires the library psycopg2.

License

The code of both classes and a few auxiliary classes is available on Github (https://github.com/biolab/orange3/blob/master/Orange/data/sql). All the code is released as a part of Orange 3.0 under BSD license.

4. Future Work

New features will be added as needed. The new version of Orange currently supports only a subset of functionality of the old Orange and we expect that during development of Orange, we will need to add new functions to the in-memory storage, which will also be reflected in its database counterpart.

As a consequence of restructuring the base classes in Orange, the current implementation required no server-side code. This *may* change when we add new functions, as described above, and *will* change when we move to the big-data territory. The necessary modifications will need to be programmed in C as a low-level database plugins, while on the client side the corresponding changes will probably be limited to the backend.