LEVERAGING DATA ANALYSIS FOR DOMAIN EXPERTS: AN EMBEDDABLE FRAMEWORK FOR BASIC DATA SCIENCE TASKS

Johannes-Y. Lohrer, Daniel Kaltenthaler and Peer Kröger
Institute for Informatics, Ludwig-Maximilians-Universität München, Germany

ABSTRACT

In this paper, we describe a framework for data analysis that can be embedded into a base application. Since it is important to analyze the data directly inside the application where the data is entered, a tool that allows the scientists to easily work with their data, supports and motivates the execution of further analysis of their data, which would lead to new and interesting findings is desirable. If the analysis process is too complicated, tedious or not apparent, this could restrict scientists, especially in non-IT-related disciplines, from analyzing the data in depth. To enable this solution we first describe the requirements for an analysis tool and explain the steps we took to meet those requirements. Then we describe the steps that are necessary to integrate the analysis into a base application. We also explain how the analysis framework can be extended with new specific components that allow the users to add exactly the features they need for their analysis.

KEYWORDS

Data Analysis, Embeddable Framework, Information Management, Knowledge Management, Modular

1. INTRODUCTION

In a database application for scientific data, the accessibility to analysis of existing data is as important as the ease of correct input. Collecting and analyzing data is the most important part of scientific work, but often scientists require a high degree of time and patience to learn, evaluate, and validate an external tool. This effort drains research resources and creates work. Scientists working in areas that are not related to IT technologies often do not have the motivation and the resources to get familiar with external applications. A build-in tool would provide the needed data analysis features relevant for these scientific areas.

Therefore, exporting data into a spreadsheet (like Microsoft Excel, LibreOffice Calc, etc.) or CSV file and importing this into a separate analysis tool, can in the worst hinder them completely from analyzing the data, since the process may be too complex, time consuming, or error-prone. Also generating the analysis directly in a spreadsheet is not always possible, since complex analyses might require additional functions that are not provided by a generic spreadsheet application, and cannot easily be added without programming skills. Clearly, the process of the data analysis should be easier for the researcher. The best case would be if the feature to analyze the data is already built into the application, where all data is stored and new entries can be added, because the scientist using the build-in application is already accustomed to this working environment.

But even if the analysis tools are built into the application, a crucial problem still remains: The scientists, who are carrying out the analysis, are often not responsible for creating the analysis tools, nor are they even involved in the process of creating them. Not all variations of analysis can be known beforehand since there are often tasks specifically dependent on the scientific work. So the domain experts must be able to create exactly the analyses they require.

To allow this, the application must provide not only predefined analysis methods, but also offer dynamic generation of analysis by chaining together different, simple and configurable modules. However, some scientific tasks are so special that these modules are not sufficient. Therefore, there must be a way to add own, specific modules as well.

In this paper, we provide a dynamic framework for data analysis that can be embedded and thus can be used in different applications. The framework must be extendible, easy to integrate into an existing application, and provide multiple operations that are necessary to analyze data. The framework must be flexible and easy to be used by scientists that are not willing to get familiar with external technologies, to support them in their work.

In summary, the main contribution is as follows: We list a set of requirements that should be addressed by an embeddable framework for scientific data analysis. Then we describe the data structure, the functionality of the modules, and the classification that describes the values in the database. We describe the integration of the framework to another base application and the definition of new, custom modules. Finally we introduce some of the most common modules as examples and use them to present an example of a workflow.

2. REQUIREMENTS

Allowing the users to generate their own analyses out of arbitrary data brings the challenge, that neither the input, nor the output is known. Also all steps in between are up to the users. Still, the users must be able to work in a responsive environment, which allows them to carry out the steps they want and need. But at the same time, not all operations should be allowed, or are possible in a given step of the analysis pipeline.

As a consequence, we have defined several requirements that have to be met to allow a dynamic generation of analyses.

- **Dynamic data structure:** We need a data structure, that allows data to be easily added, removed, merged, and accessed, since the data must be generated, transformed, searched in, and of course, also be displayed. This data structure must also be usable in every part of the analyses, independent of the previous steps.
- **Modular components**: Since the users will define the operational steps, the order in which specific tasks are run is not known beforehand. Therefore we need modular components, that can be linked together and define a "workflow". The components must be able to accept the data structure – no matter what components were used before, but not all inputs must be valid for the component. So it is possible that none of the inputs can be used by the component, but still the data structure itself must be accepted. To make the component more dynamic it also should provide settings like which input to use or the order to sort. These settings should, if applicable, dynamically change depending on the input.
- **Extensibility**: It must be possible for everyone, to extend the application with new components that offer new, possibly very specific, functions. Since there are many special analyses that cannot be put together with generic modules, it must be possible to include these in the list of available workers.
- **Embeddability**: Since the analysis shall be done with the data the users might just momentarily have entered, we require the analysis to run directly from the application, without first having to extract the data into spreadsheet or CSV files, or similar. This means that the analysis must be able to connect itself to another application and use the structure defined inside the program for its analysis. This might be a difficult requirement but without it using an analysis tool might not be so easy for the user.

A good overview of commonly used tools can be found at KDNuggets (Jones 2014) where some of the most known analysis tools are described (KNIME 2016; RapidMiner 2016; TableauPublic 2016). While these and many other different tools and applications for data analysis offer a variety of analysis methods and different options how to import the data, to the best of our knowledge there is no analysis tool available that can be integrated into an existing application. Therefore an in-depth comparison to these tools is outside the scope of this paper.

3. REALIZATION

We discussed the requirements for dynamic analyses. Now we take up these requirements and present our approach to meet them

.

3.1 Data Structure

A data structure is required that allows dynamic, flexible restructuring while still providing access to the data. Therefore the data structure for our framework consists of several elements.

- **Column Header:** This holds the important information about the specific field, such as the type and the display name. Since a field is basically a column in the database, we use the name of the field and the column. These are synonymous. The Column Header itself does not store any information about the value of the field, but serves more as reference information and metadata for the column.
- **Entry Value:** Is the smallest data type to hold the values. It holds a list of strings, which represent the values for a specific entry. This is necessary if a field has several options, e.g. different values for specific measurements. So every distinct value is one string, and all of them are saved inside the Entry Value.
- **Entry Map:** It maps the Column Header to the Entry Value inside a map. This map now represents a complete entry. It can easily be accessed because of the nature of the map. Therefore, writing and reading is no problem. Also editing values can easily be done.
- **Data List:** Is a list of all Entry Maps, which represents a complete data set. It also contains the list of all Column Headers that are in any of the Entry Maps. This serves as a utility method to allow components easy access to the list of all fields, without having to iterate over all entries. The list is generated by adding all unknown Column Headers whenever a new Entry Map is added to the Data List.

3.2 Worker

Each component, or "Worker" as we call it, can have multiple inputs and outputs of data. Some Workers do not necessarily have inputs, like Workers that retrieve data from the database, since they generate data without manipulating it. At the same time there can be Workers without outputs, like a Worker that allows the users to display the data as a diagram.

Properties: The Worker consists of inputs and outputs, settings, and the actual logic of the Worker. The latter either uses the input(s) or creates a new Data List, runs its logic according to the settings, and finally outputs the data or displays it. For this the Worker defines a list of Properties which can represent a setting. So for every setting type there must be an own property, e.g. text properties are common properties. The users can enter text (for example to describe a name) or combo properties, where the users can select one value from a list of values. Below, we describe the methods a Property has to implement:

- `getLabel()`: It returns the label to describe the property. This should make clear to the users which setting they can manipulate.
- `setValue()`: It is called by the GUI with the specific value. This is used to tell the Worker, that the value has changed and therefore must update its data structure and possibly additional properties.
- `addPropertyListener()`: All elements that are dependent to this property, can register themselves as a listener, which would be notified if this property has changed. The method is called when either new options are available for selection, or the value of the property itself was set.
- `onNewOptionAvailable()`: This is called if new options for this property are available. This causes all property listeners to be notified, so that the GUI can now display the updated values.
- `onSelectionChanged()`: This is almost identical to the method `onNewOptionsAvailable()`, but instead it is called when the value of the property is changed, e.g. after the `setValue()` method was called.

The Worker also defines the number of inputs and outputs. The users then connect different Workers, which basically is a directed graph or multigraph, with the nodes representing the single Workers, and the edges representing the connections between the Workers. The number of inputs can vary as some Workers require no input, some require an exact amount, and some can work with an arbitrary number of inputs. If the Worker provides an output this can be used to pass the data on to other Workers.

Data handling: Each Worker fulfills a predefined task, like retrieving or merging data. However, the output of the Worker is only defined after the input(s) and the Worker settings are set. Therefore it is not necessary to instantly generate the output, since the input(s) may change if a setting in a previous Worker has changed. Equally, it is important to establish at least the Column Headers of the data. This makes it possible to update the settings of the successive connected Workers. This is the reason why the output is separated in two parts:

- **Output Scheme:** This is the list of all Column Headers that will be returned by the Worker. This list is instantaneously generated as soon as an input or setting is changed. The successively connected Workers are immediately notified with an event, that the Output Scheme of the Worker was changed. This enables them to check if their Output Scheme has also changed. This list has the same value that the list of Column Headers inside the Data List should have in the real output. Therefore, Workers only need this list to define what settings they provide and what their Output Scheme is.
- **Output Data:** This is the Data List with the "real" output containing the data. This is only generated if necessary, since the composition of the data could take some time. A complete recalculation is not required if only the Output Scheme is important. The real data must be generated if a diagram representing the data has to be displayed or the values should be listed. This is done recursively with each Worker from the end requesting the Output Data of the Workers that are connected to its input. This means the first Worker or Workers have to be able to generate data without any input.

This separation into Output Scheme and Output Data allows a fast application of updated settings and rearranged inputs or outputs while still ensuring that the correct type is used.

Interaction with the graphical user interface: The Worker itself is only responsible for the logic. But since the users need to be able to easily arrange, connect, and configure the Workers there has to be a graphical representation. The graphical user interface is notified with events of changes in the properties. At the same time it uses the properties to notify back to the Worker if any value of the settings was changed, such as entering a text or selecting a new value. This allows the graphical user interface to be created independently of the logic, and therefore is not limited to a specific format. The exact design of the graphical user interface however is not part of this paper. Here we want to just describe the technical connection to the graphical user interface.

4. INTEGRATION

To reuse the analysis tool in different applications, it is important that the integration requires as few changes to the base application as possible. But since the analysis tool cannot know how the data is stored, or how the connection to the data is realized, the base application must implement some wrapper methods.

The analysis tool itself is composed in a `JPanel` or `JFXScene`. For this, the base application can either create a new `AnalysisSwing` or `AnalysisFX` instance, which both require an `IController`. The `IController` is the interface that serves as the combiner between the base application and the analysis. It provides the following methods that the base application has to implement:

- `getTableNames()`: This returns the list of different names of tables, that can be selected for the analysis. This should only return tables in which the users have entered data.
- `getColumnsForTable()`: This returns the columns for the given table, that can be included in the analysis. This should return only columns that are important for the users, but not columns that contain information that is irrelevant to the user. The expected return value is a list of Column Headers so therefore all columns have to be transformed into this format. This is important, because all future analysis options are based upon the information stored inside these Column Headers.
- `getKeysForTable()`: This returns the key columns for the given table. This can be used for example in the Combiner to provide default mappings. Also this method requires the returned values to be a list of Column Headers.
- `getProjects()`: The database can be structured in different "projects", which is a logical separation of different data sets. To also allow this separation inside the analysis framework, a list of a unique identifiers can be returned. This could be just a name or an integer, but can also be a more complex data structure, in which the `toString()` method returns the name of the project, to be able to display it to the users. If the separation into different projects is not desired or supported, this method can return `null`.
- `getDataForColumns()`: This returns the Data List for the given columns in the given table for the optional project. As stated, the analysis tool has no knowledge about the structure of the database, so the base application must generate the Data List. The list of projects is only required if `getProjects()` returns a value and can therefore be ignored if it is not applicable. The values that are returned would

most likely not the immediate values as they are stored in the database, but they are already translated into human readable form, e.g. by translating IDs into the appropriate values.

If necessary, all Workers can then use the `IController` to retrieve the data they require. This is the only connection data-wise needed for the analysis, as all further analysis is built onto the retrieved data. The base application therefore need not know about any internals of the analyses or vice versa. Since the analysis is done inside one panel, it can easily be included into the base application, which can decide where and when the analysis shall be displayed.

5. DEFINITION OF CUSTOM WORKERS

While creating an analysis tool, not all use cases can be known, since the area in which the analysis is used is not always known in advance. Therefore it is very important to be able to add new Workers, which exactly fulfill the requirements of the individual analysis to be applied. For this we defined a very simple API to allow new Workers to be easily implemented. To add a new Worker, the interface `IWorker` has to be implemented. It defines the basic functions that are required for the integration in the workflow.

- `getTitle()`: This method expects the name of the Worker to be returned as a string value. This name is displayed for the users inside the graphical user interface. It should be a short but meaningful name, which allows the users to instantly comprehend the function of the specific Worker.
- `getProperties()`: This returns a list of Properties which defines the settings of the Worker. With these settings, specific aspects for the Worker can be set. This list of Properties also includes Properties for the input and output, which define the connection to other Workers. The Worker is notified through the Properties if the input or a setting has been changed. Since the Properties are abstract classes, all methods of these classes must be implemented when creating a new Worker. This includes the displayed name, the logic for setting and retrieving data, and additionally, which Property values are displayed in the graphical user interface.
- `getOutputData()`: This returns the Output Data of the Worker after it has completed its job. If neither the input nor the Worker settings change, this will return the generated values from the previous call, otherwise the process has to be run again. Therefore, in this case the method has to call the `startWorking()` method and return the generated values after it has completed.
- `getOutputScheme()`: This returns the Output Scheme of the Worker, that would be returned in the Output Data of the specific Worker. The Output Scheme should be calculated with respect to the Output Scheme of the connected previous Workers, if any, and the settings of this Worker. If the input and settings of the Worker did not change, this method does not have to recreate the Output Scheme again, but can simply return the previously generated data. Additionally, this method does run the complete calculation method, but it calculates the Output Scheme.
- `onInputChanged()`: This method is called by the Properties to notify the Worker that something has changed and the Output Scheme and Output Data have to be recalculated if requested. This method, however, does not start the update process itself.
- `startWorking()`: In this method the actual logic is carried out. The input is collected by iterating over all input Properties and getting the Output Data of the connected Workers. This triggers them to generate their results themselves if needed by carrying out the same logic recursively. Then the result of the Worker is calculated with regards to the settings defined in the Properties. This method is usually only called inside the `getOutputData()` method.
- `setController()`: This method is used to set the `IController`, which is used for interaction with the base application. This is required to be an own method and not part of the constructor, since all Workers are created with reflection, therefore the constructor must be the default constructor. With the `IController` the Worker is able to query the base application for information that is possibly required for the Worker with regard to carry out specific operations.

After the Worker has been created, there are two options to register it with the application.

- **Direct registration:** The API provides a registry class, which allows Workers to be registered for inclusion in the analysis. In addition to the Workers available by default, the registered Workers can then be selected by the users. This method naturally requires access at runtime and therefore can usually only be executed from the base application.
- **Indirect registration:** Externally created Workers can be added to the analysis. For this all .jar files inside a specified folder (default "./analyses/workers") are analyzed if they contain classes that implement `IWorker`, and if so they are added to the list of available Workers for analyses.

Since the Workers use Properties to tell the graphical user interface what to display, it is important to also add new Properties if the available Properties are insufficient. This consists of two parts: The definition of the property itself and also of the definition of the graphical representation of the Property.

To define a new property, it has to extend the `Property` class. It can then define additional methods that are used by the representation of the graphical user interface. Then it can be registered together with the graphical representation in the registry.

6. EXAMPLES

As mentioned before there are types of Workers that fulfill different tasks. Here we want to give an example of the most common Workers that are sufficient for a basic analysis. There are far more different Worker types possible, but the focus here is to give an overview over the possibilities the Workers provide.

- **Retriever:** The most basic worker that will be used in every analysis. The Retriever, as the name suggests, retrieves data from the database. The users can define the fields and projects they want to include in their analyses, with the fields and projects being a structure for the data in the application the analysis tool is developed for. This would normally be the fields the users either want to filter, display, or further analyze. The Retriever is the only Worker that needs a connection to the database to get any data. It uses the `IController` to get the required data.
- **Filter:** The Filter can be used to filter out entries that are not required in the analyses. For example, an analysis about specific animals may only require the entries containing data of these animals. So the users can filter out all other animals using the Filter. The users have several options: They can specify the fields for which the data shall be filtered. The list of available fields is defined by the Output Scheme of the previous Worker. Depending on the column header, they can specify whether the value of the field contains or equals a specific text. For dates or numbers it is possible to check, if the value is smaller, greater or equal than a specific user input. It is also supported to define a combination of different fields that can be filtered at the same time.
- **Combiner:** This allows different data sets to be combined. The users can define the fields which should be considered when combining the data sets. The list of available fields is defined by the Output Scheme of the previous Workers. For example, the Combiner can be used if a user has already created two different analyses with different data sets, and now wants to combine the data for a third analysis. The Combiner searches for entries in the given list of datasets that are equal on the fields the user entered. The result of the combination can be compared to the SQL join. There are two possibilities, that either only entries having a match, or also entries that have no match are combined. In this case the other columns are filled with empty values. Then these entries are combined into a new entry.
- **Sorter:** This sorts entries according to the comparator set in the Column Header. It only has one input, but allows more than one column to be set as sorting column. This allows different priorities, in case of the values of the column with a higher priority are equal. This can be useful for example if a specific order of entries is required in a diagram.
- **Diagram:** The Diagram is the umbrella term for Workers that display the result in a graphical representation. They can be used in different and complex ways. As an example, we describe a two-dimensional, axis-oriented bar chart. The users can select the field which values shall be used for the x-axis. The list of available fields is defined by the Output Scheme of the previous Worker. The different values of this field are then listed as values of the x-axis. The number of entries for the given value are displayed on the y-axis. An example of a Diagram can be seen in the workflow example in Figure 1.

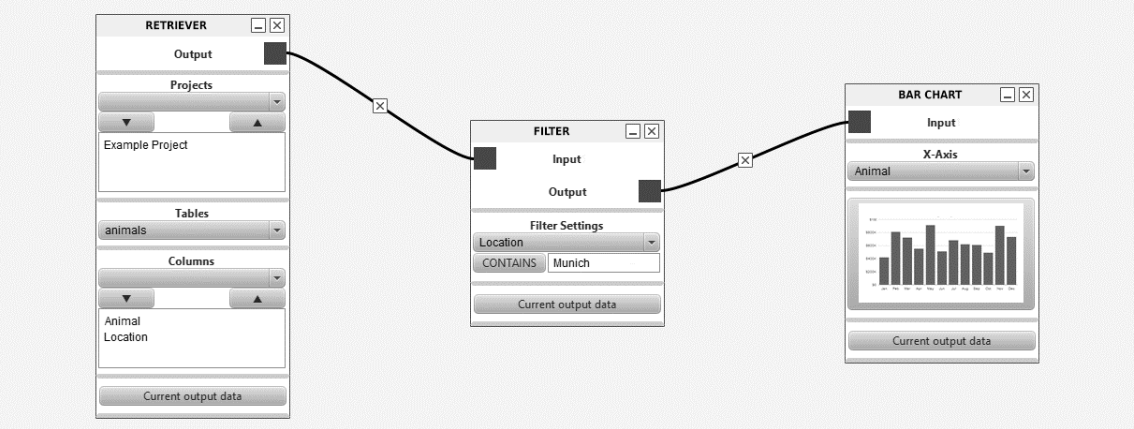

Figure 1. Composition of the analysis for animal distribution in Munich
The analysis framework was embedded to the zooarchaeological database *OssoBook* (2016)

7. WORKFLOW

We discussed the structure required for the analysis, and also gave examples of the most basic Workers. In the next step, we want to put the pieces together and describe how to build a typical analysis. As an example, we want to calculate a distribution of animals in Munich, and generate a graphical representation of the data as a bar chart. The composition of the data in the Analysis Framework is shown in Figure 1.

We take a sample table with the columns "*ID*", "*Animal*", and "*Location*", as shown in Table 1. Of course, in real scientific databases, a table like that would contain several more columns and datasets. Because of reasons of clearness we reduce the example data to a minimum.

Table 1. Sample data for Animals

ID	Animal	Location	...
1	Dog	Munich	…
2	Mouse	Los Angeles	…
3	Dog	Guarujá	…
4	Kangaroo	Melbourne	…
…	…	…	…

The analyzing process can be divided into three different steps:

- **Planning and collection of data:**
  Gather all required data from different sources and combine it, so it can be used for further processing.
  We first identify the fields we need in our analysis. These are: "*Animal*" and "*Location*". The field "*ID*" is not important for our goal and can be disregarded. The first Worker we use is a Retriever to get the necessary data. We configure it to get the fields "*Animal*" and "*Location*" only.
- **Processing data:**
  The data is processed to remove unimportant data or structure the data, e.g. by filtering or sorting it.
  Next we only want to filter all animals from Munich. Therefore we add a Filter. The input of the Filter is connected to the output of the Retriever, which makes the Fields "*Animal*" and "*Location*" available as options for the Filter. There, we select "*Location*" and define only to allow datasets in which the "*Location*" value equals the term "Munich".
- **Generating result:**
  The processed data is used to display a result like a diagram or any other visualization.
  The last Worker is a Diagram. It takes the filtered data from the Filter and displays a graphical representation of it. In our case, we choose the bar chart. We can now select the column, which values we want to use as the x-axis. We select the "*Animal*" column and the Worker generates the chart as seen in Figure 2.

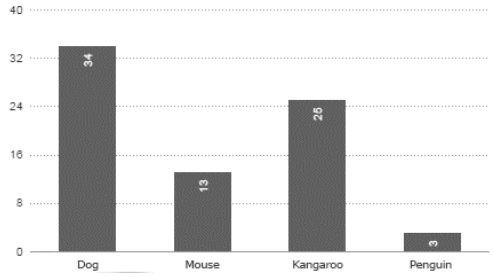

Figure 2. Possible result of the analysis in Figure 1

8. DISCUSSION

In this paper, we described a framework for generating analyses that can be embedded into a base application. The framework aims to be intuitively used by scientists without any IT background, inside their accustomed working environment. We identified the requirements for the framework, which must be fulfilled to allow working with it. Then we discussed the applicability of the requirements with some existing analysis tools. We described the realization of our tool, to be able to meet the requirements, and discussed how the framework can be integrated into a base application and extended to the demands of the user. Finally, we presented some of the basic Workers and used them in an example to show the analysis workflow.

While the framework can already be used for generating interesting analyses, there are still some issues that could be addressed in the future, to make working with the framework go more smoothly.

While the integration into the base application is straight forward and can easily done, it still requires both access to the source code and programming skills. Therefore, the typical users cannot do the integration themselves. This means that the developer of the base application has to integrate the analysis tool to provide the functionality to the users. For these cases, the framework could be extended to run as a standalone application, which can be connected to the database directly. This would require additional settings which handle the connection to the database itself. At the same time, a stand-alone tool could benefit from the same level of flexibility and extendibility that the framework offers while being integrated into a base application.

There are already a wide range of possible analyses, but still many additional Workers have to be created. These Workers should be part of the framework itself, since they can be used for analyses in different areas. Possible Workers include clustering algorithms, different diagram types, etc.

The logic handling the datasets is currently focused on processing speed. For this `DataLists` are often just copied and remain in the primary memory. For small datasets this is a fast and efficient way, but for complex databases with thousands or millions of entries, this method can quickly reach the computer's capacity limit. To meet this problem several solutions would be possible. The generated result could only be kept in memory during one operation, until succeeding Workers have used the Output Data. This would slow down the analysis process, since even for a small adjustment, all Output Data would have to be recalculated.

Additionally, the generated Output Data could be saved in a temporary file, so that main memory is freed because the Output Data is not used for calculation. Again, this would slow down the calculation process, since disk operations are relatively slow.

REFERENCES

Jones, A. (2014). *Top 10 Data Analysis Tools for Business*, [online] http://www.kdnuggets.com/2014/06/top-10-data-analysis-tools-business.html [29.09.2016].

Kaltenthaler, D., Lohrer, J., Kröger, P., van der Meijden, C., Granado, E., Lamprecht, J., Nücke, F., Obermaier, H., Stopp, B., Baly, I., Callou, C., Gourichon, L., Pöllath, N., Peters, J., and Schibler, J. (2016). *OssoBook v5.4.4*. Munich, Basel. http://xbook.vetmed.uni-muenchen.de/.

KNIME Analytics Platform, [online] http://www.knime.org/knime [29.09.2016].

RapidMiner, [online] http://www.rapidminer.com [29.09.2016].

Tableau Public, [online] http://public.tableau.com [29.09.2016].