Towards Exploratory Data Analysis for Pharo

Oleksandr Zaytsev
Ivan Franko National University of Lviv
Faculty of Applied Mathematics and Informatics, Ukraine
olk.zaytsev@gmail.com

Nick Papoulias
UMMISCO IRD France Nord, Bondy
Sorbonne Universités UPMC, Univ. Paris 06, France
npapoylias@gmail.com

Serge Stinckwich
UMMISCO IRD France Nord, Bondy
Sorbonne Universités UPMC, Univ. Paris 06, France
Université de Caen Normandie, Caen
serge.stinckwich@ird.fr

Abstract

Data analysis and visualizations techniques (such as split-apply-combine) make extensive use of associative tabular data-structures that are cumbersome to use with common aggregation APIs (for arrays, lists or dictionaries). In these cases a fluent API for querying associative tabular data (like the ones provided by Pandas, Mathematica or LINQ) is more appropriate for interactive exploration environments. In Smalltalk despite the fact that many important analysis tools are already present (for *e.g.,* in the PolyMath library), we are still missing this essential part of the data science toolkit. These specialized data structures for tabular data sets can provide us with a simple and powerful API for summarizing, cleaning, and manipulating a wealth of data-sources that are currently cumbersome to use. In this paper we introduce the `DataFrame` and `DataSeries` collections - that are specifically designed for working with structured data. We demonstrate how these tools can be used for descriptive statistics and exploratory data analysis - the critical first step of data analysis which allows us to get the summary of a data set, detect mistakes, determine the relations, and select the appropriate model for further confirmatory analysis. We then detail the implementation trade-offs that we are currently facing in our implementation for Pharo and discuss future perspectives.

CCS Concepts • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Keywords Tabular Data-structures, Fluent APIs, Exploratory Data Analysis, Live Environments

ACM Reference format:
Oleksandr Zaytsev, Nick Papoulias, and Serge Stinckwich. 2017. Towards Exploratory Data Analysis for Pharo. In *Proceedings of International Workshop on Smalltalk Technologies, Maribor, Slovenia 4-8 September 2017, ESUG 2017 (IWST'17),* 6 pages.
https://doi.org/10.475/123_4

IWST'17, ESUG 2017, Maribor, Slovenia 4-8 September 2017

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 Introduction

The simplicity and power of Smalltalk combined with the live environment of Pharo creates a productive combination for data analysis. Provided the proper tools and open source libraries for machine learning, statistics, and optimization, Pharo can become both a powerful tool for professional data analysts, and a simple environment for everyone who wants to experiment with a simple data set. Many important tools and algorithms are already implemented in libraries such as PolyMath[1], but we are still missing essential data-structures and a fluent API for advanced data-analysis techniques. To overcome this problem we introduce in this paper the `DataFrame` and `DataSeries` collections for working with structured data, and through several examples demonstrate how these tools can be used for descriptive statistics and exploratory data analysis allowing us to get the summaries of data sets, detect mistakes, determine relations, and select the appropriate models for further analysis.

Our work is motivated by popular data-analysis techniques (such as split-apply-combine [12] and collection pipelines [1]) as well as from dedicated data-structure and APIs for data-analysis environments for Python [6], R [11], Mathematica [13] and the .Net platform (through the LINQ embedded query language [7]). Nevertheless our primary inspiration is drawn from Smalltalk itself and can be traced back to the extensive coverage of the Smalltalk-80 book on Collections [5] (with over four chapters dedicated to the usage and design of the Collection hierarchy). As well as from more recent work

[1] https://github.com/PolyMathOrg/PolyMath

from our community that refined the implementation of collections using traits [3, 4, 9] and promises [2].

To facilitate our readers, we should note here that all figures in this paper are DataFrame visualizations that are created with Roassal2[2] and can be reproduced by the steps described in Section 3. The rest of this paper is structured as follows: Section 2 provides a brief introduction into explanatory data analysis, answering the following question: *What is EDA good for and how to do it right?* It also provides basic knowledge about statistics and data analysis, such as: statistical variables, types of variables etc. Then Section 3 details the new DataSeries and DataFrame Collections for structured data. Section 4 gives a step-by-step example of how to perform EDA on the well-known Iris data set using the new collections and API. Finally Section 5 concludes the paper and discusses implementation trade-offs as well as future perspectives.

2 Exploratory Data Analysis

Exploratory data analysis (EDA) is an approach for analyzing data sets to summarize their main characteristics. It allows us to make some sense of the data by visualizing it and exploring its statistical properties. According to Howard J. Seltman, any method for looking at data that does not include formal statistical modeling and inference falls under the term exploratory data analysis [10]. It helps us to select the model that we will be fitting to the data during the following steps of confirmatory data analysis.

EDA can be particularly useful for uncovering the underlying structure of a dataset, detecting outliers and anomalies, determining relationships among the explanatory variables, assessing the direction and rough size of relationships between explanatory and outcome variables, and selecting appropriate models for further analysis[8].

The techniques of exploratory data analysis can be classified into four categories: univariate non-graphical (Sub-section 4.1), univariate graphical (Sub-section 4.2), multivariate non-graphical (Sub-section 4.3) and multivariate graphical (Sub-section 4.4).

3 DataSeries and DataFrame

DataFrame and DataSeries are the high-level data structures with which we intend to make data analysis in Pharo fast and easy.

They can be loaded into a Pharo image with the following Metacello script:

```
Metacello new
    baseline: 'DataFrame';
    repository: 'github://PolyMathOrg/DataFrame';
    load.
```

[2]http://agilevisualization.com/

A DataSeries can be seen as an Ordered Collection that combines the properties of an Array and a Dictionary, while extending the functionality of both. Every DataSeries has a name and contains an array of data mapped to a corresponding array of keys (that are used as index values).

The easiest way of to create a series is by converting an array:

```
series := #(a b c) asDataSeries.
```

The keys will be automatically set to the numeric sequence of the array indexes, which can be described as an interval (1 to: n), where n is the size of array. The name of the series at this point will remain empty. Both the name and the keys of a DataSeries can be changed later, as follows:

```
series name: 'letters'.
series keys: #(r1 r2 r3).
```

A DataFrame is a tabular data structure that can be seen as an ordered collection of columns. It works like a spreadsheet or a relational database with one row per subject and one column for each subject identifier, outcome variable, explanatory variable etc. A DataFrame has both row and column indices which can be changed if needed. The important feature of a DataFrame is that whenever we ask for a specific row or column, it responds with a DataSeries object that preserves the same indexing.

A simple DataFrame can be created from an array of rows or columns.

```
df:=DataFrame
      rows: #((John 25 true)(Jane 21 false)).
df:=DataFrame
      columns: #((John Jane)(25 21)(true false)).
```

Those two lines produce exactly the same DataFrame. Once again, the names (key values) of both rows and columns were not explicitly specified, so by default they will set to numerical indices: #(1 2) for rows and #(1 2 3) for columns. These names can be changed later:

```
df columnNames: #(Name Age IsMarried).
```

3.1 Discussion: DataFrame Internals

Before continuing with a detailed example on how to use these classes for data exploration, we would like to share our thoughts on their implementation.

Our goal for these classes is to fully encapsulate their internal structure so that it is both easy to experiment with optimized representations, but also allow for a polymorphic fluent API that can be implemented separately without breaking existing code.

This is why we are evolving the API and the implementation according to concrete needs we are facing

while analyzing specific data-sets (like the Iris or Housing data-sets that we will shortly discuss). We are currently in the fourth iteration of the model, since there are many trade-offs to consider, between for *e.g.,* using inheritance, instance composition and/or trait composition for the implementation. Even inheriting directly for *e.g.,* from Array, Dictionary or OrdreredDictionary comes with its own set of trade-offs. In the future we might consider reusing the internal structure of the pandas DataFrame in which the data is stored as one or more two-dimensional blocks rather than a collection of one-dimensional arrays [6].

Finally functionalities such as the head/tail protocol or the protocol allowing us to change the keys of a data structure are all currently implemented as traits. In future iterations, we expect to extend the use of traits to include the data visualization and aggregation protocols, that should be shared among the DataSeries and DataFrame classes.

4 Exploring Iris Dataset

In order to exemplify a real use-case of our library, we will start by loading the Iris dataset[3] from a CSV file.

```
data := DataFrame fromCsv: '/path/to/iris.csv'.
```

DataFrame comes with a built-in collection of datasets that are widely used as examples for data analysis and machine learning problems. Iris is among them, so here is an alternative way of loading it

```
data := DataFrame loadIris.
```

To make sure that the data was loaded and to take a quick look on it, we can print its head (first 5 rows) or tail (last 5 rows). It is also possible to specify the number of rows that must be printed. As it was mentioned in 3.1, the same messages are supported by objects of the DataSeries class. This means that we can also look at a head or tail of a specific column. In the following example we ask for the first 5 and the last 3 rows of a DataFrame. Then we ask for the first and the last 5 elements of a `sepal_length` column (which is a DataSeries).

```
data head.
data tail: 3.
(data column: #sepal_length) head.
(data column: #sepal_length) tail.
```

The first two messages were sent to a DataFrame. It will respond with another DataFrame object containing the requested rows. Here is the example output of the `data head` message.

```
1    (5.1 3.5 1.4 0.2 #setosa)
2    (4.9 3 1.4 0.2 #setosa)
3    (4.7 3.2 1.3 0.2 #setosa)
4    (4.6 3.1 1.5 0.2 #setosa)
5    (5 3.6 1.4 0.2 #setosa)
```

When asked for a column named `sepal_length`, DataFrame responds with a DataSeries object (how this object is constructed depends on the implementation and should not concern the user). Therefore, the second two messages are sent to a DataSeries which responds with another DataSeries containing all the requested elements. The output of `(data column: #sepal_length) head` should look like this:

```
sepal_length
1    5.1
2    4.9
3    4.7
4    4.6
5    5
```

First line holds the name of a series. The numbers in the following five lines are the key-value pairs. It is important that whenever a row or column is extracted from a DataFrame, it remembers all the key values (they are also called indices). So, for example, if we ask a DataFrame for its first row

```
data row: 1.
```

The DataSeries received as a responce will store the column name of each value in that row as a corresponding key. And the key of the row (1 is a row name or a key of the first row in our DataFrame, not the numeric index like in Array) will be stored as the name of a DataSeries.

```
1
sepal_length   5.1
sepal_width    4.9
petal_length   4.7
petal_width    4.6
species        5
```

4.1 Univariate non-graphical EDA

DataFrame can be viewed as a collection of statistical variables (columns). As demonstrated in 4, we can ask DataFrame for a single column by its name. Since the rows and columns of a DataFrame are ordered, we can also access a column by its number. The following two lines return the same `sepal_width` column:

```
data column: #sepal_width.
data columnAt: 2.
```

The best univariate non-graphical EDA technique for categorical data is a simple tabulation of frequencies for each category[10]. Let's look at a `species` column. As we can see by printing its unique values, `species` is a categorical variable with 3 categories: setosa, versicolor, and virginica

```
(data column: #species) unique.
```

[3] https://en.wikipedia.org/wiki/Iris_flower_data_set

Now we can tabulate this variable:

```
(data column: #species) frequencyTable.
```

The response will be the following DataFrame. The first column (row names) holds the unique categories, the second column - the number of occurences of each category, and the third one shows the fraction of data that belongs to each category. We can see that the sample data is evenly distributed among three categories.

```
setosa      50  (1/3)
versicolor  50  (1/3)
virginica   50  (1/3)
```

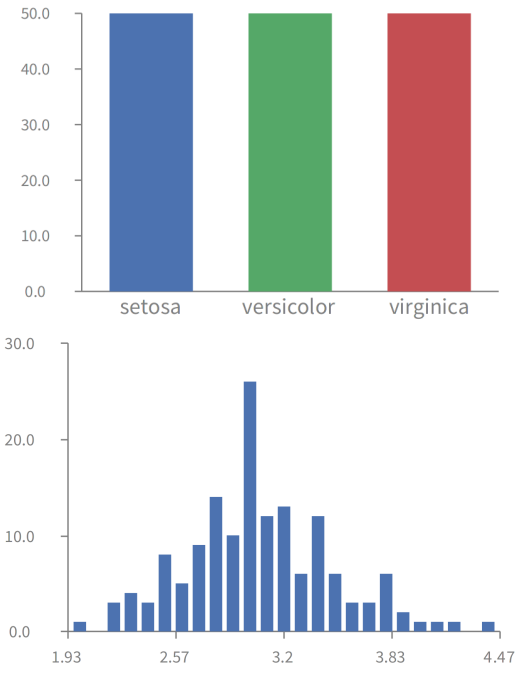
If the variable is quantitative, we can summarize it with the following statistics: `min, max, range, average, median, mode, stdev, variance`

```
(data column: #sepal_length) average.
(data column: #petal_width) stdev.
```

4.2 Univariate graphical EDA

The only graphical technique that can be used for a categorical variable is histogram - a barplot where the hight of each bar represents the count of cases for a range of values. Histogram is a graphical equivalent of `frequencyTable`, described in 4.1. If applied to a categorical variable, the height of each bar will be defined by the amount of data that belongs to a corresponding category.

```
(data column: #species) histogram.
(data column: #sepal_width) histogram.
```

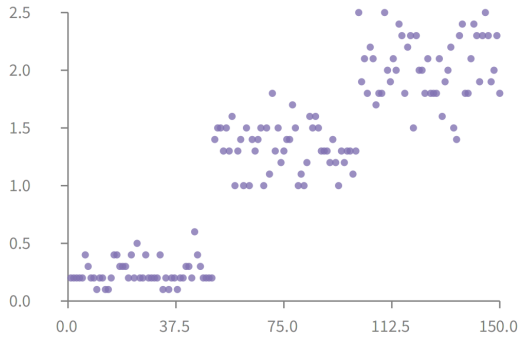


The important property of a histogram is the bin width - the number of cases that are summarized by a single bar. According to statisticians, like [10], choosing a bin width is an iterative process. Usually it is chosen between 5 and 30, depending on the amount of data and the shape of distribution, which can be seen on a histogram.

```
(data column: #sepal_width) histogram: 5.
```

Quantitative variables can be visualized in many different ways: `plot, scatterplot, boxplot, histogram`. The following example demonstrates how we can ask a `petal_width` column for a scatterplot of itself.

```
(data column: #petal_width) scatterplot.
```



4.3 Multivariate non-graphical EDA

Multivariate non-graphical EDA shows the relationship between two variables in form of either cross-tabulation (categorical data) or statistics (quantitative data).

To get multiple columns from a DataFrame we can give it an array of column names or numbers, or ask for all the columns in a given range, specified by two numbers. The response to any of these messages will also be a DataFrame, and the order of columns will be the same as requested.

```
data columns: #(petal_length sepal_width sepal_length).
data columns: #(3 2 1).
data columnsFrom: 3 to: 1.
```

When asked for a value of certain statistical measure, DataFrame will answer a DataSeries of the same size as one of its rows, with keys equal to the column names and the values of that stitistics applied to the corresponding columns. For example, we can ask the Iris DataFrame for its average

```
data average.
```

The response will be a DataSeries storing with average values of every quantitative column of a DataFrame and `Float nan` for all the categorical ones.

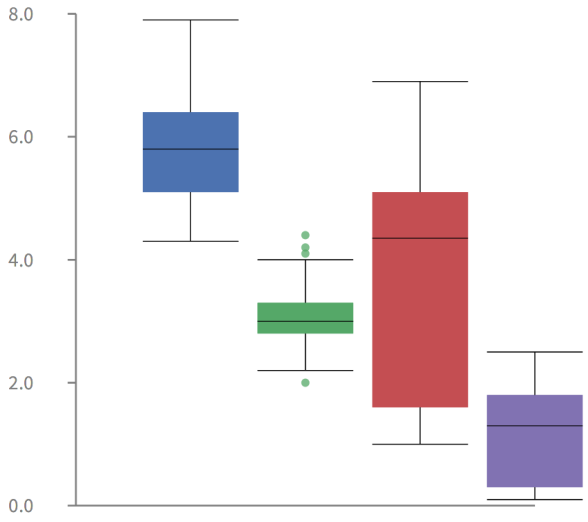
```
sepal_length   5.843
sepal_width    3.054
petal_length   3.759
petal_width    1.199
species        Float nan
```

4.4 Multivariate graphical EDA

According to [10], the most commonly used technique of graphical EDA is a grouped barplot with each group representing one level of one of the variables and each bar within a group representing the levels of the other variable. Another useful visualization is a side-by-side boxplot. It is considered the best graphical EDA technique for examining the relationship between a categorical variable and a quantitative variable. These two visualizations are not present in the current release, but are to be integrated in a future version of DataFrame.

Let's ask a DataFrame to show us the barplots of all the quantitative columns, aligned on the same axis. This can be useful for comparing variables among themselves.

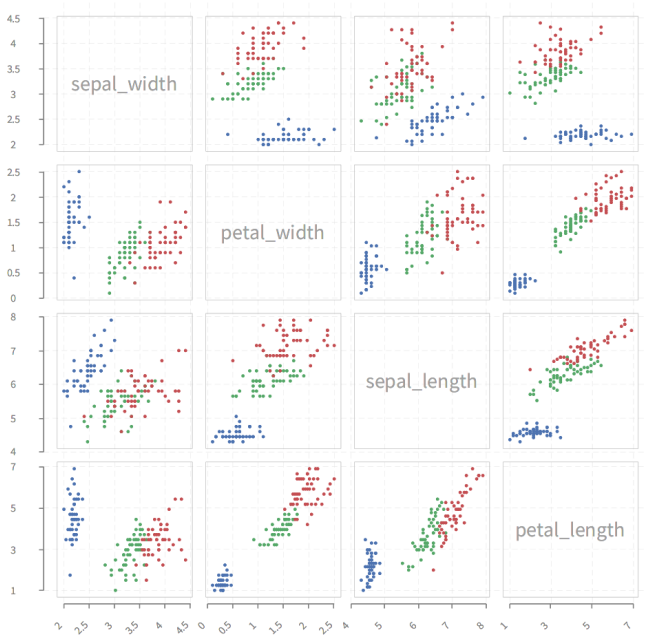
```
data boxplot.
```



We can also ask for a matrix that contains all the pairwise scatterplots of the variables on a single page in a matrix format. Scatterplot matrix is a complex visualization which allows us to roughly determine if we have a linear correlation between multiple variables.

```
data scatterplotMatrixOn: #species.
```

Quantitative variables are plotted against each other, and the values of a given categorical variable are mapped with colors. The column names are written in a diagonal line from top left to bottom right.



5 Conclusion & Future work

We have introduced the `DataFrame` and `DataSeries` collections in the Pharo Collection hierarchy. These classes are specifically designed for working with structured data. We have demonstrated how these classes can be used for descriptive statistics and exploratory data analysis, such as getting the summary of a data set, detect mistakes, determine relations, and select appropriate models for further investigation. We detailed the implementation trade-offs that we are currently facing in our implementation for Pharo and exemplified our contribution through an *Exploratory Data Analysis* session for the Iris Data-set, that included both univariate and multivariate (graphical and non-graphical) analyses.

In terms of future work we intend to evolve our API and implementation by investigating more complicated scenarios and data-sets, extend the usage of traits in our design and finally add support for more advanced protocols for data wrangling, aggregation and grouping.

References

[1] Collection pipeline. https://martinfowler.com/articles/collection-pipeline/, 2015.

[2] Juan Pablo Sandoval Alcocer, Marcus Denker, Alexandre Bergel, and Yasett Acurana. Dynamically composing collection operations through collection promises. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 8:1–8:5, New York, NY, USA, 2016. ACM.

[3] Andrew P Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the smalltalk collection classes. In *ACM SIGPLAN Notices*, volume 38, pages 47–64. ACM, 2003.

[4] Tristan Bourgois, Jannik Laval, Stéphane Ducasse, and Damien Pollet. Bloc: a trait-based collections library–a preliminary experience report. In *International Workshop on Smalltalk Technologies*, 2010.

[5] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[6] Wes McKinney. *Python for Data Analysis*. O'Reilly Media Inc., 2012.

[7] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[8] NIST/SEMATECH. *e-Handbook of Statistical Methods*. http://www.itl.nist.gov/div898/handbook/. Accessed: 2017-06-20.

[9] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[10] Howard J. Seltman. *Experimental Design and Analysis*. 2015.

[11] R Core Team. R language definition. *Vienna, Austria: R foundation for statistical computing*, 2000.

[12] Hadley Wickham et al. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011.

[13] Stephen Wolfram. *The Mathematica*. Cambridge University Press, 1999.