Data Analysis in Ruby (Daru)

Seminar Report

Submitted in partial fulfilment of the requirements of degree of

Bachelor of Technology

By

Shekhar Prasad Rajak (137152)

Supervisor

**Dr. K. Ramesh**

Associate Professor, CSE Dept.



**Department of Computer Science and Engineering**

**National Institute of Technology,**

**Warangal – 506004**

**2013 – 2017**

**Data Analysis in Ruby (Daru)**

by

Shekhar Prasad Rajak (137152)

is approved for the degree Bachelor of Technology

Examiners

---------------------------------------

---------------------------------------

---------------------------------------

Supervisor

----------------------------------------

(signature)

Dr. K. Ramesh

Associate Professor, CSE Dept.

Chairman

----------------------------------------

Date: 17 April 2017

Place: Warangal

**Department of Computer Science and Engineering**

**National Institute of Technology,**

**Warangal – 506004**

****

**CERTIFICATE**

This is to certify that the seminar report titled “**Data Analysis in Ruby (Daru)”** is a bonafide work carried out by *Shekhar Prasad Rajak* in partial fulfilment of the requirements for the award of the degree Bachelor of Technology (B.Tech) and submitted to the Department of Computer Science and Engineering, National Institute of Technology Warangal.

Dr. K. Ramesh Dr. Ch. Sudhakar

Seminar Supervisor Head of Department

CSE Department CSE Department

NIT Warangal NIT Warangal

# Declaration

I declare that this written submission represents my ideas in my own words and where other ideas or words have been included, I have adequately cited and referenced the original sources I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact source in my submission I understand that any violation of the above will be cause for disciplinary action by the institute and can also evoke penal actions from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

----------------------

Shekhar Prasad Rajak

**Introduction :**

Daru (Data Analysis in RUby) is a library for storage, analysis, manipulation and visualization of data in Ruby.

Daru makes it easy and intuitive to process data predominantly through 2 data structures: Daru::DataFrame and Daru::Vector. Written in pure Ruby works with all ruby implementations. Tested with MRI 2.0, 2.1, 2.2 and 2.3.

Daru (Data Analysis in RUby) is a ruby gem for performing various data analysis and manipulation tasks in Ruby. It draws inspiration from pandas (python) and aims to be completely cross-compatible between all ruby implementations (MRI/JRuby etc.) yet leverage the individual benefits that each interpreter offers (for example the speed of C in MRI), while offering a simple and powerful API for data analysis, manipulation and visualization.

**Data structures**

daru consists of two major data structures:

* Vector - A named one-dimensional array-like structure.
* DataFrame - A named spreadsheet-like two-dimensional frame of data.

A Vector can either be represented by a Ruby Array, NMatrix(MRI) or MDArray(JRuby) internally. This allows for fast data manipulation in native code

Both of these can be indexed by the Daru::Index or Daru::MultiIndex class, which allows us to reference and operate on data by name instead of the traditional numeric indexing, and also perform index-based manipulation, equality and plotting operations.

#### Vector

The easiest way to create a vector is to simply pass the elements to a Daru::Vector constructor:

```

v = Daru::Vector.new [23,44,66,22,11]

# This will create a Vector object

v

# =>

##<Daru::Vector:78168790 @name = nil @size = 5 >

# ni

# 0 23

# 1 44

# 2 66

# 3 22

# 4 11

```

Since no name has been specified, the vector is named nil, and since no index has been specified either, a numeric index from 0..4 has been generated for the vector (leftmost column).

A better way to create vectors would be to specify the name and the indexes:

```

sherlock =

Daru::Vector.new [3,2,1,1,2], name: :sherlock, index: [:pipe, :hat, :violin, :cloak, :shoes]

#=>

#<Daru::Vector:78061610 @name = sherlock @size = 5 >

# sherlock

# pipe 3

# hat 2

# violin 1

# cloak 1

# shoes 2

```

This way we can clearly see the quantity of each item possesed by Sherlock.

Data can be retrieved with the [] operator:

```

sherlock[:pipe] #=> 3

```

#### DataFrame

A basic DataFrame can be constructed by simply specifying the names of columns and their corresponding values in a hash:

```

df = Daru::DataFrame.new({a: [1,2,3,4,5], b: [10,20,30,40,50]}, name: :normal)

# =>

##<Daru::DataFrame:77782370 @name = normal @size = 5>

# a b

# 0 1 10

# 1 2 20

# 2 3 30

# 3 4 40

# 4 5 50

```

You can also specify an index for the DataFrame alongwith the data and also specify the order in which the vectors should appear. Every vector in the DataFrame will carry the same index as the DataFrame once it has been created.

```

plus\_one = Daru::DataFrame.new({a: [1,2,3,4,5], b: [10,20,30,40,50], c: [11,22,33,44,55]}, name: :plus\_one, index: [:a, :e, :i, :o, :u], order: [:c, :a, :b])

# =>

##<Daru::DataFrame:77605450 @name = plus\_one @size = 5>

# c a b

# a 11 1 10

# e 22 2 20

# i 33 3 30

# o 44 4 40

# u 55 5 50

```

daru will also add nil values to vectors that fall short of elements.

```

missing = Daru::DataFrame.new({a: [1,2,3], b: [1]}, name: :missing)

#=>

#<Daru::DataFrame:76043900 @name = missing @size = 3>

# a b

# 0 1 1

# 1 2 nil

# 2 3 nil

```

Creating a DataFrame by specifying Vector objects in place of the values in the hash will correctly align the values according to the index of each vector. If a vector is missing an index present in another vector, that index will be added to the vector with the corresponding value set to nil.

```

a = Daru::Vector.new [1,2,3,4,5], index: [:a, :e, :i, :o, :u]

b = Daru::Vector.new [43,22,13], index: [:i, :a, :queen]

on\_steroids = Daru::DataFrame.new({a: a, b: b}, name: :on\_steroids)

#=>

#<Daru::DataFrame:75841450 @name = on\_steroids @size = 6>

# a b

# a 1 22

# e 2 nil

# i 3 43

# o 4 nil

# queen nil 13

# u 5 nil

```

A DataFrame can be constructed from multiple sources:

* To construct by columns:
  + Array of hashes - Where the key of each hash is the name of the column to which the value belongs.
  + Name-Array Hash - Where the hash key is set as the name of the vector and the data the corresponding value.
  + Name-Vector Hash - This is the most advanced way of creating a DataFrame. Treats the hash key as the name of the vector. Also aligns the data correctly based on index.
  + Array of Arrays - Each sub array will be considered as a Vector in the DataFrame.
* To construct by rows using the .rows class method:
  + Array of Arrays - This will treat each sub-array as an independent row.
  + Array of Vectors - Uses each Vector in the Array as a row of the DataFrame. Sets vector names according to the index of the Vector. Aligns vector elements by index.

#### Handling Data

Some more features of daru by loading some real-life data from a CSV file and performing some operations on it.

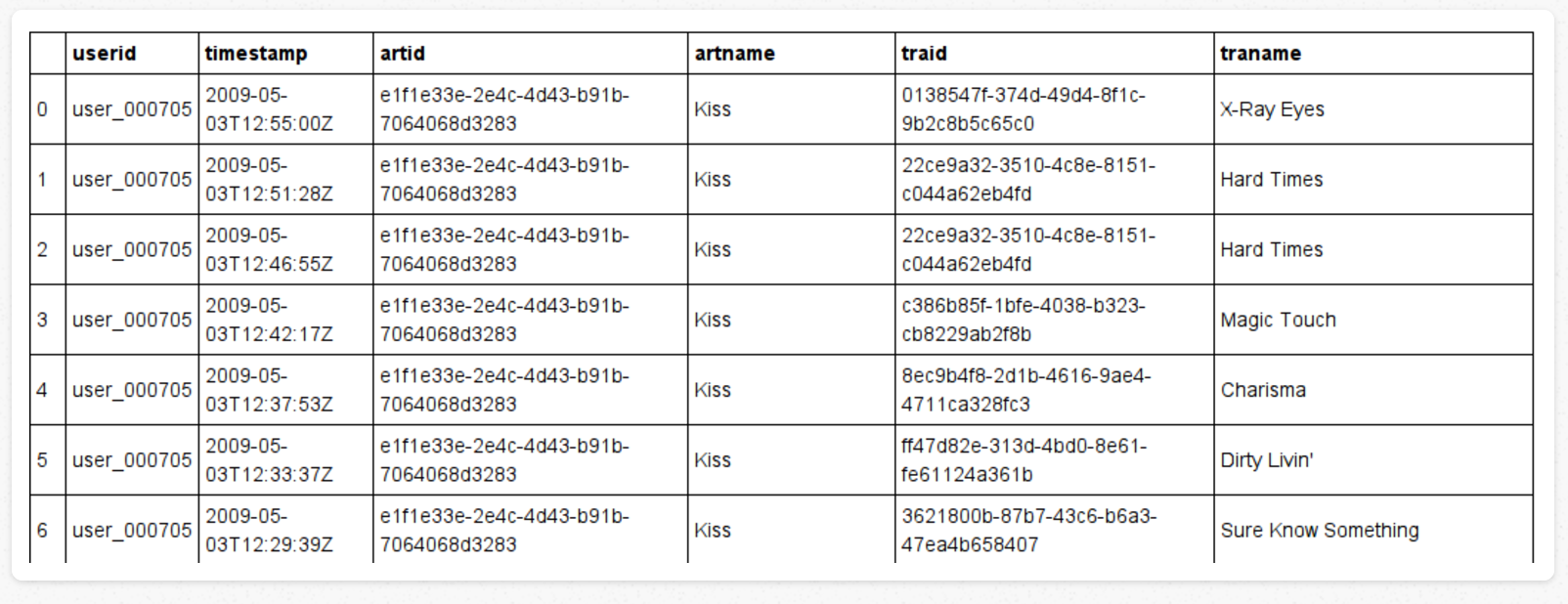
#### Loading Data From Files

```

require 'daru'

df = Daru::DataFrame.from\_csv 'music\_data.tsv', col\_sep: "\t"

```



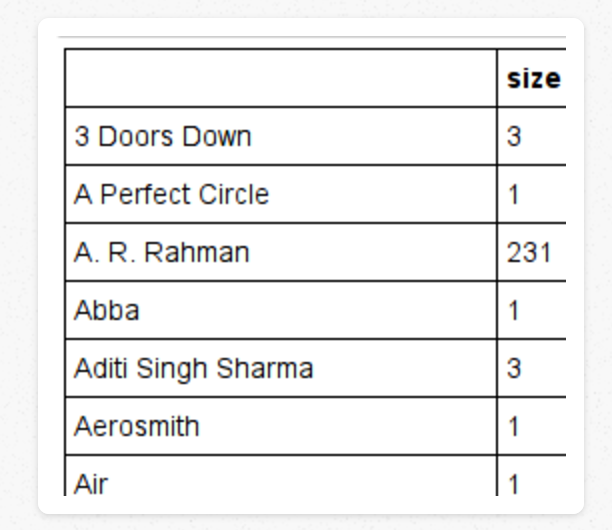
#### Data Analysis

Lets dive deeper by actually trying to extract something useful from the data that we have. Say we want to know the name of the artist heard the maximum number of times. So we create a Vector which consists of the names of the artists as the index and the number of times the name appears in the data as the corresponding values:

```

artists = df.group\_by(:artname).size

```



#### Plotting

daru uses [Nyaplot](https://github.com/domitry/nyaplot) for plotting, which is an optional dependency. Install nyaplot with gem install nyaplot and proceed.

To demonstrate, lets find the top ten artists heard by this user and plot the number of times their songs have been heard against their names in a bar graph. For this, use the #sort function, which will preserve the indexing of the vector.

```

top\_ten = artists.sort(ascending: false)[0..10]

top\_ten.plot type: :bar do |plt|

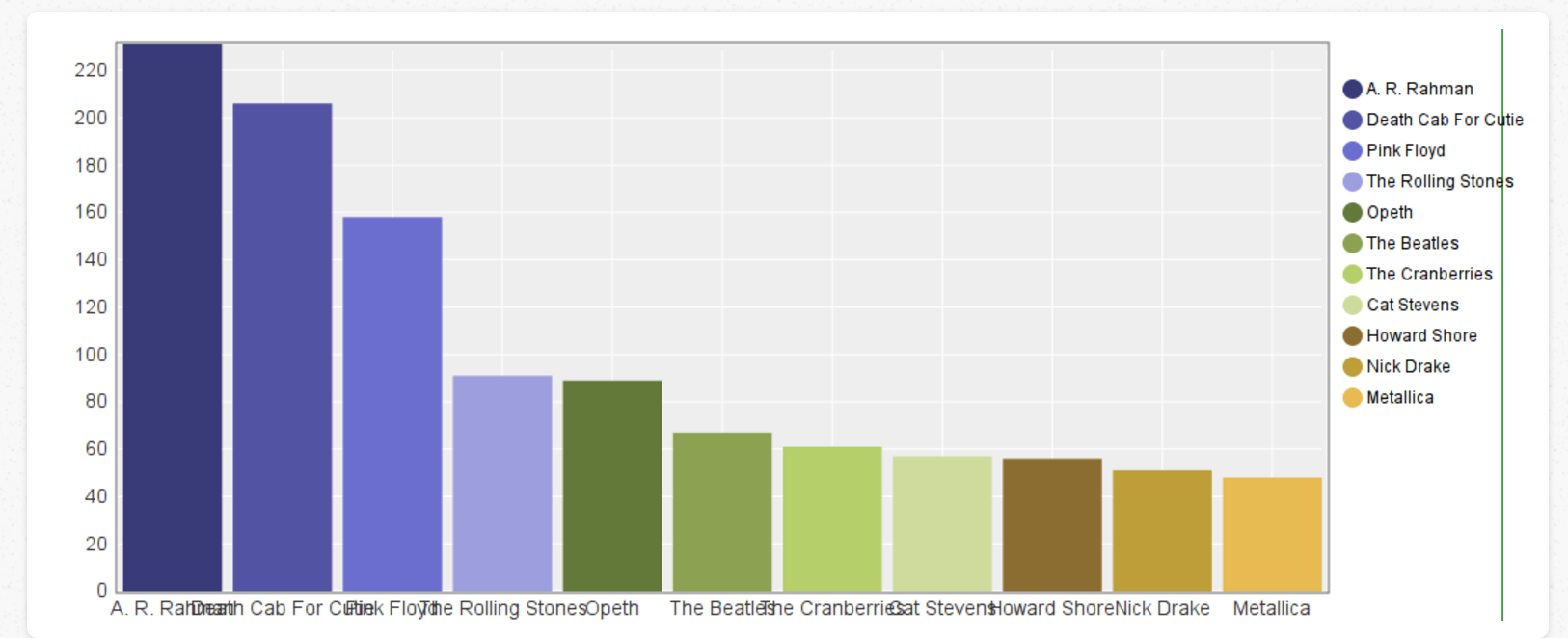
plt.width 1120

plt.height 500

plt.legend true

end

``



## Data Types

You can now either use Ruby Arrays or NMatrix as the underlying implementation. Since NMatrix is fast and makes use of C storage, it is recommended to use nmatrix when dealing with large sets of data. Daru will store any data as Ruby Array unless explicitly specified.

Thus to specify the data type of a Vector use the option :dtype and either supply it with :array or :nmatrix, and if using the NMatrix dtype, you can also specify the C data type that NMatrix will use internall by using the option :nm\_dtype and supplying it with one of the NMatrix data types.

As an example, consider creating a Vector which uses NMatrix underneath, and stores data using the :float64 NMatrix data type, which stands for double precision floating point numbers.

```

v = Daru::Vector.new([1.44,55.54,33.2,5.6],dtype: :nmatrix, nm\_dtype: :float64)

# nil

# 0 1.44

# 1 55.54

# 2 33.2

# 3 5.6

v.dtype #=> :nmatrix

v.type #=> :float64

```

Another distinction between types of data that daru offers is :numeric and :object. This is a generic feature for distinguishing numerical data from other types of data (like Strings or DateTime objects) that might be contained inside Vectors or DataFrames. These distinctions are important because statistical and arithmetic operations can only be applied on structures with type numeric.

To query the data structure for its type, use the #type method. If the underlying implemetation is an NMatrix, it will return the NMatrix data type, otherwise for Ruby Arrays, it will be either :numeric or :object.

```

v = Daru::Vector.new([1,2,3,4], dtype: :array)

v.type #=> :numeric

```

Thus Daru exposes three methods for querying the type of data: \* #type - Get the generic type of data to know whether numeric computation can be performed on the object. Get the C data type used by nmatrix in case of dtype NMatrix. \* #dtype - Get the underlying data representation (either :array or :nmatrix).

## Working with Missing Data

Any data scientist knows how common missing data is in real-life data sets, and to address that need, daru provides a host of functions for this purpose. This functionality is still in its infancy but should be up to speed soon.

The #is\_nil? function will return a Vector object with true if a value is nil and false otherwise.

```

v = Daru::Vector.new([1,2,3,nil,nil,4], index: [:a, :b, :c, :d, :e, :f])

v.is\_nil?

#=>

##<Daru::Vector:93025420 @name = nil @size = 6 >

# nil

# a nil

# b nil

# c nil

# d true

# e true

# f nil

```

The #nil\_positions function returns an Array that contains the indexes of all the nils in the Vector.

```

v.nil\_positions #=> [:d, :e]

```

The #replace\_nils functions replaces nils with a supplied value.

```

v.replace\_nils 69

#=>

##<Daru::Vector:92796730 @name = nil @size = 6 >

# nil

# a 1

# b 2

# c 3

# d 69

# e 69

# f 4

```

The statistics functions implemented on Vectors ensure that missing data is not considered during computation and are thus safe to call on missing data.

## Hierarchical sorting of DataFrame

It is now possible to use the #sort function on Daru::DataFrame such that sorting happens hierarchically according to the order of the specified vector names.

In case you want to sort according to a certain attribute of the data in a particular vector, for example sort a Vector of strings by length, then you can supply a code block to the :by option of the sort method.

Supply the :ascending option with an Array containing ‘true’ or ‘false’ depending on whether you want the corresponding vector sorted in ascending or descending order.

```

df = Daru::DataFrame.new({

a: ['ff' , 'fwwq', 'efe', 'a', 'efef', 'zzzz', 'efgg', 'q', 'ggf'],

b: ['one' , 'one', 'one', 'two', 'two', 'one', 'one', 'two', 'two'],

c: ['small','large','large','small','small','large','small','large','small'],

d: [-1,2,-2,3,-3,4,-5,6,7], e: [2,4,4,6,6,8,10,12,14]

})

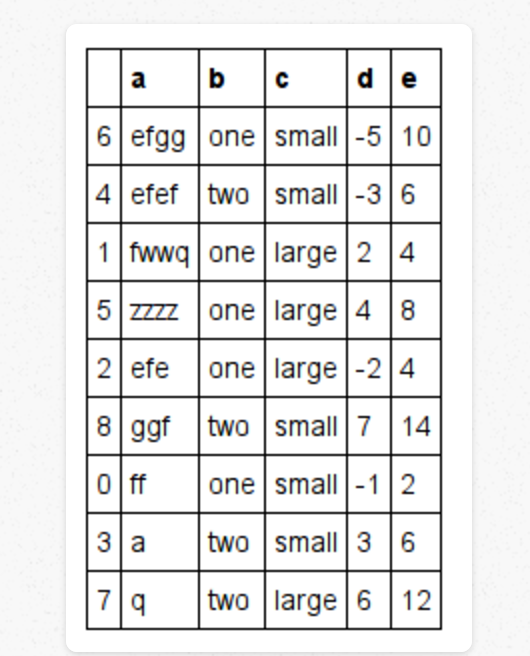
df.sort([:a,:d], by: {

a: lambda { |a,b| a.length <=> b.length },

b: lambda { |a,b| a.abs <=> b.abs } },

ascending: [false, true] )

```



Vector objects also have a similar sorting method implemented.

## DSL for plotting with [Nyaplot](https://github.com/domitry/nyaplot)

To plot a line graph with data present in a DataFrame:

```

df = Daru::DataFrame.new({a: [1,2,3,4,5], b: [10,14,15,17,44]})

df.plot type: :line, x: :a, y: :b do |p,d|

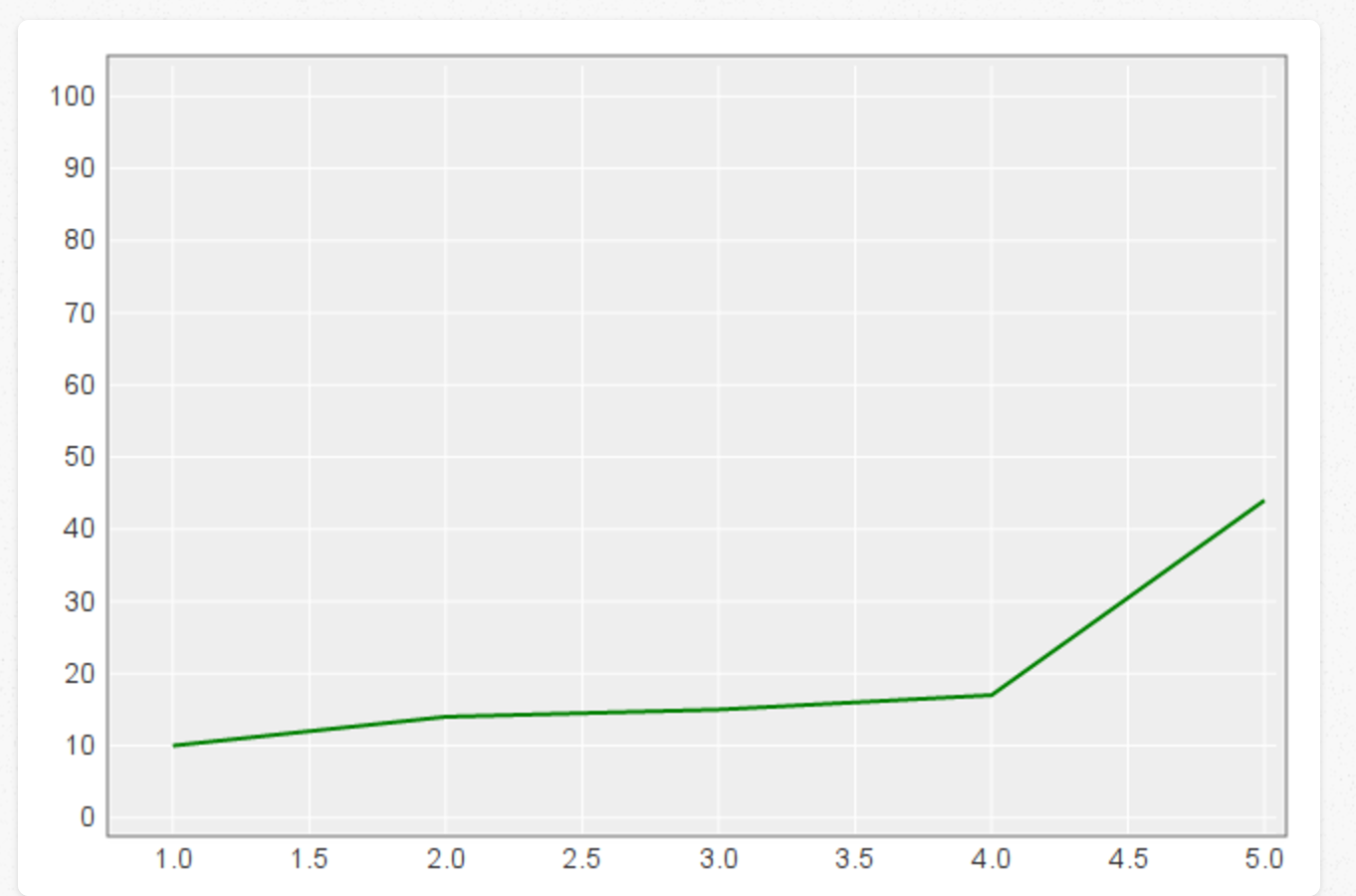
p.yrange [0,100]

p.legend true

d.color "green"

end

```



As you can see, the #plot function exposes the Nyaplot::Plot and Nyaplot::Diagram objects to user after populating them with the relevant data. So the new interface lets experienced users utilize the full power of nyaplot but keeps basic plotting very simple to use for new users or for quick and dirty visualization needs. Unfortunately for now, until a viable solution to interfacing with nyaplot is found, you will need to use the nyaplot API directly.

## Statistics and arithmetic on DataFrames.

Daru includes a host of methods for simple statistical analysis on numeric data. You can call mean, std, sum, product, etc. directly on the DataFrame. The corresponding computation is performed on numeric Vectors within the DataFrame, and missing data if any is excluded from the calculation by default.

```

df = Daru::DataFrame.new({

a: ['foo' , 'foo', 'foo', 'foo', 'foo', 'bar', 'bar', 'bar', 'bar'],

b: ['one' , 'one', 'one', 'two', 'two', 'one', 'one', 'two', 'two'],

c: ['small','large','large','small','small','large','small','large','small'],

d: [1,2,2,3,3,4,5,6,7],

e: [2,4,4,6,6,8,10,12,14],

f: [10,20,20,30,30,40,50,60,70]

})

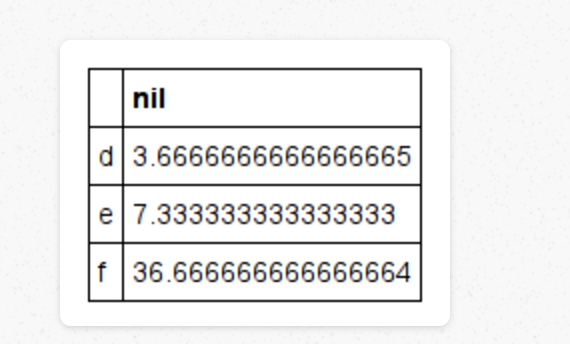
```

To calculate the mean of numeric vectors:

```

df.mean

```

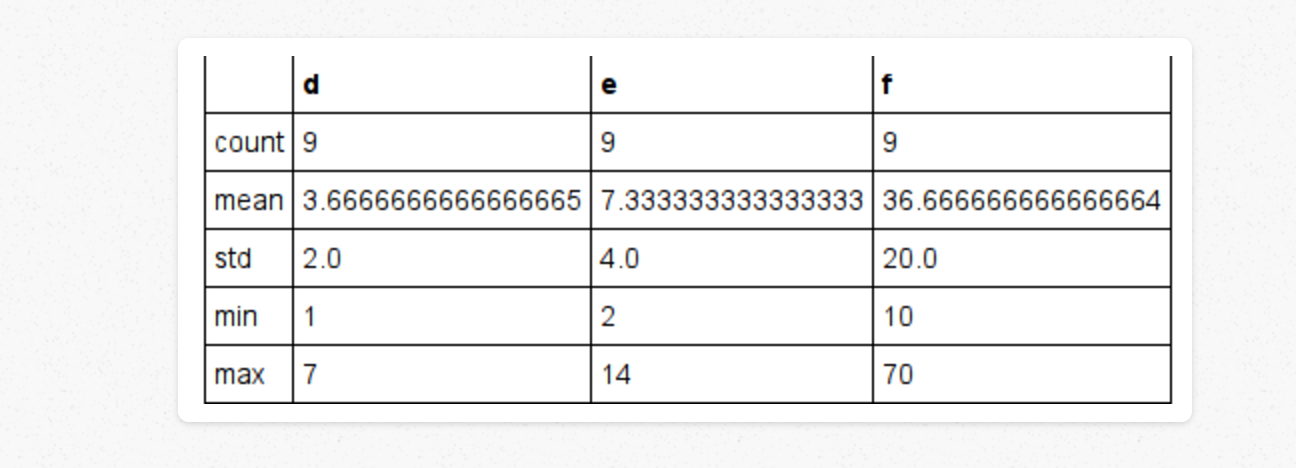


Apart from that you can use the #describe method to calculate many statistical features of numeric Vectors in one shot and see a summary of statistics for numerical vectors in the DataFrame that is returned. For example,

```

df.describe

```



The covariance and correlation coeffiecients between the numeric vectors can also be found with #cov and #corr

```

df.cov

# =>

# #<Daru::DataFrame:91700830 @name = f5ae5d7e-9fcb-46c8-90ac-a6420c9dc27f @size # = 3>

# d e f

# d 4 8 40

# e 8 16 80

# f 40 80 400

```

## Hierarchial indexing

A new way of hierarchially indexing data has been introduced in version 0.0.5. This is done with the new Daru::MultiIndex class. Hierarchial indexing allows grouping sets of similar data by index and lets you select sub sets of data by specifying an index name in the upper hierarchy.

A MultiIndex can be created by passing a bunch of tuples into the Daru::MultiIndex class. A DataFrame or Vector can be created by passing it a MultiIndex object into the index option. A MultiIndex can be used for determining the order of Vectors in a DataFrame too.

```

tuples = [

[:a,:one,:bar],

[:a,:one,:baz],

[:a,:two,:bar],

[:a,:two,:baz],

[:b,:one,:bar],

[:b,:two,:bar],

[:b,:two,:baz],

[:b,:one,:foo],

[:c,:one,:bar],

[:c,:one,:baz],

[:c,:two,:foo],

[:c,:two,:bar] ]

multi\_index = Daru::MultiIndex.new(tuples)

vector\_arry1 = [11,12,13,14,11,12,13,14,11,12,13,14]

vector\_arry2 = [1,2,3,4,1,2,3,4,1,2,3,4]

order\_mi = Daru::MultiIndex.new([

[:a,:one,:bar],

[:a,:two,:baz],

[:b,:two,:foo],

[:b,:one,:foo]])

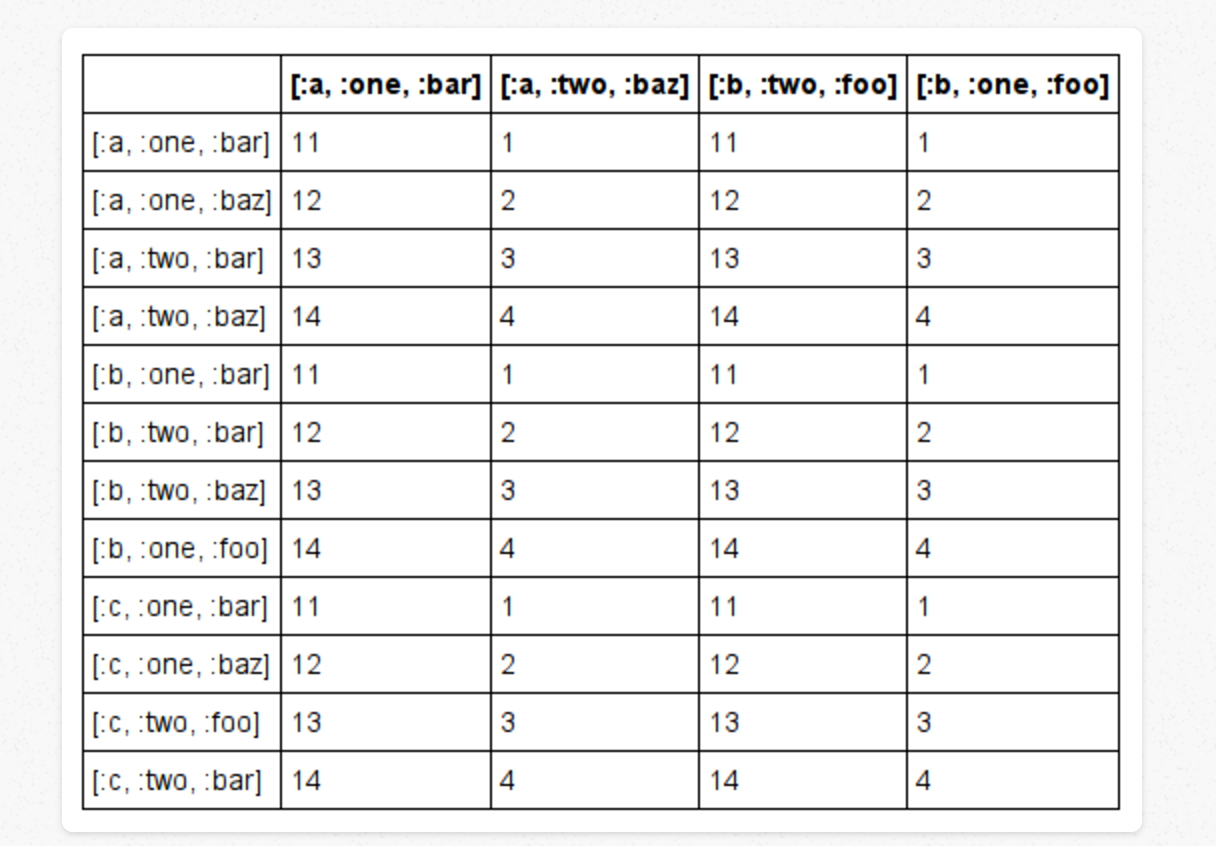
df\_mi = Daru::DataFrame.new([

vector\_arry1, vector\_arry2,

vector\_arry1, vector\_arry2],

order: order\_mi, index: multi\_index)

```

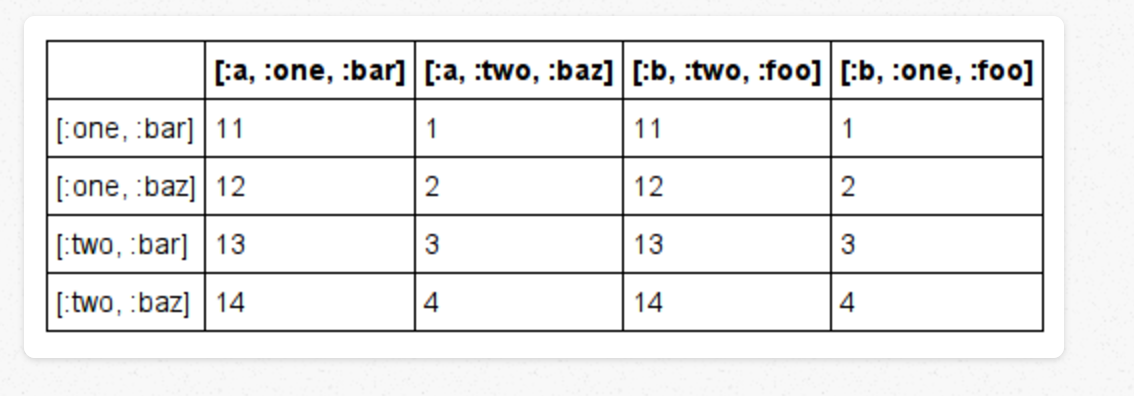


Selecting a top level index from the hierarchy will select all the rows under that name, and return a new DataFrame with just that much data and indexes.

```

df\_mi.row[:a]

```

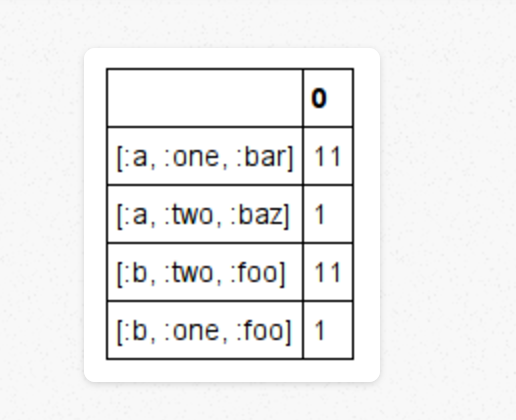


Alternatively passing the entire tuple will return just that row as a Daru::Vector, indexed according to the column index.

```

df\_mi.row[:a, :one,:bar]

```



Hierachical indexing is especially useful when aggregating or splitting data, or generating data summaries as we’ll see in the following examples.

## Splitting and aggregation of data

When dealing with large sets of scattered data, it is often useful to ‘see’ the data grouped according to similar values in a Vector instead of it being scattered all over the place.

The #group\_by function does exactly that. For those familiar SQL, #group\_by works exactly like the GROUP BY clause, but is much easier since its all Ruby.

The #group\_by function will accept one or more Vector names and will scan those vectors for common elements that can be grouped together. In case multiple names are specified it will check for common attributes accross rows.

So for example consider this DataFrame:

```

df = Daru::DataFrame.new({

a: %w{foo bar foo bar foo bar foo foo},

b: %w{one one two three two two one three},

c: [1 ,2 ,3 ,1 ,3 ,6 ,3 ,8],

d: [11 ,22 ,33 ,44 ,55 ,66 ,77 ,88] })

#<Daru::DataFrame:88462950 @name = 0dbc2869-9a82-4044-b72d-a4ef963401fc @size = 8>

# a b c d

# 0 foo one 1 11

# 1 bar one 2 22

# 2 foo two 3 33

# 3 bar three 1 44

# 4 foo two 3 55

# 5 bar two 6 66

# 6 foo one 3 77

# 7 foo three 8 88

```

To group this DataFrame by the columns :a and :b, pass them as arguments to the #group\_by function, which returns a Daru::Core::GroupBy object.

Calling #groups on the returned GroupBy object returns a Hash with the grouped rows

```

grouped = df.group\_by([:a, :b])

grouped.groups

# =>

{

# ["bar", "one"]=>[1],

# ["bar", "three"]=>[3],

# ["bar", "two"]=>[5],

# ["foo", "one"]=>[0, 6],

# ["foo", "three"]=>[7],

# ["foo", "two"]=>[2, 4]}

```

To see the first group of each group from this collection, call #first on the grouped variable. Calling #last will return the last member of each group.

```

grouped.first

#=>

a b c d

# 1 bar one 2 22

# 3 bar three 1 44

# 5 bar two 6 66

# 0 foo one 1 11

# 7 foo three 8 88

# 2 foo two 3 33

```

On a similar note #head(n) will return the first n groups and #tail(n) the last n groups.

The #get\_group function will select only the rows that a particular group belongs to and return a DataFrame with those rows. The original indexing is ofcourse preserved.

```

grouped.get\_group(["foo", "one"])

# =>

# #<Daru::DataFrame:90777050 @name = cdd0afa8-252d-4d07-ad0f-76c7581a492a @size # = 2>

# a b c d

# 0 foo one 1 11

# 6 foo one 3 77

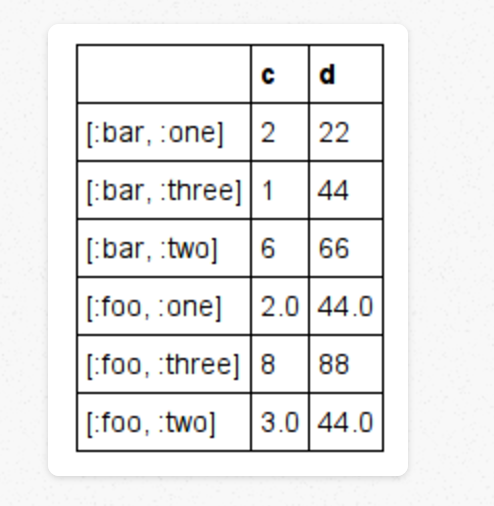
```

The Daru::Core::GroupBy object contains a bunch of methods for creating summaries of the grouped data. These currently include #mean, #std, #product, #sum, etc. and many more to be added in the future. Calling any of the aggregation methods will create a new DataFrame which will have the index as the group and the aggregated data of the non-group vectors as the corresponding value. Of course this aggregation will apply only to :numeric type Vectors and missing data will not be considered while aggregation.

```

grouped.mean

```



**Conclusion**

There are many other things regarding data manipulation and visualization, that Daru can do very easily. It is in the way of replacing some popular data analysis tool in web application and software.

**Reference**

[1] [Data Analysis in RUby: Basic data manipulation and plotting](http://v0dro.github.io/blog/2014/11/25/data-analysis-in-ruby-basic-data-manipulation-and-plotting/)

[2] [Data Analysis in RUby: Splitting, sorting, aggregating data and data types](http://v0dro.github.io/blog/2015/02/24/data-analysis-in-ruby-part-2/)

[3] [Finding and Combining data in daru](http://v0dro.github.io/blog/2015/08/03/finding-and-combining-data-in-daru/)

[4] [Github repository Daru](https://github.com/SciRuby/daru)