



vertica-ml-python0.1 Documentation

Flexible as Python, Fast as Vertica

Ouali Badr

May 2, 2018

Executive Summary

This documentation explains the `vertica-ml-python` library by detailing all the functions and providing significant examples to each one. It allows the user to use his `Vertica` Database with `Python` without loading the data in his personal machine first. All the functions execute requests directly in the database in order to gain in efficiency. It combines `Vertica` aggregations and `Python` flexibility to create an object similar to `pandas.DataFrame` with the power of a columnar oriented analytic database: `Vertica`.

`vertica-ml-python` allows users to use the RVD (Resilient Vertica Dataset). This object keeps in memory all the users modifications in order to use optimized SQL queries to compute all the necessary aggregations. Thanks to this object, the table is intact and will never be modified. The purpose is to explore, preprocess and clean the object without changing the initial table.

What contains `vertica-ml-python`?

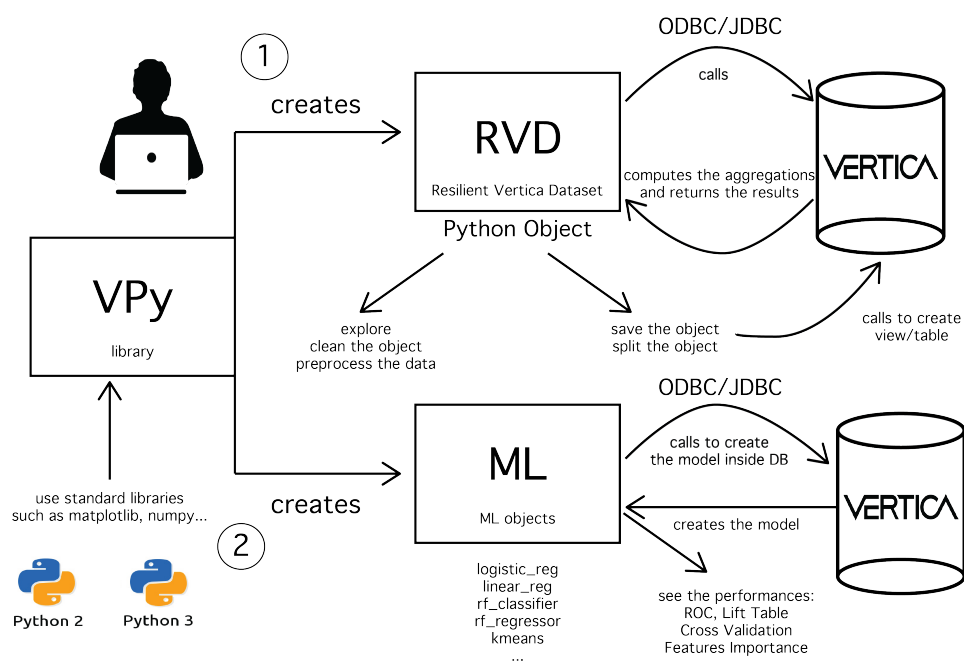
This library contains many functions for:

- Data Exploration, Preprocessing and Cleaning: `vertica_ml_python.rvd`
- Machine Learning (Regression, Classification, Clustering): `vertica_ml_python.vml`

`vertica-ml-python` helps to explore, preprocess and clean the data without changing the initial table. It uses scalable Machine Learning Algorithms such as Logistic Regression, Random Forest and SVM. It allows also to use cross validation of the different models and to compare them.

`vertica-ml-python` uses only the standard `Python` libraries. To connect to the database, it can use both `JDBC` and `ODBC` connection. When the structure is well understood, it is very easy to create the object that the user really wants.

Significant examples using very well-known datasets are available and will help the user to master the different objects. The next figure sums up how the `vertica-ml-python` library works.



Contents

1 Prerequisites	8
1.1 Python Version	8
1.2 Standard Libraries	8
1.3 Installation	8
1.4 Connection to the Database	9
1.4.1 ODBC	9
1.4.2 JDBC	9
2 Jupyter	10
3 Comparison between vertica-ml-python and pandas+scikit	10
3.1 Limitations	10
3.1.1 vertica-ml-python	10
3.1.2 pandas+scikit	10
3.2 Time to load the data	11
3.2.1 vertica-ml-python	11
3.2.2 pandas+scikit	11
3.3 Object Size	11
3.3.1 vertica-ml-python	11
3.3.2 pandas+scikit	12
3.4 Time to execute some queries	12
3.4.1 vertica-ml-python	12
3.4.2 pandas+scikit	13
3.5 Conclusion	15
4 vertica_ml_python.rvd	15
4.1 Warning	15
4.2 Resilient Vertica Dataset (RVD)	15
4.2.1 why RVD?	18
4.2.2 initialization	18
4.2.3 attributes	20
4.2.4 methods	20
4.2.4.1 add_feature	20
4.2.4.2 bar	21
4.2.4.3 corr	22
4.2.4.4 corr_log	23
4.2.4.5 count	25

4.2.4.6	current_table	25
4.2.4.7	describe	26
4.2.4.8	drop_columns	28
4.2.4.9	dsn_restart	28
4.2.4.10	dtypes	29
4.2.4.11	filter	29
4.2.4.12	fully_stacked_bar	30
4.2.4.13	group_by	31
4.2.4.14	head	32
4.2.4.15	help	33
4.2.4.16	hexbin	34
4.2.4.17	hist	35
4.2.4.18	history	36
4.2.4.19	missing	37
4.2.4.20	multiple_hist	38
4.2.4.21	normalize	39
4.2.4.22	pivot_table	40
4.2.4.23	save	41
4.2.4.24	scatter	42
4.2.4.25	scatter_matrix	44
4.2.4.26	select	45
4.2.4.27	set_colors	46
4.2.4.28	set_cursor	46
4.2.4.29	set_dsn	47
4.2.4.30	set_figure_size	47
4.2.4.31	set_legend_loc	47
4.2.4.32	set_limit	48
4.2.4.33	set_offset	48
4.2.4.34	sql_on_off	49
4.2.4.35	stacked_bar	50
4.2.4.36	stacked_hist	51
4.2.4.37	time_on_off	52
4.2.4.38	train_test_split	53
4.2.4.39	undo_all_filters	54
4.2.4.40	undo_filter	55
4.2.4.41	version	55
4.3	Resilient Vertica Column (RVC)	55
4.3.1	attributes	55

4.3.2	methods	56
4.3.2.1	abs	56
4.3.2.2	acos	56
4.3.2.3	add	56
4.3.2.4	asin	56
4.3.2.5	atan	56
4.3.2.6	bar	57
4.3.2.7	boxplot	58
4.3.2.8	cardinality	59
4.3.2.9	category	59
4.3.2.10	convert_to_num	60
4.3.2.11	count	60
4.3.2.12	cos	60
4.3.2.13	cosh	60
4.3.2.14	cot	60
4.3.2.15	date_part	61
4.3.2.16	decode	61
4.3.2.17	degrees	62
4.3.2.18	density	62
4.3.2.19	describe	63
4.3.2.20	distinct	64
4.3.2.21	div	64
4.3.2.22	donut	65
4.3.2.23	drop_column	66
4.3.2.24	dropna	66
4.3.2.25	dtype	66
4.3.2.26	duplicate	67
4.3.2.27	enum	67
4.3.2.28	exp	68
4.3.2.29	fillna	68
4.3.2.30	final_transformation	68
4.3.2.31	floor	69
4.3.2.32	head	69
4.3.2.33	hist	69
4.3.2.34	label_encode	70
4.3.2.35	log	71
4.3.2.36	max	71
4.3.2.37	mean	72

4.3.2.38 mean_encode	72
4.3.2.39 median	72
4.3.2.40 min	73
4.3.2.41 mod	73
4.3.2.42 mult	73
4.3.2.43 normalize	74
4.3.2.44 one_hot_encoder	74
4.3.2.45 outliers	75
4.3.2.46 percentile_cont	76
4.3.2.47 pie	76
4.3.2.48 pow	77
4.3.2.49 radians	77
4.3.2.50 regexp_substr	78
4.3.2.51 rename	78
4.3.2.52 round	79
4.3.2.53 sign	79
4.3.2.54 sin	79
4.3.2.55 sinh	79
4.3.2.56 sqrt	79
4.3.2.57 std	79
4.3.2.58 sub	80
4.3.2.59 tan	80
4.3.2.60 tanh	80
4.3.2.61 undo_impute	80
4.3.2.62 value_counts	80
4.4 Functions	81
4.4.1 drop_table	81
4.4.2 drop_view	82
4.4.3 read_csv	82
4.4.4 run_query	84
5 vertica_ml_python.vml	85
5.1 Functions	85
5.1.1 accuracy	85
5.1.2 auc	86
5.1.3 champion_challenger_binomial	86
5.1.4 confusion_matrix	89
5.1.5 drop_model	90

5.1.6	elbow	91
5.1.7	error_rate	92
5.1.8	features_importance	93
5.1.9	lift_table	94
5.1.10	load_model	96
5.1.11	logloss	96
5.1.12	metric_rf_curve_ntree	97
5.1.13	metric_rf_curve_depth	100
5.1.14	mse	102
5.1.15	reg_metrics	102
5.1.16	rsquared	103
5.1.17	roc	104
5.1.18	summarize_model	105
5.1.19	tree	106
5.2	Machine Learning Models	108
5.2.1	Cross Validation (cross_validate)	108
5.2.1.1	initialization	108
5.2.1.2	attributes	110
5.2.1.3	methods	110
5.2.2	Kmeans (kmeans)	110
5.2.2.1	initialization	110
5.2.2.2	attributes	112
5.2.2.3	methods	112
5.2.3	Linear Regression (linear_reg)	113
5.2.3.1	initialization	113
5.2.3.2	attributes	115
5.2.3.3	methods	115
5.2.4	Logistic Regression (logistic_reg)	116
5.2.4.1	initialization	116
5.2.4.2	attributes	118
5.2.4.3	methods	118
5.2.5	Naive Bayes (naive_bayes)	119
5.2.5.1	initialization	119
5.2.5.2	attributes	120
5.2.5.3	methods	121
5.2.6	Random Forest Classifier (rf_classifier)	122
5.2.6.1	initialization	122
5.2.6.2	attributes	123

5.2.6.3	methods	123
5.2.7	Random Forest Regressor (rf_regressor)	124
5.2.7.1	initialization	124
5.2.7.2	attributes	126
5.2.7.3	methods	126
5.2.8	SVM Classifier (svm_classifier)	127
5.2.8.1	initialization	127
5.2.8.2	attributes	128
5.2.8.3	methods	129
5.2.9	SVM Regressor (svm_regressor)	130
5.2.9.1	initialization	130
5.2.9.2	attributes	131
5.2.9.3	methods	132



VERTICA PYTHON

"Science knows no country, because knowledge belongs to humanity, and is the torch which illuminates the world."

Louis Pasteur

1 Prerequisites

1.1 Python Version

vertica-ml-python works with Python2 and Python3 and it will try to be adapted for both if it is possible.

1.2 Standard Libraries

vertica-ml-python library is only using the standard Python libraries such as pyodbc, matplotlib, time, shutil (only for Python3) and numpy. If one is missing, you can use the following command in your terminal to install it:

```
root@ubuntu:~$ pip install pyodbc
```

Other libraries can be used as anytree for tree visualization or sqlparse for SQL indentation but they are optional.

1.3 Installation

vertica-ml-python doesn't really need installation.

To import easily the vertica-ml-python library from anywhere in your computer just copy paste the **entire** vertica_ml_python folder in the site-package folder of the Python framework. In the MAC environment, you can find it in:
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages

Another way is to call the library from where it is located.

You can then import each library element using a simple syntax.

```
# to import the RVD
from vertica_ml_python import RVD
```

```
3 # to import the logistic regression
  from vertica_ml_python import logistic_reg
```

Everything is well detailed in the following documentation.

1.4 Connection to the Database

This step is useless if `pyodbc` is already installed and you have a DSN in your machine. With this configuration, you do not need to manually create a cursor. It is possible to create a RVD using directly the DSN.

1.4.1 ODBC

To connect to the database, the user can use an ODBC connection to the Vertica database. `pyodbc` provides a cursor that will point to the database. It will be used by the `vertica-ml-python` to create all its objects.

```
import pyodbc

2 # Connection using all the DSN information
4 driver="/Library/Vertica/ODBC/lib/libverticaodbc.dylib"
  server="10.211.55.14"
6 database="testdb"
  port="5433"
8 uid="dbadmin"
  pwd="XxX"
10 dsn= ("DRIVER={}; SERVER={}; DATABASE={}; PORT={}; UID={}; PWD={}; ") .format (
      driver, server, database, port, uid, pwd)
12 cur=pyodbc.connect(dsn).cursor()

14 # Connection using directly the DSN
  dsn= ("DSN=VerticaDSN")
16 cur=pyodbc.connect(dsn).cursor()
```

1.4.2 JDBC

The user can also use a JDBC connection to the Vertica database. For example, `jaydebeapi` provides the cursor used by `vertica-ml-python`.

```
import jaydebeapi

2 uid="dbadmin"
  pwd="XxX"
4 driver="/Library/Vertica/JDBC/vertica-jdbc-9.0.1-0.jar" #Path to JDBC Driver
  url='jdbc:vertica://10.211.55.14:5433/'
6 name='com.vertica.jdbc.Driver'
  cur=jaydebeapi.connect(name,[url,uid,pwd],driver).cursor()
```

2 Jupyter

Jupyter offers a really beautiful interface to use vertica-ml-python. If you want to see how to do fast and easy analytics and ML, feel free to install and use Jupyter.

The screenshot shows a Jupyter Notebook titled "titanic" with a "Last Checkpoint: 24 minutes ago (autosaved)" status. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations, running cells, and markdown. The notebook content shows two code cells:

In [12]: `titanic.describe()`

	count	mean	std	min	25%	50%	75%	max	cardinality
age	1046	29.881137667304	14.4134932112713	0.17	21.0	28.0	39.0	80.0	98
body	121	160.809917355372	97.6969219960031	1.0	72.0	155.0	256.0	328.0	121
fare	1308	33.2954792813456	51.7586682391742	0.0	7.8958	14.4542	31.275	512.3292	281
parch	1309	0.385026737967914	0.865560275349514	0.0	0.0	0.0	0.0	9.0	8
pclass	1309	2.29488158899923	0.837836018970127	1.0	2.0	3.0	3.0	3.0	3
sibsp	1309	0.498854087089382	1.0416583905961	0.0	0.0	0.0	1.0	8.0	7
survived	1309	0.381970970206265	0.486055170866483	0.0	0.0	0.0	1.0	1.0	2

Out[12]: `<column_matrix>`

All the columns seem to have correct information but a lot of ones seem to be useless. Let's now see the missing values.

In [13]: `titanic.missing()`

	total	percent
body	1188	0.908
cabin	1014	0.775
boat	823	0.629
homedest	564	0.431
age	263	0.201

Besides a lot of notebooks will be available in order to understand the library.

3 Comparison between vertica-ml-python and pandas+scikit

All the results of this section are obtained using a single node machine (to have a real comparison) with 128Gb of memory. As pandas is highly distributed, the comparison using this machine is really fare and significant.

3.1 Limitations

3.1.1 vertica-ml-python

vertica-ml-python has no limitation as it uses Vertica to compute all the aggregation it needs. If we want to increase the speed, we can increase the number of nodes or their memory.

3.1.2 pandas+scikit

pandas has real limitation. It loads data in memory and we can not increase it indefinitely.

3.2 Time to load the data

3.2.1 vertica-ml-python

If the data is inside a Vertica database, no operation is needed. In the other case, we need to transfer the data to Vertica. It is highly recommended to already have the data inside Vertica even if the time to create the table is still largely lower than the time to create the pandas.DataFrame.

3.2.2 pandas+scikit

The data must be loaded in memory. It can take a lot of time depending on where the data is. To give a first idea, you can see the following result.

```
# Connector Creation
2 import pandas
  import time
4 conn=pyodbc.connect(dsn)

6 # Using 35M of rows (the whole dataset)
  start_time=time.time()
8 expedia_df=pandas.read_sql("select * from expedia_train",conn)
  print(time.time()-start_time)
10
12 # Output
  1135.22843092306093

14 # Using 10M of rows
  start_time=time.time()
16 expedia_df=pandas.read_sql("select * from expedia_train limit 10000000",conn)
  print(time.time()-start_time)
18
20 # Output
  281.51524472236633

22 # Using 1M of rows
  start_time=time.time()
24 expedia_df=pandas.read_sql("select * from expedia_train limit 1000000",conn)
  print(time.time()-start_time)
26
28 # Output
  42.53745484352112
```

3.3 Object Size

3.3.1 vertica-ml-python

The size of a RVD will never exceed some bytes.

```
import sys
```

```
2 print(sys.getsizeof(expedia_rvd))  
  
4 # Output  
56
```

3.3.2 pandas+scikit

The size of a pandas.DataFrame can become really big.

```
1 print(sys.getsizeof(expedia_df))  
  
3 # Output  
10846031144
```

More than a GB of memory is used... We can imagine the impact in our personal machine. If the user wants to use pandas on very huge dataset, he needs a machine with a lot of memories. Knowing that it is hard to have today a personal machine exceeding 32GB of RAM, the limitation is obvious to small datasets (less than 10GB) and we can not call it big data (we can consider the big data border as 1TB).

3.4 Time to execute some queries

3.4.1 vertica-ml-python

Let's see what it gives for vertica-ml-python.

```
# Using 35M of rows (the whole dataset)  
2 #  
# describe  
4 #  
start_time=time.time()  
6 expedia_rvd.describe()  
print(time.time()-start_time)  
8  
# Output  
10 94.45896601676941  
12 #  
# categorical hist  
14 #  
start_time=time.time()  
16 expedia_rvd["is_mobile"].hist()  
print(time.time()-start_time)  
18  
# Output  
20 0.480072021484375  
22 # Using 10M of rows  
#
```

```
24 # describe
25 #
26 start_time=time.time()
27 expedia_rvd.describe()
28 print(time.time()-start_time)
29
30 # Output
31 28.198142528533936
32
33 #
34 # categorical hist
35 #
36 start_time=time.time()
37 expedia_rvd["is_mobile"].hist()
38 print(time.time()-start_time)
39
40 # Output
41 0.4742517471313477
42
43 # Using 1M of rows
44 #
45 # describe
46 #
47 start_time=time.time()
48 expedia_rvd.describe()
49 print(time.time()-start_time)
50
51 # Output
52 2.8560750484466553
53
54 #
55 # categorical hist
56 #
57 start_time=time.time()
58 expedia_rvd["is_mobile"].hist()
59 print(time.time()-start_time)
60
61 # Output
62 0.3278229236602783
```

3.4.2 pandas+scikit

And now for pandas.

```
# Using 35M of rows (the whole dataset)
2 #
# describe
```

```
4 #
  start_time=time.time()
6 expedia_df.describe()
  print(time.time()-start_time)
8
  # Output
10 75.72856092453003
12
  # categorical hist
14 #
  start_time=time.time()
16 expedia_df["is_mobile"].hist()
  print(time.time()-start_time)
18
  # Output
20 3.210484266281128
22
  # Using 10M of rows
  #
24 # describe
  #
26 start_time=time.time()
  expedia_df.describe()
28 print(time.time()-start_time)
30
  # Output
  20.789332389831543
32
  #
34 # categorical hist
  #
36 start_time=time.time()
  expedia_df["is_mobile"].hist()
38 print(time.time()-start_time)
40
  # Output
  0.36867237091064453
42
  # Using 1M of rows
  #
44 # describe
  #
46 start_time=time.time()
  expedia_df.describe()
48 print(time.time()-start_time)
50
```

```
# Output
52 1.2540433406829834

54 #
56 #
58 start_time=time.time()
expedia_df["is_mobile"].hist()
print(time.time()-start_time)


60 # Output
62 0.08945250511169434
```

3.5 Conclusion

At the end, we have more or less the same execution time with a little advantage to scikit-learn for some functions but do not forget that when the data is inside memory some computation can be faster if the system is highly parallelized. Besides, the loading time can increase considerably if the dataset is bigger (in this example, vertica-ml-python is using an ODBC connection which can add some additional time depending on the case). At the end, vertica-ml-python is more robust and offers more possibilities (without speaking about data exploration where vertica-ml-python is only using standard libraries and offers much more possibilities). Besides some functions to prepare the data will take a lot of time using pandas whereas it will take no time for vertica-ml-python as only the grammar of the transformation is kept in memory. If the projections are correctly made and if we use many nodes, we can use really huge dataset very fast using vertica-ml-python and the power of Vertica.

4 vertica_ml_python.rvd

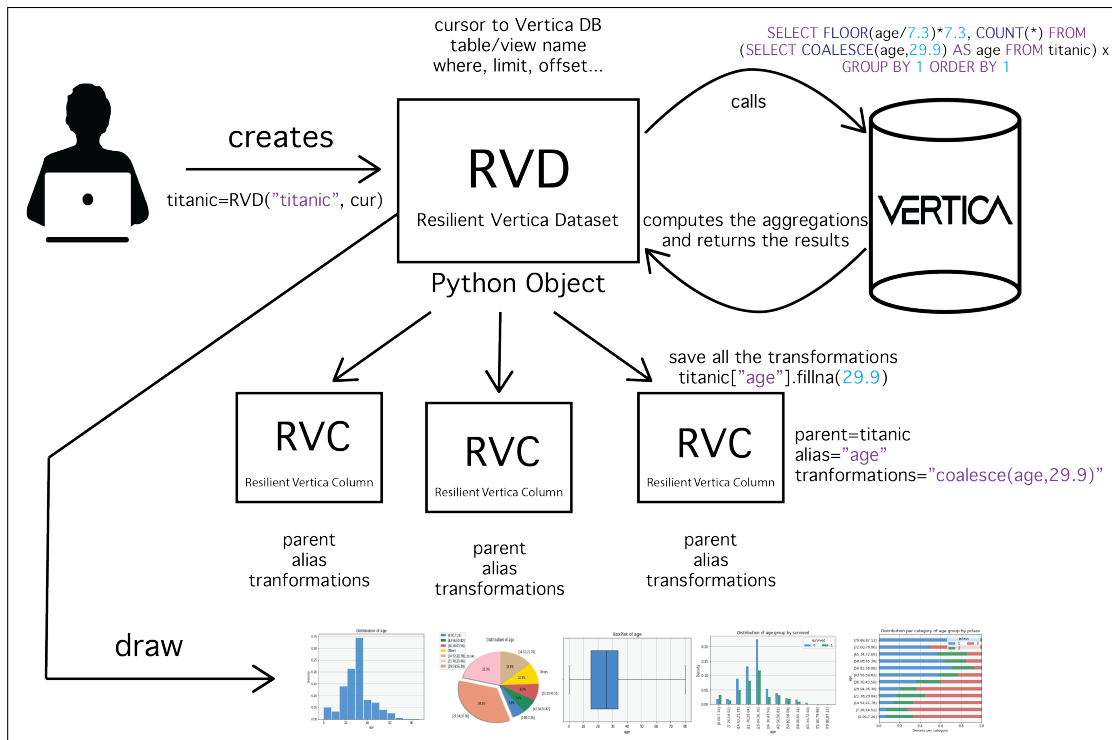
4.1 Warning

 **Before creating a RVD be sure that the names of your table/view columns do not contain illegal characters such as space ' ', dot '.', comma ','... Be sure to have the required libraries installed and check your Vertica version. You can have information about what you can do using the `RVD.version()` method.**

4.2 Resilient Vertica Dataset (RVD)

The RVD is a `Python` object which will keep in mind all the user modifications in order to use an optimized SQL query. It will send the query to the database which will use its aggregations to compute fast results. It is created using a view or a table stored in the user database and a database cursor. It will create for each column of the table a RVC (Resilient Vertica Column) which will store for each column its name, its imputations and allows to do easy modifications and explorations.

RVC and RVD coexist and one can not live without the other. RVC will use the RVD information and reciprocally. It is imperative to understand both structures to know how to use the entire object.



When the user imputes or filters the data, the RVD gets in memory all the transformations to select for each query the needed data in the input relation. Let's try to understand thanks to an example.

```
# We create the RVD
2 titanic=RVD('titanic', cur)

4 # We filter some values
titanic.filter("fare<100")

6
85 elements were filtered

8
# We impute the column age
10 titanic["age"].fillna(method="mean")

12
258 elements were filled

14
# We drop some missing values
titanic["fare"].dropna()

16
Nothing was dropped

18
# We encode the column embarked
20 titanic["embarked"].label_encode()

22 embarked      encoding
Q                0
```

```

24 C                1
   S                2
26 The label encoding was successfully done.

28 # We print all the queries used during the next executions
   titanic.sql_on_off(reindent=True)

30 # We summarize our RVD
32 titanic.describe()

```

We can see as follows, the query which was generated by our RVD.

```

1  select summarize_numcol(age, body, embarked, fare, parch, pclass, sibsp,
   summarize_numcol(survived) over ()
   from
3  (select *
   from
5      (select coalesce(age,29.3879399585921) as age,
        boat as boat,
7        body as body,
        cabin as cabin,
9        decode(embarked,NULL,0,'Q',1,'C',2,'S',3,4) as embarked,
        fare as fare,
11       homedest as homedest,
        name as name,
13       parch as parch,
        pclass as pclass,
15       sex as sex,
        sibsp as sibsp,
17       survived as survived,
        ticket as ticket
19   from
        (select age as age,
21          boat as boat,
          body as body,
23          cabin as cabin,
          embarked as embarked,
25          fare as fare,
          homedest as homedest,
27          name as name,
          parch as parch,
29          pclass as pclass,
          sex as sex,
31          sibsp as sibsp,
          survived as survived,
33          ticket as ticket
        from titanic) t1
35   where fare<100

```

```

offset 0) t2
37 where fare is not null) new_table

```

The RVD will try to keep in mind where the transformations occurred in order to use the appropriate query. In that case, when the user has done a lot of transformations, it is highly recommended to save the RVD in order to gain in efficiency (using the save method). We can also see all the modifications using the `history` method.

```

1 titanic.history()
3 #Output
The RVD was modified many times:
5 * {Tue Feb  6 15:44:51 2018} [Filter]: 85 elements were filtered using the
   filter 'fare<100'
   * {Tue Feb  6 15:44:51 2018} [Fillna]: 258 missing values of the RVC 'age'
   were filled using the imputation 'coalesce({},29.3879399585921)'.
7 * {Tue Feb  6 15:44:52 2018} [Label Encode]: The RVC 'embarked' was imputed
   with the 'label encoding'.
   None => 0
9   Q => 1
   C => 2
11  S => 3
   others => 4

```

4.2.1 why RVD?

It is normal to ask the question. When we look at the definition of resilience, we can see that "Resilience is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation" (Wikipedia). As the RVD keeps in mind all the user actions, it can easily be recreated from scratch. Besides, if the connection to the database failed, it is easy to set a new database cursor using the `set_cursor` method or if the connection was made using a DSN the `dsn_restart` method. The RVD will never load data inside, all the aggregations are computed thanks to Vertica. Having this information, we know that the object will never crash because of the data. It is then resilient according to the definition.

As all the data are inside Vertica, we can call it Vertica Dataset. We can even call the object RVVD (Resilient Virtual Vertica Dataset) as none of the modifications are made in the table. The RVD will just send SQL queries to get the information it needs.

4.2.2 initialization

To instantiate a new RVD, the user can use a database cursor and the name of a relation (table or view).

```

1 from vertica_ml_python import RVD
2
myRVD=RVD(input_relation,cursor)

```

The simplest way is to use directly a DSN (without setting a cursor).

```

1 myRVD=RVD(input_relation,dsn="VerticaDSN")

```

Using this method, the RVD keeps in mind the DSN name and in case of connection failure the user can restart a connection using the `dsn_restart` method.

Example

```

1 titanic=RVD("titanic", cursor)
2 print(titanic)
3
4 #Output
5
6      age      boat      body      cabin      embarked      fare      \\
7 0      None      1      None      None      S      26.00000      \\
8 1      None      10     None      E101      Q      12.35000      \\
9 2      None      10     None      None      S      16.10000      \\
10 3      None      11     None      None      S      33.00000      \\
11 4      None      13     None      None      Q      7.72080      \\
12 5      None      13     None      None      Q      7.73330      \\
13 6      None      13     None      None      Q      7.75000      \\
14 7      None      13     None      None      Q      7.78750      \\
15 8      None      13     None      None      Q      7.82920      \\
16 9      None      13     None      None      Q      7.87920      \\
17 10     None      13     None      None      S      8.11250      \\
18 11     None      13     None      None      S      56.49580      \\
19 12     None      14     None      None      C      30.69580      \\
20 13     None      14     None      None      Q      7.75000      \\
21 14     None      14     None      None      S      13.00000      \\
22 15     None      15     None      None      C      7.22920      \\
23 16     None      15     None      None      Q      7.75000      \\
24 17     None      15     None      None      S      7.05000      \\
25 18     None      15 16     None      None      Q      7.75000      \\
26 19     None      16     None      None      Q      7.73330      \\
27 20     None      16     None      None      Q      7.73750      \\
28 21     None      16     None      None      Q      7.75000      \\
29 22     None      16     None      None      Q      7.75000      \\
30 23     None      16     None      None      Q      7.87920      \\
31 24     None      16     None      None      Q      15.50000      \\
32 25     None      16     None      None      Q      15.50000      \\
33 26     None      16     None      None      Q      15.50000      \\
34 27     None      16     None      None      Q      23.25000      \\
35 28     None      16     None      None      Q      23.25000      \\
36 29     None      16     None      None      Q      23.25000      \\
37 ...      ...      ...      ...      ...      ...      ...      \\
38 --More--
39 Name: titanic, Number of rows: 1309, Number of columns: 14

```

In all the examples used, we consider that we created the four following RVDs:

```

1 titanic=RVD("titanic", cursor)
2 iris=RVD("iris", cursor)

```

```
expedia=RVD("expedia", cursor)
```

4.2.3 attributes

When the RVD is created, it will create as many RVC as there are columns in the input relation (`columns` attribute). It will keep in mind the cursor, the input relation and the dsn if this connection method is used. It will also create 8 other attributes in order to keep in mind the user modifications.

- `limit`: the maximum number of elements taken into account.
- `offset`: the number of skipped elements in the relation.
- `query_on`: print all the queries executed by the RVD in the terminal.
- `reindent`: indent all the queries printed by the RVD using `sqlparse` (available in github)
- `time_on`: print all the queries elapsed time in the terminal.
- `legend_loc`: the legend location.
- `rvd_history`: keep in mind all the user actions.
- `colors`: the colors used to draw the different charts.

For example, when the user wants to create a new column in the RVD. The RVD will create a new RVC and will add it in the list of RVC (`columns` attribute). If it wants to delete a column, the RVD will simply delete it from the list of RVC. The table initial is then never changed !

4.2.4 methods

4.2.4.1 add_feature

```
RVD.add_feature(alias, imputation)
```

Add a new feature to the RVD (a new RVC will be created).

Parameters

- **alias**: `<str>`
Name of the new feature.
- **imputation**: `<str>`
Relation used to compute the new feature.

Example

```
titanic.add_feature(alias="family_size", imputation="parch+sibsp+1")

#Output
The new RVC 'family_size' was added to the RVD.
```

4.2.4.2 bar

```
RVD.bar(columns, method="density", of=None, max_cardinality=[6,6], h=[None,
None], color=None, limit_distinct_elements=200)
```

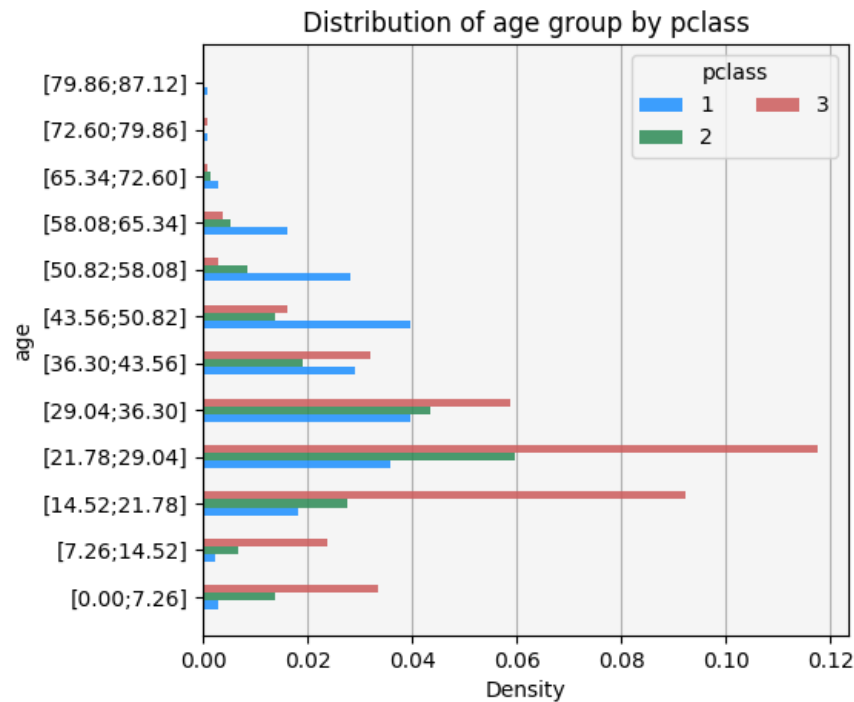
Draw the corresponding 2 variables bar chart.

Parameters

- **columns:** *<list of str>*
The two columns used to draw the bar chart (first will be on the x-axis and the second in the y-axis)
- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<list of positive int>*, optional
The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<list of positive float>*, optional
The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **color:** *<list of str>*, optional
The color list for each category
- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.

Example

```
titanic.bar(columns=["age", "pclass"])
```



4.2.4.3 corr

```
RVD.corr(columns=[], cmap="PRGn", show=True)
```

Compute the correlation matrix of the corresponding RVD columns, excluding NA/null values.

Parameters

- **columns:** *<list of str>*, optional
List of the columns the user wants to consider.
- **cmap:** *<str>*, optional
Color Maps.
- **show:** *<bool>*, optional
Display the result using matplotlib.

Returns

An object named `column_matrix` containing the matrix (the information will be stored in the `data_columns` attribute).

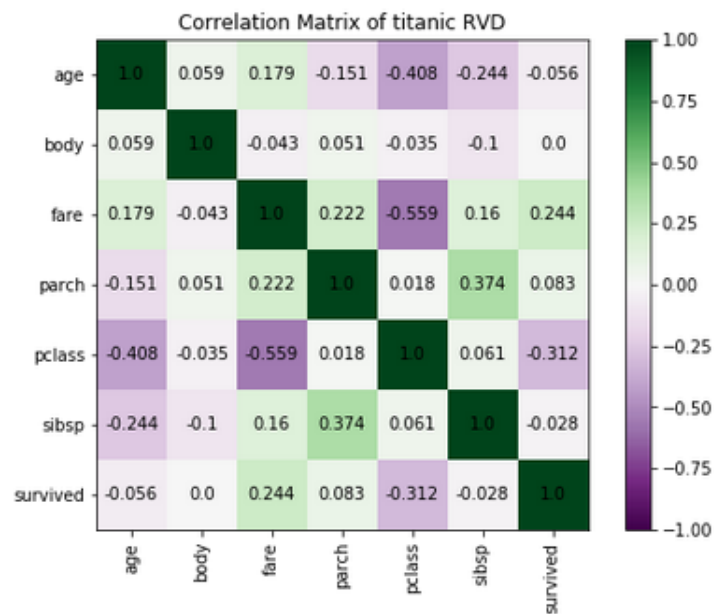
Example

```
1 titanic.corr()
3 #Output
```

```

5      age      body      fare      parch      pclass      \\
age      1      0.059      0.179      -0.151      -0.408      \\
body     0.059      1      -0.043      0.051      -0.035      \\
7  fare     0.179      -0.043      1      0.222      -0.559      \\
parch    -0.151      0.051      0.222      1      0.018      \\
9  pclass   -0.408      -0.035      -0.559      0.018      1      \\
sibsp    -0.244      -0.1      0.16      0.374      0.061      \\
11 survived -0.056      0      0.244      0.083      -0.312      \\
      sibsp      survived
13 age      -0.244      -0.056
body      -0.1      0
15 fare      0.16      0.244
parch      0.374      0.083
17 pclass    0.061      -0.312
sibsp      1      -0.028
19 survived  -0.028      1

```



4.2.4.4 corr_log

```
1 RVD.corr_log(columns=[], cmap="PRGn", epsilon=1e-8, show=True)
```

Compute the log correlation matrix of the corresponding RVD columns, excluding NA/null values.

Parameters

- **columns:** *<list of str>*, optional
List of the columns the user wants to consider.

- **cmap:** *<str>*, optional
Color Maps.
- **epsilon:** *<float>*, optional
Add epsilon to the log computation in order to avoid forbidden values.
- **show:** *<bool>*, optional
Display the result using matplotlib.

Returns

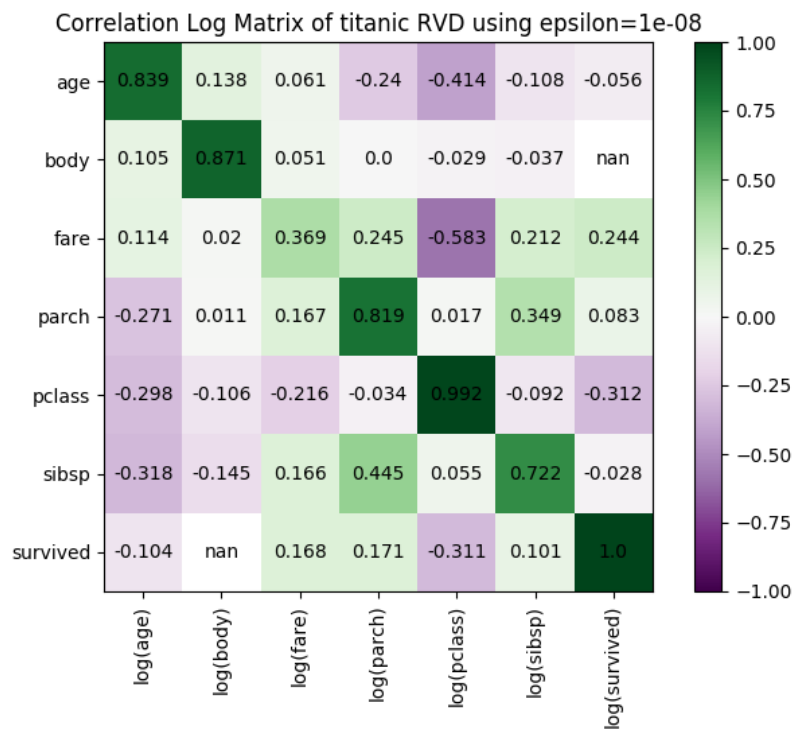
An object named `column_matrix` containing the matrix (the information will be stored in the `data_columns` attribute).

Example

```

1 titanic.corr()
3 #Output
      pclass)  \
5 age      0.839      0.138      0.061      -0.24
      -0.414  \
      body      0.105      0.871      0.051      0.0
      -0.029  \
7 fare      0.114      0.02      0.369      0.245
      -0.583  \
      parch      -0.271      0.011      0.167      0.819
      0.017  \
9 pclass      -0.298      -0.106      -0.216      -0.034
      0.992  \
      sibsp      -0.318      -0.145      0.166      0.445
      0.055  \
11 survived      -0.104      None      0.168      0.171
      -0.311  \
      log(sibsp)  log(survived)
13 age      -0.108      -0.056
      body      -0.037      None
15 fare      0.212      0.244
      parch      0.349      0.083
17 pclass      -0.092      -0.312
      sibsp      0.722      -0.028
19 survived      0.101      1.0

```



4.2.4.5 count

```
1 RVD.count()
```

Returns

Returns the RVD count.

Example

```
1 titanic.count()
3 #Output
1309
```

4.2.4.6 current_table

```
RVD.current_table()
```

Returns

Returns the RVD current virtual table.

Example

```

1 titanic.current_table()
3
3 #Output
"(select * from (select coalesce(age,avg(age) over (partition by pclass,gender
  )) as age, decode(embarked,'Q',0,'C',1,'S',2,3) as embarked, fare as fare,
  parch as parch, pclass as pclass, sibsp as sibsp, survived as survived,
  decode(gender,'female',0,'male',1,2) as gender from (select age as age,
  embarked as embarked, fare as fare, parch as parch, pclass as pclass,
  sibsp as sibsp, survived as survived, sex as gender from titanic) t1
  offset 0) t2 where embarked is not null and fare is not null) new_table"

```

4.2.4.7 describe

```
RVD.describe(mode="auto", columns=None, include_cardinality=True)
```

Summarize the dataset with mathematical information.

Parameters

- **mode:** <str>, optional
 auto | all | categorical | date
 auto (default): This mode is used to have only numerical information. Other types are ignored.
 all: This mode is used to print each column information one by one.
 categorical: This mode is used to only print the categorical variables information (text or *cardinality* ≤ 6)
 date: This mode is used to only print the date variables information
- **columns:** <list of str>, optional
 The columns used to compute the mathematical information.
- **include_cardinality:** <bool>, optional
 Include the cardinality of each element in the computation (only used when mode is "auto")

Returns

Only when mode is "auto": An object named column_matrix containing all the summarized information (they will be stored in the data_columns attribute).

Note

The mathematical information are different depending on the data type and if they are categorical. For more flexibility, the user can use the RVC `describe` method which is specific to each column of the dataset.

Example

```

1 #auto
titanic.describe()
3
3 #Output

```

```

5      count      mean      std      min  \\
age      1046      29.881137667304      14.4134932112713      0.17  \\
7 body      121      160.809917355372      97.6969219960031      1.0  \\
fare      1308      33.2954792813456      51.7586682391741      0.0  \\
9 parch      1309      0.385026737967914      0.865560275349515      0.0  \\
pclass      1309      2.29488158899923      0.837836018970128      1.0  \\
11 sibsp      1309      0.498854087089381      1.0416583905961      0.0  \\
survived      1309      0.381970970206264      0.486055170866483      0.0  \\
13      25%      50%      75%      max      cardinality
age      21.0      28.0      39.0      80.0      98
15 body      72.0      155.0      256.0      328.0      121
fare      7.8958      14.4542      31.275      512.3292      281
17 parch      0.0      0.0      0.0      9.0      8
pclass      2.0      3.0      3.0      3.0      3
19 sibsp      0.0      0.0      1.0      8.0      7
survived      0.0      0.0      1.0      1.0      2
21
#all
23 titanic.describe(mode="all")

25 #Output
count      1046
27 mean      29.881137667304
std      14.4134932112713
29 min      0.17
25%      21.0
31 50%      28.0
75%      39.0
33 max      80.0
cardinality      98
35 Name: age, dtype: numeric(6,3)
-----
37 823      None
252      Others
39 39      13
38      C
41 37      15
33 33      14
43 31      4
cardinality      27
45 Name: boat, dtype: varchar(30)
-----
47 count      121
mean      160.809917355372
49 std      97.6969219960031
min      1.0
51 25%      72.0

```

```

53 50%                155.0
75%                256.0
max                328.0
55 cardinality        121
Name: body, dtype: int
57 -----
--More--

```

4.2.4.8 drop_columns

```
RVD.drop_columns(columns=[])
```

Drop all the selected RVC from the RVD.

Parameters

- **columns:** <list of str>
Name of the different columns.

Example

```

1 titanic.drop_columns(columns=["body", "cabin", "boat", "homedest", "ticket"])
3 #Output
RVC 'body' deleted from the RVD.
5 RVC 'cabin' deleted from the RVD.
RVC 'boat' deleted from the RVD.
7 RVC 'homedest' deleted from the RVD.
RVC 'ticket' deleted from the RVD.

```

4.2.4.9 dsn_restart

```
RVD.dsn_restart()
```

Set a new RVD cursor using the corresponding DSN.

Example

```

1 # If the connection to the Vertica DB failed and you used a DSN to create the
  RVD
titanic.dsn_restart()

```

4.2.4.10 dtypes

```
RVD.dtypes()
```

Print all the RVC of the RVD and their corresponding types.

Example

```
1 titanic.dtypes()
3 #Output
                                     type
5 age          numeric(6,3)
6 boat         varchar(30)
7 body          int
8 cabin         varchar(30)
9 embarked     varchar(20)
10 fare         numeric(10,5)
11 homedest     varchar(100)
12 name         varchar(164)
13 parch        int
14 pclass       int
15 sex          varchar(20)
16 sibsp        int
17 survived     int
18 ticket       varchar(36)
19 Name: titanic, Number of rows: 1309, Number of columns: 14
```

4.2.4.11 filter

```
1 RVD.filter(conditions)
```

Filter the values of the RVD (adding a where clause) following the conditions.

Parameters

- **conditions:** <list of str>
The filtering conditions.

Example

```
1 titanic.filter(["age>3", "fare<100"])
3 #Output
304 elements were filtered
```

```
5 78 elements were filtered
```

4.2.4.12 fully_stacked_bar

```
1 RVD.fully_stacked_bar(columns, max_cardinality=[6,6], h=[None, None], color=
  None, limit_distinct_elements=200)
```

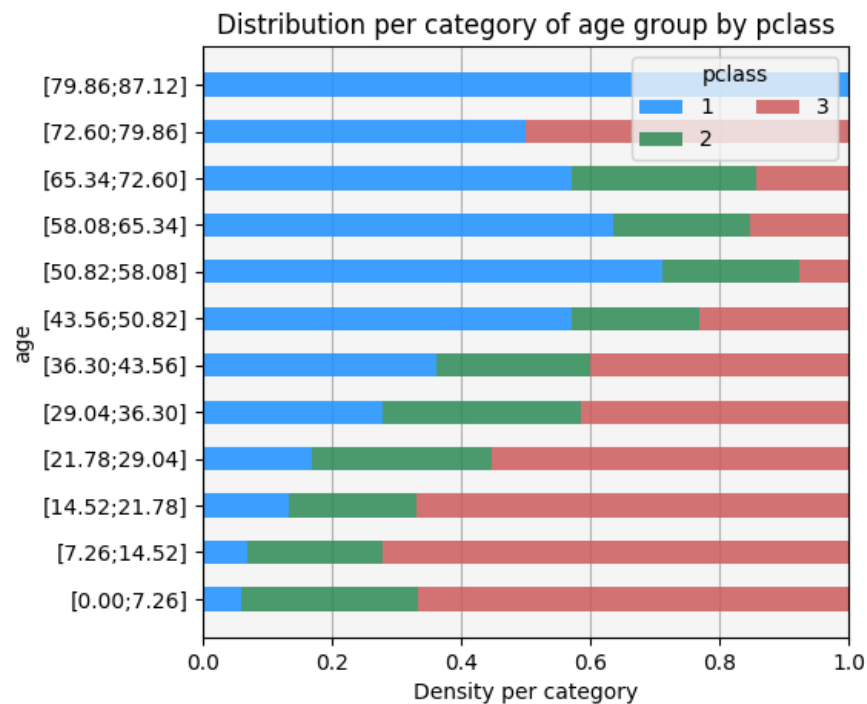
Draw the corresponding 2 variables fully stacked bar chart.

Parameters

- **columns:** *<list of str>*
The two columns used to draw the bar chart (first will be on the x-axis and the second in the y-axis)
- **max_cardinality:** *<list of positive int>*, optional
The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<list of positive float>*, optional
The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically.
- **color:** *<list of str>*, optional
The color list for each category
- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.

Example

```
1 titanic.fully_stacked_bar(columns=["age", "pclass"])
```



4.2.4.13 group_by

```
RVD.group_by(columns, aggregations, order_by=None, limit=1000)
```

Group the aggregations by the corresponding columns.

Parameters

- **columns:** *<list of str>*
List of the columns to use for the group by.
- **aggregations:** *<list of str>*
List of all the aggregations to compute.
- **order_by:** *<list of str>*, optional
List of all the columns to order with.
- **limit:** *<positive int>*, optional
The maximum number of elements to take into account.

Returns

An object named `column_matrix` containing all the grouped information (they will be stored in the `data_columns` attribute).

Example


```

1 titanic.groupby(['pclass','age'], ["count(*)","sum(survived)","sum(fare)"],
   limit=20)
3 #Output
   pclass      age  count  sum_survived  sum_fare
5 0      1      None    39            19  2072.26650
1 1      1    0.920     1             1   151.55000
7 2      1    2.000     1             0   151.55000
3 3      1    4.000     1             1    81.85830
9 4      1    6.000     1             1   134.50000
5 5      1   11.000     1             1   120.00000
11 6      1   13.000     1             1   262.37500
7 7      1   14.000     1             1   120.00000
13 8      1   15.000     1             1   211.33750
9 9      1   16.000     3             3   183.87920
15 10     1   17.000     4             3   323.88330
11 11     1   18.000     6             5   791.55000
17 12     1   19.000     5             3   463.46250
13 13     1   21.000     5             4   505.55000
19 14     1   22.000     7             6   579.66250
15 15     1   23.000     6             5   698.55830
21 16     1   24.000     9             6  1003.25000
17 17     1   25.000     5             3   379.51260
23 18     1   26.000     3             3   245.62920
19 19     1   27.000     7             5   808.12920
25 ...      ...      ...      ...      ...
count=212 rows, elapsed_time=0.012332916259765625

```

4.2.4.14 head

```
RVD.head(n=5)
```

Print in the terminal the first RVD rows.

Parameters

- **n:** *<positive int>*
The number of rows to print.

Example

```

1 titanic.head()
3 #Output
   age  boat  body  cabin  embarked  fare  \\

```

```

5 0      None      1      None      None      S      26.00000  \\
1      None      10     None      E101     Q      12.35000  \\
7 2      None      10     None      None     S      16.10000  \\
3      None      11     None      None     S      33.00000  \\
9 4      None      13     None      None     Q       7.72080  \\
...      ...      ...      ...      ...      ...      ...  \\
11 ...      homedest                                     name  \\
0      New York, NY                                     Salomon, Mr. Abraham L  \\
13 1      Harrisburg, PA                                Keane, Miss. Nora A  \\
2      None      Thorneycroft, Mrs. Percival (Florence...  \\
15 3      London / Chicago, IL                          Leitch, Miss. Jessie Wills  \\
4      None      Riordan, Miss. Johanna Hannah          \\
17 ...      ...      ...      ...      ...      ...  \\
--More--
19 Name: titanic, Number of rows: 1309, Number of columns: 14

```

4.2.4.15 help

```
1 RVD.help()
```

Return information about the RVD.

Example

```

1 titanic.help()

3 #Output

5 #####
#      _____      #
7 # |  __ \ \ / /  __ \ #
# | |__ \ \ / / | | | #
9 # |  _/ \ \ / / | | | #
# | | \ \ / / | |__ | #
11 # |__ \ \ / / |_____/ #
#      #
13 #####
#      #
15 # Resilient Vertica Dataset #
#      #
17 #####

19 The RVD is a Python object which will keep in mind all the user modifications
in order to use an optimized SQL query.
--More--

```

4.2.4.16 hexbin

```
RVD.hexbin(columns, method="count", of=None, cmap='Blues', gridsize=10, color=None)
```

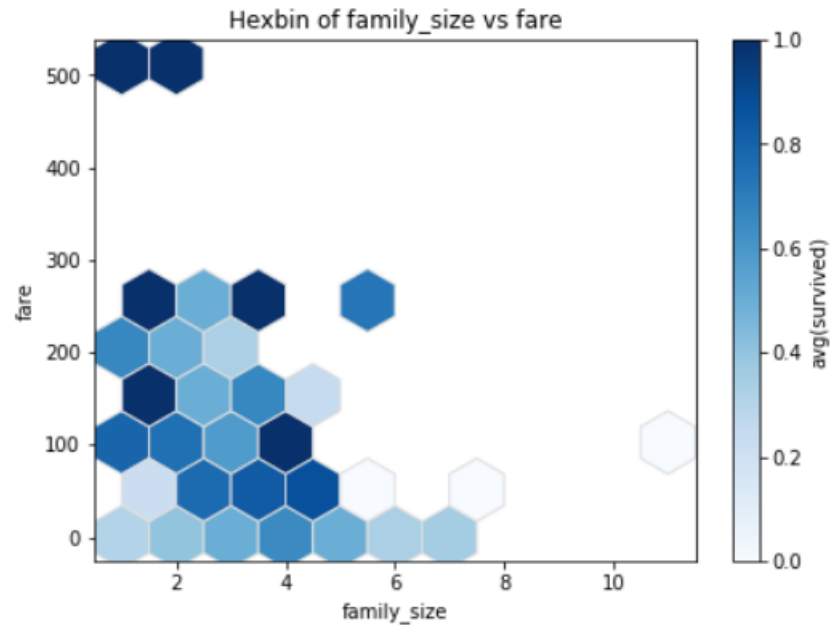
Draw the corresponding hexbin plot.

Parameters

- **columns:** *<list of str>*
The two columns used to draw the hexbin (first will be on the x-axis and the second in the y-axis)
- **method:** *<str>*, optional
count | density | avg | min | max | sum
count (default): count is used as aggregation
density: density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **cmap:** *<str>*, optional
Color Maps.
- **gridsize:** *<positive int>*, optional
Grid Size.
- **color:** *<str>*, optional
Hexbin outline color.

Example

```
titanic.hexbin(columns=["family_size", "fare"], method="avg", of="survived")
```



4.2.4.17 hist

```
RVD.hist(columns, method="density", of=None, max_cardinality=[6,6], h=[None,
None], color=None, limit_distinct_elements=200)
```

Draw the corresponding 2 variables histogram.

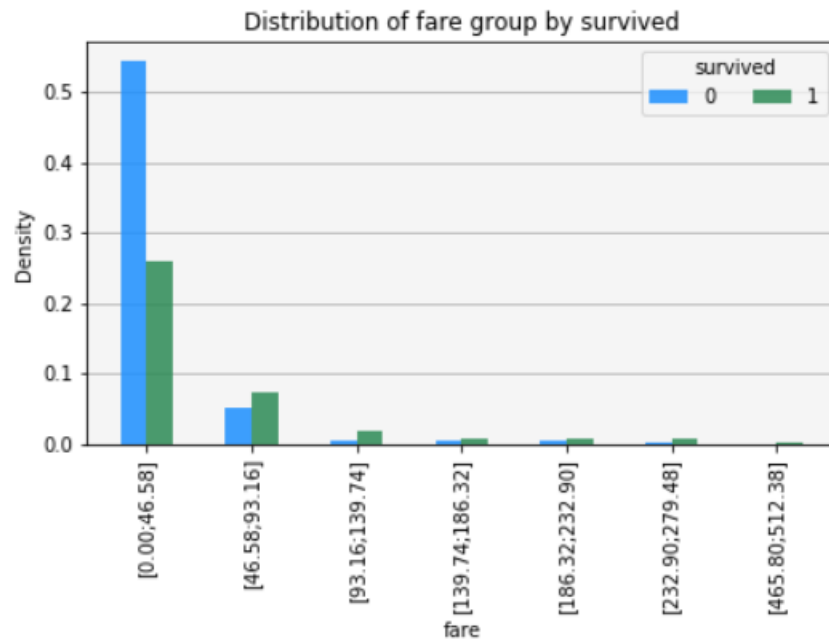
Parameters

- **columns:** *<list of str>*
The two columns used to draw the hist (first will be on the x-axis and the second in the y-axis)
- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<list of positive int>*, optional
The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<list of positive float>*, optional
The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **color:** *<list of str>*, optional
The color list for each category

- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.

Example

```
1 titanic.hist(columns=["fare", "survived"])
```



4.2.4.18 history

```
1 RVD.history()
```

Resume all the modifications made on the RVD.

Example

```
1 # Modifications
titanic["body"].drop_column()
3 titanic["cabin"].drop_column()
titanic["boat"].drop_column()
5 titanic["homedest"].drop_column()
titanic["sex"].rename('gender')
7 titanic["ticket"].drop_column()
titanic["gender"].label_encode()
9 titanic["embarked"].dropna()
titanic["embarked"].label_encode()
11 titanic["name"].drop_column()
```

```

titanic["fare"].dropna()
13 titanic["age"].fillna(method="avg",by=["pclass","gender"])

15 # History
titanic.history()

17 #Output
19 The RVD was modified many times:
    * {Tue Jan 30 18:10:48 2018} [Drop Column]: Column 'body' was deleted from
      the RVD.
21 * {Tue Jan 30 18:10:48 2018} [Drop Column]: Column 'cabin' was deleted from
      the RVD.
    * {Tue Jan 30 18:10:48 2018} [Drop Column]: Column 'boat' was deleted from
      the RVD.
23 * {Tue Jan 30 18:10:48 2018} [Drop Column]: Column 'homedest' was deleted
      from the RVD.
    * {Tue Jan 30 18:10:48 2018} [Drop Column]: Column 'sex' was deleted from the
      RVD.
25 * {Tue Jan 30 18:10:48 2018} [Rename]: The RVC 'sex' was renamed 'gender'.
    * {Tue Jan 30 18:10:49 2018} [Drop Column]: Column 'ticket' was deleted from
      the RVD.
27 * {Tue Jan 30 18:10:49 2018} [label Encode]: The RVC 'gender' was imputed
      with the 'label encoding'.
      female => 0
29      male => 1
      others => 2
31 * {Tue Jan 30 18:10:49 2018} [Dropna]: The 2 missing elements of column '
      embarked' were dropped from the RVD.
    * {Tue Jan 30 18:10:49 2018} [label Encode]: The RVC 'embarked' was imputed
      with the 'label encoding'.
33      Q => 0
      C => 1
35      S => 2
      others => 3
37 * {Tue Jan 30 18:10:49 2018} [Drop Column]: Column 'name' was deleted from
      the RVD.
    * {Tue Jan 30 18:10:49 2018} [Dropna]: The only missing element of column '
      fare' was dropped from the RVD.
39 * {Tue Jan 30 18:10:49 2018} [Fillna]: 263 missing values of the RVC 'age'
      were filled using the imputation 'coalesce({},avg({}) over (partition by
      pclass,gender))'.
```

4.2.4.19 missing

```
1 RVD.missing()
```

Print the array of all the RVD missing values.

Returns

An object named `column_matrix` containing all the missing information (they will be stored in the `data_columns` attribute).

Example

```
1 titanic.missing()

3 #Output

5      total    percent
6 body      1188      0.908
7 cabin      1014      0.775
8 boat        823      0.629
9 homedest     564      0.431
10 age        263      0.201
11 embarked     2      0.002
12 fare         1      0.001
13 name         0        0.0
14 ticket        0        0.0
15 pclass        0        0.0
16 sibsp         0        0.0
17 survived      0        0.0
18 parch         0        0.0
19 sex          0        0.0
```

4.2.4.20 multiple_hist

```
RVD.multiple_hist(columns, method="density", of=None, h=None, color=None)
```

Draw the corresponding numerical variables histograms in the same figure.

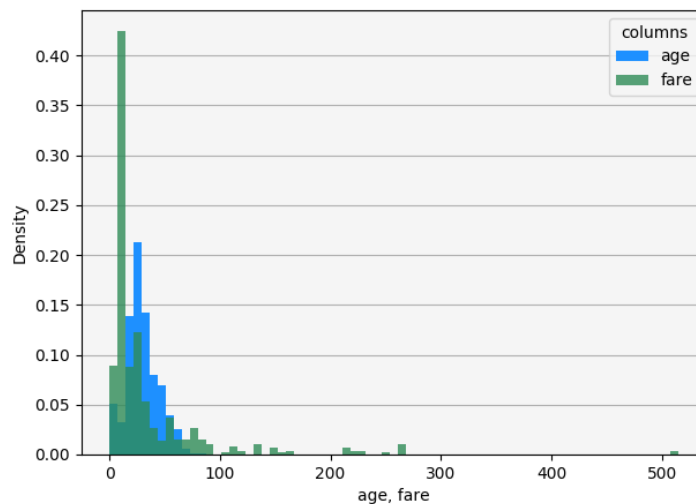
Parameters

- **columns:** *<list of str>*
The columns used to draw the histograms (they must be numerical)
The maximum number of columns is 5.
- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed

- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **h:** *<list of positive float>*, optional
The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **color:** *<list of str>*, optional
The color list for each column.
- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.

Example

```
titanic.multiple_hist(columns=["age", "fare"])
```



4.2.4.21 normalize

```
RVD.normalize(method="zscore", with_int=False)
```

Normalize the numerical columns of the RVD using the corresponding method.

Parameters

- **method:** *<str>*, optional
The method to be used: {zscore | robust_zscore | minmax}
- **with_int:** *<bool>*, optional
Include integer columns for the global normalization.

Example


```

1 titanic.normalize()

3 #Output
The RVC 'age' was successfully normalized.
5 The RVC 'fare' was successfully normalized.

```

4.2.4.22 pivot_table

```

1 RVD.pivot_table(columns, method="count", of=None, max_cardinality=[20,20], h=[
    None, None], cmap='Blues', limit_distinct_elements=1000)

```

Draw the corresponding pivot table.

Parameters

- **columns:** *<list of str>*
The two columns used to build the pivot table.
- **method:** *<str>*, optional
count | density | avg | min | max | sum
count (default): count is used as aggregation
density: density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed.
- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<list of positive int>*, optional
The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<list of positive float>*, optional
The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **cmap:** *<str>*, optional
Color Maps
- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.
- **show:** *<bool>*, optional
Draw the pivot table using matplotlib.
- **show:** *<bool>*, optional
Draw the pivot table using matplotlib.

Returns

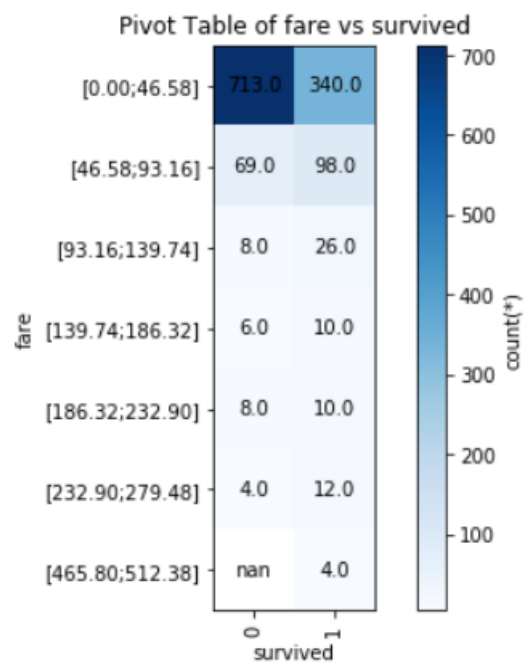
An object named column_matrix containing the pivot table (the information will be stored in the data_columns attribute).

Example

```

1 titanic.pivot_table(columns=["fare", "survived"])
3 #Output
4 fare/survived      0      1
5 [0.00;46.58]      713    340
6 [46.58;93.16]      69     98
7 [93.16;139.74]      8     26
8 [139.74;186.32]      6     10
9 [186.32;232.90]      8     10
10 [232.90;279.48]      4     12
11 [465.80;512.38]      4

```



4.2.4.23 save

```

1 RVD.save(name, columns=None, mode="view", affect=True)

```

Save the RVD by creating a new relation in the database.

- **name:** <str>
Name of the new relation.
- **columns:** <list of str>, optional
The columns used to build the relation. If it is not a list, all the columns will be considered in the creation.
- **mode:** <str>, optional
view (default) | table | temporary table

- **affect:** *<bool>*, optional
Affect the new RVD created thanks to the new input relation. In this case, it will be impossible to undo imputation or filtering.

Note

By saving, the efficiency will be maximal as the requests to the database will contain smaller queries. The input relation of the RVD will become this new relation.

Example

```
1 titanic.save(name="titanic_temp")
3 #Output
The RVD was successfully saved.
```

4.2.4.24 scatter

```
RVD.scatter(columns, h=None, max_cardinality=3, cat_priority=None, with_others=
    True, color=None, marker=["^", "o", "+", "*", "h", "x", "D", "1"]*10,
    max_nb_points=1000)
```

Draw the scatter plot of the considered columns. It is recommended to use the method 'scatter2D' or 'scatter3D' for respectively 2D and 3D scatter Plot (they have exactly the same parameters). This function will try to find the most adapted plot between 2D and 3D

Parameters

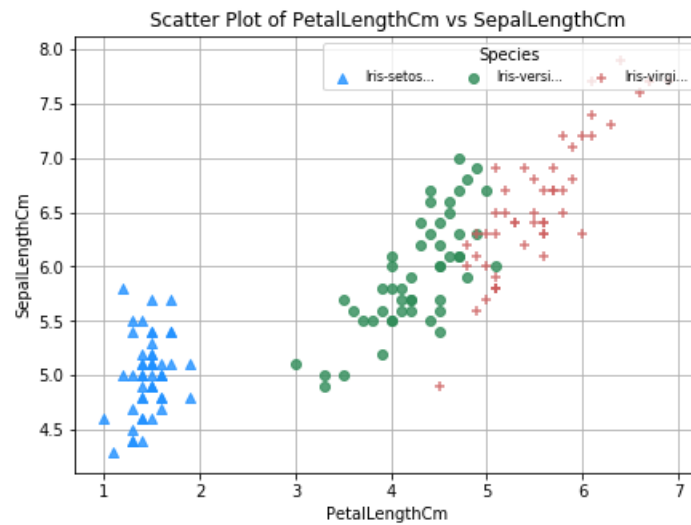
- **columns:** *<list of str>*
The two, three or four columns used to draw the scatter plot
- **h:** *<positive float>*, optional
The interval size of the categorical column (third or fourth position in "columns" depending on the case)
- **max_cardinality:** *<positive int>*, optional
The maximum cardinality of the categorical column (third or fourth position in "columns" depending on the case), all the other categories are merged to create the "others" category.
- **cat_priority:** *<list of str>*, optional
The list of the categories took into account during the computation.
- **with_others:** *<bool>*, optional
Include the "others" category.
- **color:** *<list of str>*, optional
The color list for each category.
- **marker:** *<list of str>*, optional
The list of categories markers.

- **max_nb_points:** *<positive int>*, optional

The maximum number of points in the scatter plot. The points are taken randomly from the table.

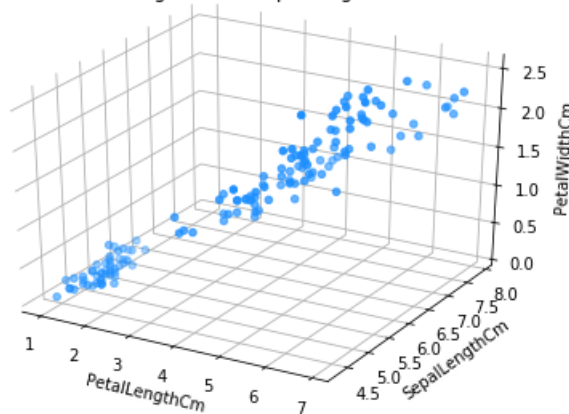
Examples

```
iris.scatter(columns=["PetalLengthCm", "SepalLengthCm", "Species"])
```

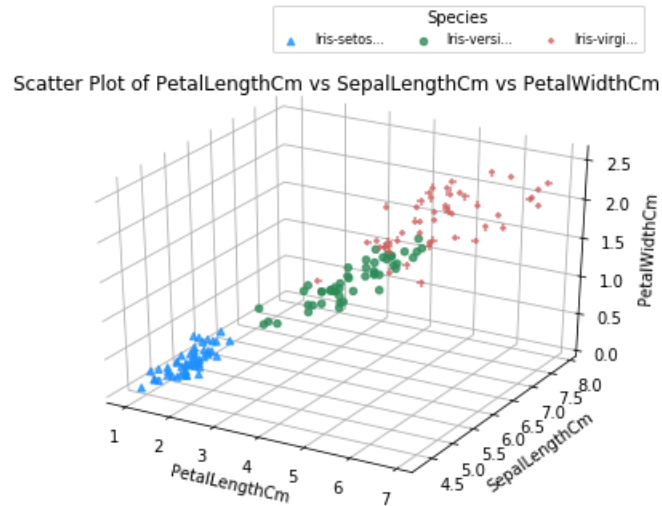


```
iris.scatter(columns=["PetalLengthCm", "SepalLengthCm", "PetalWidthCm"], mode="3D")
```

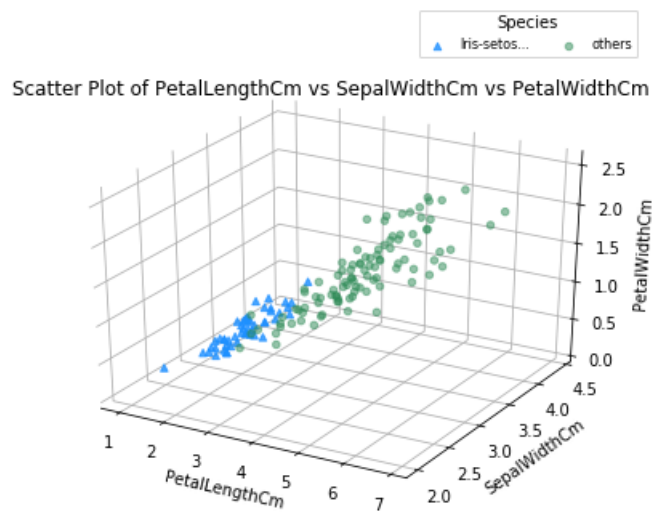
Scatter Plot of PetalLengthCm vs SepalLengthCm vs PetalWidthCm



```
iris.scatter(columns=["PetalLengthCm", "SepalLengthCm", "PetalWidthCm", "Species"])
```



```
iris.scatter(columns=["PetalLengthCm", "SepalWidthCm", "PetalWidthCm", "Species"],
             cat_priority=["Iris-setosa"])
```



4.2.4.25 scatter_matrix

```
RVD.scatter_matrix(columns=None, color=None)
```

Draw the scatter matrix of the corresponding columns.

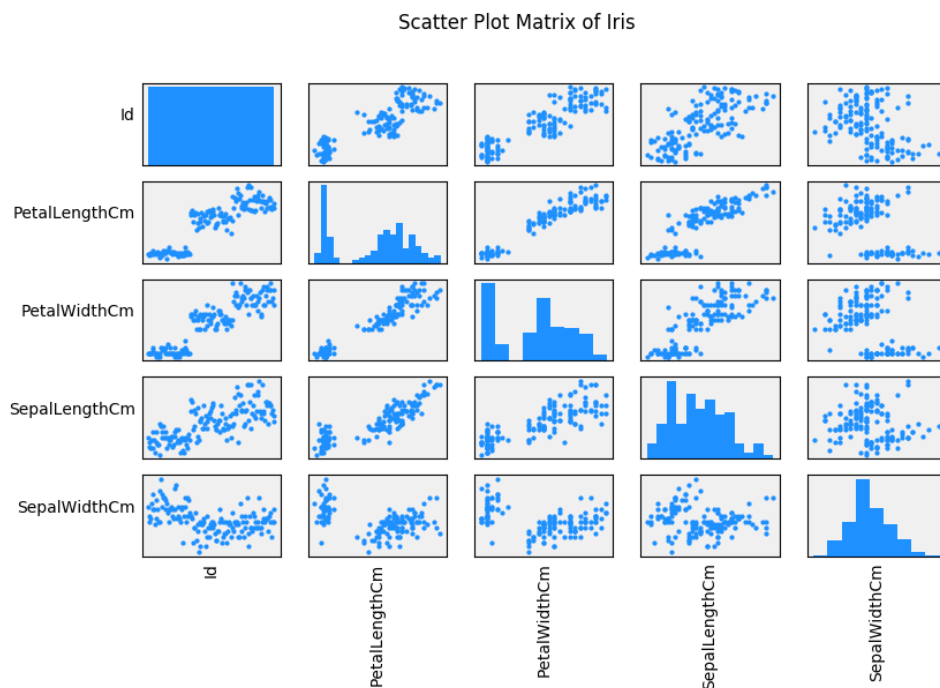
Parameters

- **columns:** *<list of str>*
The columns used to draw the scatter matrix.

- **color:** *<str>*, optional
The color of the scatter matrix.

Example

```
iris.scatter_matrix()
```



4.2.4.26 select

```
RVD.select(columns, order_by=None, asc=True, limit=100)
```

Group the aggregations by the corresponding columns.

Parameters

- **columns:** *<list of str>*
List of the columns to select.
- **order_by:** *<list of str>*, optional
List of all the columns to order with.
- **asc:** *<bool>*, optional
To order asc.
- **limit:** *<positive int>*, optional
The maximum number of elements to take into account.

Returns

An object named `column_matrix` containing all the selected information (they will be stored in the `data_columns` attribute).

Example

```

1 titanic.select(['pclass', 'age'], limit=20)
3 #Output
      pclass    age
5 0         1  None
6 1         2  None
7 2         3  None
8 3         2  None
9 4         3  None
10 5         3  None
11 6         3  None
12 7         3  None
13 8         3  None
14 9         3  None
15 10        3  None
16 11        3  None
17 12        1  None
18 13        3  None
19 14        2  None
20 15        3  None
21 16        3  None
22 17        3  None
23 18        3  None
24 19        3  None
25 ...      ...    ...
count=1309 rows, elapsed_time=0.006971836090087891

```

4.2.4.27 set_colors

```
RVD.set_colors(colors)
```

Replace the current colors used to plot the different charts for the new ones.

- **colors:** *<list of str>*
The list of colors.

4.2.4.28 set_cursor

```
1 RVD.set_cursor(cursor)
```

Replace the current cursor for a new one.

- **cursor:** *<object>*
The database cursor.

4.2.4.29 set_dsn

```
1 RVD.set_dsn(dsn)
```

Replace the current dsn for a new one.

- **dsn:** *<object>*
Vertica DSN.

Example

```
1 titanic.set_dsn("VerticaDSN")
```

4.2.4.30 set_figure_size

```
1 RVD.set_figure_size(figsize=(7,5))
```

Change the figure size for all the RVD chart.

- **figsize:** *<tuple>*, optional
Size of the figures.

Example

```
1 titanic.set_figure_size(figsize=(10,8))
```

4.2.4.31 set_legend_loc

```
1 RVD.set_legend_loc(bbox_to_anchor=None, ncol=None, loc=None)
```

Change the legend position for all the RVD chart.

- **bbox_to_anchor:** *<tuple>*, optional
Position of the legend following "loc".
- **ncol:** *<positive int>*, optional
Number of columns of the legend.
- **loc:** *<str>*, optional
Location of the legend.

Example

```
1 titanic.set_legend_loc(bbox_to_anchor=(0.5,1), ncol=2, loc="upper right")
```

4.2.4.32 set_limit

```
1 RVD.set_limit(limit=None)
```

Change the RVD limit.

- **limit:** *<positive int>*, optional
New RVD limit value.

Example

```
1 titanic.set_limit(limit=100)
```

4.2.4.33 set_offset

```
1 RVD.set_offset(offset=0)
```

Change the RVD offset.

- **offset:** *<positive int>*, optional
New RVD offset value.

Example

```
1 titanic.set_offset(offset=100)
```

4.2.4.34 sql_on_off

```
1 RVD.sql_on_off(reindent=False)
```

Print all the sql queries used by the RVD in the terminal. If it is already enable, it will turn it off.

Parameters

- **reindent:** <bool>, optional
Use sqlparse available in github to indent all the queries.

Example

```
1 titanic.sql_on_off(reindent=True)
2 titanic.describe()
3
4 #Output
5 -----
6
7 select summarize_numcol(age,body,fare,parch,pclass,sibsp,survived) over ()
8 from
9     (select age as age,
10         boat as boat,
11         body as body,
12         cabin as cabin,
13         embarked as embarked,
14         fare as fare,
15         homedest as homedest,
16         name as name,
17         parch as parch,
18         pclass as pclass,
19         sex as sex,
20         sibsp as sibsp,
21         survived as survived,
22         ticket as ticket
23     from titanic) x
24 -----
25
26 select count(distinct age),
27        count(distinct body),
28        count(distinct fare),
29        count(distinct parch),
30        count(distinct pclass),
31        count(distinct sibsp),
32        count(distinct survived)
```

```

33 from
34     (select age as age,
35            boat as boat,
36            body as body,
37            cabin as cabin,
38            embarked as embarked,
39            fare as fare,
40            homedest as homedest,
41            name as name,
42            parch as parch,
43            pclass as pclass,
44            sex as sex,
45            sibsp as sibsp,
46            survived as survived,
47            ticket as ticket
48     from titanic) x
49
--More--

```

4.2.4.35 stacked_bar

```

1 RVD.stacked_bar(columns, method="density", of=None, max_cardinality=[6,6], h=[
    None, None], color=None, limit_distinct_elements=200)

```

Draw the corresponding 2 variables stacked bar chart.

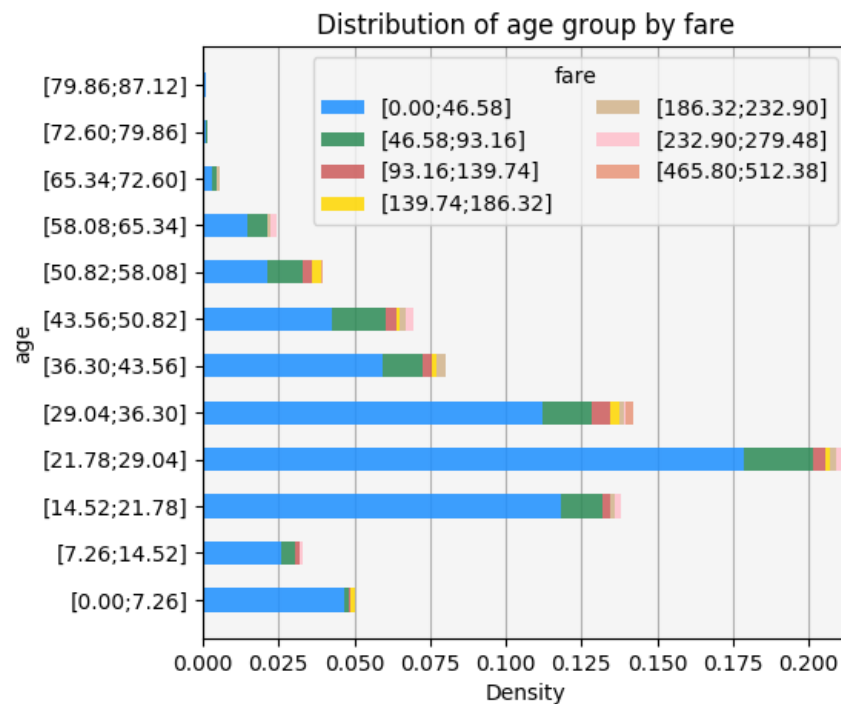
Parameters

- columns:** *<list of str>*
 The two columns used to draw the stacked bar (first will be on the x-axis and the second in the y-axis)
- method:** *<str>*, optional
 count | density | avg | min | max | sum
 count: count is used as aggregation
 density (default): density is used as aggregation
 avg | min | max | sum: these aggregations are used only if "of" is informed
- of:** *<str>*, optional
 The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- max_cardinality:** *<list of positive int>*, optional
 The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- h:** *<list of positive float>*, optional
 The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.

- **color:** *<list of str>*, optional
The color list for each category
- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.

Example

```
titanic.stacked_bar(columns=["age", "fare"])
```



4.2.4.36 stacked_hist

```
RVD.stacked_hist(columns, method="density", of=None, max_cardinality=[6,6], h=[None, None], color=None, limit_distinct_elements=200)
```

Draw the corresponding 2 variables stacked histogram.

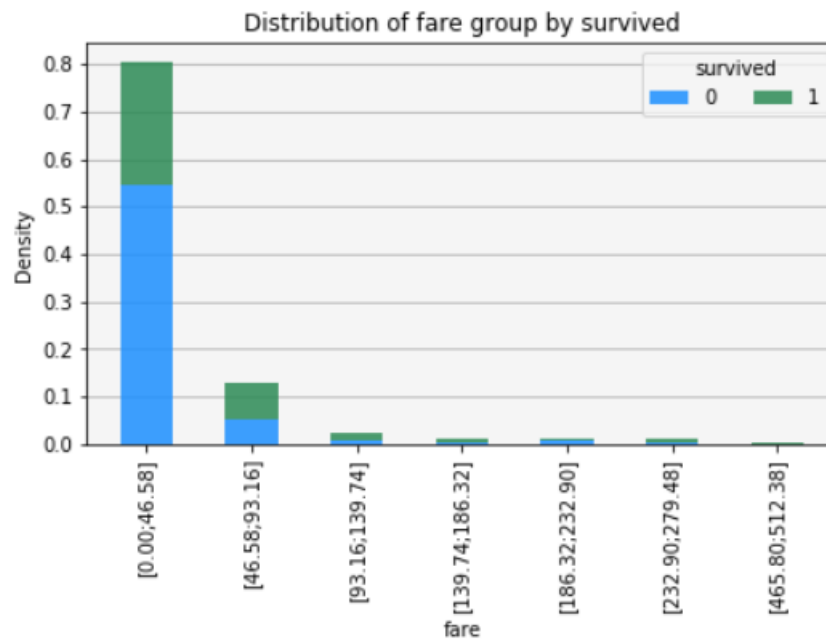
Parameters

- **columns:** *<list of str>*
The two columns used to draw the stacked hist (first will be on the x-axis and the second in the y-axis)
- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed

- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<list of positive int>*, optional
The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<list of positive float>*, optional
The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **color:** *<list of str>*, optional
The color list for each category
- **limit_distinct_elements:** *<positive int>*, optional
The maximum number of distinct elements. The other categories will be ignored.

Example

```
titanic.stacked_hist(columns=["fare", "survived"])
```



4.2.4.37 time_on_off

```
RVD.time_on_off()
```

Print all the queries elapsed time used by the RVD in the terminal. If it is already enable, it will turn it off.

Example

```

1 titanic.time_on_off()
  titanic.describe()
3
  #Output
5 Elapsed Time: 0.013292789459228516
  -----
7 Elapsed Time: 0.04647397994995117
  -----

```

4.2.4.38 train_test_split

```

RVD.train_test_split(split=None, test_name=None, train_name=None, columns="*",
                      test_size=0.33, mode="view")

```

Separate the data into two relations using a specific column.

Parameters

- **split:** <str>, optional
Name of the column used to split the data. If it doesn't exist a table with one random float column will be created and used to split the data.
- **test_name:** <str>, optional
Name of the test set.
- **train_name:** <str>, optional
Name of the training set.
- **columns:** <list of str>, optional
List of the columns to be used.
- **test_size:** <float in [0,1]>, optional
Size of the test set.
- **mode:** <str>, optional
view | table | temporary table
The mode is used to create the new relations.

Warning

If no name is given to test or train, the function will always generate default names (test_(input_relation)_0(test_size) and train_(input_relation)_0(test_size)). If these relations already exist, it will drop them. Besides, if the column "split" does not exist, a new table having only one column of random numbers (between 0 and 1) and the same number of rows than the RVD will be created. If the table already exists (default name = random_vpython_table_(input_relation)), it will use it for computation without creating a new one. This table will be used to separate the data (using a natural join on the row number). Do not delete this column if you are using the view mode !

Example

```

1 titanic.train_test_split()

3 #Output
The random table random_vpython_table_titanic was successfully created
5 The views test_titanic033 and train_titanic067 were successfully created.

7 (      age      boat      body      cabin      embarked      fare      \\
0      None        1      None      None        S        26.00000    \\
9      1      None       10      None      E101        Q        12.35000    \\
10     2      None       11      None      None        S        33.00000    \\
11     3      None       13      None      None        Q         7.72080    \\
12     4      None       13      None      None        Q         7.73330    \\
13     5      None       13      None      None        Q         7.75000    \\
14     6      None       13      None      None        Q         7.78750    \\
15     7      None       13      None      None        Q         7.82920    \\
16     8      None       13      None      None        Q         7.87920    \\
17     9      None       13      None      None        S         8.11250    \\
18    10      None       13      None      None        S        56.49580    \\
19    11      None       14      None      None        C        30.69580    \\
20    12      None       15      None      None        S         7.05000    \\
21    13      None       16      None      None        Q         7.73330    \\
22    14      None       16      None      None        Q         7.75000    \\
23    15      None       16      None      None        Q         7.75000    \\
24    16      None       16      None      None        Q         7.87920    \\
25    17      None       16      None      None        Q        15.50000    \\
26    18      None       16      None      None        Q        15.50000    \\
27    19      None       16      None      None        Q        15.50000    \\
28    20      None       16      None      None        Q        23.25000    \\
29    21      None       16      None      None        Q        23.25000    \\
30    22      None       16      None      None        Q        23.25000    \\
31    23      None       16      None      None        S        16.10000    \\
32    24      None        3      None      C106        S        30.50000    \\
33    25      None        4      None      None        C       110.88330    \\
34    26      None        5      None      None        C        27.72080    \\
35    27      None        5      None      None        S       133.65000    \\
36    28      None       5 7      None      C126        S         52.00000    \\
37    29      None        6      None      B78        C       146.52080    \\
38    ...      ...      ...      ...      ...      ...      ...    \\
39 --More--

```

4.2.4.39 undo_all_filters

```
1 RVD.undo_all_filters()
```

Undo all the filters.

4.2.4.40 undo_filter

```
1 RVD.undo_filter()
```

Undo the last filter.

4.2.4.41 version

```
1 RVD.version()
```

Return the Vertica DB version and information about the RVD adaptation for this version.

Example

```
1 titanic.version()
3 #Output
5 #####
6 #
7 #               _ _
8 #               | | | |
9 #      _ _      _ _ _ _ _ _ | | | | _ _ _ _ _ _
10 #     \ \ / / ' _ \ | | | _ _ | ' _ \ / _ \ | ' _ \
11 #      \ V / | |_) | | | | | | | | | ( _ ) | | | |
12 #       \_/ | .__ / \__, | \__ \ | | \__ \ / | | | |
13 #           | |      __/ |
14 #           | |      |___/
15 #####
16 #
17 # Author: Badr Ouali, Data scientist at Vertica
18 #
19 # You are currently using Vertica Analytic Database v9.0.1-0
20 #
21 # You have a perfectly adapted version for using RVD and Vertica ML
22 #
23 # For more information about the RVD you can use the help() method
   (9, 0)
```

4.3 Resilient Vertica Column (RVC)

4.3.1 attributes

When the RVD is created, it will create as many RVC as there are columns in the input relation. The RVC has only 3 attributes.

- **parent**: the RVC parent (must be a RVD).
- **alias**: the alias of the column.
- **transformations**: list of all the transformations since the beginning with their types and categories.

4.3.2 methods

4.3.2.1 abs

```
1 RVC.abs()
```

Apply the abs function to the RVC.

4.3.2.2 acos

```
1 RVC.acos()
```

Apply the acos function to the RVC.

4.3.2.3 add

```
1 RVC.add(x)
```

Add a number to the RVC.

Parameters

- **x**: *<positive float or int>*
Value to add.

4.3.2.4 asin

```
1 RVC.asin()
```

Apply the asin function to the RVC.

4.3.2.5 atan

```
1 RVC.atan()
```

Apply the atan function to the RVC.

4.3.2.6 bar

```
RVC.bar(method="density", of=None, max_cardinality=6, bins=None, h=None, color=None)
```

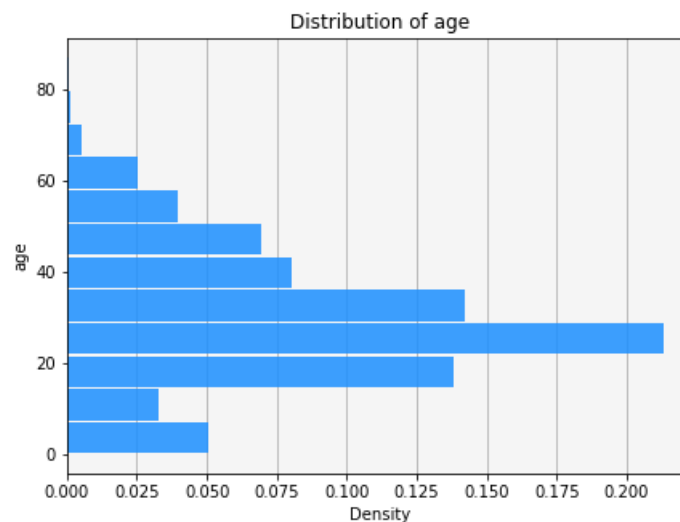
Draw the RVC bar.

Parameters

- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<positive int>*, optional
The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **bins:** *<positive int>*, optional
The number of the histogram bins.
- **h:** *<positive float>*, optional
The interval size of the column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **color:** *<list of str>*, optional
The histogram color.

Example

```
titanic["age"].bar()
```



4.3.2.7 boxplot

```
RVC.boxplot(by=None, h=None, max_cardinality=8, cat_priority=None)
```

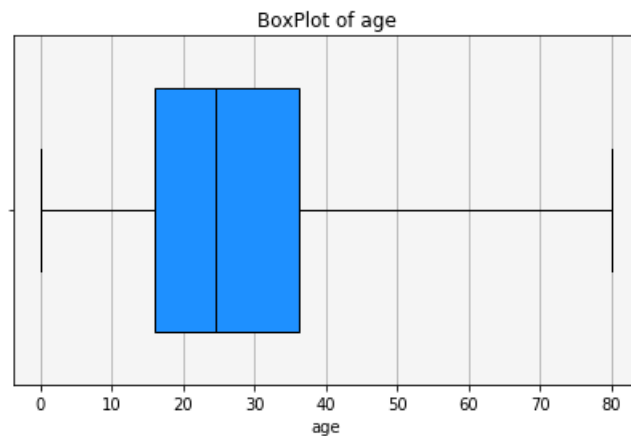
Draw the RVC boxplot.

Parameters

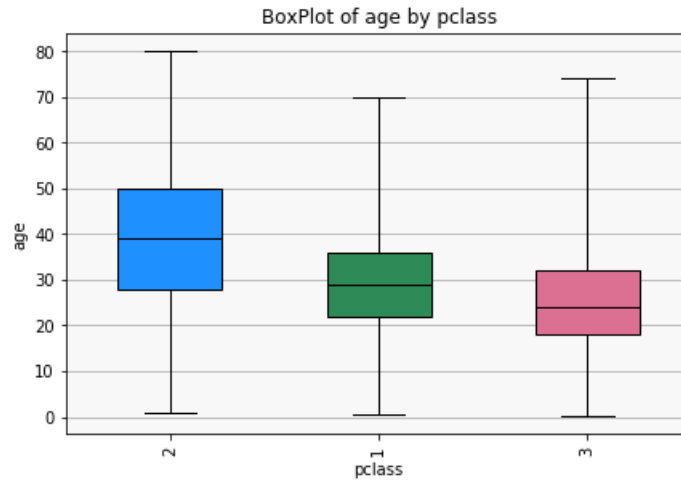
- **by:** *<str>*, optional
The group by column. It is used to separate the column per categories.
- **max_cardinality:** *<positive int>*, optional
The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **h:** *<positive float>*, optional
The interval size of the column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically.
- **cat_priority:** *<list of str>*, optional
The principal categories to show.

Example

```
titanic["age"].boxplot()
```



```
titanic["age"].boxplot(by="pclass")
```



4.3.2.8 cardinality

```
1 RVC.cardinality()
```

Returns

Returns the RVC cardinality.

Example

```
1 titanic["pclass"].cardinality()
3
3 #Output
3
```

4.3.2.9 category

```
RVC.category()
```

Returns

Returns the RVC current category.

Example

```
1 titanic["pclass"].category()
3
3 #Output
3 int
```

4.3.2.10 convert_to_num

```
RVC.convert_to_num()
```

Try to convert a categorical RVC to a numerical one.

4.3.2.11 count

```
1 RVC.count()
```

Returns

Returns the RVC count (number of non-missing values).

Example

```
1 titanic["age"].count()  
3 #Output  
1046
```

4.3.2.12 cos

```
RVC.cos()
```

Apply the cos function to the RVC.

4.3.2.13 cosh

```
1 RVC.cosh()
```

Apply the cosh function to the RVC.

4.3.2.14 cot

```
1 RVC.cot()
```

Apply the cot function to the RVC.

4.3.2.15 date_part

```
1 RVC.date_part (field="month")
```

Extract the date part from the RVC.

Parameters

- **field:** <str>
The field must be in {century | day | decade | doq | dow | doy | epoch | hour | isodow | isoweek | isoyear | microseconds | millenium | milliseconds | minute | month | quarter | second | timezone | timezone_hour | timezone_minute | week | year}

Example

```
1 expedia["date_time"].date_part (field="month")
expedia["date_time"]
3
#Output
5 0      1.0
1      1.0
7 2      1.0
3      1.0
9 4      1.0
5      1.0
11 --More--
...      ...
13 Name: date_time, dtype: int
```

4.3.2.16 decode

```
1 RVC.decode(categories, values, others=None)
```

Apply the decode function to the RVC.

Parameters

- **categories:** <list of str>
The different categories to compare.
- **values:** <list of str/int/float>
The new categories values.
- **others:** <str>, optional
The value in the other case.

Example

```

1 titanic["name"].regex_substr(' ([A-Za-z]+)\.')
titanic["name"].decode([" Mr.", " Miss.", " Mrs.", " Ms.", " Mlle.", " Lady.", " Mme",
    ., " Sir.", " Rev.", " Dr.", " Col.", " Major.", " Capt."], ["Mr", "Miss", "Mrs",
    "Miss", "Miss", "Mrs", "Mrs", "Mr", "Rev", "Dr", "Army", "Army", "Army"], "Rare")
3 titanic["name"].describe(max_cardinality=8)

5 #Output
758      Mr
7 264    Miss
199     Mrs
9 65     Rare
8      Rev
11 8      Dr
7     Army
13 Name: name, dtype: varchar(6)

```

4.3.2.17 degrees

```
1 RVC.degrees()
```

Apply the degrees function to the RVC.

4.3.2.18 density

```
1 RVC.density(a=None, kernel="gaussian", smooth=200, color=None)
```

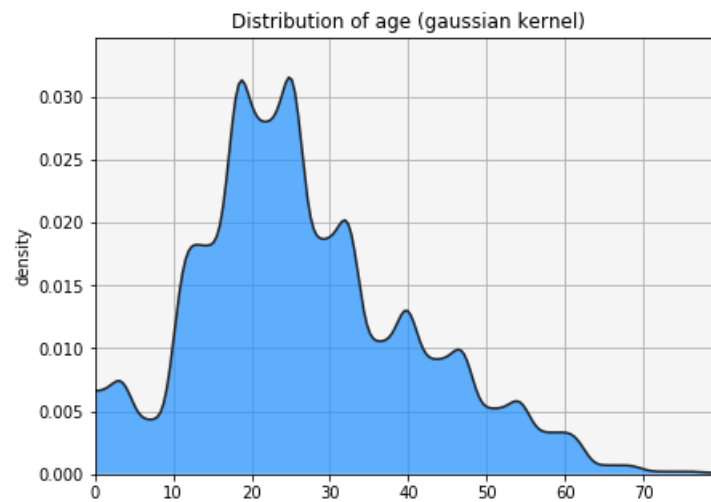
Draw the RVC density plot.

Parameters

- **a:** *<float>*, optional
The kernel window. If it is not informed, an optimal one is computed.
- **kernel:** *<str>*, optional
gaussian (default) | logistic | sigmoid | silverman
The Kernel used for the plot.
- **smooth:** *<positive int>*, optional
The number of points used for the smoothing.
- **color:** *<str>*, optional
The density plot color.

Example

```
titanic["age"].density()
```



4.3.2.19 describe

```
RVC.describe(mode="auto", max_cardinality=6)
```

Summarize the RVC with mathematical information.

Parameters

- **mode:** *<str>*, optional
 auto | categorical | numerical
 auto (default): This mode is used to detect the correct category.
 numerical: This mode is used to print numerical information if it is possible.
 categorical: This mode is used to only print the categorical variables information (text or *cardinality* \leq *max_cardinality*).
 date: This mode is used to only print the date variables information
- **max_cardinality:** *<bool>*, optional
 The maximum cardinality of the column. Under this number the column is automatically considered as categorical.

Returns

An object named `column_matrix` containing all the summarized information (they will be stored in the `data_column` attribute).

Note

The mathematical information are different depending on the data type and if they are categorical.

Example


```
1 titanic["age"].describe()

3 #Output
count                1046
5 mean                29.881137667304
std                 14.4134932112713
7 min                 0.17
25%                 21.0
9 50%                 28.0
75%                 39.0
11 max                80.0
cardinality           98
13 Name: age, dtype: numeric(6,3)

15 titanic["pclass"].describe()

17 #Output
709  3
19 323  1
277  2
21 Name: pclass, dtype: int
```

4.3.2.20 distinct

```
1 RVC.distinct()
```

Returns

Returns the RVC list of distinct elements.

Example

```
1 titanic["pclass"].distinct()

3 #Output
[1, 2, 3]
```

4.3.2.21 div

```
RVC.div(x)
```

Div the RVC by a number.

Parameters

- **x:** *<float or int>*
Value to div with.

4.3.2.22 donut

```
RVC.donut(method="density", of=None, max_cardinality=6, h=None, colors=[
    'dodgerblue', 'seagreen', 'indianred', 'gold', 'tan', 'pink', 'darksalmon',
    'lightskyblue', 'lightgreen', 'palevioletred', 'coral'] * 10)
```

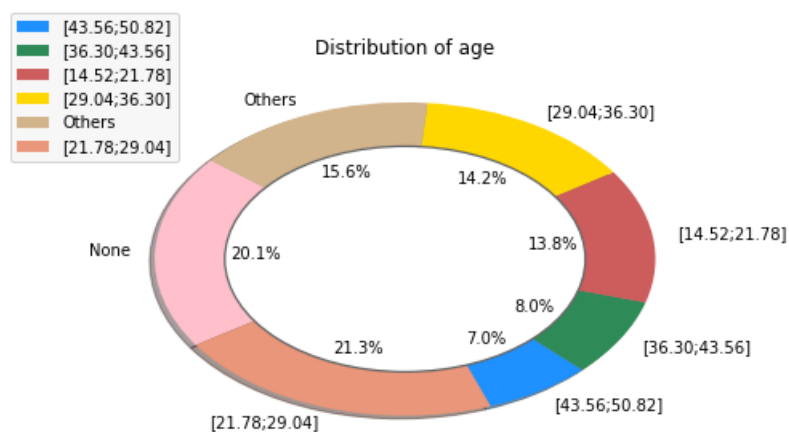
Draw the RVC donut chart.

Parameters

- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<positive int>*, optional
The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **h:** *<positive float>*, optional
The interval size of the column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **colors:** *<list of str>*, optional
The donut chart colors.

Example

```
titanic["age"].donut()
```



4.3.2.23 drop_column

```
1 RVC.drop_column()
```

Drop the RVC from the RVD.

Example

```
1 titanic["pclass"].drop_column()
3 #Output
RVC 'pclass' deleted from the RVD.
5
titanic["pclass"]
7
#Output
9 Error: 'RVD' object has no attribute 'pclass'
```

4.3.2.24 dropna

```
1 RVC.dropna()
```

Drop the RVC missing values.

Example

```
1 titanic["pclass"].dropna()
3 #Output
263 elements were dropped
```

4.3.2.25 dtype

```
RVC.dtype()
```

Print the RVC data type.

Example

```
1 titanic["pclass"].dtype()
3 #Output
col    numeric(6,3)
5 dtype: object
```

4.3.2.26 duplicate

```
1 RVC.duplicate(name=None)
```

Duplicate the RVC.

Parameters

- **name:** <str>, optional
Duplicate RVC name.

Example

```
1 titanic["age"].duplicate()
3 #Output
  Duplication age4779 added to the RVD.
5 'age4779'
```

4.3.2.27 enum

```
1 RVC.enum(h=None)
```

Try to convert the RVC to enum.

Parameters

- **h:** <positive float>, optional
The interval size of the column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically.

Example

```
1 titanic["fare"].enum()
  print(titanic["fare"])
3
  #Output
5 0      [0.00;46.58]
  1      [0.00;46.58]
  2      [0.00;46.58]
  3      [0.00;46.58]
  4      [0.00;46.58]
  5      [0.00;46.58]
11 --More--
  ...
13 Name: fare, dtype: varchar
```

4.3.2.28 exp

```
1 RVC.exp()
```

Apply the exp function to the RVC.

4.3.2.29 fillna

```
1 RVC.fillna(val=None, method=None, by=[], compute_before=True)
```

Impute the RVC following the corresponding method.

Parameters

- **val:** <str>, optional
The value used for the imputation.
- **method:** <str>, optional
mean | median | lead | lag
The method used for the imputation.
- **by:** <list of str>, optional
List of the group by columns.
- **compute_before:** <bool>, optional
Compute all the needed statistical information before the imputation.

Example

```
1 titanic["age"].fillna(method="avg", by=["pclass", "sex"])
3 #Output
263 elements were filled
```

4.3.2.30 final_transformation

```
RVC.final_transformation()
```

Return the RVC final transformation (if every transformation is applied).

Example

```
1 titanic["embarked"].label_encode()
2 titanic["embarked"].final_transformation()
3 #Output
5 decode(embarked, 'Q', 0, 'C', 1, 'S', 2, NULL)
```

4.3.2.31 floor

```
1 RVC.floor()
```

Apply the floor function to the RVC.

4.3.2.32 head

```
1 RVC.head(n=5)
```

Print in the terminal the first RVC rows.

Parameters

- **n:** *<positive int>*
The number of rows to print

Example

```
1 titanic["age"].head()
3 #Output
0      None
5 1      None
2      None
7 3      None
4      None
9 ...      ...
Name: age, dtype: numeric(6,3)
```

4.3.2.33 hist

```
RVC.hist(method="density", of=None, max_cardinality=6, bins=None, h=None,
         color=None)
```

Draw the RVC histogram.

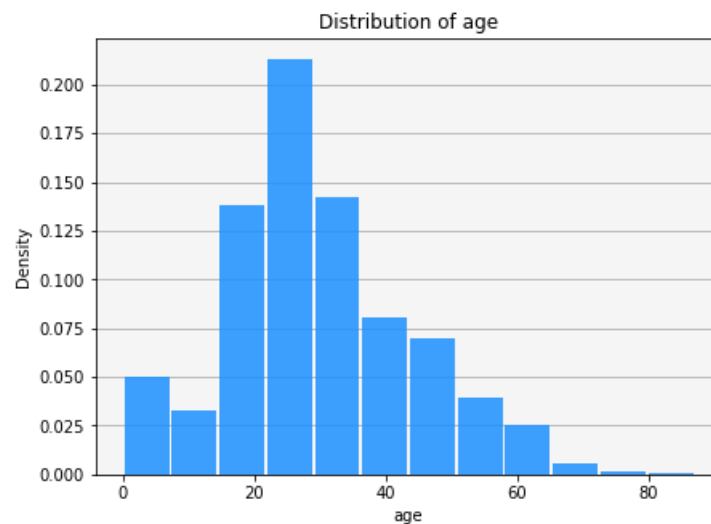
Parameters

- **method:** *<str>*, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed

- **of:** *<str>*, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}
- **max_cardinality:** *<positive int>*, optional
The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **bins:** *<positive int>*, optional
The number of the histogram bins.
- **h:** *<positive float>*, optional
The interval size of the column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **color:** *<list of str>*, optional
The histogram color.

Example

```
titanic["age"].hist()
```



4.3.2.34 label_encode

```
RVC.label_encode(cat_priority=None, values=None, others=None, order_by="count",
                  , force_encoding=False, show=True)
```

Use the label encoding to encode the RVC.

Parameters

- **cat_priority:** *<list of str>*
The different categories to encode.

- **values:** *<list of float/int>*
The new categories values.
- **others:** *<str>*, optional
The value in the other case.
- **order_by:** *<str>*, optional
Order the encoding by count.
- **force_encoding:** *<bool>*, optional
Allow the encoding even for numerical variables.
- **show:** *<bool>*, optional
Print all the label encoding in the terminal.

Example

```

1 titanic["embarked"].label_encode()

3 #Output
embarked      encoding
5 Q              0
  C              1
  S              2
7
The label encoding was successfully done.

```

4.3.2.35 log

```
RVC.log()
```

Apply the log function to the RVC.

4.3.2.36 max

```
1 RVC.max()
```

Returns

Returns the RVC maximum.

Example

```

1 titanic["age"].max()

3 #Output
80.0

```


4.3.2.37 mean

```
RVC.mean()
```

Returns

Returns the RVC average.

Example

```
1 titanic["age"].mean()
3 #Output
29.881137667304
```

4.3.2.38 mean_encode

```
RVC.mean_encode(response_column)
```

Apply a mean encoding using a specific response column.

Parameters

- **response_column:** <str>
the response column (must be a RVC in the main RVD)

Example

```
1 titanic["embarked"].mean_encode("survived")
3 The mean encoding was successfully done.
5 titanic["embarked"].describe()
#Output
7
      value
0.332603938730853    914
9 0.555555555555556    270
0.357723577235772    123
11 1                2
Name: embarked, dtype: int
```

4.3.2.39 median

```
RVC.median()
```

Returns

Returns the RVC median.

Example

```
1 titanic["age"].median()  
  
3 #Output  
28.0
```

4.3.2.40 min

```
RVC.min()
```

Returns

Returns the RVC minimum.

Example

```
1 titanic["age"].min()  
  
3 #Output  
0.17
```

4.3.2.41 mod

```
RVC.mod(n)
```

Use the mod function on the RVC.

Parameters

- **n:** *<int>*
Value of the mod.

4.3.2.42 mult

```
1 RVC.mult(x)
```

Mult the RVC by a number.

Parameters

- **x:** *<float or int>*
Value to mult with.

4.3.2.43 normalize

```
1 RVC.normalize(method="zscore", compute_before=True)
```

Normalize the RVC.

Parameters

- **compute_before:** *<bool>*, optional
Compute all the needed statistical information before the normalization.
- **method:** *<str>*, optional
zscore (default) | robust_zscore | minmax
The method used for the normalization.

Example

```
1 titanic["fare"].normalize()
  print(titanic["fare"])
3
#Output
5 The RVC 'fare' was successfully normalized.
  0      -0.14095183530676586
  1      -0.4046757769067325
  2      -0.33222414459904936
  3      -0.005708788332423966
  4      -0.494113935914393
11 5      -0.4938724304733674
  --More--
13 ...
  Name: fare, dtype: numeric(10,5)
```

4.3.2.44 one_hot_encoder

```
RVC.one_hot_encoder()
```

Apply a one hot encoder to encode the RVC.

Example

```
1 titanic["pclass"].one_hot_encoder()
3
#Output
  3 new features: pclass_1, pclass_2, pclass_3
5
  print(titanic["pclass_1"])
```

```

7 |
9 | #Output
11| 0      1
13| 1      0
15| 2      0
17| 3      0
19| 4      0
21| 5      0
23| 6      0
   | 7      0
   | 8      0
   | 9      0
   | 10     0
   | 11     0
   | 12     1
   | 13     0
   | 14     0
   | --More--

```

4.3.2.45 outliers

```
RVC.outliers(max_number=10, threshold=3)
```

Detect the RVC outliers using the corresponding threshold.

Parameters

- **max_number:** *<positive int>*, optional
The maximum of detected outliers.
- **threshold:** *<float>*, optional
The outliers threshold

Example

```

1 | titanic["fare"].outliers()
3 | #Output
   |
   |      fare      normalized_fare
5 | 0      512.3292      9.25513999906382
   | 1      512.3292      9.25513999906382
   | 2      512.3292      9.25513999906382
   | 3      512.3292      9.25513999906382
   | 4       263.0      4.43799132653881
   | 5       263.0      4.43799132653881
11| 6       263.0      4.43799132653881

```

```

7          263.0      4.43799132653881
13 8          263.0      4.43799132653881
9          263.0      4.43799132653881
15 ...          ...          ...
Total of 10 outliers detected.

```

4.3.2.46 percentile_cont

```
RVC.percentile_cont(x)
```

Returns

The corresponding percentile cont.

Parameters

- **x:** <float in [0,1]>
Percentile Cont.

Example

```

1 titanic["age"].percentile_cont(0.25)
3 #Output
21.0

```

4.3.2.47 pie

```
RVC.pie(method="density", of=None, max_cardinality=6, h=None, colors=['
dodgerblue', 'seagreen', 'indianred', 'gold', 'tan', 'pink', 'darksalmon', '
lightskyblue', 'lightgreen', 'palevioletred', 'coral']*10)
```

Draw the RVC pie chart.

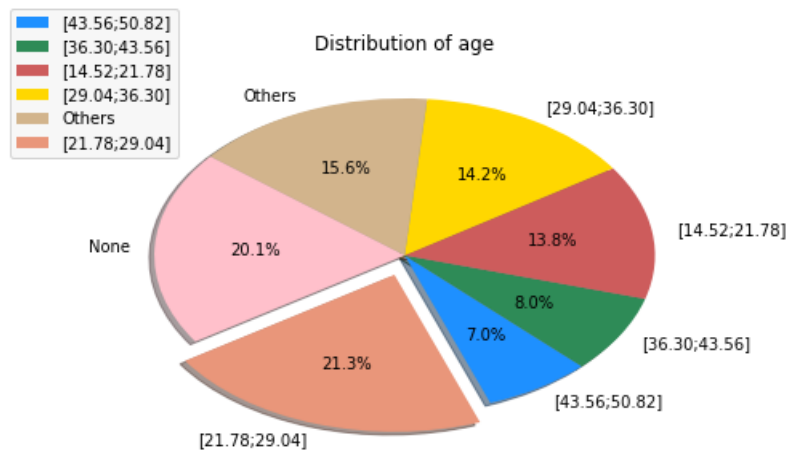
Parameters

- **method:** <str>, optional
count | density | avg | min | max | sum
count: count is used as aggregation
density (default): density is used as aggregation
avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** <str>, optional
The column used to compute the aggregation. This variable is used only if "method" in {avg | min | max | sum}

- **max_cardinality:** *<positive int>*, optional
The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **h:** *<positive float>*, optional
The interval size of the column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically. If the column is a date, h represents the interval size in seconds.
- **colors:** *<list of str>*, optional
The pie chart colors.

Example

```
1 titanic["age"].pie()
```



4.3.2.48 pow

```
1 RVC.pow(n)
```

Use the pow function on the RVC.

Parameters

- **n:** *<float>*
Value of the pow.

4.3.2.49 radians

```
1 RVC.radians()
```

Apply the radians function to the RVC.

4.3.2.50 regexp_substr

```
1 RVC.regexp_substr(expression)
```

Extract regular expression from each RVC raw.

Parameters

- **expression:** <str>
The Regular Expression.

Example

```
1 titanic["name"].regexp_substr(' ([A-Za-z]+)\.')
```

```
2 titanic["name"].describe()
```

```
3
```

```
4 #Output
```

```
5 757           Mr.
```

```
6 260           Miss.
```

```
7 197           Mrs.
```

```
8 61            Master.
```

```
9 14            Others
```

```
10 8             Dr.
```

```
11 8             Rev.
```

```
12 cardinality    18
```

```
13 Name: name, dtype: varchar(164)
```

4.3.2.51 rename

```
1 RVC.rename(new_name)
```

Rename the RVC.

Parameters

- **new_name:** <str>
New RVC name.

Example

```
1 titanic["sex"].rename(new_name="gender")
```

4.3.2.52 round

```
1 RVC.round(n)
```

Rounds the RVC.

Parameters

- **n:** *<int>*
Value for the round.

4.3.2.53 sign

```
1 RVC.sign()
```

Apply the sign function to the RVC.

4.3.2.54 sin

```
1 RVC.sin()
```

Apply the sin function to the RVC.

4.3.2.55 sinh

```
1 RVC.sinh()
```

Apply the sinh function to the RVC.

4.3.2.56 sqrt

```
1 RVC.sqrt()
```

Apply the sqrt function to the RVC.

4.3.2.57 std

```
1 RVC.std()
```

Returns

Returns the RVC standard deviation.

Example


```
1 titanic["age"].std()  
  
3 #Output  
14.4134932112713
```

4.3.2.58 sub

```
RVC.sub(x)
```

Sub a number to the RVC.

Parameters

- **x:** *<positive float or int>*
Value to sub.

4.3.2.59 tan

```
1 RVC.tan()
```

Apply the tan function to the RVC.

4.3.2.60 tanh

```
1 RVC.tanh()
```

Apply the tanh function to the RVC.

4.3.2.61 undo_impute

```
1 RVC.undo_impute()
```

Undo the last imputation.

4.3.2.62 value_counts

```
1 RVC.value_counts(max_cardinality=6)
```

Summarize the RVC categories using the count aggregation.

Parameters

- **max_cardinality:** *<bool>*, optional
The maximum cardinality of the column. Under this number the column is automatically considered as categorical.

Returns

An object named `column_matrix` containing all the summarized information (they will be stored in the `data_column` attribute).

Example

```
1 titanic["pclass"].value_counts()  
  
3 #Output  
709 3  
323 1  
277 2  
7 Name: pclass, dtype: int
```

4.4 Functions

4.4.1 drop_table

```
1 drop_table(input_relation, cursor)
```

Drop the corresponding input relation.

Parameters

- **input_relation:** *<str>*
Name of the input relation (table or temporary table).
- **cursor:** *<object>*
Database cursor.

Warning

Be sure before dropping a table, this action is irreversible !

Example

```
1 drop_table("titanic", cur)  
  
3 #Output  
The table titanic was successfully dropped.
```

4.4.2 drop_view

```
drop_view(view_name, cursor)
```

Drop the corresponding view.

Parameters

- **view_name:** <str>
Name of the view.
- **cursor:** <object>
Database cursor.

Warning

Be sure before dropping a view, this action is irreversible !

Example

```
1 drop_view("titanic_temp", cur)
3 #Output
The view titanic_temp was successfully dropped.
```

4.4.3 read_csv

```
read_csv(path, cursor, local=True, input_relation=None, delimiter=',', columns
         =None, types=None, null='', enclosed_by='"', escape='\\', skip=1, temporary=
         False, skip_all=False, split=False, split_name='vpython_split')
```

Read a CSV file and create an input relation automatically or manually.

Parameters

- **path:** <str>
Path to the file in the Vertica server.
- **cursor:** <object>
Database cursor.
- **local:** <bool>
To use a local path instead of the node path.
- **input_relation:** <str>, optional
Name of the new input relation.
- **delimiter:** <str>, optional
Delimiter used to parse the file.

- **columns:** *<list of str>*, optional
List of the different columns ("types" and "columns" must have the same size).
- **types:** *<list of str>*, optional
List of the different types ("types" and "columns" must have the same size).
- **null:** *<str>*, optional
How the null elements are encoded.
- **enclosed_by:** *<str>*, optional
How the text elements are enclosed.
- **escape:** *<str>*, optional
How the escape is encoded.
- **skip:** *<positive int>*, optional
Number of elements to skip.
- **temporary:** *<bool>*, optional
Create a temporary table instead of a table.
- **skip_all:** *<bool>*, optional
To skip the types changements.
- **split:** *<bool>*, optional
Add a split column to the table (random numbers between 0 and 1).
- **split_name:** *<str>*, optional
Name of the split column.

Returns

The RVD of the new table.

Example

```

1 titanic=read_csv('titanic.csv', cur)

3 #Output
The parser guess the following columns and types:
5 age: Numeric(6,3)
  boat: Integer
7 body: Integer
  cabin: Varchar(30)
9 embarked: Varchar(20)
  fare: Numeric(10,5)
11 home.dest: Varchar(100)
  name: Varchar(164)
13 parch: Integer
  pclass: Integer
15 sex: Varchar(20)
  sibsp: Integer
17 survived: Integer
  ticket: Varchar(36)

```

```

19 Illegal characters in the columns names will be erased.
   Is any type wrong?
21 If one of the types is not correct, it will be considered as Varchar(100).
   0 - There is one type that I want to modify.
23   1 - I wish to continue.
   2 - I wish to see the columns and their types again.
25 #Input=1
   /\ Warning: Type of boat was changed to Varchar(100)
27 The table titanic has been successfully created.

```

4.4.4 run_query

```
1 run_query(query, cursor, limit=1000)
```

Return a query result (only for a select statement).

Parameters

- **query:** *<str>*
The sql query.
- **cursor:** *<object>*
The database cursor.
- **limit:** *<positive int>*, optional
The maximum number of element to return.

Returns

An object named column_matrix containing the query result (the information will be stored in the data_columns attribute).

Example

```

1 from vertica_ml_python import run_query

3 run_query("select survived, pclass, sex, count(*) from titanic group by sex,
   pclass, survived;", cur)

5 #Output

7      survived    pclass      sex    count
9  0           0         1    male     118
10  1           1         1    male      61
11  2           1         2  female      94
12  3           0         2  female      12
13  4           1         2    male      25
14  5           0         2    male     146

```

```

6          0          3    female    110
15 7          1          3    female    106
8          0          3     male     418
17 9          1          3     male      75
10         1          1    female    139
19 11         0          1    female      5
count=12 rows, elapsed_time=0.0224916934967041

```

5 vertica_ml_python.vml

VML (Vertica Machine Learning) is a package containing a set of functions for Machine Learning. Each Machine Learning algorithm is an object easy to explore. The user can see the performance of each algorithm. He will be able to see which algorithm is the best for the case and he will have all he needs to try to build a better one. He can also add the prediction to the RVD in order to use all the RVD methods.

5.1 Functions

5.1.1 accuracy

```
accuracy(model, threshold=0.5, input_class=1)
```

Compute the accuracy of a model using the corresponding threshold for the input class.

Parameters

- **model:** *<object>*
The model used to compute the accuracy.
- **threshold:** *<float in [0,1]>*, optional
The model threshold.
- **input_class:** *<str>*, optional
The input class used to compute the accuracy.

Returns

The model accuracy.

Example

```

1 from vertica_ml_python import accuracy
3 accuracy(logit)
5 #Output
0.8051044083526679

```

5.1.2 auc

```
accuracy(model, input_class=1)
```

Compute the auc of a model for the input class.

Parameters

- **model:** *<object>*
The model used to compute the auc.
- **input_class:** *<str>*, optional
The input class used to compute the auc.

Returns

The model auc.

Example

```
1 from vertica_ml_python import auc
3 auc(logit)
5 #Output
0.8317227859778604
```

5.1.3 champion_challenger_binomial

```
champion_challenger_binomial(input_relation, response_column,
    predictor_columns, cursor, fold_count=3, max_iterations=100,
    logit_optimizer='Newton', logit_regularization='None', logit_alpha=0.5,
    rf_ntree=20, rf_mtry=None, rf_max_depth=5, rf_sampling_size=0.632)
```

Champion Challenger for Binomial Model: Compare all the binomial models to find the best one.

Parameters

- **input_relation:** *<str>*
The table or view to consider.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.

- **fold_count:** *<positive int>*, optional
The number of fold to compute.
- **rf_ntree:** *<positive int>*, optional
Number of trees for the RF.
- **rf_mtry:** *<positive int>*, optional
A positive integer number that indicates the number of features to be considered at the split of a tree node for the RF.
- **rf_sampling_size:** *<float in [0,1]>*, optional
A number that indicates what portion of the input data set will randomly be picked for training each tree for the RF.
- **rf_max_depth:** *<int in [1,100]>*, optional
A positive integer number that specifies the maximum depth for growing each tree for the RF.
- **max_iterations:** *<positive int>*, optional
Determines the maximum number of iterations each algorithm performs before achieving the specified accuracy result.
- **logit_regularization:** *<str>*, optional
L1 | L2 | ENet | None (default)
Determines the method of regularization for the Logit.
- **logit_alpha:** *<positive float in [0,1]>*, optional
ENet mixture parameter that defines how much L1 versus L2 regularization to provide for the Logit. This argument will send a warning if it is used without ENet regularization.
- **logit_optimizer:** *<str>*, optional
BFGS | CGD | Newton (default)
The optimizer method used to train the model for the Logit. If no optimizer is set and regularization is set to L1, the default optimizer switches to CGD.

Returns

An object named `column_matrix` containing the champion challenger result (the information will be stored in the `data_columns` attribute).

Warning

This function can take a lot of time as a lot of models are computed.

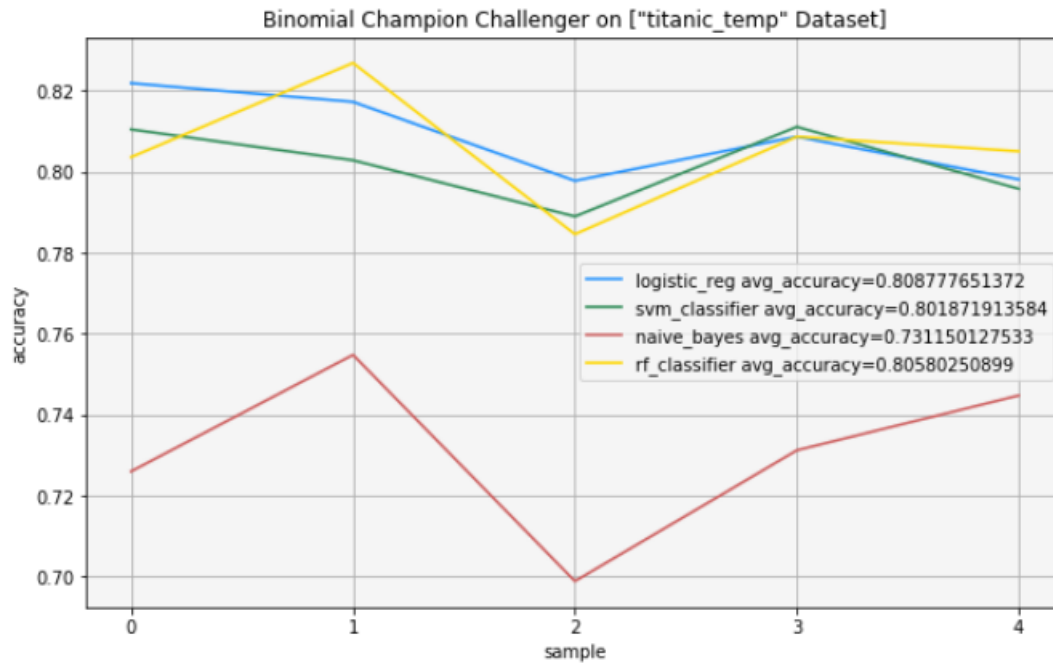
Example

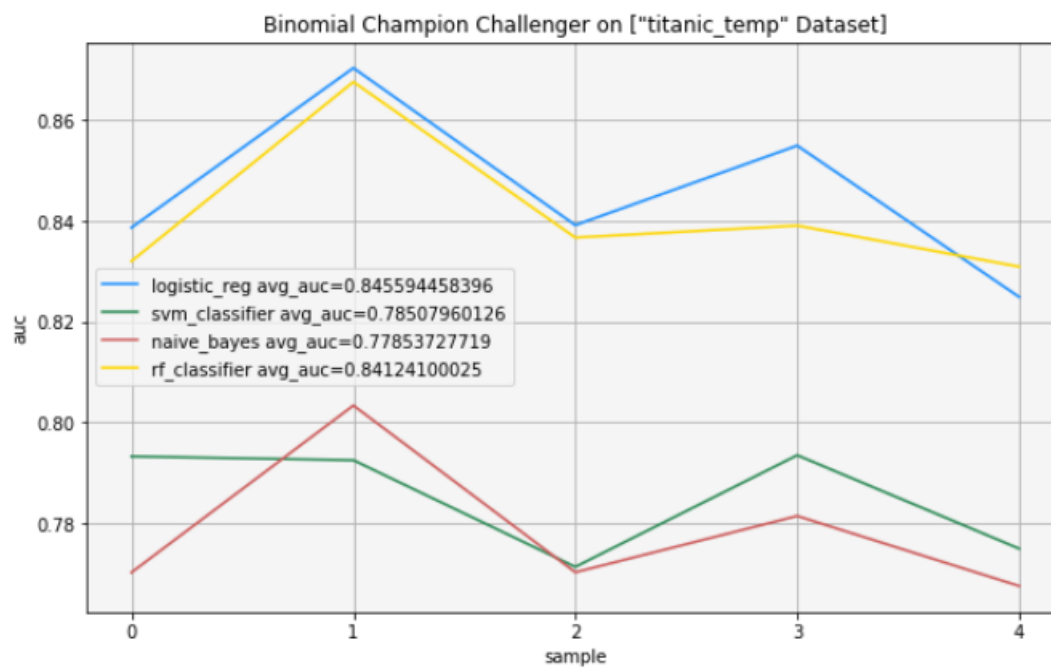
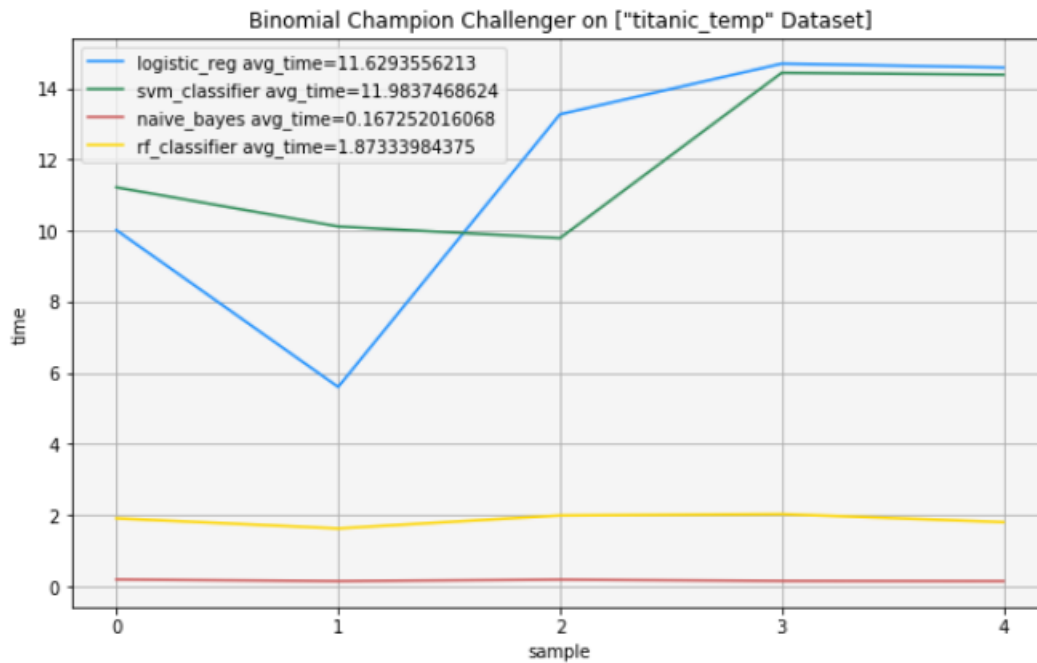
```

1 from vertica_ml_python import champion_challenger_binomial
3 champion_challenger_binomial('titanic_temp', 'survived', ['is_Army', 'is_Dr', '
    is_Rev', 'is_Mrs', 'is_Miss', 'is_Mr', 'embarked_C', 'embarked_Q', '
    embarked_S', 'fare', 'age', 'pclass', 'gender', 'family_size', 'parch', '
    sibsp'], cur, logit_optimizer="BFGS", fold_count=5)
5 #Output
                                avg_time          avg_auc          avg_accuracy
                                std_accuracy

```


7	logistic_reg	11.6293556213	0.845594458396	0.808777651372
	0.00978741350056			
	svm_classifier	11.9837468624	0.78507960126	0.801871913584
	0.00852471877201			
9	naive_bayes	0.167252016068	0.77853727719	0.731150127533
	0.0190351874381			
	rf_classifier	1.87333984375	0.84124100025	0.80580250899
	0.0134801561568			





5.1.4 confusion_matrix

```
confusion_matrix(model, threshold=0.5, input_class=1)
```

Compute the confusion matrix of a model using the corresponding threshold for the input class.

Parameters

- **model:** *<object>*
The model used to compute the confusion matrix.
- **threshold:** *<float in [0,1]>*, optional
The model threshold.
- **input_class:** *<str>*, optional
The input class used to compute the confusion matrix.

Returns

An object named `column_matrix` containing the confusion matrix (the information will be stored in the `data_columns` attribute).

Example

```

1 from vertica_ml_python import confusion_matrix
3 confusion_matrix(logit)
5 #Output
Confusion Matrix
7           0           1
0          459           72
9 1          109          217

```

5.1.5 drop_model

```

1 drop_model(model_name, cursor)

```

Drop the corresponding model.

Parameters

- **model_name:** *<str>*, optional
Model Name.
- **cursor:** *<object>*, optional
The database cursor.

Warning

Be sure before dropping a model, this action is irreversible !

Example

```
1 from vertica_ml_python import drop_model
3 drop_model("iris_kmeans", cur)
5 #Output
The model iris_kmeans was successfully dropped.
```

5.1.6 elbow

```
elbow(input_relation, input_columns, cursor, max_num_cluster=15,
      max_iterations=10, epsilon=1e-4, init_method="kmeanspp")
```

Compute the Elbow curve in order to determine the optimal number of clusters.

Parameters

- **input_relation:** *<str>*
The input relation.
- **input_columns:** *<list of str>*
The columns used to perform the kmeans.
- **cursor:** *<object>*
The database cursor.
- **max_num_cluster:** *<positive int>*, optional
The maximum number of cluster, the user wants to have.
- **max_iterations:** *<positive int>*, optional
The maximum number of iterations the algorithm performs.
- **epsilon:** *<positive float>*, optional
Determines whether the algorithm has converged. If, after an iteration, no component of any cluster center changes more than the value of epsilon, the algorithm has converged.
- **init_method:** *<str>*, optional
random | kmeanspp (default)
The method used to find the initial cluster centers.

Returns

An object named `column_matrix` containing the all within cluster SS per number of clusters (the information will be stored in the `data_columns` attribute).

Warning

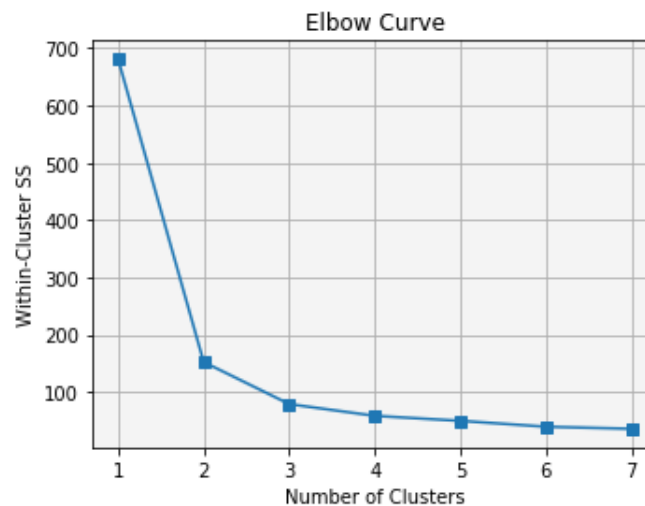
`max_num_cluster` models are computed. It can take a lot of time depending on the case.

Example

```

1 from vertica_ml_python import elbow
3 elbow(input_relation="Iris", input_columns=["PetalLengthCm", "SepalLengthCm", "PetalWidthCm", "SepalWidthCm"], cursor=cur, max_num_cluster=8)
5 #Output
num_clusters      all_within_cluster_SS
7 1                680.8244
8 2                152.36871
9 3                78.945066
10 4                58.458695
11 5                49.748082
12 6                39.498349
13 7                35.826633

```



5.1.7 error_rate

```
error_rate(model, threshold=0.5, input_class=1)
```

Compute the error rate of a model using the corresponding threshold for the input class.

Parameters

- **model:** *<object>*
The model used to compute the error rate.
- **threshold:** *<float in [0,1]>*, optional
The model threshold.
- **input_class:** *<str>*, optional
The input class used to compute the error rate.

Returns

An object named `column_matrix` containing the error rate (the information will be stored in the `data_columns` attribute).

Example

```

1 from vertica_ml_python import error_rate
3 error_rate(logit)
5 #Output
Error Rate
7
          error_rate
0          0.135593220338983
9 1          0.334355828220859
total        0.21120186697783

```

5.1.8 features_importance

```
features_importance(model, show=True, with_intercept=False)
```

Compute the features importance of a model.

Parameters

- **model:** *<object>*
The model used to compute the features importance.
- **show:** *<bool>*, optional
Display the result using matplotlib.
- **with_intercept:** *<bool>*, optional
Include the intercept coefficient in the computation if there is one.

Returns

An object named `column_matrix` containing the features importance (the information will be stored in the `data_columns` attribute).

Example

```

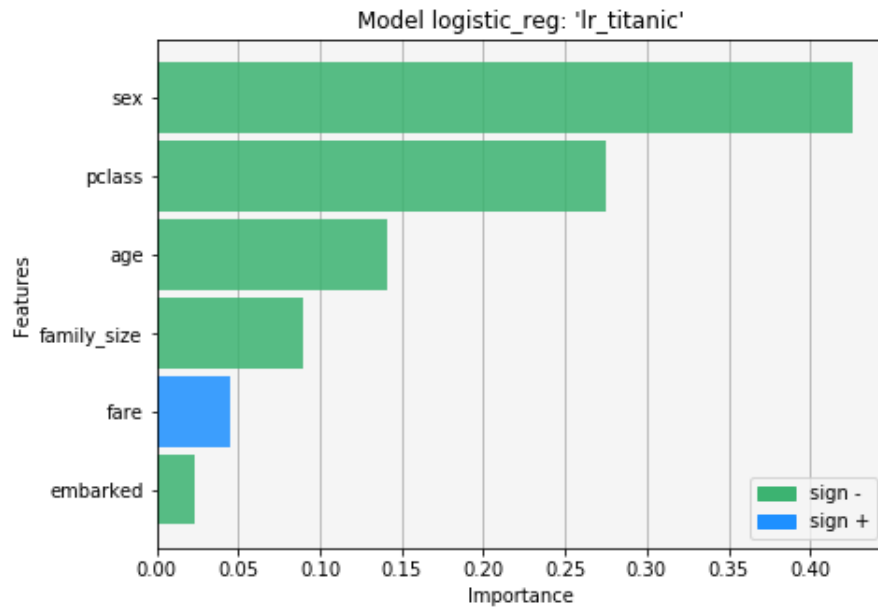
1 from vertica_ml_python import features_importance
3 features_importance(logit)
5 #Output
          Importance
7 embarked        0.023673338700660726

```

```

fare          0.04501061381123661
family_size   0.08998333220194923
age           0.14109667320278477
pclass        0.2746966760328248
sex           0.4255393660505437

```



5.1.9 lift_table

```

lift_table(model, num_bins=200, color=["dodgerblue", "#444444"], show=True,
            input_class=1)

```

Draw the lift table of a model.

Parameters

- **model:** *<object>*
The model used to compute the lift table.
- **num_bins:** *<positive int>*, optional
The number of bins used to draw the lift table.
- **color:** *<list of str>*, optional
The color of the lift table and random line.
- **show:** *<bool>*, optional
Display the result using matplotlib.
- **input_class:** *<str>*, optional
The input class used to draw the lift table.

Returns

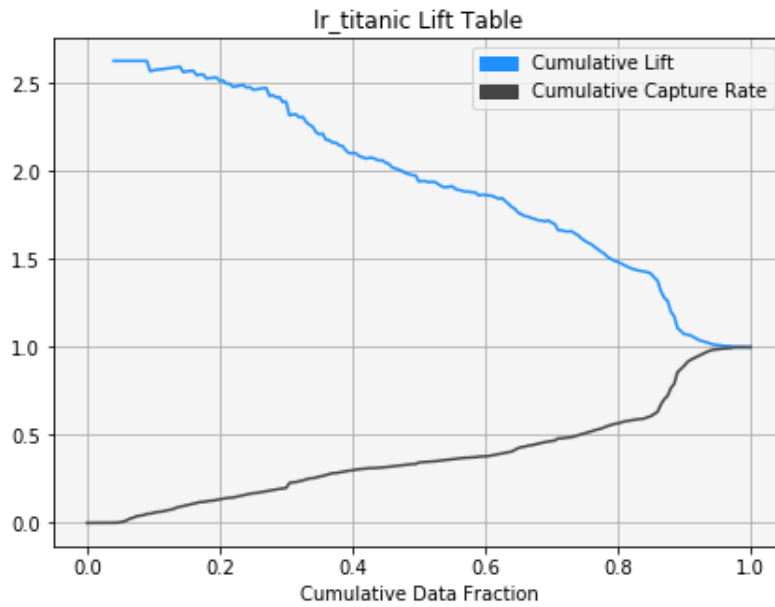
An object named `column_matrix` containing the lift table information (the information will be stored in the `data_columns` attribute).

Example

```

1 from vertica_ml_python import lift_table
3 lift_table(logit)
5 #Output
   decision_boundary  positive_prediction_ratio  lift
7 0                0.0                0.0      nan
8 1                0.005              0.0      nan
9 2                0.01               0.0      nan
10 3               0.015              0.0      nan
11 4               0.02               0.0      nan
12 5               0.025              0.0      nan
13 6               0.03               0.0      nan
14 7               0.035              0.0      nan
15 8               0.04      0.00116686114352392  2.62883435582822
16 9               0.045      0.00116686114352392  2.62883435582822
17 10              0.05      0.00466744457409568  2.62883435582822
18 11              0.055      0.00816802800466744  2.62883435582822
19 12              0.06      0.015169194865811    2.62883435582822
20 13              0.065      0.0256709451575263  2.62883435582822
21 14              0.07      0.0303383897316219  2.62883435582822
22 15              0.075      0.0385064177362894  2.62883435582822
23 16              0.08      0.0408401400233372  2.62883435582822
24 17              0.085      0.044340723453909  2.62883435582822
25 18              0.09      0.0513418903150525  2.62883435582822
26 19              0.095      0.0525087514585764  2.57041581458759
27 20              0.1       0.0571761960326721  2.57518467509703
--More--

```

5.1.10 load_model

```
load_model(model_name, cursor, input_relation=None)
```

Load the ML model.

Parameters

- **model_name:** <str>
The model name.
- **cursor:** <object>
The database cursor.
- **input_relation:** <str>, optional
The input relation used by the model.

Returns

The corresponding ML model.

Example

```
1 from vertica_ml_python import load_model
3 rf_test=load_model("rf_titanic", cur, input_relation="test_titanic033")
```

5.1.11 logloss

```
1 logloss(model)
```

Compute the logloss of a multinomial model.

Parameters

- **model:** *<object>*
The model used to compute the logloss.

Returns

The model logloss (classification only).

Example

```
1 from vertica_ml_python import logloss
3 logloss(logit)
5 #Output
0.201103505898445
```

5.1.12 metric_rf_curve_ntree

```
metric_rf_curve_ntree(input_relation, test_relation, response_column,
    predictor_columns, cursor, mode='logloss', ntree_begin=1, ntree_end=20,
    mtry=None, sampling_size=0.632, max_depth=5, max_breadth=32, min_leaf_size
    =5, min_info_gain=0.0, nbins=32, test_only=True)
```

Compute the corresponding metric RF curve in order to determine the optimal number of trees.

Parameters

- **input_relation:** *<str>*
The table or view that contains the training samples.
- **test_relation:** *<str>*
The table or view where the function will compute the corresponding metric.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.

- **mode:** *<str>*, optional
logloss | accuracy | error_rate | auc
The metric to be computed.
- **ntree_begin:** *<positive int>*, optional
Number of trees to begin with.
- **ntree_end:** *<positive int>*, optional
Number of trees to end with.
- **mtry:** *<positive int>*, optional
A positive integer number that indicates the number of features to be considered at the split of a tree node.
- **sampling_size:** *<float in [0,1]>*, optional
A number that indicates what portion of the input data set will randomly be picked for training each tree.
- **max_depth:** *<int in [1,100]>*, optional
A positive integer number that specifies the maximum depth for growing each tree.
- **max_breadth:** *<int in [1,1e9]>*, optional
A positive integer number that specifies the maximum number of leaf nodes a tree in the forest can have.
- **min_leaf_size:** *<int in [1,1e6]>*, optional
A positive integer number that specifies the minimum samples each branch must have after splitting a node. A split that causes fewer remaining samples will be discarded.
- **min_info_gain:** *<float in [0,1]>*, optional
A non-negative number. Any split with information gain less than this threshold will be discarded.
- **nbins:** *<int in [2,1000]>*, optional
A positive integer number that indicates the number of bins to use for continuous features.
- **test_only:** *<bool>*, optional
Plot the two curves (train and test).

Returns

A list of two objects [mode_test, mode_train] named column_matrix containing the corresponding metric per number of trees (the information will be stored in the data_columns attribute).

Warning

ntree_end-ntree_begin+1 models are computed. It can take a lot of time depending on the case.

Example

```

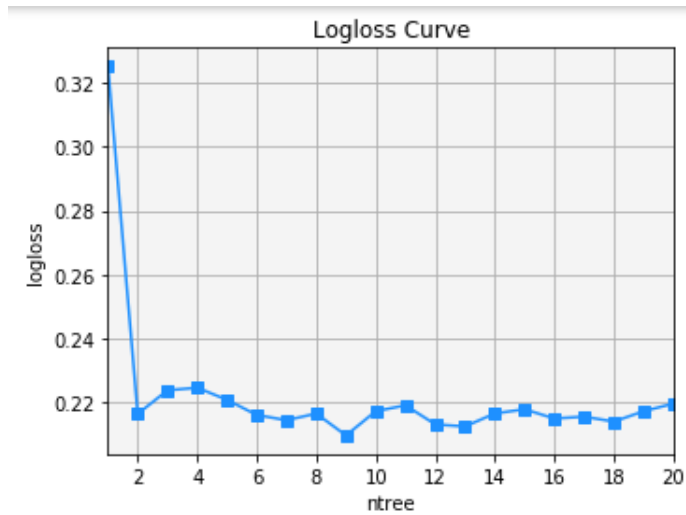
1 from vertica_ml_python import metric_rf_curve_ntree
3 metric_rf_curve_ntree(input_relation='train_titanic067', test_relation='
    test_titanic033' response_column='survived', predictor_columns=["age", "sex
    ", "family_size", "embarked", "pclass", "name"], cursor=cur)
5 #Output
    (ntree          logloss_test
7  1          0.3253279848340835
    2          0.21629507754633498

```

```

9  3          0.2237076208677535
11 4          0.22451867403603498
12 5          0.2207204740572145
--More--

```



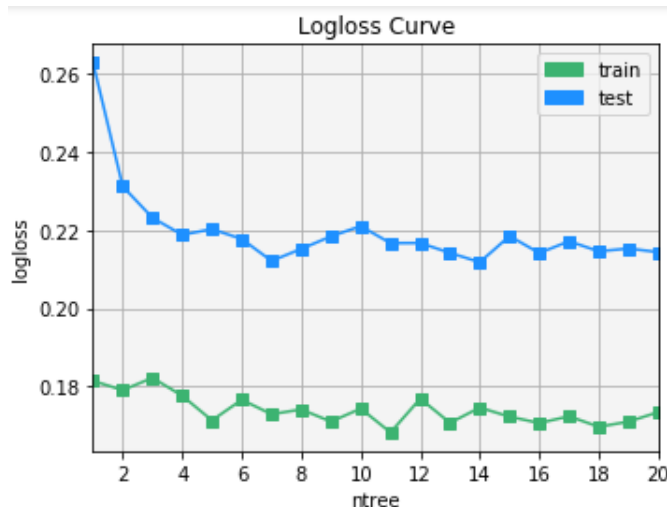
```

from vertica_ml_python import metric_rf_curve_ntree

metric_rf_curve_ntree(input_relation='train_titanic067', test_relation='
test_titanic033' response_column='survived', predictor_columns=["age", "sex
", "family_size", "embarked", "pclass", "name"], cursor=cur, test_only=False)

#Output
(ntree          logloss_test
1          0.2629559858572835
2          0.23122991569256252
3          0.2231809471918855
4          0.218823242607503
5          0.2202786826615065
--More--

```



5.1.13 metric_rf_curve_depth

```
metric_rf_curve_depth(input_relation, test_relation, response_column,
    predictor_columns, cursor, mode='logloss',
    ntree=20, mtry=None, sampling_size=0.632, max_depth_begin=1,
    max_depth_end=12,
    max_breadth=32, min_leaf_size=5, min_info_gain=0.0, nbins=32,
    test_only=True)
```

Compute the corresponding metric RF curve in order to determine the optimal max depth.

Parameters

- **input_relation:** *<str>*
The table or view that contains the training samples.
- **test_relation:** *<str>*
The table or view where the function will compute the corresponding metric.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.
- **mode:** *<str>*, optional
logloss | accuracy | error_rate | auc
The metric to be computed.
- **ntree:** *<positive int>*, optional
Number of trees.
- **mtry:** *<positive int>*, optional
A positive integer number that indicates the number of features to be considered at the split of a tree node.

- **sampling_size:** *<float in [0,1]>*, optional
A number that indicates what portion of the input data set will randomly be picked for training each tree.
- **max_depth_begin:** *<int in [1,100]>*, optional
A positive integer number that specifies the maximum depth for growing each tree to begin with.
- **max_depth_end:** *<int in [1,100]>*, optional
A positive integer number that specifies the maximum depth for growing each tree to end with.
- **max_breadth:** *<int in [1,1e9]>*, optional
A positive integer number that specifies the maximum number of leaf nodes a tree in the forest can have.
- **min_leaf_size:** *<int in [1,1e6]>*, optional
A positive integer number that specifies the minimum samples each branch must have after splitting a node. A split that causes fewer remaining samples will be discarded.
- **min_info_gain:** *<float in [0,1]>*, optional
A non-negative number. Any split with information gain less than this threshold will be discarded.
- **nbins:** *<int in [2,1000]>*, optional
A positive integer number that indicates the number of bins to use for continuous features.
- **test_only:** *<bool>*, optional
Plot the two curves (train and test).

Returns

A list of two objects [mode_test, mode_train] named column_matrix containing the metric per depth (the information will be stored in the data_columns attribute).

Warning

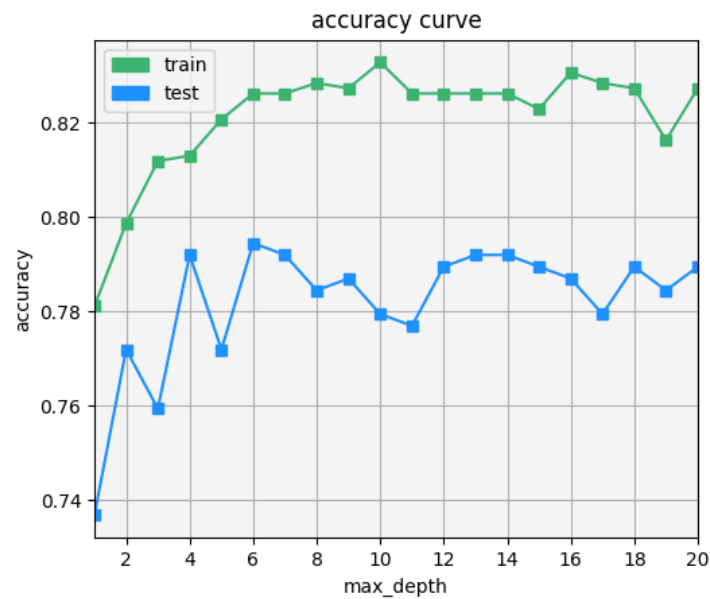
max_depth_end-max_depth_begin+1 models are computed. It can take a lot of time depending on the case.

Example

```

1 from vertica_ml_python import metric_rf_curve_depth
3 metric_rf_curve_depth(input_relation='train_titanic067', test_relation='
    test_titanic033' response_column='survived', predictor_columns=["age","sex
    ","family_size","embarked","pclass","name"], cursor=cur, test_only=False,
    mode='accuracy')
5 #Output
(max_depth      accuracy_test
7 1              0.736842105263
  2              0.771929824561
9 3              0.759398496241
  4              0.791979949875
11 5              0.771929824561
--More--

```



5.1.14 mse

```
mse(model)
```

Compute the mse of a regression model.

Parameters

- **model:** *<object>*
The model used to compute the mse.

Returns

The model mse.

Example

```
1 from vertica_ml_python import mse
3 mse(my_reg)
5 #Output
0.00230293988121166
```

5.1.15 reg_metrics

```
reg_metrics(model)
```

Compute the rsquared and mse of a regression model.

Parameters

- **model:** *<object>*
The model used to compute the rsquared and mse.

Returns

An object named `column_matrix` containing the mse and rsquared of the model (the information will be stored in the `data_columns` attribute).

Example

```
1 from vertica_ml_python import reg_metrics
3 reg_metrics(my_reg)
5 #Output
7 mse          0.00230293988121166
  rsquared      0.266308495008143
```

5.1.16 rsquared

```
rsquared(model)
```

Compute the rsquared of a regression model.

Parameters

- **model:** *<object>*
The model used to compute the rsquared.

Returns

The model rsquared.

Example

```
1 from vertica_ml_python import rsquared
3 rsquared(my_reg)
5 #Output
  0.266308495008143
```


5.1.17 roc

```
roc(model, num_bins=200, color=["dodgerblue", "#444444"], show=True,
    input_class=1)
```

Draw the roc curve of a model.

Parameters

- **model:** *<object>*
The model used to compute the roc.
- **num_bins:** *<positive int>*, optional
The number of bins used to draw the roc.
- **color:** *<list of str>*, optional
The color of the roc and random line.
- **show:** *<bool>*, optional
Display the result using matplotlib.
- **input_class:** *<str>*, optional
The input class used to draw the roc.

Returns

A list of two objects named `column_matrix` containing the roc information (the information will be stored in the `data_columns` attribute).

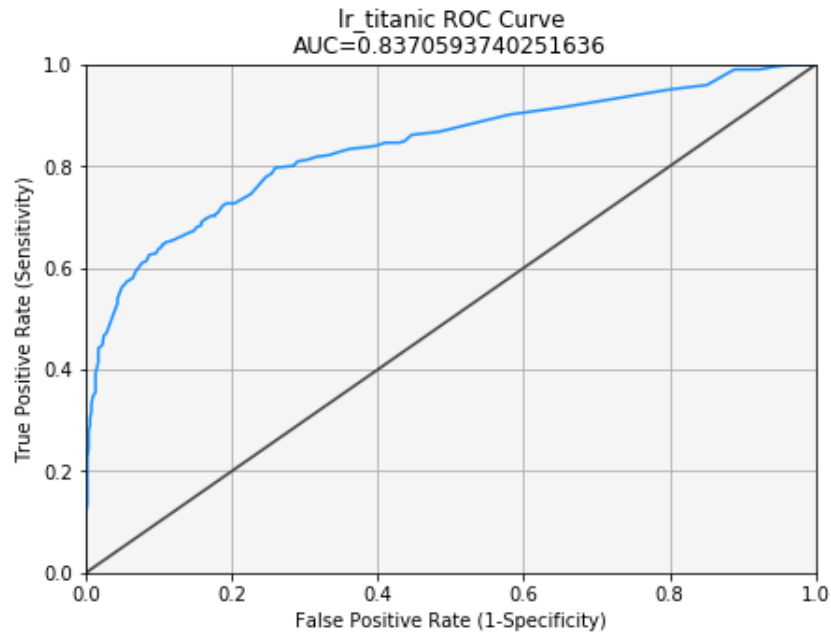
Example

```
1 from vertica_ml_python import roc
3 roc(logit)
5 #Output
6 (
7     auc          0.8370593740251636
8     best_threshold 0.57 ,
9     threshold      false_positive      true_positive
10    0          0.0          1.0          1.0
11    1          0.01         1.0          1.0
12    2          0.02         1.0          1.0
13    3          0.03    0.990583804143126          1.0
14    4          0.04    0.988700564971751          1.0
15    5          0.05    0.981167608286252          1.0
16    6          0.06    0.969868173258004          1.0
17    7          0.07    0.947269303201507    0.996932515337423
18    8          0.08    0.922787193973635    0.99079754601227
19    9          0.09    0.8888888888888889    0.99079754601227
20   10          0.1    0.851224105461394    0.960122699386503
```

```

21 11          0.11          0.7984934086629          0.950920245398773
22 12          0.12          0.657250470809793          0.917177914110429
23 --More--

```



5.1.18 summarize_model

```

1 summarize_model(model_name, cursor)

```

Summarize the corresponding model.

Parameters

- **model_name:** <str>
Name of the model to summarize.
- **cursor:** <object>
The database cursor.

Returns

The summarized information.

Example

```

1 from vertica_ml_python import summarize_model
3 print(summarize_model("lr_titanic", cur))

```

```

5 #Output
coeff names: {Intercept, age, sex, family_size, embarked, fare, pclass}
7 coefficients: {4.494253459, -0.03180813173, -2.55322967, -0.1658014114,
  -0.1024176438, 0.002514827327, -0.9404607164}
std_err:      {0.5927, 0.008183, 0.1911, 0.0671, 0.1353, 0.002364, 0.1438}
9 z_value:     {7.583, -3.887, -13.36, -2.471, -0.7568, 1.064, -6.541}
p_value:      {3.376e-14, 0.0001014, < 1e-20, 0.01347, 0.4492, 0.2875, 6.116e
  -11}
11 Regularization method: none, lambda: 1
Number of iterations: 4, Number of skipped samples: 0, Number of processed
  samples: 857
13 Call:
logistic_reg('public.lr_titanic', 'train_titanic067', '"survived"', 'age,sex,
  family_size,embarked,fare,pclass'
15 USING PARAMETERS optimizer='newton', epsilon=1e-06, max_iterations=100,
  regularization='none', lambda=1, alpha=0.5)

```

5.1.19 tree

```
1 tree(model, n=0)
```

Print the tree in the terminal using the `anytree` API.

Parameters

- **model:** *<str>*
The model used to compute the tree (must be a random forest).
- **n:** *<positive int>*
The tree ID.

Warning

Be sure to have `anytree` installed in your computer and to have a folder named `anytree` where you execute the command (to draw a png of the tree, the name will be `(model_name)_(input_relation)(n).png`).

```
1 root@ubuntu:~$ pip install anytree
```

Returns

An objects named `column_matrix` containing the tree information (the information will be stored in the `data_columns` attribute).

Example

```

1 from vertica_ml_python import tree
  from vertica_ml_python import rf_classifier
3

```

```

rf=rf_classifier(model_name="rf_titanic", input_relation="train_titanic067",
    response_column="survived", predictor_columns=["age","gender","pclass","
    title"], cursor=cur, max_depth=3)
5 print(rf.tree(n=0))

7 #Output
-----

9 Tree Id: 0
Number of Nodes: 7
11 Tree Depth: 3
Tree Breadth: 3
-----

+
15 +++ (title=Mr)
+   +++ 0 (probability=0.869301)
17 +++ (title!=Mr)
    +++ (title=Mrs)
19   +   +++ 1 (probability=0.825)
    +++ (title!=Mrs)
21      +++ (pclass<2.0625)
    +   +++ 1 (probability=0.852941)
23      +++ (pclass>=2.0625)
    +++ 0 (probability=0.630952)

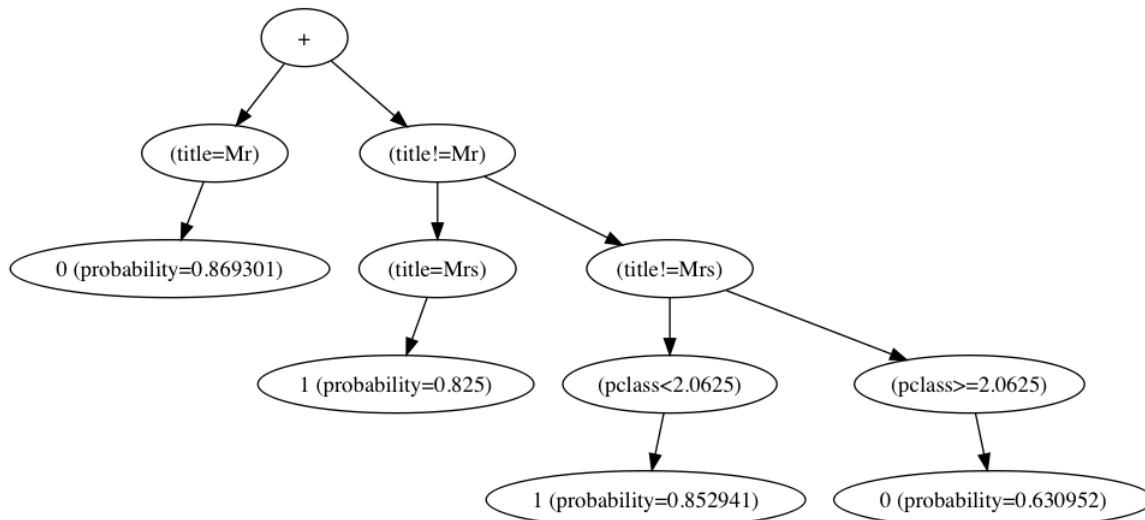
25 Tree0
NodeID      Node Depth      isLeaf      isSplitCategorical      split on      \\
27 1              0              0              1              title      \\
2 2              1              1              -              -      \\
29 3              1              0              1              title      \\
6 6              2              1              -              -      \\
31 7              2              0              0              pclass      \\
14 14             3              1              -              -      \\
33 15             3              1              -              -      \\
NodeID      threshold      leftChildID      rightChildID      prediction      \\
35 1              Mr              2              3              -      \\
2 2              -              -              -              0      \\
37 3              Mrs              6              7              -      \\
6 6              -              -              -              1      \\
39 7              2.0625          14             15             -      \\
14 14             -              -              -              1      \\
41 15             -              -              -              0      \\
NodeID      probability
43 1              -
2 2              0.869301
45 3              -
6 6              0.825

```

```

47 7          -
14      0.852941
49 15      0.630952

```



5.2 Machine Learning Models

Each machine learning model is represented by an object. This object has many methods and can sometimes call one of the ML functions described before. If it is the case, please refer directly to the function to know the parameters significations.

5.2.1 Cross Validation (`cross_validate`)

5.2.1.1 initialization

```

cross_validate(algorithm, input_relation, response_column, predictor_columns,
               cursor, model_name=None, fold_count=5, hyperparams=None, prediction_cutoff
               =0.5)

```

Performs cross validation on a learning algorithm using an input relation, and performs grid search for hyper parameters. The output is an average performance indicator of the selected algorithm.

Parameters

- **algorithm:** `<str>`
svm_classifier | naive_bayes | logistic_reg
The name of the training function of the algorithm.
- **input_relation:** `<str>`
The table or view that contains the data used for training and testing.
- **response_column:** `<str>`
The name of the column in the input_relation that contains the response.

- **predictor_columns:** *<list of str>*
A list of the columns in the input_relation that are passed to the algorithm as predictors.
- **cursor:** *<object>*
The database cursor.
- **model_name:** *<str>*, optional
The name that is used to retrieve the result of the cross validation process.
- **fold_count:** *<int>*, optional
The number of folds to split the data into.
- **hyperparams:** *<str>*, optional
A JSON string that describes the combination of parameters for use in grid search of hyper parameters.
- **prediction_cutoff:** *<float in [0,1]>*, optional
The cutoff threshold that is passed to the prediction stage of logistic regression.

Example

```

1 from vml import cross_validate
3 cross_validate(algorithm="logistic_reg", input_relation="train_titanic067",
   response_column="survived", predictor_columns=["age", "sex", "family_size", "
   embarked", "fare", "pclass"], cursor=cur)
5 #Output
   model_type='cross_validation'
7 model_name='_vpython_cv_2638'
   Counters:
9         counter_name      counter_value
10 0      accepted_row_count          881
11 1      rejected_row_count           0
12 2         feature_count           6
13 Fold Info:
   fold_id      row_count
15 0           0          160
16 1           1          164
17 2           2          179
18 3           3          208
19 4           4          170
   Details:
21 fold_id      iteration_count      accuracy      error_rate
22 0           0                5      0.81875      0.18125
23 1           1                5      0.786585365853659      0.213414634146341
24 2           2                5      0.776536312849162      0.223463687150838
25 3           3                5      0.798076923076923      0.201923076923077
26 4           4                5      0.817647058823529      0.182352941176471
27 Averages:
   accuracy      error_rate
29 0      0.799519132120655      0.200480867879345

```

```
The model _vpython_cv_2638 was successfully dropped.
```

5.2.1.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **model_type:** Model type (="cross_validation").

5.2.1.3 methods

Cross Validation has only one method:

```
cross_validate.get_model_attribute(attr_name="run_details")
```

Returns

A list of two objects named `column_matrix` containing the roc information (the information will be stored in the `data_columns` attribute).

Parameters

- **attr_name:** *<str>*
run_details (default) | run_average | fold_info | counters | call_string
Attribute name.

5.2.2 Kmeans (kmeans)

5.2.2.1 initialization

```
kmeans(model_name, input_relation, input_columns, num_clusters, cursor,
        max_iterations=10, epsilon=1e-4, init_method="kmeanspp", initial_centers=
        None )
```

Executes the k-means algorithm on an input table or view.

Parameters

- **model_name:** *<str>*
Model name.
- **input_relation:** *<str>*
The input relation.
- **input_columns:** *<list of str>*
The columns used to perform the kmeans.
- **cursor:** *<object>*
The database cursor.

- **max_iterations:** *<positive int>*, optional
The maximum number of iterations the algorithm performs.
- **epsilon:** *<positive float>*, optional
Determines whether the algorithm has converged. If, after an iteration, no component of any cluster center changes more than the value of epsilon, the algorithm has converged.
- **init_method:** *<str>*, optional
random | kmeanspp (default)
The method used to find the initial cluster centers.
- **initial_centers:** *<list of points>*, optional
The list of the initial cluster centers to use.

Example

```

1 from vml import kmeans

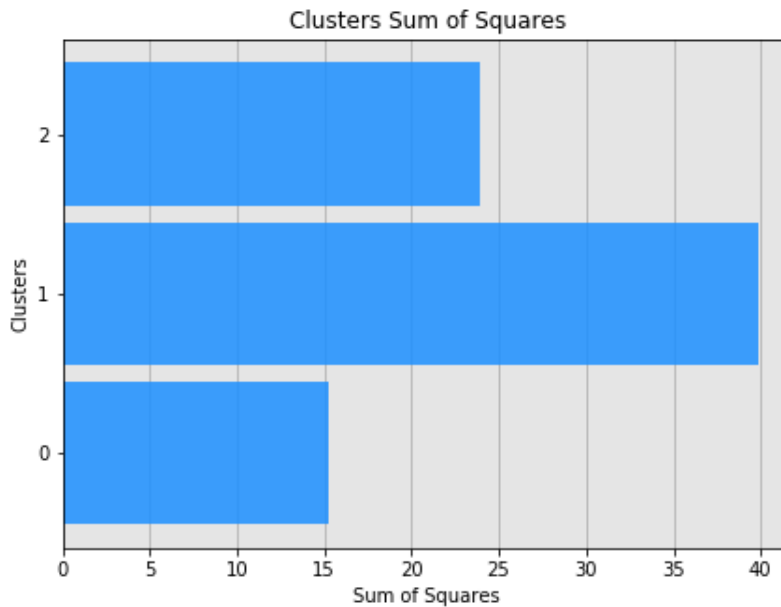
3 iris_kmeans=kmeans(model_name="iris_kmeans", input_relation="Iris",
    input_columns=["PetalLengthCm","SepalLengthCm","PetalWidthCm","
    SepalWidthCm"], num_clusters=3, cursor=cur, initial_centers=[(1,2,3,4)
    , (2,3,4,5) , (3,4,5,6)])

5 print(iris_kmeans)
iris_kmeans.cluster_SS()

7
#Output
9 model_type='kmeans'
model_name='iris_kmeans'
11 input_relation='Iris'
input_columns='PetalLengthCm,SepalLengthCm,PetalWidthCm,SepalWidthCm'
13 Clusters:
    PetalLengthCm      SepalLengthCm      PetalWidthCm      \
15 0          1.464          5.006          0.244      \
1      4.39354838709677      5.90161290322581      1.43387096774194      \
17 2      5.7421052631579          6.85      2.07105263157895      \
    SepalWidthCm
19 0          3.418
1      2.74838709677419
21 2      3.07368421052632

23 Cluster SS
0      15.2404
25 1      39.820968
2      23.879474

```

5.2.2.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **input_columns:** Input Columns.
- **num_clusters:** Number of Clusters.
- **model_type:** Model type (="kmeans").
- **model_category:** Model category (="clustering").

5.2.2.3 methods

```

kmeans.add_to_rvd(rvd, name="kmeans_cluster"+str(np.random.randint(10000)))
2 # Add the prediction to the RVD
# rvd = the corresponding RVD
4 # name = RVC name

6 kmeans.converged()
# Returns if the kmeans converged

8
kmeans.features_importance(show=True)
10 # Returns the model features importance

12 kmeans.between_cluster_SS()
# Returns the kmeans between cluster SS
14

```

```

kmeans.cluster_SS(show=True, display=True)
16 # Returns the kmeans cluster SS
# show = print the result in the terminal
18 # display = display the result using matplotlib

20 kmeans.total_SS()
# Returns the kmeans total SS

22
kmeans.within_cluster_SS()
24 # Returns the kmeans within cluster SS

```

5.2.3 Linear Regression (linear_reg)

5.2.3.1 initialization

```

linear_reg(model_name, input_relation, response_column, predictor_columns,
           cursor, optimizer='Newton', epsilon=1e-6, max_iterations=100,
           regularization="None", l=0.0, alpha=0.5)

```

Executes linear regression on an input table or view.

Parameters

- **model_name:** <str>
Model name.
- **input_relation:** <str>
The input relation.
- **response_column:** <str>
The name of the column in the input_relation that represents the dependent variable, or outcome.
- **predictor_columns:** <list of str>
A list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** <object>
The database cursor.
- **optimizer:** <str>, optional
BFGS | CGD | Newton (default)
The optimizer method used to train the model. If no optimizer is set and regularization is set to L1, the default optimizer switches to CGD.
- **epsilon:** <positive float>, optional
Determines whether the algorithm has reached the specified accuracy result.
- **max_iterations:** <positive int>, optional
Determines the maximum number of iterations the algorithm performs before achieving the specified accuracy result.
- **regularization:** <str>, optional
L1 | L2 | ENet | None (default)
Determines the method of regularization.

- **l:** *<positive float>*, optional
The regularization parameter value. The value must be zero or positive.
- **alpha:** *<positive float in [0,1]>*, optional
ENet mixture parameter that defines how much L1 versus L2 regularization to provide. This argument will send a warning if it is used without ENet regularization.

Example

```

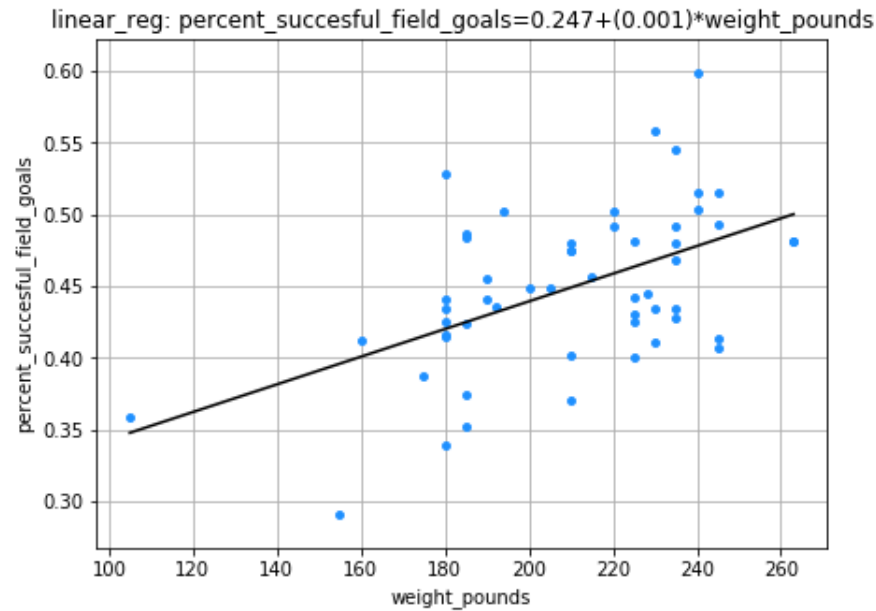
1 from vertica_ml_python import linear_reg

3 basketball=linear_reg(model_name="linear_reg_basketball", input_relation="
    basketball", response_column="percent_succesful_field_goals",
    predictor_columns=["weight_pounds"], cursor=cur)
4 print(basketball)
5 basketball.plot()

7 #Output
model_type='linear_reg'
9 model_name='linear_reg_basketball'
input_relation='basketball'
11 response_column='percent_succesful_field_goals'
predictor_columns='weight_pounds'
13 regularization: none
lambda: 0.0
15 rejected_row_count: 0
accepted_row_count: 54
17 Parameters:

```

	coefficient	std_error
t_value \\\		
19 Intercept	0.246705484199406	0.047062130975949
5.24212310584671 \\\		
weight_pounds	0.000964261478009015	0.000221951360156074
4.34447203806707 \\\		
	p_value	
21 Intercept	2.93605818609063e-06	
23 weight_pounds	6.49503578237179e-05	



5.2.3.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.
- **model_type:** Model type (="linear_reg").
- **model_category:** Model category (="regression").

5.2.3.3 methods

```

1 linear_reg.add_to_rvd(rvd, name="linear_reg_pred"+str(np.random.randint(10000)
  ))
  # Add the prediction to the RVD
3 # rvd = the corresponding RVD
  # name = RVC name
5
  linear_reg.details()
7 # Returns the model details.

9 linear_reg.features_importance(show=True)
  # Returns the model features importance.
11
  linear_reg.metrics()

```

```

13 # Returns the model metrics.

15 linear_reg.mse()
# Returns the model mse.

17 linear_reg.parameter_value(parameter_name="*", show=True)
19 # Returns the parameter value
# parameter_name = regularization | lambda | rejected_row_count |
#   accepted_row_count | *
21 # show = Print the parameter in the Terminal

23 linear_reg.plot(color=None, projection=None, max_nb_points=1000)
# Plot the linear regression using matplotlib
25 # color = [points_color, plan_color]
# projection = project the plan in a smaller space
27 # max_nb_points = maximum number of points in the graph

29 linear_reg.rsquared()
# Returns the model rsquared.

```

5.2.4 Logistic Regression (logistic_reg)

5.2.4.1 initialization

```

logistic_reg(model_name, input_relation, response_column, predictor_columns,
              cursor, optimizer='Newton', epsilon=1e-6, max_iterations=100,
              regularization='None', l=1, alpha=0.5)

```

Executes logistic regression on an input table or view.

Parameters

- **model_name:** <str>
Model name.
- **input_relation:** <str>
The input relation.
- **response_column:** <str>
The name of the column in the input_relation that represents the dependent variable, or outcome.
- **predictor_columns:** <list of str>
A list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** <object>
The database cursor.
- **optimizer:** <str>, optional
BFGS | CGD | Newton (default)
The optimizer method used to train the model. If no optimizer is set and regularization is set to L1, the default optimizer switches to CGD.

- **epsilon:** *<positive float>*, optional
Determines whether the algorithm has reached the specified accuracy result.
- **max_iterations:** *<positive int>*, optional
Determines the maximum number of iterations the algorithm performs before achieving the specified accuracy result.
- **regularization:** *<str>*, optional
L1 | L2 | ENet | None (default)
Determines the method of regularization.
- **l:** *<positive float>*, optional
The regularization parameter value. The value must be zero or positive.
- **alpha:** *<positive float in [0,1]>*, optional
ENet mixture parameter that defines how much L1 versus L2 regularization to provide. This argument will send a warning if it is used without ENet regularization.

Example

```

1 from vertica_ml_python import logistic_reg

3 logit=logistic_reg(model_name="lr_titanic", input_relation="train_titanic067",
    response_column="survived", predictor_columns=["age", "sex", "family_size",
    "embarked", "fare", "pclass"], cursor=cur)
print(logit)

5
#Output
7 model_type='logistic_reg'
  model_name='lr_titanic'
9 input_relation='train_titanic067'
  response_column='survived'
11 predictor_columns='age,sex,family_size,embarked,fare,pclass'
  regularization: none
13 lambda: 1.0
  rejected_row_count: 0
15 accepted_row_count: 881
  Parameters:
17
      coefficient                                std_error
   t_value  \
Intercept      5.26886364572958      0.620218968589887
      8.49516688873403  \
19 age          -0.0414010724097224      0.00796877387136067
      -5.19541313106092  \
   sex          -2.40595338730162      0.188879068874741
      -12.7380625160598  \
21 family_size    -0.181861830912543      0.0672217913391124
      -2.70539994977385  \
   embarked      -0.260251676377847      0.135648823577091
      -1.91856935810389  \

```

```

23 fare                0.00249310752912507      0.00272746881552252
    0.914073706337666    \
pclass                -1.08507579965827      0.151212578985253
    -7.17583025790531    \
25                                     p_value
Intercept             1.97648714752969e-17
27 age                 2.04265703445367e-07
sex                   3.63305310975195e-37
29 family_size         0.00682221957084204
embarked              0.0550388566073697
31 fare                0.36067811758821
pclass                7.18696796176092e-13

```

5.2.4.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.
- **model_type:** Model type (="logistic_reg").
- **model_category:** Model category (="binomial").

5.2.4.3 methods

```

logistic_reg.accuracy(threshold=0.5)
2 # Returns the model accuracy

4 logistic_reg.add_to_rvd(rvd, name="logistic_reg_pred"+str(np.random.randint
    (10000)), prediction_type='response', cutoff=0.5)
# Add the prediction to the RVD
6 # rvd = the corresponding RVD
# name = RVC name
8 # prediction_type = response (default) | probability
# cutoff = logistic regression cut-off
10
logistic_reg.auc()
12 # Returns the model auc

14 logistic_reg.confusion_matrix(threshold=0.5)
# Returns the model confusion matrix
16
logistic_reg.details()

```

```

18 # Returns the model details.

20 logistic_reg.error_rate(threshold=0.5)
  # Returns the model error rate.

22 logistic_reg.features_importance(show=True)
24 # Returns the model features importance.

26 logistic_reg.lift_table(num_bins=100, color=["dodgerblue", "#444444"], show=
    True, input_class=1)
  # Draw model lift table.

28 logistic_reg.logloss()
30 # Returns the model logloss.

32 logistic_reg.parameter_value(parameter_name="*", show=True)
  # Returns the parameter value
34 # parameter_name = regularization | lambda | rejected_row_count |
    accepted_row_count | *
  # show = Print the parameter in the Terminal

36 logistic_reg.plot(marker=["o", "^"], color=None, projection=None, max_nb_points
    = None)
38 # Plot the logistic regression regression using matplotlib
  # marker = [0_marker, 1_marker]
40 # color = [0_color, 1_color, logit_color]
  # projection = project the plan in a smaller space
42 # max_nb_points = maximum number of points in the graph

44 logistic_reg.roc(num_bins=100, color=["dodgerblue", "#444444"], show=True)
  # Draw model roc.

```

5.2.5 Naive Bayes (naive_bayes)

5.2.5.1 initialization

```

1 naive_bayes(model_name, input_relation, response_column, predictor_columns,
    cursor, alpha=1.0)

```

Executes the Naive Bayes algorithm on an input table or view.

Parameters

- **model_name:** <str>
Model name.

- **input_relation:** *<str>*
The input relation.
- **response_column:** *<str>*
The name of the column in the input_relation that represents the dependent variable, or outcome.
- **predictor_columns:** *<list of str>*
A list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.
- **alpha:** *<float>*, optional
The parameter used to control Laplace smoothing. Specifies use of Laplace smoothing if the event model is categorical, multinomial, or Bernoulli.

Example

```

1 from vertica_ml_python import naive_bayes
3 nb=naive_bayes(model_name="nb_titanic",input_relation="train_titanic067",
   response_column="survived",
   predictor_columns=["age","sex","family_size","embarked","pclass"],cursor=cur)
5 print(nb)

7 #Output
model_type='naive_bayes'
9 model_name='nb_titanic'
input_relation='train_titanic067'
11 response_column='survived'
predictor_columns='age,sex,family_size,embarked,pclass'
13 alpha: 1.0
rejected_row_count: 0
15 accepted_row_count: 881
Probabilities:
17 class          value
0          0.625425652667423
19 1          0.374574347332577

```

5.2.5.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.

- **model_type:** Model type (="naive_bayes").
- **model_category:** Model category (="binomial"/"multinomial").

5.2.5.3 methods

```

1 naive_bayes.accuracy(threshold=0.5, input_class=None)
  # Returns the model accuracy
3
4 naive_bayes.add_to_rvd(rvd, name="naive_bayes_pred"+str(np.random.randint
  (10000)), prediction_type='response', input_class=None)
5 # Add the prediction to the RVD
  # rvd = the corresponding RVD
7 # name = RVC name
  # prediction_type = response (default) | probability
9 # input_class = The class used when prediction_type="probability"

11 naive_bayes.auc(input_class=None)
  # Returns the model auc
13
14 naive_bayes.confusion_matrix(threshold=0.5, input_class=None)
15 # Returns the model confusion matrix

17 naive_bayes.details()
  # Returns the model details.
19
20 naive_bayes.error_rate(threshold=0.5, input_class=None)
21 # Returns the model error rate.

23 naive_bayes.lift_table(num_bins=100, color=["dodgerblue", "#444444"], show=True
  , input_class=None)
  # Draw the model lift table.
25
26 naive_bayes.logloss()
27 # Returns the model logloss.

29 naive_bayes.parameter_value(parameter_name="*", show=True)
  # Returns the parameter value
31 # parameter_name = alpha | rejected_row_count | accepted_row_count | *
  # show = Print the parameter in the Terminal
33
34 naive_bayes.roc(num_bins=100, color=["dodgerblue", "#444444"], show=True,
  input_class=None)
35 # Draw the model roc.

```

5.2.6 Random Forest Classifier (rf_classifier)

5.2.6.1 initialization

```
rf_classifier(model_name, input_relation, response_column, predictor_columns,
             cursor, ntree=20, mtry=None, sampling_size=0.632, max_depth=5, max_breadth=
             =32, min_leaf_size=1, min_info_gain=0.0, nbins=32)
```

Trains a random forest model for classification on an input table or view.

Parameters

- **model_name:** *<str>*
The model name.
- **input_relation:** *<str>*
The table or view that contains the training samples.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.
- **ntree:** *<positive int>*, optional
Number of trees.
- **mtry:** *<positive int>*, optional
A positive integer number that indicates the number of features to be considered at the split of a tree node.
- **sampling_size:** *<float in [0,1]>*, optional
A number that indicates what portion of the input data set will randomly be picked for training each tree.
- **max_depth:** *<int in [1,100]>*, optional
A positive integer number that specifies the maximum depth for growing each tree.
- **max_breadth:** *<int in [1,1e9]>*, optional
A positive integer number that specifies the maximum number of leaf nodes a tree in the forest can have.
- **min_leaf_size:** *<int in [1,1e6]>*, optional
A positive integer number that specifies the minimum samples each branch must have after splitting a node. A split that causes fewer remaining samples will be discarded.
- **min_info_gain:** *<float in [0,1]>*, optional
A non-negative number. Any split with information gain less than this threshold will be discarded.
- **nbins:** *<int in [2,1000]>*, optional
A positive integer number that indicates the number of bins to use for continuous features.

Example

```

1 from vertica_ml_python import rf_classifier

3 rf=rf_classifier(model_name="rf_titanic", input_relation="train_titanic067",
    response_column="survived", predictor_columns=["age","sex","family_size","
    embarked","pclass","name"], cursor=cur)

5 #Output
model_type='rf_classifier'
7 model_name='rf_titanic'
input_relation='train_titanic067'
9 response_column='survived'
predictor_columns='age,sex,family_size,embarked,pclass,name'
11 tree_count: 20
rejected_row_count: 2
13 accepted_row_count: 897
column                                type
15 age                                float
sex                                char or varchar
17 family_size                        int
embarked                            char or varchar
19 pclass                            int
name                                char or varchar

```

5.2.6.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.
- **model_type:** Model type (="rf_classifier").
- **model_category:** Model category (="binomial"/"multinomial").

5.2.6.3 methods

```

rf_classifier.accuracy(threshold=0.5, input_class=None)
2 # Returns the model accuracy

4 rf_classifier.add_to_rvd(rvd, name="rf_classifier_pred"+str(np.random.randint
    (10000)), prediction_type='response', input_class=None)
# Add the prediction to the RVD
6 # rvd = the corresponding RVD

```

```

# name = RVC name
8 # prediction_type = response (default) | probability
# input_class = The class used when prediction_type="probability"
10
rf_classifier.auc(input_class=None)
12 # Returns the model auc

rf_classifier.confusion_matrix(threshold=0.5, input_class=None)
14 # Returns the model confusion matrix

rf_classifier.details()
16 # Returns the model details.

rf_classifier.error_rate(threshold=0.5, input_class=None)
20 # Returns the model error rate.

rf_classifier.features_importance(show=True)
22 # /\ Coming Soon: Returns the model features importance.

rf_classifier.lift_table(num_bins=100, color=["dodgerblue", "#444444"], show=
24     True, input_class=None)
# Draw the model lift table.

rf_classifier.logloss()
28 # Returns the model logloss.

rf_classifier.parameter_value(parameter_name="*", show=True)
32 # Returns the parameter value
# parameter_name = tree_count | rejected_row_count | accepted_row_count | *
# show = Print the parameter in the Terminal

rf_classifier.roc(num_bins=100, color=["dodgerblue", "#444444"], show=True,
36     input_class=None)
# Draw the model roc.

rf_classifier.tree(n=0)
40 # Print the Tree with the corresponding ID.

```

5.2.7 Random Forest Regressor (rf_regressor)

5.2.7.1 initialization

```

1 rf_regressor(model_name, input_relation, response_column, predictor_columns,
    cursor, ntree=20, mtry=None, sampling_size=0.632, max_depth=5, max_breadth
    =32, min_leaf_size=1, min_info_gain=0.0, nbins=32)

```

Trains a random forest model for regression on an input table or view.

Parameters

- **model_name:** *<str>*
The model name.
- **input_relation:** *<str>*
The table or view that contains the training samples.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.
- **ntree:** *<positive int>*, optional
Number of trees.
- **mtry:** *<positive int>*, optional
A positive integer number that indicates the number of features to be considered at the split of a tree node.
- **sampling_size:** *<float in [0,1]>*, optional
A number that indicates what portion of the input data set will randomly be picked for training each tree.
- **max_depth:** *<int in [1,100]>*, optional
A positive integer number that specifies the maximum depth for growing each tree.
- **max_breadth:** *<int in [1,1e9]>*, optional
A positive integer number that specifies the maximum number of leaf nodes a tree in the forest can have.
- **min_leaf_size:** *<int in [1,1e6]>*, optional
A positive integer number that specifies the minimum samples each branch must have after splitting a node. A split that causes fewer remaining samples will be discarded.
- **min_info_gain:** *<float in [0,1]>*, optional
A non-negative number. Any split with information gain less than this threshold will be discarded.
- **nbins:** *<int in [2,1000]>*, optional
A positive integer number that indicates the number of bins to use for continuous features.

Example

```
1 from vertica_ml_python import rf_regressor
3 rf_basketball=rf_regressor(model_name="rf_basketball", input_relation="
    basketball_temp", response_column="percent_succesful_field_goals",
    predictor_columns=["weight_pounds"], cursor=cur)
4 print(rf_basketball)
5
6 #Output
7 model_type='rf_regressor'
```

```

model_name='rf_basketball'
9 input_relation='basketball_temp'
response_column='percent_successful_field_goals'
11 predictor_columns='weight_pounds'
tree_count: 20
13 rejected_row_count: 0
accepted_row_count: 54
15 column          type
weight_pounds     float

```

5.2.7.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.
- **model_type:** Model type (="rf_regressor").
- **model_category:** Model category (="regression").

5.2.7.3 methods

```

rf_regressor.add_to_rvd(rvd, name="rf_regressor_pred"+str(np.random.randint
(10000)))
2 # Add the prediction to the RVD
# rvd = the corresponding RVD
4 # name = RVC name

6 rf_regressor.details()
# Returns the model details.
8
rf_regressor.features_importance(show=True)
10 # /\ Coming Soon: Returns the model features importance.

12 rf_regressor.metrics()
# Returns the model metrics.
14
rf_regressor.mse()
16 # Returns the model mse.

18 rf_regressor.parameter_value(parameter_name="*", show=True)
# Returns the parameter value
20 # parameter_name = tree_count | rejected_row_count | accepted_row_count | *

```

```

# show = Print the parameter in the Terminal
rf_regressor.rsquared()
# Returns the model rsquared.
rf_regressor.tree(n=0)
# Print the Tree with the corresponding ID.

```

5.2.8 SVM Classifier (svm_classifier)

5.2.8.1 initialization

```

svm_classifier(model_name, input_relation, response_column, predictor_columns,
               cursor, C=1.0, epsilon=1e-3, max_iterations=100)

```

Trains the SVM model on an input table or view.

Parameters

- **model_name:** *<str>*
The model name.
- **input_relation:** *<str>*
The table or view that contains the training samples.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.
- **C:** *<positive float>*, optional
Sets the weight for misclassification cost. The algorithm minimizes the regularization cost and the misclassification cost.
- **epsilon:** *<positive float>*, optional
Used to control accuracy.
- **max_iterations:** *<positive int>*, optional
Determines the maximum number of iterations that the algorithm performs before achieving the specified accuracy result.

Example

```

from vertica_ml_python import svm_classifier

svm_iris=svm_classifier(model_name="svm_iris", input_relation="iris_temp",
                        response_column="Species_Iris_setosa", predictor_columns=["PetalLengthCm",
                                         "SepalLengthCm",], cursor=cur)

```



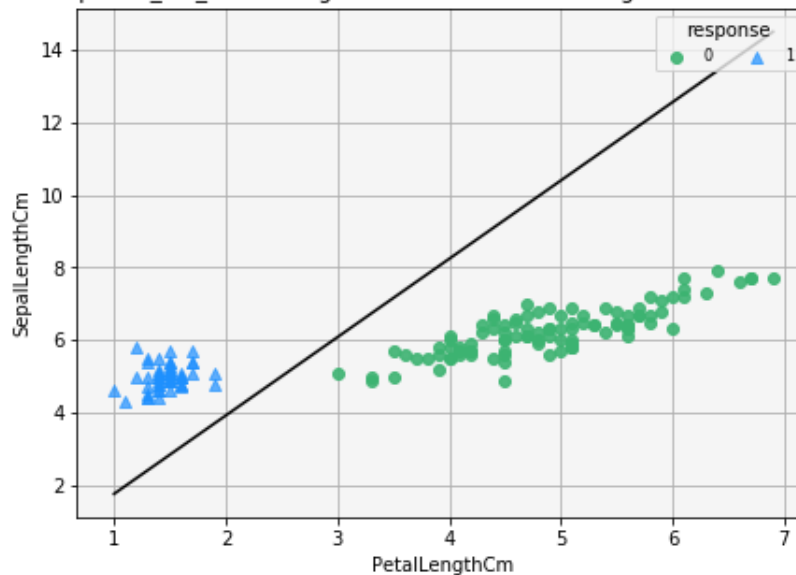
```

print(svm_iris)
5 svm_iris.plot()

7 #Output
model_type='svm_classifier'
9 model_name='svm_iris'
input_relation='iris_temp'
11 response_column='Species_Iris_setosa'
predictor_columns='PetalLengthCm,SepalLengthCm'
13 iteration_count: 6
rejected_row_count: 0
15 accepted_row_count: 150
Parameters:
17
               coefficient
Intercept      0.22005691793408
19 petallengthcm -1.21492250709494
sepalengthcm    0.563205058441283

```

svm_classifier: Species_Iris_setosa=sign(0.22+(-1.215)*PetalLengthCm+(0.563)*SepalLengthCm)



5.2.8.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.

- **model_type**: Model type (="svm_classifier").
- **model_category**: Model category (="binomial").

5.2.8.3 methods

```

svm_classifier.accuracy(threshold=0.5)
2 # Returns the model accuracy

4 svm_classifier.add_to_rvd(rvd, name="svm_classifier_pred"+str(np.random.
    randint(10000)))
    # Add the prediction to the RVD
6 # rvd = the corresponding RVD
    # name = RVC name

8
svm_classifier.auc()
10 # Returns the model auc

12 svm_classifier.confusion_matrix()
    # Returns the model confusion matrix

14
svm_classifier.details()
16 # Returns the model details.

18 svm_classifier.error_rate()
    # Returns the model error rate.

20
svm_classifier.features_importance(show=True, with_intercept=False)
22 # Returns the model features importance.

24 svm_classifier.lift_table(num_bins=100, color=["dodgerblue", "#444444"], show=
    True)
    # Draw model lift table.

26
svm_classifier.logloss()
28 # Returns the model logloss.

30 svm_classifier.parameter_value(parameter_name="*", show=True)
    # Returns the parameter value
32 # parameter_name = iteration_count | rejected_row_count | accepted_row_count |
    *
    # show = Print the parameter in the Terminal

34
svm_classifier.plot(marker=["o", "^"], color=None, projection=None,
    max_nb_points=None)
36 # Plot the logistic regression regression using matplotlib
    # marker = [0_marker, 1_marker]

```

```

38 # color = [0_color, 1_color, logit_color]
# projection = project the plan in a smaller space
40 # max_nb_points = maximum number of points in the graph

42 svm_classifier.roc(color=["dodgerblue", "#444444"], show=True)
# Draw model roc.

```

5.2.9 SVM Regressor (svm_regressor)

5.2.9.1 initialization

```

1 svm_regressor(model_name, input_relation, response_column, predictor_columns,
  cursor, C=1.0, epsilon=1e-3, max_iterations=100)

```

Trains the SVM model on an input table or view.

Parameters

- **model_name:** *<str>*
The model name.
- **input_relation:** *<str>*
The table or view that contains the training samples.
- **response_column:** *<list of str>*
The name of the column in input_relation that represents the dependent variable.
- **predictor_columns:** *<list of str>*
The list of the columns in the input_relation that represent the independent variables for the model.
- **cursor:** *<object>*
The database cursor.
- **C:** *<positive float>*, optional
Sets the weight for misclassification cost. The algorithm minimizes the regularization cost and the misclassification cost.
- **epsilon:** *<positive float>*, optional
Used to control accuracy.
- **max_iterations:** *<positive int>*, optional
Determines the maximum number of iterations that the algorithm performs before achieving the specified accuracy result.

Example

```

1 from vertica_ml_python import svm_regressor

3 svm_basketball=svm_regressor(model_name="svm_reg_basketball", input_relation="
  basketball", response_column="percent_successful_field_goals",
  predictor_columns=["weight_pounds"], cursor=cur)

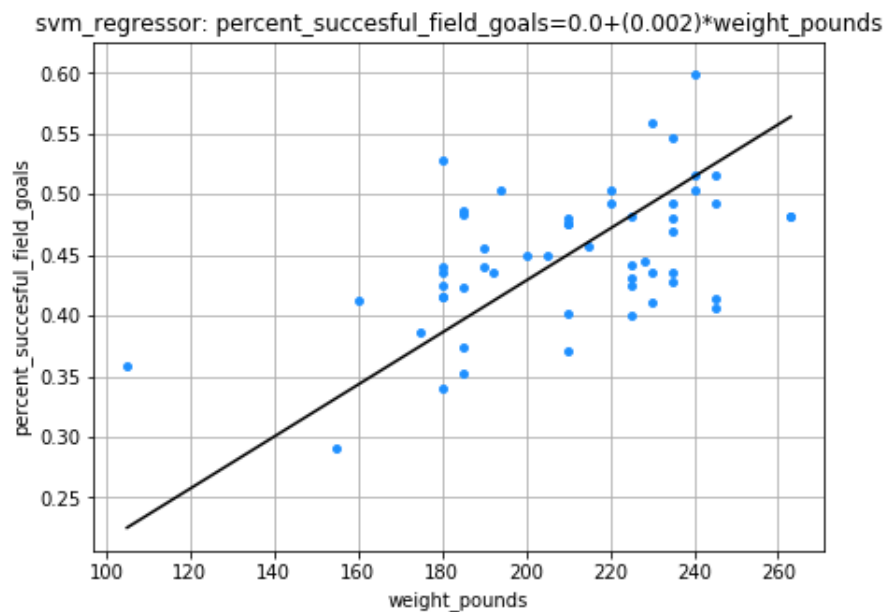
```

```

print(svm_basketball)
5 svm_basketball.plot()

7 #Output
model_type='svm_regressor'
9 model_name='svm_reg_basketball'
input_relation='basketball'
11 response_column='percent_successful_field_goals'
predictor_columns='weight_pounds'
13 iteration_count: 39
rejected_row_count: 0
15 accepted_row_count: 54
Parameters:
17
                                coefficient
Intercept                      1.06783202914075e-05
19 weight_pounds                 0.00214424218188763

```



5.2.9.2 attributes

- **cursor:** Database cursor.
- **model_name:** Model name.
- **input_relation:** Input Relation.
- **response_column:** Response Column.
- **predictor_columns:** Predictor Columns.
- **model_type:** Model type (="svm_regressor").
- **model_category:** Model category (="regression").

5.2.9.3 methods

```
1 svm_regressor.add_to_rvd(rvd, name="svm_regressor_pred"+str(np.random.randint
    (10000)))
    # Add the prediction to the RVD
3 # rvd = the corresponding RVD
    # name = RVC name
5
    svm_regressor.details()
7 # Returns the model details.

9 svm_regressor.features_importance(show=True)
    # Returns the model features importance.

11
    svm_regressor.metrics()
13 # Returns the model metrics.

15 svm_regressor.mse()
    # Returns the model mse.

17
    svm_regressor.parameter_value(parameter_name="*", show=True)
19 # Returns the parameter value
    # parameter_name = iteration_count | rejected_row_count | accepted_row_count |
        *
21 # show = Print the parameter in the Terminal

23 svm_regressor.plot(color=None, projection=None, max_nb_points=1000)
    # Plot the svm regression using matplotlib
25 # color = [points_color, plan_color]
    # projection = project the plan in a smaller space
27 # max_nb_points = maximum number of points in the graph

29 svm_regressor.rsquared()
    # Returns the model rsquared.
```