vertica-ml-python1.0 Documentation

Flexible as Python, Fast and Scalable as Vertica

Ouali Badr

November 27, 2019

Executive Summary

This documentation explains the `vertica-ml-python` API by detailing all the functions and by 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 objects similar to the ones available in `pandas` and `sklearn` with the power of a columnar oriented analytic database: Vertica.

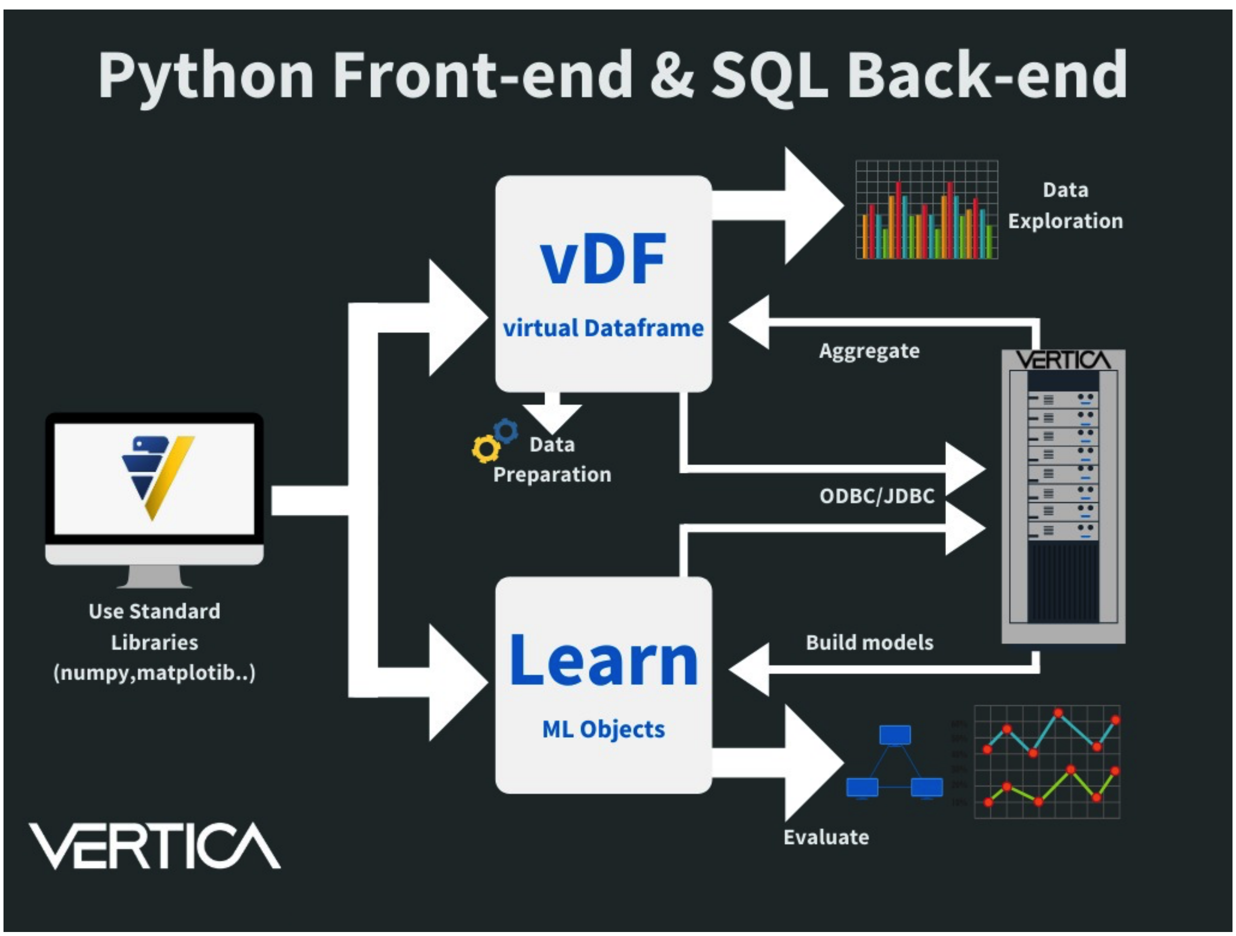
`vertica-ml-python` allows users to use the vDataframe (Virtual Dataframe). This object keeps in memory all the users modifications in order to use optimised SQL queries to compute all the necessary aggregations. Thanks to this object, the initial relation is intact and will never be modified. The purpose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

What contains vertica-ml-python?

This API contains many functions for:

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

`vertica-ml-python` helps to explore, preprocess and clean the data without changing the initial relation. It uses scalable Machine Learning Algorithms such as Logistic Regression, Random Forest, SVM and much more... It allows also to evaluate and to optimise models (Classification/Regression Reports, ROC/PRC curves, Parameters tuning...).



Contents

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

Louis Pasteur

1 About

The 'Big Data' (Tb of data) is now one of the main topics in the Data Science World. Data Scientists are now very important for any organisation. Becoming Data-Driven is mandatory to survive. Vertica is the first real analytic columnar Database and is still the fastest in the market. However, SQL is not enough flexible to be very popular for Data Scientists. Python flexibility is priceless and provides to any user a very nice experience. The level of abstraction is so high that it is enough to think about a function to notice that it already exists. Many Data Science APIs were created during the last 15 years and were directly adopted by the Data Science community (examples: pandas and scikit-learn). However, Python is only working in-memory for a single node process. Even if some famous highly distributed programming languages exist to face this challenge, they are still in-memory and most of the time they can not process on all the data. Besides, moving the data can become very expensive. Data Scientists must also find a way to deploy their data preparation and their models. We are far away from easiness and the entire process can become time expensive.

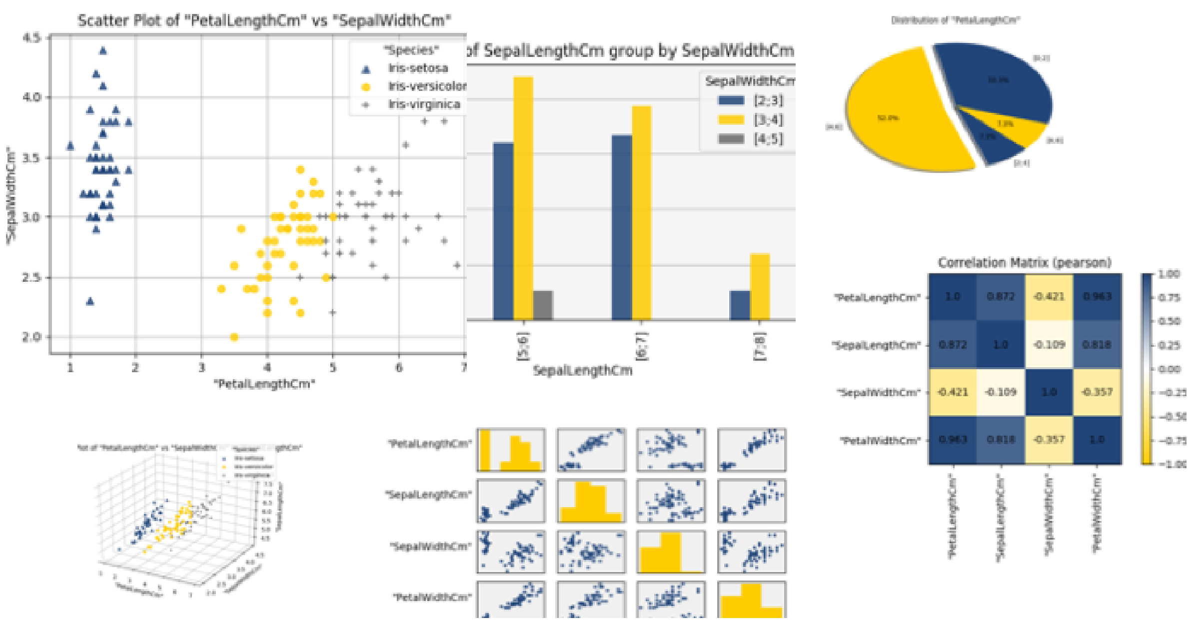
The idea behind VERTICA ML PYTHON is simple: Combining the Scalability of VERTICA with the Flexibility of Python to give to the community what they need *Bringing the logic to the data and not the opposite*. This version 1.0 is the work of 3 years of new ideas and improvement. I couldn't have my current skills at the beginning of my journey in the Database World. I needed time and great managers to successfully release this version. I want to thank:

- Yassine Faihe for hiring me as intern to start this project. He is a very pragmatic element who exactly knows in advance if we are following the correct roadmap. Without him, this project would have never seen the light.
- Fouad Teban who is the biggest project supporter. He always trusted the project and provided me many customers to help me on the testing.
- Eugenia Moreno who is a great manager and who knows how to efficiently support my ideas and how to guide me to the correct opportunities.

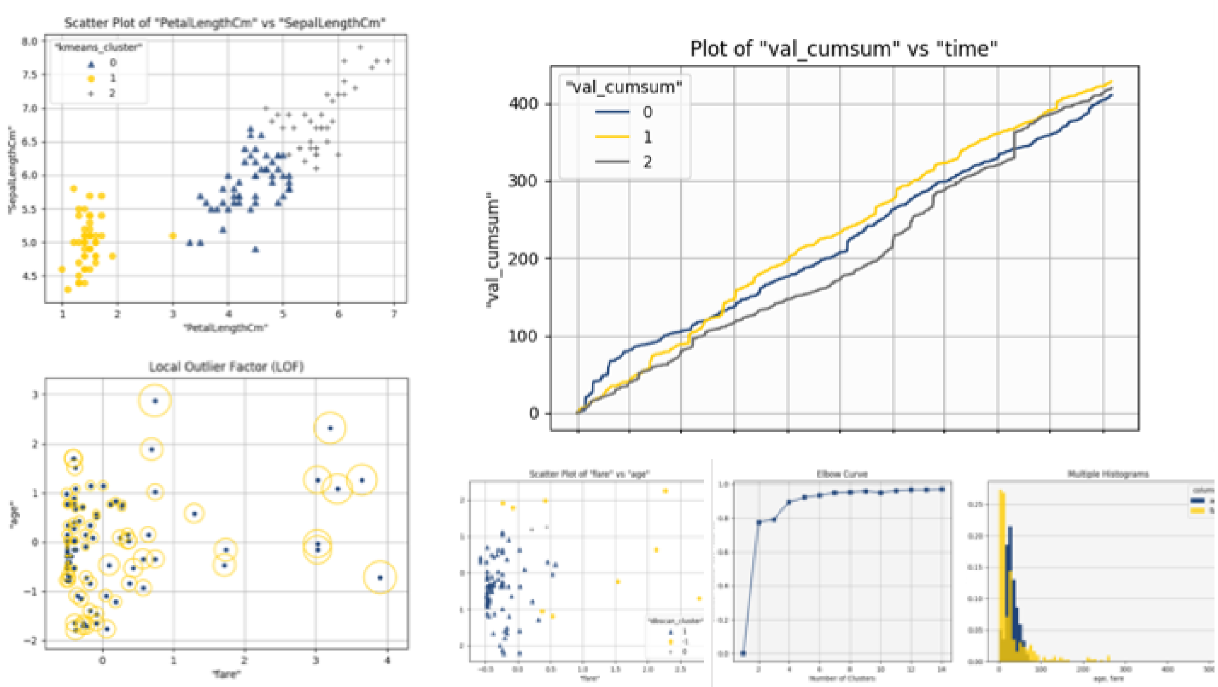
Badr Ouali - Author of the VERTICA ML PYTHON API

2 Features

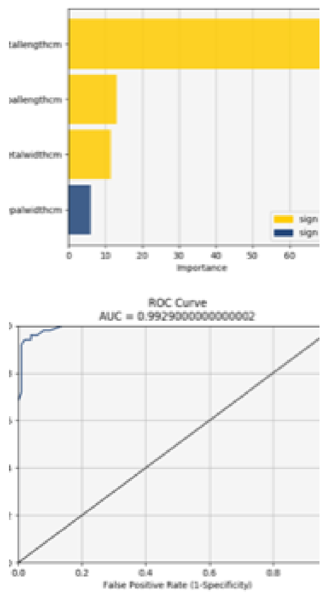
Vertica ML Python is the perfect combination between Vertica and Python. It uses Vertica Scalability and Python Flexibility to help any Data Scientist achieving his goals by bringing the logic to the data and not the opposite. With VERTICA ML PYTHON, start your journey with easy Data Exploration.



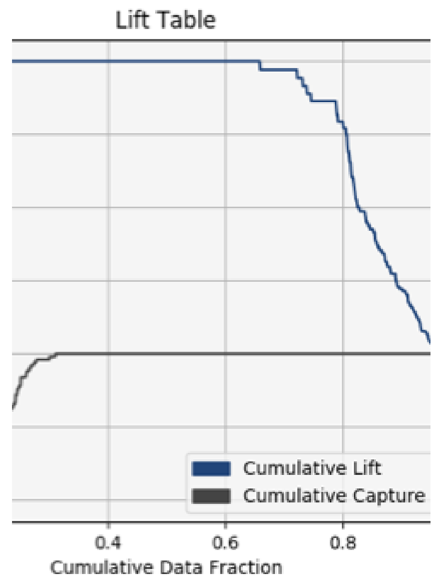
Find patterns that you don't know and Detect Anomalies.



Prepare your data easily and build a model with Highly Scalable Vertica ML. Evaluate your model and try to create the most efficient and performant one.



	value
auc	0.9911000000000003
prc_auc	0.9637269964565092
accuracy	0.713333333333333
log_loss	0.232156667020165
precision	1.0
recall	0.5376344086021505
f1-score	0.6993006993006993
mcc	0.5535807194106618
informedness	0.5376344086021505
markedness	0.5699999999999998
csi	0.5376344086021505



Everything will happen in one place and where it should be: your Database. Without modifying anything but using the speed of Vertica to aggregate the data.

3 Quick Start

Install the library using the `pip` command:

```
root@ubuntu:~$ pip3 install vertica_ml_python
```

Install `vertica_python` or `pyodbc` to build a DB cursor:

```
root@ubuntu:~$ pip3 install vertica_python
```

Create a vertica cursor

```
from vertica_ml_python.utilities import vertica_cursor
cur = vertica_cursor("VerticaDSN")
```

Create the Virtual Dataframe of your relation:

```
from vertica_ml_python import vDataframe
vdf = vDataframe("my_relation", cursor = cur)
```

If you don't have data to play, you can easily load well known datasets

```
from vertica_ml_python.learn.datasets import load_titanic
vdf = load_titanic(cursor = cur)
```

You can now play with the data...

```
vdf.describe()

# Output
                   min        25%        50%        75%    \\
age               0.33       21.0       28.0       39.0    \\
body               1.0      79.25      160.5      257.5    \\
fare               0.0     7.8958    14.4542    31.3875    \\
parch              0.0        0.0        0.0        0.0    \\
pclass             1.0        1.0        3.0        3.0    \\
sibsp              0.0        0.0        0.0        1.0    \\
survived           0.0        0.0        0.0        1.0    \\
                       max     unique
age                   80.0         96
body                 328.0        118
fare              512.3292        277
parch                  9.0          8
pclass                 3.0          3
sibsp                  8.0          7
survived               1.0          2
```

You can also print the SQL code generation using the `sql_on_off` method.

```
vdf.sql_on_off()
vdf.describe()

# Output
$ Compute the descriptive statistics of all the numerical columns $

SELECT SUMMARIZE_NUMCOL("age","body","survived","pclass","parch","fare","sibsp
    ") OVER ()
FROM
  (SELECT "age" AS "age",
          "body" AS "body",
          "survived" AS "survived",
          "ticket" AS "ticket",
          "home.dest" AS "home.dest",
          "cabin" AS "cabin",
          "sex" AS "sex",
          "pclass" AS "pclass",
          "embarked" AS "embarked",
          "parch" AS "parch",
          "fare" AS "fare",
```

```
            "name" AS "name",
            "boat" AS "boat",
            "sibsp" AS "sibsp"
    FROM public.titanic) final_table
```

With Vertica ML Python, it is now possible to solve a ML problem with four lines of code (two if we don't consider the libraries loading).

```
from vertica_ml_python.learn.model_selection import cross_validate
from vertica_ml_python.learn.linear_model import LogisticRegression

# Data Preparation
vdf["sex"].label_encode().parent["boat"].fillna(method = "0ifnull").parent["
    name"].str_extract(' ([A-Za-z]+)\.').parent.eval("family_size", expr = "
    parch + sibsp + 1").drop(columns = ["cabin", "body", "ticket", "home.dest"
    ])["fare"].fill_outliers().parent.fillna().to_db("titanic_clean")

# Model Evaluation
cross_validate(RandomForestClassifier("logit_titanic", cur, max_leaf_nodes =
    100, n_estimators = 30), "titanic_clean", ["age", "family_size", "sex", "
    pclass", "fare", "boat"], "survived", cutoff = 0.35)

# Output
                            auc                  prc_auc   \\
1-fold      0.9877114427860691    0.9530465915039339   \\
2-fold      0.9965555014605642    0.7676485351425721   \\
3-fold      0.9927239216549301    0.6419135521132449   \\
avg             0.992330288634        0.787536226253   \\
std           0.00362128464093         0.12779562393   \\
                      accuracy                log_loss   \\
1-fold      0.971291866028708    0.0502052541223871   \\
2-fold      0.983253588516746    0.0298167751798457   \\
3-fold      0.964824120603015    0.0392745694400433   \\
avg            0.973123191716       0.0397655329141   \\
std           0.0076344236729      0.00833079837099   \\
                     precision                  recall   \\
1-fold                    0.96                  0.96   \\
2-fold      0.9556962025316456                   1.0   \\
3-fold      0.9647887323943662    0.9383561643835616   \\
avg             0.960161644975        0.966118721461   \\
std           0.00371376912311        0.025535200301   \\
                      f1-score                     mcc   \\
1-fold      0.9687259282082884    0.9376119402985075   \\
2-fold      0.9867172675521821    0.9646971010878469   \\
3-fold      0.9588020287309097    0.9240569687684576   \\
avg              0.97141507483        0.942122003385   \\
std            0.0115538960753       0.0168949813163   \\
                  informedness              markedness   \\
```

```
1-fold      0.9376119402985075    0.9376119402985075   \\
2-fold      0.9737827715355807    0.9556962025316456   \\
3-fold      0.9185148945422918    0.9296324823943662   \\
avg             0.943303202125        0.940980208408   \\
std            0.0229190954261       0.0109037699717   \\
                           csi
1-fold      0.9230769230769231
2-fold      0.9556962025316456
3-fold      0.9072847682119205
avg             0.928685964607
std            0.0201579224026
```

Happy Playing !

4 Prerequires

4.1 Python Version

`vertica-ml-python` works with at least:

- **Vertica:** 9.1 (with previous versions, some functions and algorithms may not be available)
- **Python:** 3.6

4.2 Standard Libraries

`vertica-ml-python` library is only using the standard Python libraries such as `matplotlib`, `numpy`... Other libraries can be used as `anytree` for tree visualization or `sqlparse` for SQL indentation but they are optional.

4.3 Installation

To install `vertica-ml-python`, you can use the pip command:

```
root@ubuntu:~$ pip3 install vertica_ml_python
```

You can also drag and drop the `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 the usual Python syntax.

```
# to import the vDataframe
from vertica_ml_python import vDataframe
# to import the Logistic Regression
from vertica_ml_python.learn.linear_model import LogisticRegression
```

Everything is well detailed in the following documentation.

4.4 Connection to the Database

This step is useless if `vertica-python` or `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 vDataframe using directly the DSN (`dsn` parameter of the vDataframe).

4.4.1 ODBC

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

```
#
# vertica_python
#
import vertica_python

# Connection using all the DSN information
conn_info = {'host': "10.211.55.14", 'port': 5433, 'user': "dbadmin", '
    password': "XxX", 'database': "testdb"}
cur = vertica_python.connect(** conn_info).cursor()

# Connection using directly the DSN
from vertica_ml_python.utilities import to_vertica_python_format # This
    function will parse the odbc.ini file
dsn = "VerticaDSN"
cur = vertica_python.connect(** to_vertica_python_format(dsn)).cursor()

#
# pyodbc
#
import pyodbc

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

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

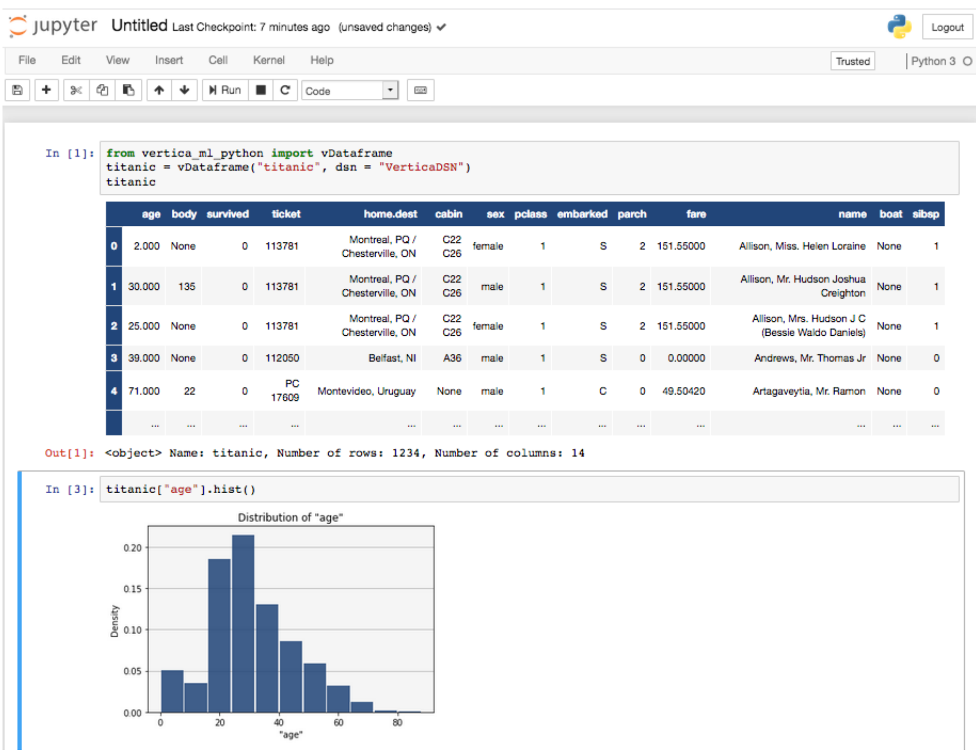
4.4.2 JDBC

The user can also use a JDBC connection to the Vertica Database.

```python
import jaydebeapi

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

5 Jupyter

Jupyter offers a really beautiful interface to use `vertica-ml-python`.



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

6 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 comparaison) with 128Gb of memory.

6.1 Limitations

6.1.1 vertica-ml-python

`vertica-ml-python` has no limitation as it uses Vertica to compute all the aggregations it needs. If we want to increase the speed, we can increase the number of nodes or do query tuning by building the appropriate projections.

6.1.2 pandas + scikit

`pandas` and `scikit` have real limitations. They both need to load data in memory and we can not increase it indefinitely.

6.2 Time to load the data

6.2.1 vertica-ml-python

If the data is inside a Vertica database, no operation is needed. In the other cases, we need to transfer the data to Vertica. As we are trying to solve the main 'Big Data' issues, the purpose of this library is to avoid moving the data and to bring the logic to the data (not the opposite). It is highly recommended to already have the data in Vertica (even if the time to create the table is still lower than the one to create the `pandas.Dataframe`).

6.2.2 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.

```
# Using 35M of rows (the whole dataset) - Less than 4Gb
start_time = time.time()
expedia_df = pandas.read_sql("SELECT * FROM expedia_train", conn)
print(time.time() - start_time)

# Output
1135.22843092306093

# Using 10M of rows - Less than 1Gb
start_time = time.time()
expedia_df = pandas.read_sql("SELECT * FROM expedia_train LIMIT 10000000",
    conn)
print(time.time() - start_time)

# Output
281.51524472236633

# Using 1M of rows - Less than 100Mb
start_time = time.time()
expedia_df = pandas.read_sql("SELECT * FROM expedia_train LIMIT 1000000",conn)
print(time.time() - start_time)

# Output
42.53745484352112
```

We can notice the complexity of in-memory processing when reaching Big Data. Besides, the limitation is the power of the single-node machine you are using.

6.3 Object Size

6.3.1 vertica-ml-python

The size of a vDataframe will never exceed some kilo bytes.

```
print(expedia.memory_usage())

# Output
5566
```

6.3.2 pandas

The size of a `pandas.Dataframe` will be proportional to the dataset volume.

```
print(sys.getsizeof(expedia_df))

# 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.Dataframe` on very huge dataset, he needs a very powerful machine. Knowing that it is hard to have today a personal machine exceeding 32GB of RAM, the limitation is obvious. These in-memory libraries can only process small datasets (less than 10GB) which is not nowadays 'Big Data' (we can consider the 'Big Data' border as 1TB).

6.4 Time to execute some queries

6.4.1 vertica-ml-python

Let's start with `vertica-ml-python`:

```
# Using 35M of rows (the whole dataset)
#
# describe
#
start_time = time.time()
expedia.describe()
print(time.time() - start_time)

# Output
94.45896601676941


#
# categorical hist
#
start_time = time.time()
```

```
expedia["is_mobile"].hist()
print(time.time() - start_time)

# Output
0.480072021484375

# Using 10M of rows
#
# describe
#
start_time=time.time()
expedia.describe()
print(time.time() - start_time)

# Output
28.198142528533936

#
# categorical hist
#
start_time = time.time()
expedia["is_mobile"].hist()
print(time.time() - start_time)

# Output
0.4742517471313477

# Using 1M of rows
#
# describe
#
start_time = time.time()
expedia.describe()
print(time.time() - start_time)

# Output
2.8560750484466553

#
# categorical hist
#
start_time = time.time()
expedia["is_mobile"].hist()
print(time.time() - start_time)

# Output
0.3278229236602783
```

6.4.2 pandas

And now `pandas`:

```
# Using 35M of rows (the whole dataset)
#
# describe
#
start_time = time.time()
expedia_df.describe()
print(time.time() - start_time)

# Output
75.72856092453003

#
# categorical hist
#
start_time = time.time()
expedia_df["is_mobile"].hist()
print(time.time() - start_time)

# Output
3.210484266281128

# Using 10M of rows
#
# describe
#
start_time = time.time()
expedia_df.describe()
print(time.time() - start_time)

# Output
20.789332389831543

#
# categorical hist
#
start_time = time.time()
expedia_df["is_mobile"].hist()
print(time.time() - start_time)

# Output
0.36867237091064453
```

```
# Using 1M of rows
#
# describe
#
start_time = time.time()
expedia_df.describe()
print(time.time() - start_time)

# Output
1.2540433406829834

#
# categorical hist
#
start_time = time.time()
expedia_df["is_mobile"].hist()
print(time.time() - start_time)

# Output
0.08945250511169434
```

6.5 Conclusion

At the end, we have more or less the same execution time with a little advantage to `pandas` for some functions. However, do not forget that `pandas` is in-memory processing. At the end, `vertica-ml-python` is more robust and offers more possibilities by processing the data directly where they live. Besides some functions to prepare the data will be time consuming using `pandas` whereas it will be very fast for `vertica-ml-python` as only the grammar of the different transformations is kept in memory. If the projections are correctly made, we could reduce drastically the processing time. We can also reduce the processing time by increasing the number of nodes. `vertica-ml-python` uses the Python flexibility and Vertica Scalability to help any Python user to follow the entire Data Science cycle without moving the data.

7 vertica_ml_python.vdataframe

7.1 Functions

The following functions will help the user to build a vDataframe from the `vertica-ml-python` format (.vdf) or csv files. There is also the possibility to build the vDataframe from more complex relations using the `vdf_from_relation` function.

7.1.1 read_csv

```
read_csv(
        path: str,
        cursor,
```

```
        schema: str = 'public',
        table_name: str = '',
        delimiter: str = ',',
        header_names: list = [],
        dtype: dict = {},
        null: str = '',
        enclosed_by: str = '"',
        escape: str = '\\',
        skip: int = 1,
        genSQL: bool = False,
        return_dlist: bool = False,
        parse_n_lines: int = -1)
```

Read a csv file and store it in the Vertica Database.

Parameters

- **path:** *<str>*
  Path to the csv file.
- **cursor:** *<object>*
  Database Cursor.
- **schema:** *<str>*, optional
  Schema used to store the csv file.
- **table_name:** *<str>*, optional
  Table name used to store the csv file.
- **delimiter:** *<str>*, optional
  Delimiter used to parse the file.
- **header_names:** *<list>*, optional
  List with the columns name (to use if the csv file has no header).
- **dtype:** *<dict>*, optional
  Dictionary of all the columns types (it is used if header_names is defined). It makes the loading process faster as the parser has not to identify the types.
- **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.
- **genSQL:** *<bool>*, optional
  Generate the SQL used to create the table.

- **return_dlist:** *<bool>*, optional
  Return a dictionary of the columns names and their respective types. Do not store the csv file in the Database. This parameter can be useful if we want to be sure that the parser guessed the right types.
- **parse_n_lines:** *<int>*, optional
  The parser will only parse a limited number of lines to guess the types. This parameter must be used if the file volume is big.

Returns

The Virtual Dataframe of the new relation.

Example

```
from vertica_ml_python.vdataframe import read_csv
titanic = read_csv('titanic.csv', cur)

# Output
          cabin       sex    pclass    embarked   \\
0     C22 C26    female         1           S   \\
1     C22 C26      male         1           S   \\
2     C22 C26    female         1           S   \\
3         A36      male         1           S   \\
4        None      male         1           C   \\
...       ...       ...       ...         ...   \\
     parch         fare                                 name   \\
0        2    151.55000              Allison, Miss. Helen Loraine   \\
1        2    151.55000        Allison, Mr. Hudson Joshua Creighton   \\
2        2    151.55000   Allison, Mrs. Hudson J C (Bessie Wald...   \\
3        0      0.00000                 Andrews, Mr. Thomas Jr   \\
4        0     49.50420                Artagaveytia, Mr. Ramon   \\
...    ...          ...                                      ...   \\
Name: titanic, Number of rows: 1234, Number of columns: 14
```

7.1.2 read_vdf

```
read_vdf(path: str, cursor)
```

Read a vdf file and load the corresponding vDataframe.

Parameters

- **path:** *<str>*
  Path to the csv file.
- **cursor:** *<object>*
  Database Cursor.

Returns

The Virtual Dataframe corresponding to the one described in the vdf file.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.to_vdf('titanic')
from vertica_ml_python.vdataframe import read_vdf
titanic = read_vdf('titanic.vdf', cur)

# Output
              cabin        sex    pclass    embarked    \\
0        C22 C26     female         1           S    \\
1        C22 C26       male         1           S    \\
2        C22 C26     female         1           S    \\
3            A36       male         1           S    \\
4           None       male         1           C    \\
...          ...        ...       ...         ...    \\
         parch           fare                                name    \\
0            2     151.55000            Allison, Miss. Helen Loraine    \\
1            2     151.55000      Allison, Mr. Hudson Joshua Creighton    \\
2            2     151.55000   Allison, Mrs. Hudson J C (Bessie Wald...    \\
3            0       0.00000                Andrews, Mr. Thomas Jr    \\
4            0      49.50420              Artagaveytia, Mr. Ramon    \\
...        ...           ...                                     ...    \\
Name: titanic, Number of rows: 1234, Number of columns: 14
```

7.1.3 vdf_from_relation

```
vdf_from_relation(
        relation: str,
        name: str = "VDF",
        cursor = None,
        dsn: str = "")
```

Build a vDataframe using the defined relation.

Parameters

- **relation:** *<str>*
  SQL Relation.
- **name:** *<str>*, optional
  Name of the new vDataframe.

- **cursor:** *<object>*, optional
  Database Cursor.
- **dsn:** *<str>*, optional
  Database DSN.

Returns

The vDataframe of the input relation.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 2 AS x, 1 AS y) UNION ALL (SELECT 10 AS x, 4 AS y) UNION
    ALL (SELECT 10 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

# Output
      x    y
0     2    1
1    10    4
2    10    5
Name: VDF, Number of rows: 3, Number of columns: 2
```

7.2 Virtual Dataframe

The vDataframe is a Python object which will keep in mind all the user modifications in order to use an optimised SQL query. It will send the optimised SQL query to the database which will aggregate and return the final result. vDataframe will create for each column of the relation a vColumn (Virtual Column) which will store for each column its name and its transformations. Thus, vDataframe allows to do easy data preparation and exploration without modifying the data.

vColumn and vDataframe coexist and one can not live without the other. vColumn will use the vDataframe 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 vDataframe keeps all the transformations in memory to select for each query the needed data in the main relation. Let's try to understand thanks to an example.

```
# We create the vDataframe
titanic = vDataframe('titanic', dsn = 'VerticaDSN')

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

84 elements were filtered

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

232 elements were filled

```

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

/!\\ Warning: Nothing was dropped

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

# We print all the queries used during the next executions
titanic.sql_on_off()

# We summarize our vDataframe
titanic.describe()
```

To compute the descriptives statistics, the following query was generated by our vDataframe:

```
SELECT SUMMARIZE_NUMCOL("age", "body", "passenger class", "embarked", "parch",
    "fare", "sibsp", "survived") OVER ()
FROM
  (SELECT COALESCE("age", 29.6471459694989) AS "age",
          "body" AS "body",
          "passenger class" AS "passenger class",
          "ticket" AS "ticket",
          "cabin" AS "cabin",
          "sex" AS "sex",
          "home.dest" AS "home.dest",
          DECODE("embarked", 'C', 0, 'Q', 1, 'S', 2, 3) AS "embarked",
          "parch" AS "parch",
          "fare" AS "fare",
          "name" AS "name",
          "boat" AS "boat",
          "sibsp" AS "sibsp",
          "survived" AS "survived"
   FROM
     (SELECT "age" AS "age",
             "body" AS "body",
             "passenger class" AS "passenger class",
             "ticket" AS "ticket",
             "cabin" AS "cabin",
             "sex" AS "sex",
             "home.dest" AS "home.dest",
             "embarked" AS "embarked",
             "parch" AS "parch",
             "fare" AS "fare",
             "name" AS "name",
             "boat" AS "boat",
             "sibsp" AS "sibsp",
             "survived" AS "survived"
```

```
        FROM public.titanic) t1
    WHERE fare < 100) final_table
```

The vDataframe 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 vDataframe in the Database in order to gain in efficiency (using the `to_db` method). We can also see all the modifications using the `info` method.

```
titanic.info()

#Output
The vDataframe was modified many times:
 * {Fri Nov 22 18:06:10 2019} [Filter]: 84 elements were filtered using the
   filter 'fare < 100'
 * {Fri Nov 22 18:06:26 2019} [Fillna]: 232 missing values of the vColumn '"
   age"' were filled.
 * {Fri Nov 22 18:06:54 2019} [Label Encoding]: Label Encoding was applied to
   the vColumn '"embarked"' using the following mapping:
  C => 0   Q => 1   S => 2
```

7.2.1 why Virtual Dataframe ?

As the vDataframe 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 Virtual Dataframe will never load data in-memory, all the aggregations are computed thanks to Vertica. The user feels like using `pandas` whereas the data is not stored in-memory but exactly where it should be (in the Database !).

7.2.2 initialization

```
class vDataframe(
        input_relation: str, # The relation used to build the vDataframe
        cursor = None, # The DB cursor
        dsn: str = "", # The DB DSN
        usecols: list = []) # Build the vDataframe using only specific columns
```

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

```
from vertica_ml_python import vDataframe

vdf = vDataframe(relation, cur)
```

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

```
vdf = vDataframe(relation, dsn = "VerticaDSN")
```

Using this method, the Virtual Dataframe will keep in mind the DSN name and in case of connection failure, the user can restart a connection using the `dsn_restart` method.

Example

```
titanic = vDataframe("titanic", cursor)

#Output
        age     boat    body    cabin    embarked       fare    \\
0       None       1    None    None            S   26.00000    \\
1       None      10    None    E101            Q   12.35000    \\
2       None      10    None    None            S   16.10000    \\
3       None      11    None    None            S   33.00000    \\
4       None      13    None    None            Q    7.72080    \\
5       None      13    None    None            Q    7.73330    \\
...      ...     ...     ...     ...          ...        ...    \\
--More--
Name: titanic, Number of rows: 1234, Number of columns: 14
```

7.2.3 attributes

When the Virtual Dataframe is created, it will create as many Virtual Columns as there are columns in the input relation (`columns` attribute). It will keep in mind the cursor, the relation and the DSN if this connection method is used. It will also create 6 other attributes in order to keep in mind the user modifications.

- `order_by`: how to sort the data
- `where`: all the filters.
- `query_on`: print all the queries executed by the vDataframe in the terminal.
- `time_on`: print all the queries elapsed time in the terminal.
- `history`: all the user modifications.
- `saving`: all the user saving.

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

7.2.4 methods

7.2.4.1 abs

```
vDataframe.abs(self, columns: list = [])
```

Apply the Absolute function to the selected columns.

Parameters

- **columns:** *<list>*, optional
  List of Columns. If this parameter is empty, the method will be applied to all the numerical columns.

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, -1 AS y) UNION ALL (SELECT -3 AS x, -4 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
       x      y
0      1     -1
1     -3     -4
Name: VDF, Number of rows: 2, Number of columns: 2

vdf.abs()

#Output
      x     y
0     1     1
1     3     4
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.2 aggregate / agg

```
vDataframe.aggregate(self, func: list, columns: list = [])
```

Aggregate all the different columns.

Parameters

- **func:** *<list>*
  List of the different aggregations.
- **columns:** *<list>*, optional
  List of Columns. If this parameter is empty, all the columns will be aggregated.

Returns

The `tablesample` type containing the aggregations (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.agg(["sum", "min"])

#Output
                                  sum      min
"age"                         30062.0     0.33
"body"                        19369.0      1.0
"passenger class"              2819.0      1.0
"parch"                         467.0      0.0
"fare"                       41877.3576    0.0
"sibsp"                         622.0      0.0
"survived"                      450.0      0.0
```

7.2.4.3 all

```
vDataframe.all(columns: list)
```

Aggregate the different columns by verifying if all the elements are True.

Parameters

- **columns:** *<list>*
  List of the vDataframe Columns.

Returns

The `tablesample` type containing the aggregations (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT True AS x, False AS y) UNION ALL (SELECT True AS x, True
    AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
         x         y
0     True     False
1     True      True

vdf.all(['x', 'y'])

#Output
```

```
            bool_and
"x"              1.0
"y"              0.0
```

7.2.4.4 any

```
vDataframe.any(columns: list)
```

Aggregate the different columns by verifying if at least one element is True.

Parameters

- **columns:** *<list>*
  List of the vDataframe Columns.

Returns

The `tablesample` type containing the aggregations (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT True AS x, False AS y) UNION ALL (SELECT True AS x, True
    AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
        x         y
0     True     False
1     True      True

vdf.any(['x', 'y'])

#Output
            bool_or
"x"             1.0
"y"             1.0
```

7.2.4.5 append

```
vDataframe.append(self, vdf = None, input_relation: str = "")
```

Merge the Virtual Dataframe with another one or an existing relation.

Parameters

- **vdf:** *<vDataframe>*, optional
  The Vertica Dataframe to merge with the current one.
- **input_relation:** *<str>*, optional
  The input_relation to merge with the Virtual Dataframe.

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation1 = "((SELECT 1 AS x, -1 AS y) UNION ALL (SELECT -3 AS x, -4 AS y)) z"
vdf1 = vdf_from_relation(relation1, dsn = "VerticaDSN")

#Output
       x      y
0      1     -1
1     -3     -4
Name: VDF, Number of rows: 2, Number of columns: 2

relation2 = "((SELECT 9 AS x, -3 AS y) UNION ALL (SELECT -1 AS x, 2 AS y)) z"
vdf2 = vdf_from_relation(relation2, dsn = "VerticaDSN")

#Output
       x      y
0      9     -3
1     -1      2
Name: VDF, Number of rows: 2, Number of columns: 2

vdf1.append(vdf = vdf2)

#Output
       x      y
0      1     -1
1     -3     -4
2      9     -3
3     -1      2
Name: VDF, Number of rows: 4, Number of columns: 2
```

7.2.4.6 apply

```
vDataframe.apply(self, func: dict)
```

Apply the input functions to the vDataframe columns.

Parameters

- **func:** *<dict>*
  Dictionary of the different columns with the functions to apply. The function's variable must be written using flower brackets '{}' (Example: EXP({}))

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, 4 AS y) UNION ALL (SELECT 2 AS x, 9 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
     x     y
0    1     4
1    2     9
Name: VDF, Number of rows: 2, Number of columns: 2

vdf.apply({'x': 'EXP({})', 'y': 'SQRT({})'})

#Output
                    x        y
0     2.71828182845905     2.0
1     7.38905609893065     3.0
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.7 applymap

```
vDataframe.applymap(self, func: str, numeric_only: bool = True)
```

Apply the corresponding function to all the vDataframe columns.

Parameters

- **func:** *<str>*
  The function to apply. The function's variable must be written using flower brackets '{}' (Example: EXP({}))

- **numeric_only:** *<bool>*, optional
  Bool to apply the function only on the vDataframe numerical columns.

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, 4 AS y) UNION ALL (SELECT 2 AS x, 9 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
     x     y
0    1     4
1    2     9
Name: VDF, Number of rows: 2, Number of columns: 2

vdf.applymap('EXP({})')

#Output
                         x                       y
0     2.71828182845905     54.5981500331442
1     7.38905609893065     8103.08392757538
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.8 asfreq

```
vDataframe.asfreq(self, ts: str, rule: str, method: dict, by: list = [])
```

Slice the time series using a specific rules and a method for each column.

Parameters

- **ts:** *<str>*
  The time series column to use.
- **rule:** *<str>*
  The slice rule to use (it must be an interval).
- **method:** *<dict>*
  A dictionary of different columns to use and the interpolation method to use. The method can be in {bfill (Back Propagation), ffill (First Element Propagation), linear (Linear Interpolation)}
- **by:** *<list>*, optional
  The columns used to group the elements.

Returns

The Sliced Virtual Dataframe.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.sort(["id", "time"])

#Output
                        time            val     id
0     2014-01-01 11:00:00     0.0290000      0
1     2014-01-01 13:45:00     0.2770000      0
2     2014-01-02 10:45:00     0.3210000      0
3     2014-01-02 11:15:00     0.3050000      0
4     2014-01-02 13:45:00     0.3580000      0
...                      ...            ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3

sm.asfreq(ts = "time", rule = "5 minutes", method = {"val": "linear"}, by = ["
    id"])

#Output
                        time     id                     val
0     2014-01-01 11:00:00      0                   0.029
1     2014-01-01 12:00:00      0     0.119181818181818
2     2014-01-01 13:00:00      0     0.209363636363636
3     2014-01-01 14:00:00      0      0.27752380952381
4     2014-01-01 15:00:00      0     0.279619047619048
...                      ...    ...                     ...
Name: smart_meters, Number of rows: 148189, Number of columns: 3
```

7.2.4.9 astype

```
vDataframe.astype(self, dtype: dict)
```

Convert the different columns to the input types.

Parameters

- **dtype:** *<dict>*
  A dictionary of different columns and the types to use for the conversion.

Returns

The Virtual Dataframe.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT True AS x, 4 AS y) UNION ALL (SELECT False AS x, 9 AS y))
    z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
          x     y
0      True     4
1     False     9
Name: VDF, Number of rows: 2, Number of columns: 2

vdf.astype({"x": "int"})

#Output
     x     y
0    1     4
1    0     9
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.10 at_time

```
vDataframe.at_time(self, ts: str, time: str)
```

Filter the elements of the vDataframe by considering only the events happening at the exact time.

Parameters

- **ts:** *<str>*
  The time series column to use.
- **time:** *<str>*
  The time to consider.

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)

```

```
#Output
                        time             val     id
0     2014-01-01 01:15:00     0.0370000      2
1     2014-01-01 02:30:00     0.0800000      5
2     2014-01-01 03:00:00     0.0810000      1
3     2014-01-01 05:00:00     1.4890000      3
4     2014-01-01 06:00:00     0.0720000      5
...                      ...             ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3

sm.at_time(ts = "time", time = "12:00")

#Output
                        time             val     id
0     2014-01-04 12:00:00     0.0760000      2
1     2014-01-09 12:00:00     0.1610000      6
2     2014-01-15 12:00:00     0.0490000      9
3     2014-01-19 12:00:00     0.0200000      4
4     2014-01-21 12:00:00     0.1440000      7
...                      ...             ...    ...
Name: smart_meters, Number of rows: 140, Number of columns: 3
```

7.2.4.11 avg / mean

```
vDataframe.avg(columns: list = [])
```

Aggregate the different columns by computing the average of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe Columns. If this parameter is empty, all the numerical columns of the vDataframe will be aggregated.

Returns

The `tablesample` type containing the avg (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.avg()

#Output
```

```
                              avg
"age"             30.1524573721163
"body"             164.14406779661
"survived"       0.364667747163695
"pclass"          2.28444084278768
"parch"          0.378444084278768
"fare"            33.9637936739659
"sibsp"          0.504051863857374
```

7.2.4.12 bar

```
Dataframe.bar(
      self,
      columns: list,
      method: str = "density",
      of: str = "",
      max_cardinality: tuple = (6, 6),
      h: tuple = (None, None),
      limit_distinct_elements: int = 200,
      hist_type: str = "auto")
```

Draw the corresponding variables Bar Chart.

Parameters

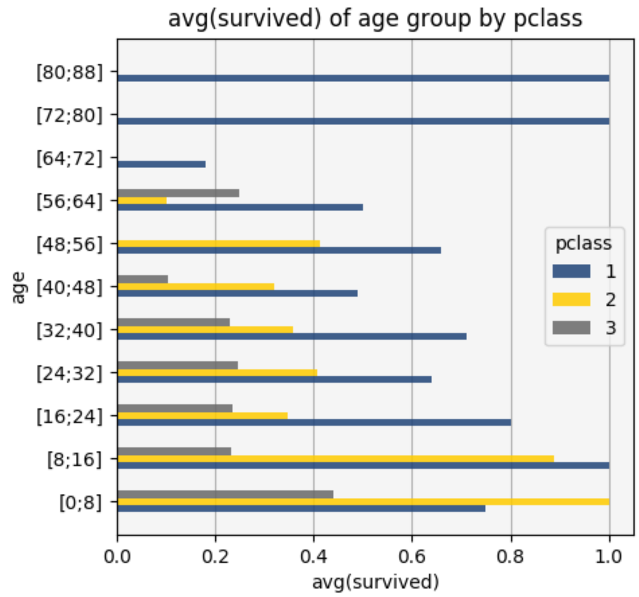
- **columns:** *<list>*
  List of the vDataframe Columns.
- **method:** *<str>*, optional
  count | density | avg | min | max | sum
  count (default): count is used as aggregation
  density: density is used as aggregation
  avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
  The column used to compute the aggregation. This parameter is used only if "method" in {avg | min |max | sum}
- **max_cardinality:** *<tuple>*, optional
  The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<tuple>*, 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.
- **limit_distinct_elements:** *<int>*, optional
  The maximum number of distinct elements. The other categories will be ignored.
- **hist_type:** *<positive int>*, optional
  The Bar Chart type. It can be in {fully_stacked | stacked | auto}

Returns

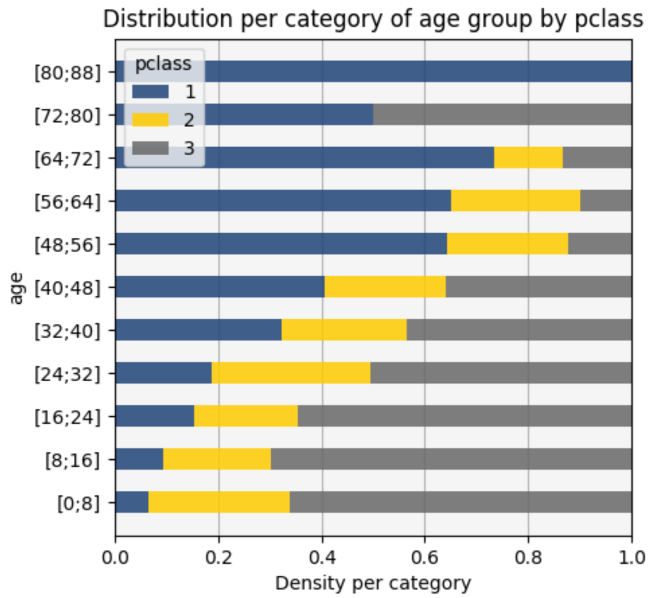
The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.bar(columns = ["age", "pclass"], method = "avg", of = "survived")
```




```
titanic.bar(columns = ["age", "pclass"], method = "density", hist_type = "
    fully_stacked")
```



7.2.4.13 beta

```
vDataframe.beta(columns: list = [])
```

Compute the beta matrix of the corresponding vDataframe columns.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the matrix (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.beta(columns = ["age", "fare", "pclass"])

#Output
                        "age"                  "fare"               "pclass"
"age"                       1       0.696124885853721    -0.0234150242013145
"fare"     1.4365238484091967                       1   -0.00898230868463811
"pclass"   -42.70762188423708     -111.3299525889417                       1
```

7.2.4.14 between_time

```
vDataframe.between_time(self, ts: str, start_time: str, end_time: str)
```

Filter the elements of the vDataframe by considering only the events happening at the time range.

Parameters

- **ts:** *<str>*
  The time series column to use.
- **start_time:** *<str>*
  The start time.
- **end_time:** *<str>*
  The end time.

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)

#Output
                      time            val     id
0     2014-01-01 01:15:00     0.0370000      2
1     2014-01-01 02:30:00     0.0800000      5
2     2014-01-01 03:00:00     0.0810000      1
3     2014-01-01 05:00:00     1.4890000      3
4     2014-01-01 06:00:00     0.0720000      5
...                    ...          ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3

sm.between_time(ts = "time", start_time = "12:00", end_time = "14:00")

#Output
                      time            val     id
0     2014-01-01 13:00:00     0.6220000      4
1     2014-01-01 13:45:00     0.2770000      0
2     2014-01-02 12:30:00     0.3970000      5
3     2014-01-02 13:45:00     0.3580000      0
4     2014-01-03 12:15:00     0.1030000      8
...                    ...          ...    ...
Name: smart_meters, Number of rows: 1151, Number of columns: 3
```

7.2.4.15 boxplot

```
vDataframe.boxplot(self, columns: list = [])
```

Draw the Boxplot of the different vDataframe columns.

Parameters

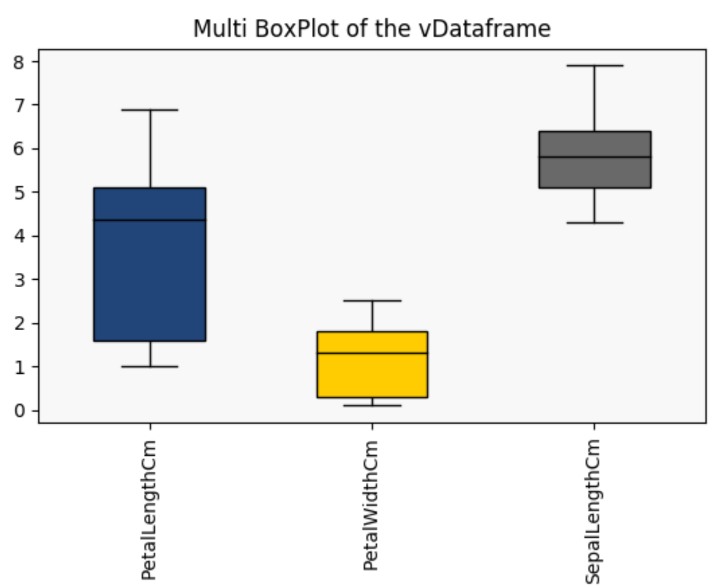
- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The Virtual Dataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_iris
iris = load_iris(cur)
iris.boxplot(columns = ["PetalLengthCm", "PetalWidthCm", "SepalLengthCm"])
```



7.2.4.16 catcol

```
vDataframe.catcol(self, max_cardinality: int = 12)
```

Returns all the different vDataframe categorical columns.

Parameters

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

Returns

List of all the categorical columns.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.catcol()

#Output
['"survived"', '"sex"', '"pclass"', '"embarked"', '"parch"', '"sibsp"']
```

7.2.4.17 copy

```
vDataframe.copy(self)
```

Return a copy of the vDataframe.

Returns

The copy of the vDataframe.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic2 = titanic.copy() # titanic2 is now a copy of titanic
```

7.2.4.18 corr

```
vDataframe.corr(
        self,
        columns: list = [],
        method: str = "pearson",
        cmap: str = "",
        round_nb: int = 3,
        show: bool = True)
```

Compute the correlation matrix of the vDataframe.

Parameters

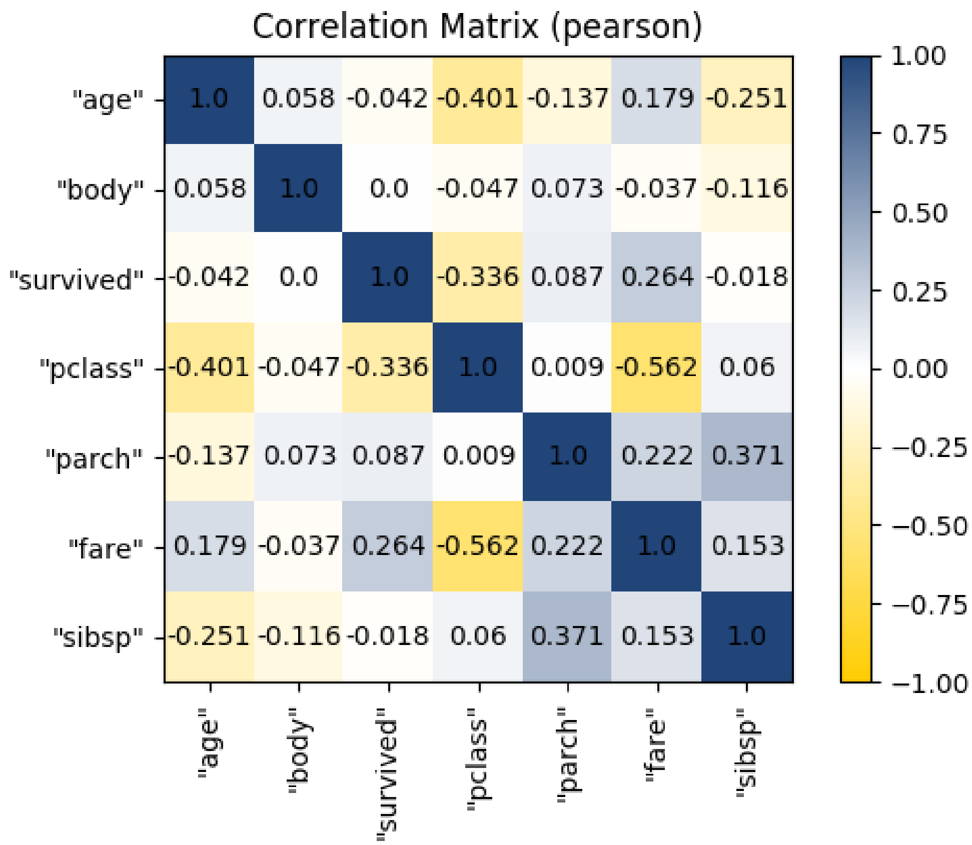
- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.
- **method:** *<str>*, optional
  The method must be in pearson (Pearson Coefficient) | kendall (Kendall Coefficient) | spearman (Spearman Coefficient) | biserial (Biserial Point) | cramer (Cramer'sV)
- **cmap:** *<str>*, optional
  Color Maps.
- **round_nb:** *<int>*, optional
  Integer used to round the numerical values.
- **show:** *<bool>*, optional
  Display the result using matplotlib.

Returns

The `tablesample` type containing the matrix (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.corr()
```



7.2.4.19 cov

```
vDataframe.cov(self, columns: list = [])
```

Compute the covariance matrix of the vDataframe.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the matrix (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.cov(columns = ["survived", "age", "fare"])
```

```
#Output
                        "survived"                  "age"                "fare"
"survived"       0.231685181342251    -0.297583583247234    6.69214075159394
"age"           -0.297583583247234      208.169014723609    145.057125218791
"fare"            6.69214075159394      145.057125218791    2769.36114247479
```

7.2.4.20 count

```
vDataframe.count(self, columns: list = [], percent: bool = True)
```

Aggregate the different columns by computing the count (number of non-NULL element) of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe Columns. If this parameter is empty, the method will consider all the numerical columns.
- **percent:** *<list>*, optional
  If true, compute the percent of non-NULL value of each column.

Returns

The `tablesample` type containing the counts (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.count()

#Output
                    count    percent
"age"               997.0     80.794
"body"              118.0      9.562
"survived"         1234.0      100.0
"ticket"           1234.0      100.0
"home.dest"         706.0     57.212
"cabin"             286.0     23.177
"sex"              1234.0      100.0
"pclass"           1234.0      100.0
"embarked"         1232.0     99.838
"parch"            1234.0      100.0
"fare"             1233.0     99.919
"name"             1234.0      100.0
"boat"              439.0     35.575
```

```
"sibsp"            1234.0       100.0
```

7.2.4.21 cummax

```
vDataframe.cummax(
          self,
          name: str,
          column: str,
          by: list = [],
          order_by: list = [])
```

Add a new column to the vDataframe which is the cumulative max of another one.

Parameters

- **name:** *<str>*
  Name of the new feature.
- **column:** *<str>*
  The column used to compute the cumulative max.
- **by:** *<list>*, optional
  The columns used to group the vDataframe elements.
- **order_by:** *<list>*, optional
  The columns used to order the vDataframe elements. If it is empty, the vDataframe will be ordered by the input column.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.cummax(name = "val_cummax", column = "val", by = ["id"], order_by = ["time"
    ])

#Output
                    time           val     id      val_cummax
0      2014-01-01 11:00:00     0.0290000     0       0.0290000
1      2014-01-01 13:45:00     0.2770000     0       0.2770000
2      2014-01-02 10:45:00     0.3210000     0       0.3210000
3      2014-01-02 11:15:00     0.3050000     0       0.3210000
4      2014-01-02 13:45:00     0.3580000     0       0.3580000
5      2014-01-02 15:30:00     0.1150000     0       0.3580000
6      2014-01-03 08:30:00     0.0710000     0       0.3580000
```

```
7      2014-01-04 23:45:00     0.3230000      0      0.3580000
8      2014-01-06 01:15:00     0.0850000      0      0.3580000
9      2014-01-06 21:45:00     0.7130000      0      0.7130000
...                      ...           ...    ...            ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.2.4.22 cummin

```
vDataframe.cummin(
         self,
         name: str,
         column: str,
         by: list = [],
         order_by: list = [])
```

Add a new column to the vDataframe which is the cumulative min of another one.

Parameters

- **name:** *<str>*
  Name of the new feature.
- **column:** *<str>*
  The column used to compute the cumulative min.
- **by:** *<list>*, optional
  The columns used to group the vDataframe elements.
- **order_by:** *<list>*, optional
  The columns used to order the vDataframe elements. If it is empty, the vDataframe will be ordered by the input column.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.cummin(name = "val_cummin", column = "val", by = ["id"], order_by = ["time"
   ])

#Output
                     time            val     id     val_cummin
0      2014-01-01 11:00:00     0.0290000      0      0.0290000
1      2014-01-01 13:45:00     0.2770000      0      0.0290000
2      2014-01-02 10:45:00     0.3210000      0      0.0290000
```

```
3      2014-01-02 11:15:00    0.3050000     0      0.0290000
4      2014-01-02 13:45:00    0.3580000     0      0.0290000
5      2014-01-02 15:30:00    0.1150000     0      0.0290000
6      2014-01-03 08:30:00    0.0710000     0      0.0290000
7      2014-01-04 23:45:00    0.3230000     0      0.0290000
8      2014-01-06 01:15:00    0.0850000     0      0.0290000
9      2014-01-06 21:45:00    0.7130000     0      0.0290000
...                      ...           ...    ...           ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.2.4.23 cumprod

```
vDataframe.cumprod(
         self,
         name: str,
         column: str,
         by: list = [],
         order_by: list = [])
```

Add a new column to the vDataframe which is the cumulative product of another one.

Parameters

- **name:** *<str>*
  Name of the new feature.
- **column:** *<str>*
  The column used to compute the cumulative product.
- **by:** *<list>*, optional
  The columns used to group the vDataframe elements.
- **order_by:** *<list>*, optional
  The columns used to order the vDataframe elements. If it is empty, the vDataframe will be ordered by the input column.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.cumprod(name = "val_cumprod", column = "val", by = ["id"], order_by = ["
    time"])

#Output
```

```
                            time           val    id                val_cumprod
0      2014-01-01 11:00:00     0.0290000      0                      0.029
1      2014-01-01 13:45:00     0.2770000      0                   0.008033
2      2014-01-02 10:45:00     0.3210000      0                0.002578593
3      2014-01-02 11:15:00     0.3050000      0             0.000786470865
4      2014-01-02 13:45:00     0.3580000      0           0.00028155656967
5      2014-01-02 15:30:00     0.1150000      0        3.237900551205e-05
6      2014-01-03 08:30:00     0.0710000      0      2.29890939135555e-06
7      2014-01-04 23:45:00     0.3230000      0      7.42547733407843e-07
8      2014-01-06 01:15:00     0.0850000      0      6.31165573396666e-08
9      2014-01-06 21:45:00     0.7130000      0      4.50021053831823e-08
...                    ...           ...    ...                       ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.2.4.24 cumsum

```
vDataframe.cumsum(
         self,
         name: str,
         column: str,
         by: list = [],
         order_by: list = [])
```

Add a new column to the vDataframe which is the cumulative sum of another one.

Parameters

- **name:** *<str>*
  Name of the new feature.
- **column:** *<str>*
  The column used to compute the cumulative sum.
- **by:** *<list>*, optional
  The columns used to group the vDataframe elements.
- **order_by:** *<list>*, optional
  The columns used to order the vDataframe elements. If it is empty, the vDataframe will be ordered by the input column.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
```

```
sm.cumsum(name = "val_cumsum", column = "val", by = ["id"], order_by = ["time"
    ])

#Output
                         time           val     id     val_cumsum
0      2014-01-01 11:00:00    0.0290000      0      0.0290000
1      2014-01-01 13:45:00    0.2770000      0      0.3060000
2      2014-01-02 10:45:00    0.3210000      0      0.6270000
3      2014-01-02 11:15:00    0.3050000      0      0.9320000
4      2014-01-02 13:45:00    0.3580000      0      1.2900000
5      2014-01-02 15:30:00    0.1150000      0      1.4050000
6      2014-01-03 08:30:00    0.0710000      0      1.4760000
7      2014-01-04 23:45:00    0.3230000      0      1.7990000
8      2014-01-06 01:15:00    0.0850000      0      1.8840000
9      2014-01-06 21:45:00    0.7130000      0      2.5970000
...                      ...           ...    ...            ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.2.4.25 current_relation

```
vDataframe.current_relation(self)
```

Returns

Returns the vDataframe current relation.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].fillna()
titanic.current_relation()

#Output
(SELECT COALESCE("age", 30.1524573721163) AS "age", "body" AS "body", "
    survived" AS "survived", "ticket" AS "ticket", "home.dest" AS "home.dest",
     "cabin" AS "cabin", "sex" AS "sex", "pclass" AS "pclass", "embarked" AS "
    embarked", "parch" AS "parch", "fare" AS "fare", "name" AS "name", "boat"
    AS "boat", "sibsp" AS "sibsp" FROM (SELECT "age" AS "age", "body" AS "body
    ", "survived" AS "survived", "ticket" AS "ticket", "home.dest" AS "home.
    dest", "cabin" AS "cabin", "sex" AS "sex", "pclass" AS "pclass", "embarked
    " AS "embarked", "parch" AS "parch", "fare" AS "fare", "name" AS "name", "
    boat" AS "boat", "sibsp" AS "sibsp" FROM public.titanic) t1) final_table
```

7.2.4.26 datecol

```
vDataframe.datecol(self)
```

Returns all the timestamps columns.

Returns

List of all the timestamps columns.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.datecol()

#Output
['"time"']
```

7.2.4.27 describe

```
vDataframe.describe(
         self,
         method: str = "numerical",
         columns: list = [],
         unique: bool = True)
```

Summarise the dataset with mathematical information.

Parameters

- **method:** *<str>*, optional
  numerical | categorical
  numerical (default): This mode is used to have only numerical information. Other types are ignored.
  categorical: This mode is available for any type.
- **columns:** *<list>*, optional
  The columns used to compute the mathematical information. If this parameter is empty, the method will consider all the vDataframe columns when the method is 'categorical' otherwise it will only consider the numerical columns.
- **unique:** *<bool>*, optional
  Include the cardinality of each element in the computation

Returns

The `tablesample` type containing the mathematical information (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)

#numerical
titanic.describe()

#Output
              count              mean                 std     \\
age             997    30.1524573721163    14.4353046299159    \\
body            118    164.14406779661    96.5760207557808    \\
fare           1233    33.963793673966    52.6460729831293    \\
parch          1234   0.378444084278768   0.868604707790393    \\
pclass         1234    2.28444084278768   0.842485636190292    \\
sibsp          1234   0.504051863857374    1.04111727241629    \\
survived       1234   0.364667747163696   0.481532018641288    \\
              min        25%        50%        75%    \\
age           0.33       21.0       28.0       39.0    \\
body           1.0      79.25      160.5      257.5    \\
fare           0.0     7.8958    14.4542    31.3875    \\
parch          0.0        0.0        0.0        0.0    \\
pclass         1.0        1.0        3.0        3.0    \\
sibsp          0.0        0.0        0.0        1.0    \\
survived       0.0        0.0        0.0        1.0    \\

#categorical
titanic.describe(method = "categorical")

#Output
                        dtype    unique    count    \\
"age"             numeric(6,3)       96      997    \\
"body"                     int      118      118    \\
"survived"                 int        2     1234    \\
"ticket"           varchar(36)      887     1234    \\
"home.dest"       varchar(100)      359      706    \\
"cabin"            varchar(30)      182      286    \\
"sex"              varchar(20)        2     1234    \\
"pclass"                   int        3     1234    \\
"embarked"         varchar(20)        3     1232    \\
"parch"                    int        8     1234    \\
"fare"           numeric(10,5)      277     1233    \\
"name"            varchar(164)     1232     1234    \\
"boat"            varchar(100)       26      439    \\
"sibsp"                    int        7     1234    \\
                                  top    top_percent
"age"                          24.000          4.413
"body"                              1          0.847
```

```
"survived"                        0         63.533
"ticket"                   CA. 2343           0.81
"home.dest"            New York, NY          8.782
"cabin"                 C23 C25 C27          2.098
"sex"                          male         65.964
"pclass"                          3         53.728
"embarked"                        S          70.86
"parch"                           0         76.904
"fare"                      8.05000          4.704
"name"          Connolly, Miss. Kate         0.162
"boat"                           13          8.428
"sibsp"                           0         67.747
```

7.2.4.28 drop

```
vDataframe.drop(self, columns: list = [])
```

Drop the selected columns from the vDataframe.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT True AS x, 4 AS y) UNION ALL (SELECT False AS x, 9 AS y))
    z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
         x    y
0     True    4
1    False    9
Name: VDF, Number of rows: 2, Number of columns: 2

vdf.drop(["x"])

#Output
     y
0    4
```

```
1     9
Name: y, Number of rows: 2, dtype: int
```

7.2.4.29 drop_duplicates

```
vDataframe.drop_duplicates(self, columns: list = [])
```

Drop the vDataframe duplicates (the duplicates are defined according to specific columns of the vDataframe).

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, 4 AS y) UNION ALL (SELECT 1 AS x, 4 AS y) UNION
   ALL (SELECT 1 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
      x     y
0     1     4
1     1     4
2     1     5
Name: VDF, Number of rows: 3, Number of columns: 2

vdf.drop_duplicates()

#Output
      x     y
0     1     4
1     1     5
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.30 dropna

```
vDataframe.dropna(self, columns: list = [])
```

Drop the vDataframe missing values.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns to consider. If this parameter is empty, it will filter all the rows having at least one missing element.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, NULL AS y) UNION ALL (SELECT 1 AS x, 4 AS y)
    UNION ALL (SELECT 1 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
      x        y
0     1      None
1     1        4
2     1        5
Name: VDF, Number of rows: 3, Number of columns: 2

vdf.dropna()

#Output
      x     y
0     1     4
1     1     5
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.31 dsn_restart

```
vDataframe.dsn_restart(self)
```

Set a new vDataframe cursor connection using the stored DSN. This method is useful if the connection to the Vertica DB failed.

7.2.4.32 dtypes

```
vDataframe.dtypes(self)
```

Returns all the vDataframe columns types.

Returns

The `tablesample` type containing the columns types (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.dtypes()

#Output
                           dtype
"age"                numeric(6,3)
"body"                        int
"survived"                    int
"ticket"              varchar(36)
"home.dest"          varchar(100)
"cabin"               varchar(30)
"sex"                 varchar(20)
"pclass"                      int
"embarked"            varchar(20)
"parch"                       int
"fare"              numeric(10,5)
"name"               varchar(164)
"boat"               varchar(100)
"sibsp"                       int
```

7.2.4.33 duplicated

```
vDataframe.duplicated(self, columns: list = [], count: bool = False)
```

Find all the vDataframe duplicates (the duplicates are defined according to specific columns of the vDataframe).

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns.
- **count:** *<bool>*, optional
  If True, the function will return the number of duplicates.

Returns

The `tablesample` type containing the duplicates (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, 4 AS y) UNION ALL (SELECT 1 AS x, 4 AS y) UNION ALL (SELECT 1 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
     x     y
0    1     4
1    1     4
2    1     5
Name: VDF, Number of rows: 3, Number of columns: 2

vdf.duplicated()

#Output
     x     y     occurrence
0    1     4              2
Name: Duplicated Rows, Number of rows: 1, Number of columns: 3
```

7.2.4.34 empty

```
vDataframe.empty(self)
```

Returns True if the vDataframe is empty.

7.2.4.35 eval

```
vDataframe.eval(self, name: str, expr: str)
```

Evaluate an expression and create a new column if the expression is correct.

Parameters

- **name:** *<str>*
  Name of the new column
- **expr:** *<str>*
  The expression used to create the new column

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 2 AS x, 4 AS y) UNION ALL (SELECT 3 AS x, 4 AS y) UNION
    ALL (SELECT 10 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")


#Output
      x    y
0     2    4
1     3    4
2    10    5
Name: VDF, Number of rows: 3, Number of columns: 2

vdf.eval("z", "x * y")

#Output
      x    y     z
0     2    4     8
1     3    4    12
2    10    5    50
Name: VDF, Number of rows: 3, Number of columns: 3
```

7.2.4.36 expected_store_usage

```
vDataframe.expected_store_usage(self, unit = 'b')
```

Expected data volume if the final relation is stored in the Database.

Parameters

- **unit:** *<str>*, optional
  Storage Unit, can be in {b | kb | mb | gb | tb}

Returns

The `tablesample` type containing the expected store usage of each column (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.expected_store_usage(unit = 'kb')

#Output
                    expected_size (kb)    max_size (kb)           type
"age"                     0.0087890625     8.7626953125   numeric(6,3)
"body"                     0.306640625            7.375            int
"survived"                 1.205078125           77.125            int
"ticket"                  8.2001953125       43.3828125    varchar(36)
"home.dest"              13.130859375       68.9453125   varchar(100)
"cabin"                      1.0390625       8.37890625    varchar(30)
"sex"                         5.640625       24.1015625    varchar(20)
"pclass"                   1.205078125           77.125            int
"embarked"                    1.203125          24.0625    varchar(20)
"parch"                    1.205078125           77.125            int
"fare"                    0.0146484375    18.0615234375  numeric(10,5)
"name"                   31.9521484375      197.6328125   varchar(164)
"boat"                    0.6318359375      42.87109375   varchar(100)
"sibsp"                    1.205078125           77.125            int
separator                  16.87109375      16.87109375
header                      0.11328125       0.11328125
rawsize                  83.9326171875     769.05859375
```

7.2.4.37 fillna

```
vDataframe.fillna(
         self,
         val: dict = {},
         method: dict = {},
         numeric_only: bool = False)
```

Fill the missing values using the input methods. If the parameters val and method are empty, all the missing values will be filled automatically (using the average of the column for the numerical columns and the mode for the categorical ones)

Parameters

- **val:** *<dict>*, optional
  Dictionary of columns with the constant value used to fill their missing values.
- **method:** *<dict>*, optional
  Dictionary of columns with the method used to fill their missing values. The method can be in {avg | median | mode}. For more flexibility, use the method `fillna` of the vColumn.
- **numeric_only:** *<bool>*, optional
  If true, only the numerical columns will be filled

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 2 AS x, NULL AS y) UNION ALL (SELECT NULL AS x, 4 AS y)
   UNION ALL (SELECT 10 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")


#Output
        x       y
0       2    None
1    None       4
2      10       5
Name: VDF, Number of rows: 3, Number of columns: 2

vdf.fillna()

#Output
        x      y
0     2.0    4.5
1     6.0    4.0
2    10.0    5.0
Name: VDF, Number of rows: 3, Number of columns: 2
```

7.2.4.38 filter

```
vDataframe.filter(self, expr: str = "", conditions: list = [])
```

Filter the elements of the vDataframe using an expression or multiple conditions.

Parameters

- **val:** *<expr>*, optional
  The expression used to filter the data.
- **conditions:** *<list>*, optional
  A list of conditions used to filter the data.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 2 AS x, 1 AS y) UNION ALL (SELECT 10 AS x, 4 AS y) UNION
    ALL (SELECT 10 AS x, 5 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
       x     y
0      2     1
1     10     4
2     10     5
Name: VDF, Number of rows: 3, Number of columns: 2

vdf.filter(expr = "x = 10")

#Output
       x     y
0     10     4
1     10     5
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.2.4.39 first

```
vDataframe.first(self, ts: str, offset: str)
```

Keep only the first events of the specific time series.

Parameters

- **ts:** *<str>*
  The time series column.
- **offset:** *<list>*, optional
  The offset time interval used to filter the data.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
# We only keep the first day of the dataset
sm.first(ts = "time", offset = "1 day")

#Output
```

```
                        time          val     id
0     2014-01-01 01:15:00     0.0370000      2
1     2014-01-01 02:30:00     0.0800000      5
2     2014-01-01 03:00:00     0.0810000      1
3     2014-01-01 05:00:00     1.4890000      3
4     2014-01-01 06:00:00     0.0720000      5
...                    ...          ...    ...
Name: smart_meters, Number of rows: 20, Number of columns: 3
```

7.2.4.40 get_columns

```
vDataframe.get_columns(self)
```

Returns the vDataframe columns.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.get_columns()

#Output
['"time"', '"val"', '"id"']
```

7.2.4.41 groupby

```
vDataframe.groupby(self, columns: list, expr: list = [])
```

Group the different categories of the input columns and compute the different aggregations.

Parameters

- **columns:** *<list>*
  List of the columns to group with.
- **expr:** *<list>*, optional
  List of the different aggregations to apply.

Returns

A new vDataframe corresponding to the group-by result.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.groupby(columns = ["pclass"], expr = ["AVG(survived)", "COUNT(*)"])

#Output
     pclass                  AVG    COUNT
0         1    0.612179487179487      312
1         2    0.416988416988417      259
2         3    0.227752639517345      663
Name: titanic, Number of rows: 3, Number of columns: 3
```

7.2.4.42 head

```
vDataframe.head(self, limit: int = 5)
```

Returns the vDataframe first elements.

Parameters

- **limit:** *<int>*
  The number of elements to return.

Returns

The `tablesample` type containing the head of the vDataframe (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.head(limit = 10)

#Output
                      time          val     id
0      2014-01-01 01:15:00    0.0370000      2
1      2014-01-01 02:30:00    0.0800000      5
2      2014-01-01 03:00:00    0.0810000      1
3      2014-01-01 05:00:00    1.4890000      3
4      2014-01-01 06:00:00    0.0720000      5
5      2014-01-01 07:15:00    2.3060000      9
6      2014-01-01 07:45:00    0.1020000      4
7      2014-01-01 10:45:00    0.0970000      8
8      2014-01-01 11:00:00    0.0290000      0
9      2014-01-01 11:00:00    0.5060000      6
```

```
...                     ...          ...   ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3
```

7.2.4.43 help

```
vDataframe.help(self)
```

Prints information about the vDataframe.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.help()

#Output

#############################
#                           #
# Vertica Virtual Dataframe #
#                           #
#############################

The vDataframe is a Python object which will keep in mind all the user
    modifications in order to use an optimised 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
    vColumn (Vertica Virtual Column) which will store for each column its name
    , its imputations and allows to do easy modifications and explorations.

vColumn and vDataframe coexist and one can not live without the other. vColumn
     will use the vDataframe 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 vDataframe gets in memory all
    the transformations to select for each query the needed data in the input
    relation.

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

```
If you find any difficulties using vertica_ml_python, please contact me: badr.
    ouali@microfocus.com / I'll be glad to help.

For more information about the different methods or the entire vDataframe
    structure, please see the entire documentation
```

7.2.4.44 hexbin

```
vDataframe.hexbin(
            self,
            columns: list,
            method: str = "count",
            of: str = "",
            cmap: str = '',
            gridsize: int = 10,
            color: str = "white")
```

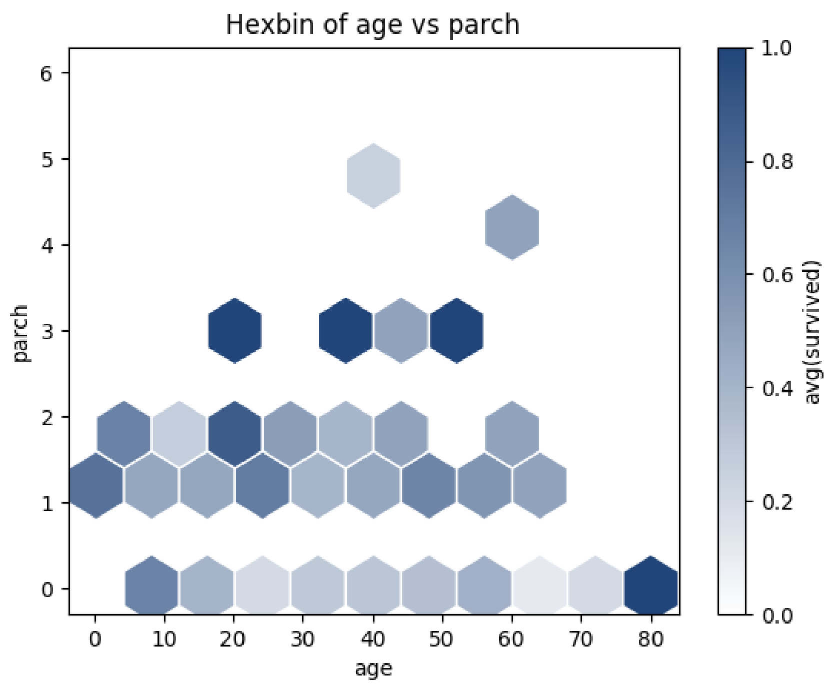
Draw the corresponding Hexbin plot.

Parameters

- **columns:** *<list>*
  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:** *<int>*, optional
  Grid Size.
- **color:** *<str>*, optional
  Hexbin outline color.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.hexbin(columns = ["age", "parch"], method = "avg", of = "survived")
```



7.2.4.45 hist

```
Dataframe.hist(
        self,
        columns: list,
        method: str = "density",
        of: str = "",
        max_cardinality: tuple = (6, 6),
        h: tuple = (None, None),
        limit_distinct_elements: int = 200,
        hist_type: str = "auto")
```

Draw the corresponding variables Histogram.

Parameters

- **columns:** *<list>*
  List of the vDataframe columns.
- **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 parameter is used only if "method" in {avg | min |max | sum}

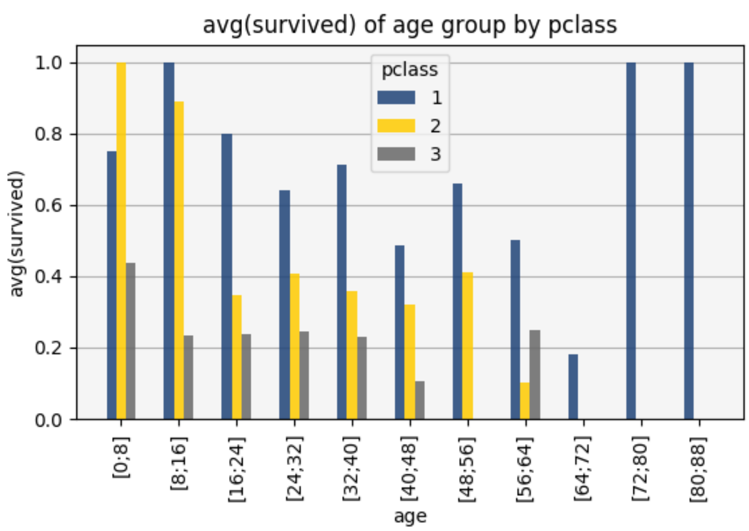
- **max_cardinality:** *<tuple>*, optional
  The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<tuple>*, optional
  The interval size of each column. It is used if the column is numerical. In the other case, if h is not informed. The best "h" will be computed automatically.
- **limit_distinct_elements:** *<int>*, optional
  The maximum number of distinct elements. The other categories will be ignored.
- **hist_type:** *<positive int>*, optional
  The Histogram type. It can be in {multi | stacked | auto}

Returns

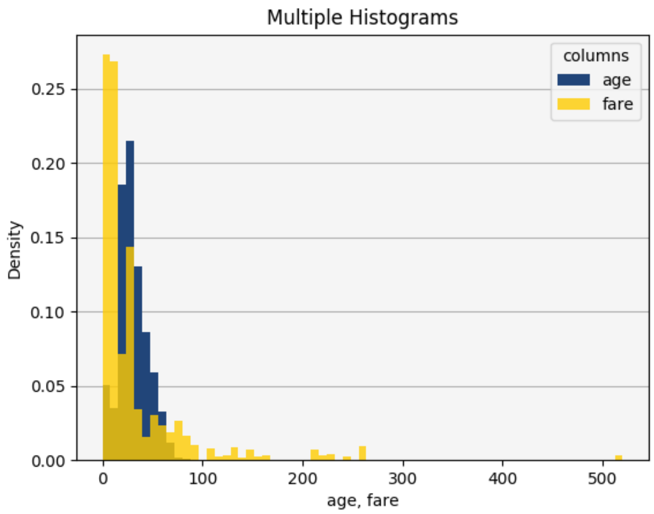
The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.hist(columns = ["age", "pclass"], method = "avg", of = "survived")
```




```
titanic.hist(columns = ["age", "fare"], hist_type = "multi")
```



7.2.4.46 info

```
Dataframe.info(self)
```

Summarise all the modifications made to the vDataframe.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["sex"].label_encode()
titanic["age"].fillna()
titanic.info()

#Output
The vDataframe was modified many times:
 * {Sat Nov 23 05:33:55 2019} [Label Encoding]: Label Encoding was applied to the vColumn '"sex"' using the following mapping:
   female => 0 male => 1
 * {Sat Nov 23 05:33:55 2019} [Fillna]: 237 missing values of the vColumn '"age"' were filled.
```

7.2.4.47 isin

```
vDataframe.isin(self, val: dict)
```

Verify if the elements are in the vDataframe.

Parameters

- **val:** *<dict>*
  Dictionary of the different columns and the values to check.

Returns

A list of booleans, result of the values checking.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
#We check if there are a 15 years old and a 50 years old passengers in first class
titanic.isin({"age": [15, 50], "pclass": [1, 1]})

#Output
[True, True]
```

7.2.4.48 join

```
vDataframe.join(
        self,
        input_relation: str = "",
        vdf = None,
        on: dict = {},
        how: str = 'natural')
```

Join the vDataframe with another one or another relation.

Parameters

- **input_relation:** *<str>*, optional
  The relation to join with the vDataframe.
- **vdf:** *<object>*, optional
  The vDataframe to join with the current vDataframe.
- **on:** *<dict>*, optional
  Dictionary of the elements which are the main keys of the joins.
- **how:** *<str>*, optional
  The join methods, it must be in {cross | natural | inner | left | right | self}

Returns

The vDataframe of the join.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 0 AS id, 'Fouad' AS name) UNION ALL (SELECT 1 AS id, '
    Colin' AS name) UNION ALL (SELECT 2 AS id, 'Badr' AS name)) z"
vdf1 = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
     id     name
0     0    Fouad
1     1    Colin
2     2     Badr
Name: VDF, Number of rows: 3, Number of columns: 2

relation = "((SELECT 0 AS id, 'Apple' AS fav_fruit) UNION ALL (SELECT 1 AS id,
     'Blueberries' AS fav_fruit) UNION ALL (SELECT 2 AS id, 'Mango' AS
    fav_fruit)) z"
vdf2 = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
     id      fav_fruit
0     0          Apple
1     1    Blueberries
2     2          Mango
Name: VDF, Number of rows: 3, Number of columns: 2

vdf1.join(vdf = vdf2)

#Output
     id     name       fav_fruit
0     0    Fouad           Apple
1     1    Colin     Blueberries
2     2     Badr           Mango
Name: VDF, Number of rows: 3, Number of columns: 3
```

7.2.4.49 kurtosis / kurt

```
vDataframe.kurtosis(self, columns: list = [])
```

Aggregate the different columns by computing the unbiased kurtosis using Fisher?s definition (kurtosis of normal = 0.0).

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe Columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the kurtosis (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.kurt()

#Output
                          kurtosis
"age"            0.15689691331997
"body"          -1.23864914040606
"survived"      -1.68576262213743
"pclass"        -1.34962169484619
"parch"          22.6438022640172
"fare"           26.2543152552867
"sibsp"          19.2138853382802
```

7.2.4.50 last

```
vDataframe.last(self, ts: str, offset: str)
```

Keep only the last events of the specific time series.

Parameters

- **ts:** *<str>*
  The time series column.
- **offset:** *<str>*
  The offset time interval used to filter the data.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.last(ts = "time", offset = "1 day")
```

```
#Output
                    time           val     id
0     2015-09-10 05:00:00     0.0930000      1
1     2015-09-10 05:30:00     0.1410000      1
2     2015-09-10 06:45:00     0.6030000      2
3     2015-09-10 07:15:00     0.8480000      3
4     2015-09-10 07:30:00     0.0570000      6
...                   ...           ...    ...
Name: smart_meters, Number of rows: 20, Number of columns: 3

# We only keep the last day of the dataset
```

7.2.4.51 load

```
vDataframe.load(self, offset: int = -1)
```

Load the vDataframe corresponding to the selected saving.

Parameters

- **offset:** *<list>*, optional
  The saving position (-1 for the last one)

Returns

Create a new vDataframe which is based on the selected saving.

Examples

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.save()
titanic["sex"]

#Output
        sex
0    female
1      male
2    female
3      male
4      male
...     ...
Name: sex, Number of rows: 1234, dtype: varchar(20)

titanic["sex"].label_encode()
```

```
titanic["sex"]

#Output
     sex
0      0
1      1
2      0
3      1
4      1
...  ...
Name: sex, Number of rows: 1234, dtype: int

titanic = titanic.load()
titanic["sex"]

#Output
        sex
0    female
1      male
2    female
3      male
4      male
...     ...
Name: sex, Number of rows: 1234, dtype: varchar(20)
```

7.2.4.52 mad

```
vDataframe.mad(self, columns: list = [])
```

Aggregate the different columns by computing the mean absolute deviation of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the mad (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.mad()
```

```
#Output
                               mad
"age"                11.254785419447906
"body"                82.97457627118644
"survived"          0.46337036268450094
"pclass"             0.7689071656916803
"parch"              0.5820801231451393
"fare"               30.625865942462237
"sibsp"              0.6829616826333305
```

7.2.4.53 max

```
vDataframe.max(self, columns: list = [])
```

Aggregate the different columns by computing the max of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the max (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.max()

#Output
                      max
"age"                80.0
"body"              328.0
"survived"            1.0
"pclass"              3.0
"parch"               9.0
"fare"           512.3292
"sibsp"               8.0
```

7.2.4.54 median

```
vDataframe.median(self, columns: list = [])
```

Aggregate the different columns by computing the median of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the medians (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.median()

#Output
                  median
"age"               28.0
"body"             160.5
"survived"           0.0
"pclass"             3.0
"parch"              0.0
"fare"           14.4542
"sibsp"              0.0
```

7.2.4.55 memory_usage

```
vDataframe.memory_usage(self)
```

Memory usage of the object in byte (it will never exceed some kb).

Returns

The `tablesample` type containing the memory usage of each column (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.memory_usage(unit = 'kb')

#Output
                  value
object              687
"age"               182
"body"              183
"survived"          187
"ticket"            185
"home.dest"         188
"cabin"             184
"sex"               182
"pclass"            185
"embarked"          187
"parch"             184
"fare"              183
"name"              183
"boat"              183
"sibsp"             184
total              3267
```

7.2.4.56 min

```
vDataframe.max(self, columns: list = [])
```

Aggregate the different columns by computing the min of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the min (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.min()
```

```
#Output
                 min
"age"           0.33
"body"           1.0
"survived"       0.0
"pclass"         1.0
"parch"          0.0
"fare"           0.0
"sibsp"          0.0
```

7.2.4.57 nlargest

```
vDataframe.nlargest(self, column: str, n: int = 10)
```

Returns the n largest elements of the vDataframe sorting by a specific column.

Parameters

- **column:** *<str>*
  The column used to sort the data.
- **n:** *<int>*, optional
  The number of elements to consider.

Returns

The `tablesample` type containing the vDataframe n largest elements (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.nlargest(column = "fare")

#Output
        age     body     survived       ticket
0     36.000     None            1     PC 17755
1     58.000     None            1     PC 17755
2     35.000     None            1     PC 17755
           cabin         sex     pclass     embarked     parch          fare    \\
0     B51 B53 B55       male          1            C         1     512.32920    \\
1     B51 B53 B55     female          1            C         1     512.32920    \\
2            B101       male          1            C         0     512.32920    \\
                                       name     boat     sibsp
0          Cardeza, Mr. Thomas Drake Martinez        3         0
1   Cardeza, Mrs. James Warburton Martine...        3         0
```

```
2                        Lesurer, Mr. Gustave J        3         0
Name: nlargest, Number of rows: 3, Number of columns: 14
```

7.2.4.58 normalize

```
vDataframe.normalize(self, method = "zscore")
```

Normalize all the numerical columns of the vDataframe using the corresponding method. Use the `normalize` method of each column for more flexibility.

Parameters

- **method:** *<str>*, optional
  The method to be used: {zscore | robust_zscore | minmax}

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.normalize()
titanic["fare"]

#Output
                                fare
0      2.23352283775686733061637440003
1      2.23352283775686733061637440003
2      2.23352283775686733061637440003
3     -0.6451344183800711827483251581
4      0.2951864297839290254556257484
...                                ...
Name: fare, Number of rows: 1234, dtype: float
```

7.2.4.59 nsmallest

```
vDataframe.nsmallest(self, column: str, n: int = 10)
```

Returns the n smallest elements of the vDataframe sorting by a specific column.

Parameters

- **column:** *<str>*
  The column used to sort the data.
- **n:** *<int>*, optional
  The number of elements to consider.

Returns

The `tablesample` type containing the vDataframe n smallest elements (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.nsmallest(column = "fare")

#Output
        age     body     survived     ticket
0    39.000     None            0     112050
1      None     None            0     112051
2      None     None            0     112058
     cabin       sex     pclass     embarked     parch        fare    \\
0      A36     male          1            S         0     0.00000    \\
1     None     male          1            S         0     0.00000    \\
2     B102     male          1            S         0     0.00000    \\
                                         name     boat     sibsp
0                     Andrews, Mr. Thomas Jr     None         0
1    Chisholm, Mr. Roderick Robert Crispin     None         0
2                           Fry, Mr. Richard     None         0
Name: nsmallest, Number of rows: 3, Number of columns: 14
```

7.2.4.60 numcol

```
vDataframe.numcol(self)
```

Returns the vDataframe numerical columns.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.numcol()

#Output
['"age"', '"body"', '"survived"', '"pclass"', '"parch"', '"fare"', '"sibsp"']
```

7.2.4.61 pivot_table

```
vDataframe.pivot_table(
          self,
          columns: list,
          method: str = "count",
          of: str = "",
          h: tuple = (None, None),
          max_cardinality: tuple = (20, 20),
          show: bool = True,
          cmap: str = '',
          limit_distinct_elements: int = 1000,
          with_numbers: bool = True)
```

Draw the corresponding pivot table.

Parameters

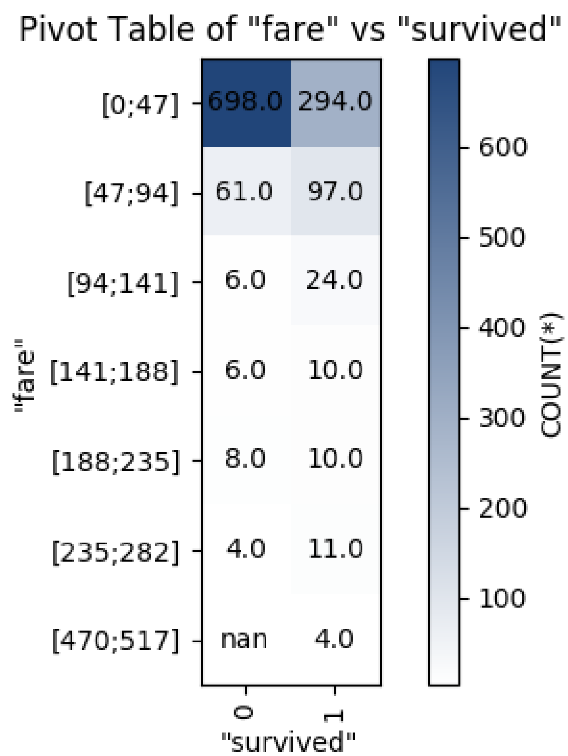
- **columns:** *<list>*
  The two columns used to build the pivot table.
- **method:** *<str>*, optional
  count | density | avg | min | max | sum
  count (default): count is used as aggregation
  density: density is used as aggregation
  avg | min | max | sum: these aggregations are used only if "of" 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:** *<tuple>*, optional
  The maximum cardinality of each column. Under this number the column is automatically considered as categorical.
- **h:** *<tuple>*, 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:** *<int>*, optional
  The maximum number of distinct elements. The other categories will be ignored.
- **show:** *<bool>*, optional
  Draw the pivot table using matplotlib.
- **with_numbers:** *<bool>*, optional
  If False, draw the pivot table without displaying the numbers.

Returns

The `tablesample` type containing the pivot table (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

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

#Output
     "fare"/"survived"       0       1
0                [0;47]    698     294
1               [47;94]     61      97
2              [94;141]      6      24
3             [141;188]      6      10
4             [188;235]      8      10
5             [235;282]      4      11
6             [470;517]              4
```



7.2.4.62 plot

```
vDataframe.plot(
         self,
         ts: str,
```

```
        columns: list = [],
        start_date: str = "",
        end_date: str = "")
```

Plot the time series selected columns.

Parameters

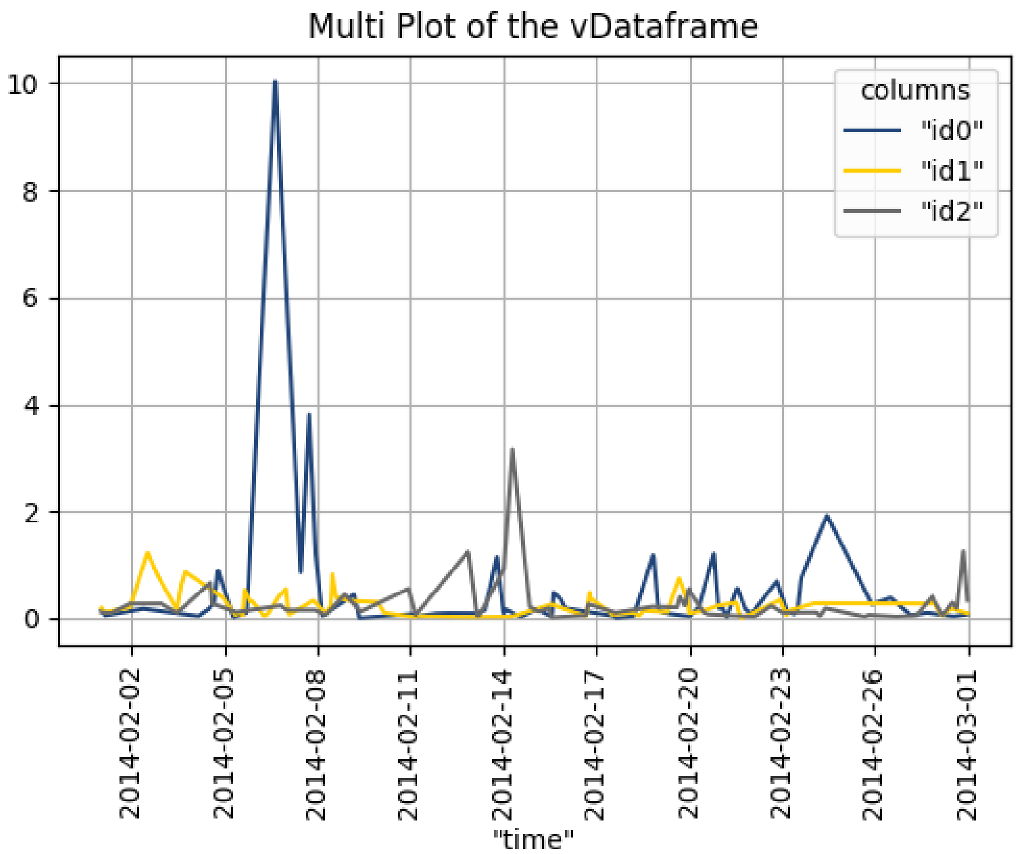
- **ts:** *<str>*
  The time series used to plot the different elements.
- **columns:** *<list>*, optional
  The columns to plot. If empty, all the numerical columns will be plotted.
- **start_date:** *<str>*, optional
  Start Date.
- **end_date:** *<str>*, optional
  End Date.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
# Slicing and interpolating the time series
sm = sm.asfreq(ts = "time", rule = "30 minutes", method = {"val": "linear"},
    by = ["id"])
# Building the features corresponding to the consumption of 3 different homes
sm.eval("id0", "DECODE(id, 0, val, NULL)")
sm.eval("id1", "DECODE(id, 1, val, NULL)")
sm.eval("id2", "DECODE(id, 2, val, NULL)")
# Drawing the time series
sm.plot(ts = "time", columns = ["id0", "id1", "id2"], start_date = "2014-02-01
     00:00:00", end_date = "2014-03-01 00:00:00")
```



7.2.4.63 product / prod

```
vDataframe.product(self, columns: list = [])
```

Aggregate the different columns by computing the product of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe Columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the products (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.prod()
```

```
#Output
                                  prod
"age"                              inf
"body"              5.04839415885224e+245
"survived"                         0.0
"pclass"                           inf
"parch"                            0.0
"fare"                             0.0
"sibsp"                            0.0
```

7.2.4.64 quantile

```
vDataframe.quantile(self, q: list, columns: list = [])
```

Aggregate the different columns by computing the different selected quantiles of each one.

Parameters

- **q:** *<list>*
  List of the different quantiles (each element must be in [0,1]).
- **columns:** *<list>*, optional
  List of the vDataframe Columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the quantiles (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.quantile(q = [0.1, 0.5, 0.9])

#Output
                 10.0%      50.0%     90.0%
"age"             14.5       28.0      50.0
"body"            37.7      160.5     297.3
"survived"         0.0        0.0       1.0
"pclass"           1.0        3.0       3.0
"parch"            0.0        0.0       1.0
"fare"          7.5892    14.4542     79.13
"sibsp"            0.0        0.0       1.0
```

7.2.4.65 rank

```
vDataframe.rank(self, order_by: list, method: str = "first", by: list = [],
    name = "")
```

Compute the rank of the selected column. This method will add the new feature to the vDataframe.

Parameters

- **order_by:** *<list>*
  The columns used to order the vDataframe elements.
- **method:** *<str>*, optional
  Method used to compute the Rank, it must be in {first | dense | percent}
- **by:** *<list>*, optional
  The columns used to group the vDataframe elements.
- **name:** *<str>*, optional
  Name of the new feature.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.rank(order_by = ["time"], by = ["id"])

#Output
                      time           val    id    first_rank_time_by_id
0     2014-01-01 11:00:00     0.0290000      0                        1
1     2014-01-01 13:45:00     0.2770000      0                        2
2     2014-01-02 10:45:00     0.3210000      0                        3
3     2014-01-02 11:15:00     0.3050000      0                        4
4     2014-01-02 13:45:00     0.3580000      0                        5
...                   ...           ...    ...                      ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.2.4.66 rolling

```
vDataframe.rolling(
        self,
        name: str,
        aggr: str,
```

```
        column: str,
        preceding,
        following,
        expr: str = "",
        by: list = [],
        order_by: list = [],
        method: str = "rows")
```

Compute a Moving Window. The method will add the new feature to the vDataframe.

Parameters

- **name:** *<str>*
  Name of the new feature.
- **aggr:** *<str>*
  Aggregation used to compute the Moving Window.
- **column:** *<str>*
  The column used to compute the aggregation.
- **preceding:** *<str>*,
  Rule for the preceding elements. It can be an integer if the method is set to 'rows' otherwise an interval. It can also be set to 'unbounded' in order to consider all the elements before the event.
- **following:** *<str>*,
  Rule for the following elements. It can be an integer if the method is set to 'rows' otherwise an interval. It can also be set to 'unbounded' in order to consider all the elements after the event.
- **expr:** *<str>*, optional
  The expression to consider instead of the aggregation. It can be used if a complex Moving Window is needed.
- **by:** *<list>*, optional
  The columns used to group the vDataframe elements.
- **order_by:** *<list>*, optional
  The columns used to order the vDataframe elements. If it is empty, the vDataframe will be ordered by the selected column.
- **method:** *<list>*, optional
  Method used to compute the Moving Window, it must be in {rows | range} (range is only available if the order_by column is a timestamp)

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
# Computing the highest consumption per home one day preceding and one day
    following the current date
```

```
sm.rolling(
        name = "highest_consumption_1p_1f",
        aggr = "max", column = "val",
        preceding = "1 days",
        following = "1 days",
        by = ["id"],
        order_by = ["time"],
        method = "range")

#Output
                        time           val     id     val_cummax
                       time           val     id     highest_consumption_1p_1f
0     2014-01-01 11:00:00     0.0290000      0                        0.3210000
1     2014-01-01 13:45:00     0.2770000      0                        0.3580000
2     2014-01-02 10:45:00     0.3210000      0                        0.3580000
3     2014-01-02 11:15:00     0.3050000      0                        0.3580000
4     2014-01-02 13:45:00     0.3580000      0                        0.3580000
...                     ...          ...    ...                            ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.2.4.67 sample

```
vDataframe.sample(self, x: float)
```

Returns a sample of the vDataframe relation.

Parameters

- **x:** *<float>*
  A float which indicate the sample value (must be in [0,1]).

Returns

The `tablesample` type containing the sample (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.sample(x = 0.005)

#Output
     cabin        sex    pclass     embarked     parch    \\
0       C6       male         1            C         0    \\
1     None       male         2            S         0    \\
```

```
2      None       male          2            S          0    \\
3        F2       male          2            S          1    \\
4      None       male          3            Q          0    \\
5      None     female          3            Q          0    \\
6      None     female          3            C          1    \\
         fare                          name     boat     sibsp
0    75.24170          Beattie, Mr. Thomson        A         0
1    13.00000         Greenberg, Mr. Samuel     None         0
2    10.50000        Wheadon, Mr. Edward H     None         0
3    26.00000     Navratil, Master. Michel M        D         1
4     7.73330       O Connell, Mr. Patrick D     None         0
5     7.73330             Smyth, Miss. Julia       13         0
6    15.24580     Touma, Miss. Maria Youssef        C         1
Name: Sample(0.005) of titanic, Number of rows: 7, Number of columns: 14
```

7.2.4.68 save

```
vDataframe.save(self)
```

Save the vDataframe current disposition. In case of unwanted operations, it will be possible to go back by loading a saving using the `load` method.

Returns

The vDataframe itself.

Examples

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.save()
titanic["sex"]

#Output
        sex
0    female
1      male
2    female
3      male
4      male
...     ...
Name: sex, Number of rows: 1234, dtype: varchar(20)

titanic["sex"].label_encode()
titanic["sex"]
```

```
#Output
      sex
0       0
1       1
2       0
3       1
4       1
...   ...
Name: sex, Number of rows: 1234, dtype: int

titanic = titanic.load()
titanic["sex"]

#Output
         sex
0     female
1       male
2     female
3       male
4       male
...      ...
Name: sex, Number of rows: 1234, dtype: varchar(20)
```

7.2.4.69 scatter

```
vDataframe.scatter(
        self,
          columns: list,
          cat_col: str = "",
          max_cardinality: int = 3,
          cat_priority: list = [],
          with_others: bool = True,
          max_nb_points: int = 20000)
```

Draw the scatter plot of the input columns.

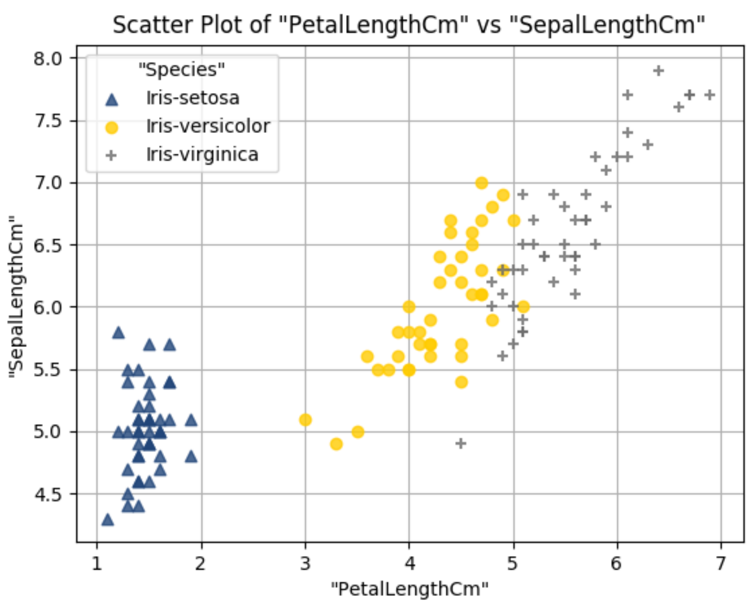
Parameters

- **columns:** *<str>*
  The two or three columns used to draw the scatter plot.
- **catcol:** *<str>*, optional
  The categorical column used as label.
- **max_cardinality:** *<int>*, optional
  The maximum cardinality of the categorical column, all the other categories are merged to create the "others" category.

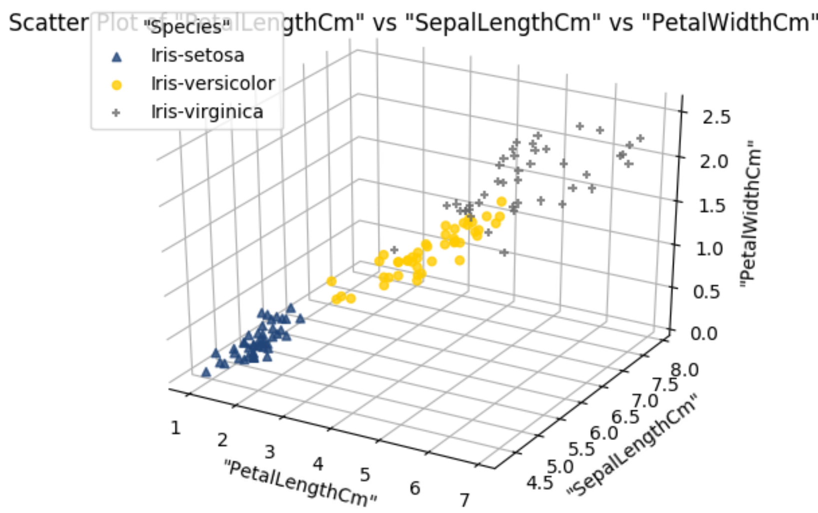
- **cat_priority:** *<list>*, optional
  The list of the categories took into account during the computation.
- **with_others:** *<bool>*, optional
  Include the "others" category.
- **max_nb_points:** *<int>*, optional
  The maximum number of points in the scatter plot. The points are taken randomly from the table.

Examples

```
from vertica_ml_python.learn.datasets import load_iris
iris = load_iris(cur)
iris.scatter(columns = ["PetalLengthCm", "SepalLengthCm"], catcol = "Species")
```




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



7.2.4.70 scatter_matrix

```
vDataframe.scatter(self, columns: list = [])
```

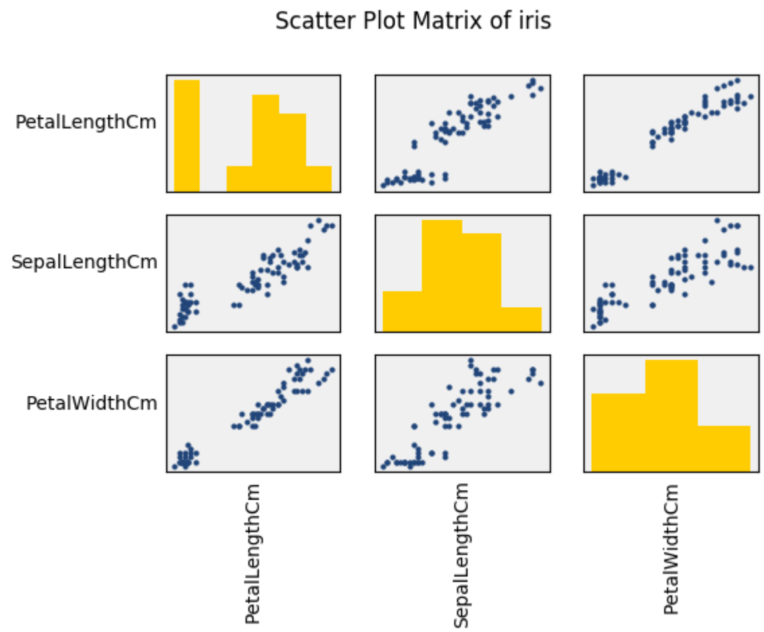
Draw the scatter matrix of the input columns.

Parameters

- **columns:** *<str>*
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Examples

```
from vertica_ml_python.learn.datasets import load_iris
iris = load_iris(cur)
iris.scatter_matrix(columns = ["PetalLengthCm", "SepalLengthCm", "PetalWidthCm
    "])
```



7.2.4.71 select

```
vDataframe.select(self, columns: list)
```

Select some of the vDataframe columns.

Parameters

- **columns:** *<list>*
  List of the columns to select.

Returns

A new vDataframe containing only the selected columns.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.select(columns = ['pclass','age'])

#Output
        age    pclass
0     2.000         1
1    30.000         1
2    25.000         1
3    39.000         1
4    71.000         1
```

```
...     ...        ...
Name: titanic, Number of rows: 1234, Number of columns: 2
```

7.2.4.72 sem

```
vDataframe.sem(self, columns: list = [])
```

Aggregate the different columns by computing the unbiased standard error of the mean of each element.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the sem (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.sem()

#Output
                                   sem
"age"              0.457170684605937
"body"              8.89054334053935
"survived"        0.0137077946622731
"pclass"           0.023983078299543
"parch"           0.0247266111413956
"fare"              1.49928585339507
"sibsp"            0.029637534446613
```

7.2.4.73 sessionize

```
vDataframe.sessionize(
            self,
            ts: str,
            by: list = [],
            session_threshold = "30 minutes",
            name = "session_id")
```

Create a new feature which will represent the different elements session. A session is defined as a user inactivity during a certain amount of time (called the session threshold).

Parameters

- **ts:** *<str>*
  The vDataframe time series.
- **by:** *<list>*
  List of the different element used to group the vDataframe (Most of the time it is a user id).
- **session_threshold:** *<str>*, optional
  The session threshold. It must be a time interval.
- **name:** *<str>*, optional
  Name of the session.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT '2013-01-24 07:06:46'::timestamp AS date_time, 0 AS id)
    UNION ALL (SELECT '2013-01-24 07:07:10'::timestamp AS date_time, 0 AS id)
    UNION ALL (SELECT '2013-01-24 07:08:11'::timestamp AS date_time, 0 AS id)
    UNION ALL (SELECT '2013-01-24 09:01:41'::timestamp AS date_time, 0 AS id)
    UNION ALL (SELECT '2013-01-24 09:10:11'::timestamp AS date_time, 0 AS id))
     z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
               date_time    id
0    2013-01-24 07:06:46     0
1    2013-01-24 07:07:10     0
2    2013-01-24 07:08:11     0
3    2013-01-24 09:01:41     0
4    2013-01-24 09:10:11     0
Name: VDF, Number of rows: 5, Number of columns: 2

vdf.sessionize(ts = "date_time", by = ["id"])

#Output
               date_time    id    session_id
0    2013-01-24 07:06:46     0             0
1    2013-01-24 07:07:10     0             0
2    2013-01-24 07:08:11     0             0
3    2013-01-24 09:01:41     0             1
4    2013-01-24 09:10:11     0             1
```

```
Name: VDF, Number of rows: 5, Number of columns: 3
```

7.2.4.74 set_cursor

```
vDataframe.set_cursor(self, cursor)
```

Replace the current cursor with a new one.

- **cursor:** *<object>*
  A Database cursor.

Returns

The vDataframe itself.

7.2.4.75 set_dsn

```
vDataframe.set_dsn(self, dsn: str)
```

Replace the current DSN with a new one.

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

Returns

The vDataframe itself.

7.2.4.76 shape

```
vDataframe.shape(self)
```

Returns the vDataframe shape (Number of rows and columns).

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.shape()

#Output
(1234, 14)
```

7.2.4.77 skewness / skew

```
vDataframe.skewness(self, columns: list = [])
```

Aggregate the different columns by computing the unbiased skewness of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the skewness (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.skew()

#Output
                           skewness
"age"               0.408876460779437
"body"             0.0617701251569532
"survived"           0.56300284427369
"pclass"           -0.576258567091907
"parch"              3.79801928269975
"fare"               4.30069918891405
"sibsp"               3.7597831472411
```

7.2.4.78 sort

```
vDataframe.sort(self, columns: list = [])
```

Sort the vDataframe using specific columns.

Parameters

- **columns:** *<list>*, optional
  The columns used to sort the data.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.sort(["fare"])

#Output
        fare                                     name    boat    sibsp
0       None                     Storey, Mr. Thomas    None        0
1    0.00000    Chisholm, Mr. Roderick Robert Crispin    None        0
2    0.00000                 Andrews, Mr. Thomas Jr    None        0
3    0.00000                      Fry, Mr. Richard    None        0
4    0.00000                  Harrison, Mr. William    None        0
...      ...                                      ...     ...      ...
Name: titanic, Number of rows: 1234, Number of columns: 14
```

7.2.4.79 sql_on_off

```
vDataframe.sql_on_off(self)
```

Prints all the sql queries used by the vDataframe for the different computations. If it is already enable, it will turn it off.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.sql_on_off()
titanic.max()

#Output
$ COMPUTE AGGREGATION(S) $

SELECT MAX("age"),
       MAX("body"),
       MAX("survived"),
       MAX("pclass"),
       MAX("parch"),
       MAX("fare"),
       MAX("sibsp")
FROM
  (SELECT "age" AS "age",
          "body" AS "body",
          "survived" AS "survived",
```

```
            "ticket" AS "ticket",
            "home.dest" AS "home.dest",
            "cabin" AS "cabin",
            "sex" AS "sex",
            "pclass" AS "pclass",
            "embarked" AS "embarked",
            "parch" AS "parch",
            "fare" AS "fare",
            "name" AS "name",
            "boat" AS "boat",
            "sibsp" AS "sibsp"
    FROM public.titanic) final_table
--------------------------------------------------------------------------------
```

7.2.4.80 statistics

```
vDataframe.statistics(self, columns: list = [], skew_kurt_only: bool = False)
```

Summarise the vDataframe with statistical information.

Parameters

- **columns:** *<list>*, optional
  The columns used to compute the mathematical information. If this parameter is empty, the method will consider all the vDataframe numerical columns.
- **skew_kurt_only:** *<bool>*, optional
  Only compute the skewness and kurtosis.

Returns

The `tablesample` type containing the statistical information (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.statistics(columns = ["age", "fare", "pclass"])

#Output
                          "age"              "fare"            "pclass"
count                     997.0              1233.0              1234.0
avg            30.1524573721163    33.9637936739659    2.28444084278768
stddev         14.4353046299159    52.6460729831293   0.842485636190292
min                        0.33                 0.0                 1.0
```

```
10%                   14.5              7.5892                   1.0
25%                   21.0              7.8958                   1.0
median                28.0             14.4542                   3.0
75%                   39.0             31.3875                   3.0
90%                   50.0               79.13                   3.0
max                   80.0            512.3292                   3.0
skewness   0.408876460779437    4.30069918891405    -0.576258567091907
kurtosis    0.15689691331997    26.2543152552867     -1.34962169484619
```

7.2.4.81 std

```
vDataframe.std(self, columns: list = [])
```

Aggregate the different columns by computing the standard deviation of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe Columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the std (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.std()

#Output
                            stddev
"age"                 14.4353046299159
"body"                96.5760207557808
"survived"           0.481532018641288
"pclass"             0.842485636190292
"parch"              0.868604707790392
"fare"                52.6460729831293
"sibsp"               1.04111727241629
```

7.2.4.82 sum

```
vDataframe.sum(self, columns: list = [])
```

Aggregate the different columns by computing the sum of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the sums (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.sum()

#Output
                          sum
"age"                 30062.0
"body"                19369.0
"survived"              450.0
"pclass"               2819.0
"parch"                 467.0
"fare"               41877.3576
"sibsp"                 622.0
```

7.2.4.83 tail

```
vDataframe.tail(self, limit: int = 5, offset: int = 0)
```

Returns a part of the vDataframe. The tail is not necessary the end of the object.

Parameters

- **limit:** *<int>*, optional
  The number of elements to return.
- **offset:** *<int>*, optional
  The number of elements to skip.

Returns

The `tablesample` type containing the tail of the vDataframe (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm.tail(limit = 10, offset = 1000)

#Output
                       time          val     id
1000  2014-02-22 16:00:00    0.1250000      9
1001  2014-02-22 16:30:00    0.8130000      5
1002  2014-02-22 18:00:00    0.3360000      3
1003  2014-02-22 19:30:00    0.3020000      5
1004  2014-02-22 20:00:00    0.6970000      0
1005  2014-02-22 21:45:00    0.2960000      8
1006  2014-02-22 21:45:00    0.3090000      9
1007  2014-02-22 22:00:00    0.3390000      4
1008  2014-02-22 22:30:00    0.3500000      1
1009  2014-02-22 23:15:00    0.5830000      8
...                        ...          ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3
```

7.2.4.84 time_on_off

```
vDataframe.time_on_off(self)
```

Prints all the vDataframe queries elapsed time. If it is already enable, it will turn it off.

Returns

The vDataframe itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.time_on_off()
titanic.max()

#Output
Elapsed Time: 0.009672880172729492
----------------------------------------------------------------------------------
```

7.2.4.85 to_csv

```
vDataframe.to_csv(
        self,
        name: str,
        path: str = '',
        sep: str = ',',
        na_rep: str = '',
        quotechar: str = '"',
        usecols: list = [],
        header: bool = True,
        new_header: list = [],
        order_by: list = [],
        nb_row_per_work: int = 0)
```

Create a csv file from the vDataframe relation.

Parameters

- **name:** *<str>*
  File Name.
- **path:** *<str>*, optional
  File Path.
- **sep:** *<str>*, optional
  Elements separator.
- **na_rep:** *<str>*, optional
  How to represent missing values.
- **quotechar:** *<str>*, optional
  How to enclose the varchar.
- **usecols:** *<list>*, optional
  Columns to use to write the csv file. If it is empty, all the vDataframe columns will be used.
- **header:** *<bool>*, optional
  Write the header.
- **new_header:** *<list>*, optional
  Replace the default header by a new one.
- **order_by:** *<list>*, optional
  Order the elements using specific columns before writing.
- **nb_row_per_work:** *<int>*, optional
  Number of rows to be read during each process of the loop. This parameter can lead to a partially wrong csv files if the data are not sorted correctly. Leave it to 0 to read the entire file on one time.

Returns

The vDataframe itself.

7.2.4.86 to_db

```
vDataframe.to_db(
        self
        name: str,
        usecols: list = [],
        relation_type: str = "view")
```

Save the vDataframe relation in the Database. The vDataframe will be then recreated from scratch using the new relation.

Parameters

- **name:** *<str>*
  Name of the new relation.
- **usecols:** *<list>*, optional
  List of the vDataframe columns to use.
- **relation_type:** *<str>*, optional
  The relation type. This parameter can be a 'view' or a 'table'.

Returns

The vDataframe itself.

7.2.4.87 to_pandas

```
vDataframe.to_pandas(self)
```

Convert the vDataframe to a pandas.DataFrame. Be careful, if the volume is huge it can break down the system.

Returns

The pandas.Dataframe of the vDataframe.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.to_pandas()

#Output
      age   body  survived    ticket                       home.dest  \
0   2.000    NaN         0    113781  Montreal, PQ / Chesterville, ON
1  30.000  135.0         0    113781  Montreal, PQ / Chesterville, ON
2  25.000    NaN         0    113781  Montreal, PQ / Chesterville, ON
3  39.000    NaN         0    112050                     Belfast, NI
4  71.000   22.0         0  PC 17609             Montevideo, Uruguay
```

```
      cabin     sex  pclass embarked  parch      fare  \
0   C22 C26  female       1        S      2  151.55000
1   C22 C26    male       1        S      2  151.55000
2   C22 C26  female       1        S      2  151.55000
3       A36    male       1        S      0    0.00000
4      None    male       1        C      0   49.50420

                                               name  boat  sibsp
0                     Allison, Miss. Helen Loraine  None      1
1             Allison, Mr. Hudson Joshua Creighton  None      1
2   Allison, Mrs. Hudson J C (Bessie Waldo Daniels)  None      1
3                           Andrews, Mr. Thomas Jr  None      0
4                          Artagaveytia, Mr. Ramon  None      0
[1234 rows x 14 columns]
```

7.2.4.88 to_vdf

```
vDataframe.to_vdf(self, name: str)
```

Save the vDataframe to the vdf format. You can use `read_vdf` to read this file type and load the vDataframe.

Parameters

- **name:** *<str>*
  Name of the vdf file

Returns

The vDataframe itself.

7.2.4.89 var

```
vDataframe.var(self, columns: list = [])
```

Aggregate the different columns by computing the variance of each one.

Parameters

- **columns:** *<list>*, optional
  List of the vDataframe columns. If this parameter is empty, the method will consider all the numerical columns.

Returns

The `tablesample` type containing the var (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic.var()

#Output
                            variance
"age"               208.378019758472
"body"              9326.92778502101
"survived"         0.231873084976754
"pclass"           0.709782047186962
"parch"            0.754474138395633
"fare"              2771.60900054498
"sibsp"             1.08392517492353
```

7.2.4.90 version

```
vDataframe.version(self)
```

Returns the Vertica DB version and useful information on the vDataframe.

Example

```
titanic.version()

#Output

#
# VERTICA-ML-PYTHON
#
# Author: Badr Ouali, Datascientist at Vertica
#
# You are currently using Vertica Analytic Database v9.2.1-0
#
# You have a perfectly adapted version for using vDataframe and Vertica ML
#
# For more information about the vDataframe you can use the help() method
(9, 2)
```

7.3 Virtual Column

7.3.1 attributes

When the vDataframe is created, it will creates as many vColumn as there are columns in the input relation. The vColumn has only 3 attributes.

- `parent`: the vColumn parent (must be a vDataframe).
- `alias`: the alias of the column.
- `transformations`: list of all the Virtual Column transformations.

7.3.2 methods

7.3.2.1 abs

```
vColumn.abs(self)
```

Apply the abs function to the column elements.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].abs()

#Output
      x
0     5
1     1
2     2
3     9
Name: x, Number of rows: 4, dtype: int
```

7.3.2.2 add

```
vColumn.add(self, x: float)
```

Add a float to the column elements.

Parameters

- **x:** *<float>*
  A float number.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].add(x = 2)

#Output
       x
0      7
1      1
2      4
3     -7
Name: x, Number of rows: 4, dtype: int
```

7.3.2.3 add_copy

```
vColumn.add_copy(self, name: str)
```

Add a column copy to the vDataframe.

Parameters

- **name:** *<str>*
  Copy name.

Returns

The vColumn copy.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].add_copy(name = "y")

#Output
       x      y
0      5      5
1     -1     -1
2      2      2
3     -9     -9
Name: VDF, Number of rows: 4, Number of columns: 2
```

7.3.2.4 add_prefix

```
vColumn.add_prefix(self, prefix: str)
```

Add a prefix to each of the column element.

Parameters

- **prefix:** *<str>*
  The prefix.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
     x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].add_prefix(prefix = "elem_")

#Output
            x
0      elem_5
1     elem_-1
2      elem_2
3     elem_-9
Name: x, Number of rows: 4, dtype: varchar(25)
```

7.3.2.5 add_suffix

```
vColumn.add_suffix(self, suffix: str)
```

Add a suffix to each of the column element.

Parameters

- **suffix:** *<str>*
  The suffix.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
     x) UNION ALL (SELECT -9 AS x)) z"
```

```
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
      x
0     5
1    -1
2     2
3    -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].add_suffix(prefix = "_end")

#Output
          x
0      5_end
1     -1_end
2      2_end
3     -9_end
Name: x, Number of rows: 4, dtype: varchar(24)
```

7.3.2.6 aggregate / agg

```
vDataframe.aggregate(self, func: list)
```

Aggregate the columns using the input aggregations.

Parameters

- **func:** *<list>*
  List of the different aggregations.

Returns

The `tablesample` type containing the aggregations (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].agg(["sum", "min"])

#Output
             "age"
sum        30062.0
```

```
min             0.33
```

7.3.2.7 apply

```
vDataframe.apply(self, func: str, copy: bool = False, copy_name: str = "")
```

Apply the input function to the column.

Parameters

- **func:** *<str>*
  The function to apply. The function's variable must be written using flower brackets '{}' (Example: EXP({}))
- **copy:** *<bool>*, optional
  Create a copy of the column in the vDataframe and apply the function on it.
- **copy_name:** *<str>*, optional
  Copy name.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 1 AS x, 4 AS y) UNION ALL (SELECT 2 AS x, 9 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
     x    y
0    1    4
1    2    9
Name: VDF, Number of rows: 2, Number of columns: 2

vdf["x"].apply('EXP({})')

#Output
                    x    y
0    2.71828182845905    4
1    7.38905609893065    9
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.3.2.8 astype

```
vColumn.astype(self, dtype: str)
```

Convert the column to a new data type.

Parameters

- **dtype:** *<str>*
  The new data type.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"].dtype()

#Output
int

vdf["x"].astype(dtype = "varchar(3)").dtype()

#Output
varchar(3)
```

7.3.2.9 avg / mean

```
vColumn.avg(self)
```

Returns the column average.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].avg()

#Output
30.1524573721163
```

7.3.2.10 bar

```
vColumn.bar(
      self,
      method: str = "density",
      of: str = "",
      max_cardinality: int = 6,
      bins: int = 0,
      h: float = 0,
      color: str = '#214579')
```

Draw the column bar chart.

Parameters

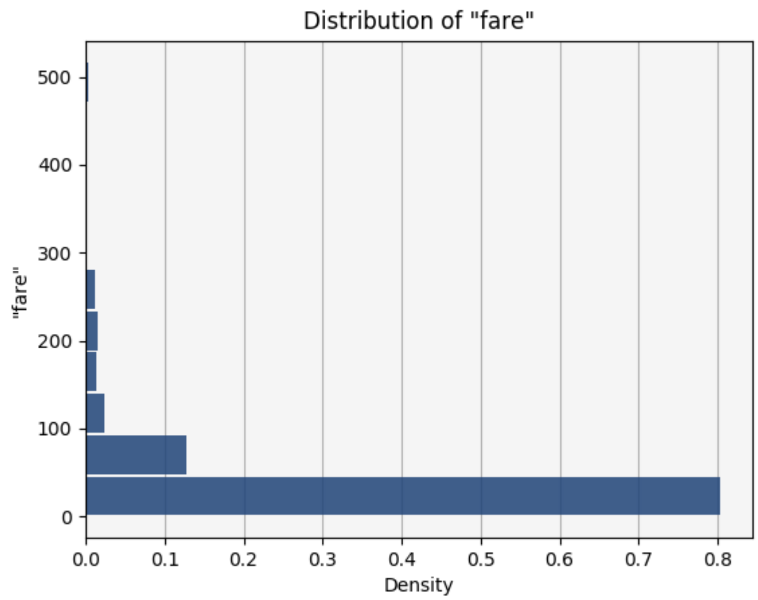
- **method:** *<str>*, optional
  count | density | avg | min | max | sum
  count: count is used as aggregation
  density (default): density is used as aggregation
  avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
  The column used to compute the aggregation. This variable is used only if "method" in {avg | min |max | sum}
- **max_cardinality:** *<int>*, optional
  The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **bins:** *<int>*, optional
  The number of bins of the histogram.
- **h:** *<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:** *<str>*, optional
  The histogram color.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].bar()
```



7.3.2.11 boxplot

```
vColumn.boxplot(
        self,
        by: str = "",
        h: float = 0,
        max_cardinality: int = 8,
        cat_priority: list = [])
```

Draw the column boxplot.

Parameters

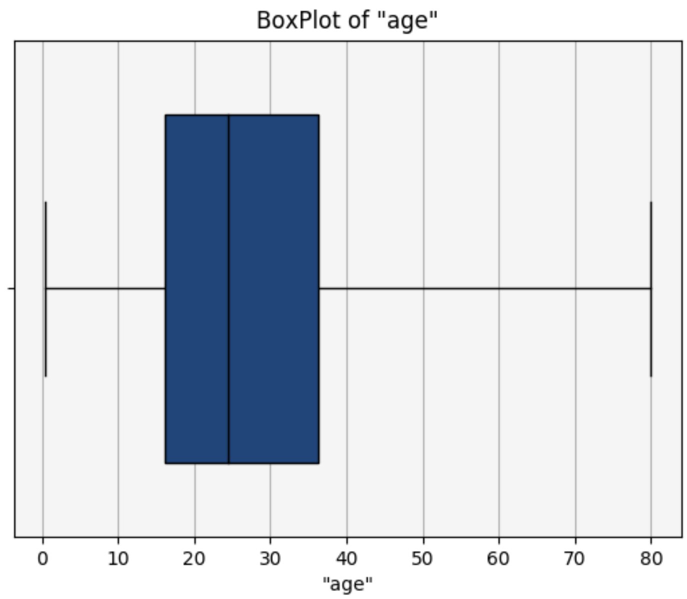
- **by:** *<str>*, optional
  The group by column. It is uses to split the different categories to draw a multi boxplot.
- **max_cardinality:** *<int>*, optional
  The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **h:** *<float>*, optional
  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>*, optional
  The principal categories to show.

Returns

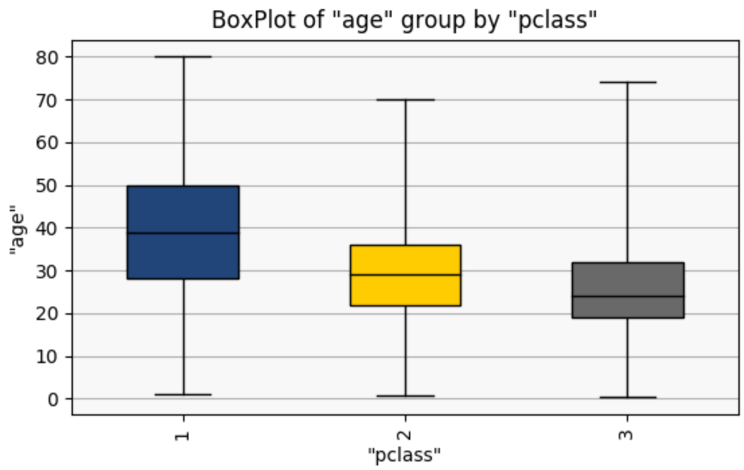
The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].boxplot()
```




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



7.3.2.12 category

```
vColumn.category(self)
```

Returns the column category.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"].category()

#Output
text
```

7.3.2.13 clip

```
vColumn.clip(self, lower = None, upper = None)
```

Clip the data by transforming the values lesser than the lower bound to the lower bound itself and the values higher than the upper bound to the upper bound itself. The column will be transformed.

Parameters

- **lower:** <*float*>, optional
  Lower bound.
- **upper:** <*float*>, optional
  Upper bound.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
       x
0      5
1     -1
2      2
```

```
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].clip(lower = -1, upper = 1)

#Output
       x
0      1
1     -1
2      1
3     -1
Name: x, Number of rows: 4, dtype: int
```

7.3.2.14 count

```
vColumn.count(self)
```

Returns the column count (number of non-NULL elements).

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].count()

#Output
997
```

7.3.2.15 date_part

```
vColumn.date_part(self, field: str)
```

Extract a specific field from the column (only if the column is a timestamp). The column will be transformed.

Parameters

- **field:** *<str>*
  The field to extract. It can be in {CENTURY | DAY | DECADE | DOQ | DOW | DOY | EPOCH | HOUR | ISODOW | ISOWEEK | ISOYEAR | MICROSECONDS | MILLENNIUM | MILLISECONDS | MINUTE | MONTH | QUARTER | SECOND | TIME ZONE | TIMEZONE_HOUR | TIMEZONE_MINUTE | WEEK | YEAR}

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)

#Output
                    time            val     id
0    2014-01-01 01:15:00    0.0370000      2
1    2014-01-01 02:30:00    0.0800000      5
2    2014-01-01 03:00:00    0.0810000      1
3    2014-01-01 05:00:00    1.4890000      3
4    2014-01-01 06:00:00    0.0720000      5
...                  ...          ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3

sm["time"].date_part("month")

#Output
     time           val     id
0       1     0.0370000      2
1       1     0.0800000      5
2       1     0.0810000      1
3       1     1.4890000      3
4       1     0.0720000      5
...   ...           ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3
```

7.3.2.16 decode

```
vColumn.decode(self, values: dict, others = None)
```

Encode the data using the input bijection.

Parameters

- **values:** *<dict>*
  Dictionary of values representing the bijection used to encode the data.
- **others:** *<float>*, optional
  How to encode the values which are not in the dictionary of values.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["sex"]

#Output
        sex
0     female
1       male
2     female
3       male
4       male
...      ...
Name: sex, Number of rows: 1234, dtype: varchar(20)

titanic["sex"].decode(values = {"male": 1, "female": 0})

#Output
     sex
0       0
1       1
2       0
3       1
4       1
...   ...
Name: sex, Number of rows: 1234, dtype: int
```

7.3.2.17 density

```
vColumn.density(
        self,
        a = None,
        kernel: str = "gaussian",
        smooth: int = 200,
        color: str = '#214579')
```

Draw the column 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.

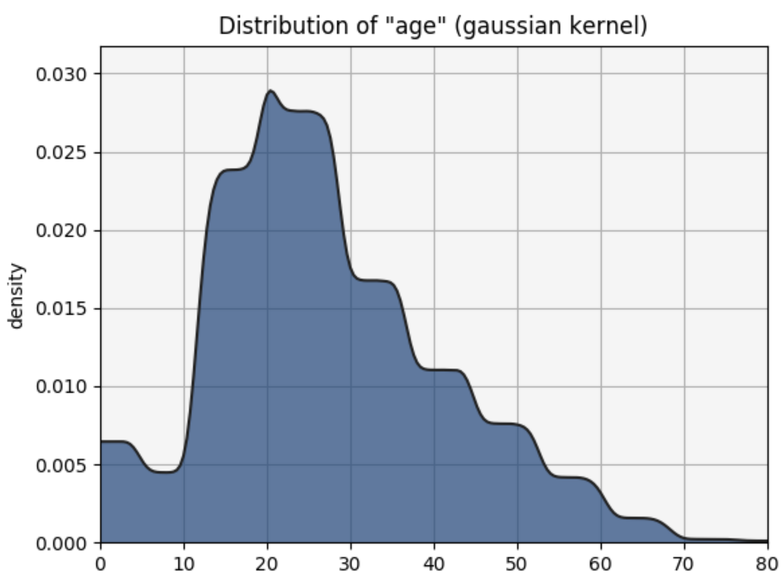
- **smooth:** *<positive int>*, optional
  The number of points used for the smoothing.
- **color:** *<str>*, optional
  The density plot color.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].density()
```



7.3.2.18 describe

```
vColumn.describe(method: str = "auto", max_cardinality: int = 6)
```

Summarise the column with mathematical information.

Parameters

- **method:** *<str>*, optional
  auto | categorical | numerical | cat_stats
  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 $cardinality \leq max\_cardinality$).
  cat_stats: This mode is used to compute descriptive statistics

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

Returns

The `tablesample` type containing the mathematical information (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Note

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

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].describe()

#Output
                         value
name                     "age"
dtype             numeric(6,3)
unique                      96
count                    997.0
mean          30.1524573721163
std           14.4353046299159
min                       0.33
25%                       21.0
50%                       28.0
75%                       39.0
max                       80.0

titanic["pclass"].describe()

#Output
                 value
name          "pclass"
dtype              int
unique               3
3                  663
1                  312
2                  259
```

7.3.2.19 distinct

```
vColumn.distinct(self)
```

Returns all the columns distinct elements.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["pclass"].distinct()

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

7.3.2.20 divide / div

```
vColumn.divide(self, x: float)
```

Div the column by a float.

Parameters

- **x:** *<float>*
  A float number (!= 0).

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
      x
0     5
1    -1
2     2
3    -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].div(x = 5)

#Output
                          x
```

```
0      1.000000000000000000
1     -0.200000000000000000
2      0.400000000000000000
3     -1.800000000000000000
Name: x, Number of rows: 4, dtype: numeric(36,18)
```

7.3.2.21 donut

```
vColumn.donut(
        self,
        method: str = "density",
        of: str = "",
        max_cardinality: int = 6,
        h: float = 0)
```

Draw the column donut chart.

Parameters

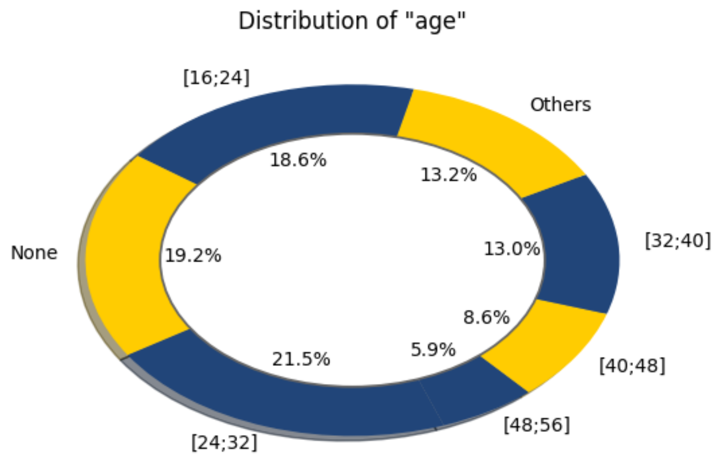
- **method:** *<str>*, optional
  count | density | avg | min | max | sum
  count: count is used as aggregation
  density (default): density is used as aggregation
  avg | min | max | sum: these aggregations are used only if "of" is informed
- **of:** *<str>*, optional
  The column used to compute the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

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].donut()
```



7.3.2.22 drop

```
vColumn.drop(self)
```

Drop the column from the vDataframe.

Returns

The parent vDataframe.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x, -1 AS y) UNION ALL (SELECT -1 AS x, 10 AS y)
   UNION ALL (SELECT 2 AS x, 3 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
       x     y
0      5    -1
1     -1    10
2      2     3
Name: VDF, Number of rows: 3, Number of columns: 2

vdf["x"].drop()

#Output
       y
```

```
0     -1
1     10
2      3
Name: y, Number of rows: 3, dtype: int
```

7.3.2.23 dropna

```
vColumn.dropna(self)
```

Drop the elements having missing values.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x, -1 AS y) UNION ALL (SELECT NULL AS x, 10 AS y)
   UNION ALL (SELECT 2 AS x, 3 AS y)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
         x      y
0        5     -1
1     None     10
2        2      3
Name: VDF, Number of rows: 3, Number of columns: 2

vdf["x"].dropna()

#Output
      x      y
0     5     -1
1     2      3
Name: VDF, Number of rows: 2, Number of columns: 2
```

7.3.2.24 drop_outliers

```
vColumn.drop_outliers(self, alpha: float = 0.05)
```

Drop the columns outliers.

Parameters

- **alpha:** *<float>*
  Number representing the outliers threshold. Values lesser than quantile(alpha) or greater than quantile(1-alpha) will be dropped.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x) UNION ALL (SELECT -600 AS x) UNION ALL (
    SELECT 100000 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
           x
0          5
1         -1
2          2
3         -9
4       -600
5     100000
Name: x, Number of rows: 6, dtype: int

vdf["x"].div(x = 5)

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int
```

7.3.2.25 dtype

```
vColumn.dtype(self)
```

Returns the column data type.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"].dtype()

#Output
varchar(164)
```

7.3.2.26 ema

```
ema(self, ts: str, by: list = [], alpha: float = 0.5)
```

Compute the exponential moving average of the column using an input time series.

Parameters

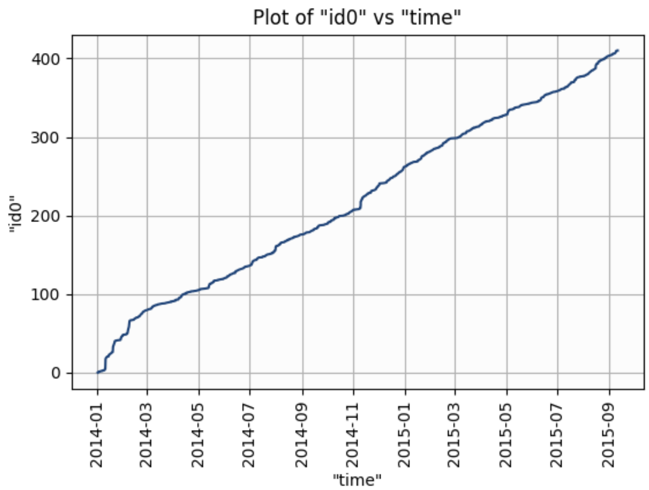
- **ts:** *<str>*
  The vDataframe time series.
- **by:** *<list>*, optional
  How to group the data.
- **alpha:** *<float>*, optional
  the EMA smoothing factor.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
# Computing the cum sum of each home
sm.cumsum(name = "val_cumsum", column = "val", by = ["id"], order_by = ["time"
    ])
# Building the features corresponding to the cum consumption of the home id 0
sm.eval("id0", "DECODE(id, 0, val_cumsum, NULL)")
# Applying the EMA
sm["id0"].ema(ts = "time", by = ["id"])
# Drawing the time series
sm["id0"].plot(ts = "time")
```



7.3.2.27 equals / eq

```
vColumn.equals(self, x)
```

Verify if each element of the column is equal to the input element.

Parameters

- **x:** *<anytype>*
  Float, int or varchar.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
        x
0       5
1      -1
2       2
3      -9
Name: x, Number of rows: 4, dtype: int

```

```
vdf["x"].eq(x = -1)

#Output
          x
0     False
1      True
2     False
3     False
Name: x, Number of rows: 4, dtype: boolean
```

7.3.2.28 fillna

```
vDataframe.fillna(
             self,
             val = None,
             method: str = "auto",
             by: list = [],
             order_by: list = [])
```

Fill the missing values using the input method. If the parameters val and method are empty, all the missing values will be filled automatically (using the average of the column for the numerical columns and the mode for the categorical ones)

Parameters

- **val:** *<anytype>*, optional
  Constant value used to impute the column.
- **method:** *<dict>*, optional
  Method used to impute the column.
  auto | avg | median | mode | ffill | backfill
  auto (default): average for numerical and mode for categorical.
  avg: Imputation using the column average.
  median: Imputation using the column median.
  mode: Imputation using the column mode (most occurrent element).
  ffill: Propagation of the first non-NULL element = constant interpolation (only for time series, `order_by` parameter must be defined).
  bfill: Back Propagation of the next non-NULL element = constant interpolation (only for time series, `order_by` parameter must be defined).
- **by:** *<list>*, optional
  The columns used to group the main one.
- **order_by:** *<list>*, optional
  The columns used to order the data (used only when method is bfill or ffill)

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT NULL AS x) UNION ALL (SELECT 2
    AS x) UNION ALL (SELECT NULL AS x) UNION ALL (SELECT -2 AS x) UNION ALL (
    SELECT 12 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
         x
0        5
1     None
2        2
3     None
4       -2
5       12
Name: x, Number of rows: 6, dtype: int

vdf["x"].fillna()

#Output
          x
0      5.00
1      4.25
2      2.00
3      4.25
4     -2.00
5     12.00
Name: x, Number of rows: 6, dtype: float
```

7.3.2.29 fill_outliers

```
vDataframe.fill_outliers(self, method: str = "winsorize", alpha = 0.05)
```

Fill the outliers using the input method.

Parameters

- **method:** *<dict>*, optional
  Method used to fill the column outliers.
  winsorize | null | mean
  winsorize (default): clip the data using as lower bound quantile(alpha) and as upper bound quantile(1-alpha).
  null: Replace the outliers by the NULL value.
  mean: Replace the upper and lower outliers by their respective average.
- **alpha:** *<float>*
  Number representing the outliers threshold. Values lesser than quantile(alpha) or greater than quantile(1-alpha) will be filled.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT 600 AS x) UNION ALL (SELECT 2
    AS x) UNION ALL (SELECT -100 AS x) UNION ALL (SELECT -2 AS x) UNION ALL (
    SELECT 12 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
        x
0       5
1     600
2       2
3    -100
4      -2
5      12
Name: x, Number of rows: 6, dtype: int

vdf["x"].fill_outliers(method = "null")

#Output
        x
0       5
1    None
2       2
3    None
4      -2
5      12
Name: x, Number of rows: 6, dtype: int
```

7.3.2.30 ge

```
vColumn.ge(self, x: float)
```

Verify if each element of the column is greater or equal to the input element.

Parameters

- **x:** *<float>*
  A numerical element.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].ge(x = 2)

#Output
          x
0      True
1     False
2      True
3     False
Name: x, Number of rows: 4, dtype: boolean
```

7.3.2.31 get_dummies

```
vColumn.get_dummies(
          self,
          prefix: str = "",
          prefix_sep: str = "_",
          drop_first: bool = False,
          use_numbers_as_suffix: bool = False)
```

Compute the different columns dummies and add them to the vDataframe.

Parameters

- **prefix:** *<str>*, optional
  Prefix of the dummies.
- **prefix_sep:** *<str>*, optional
  Prefix delimitor of the dummies.

- **drop_first:** *<bool>*, optional
  Drop the first dummy to avoid the creation of correlated features.
- **use_numbers_as_suffix:** *<bool>*, optional
  Use numbers as suffix instead of the different categories.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic = titanic.select(["embarked"])

#Output
     embarked
0           S
1           S
2           S
3           S
4           C
...       ...
Name: embarked, Number of rows: 1234, dtype: varchar(20)

titanic["embarked"].get_dummies()

#Output
     embarked    embarked_C    embarked_Q    embarked_S
0           S             0             0             1
1           S             0             0             1
2           S             0             0             1
3           S             0             0             1
4           C             1             0             0
...       ...           ...           ...           ...
Name: titanic, Number of rows: 1234, Number of columns: 4
```

7.3.2.32 gt

```
vColumn.gt(self, x: float)
```

Verify if each element of the column is greater to the input element.

Parameters

- **x:** *<float>*
  A numerical element.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
      x
0     5
1    -1
2     2
3    -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].gt(x = 2)

#Output
         x
0      True
1     False
2     False
3     False
Name: x, Number of rows: 4, dtype: boolean
```

7.3.2.33 head

```
vColumn.head(self, limit: int = 5)
```

Returns the column head.

Parameters

- **limit:** *<int>*, optional
  The number of elements to return.

Returns

The `tablesample` type containing the column head (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
```

```
titanic["fare"].head(limit = 5)

#Output
          fare
0     151.55000
1     151.55000
2     151.55000
3       0.00000
4      49.50420
...         ...
Name: fare, Number of rows: 1234, dtype: numeric(10,5)
```

7.3.2.34 hist

```
vColumn.hist(
      self,
      method: str = "density",
      of: str = "",
      max_cardinality: int = 6,
      bins: int = 0,
      h: float = 0,
      color: str = '#214579')
```

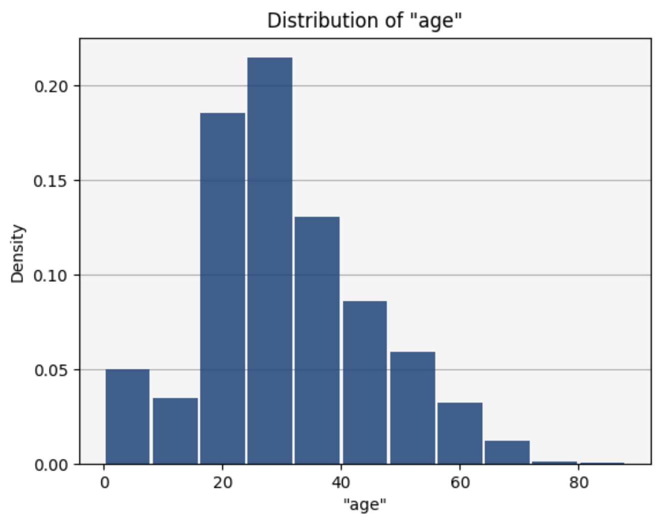
Draw the column histogram.

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:** *<int>*, optional
  The maximum cardinality of the column. Under this number the column is automatically considered as categorical.
- **bins:** *<int>*, optional
  The number of bins of the histogram.
- **h:** *<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:** *<str>*, optional
  The histogram color.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].hist()
```



7.3.2.35 isdate

```
vColumn.isdate(self)
```

Returns True if the column is a date.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].isdate()

#Output
 False
```

7.3.2.36 isin

```
vDataframe.isin(self, val: list)
```

Verify if the different elements are in the column.

Parameters

- **val:** *<list>*
  List of values to check.

Returns

The list containing the booleans

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
# We verify if there was someone having 18 years old and someone having 52
   years old
titanic["age"].isin(val = ["18", "52"])

#Output
[True, True]
```

7.3.2.37 isnum

```
vColumn.isnum(self)
```

Returns True if the column is numerical.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].isnum()

#Output
True
```

7.3.2.38 kurtosis / kurt

```
vColumn.kurtosis(self)
```

Returns the column unbiased kurtosis using Fisher?s definition.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].kurt()

#Output
0.15689691331997
```

7.3.2.39 label_encode

```
vColumn.label_encode(self)
```

Encode the column using a bijection from the different categories to [0, n - 1] (n being the number of elements).

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic = titanic.select(["embarked"])

#Output
     embarked
0           S
1           S
2           S
3           S
4           C
...       ...
Name: embarked, Number of rows: 1234, dtype: varchar(20)

titanic["embarked"].label_encode()

#Output
     embarked
0           2
1           2
2           2
3           2
4           0
...       ...
Name: embarked, Number of rows: 1234, dtype: int
```

7.3.2.40 le

```
vColumn.le(self, x: float)
```

Verify if each element of the column is lesser or equal to the input element.

Parameters

- **x:** *<float>*
  A numerical element.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
     x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].le(x = 2)

#Output
           x
0     False
1      True
2      True
3      True
Name: x, Number of rows: 4, dtype: boolean
```

7.3.2.41 lt

```
vColumn.lt(self, x: float)
```

Verify if each element of the column is lesser to the input element.

Parameters

- **x:** *<float>*
  A numerical element.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].lt(x = 2)

#Output
           x
0      False
1       True
2      False
3       True
Name: x, Number of rows: 4, dtype: boolean
```

7.3.2.42 mad

```
vColumn.mad(self)
```

Returns the mean absolute deviation of the column.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].mad()

#Output
11.254785419447906
```

7.3.2.43 max

```
vColumn.max(self)
```

Returns the column max.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].max()

#Output
80.0
```

7.3.2.44 mean_encode

```
vColumn.mean_encode(self, response_column: str)
```

Encode the column using the average of the response column for the different categories.

Parameters

- **response_column:** *<str>*
  The response column.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic = titanic.select(["embarked", "survived"])
```

```
#Output
      survived    embarked
0            0           S
1            0           S
2            0           S
3            0           S
4            0           C
...        ...         ...
Name: titanic, Number of rows: 1234, Number of columns: 2

titanic["embarked"].mean_encode(response_column = "survived")

#Output
      survived              embarked
0            1     0.537549407114625
1            1     0.537549407114625
2            1     0.537549407114625
3            1     0.537549407114625
4            1     0.537549407114625
...        ...                   ...
Name: titanic, Number of rows: 1234, Number of columns: 2
```

7.3.2.45 median

```
vColumn.median(self)
```

Returns the column median.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].median()

#Output
28.0
```

7.3.2.46 min

```
vColumn.min(self)
```

Returns the column min.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].min()

#Output
0.33
```

7.3.2.47 mod

```
vColumn.mod(self, n: int)
```

Apply the mod(n) function to the column elements.

Parameters

- **n:** *<int>*
  An integer.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
     x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].mod(n = 2)

#Output
       x
0      1
1     -1
```

```
2      0
3     -1
Name: x, Number of rows: 4, dtype: int
```

7.3.2.48 mode

```
vColumn.mode(self, dropna: bool = True)
```

Returns the column mode (most occurrent element).

Parameters

- **dropna:** *<bool>*, optional
  Do not consider null values as a category.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["pclass"].mode()

#Output
3
```

7.3.2.49 mul

```
vColumn.mul(self, x: float)
```

Multiply all the column elements by a float.

Parameters

- **x:** *<float>*
  A number.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
      x
0     5
1    -1
2     2
3    -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].mul(x = 3)

#Output
       x
0     15
1     -3
2      6
3    -27
Name: x, Number of rows: 4, dtype: int
```

7.3.2.50 neq

```
vColumn.neq(self, x)
```

Verify if each element of the column is not equal to the input element.

Parameters

- **x:** *<anytype>*
  Float, int or varchar.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
```

```
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].neq(x = -1)

#Output
          x
0      True
1     False
2      True
3      True
Name: x, Number of rows: 4, dtype: boolean
```

7.3.2.51 next

```
vColumn.next(self, order_by: list, by: list = [])
```

Replace the element of the column by the next one.

Parameters

- **order_by:** *<list>*
  How to order the data.
- **by:** *<list>*, optional
  How to group the data.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm["val"].add_copy("next")
sm["next"].next(order_by = ["time"], by = ["id"])

#Output
                     time            val     id           next
```

```
0     2014-01-01 11:00:00     0.0290000      0     0.2770000
1     2014-01-01 13:45:00     0.2770000      0     0.3210000
2     2014-01-02 10:45:00     0.3210000      0     0.3050000
3     2014-01-02 11:15:00     0.3050000      0     0.3580000
4     2014-01-02 13:45:00     0.3580000      0     0.1150000
...                   ...           ...    ...           ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.3.2.52 normalize

```
vColumn.normalize(self, method: str = "zscore")
```

normalize the column with a specific method.

Parameters

- **method:** *<str>*, optional
  zscore (default) | robust_zscore | minmax
  The method used for the normalization.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].normalize()

#Output
                                fare
0      2.23352283775686733061637440003
1      2.23352283775686733061637440003
2      2.23352283775686733061637440003
3     -0.64513441838007118274832515810
4      0.29518642978392902545562574840
...                              ...
Name: fare, Number of rows: 1234, dtype: float
```

7.3.2.53 numh

```
vColumn.numh(self, method: str = "auto")
```

Returns the interval size to convert the column to categorical using a specific method.

Parameters

- **method:** *<str>*, optional
  auto | sturges | freedman_diaconis
  auto (default): max of the interval computed by the two available method.
  sturges: Use Sturges definition of the best h.
  freedman_diaconis: Use Freedman Diaconis definition of the best h.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].numh()

#Output
47
```

7.3.2.54 nunique

```
vColumn.nunique(self)
```

Returns the column cardinality.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["pclass"].nunique()

#Output
3
```

7.3.2.55 pct_change

```
vColumn.pct_change(self, order_by: list, by: list = [])
```

Compute the ratio between the current value and the previous one. The column will be transformed.

Parameters

- **order_by:** *<list>*
  The list of elements to order the vDataframe during the process.

- **by:** *<list>*, optional
  The columns used to group the main column.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)

#Output
                    time           val     id
0     2014-01-01 01:15:00     0.0370000      2
1     2014-01-01 02:30:00     0.0800000      5
2     2014-01-01 03:00:00     0.0810000      1
3     2014-01-01 05:00:00     1.4890000      3
4     2014-01-01 06:00:00     0.0720000      5
...                   ...           ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3

sm["val"].pct_change(order_by = ["time"], by = ["id"])

#Output
                    time                     val     id
0     2014-01-01 11:00:00     9.551724137931034483      0
1     2014-01-01 13:45:00     1.158844765342960289      0
2     2014-01-02 10:45:00     0.950155763239875389      0
3     2014-01-02 11:15:00     1.173770491803278689      0
4     2014-01-02 13:45:00     0.321229050279329609      0
...                   ...                      ...    ...
Name: smart_meters, Number of rows: 11844, Number of columns: 3
```

7.3.2.56 pie

```
vColumn.pie(
        self,
        method: str = "density",
        of: str = "",
        max_cardinality: int = 6,
        h: float = 0)
```

Draw the column pie chart.

Parameters

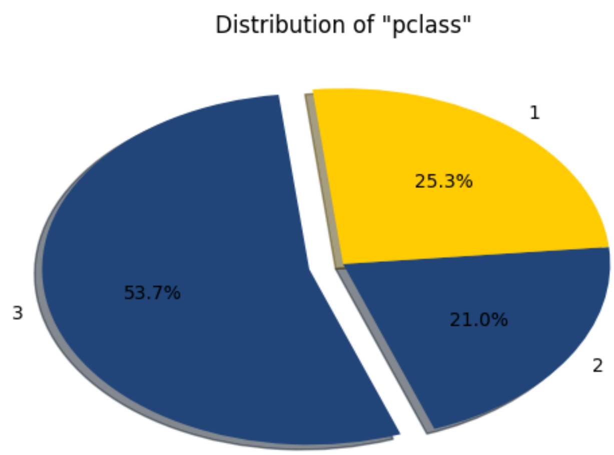
- **method:** *<str>*, optional
  count | density | avg | min | max | sum
  count: count is used as aggregation

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:** *<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.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["pclass"].pie()
```



7.3.2.57 plot

```
vDataframe.plot(
        self,
        ts: str,
        by: str,
        start_date: str = "",
        end_date: str = ""
        color: str = '#214579',
```

```
        area: bool = False)
```

Plot the time series of the column.

Parameters

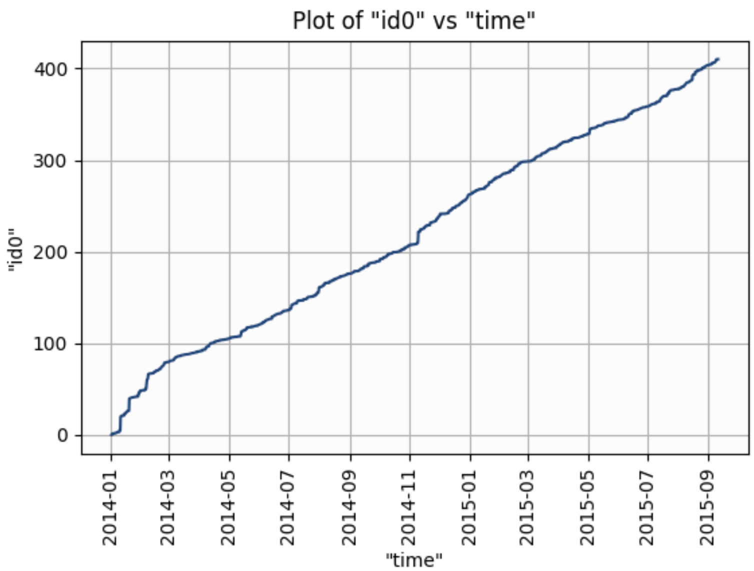
- **ts:** *<str>*
  The time series used to plot the different elements.
- **by:** *<str>*, optional
  The column to group with.
- **start_date:** *<str>*, optional
  Start Date.
- **end_date:** *<str>*, optional
  End Date.
- **color:** *<str>*, optional
  Plot color.
- **area:** *<bool>*, optional
  To plot an area plot.

Returns

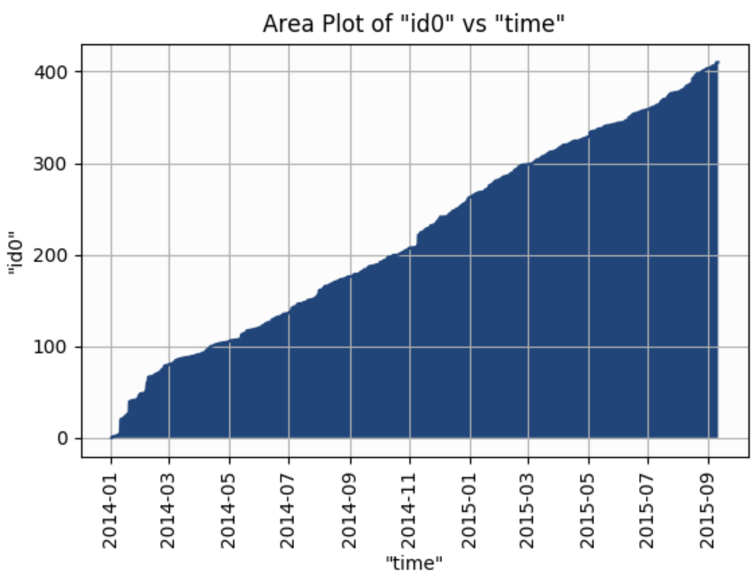
The vColumn itself.

Example

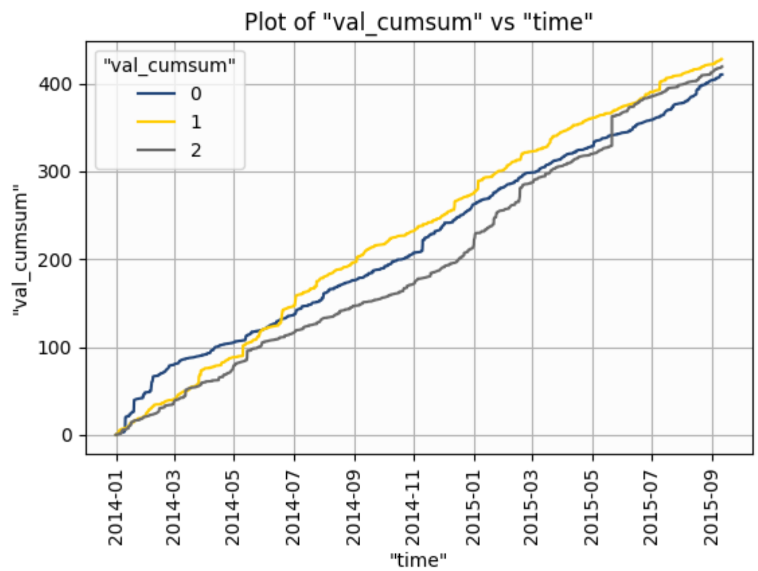
```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
# Computing the cum sum of each home
sm.cumsum(name = "val_cumsum", column = "val", by = ["id"], order_by = ["time"
    ])
# Building the features corresponding to the cum consumption of the home id 0
sm.eval("id0", "DECODE(id, 0, val_cumsum, NULL)")
# Drawing the time series
sm["id0"].plot(ts = "time")
```




```
sm["id0"].plot(ts = "time", area = True)
```




```
# Plot by grouping by id
sm.cumsum(name = "val_cumsum", column = "val", by = ["id"], order_by = ["time"]).filter("id <= 2")
sm["val_cumsum"].plot(ts = "time", by = "id")
```



7.3.2.58 pow

```
vColumn.pow(self, x: float)
```

Apply the pow(x) function to the column elements.

Parameters

- **x:** *<float>*
  A float representing the power exponent.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
        x
0       5
1      -1
2       2
```

```
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].pow(x = 2)

#Output
          x
0     25.0
1      1.0
2      4.0
3     81.0
Name: x, Number of rows: 4, dtype: float
```

7.3.2.59 prev

```
vColumn.prev(self, order_by: list, by: list = [])
```

Replace the element of the column by the previous one.

Parameters

- **order_by:** *<list>*
  How to order the data.
- **by:** *<list>*, optional
  How to group the data.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm["val"].add_copy("prev")
sm["next"].prev(order_by = ["time"], by = ["id"])

#Output
                    time          val     id         prev
0     2014-01-01 11:00:00     0.0290000      0         None
1     2014-01-01 13:45:00     0.2770000      0    0.0290000
2     2014-01-02 10:45:00     0.3210000      0    0.2770000
3     2014-01-02 11:15:00     0.3050000      0    0.3210000
4     2014-01-02 13:45:00     0.3580000      0    0.3050000
...                   ...           ...    ...          ...
Name: smart_meters, Number of rows: 11844, Number of columns: 4
```

7.3.2.60 product / prod

```
vColumn.prod(self)
```

Returns the column product.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["survived"].prod()

#Output
0.0
```

7.3.2.61 quantile

```
vColumn.quantile(self, x: float)
```

Returns the column selected quantile.

Parameters

- **x:** *<float>*
  A float living in [0,1] representing the quantile.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
      x
0     5
1    -1
2     2
3    -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].quantile(x = 0.1)

#Output
```

```
-6.6
```

7.3.2.62 rename

```
vColumn.rename(self, new_name: str)
```

Change the column alias.

Parameters

- **new_name:** *<str>*
  New alias.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
     x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
      x
0     5
1    -1
2     2
3    -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].rename(new_name = "y")

#Output
      y
0     5
1    -1
2     2
3    -9
Name: y, Number of rows: 4, dtype: int
```

7.3.2.63 round

```
vColumn.round(self, n: int)
```

Round the column elements with the input integer.

Parameters

- **n:** *<int>*
  The integer used to round the column elements.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = relation = "((SELECT 5.12132 AS x) UNION ALL (SELECT -1.213 AS x)
   UNION ALL (SELECT 2.12347 AS x) UNION ALL (SELECT -9.965 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
            x
0      5.12132
1     -1.21300
2      2.12347
3     -9.96500
Name: x, Number of rows: 4, dtype: numeric(6,5)

vdf["x"].round(n = 1)

#Output
             x
0       5.10000
1      -1.20000
2       2.10000
3     -10.00000
Name: x, Number of rows: 4, dtype: numeric(6,5)
```

7.3.2.64 sem

```
vColumn.sem(self)
```

Returns the column unbiased standard error of the mean.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].std()

#Output
1.49928585339507
```

7.3.2.65 skewness / skew

```
vColumn.skewness(self)
```

Returns the column unbiased skewness.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["age"].skew()

#Output
0.408876460779437
```

7.3.2.66 slice

```
slice(self, length: int, unit: str = "second", start: bool = True)
```

Slice the time series.

Parameters

- **length:** *<int>*
  Slice size.
- **unit:** *<str>*, optional
  Slice size unit.
- **start:** *<bool>*, optional
  Consider the floor of the slicing or the next one.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_smart_meters
sm = load_smart_meters(cur)
sm["time"]

# Output

                    time
0     2014-01-01 01:15:00
1     2014-01-01 02:30:00
2     2014-01-01 03:00:00
3     2014-01-01 05:00:00
4     2014-01-01 06:00:00
...                   ...
Name: time, Number of rows: 11844, dtype: timestamp

sm["time"].slice(length = "1", unit = "hour")

# Output

                    time
0     2014-01-01 01:00:00
1     2014-01-01 02:00:00
2     2014-01-01 03:00:00
3     2014-01-01 05:00:00
4     2014-01-01 06:00:00
...                   ...
Name: time, Number of rows: 11844, dtype: timestamp
```

7.3.2.67 std

```
vColumn.std(self)
```

Returns the column standard deviation.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].std()
```

```
#Output
52.6460729831293
```

7.3.2.68 str_contains

```
vColumn.str_contains(self, pat: str)
```

Verify if a regular expression is in each of the column elements. The column will be transformed.

Parameters

- **pat:** *<str>*
  The regular expression.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"]

#Output
                                               name
0                Allison, Miss. Helen Loraine
1        Allison, Mr. Hudson Joshua Creighton
2   Allison, Mrs. Hudson J C (Bessie Wald...
3                       Andrews, Mr. Thomas Jr
4                      Artagaveytia, Mr. Ramon
...                                        ...
Name: name, Number of rows: 1234, dtype: varchar(164)

titanic["name"].str_contains(' ([A-Za-z]+)\.')

#Output
     name
0     True
1     True
2     True
3     True
4     True
...    ...
Name: name, Number of rows: 1234, dtype: boolean
```

7.3.2.69 str_count

```
vColumn.str_count(self, pat: str)
```

Compute the number of times a regular expression is in each of the column elements. The column will be transformed.

Parameters

- **pat:** *<str>*
  The regular expression.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"]

#Output
                                        name
0               Allison, Miss. Helen Loraine
1       Allison, Mr. Hudson Joshua Creighton
2  Allison, Mrs. Hudson J C (Bessie Wald...
3                     Andrews, Mr. Thomas Jr
4                    Artagaveytia, Mr. Ramon
...                                      ...
Name: name, Number of rows: 1234, dtype: varchar(164)

titanic["name"].str_count(' ([A-Za-z]+)\.')

#Output
     name
0       1
1       1
2       1
3       1
4       1
...   ...
Name: name, Number of rows: 1234, dtype: int
```

7.3.2.70 str_extract

```
vColumn.str_extract(self, pat: str)
```

Extract the regular expression in each of the column elements. The column will be transformed.

Parameters

- **pat:** *<str>*
  The regular expression.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"]

#Output
                                          name
0               Allison, Miss. Helen Loraine
1       Allison, Mr. Hudson Joshua Creighton
2   Allison, Mrs. Hudson J C (Bessie Wald...
3                     Andrews, Mr. Thomas Jr
4                    Artagaveytia, Mr. Ramon
...                                      ...
Name: name, Number of rows: 1234, dtype: varchar(164)

titanic["name"].str_extract(' ([A-Za-z]+)\.')

#Output
       name
0     Miss.
1       Mr.
2      Mrs.
3       Mr.
4       Mr.
...     ...
Name: name, Number of rows: 1234, dtype: varchar(164)
```

7.3.2.71 str_replace

```
vColumn.str_replace(self, to_replace: str, value: str = "")
```

Replace the regular expression in each of the column elements by another value. The column will be transformed.

Parameters

- **to_replace:** *<str>*
  The regular expression to replace.
- **value:** *<str>*, optional
  The new value.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"]

#Output
                                          name
0               Allison, Miss. Helen Loraine
1        Allison, Mr. Hudson Joshua Creighton
2  Allison, Mrs. Hudson J C (Bessie Wald...
3                     Andrews, Mr. Thomas Jr
4                    Artagaveytia, Mr. Ramon
...                                        ...
Name: name, Number of rows: 1234, dtype: varchar(164)

titanic["name"].str_replace(' ([A-Za-z]+)\.')

#Output
                                          name
0                     Allison, Helen Loraine
1            Allison, Hudson Joshua Creighton
2  Allison, Hudson J C (Bessie Waldo Dan...
3                         Andrews, Thomas Jr
4                        Artagaveytia, Ramon
...                                        ...
Name: name, Number of rows: 1234, dtype: varchar(672)
```

7.3.2.72 str_slice

```
vColumn.str_slice(self, start: int, step: int)
```

Slice the column expression. The column will be transformed.

Parameters

- **start:** *<int>*
  Where to start on the expression.
- **step:** *<int>*
  The step to use (output expression length).

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["name"]

#Output
                                               name
0                 Allison, Miss. Helen Loraine
1        Allison, Mr. Hudson Joshua Creighton
2   Allison, Mrs. Hudson J C (Bessie Wald...
3                        Andrews, Mr. Thomas Jr
4                       Artagaveytia, Mr. Ramon
...                                         ...
Name: name, Number of rows: 1234, dtype: varchar(164)

titanic["name"].str_slice(0, 5)

#Output
     name
0    Alli
1    Alli
2    Alli
3    Andr
4    Arta
...   ...
Name: name, Number of rows: 1234, dtype: varchar(20)
```

7.3.2.73 sub

```
vColumn.sub(self, x: float)
```

Subtract a float from the column elements.

Parameters

- **x:** *<float>*
  A float number.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT 5 AS x) UNION ALL (SELECT -1 AS x) UNION ALL (SELECT 2 AS
    x) UNION ALL (SELECT -9 AS x)) z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")
vdf["x"]

#Output
       x
0      5
1     -1
2      2
3     -9
Name: x, Number of rows: 4, dtype: int

vdf["x"].sub(x = 2)

#Output
        x
0       3
1      -3
2       0
3     -11
Name: x, Number of rows: 4, dtype: int
```

7.3.2.74 sum

```
vColumn.sum(self)
```

Returns the column sum.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].sum()

#Output
```

```
41877.3576
```

7.3.2.75 tail

```
vColumn.tail(self, limit: int = 5, offset: int = 0)
```

Returns a part of the column. The tail is not necessary the end of the object.

Parameters

- **limit:** *<int>*, optional
  The number of elements to return.
- **offset:** *<int>*, optional
  The number of elements to skip.

Returns

The `tablesample` type containing the tail (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].tail(limit = 4, offset = 10)

#Output
          fare
10    35.50000
11    26.55000
12    30.50000
13    50.49580
...        ...
Name: fare, Number of rows: 1234, dtype: numeric(10,5)
```

7.3.2.76 to_enum

```
vColumn.to_enum(self, h: float = 0)
```

Converts the column to categorical.

Returns

The vColumn itself.

Parameters

- **h:** *<float>*, optional
  The interval size to convert used to convert the column. If this parameter is equal to 0, an optimised interval will be computed.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"]

#Output
           fare
0     151.55000
1     151.55000
2     151.55000
3       0.00000
4      49.50420
...         ...
Name: fare, Number of rows: 1234, dtype: numeric(10,5)

titanic["fare"].to_enum()

#Output
           fare
0     [141;188]
1     [141;188]
2     [141;188]
3        [0;47]
4       [47;94]
...         ...
Name: fare, Number of rows: 1234, dtype: varchar
```

7.3.2.77 topk

```
vColumn.topk(self, k: int = -1, dropna: bool = True)
```

Returns the top k most occurrent column elements.

Parameters

- **k:** *<int>*, optional
  Number of elements to consider. If this parameter is equal to -1, all the categories will be returned.
- **dropna:** *<bool>*, optional
  Do not consider null values as a category.

Returns

The `tablesample` type containing the column topk categories (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["pclass"].topk()

#Output
             count              percent
3              663     53.7277147487844
1              312     25.2836304700162
2              259     20.9886547811994
```

7.3.2.78 to_timestamp

```
vColumn.to_timestamp(self)
```

Convert the column to timestamp.

Returns

The vColumn itself.

Example

```
from vertica_ml_python.vdataframe import vdf_from_relation
relation = "((SELECT '2014-01-01' AS x) UNION ALL (SELECT '2014-02-09' AS x))
    z"
vdf = vdf_from_relation(relation, dsn = "VerticaDSN")

#Output
                x
0     2014-01-01
1     2014-02-09
Name: x, Number of rows: 2, dtype: varchar(10)

vdf["x"].to_timestamp()

#Output
                          x
0     2014-01-01 00:00:00
1     2014-02-09 00:00:00
```

```
Name: x, Number of rows: 2, dtype: timestamp
```

7.3.2.79 value_counts

```
vColumn.value_counts(self, k: int = 30)
```

Returns the top k most occurrent column elements and their respective count.

Parameters

- **k:** *<int>*, optional
  Number of elements to consider. If this parameter is equal to -1, all the categories will be returned.

Returns

The `tablesample` type containing the value counts (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["pclass"].value_counts()

#Output
                value
name         "pclass"
dtype             int
unique              3
3                 663
1                 312
2                 259
```

7.3.2.80 var

```
vColumn.var(self)
```

Returns the column variance.

Example

```
from vertica_ml_python.learn.datasets import load_titanic
titanic = load_titanic(cur)
titanic["fare"].var()
```

```
#Output
2771.60900054498
```

8 vertica_ml_python.utilities

Some functions were created to simplify the process. The object tablesample stores in memory a small quantity of data which correspond most of the time to a query result. These utilities functions and objects are very important to understand all the `vertica-ml-python` API.

8.1 Functions

8.1.1 drop_model

```
def drop_model(name: str, cursor)
```

Drop the Vertica model.

Parameters

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  Database Cursor.

8.1.2 drop_table

```
def drop_table(name: str, cursor)
```

Drop the Vertica table.

Parameters

- **name:** *<str>*
  Name of the table.
- **cursor:** *<object>*
  Database Cursor.

8.1.3 drop_text_index

```
def drop_text_index(name: str, cursor)
```

Drop the Vertica text index.

Parameters

- **name:** *<str>*
  Name of the text index.
- **cursor:** *<object>*
  Database Cursor.

8.1.4 drop_view

```
def drop_view(name: str, cursor)
```

Drop the Vertica view.

Parameters

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

8.1.5 load_model

```
def load_model(name: str, cursor, test_relation: str = "")
```

Returns the model object.

Parameters

- **name:** *<str>*
  Model name.
- **cursor:** *<object>*
  Database Cursor.
- **test_relation:** *<str>*, optional
  Name of the relation used to test the model. If it is not defined, the test relation will be the same as the one used to train the model.

8.1.6 pandas_to_vertica

```
pandas_to_vertica(df, cur, name: str)
```

Store a pandas DataFrame in the Vertica DB.

Parameters

- **df:** *<object>*
  Pandas DataFrame.
- **cursor:** *<object>*
  Database Cursor.
- **name:** *<str>*
  Name of the table.

8.1.7 to_tablesample

```
to_tablesample(query: str, cursor, name = "Sample")
```

Returns the tablesample of the query.

Parameters

- **query:** *<str>*
  SQL query.
- **cursor:** *<object>*
  Database Cursor.
- **name:** *<str>*, optional
  Name of the tablesample.

```
from vertica_ml_python.utilities import to_tablesample
to_tablesample('SELECT age, survived FROM titanic LIMIT 5', cur)

#Output
         age    survived
0      2.000           0
1     30.000           0
2     25.000           0
3     39.000           0
4     71.000           0
Name: Sample, Number of rows: 5, Number of columns: 2
```

8.1.8 vertica_cursor

```
vertica_cursor(dsn: str)
```

Returns the Vertica cursor by parsing the DSN (this function will read the ODBCINI file).

Parameters

- **dsn:** *<str>*
  Database DSN.

```
from vertica_ml_python.utilities import vertica_cursor
vertica_cursor("VerticaDSN").execute("SELECT age FROM titanic LIMIT 1").
    fetchall()

#Output
[[Decimal('2.000')]]
```

8.2 tablesample

The tablesample is the transition from 'Big Data' to 'Small Data'. This object was created to have a nice way of displaying the results and to not have any dependency to any other library. It stores the aggregated result in memory and has some useful method to transform it to pandas.Dataframe or vDataframe.

8.2.1 initialization

```
class tablesample(
      values: dict = {}, # Dictionary of the stored columns
      dtype: dict = {}, # Dictionary of the columns types
      name: str = "Sample") # Object Name
```

8.2.2 attributes

The tablesample has two main attributes:

- **values:** The Dictionary of the stored columns.
- **dtype:** The Dictionary of the columns types.

8.2.3 methods

The tablesample has 4 main methods:

- **transpose(self):** Transpose the tablesample.
- **to_pandas(self):** Convert the tablesample to a pandas.Dataframe.
- **to_sql(self):** Generate the sql to build the tablesample to the DB.
- **to_vdf(self, cursor = None, dsn: str = ""):** Convert the tablesample to a vDataframe.

9 vertica_ml_python.learn

This part of the API was built to apply Highly Distributed and Scalable Machine Learning directly on the data. Many algorithms are available and many metrics are very useful to evaluate precisely the efficiency and performance of the created models. Some models are built-in Vertica functions (Highly Distributed and Highly Scalable) and others are based on SQL code generation. In the different sections some of the algorithms methods parameters could be undefined because of the repetitiveness of the process. We will always use the following notations:

- `X` <str>: List of the predictors columns.
- `y` or `y_true` <str>: Response column.
- `y_score` <str>: Prediction.
- `input_relation` <str>: Input relation used to train the model.
- `test_relation` <str>: Relation used to test the model. If it is empty, the train relation is used as test.
- `vdf` <object>: A Virtual Dataframe.

- `method` <str>: The method to compute the score, it must be in {accuracy | auc | prc_auc | best_cutoff | recall | precision | log_loss | negative_predictive_value | specificity | mcc | informedness | markedness | critical_success_index} for classification and in {r2 | mae (mean absolute error) | mse (mean squared error) | msle (mean squared log error) | max (max error) | median (median absolute error) | var (explained variance)} for regression.
- `cutoff` <float>: Prediction threshold in the case of binary classification. It must be in]0,1[
- `pos_label` <str or int>: Main class in the prediction (the positive class), the other classes will be seen as negative.
- `labels` <list>: Labels to consider during the computation.
- `tree_id` <int>: The tree ID in case of multi-tree models.

All the objects will have all its parameters stored as attribute. For example, it is possible to change the test relation anytime by changing the `test_relation` attribute.

9.1 vertica_ml_python.learn.cluster

9.1.1 DBSCAN

Create a DBSCAN object by using the DBSCAN algorithm as defined by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu. This object is using pure SQL to compute all the distances and neighbors. It is also using Python to compute the cluster propagation (non scalable phase). This model is using CROSS JOIN and may be really expensive in some cases. It will index all the elements of the table in order to be optimal (the CROSS JOIN will happen only with IDs which are integers). As DBSCAN is using the p-distance, it is highly sensible to un-normalized data. However, DBSCAN is really robust to outliers and can find non-linear clusters. It is a very powerful algorithm for outliers detection and clustering.

initialization

```
class DBSCAN(
  name: str,
  cursor,
  eps: float = 0.5,
  min_samples: int = 5,
  p: int = 2)
```

Parameters

- **name:** *<str>*
  Name of the relation created after fitting the model.
- **cursor:** *<object>*
  DB cursor.
- **eps:** *<float>*, optional
  The radius of a neighborhood with respect to some point.
- **min_samples:** *<int>*, optional
  Minimum number of points required to form a dense region.
- **p:** *<int>*, optional
  The p corresponding to the one of the p-distance (distance metric used during the model computation).

Methods

The DBSCAN object has four main methods:

```
# Fit the model with the input columns
def fit(self, input_relation: str, X: list, key_columns: list = [], index = ""
    )

# Print the model information
def info(self)

# Plot the model (only possible if the number of columns <= 3)
def plot(self)

# Create a vDataframe using the final relation
def to_vdf(self)
```

The index parameter (name of the primary key column in the relation) of the 'fit' method is very important as without it an indexed copy of the main relation will be created (it could be really expensive). It is highly recommended to have a primary key column in the main table to avoid unnecessary computations.

Attributes

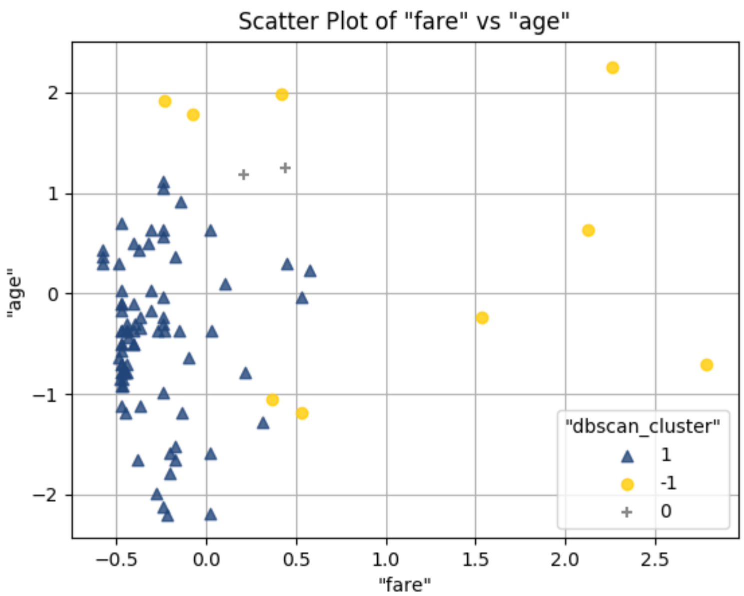
The DBSCAN object has two main attributes:

```
self.n_cluster # Number of clusters
self.n_noise # Number of elements considered as noise by the algorithm
```

Example

```
from vertica_ml_python import vDataframe
from vertica_ml_python.learn.cluster import DBSCAN

# Building a normalized relation of the features, we will use
titanic = vDataframe("titanic", cur)
# We will use a sample of the data in order to have a beautiful plot
titanic.main_relation += " TABLESAMPLE(10)"
titanic = titanic.select(["fare", "age"])
titanic.normalize()
titanic.to_db("titanic_normalize")

# We can build the model
model = DBSCAN("dbscan_titanic", cur)
model.fit("titanic_normalize", ["fare", "age"])

# We can see the different information relative to the model
model.info()

# Output
DBSCAN was successfully achieved by building 3 cluster(s) and by identifying
    21 elements as noise.
```

```
If you are not happy with the result, do not forget to normalize the data
    before applying DBSCAN. As this algorithm is using the p-distance, it is
    really sensible to the data distribution.

# And Plot the model
model.plot()
```



9.1.2 KMeans

Create a KMEANS object by using the Vertica Highly Distributed and Scalable KMEANS directly on the data.

initialization

```
class KMEANS(
  name: str,
  cursor,
  n_cluster: int = 8,
  init = "kmeanspp",
  max_iter: int = 300,
  tol: float = 1e-4)
```

Parameters

- **name:** *<str>*
  Name of the the model. The model is stored in the DB.
- **cursor:** *<object>*
  DB cursor.
- **n_cluster:** *<int>*, optional
  Number of clusters.

- **init:** *<str or list>*, optional
  The method used to find the initial cluster centers. The method can be in {random | kmeanspp}. It can be also a list with the initial cluster centers to use.
- **max_iter:** *<int>*, optional
  The maximum number of iterations the algorithm performs.
- **tol:** *<float>*, optional
  Determines whether the algorithm has converged. The algorithm is considered converged after no center has moved more than a distance of 'tol' from the previous iteration.

Methods

The KMeans object has four main methods:

```
# Add the cluster prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "")

# Drop the model
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list)

# Plot the model (only possible if the number of columns <= 3)
def plot(self)
```

Attributes

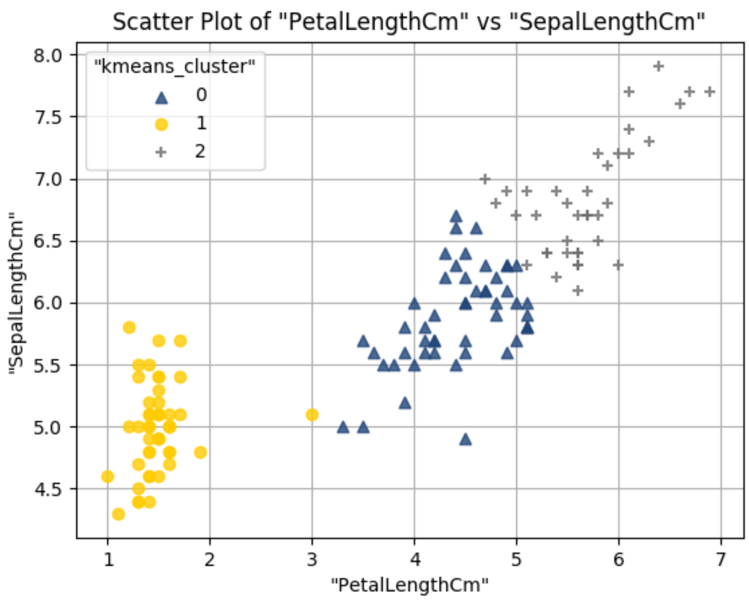
The KMEANS object has two main attributes:

```
self.cluster_centers # Position of the cluster centers
self.metrics # Different metrics to evaluate the model
```

Example

```
from vertica_ml_python.learn.cluster import KMeans

# We can build the model
model = KMeans("kmeans_iris", cur, n_cluster = 3)
model.fit("iris", ["PetalLengthCm", "SepalLengthCm"])

# We can see the different metrics relative to the model
model.metrics

# Output
                                                          value
Between-Cluster Sum of Squares                        512.23072
Total Sum of Squares                                  566.03207
Total Within-Cluster Sum of Squares                   53.801351
Between-Cluster SS / Total SS                0.9049499969144859
converged                                                  True
```

```
# And Plot the model
model.plot()
```



9.2 vertica_ml_python.learn.datasets

Many datasets are available to learn some Data Science basics or to test the API. These functions will store the data in the DB using the specific schema and table name.

9.2.1 load_iris

```
load_iris(cursor, schema: str = 'public', name = 'iris')
```

Load and store the Iris Dataset (Classification, Regression and Clustering).

9.2.2 load_smart_meters

```
load_smart_meters(cursor, schema: str = 'public', name = 'iris')
```

Load and store the Iris Dataset Smart Meter Dataset (Time series and Clustering).

9.2.3 load_titanic

```
load_titanic(cursor, schema: str = 'public', name = 'iris')
```

Load and store the Titanic Dataset (Classification and Regression).

9.2.4 load_winequality

```
load_winequality(cursor, schema: str = 'public', name = 'iris')
```

Load and store the Wine Quality Dataset (Classification and Regression).

9.3 vertica_ml_python.learn.decomposition

9.3.1 PCA

Create a PCA object by using the Vertica Highly Distributed and Scalable PCA on the data.

initialization

```
class PCA(
    name: str,
    cursor,
    n_components: int = 0,
    scale: bool = False,
    method: str = "Lapack")
```

Parameters

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  DB cursor.
- **n_components:** *<int>*, optional
  The number of components to keep in the model. If this value is not provided, all components are kept. The maximum number of components is the number of non-zero singular values returned by the internal call to SVD. This number is less than or equal to SVD (number of columns, number of rows).
- **scale:** *<bool>*, optional
  A Boolean value that specifies whether to standardize the columns during the preparation step: If true use a correlation matrix instead of a covariance matrix.
- **method:** *<str>*, optional
  The method used to calculate PCA, can be set to LAPACK.

Methods

The PCA object has 3 methods:

```
# Drop the model from the DB
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list)

# Build a vdf from the output relation
def to_vdf(self, n_components: int = 0,  cutoff: float = 1, key_columns: list
    = [], inverse: bool = False)
```

Attributes

The PCA object has only 3 attributes:

```
self.components # The principal components
self.explained_variance # The information about singular values found.
self.mean # The information about columns from the input relation used for
    creating the PCA model
```

Example

```
from vertica_ml_python.learn.decomposition import PCA

# We can build the model
model = PCA("pca_iris", cur)
model.fit("iris", ["SepalLengthCm", "SepalWidthCm", "PetalWidthCm", "
    PetalLengthCm"])

# We can evaluate the model
model.explained_variance

# Output
                       value    explained_variance   \\
1           2.05544174529956     0.924616207174268   \\
2          0.492182457659266    0.0530155678505349   \\
3          0.280221177097938    0.0171851395250067   \\
4          0.153892907978245    0.0051830854501896   \\
           accumulated_explained_variance
1                       0.924616207174268
2                       0.977631775024804
3                        0.99481691454981

# We can export the model
model.to_vdf(n_components = 2)

# Output
                  col1                    col2
0    -2.68420712510395      0.326607314764391
1    -2.71539061563413     -0.169556847556024
2    -2.88981953961792     -0.137345609605025
3    -2.74643719730873     -0.311124315751989
4    -2.72859298183131      0.333924563568457
...                ...                    ...
Name: pca_table_iris, Number of rows: 150, Number of columns: 2
```

9.3.2 SVD

Create a SVD object by using the Vertica Highly Distributed and Scalable SVD on the data.

initialization

```
class SVD(
    name: str,
    cursor,
    n_components: int = 0,
    method: str = "Lapack")
```

Parameters

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  DB cursor.
- **n_components:** *<int>*, optional
  The number of components to keep in the model. If this value is not provided, all components are kept. The maximum number of components is the number of non-zero singular values returned by the internal call to SVD. This number is less than or equal to SVD (number of columns, number of rows).
- **method:** *<str>*, optional
  The method used to calculate SVD, can be set to LAPACK.

Methods

The SVD object has 3 methods:

```
# Drop the model from the DB
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list)

# Build a vdf from the output relation
def to_vdf(self, n_components: int = 0,  cutoff: float = 1, key_columns: list
    = [], inverse: bool = False)
```

Attributes

The SVD object has only 2 attributes:

```
self.singular_values # The right singular vectors.
self.explained_variance # The information about singular values found.
```

Example

```
from vertica_ml_python.learn.decomposition import SVD

# We can build the model
```

```
model = SVD("svd_iris", cur)
model.fit("iris", ["SepalLengthCm", "SepalWidthCm", "PetalWidthCm", "
    PetalLengthCm"])

# We can evaluate the model
model.explained_variance

# Output
                      value       explained_variance    \\
1           95.9506675123581        0.965429688562226   \\
2           17.7229532787505        0.0329379703572465  \\
3           3.46929666441398      0.00126213998717665   \\
4           1.87891236262158     0.000370201093350833   \\
            accumulated_explained_variance
1                        0.965429688562226
2                        0.998367658919472
3                        0.999629798906649
4                                      1.0

# We can export the model
model.to_vdf(n_components = 2)

# Output
                    col1                   col2
0     0.0616171171531346     -0.129969428300603
1     0.0580722976924328     -0.111371451741922
2     0.0567633851673065     -0.118294769303341
3     0.0566543139689709     -0.105607729117619
4     0.0612300644170471      -0.13143114177406
...                  ...                    ...
Name: svd_table_iris, Number of rows: 150, Number of columns: 2
```

9.4 vertica_ml_python.learn.ensemble

9.4.1 RandomForestClassifier

Create a RandomForestClassifier object by using the Vertica Highly Distributed and Scalable Random Forest on the data.

initialization

```
class RandomForestClassifier(
    name: str,
    cursor,
    n_estimators: int = 10,
    max_features = "auto",
    max_leaf_nodes: int = 1e9,
    sample: float = 0.632,
```

```
    max_depth: int = 5,
    min_samples_leaf: int = 1,
    min_info_gain: float = 0.0,
    nbins: int = 32)
```

Parameters

- **name:** *<str>*
  Name of the the model. The model is stored in the DB.
- **cursor:** *<object>*
  DB cursor.
- **n_estimators:** *<int>*, optional
  The number of trees in the forest, an integer between 0 and 1000, inclusive.
- **max_features:** *<int>*, optional
  The number of randomly chosen features from which to pick the best feature to split on a given tree node. If this parameter is equal to "auto", it will be equal to the square root of the total number of predictors.
- **max_leaf_nodes:** *<int>*, optional
  The maximum number of leaf nodes a tree in the forest can have, an integer between 1 and 1e9, inclusive.
- **sample:** *<float>*, optional
  The portion of the input data set that is randomly picked for training each tree, a FLOAT between 0.0 and 1.0, inclusive.
- **max_depth:** *<int>*, optional
  The maximum depth for growing each tree, an integer between 1 and 100, inclusive.
- **min_samples_leaf:** *<int>*, optional
  The minimum number of samples each branch must have after splitting a node, an integer between 1 and 1e6, inclusive. A split that causes fewer remaining samples is discarded.
- **min_info_gain:** *<float>*, optional
  The minimum threshold for including a split, a FLOAT between 0.0 and 1.0, inclusive. A split with information gain less than this threshold is discarded.
- **nbins:** *<int>*, optional
  The number of bins to use for continuous features, an integer between 2 and 1000, inclusive.

Methods

The RandomForestClassifier object has many methods:

```
# Add the RF prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "", cutoff: float = 0.5)

# Compute different metrics to evaluate the model
def classification_report(self, cutoff: float = 0.5, labels = [])

# Draw the confusion matrix of the model
def confusion_matrix(self, pos_label = None, cutoff: float = 0.5)

# Save a table or a view in the DB corresponding to the model predictions for
   all the classes
```

```
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1, all_classes:
    bool = False)

# Drop the model from the DB
def drop(self)

# Export the selected tree to the graphviz format
def export_graphviz(self, tree_id: int = 0)

# Compute and plot the feature importance
def features_importance(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Returns a tablesample of the selected tree
def get_tree(self, tree_id: int = 0)

# Draw the Lift Chart
def lift_chart(self, pos_label = None)

# Plot the selected tree
def plot_tree(self, tree_id: int = 0, pic_path: str = "")

# Draw the PRC Curve
def prc_curve(self, pos_label = None)

# Draw the ROC Curve
def roc_curve(self, pos_label = None)

# Compute the selected metric
def score(self, pos_label = None, cutoff: float = 0.5, method: str = "accuracy
    ")
```

Attributes

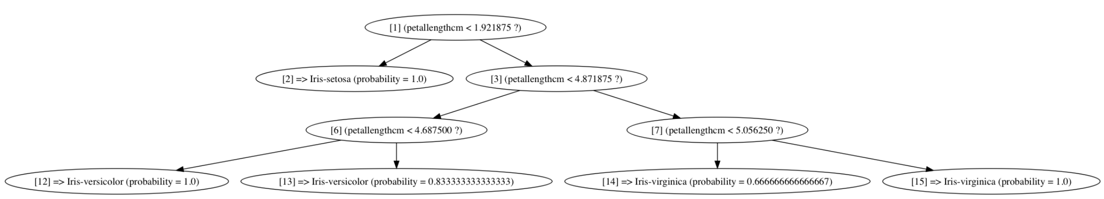
The RandomForestClassifier object has only one attribute:

```
self.classes # The response column classes
```

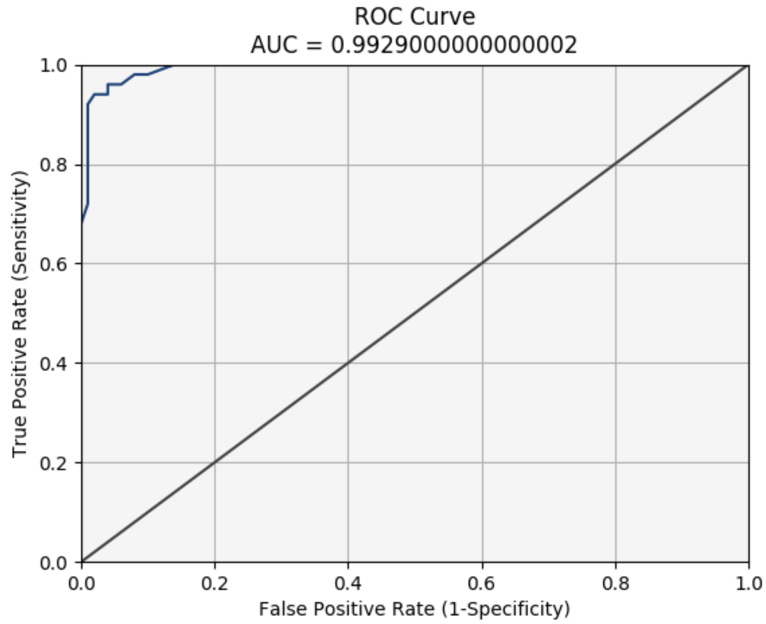
Example

```
from vertica_ml_python.learn.ensemble import RandomForestClassifier

# We can build the model
model = RandomForestClassifier("rf_iris", cur, max_depth = 3, n_estimators =
    20)
model.fit("iris", ["PetalLengthCm", "SepalLengthCm", "SepalWidthCm"], "Species
    ")
```

```
# We can look at the model confusion matrix
model.confusion_matrix()

# Output
                          Iris-setosa    Iris-versicolor    Iris-virginica
Iris-setosa                        50                  0                 0
Iris-versicolor                     0                 46                 4
Iris-virginica                      0                  3                47

# We can also plot the first tree
model.plot_tree(pic_path = "tree0.png")

# Output
[1] (petallengthcm < 1.921875 ?)
|-- [2] => Iris-setosa (probability = 1.0)
|-- [3] (petallengthcm < 4.871875 ?)
    |-- [6] (petallengthcm < 4.687500 ?)
    |    |-- [12] => Iris-versicolor (probability = 1.0)
    |    |-- [13] => Iris-versicolor (probability = 0.833333333333333)
    |-- [7] (petallengthcm < 5.056250 ?)
         |-- [14] => Iris-virginica (probability = 0.666666666666667)
         |-- [15] => Iris-virginica (probability = 1.0)
```




```
# We can also look at the Iris-virginica ROC curve
model.roc_curve(pos_label = 'Iris-virginica')
```



9.4.2 RandomForestRegressor

Create a RandomForestRegressor object by using the Vertica Highly Distributed and Scalable Random Forest on the data.

initialization

```
class RandomForestRegressor(
    name: str,
    cursor,
    n_estimators: int = 10,
    max_features = "auto",
    max_leaf_nodes: int = 1e9,
    sample: float = 0.632,
    max_depth: int = 5,
    min_samples_leaf: int = 1,
    min_info_gain: float = 0.0,
    nbins: int = 32)
```

Parameters

- **name:** *<str>*
  Name of the the model. The model is stored in the DB.
- **cursor:** *<object>*
  DB cursor.
- **n_estimators:** *<int>*, optional
  The number of trees in the forest, an integer between 0 and 1000, inclusive.
- **max_features:** *<int>*, optional
  The number of randomly chosen features from which to pick the best feature to split on a given tree node. If this parameter is equal to "auto", it will be equal to the square root of the total number of predictors

- **max_leaf_nodes:** *<int>*, optional
  The maximum number of leaf nodes a tree in the forest can have, an integer between 1 and 1e9, inclusive.
- **sample:** *<float>*, optional
  The portion of the input data set that is randomly picked for training each tree, a FLOAT between 0.0 and 1.0, inclusive.
- **max_depth:** *<int>*, optional
  The maximum depth for growing each tree, an integer between 1 and 100, inclusive.
- **min_samples_leaf:** *<int>*, optional
  The minimum number of samples each branch must have after splitting a node, an integer between 1 and 1e6, inclusive. A split that causes fewer remaining samples is discarded.
- **min_info_gain:** *<float>*, optional
  The minimum threshold for including a split, a FLOAT between 0.0 and 1.0, inclusive. A split with information gain less than this threshold is discarded.
- **nbins:** *<int>*, optional
  The number of bins to use for continuous features, an integer between 2 and 1000, inclusive.

Methods

The RandomForestRegressor object has many methods:

```
# Add the RF prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "")

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True)

# Drop the model from the DB
def drop(self)

# Export the selected tree to the graphviz format
def export_graphviz(self, tree_id: int = 0)

# Compute and plot the feature importance
def features_importance(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Returns a tablesample of the selected tree
def get_tree(self, tree_id: int = 0)

# Plot the selected tree
def plot_tree(self, tree_id: int = 0, pic_path: str = "")

# Evaluate the test relation by computing many different metrics
def regression_report(self)

```

```
# Compute the selected metric
def score(self, method: str = "r2")
```

Example

```
from vertica_ml_python.learn.ensemble import RandomForestRegressor

# We can build the model
model = RandomForestRegressor("rf_iris", cur, max_depth = 3, n_estimators =
    20)
model.fit("iris", ["PetalLengthCm", "SepalWidthCm", "Species"], "SepalLengthCm
    ")

# We can evaluate the model
model.regression_report()

# Output
explained_variance          0.855205647780937
max_error                   0.995783115048306
median_absolute_error       0.207286172097297
mean_absolute_error          0.24878157077602
mean_squared_error         0.0988507117473723
r2                           0.85487081683392

# We can also get the graphviz format of the trees
model.export_graphviz()

# Output
digraph Tree{
1 [label = "petallengthcm < 4.134375 ?", color="blue"];
1 -> 2 [label = "yes", color = "black"];
1 -> 3 [label = "no", color = "black"];
2 [label = "petallengthcm < 3.396875 ?", color="blue"];
2 -> 4 [label = "yes", color = "black"];
2 -> 5 [label = "no", color = "black"];
4 [label = "petallengthcm < 1.368750 ?", color="blue"];
4 -> 8 [label = "yes", color = "black"];
4 -> 9 [label = "no", color = "black"];
8 [label = "prediction: 4.600000, variance: 0", color="red"];
9 [label = "prediction: 5.014706, variance: 0.0994896", color="red"];
5 [label = "sepalwidthcm < 2.225000 ?", color="blue"];
5 -> 10 [label = "yes", color = "black"];
5 -> 11 [label = "no", color = "black"];
10 [label = "prediction: 6.000000, variance: 0", color="red"];
11 [label = "prediction: 5.622222, variance: 0.0150617", color="red"];
3 [label = "sepalwidthcm < 3.425000 ?", color="blue"];
3 -> 6 [label = "yes", color = "black"];
3 -> 7 [label = "no", color = "black"];
```

```
6 [label = "petallengthcm < 5.609375 ?", color="blue"];
6 -> 12 [label = "yes", color = "black"];
6 -> 13 [label = "no", color = "black"];
12 [label = "prediction: 6.164706, variance: 0.225813", color="red"];
13 [label = "prediction: 7.175000, variance: 0.189375", color="red"];
7 [label = "sepalwidthcm < 3.650000 ?", color="blue"];
7 -> 14 [label = "yes", color = "black"];
7 -> 15 [label = "no", color = "black"];
14 [label = "prediction: 7.200000, variance: 0", color="red"];
15 [label = "prediction: 7.900000, variance: 7.10543e-15", color="red"];
}
```

9.5 vertica_ml_python.learn.linear_model

9.5.1 LinearRegression

Create a ElasticNet object by using the Vertica Highly Distributed and Scalable Linear Regression on the data. The Linear Regression is the ElasticNet model without regularization. The other parameters stay the same.

initialization

```
def LinearRegression(
      name: str,
        cursor,
        tol: float = 1e-4,
        C: float = 1.0,
        max_iter: int = 100,
        solver: str = 'Newton')
```

9.5.2 ElasticNet

Create a ElasticNet object by using the Vertica Highly Distributed and Scalable Linear Regression on the data.

initialization

```
class ElasticNet(
    name: str,
    cursor,
    penalty: str = 'ENet',
    tol: float = 1e-4,
    C: float = 1.0,
    max_iter: int = 100,
    solver: str = 'CGD',
    l1_ratio: float = 0.5)
```

Parameters

- **name:** *<str>*
  Name of the model.

- **cursor:** *<object>*
  DB cursor.
- **penalty:** *<str>*, optional
  Determines the method of regularization: {None | L1 | L2 | ENet}
- **tol:** *<float>*, optional
  Determines whether the algorithm has reached the specified accuracy result.
- **C:** *<float>*, optional
  The regularization parameter value. The value must be zero or non-negative.
- **max_iter:** *<int>*, optional
  Determines the maximum number of iterations the algorithm performs before achieving the specified accuracy result.
- **solver:** *<int>*, optional
  The optimizer method used to train the model: {Newton | BFGS | CGD}
- **l1_ratio:** *<float>*, optional
  ENet mixture parameter that defines how much L1 versus L2 regularization to provide.

Methods

The ElasticNet object has many methods:

```
# Add the ElasticNet prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "")

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1)

# Drop the model from the DB
def drop(self)

# Compute the importance of each feature
def features_importance(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Plot the SVM if it is possible (The length of X must be lesser of equal to
    3)
def plot(self)

# Compute different metrics to evaluate the model
def regression_report(self)

# Compute the selected metric
def score(self, method: str = "r2")
```

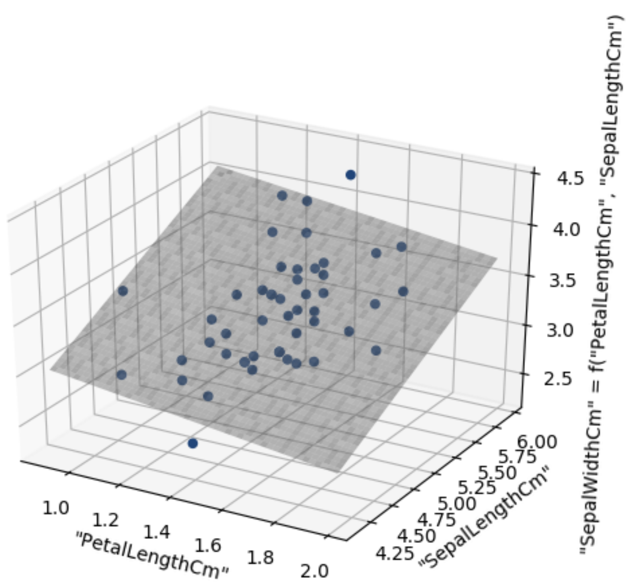
Attributes

The ElasticNet object has only one attribute:

```
self.coef # Informations about the model coefficients
```

Example

```
from vertica_ml_python.learn.linear_model import ElasticNet

# We can build the model
model = ElasticNet("enet_iris", cur, penalty = "None", tol = 1e-8)
model.fit("iris", ["PetalLengthCm", "SepalLengthCm"], "SepalWidthCm")

# We can evaluate the model
model.regression_report()

# Output
                                        value
explained_variance         0.452452439026576
max_error                  0.861565208032992
median_absolute_error      0.203743266177453
mean_absolute_error        0.251888470089032
mean_squared_error         0.102254872043582
r2                         0.452452439026074

# We can also draw the model
model.plot()
```



9.5.3 Lasso

Create a ElasticNet object by using the Vertica Highly Distributed and Scalable Linear Regression on the data. The Lasso Regression is the ElasticNet model with the L1 regularization. The other parameters stay the same.

initialization

```
def Lasso(
      name: str,
      cursor,
      tol: float = 1e-4,
      max_iter: int = 100,
      solver: str = 'CGD')
```

9.5.4 LogisticRegression

Create a LogisticRegression object by using the Vertica Highly Distributed and Scalable Logistic Regression on the data.

initialization

```
class LogisticRegression(
    name: str,
    cursor,
    penalty: str = 'L2',
    tol: float = 1e-4,
    C: int = 1,
    max_iter: int = 100,
    solver: str = 'CGD',
    l1_ratio: float = 0.5)
```

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  DB cursor.
- **penalty:** *<str>*, optional
  Determines the method of regularization: {None | L1 | L2 | ENet}
- **tol:** *<float>*, optional
  Determines whether the algorithm has reached the specified accuracy result.
- **C:** *<int>*, optional
  The regularization parameter value. The value must be zero or non-negative.
- **max_iter:** *<int>*, optional
  Determines the maximum number of iterations the algorithm performs before achieving the specified accuracy result.
- **solver:** *<int>*, optional
  The optimizer method used to train the model: {Newton | BFGS | CGD}
- **l1_ratio:** *<float>*, optional
  ENet mixture parameter that defines how much L1 versus L2 regularization to provide.

Methods

The LogisticRegression object has many methods:

```
# Add the LogisticRegression prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "", cutoff: float = 0.5)

# Compute different metrics to evaluate the model
def classification_report(self, cutoff: float = 0.5)

# Draw the confusion matrix of the model
def confusion_matrix(self, cutoff: float = 0.5)

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1)

# Drop the model from the DB
def drop(self)

# Compute the importance of each feature
def features_importance(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Draw the Lift Chart
def lift_chart(self)

# Plot the LogisticRegression if it is possible (The length of X must be
    lesser of equal to 2)
def plot(self)

# Draw the PRC Curve
def prc_curve(self)

# Draw the ROC Curve
def roc_curve(self)

# Compute the selected metric
def score(self, cutoff: float = 0.5, method: str = "accuracy")
```

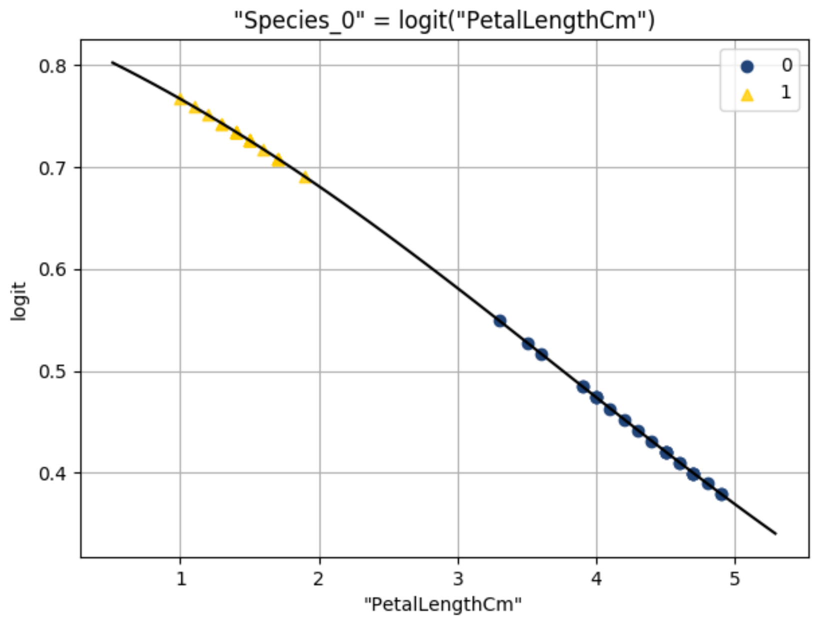
Attributes

The LogisticRegression object has only one attribute:

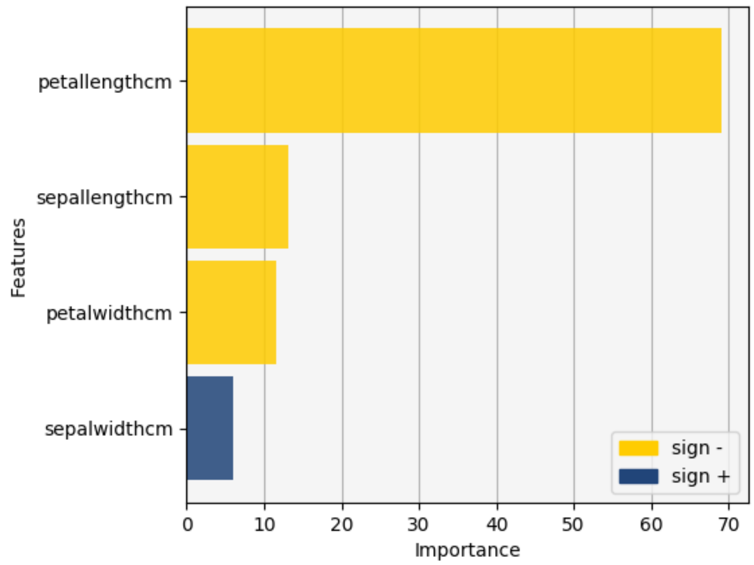
```
self.coef # Informations about the model coefficients
```

Example

```
from vertica_ml_python import vDataframe
from vertica_ml_python.learn.linear_model import LogisticRegression

# We create dummies
iris = vDataframe("iris", cur)
iris["Species"].get_dummies(use_numbers_as_suffix = True)
iris.to_db("iris_dummy")

# We can build the model
model = LogisticRegression("logit_iris", cur)
model.fit("iris_dummy", ["PetalLengthCm"], "Species_0")

# We can evaluate the model
model.classification_report()

# Output
                              value
auc                             1.0
prc_auc          0.9800000000000001
accuracy          0.953333333333333
log_loss          0.187846851053108
precision                       1.0
recall           0.8771929824561403
f1-score         0.9345794392523363
mcc              0.9032106474595007
informedness     0.8771929824561404
markedness       0.9300000000000002
csi              0.8771929824561403

# We can also draw the model
model.plot()
```




```
# We can build a new model
model.drop()
model.fit("iris_dummy", ["PetalLengthCm", "PetalWidthCm",  "SepalLengthCm", "
    SepalWidthCm"], "Species_0")

# We can see the features importance
model.features_importance()
```



9.5.5 Ridge

Create a ElasticNet object by using the Vertica Highly Distributed and Scalable Linear Regression on the data. The Lasso Regression is the ElasticNet model with the L2 regularization. The other parameters stay the same.

initialization

```
def Ridge(
      name: str,
      cursor,
      tol: float = 1e-4,
      max_iter: int = 100,
      solver: str = 'Newton')
```

9.6 vertica_ml_python.learn.metrics

Many metrics are available to evaluate a model efficiency. These functions can be used independently of the models. For more flexibility, the input columns can also be SQL expressions.

9.6.1 Regression

9.6.1.1 explained_variance

```
def explained_variance(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the explain variance of 2 input columns.

9.6.1.2 max_error

```
def max_error(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the max error of 2 input columns.

9.6.1.3 median_absolute_error

```
def median_absolute_error(
    y_true: str,
    y_score: str,
    input_relation: str,
```

```
    cursor)
```

Compute the median absolute error of 2 input columns.

9.6.1.4 mean_absolute_error

```
def mean_absolute_error(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the mean absolute error of 2 input columns.

9.6.1.5 mean_squared_error

```
def mean_squared_error(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the mean squared error of 2 input columns.

9.6.1.6 mean_squared_log_error

```
def mean_squared_log_error(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the mean squared log error of 2 input columns.

9.6.1.7 regression_report

```
def regression_report(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute different regression metrics of 2 input columns.

9.6.1.8 r2_score

```
def r2_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the r2 score of 2 input columns.

9.6.2 Classification

9.6.2.1 accuracy_score

```
def accuracy_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor)
```

Compute the accuracy of 2 input columns.

9.6.2.2 auc

```
auc(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the AUC (Area Under the ROC) of 2 input columns.

9.6.2.3 classification_report

```
classification_report(
    y_true: str = "",
    y_score: str = "",
    input_relation: str = "",
    cursor = None,
    labels: list = [],
    cutoff: float = 0.5)
```

Compute different classification metrics of 2 input columns.

9.6.2.4 confusion_matrix

```
confusion_matrix(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Confusion Matrix of 2 input columns.

9.6.2.5 critical_success_index

```
critical_success_index(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Critical Success Index of 2 input columns.

9.6.2.6 f1_score

```
f1_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the F1 Score of 2 input columns.

9.6.2.7 informedness

```
informedness(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Informedness of 2 input columns.

9.6.2.8 log_loss

```
log_loss(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Log Loss of 2 input columns.

9.6.2.9 markedness

```
log_loss(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Markedness of 2 input columns.

9.6.2.10 matthews_corrcoef

```
matthews_corrcoef(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Matthews Correlation Coefficient of 2 input columns.

9.6.2.11 multilabel_confusion_matrix

```
multilabel_confusion_matrix(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    labels: list)
```

Compute the Confusion Matrix of 2 input columns in case of Multi Classification.

9.6.2.12 negative_predictive_score

```
negative_predictive_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Negative Predictive Score of 2 input columns.

9.6.2.13 prc_auc

```
prc_auc(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the PRC AUC of 2 input columns.

9.6.2.14 precision_score

```
precision_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Precision Score of 2 input columns.

9.6.2.15 recall_score

```
recall_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Recall Score of 2 input columns.

9.6.2.16 specificity_score

```
specificity_score(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1)
```

Compute the Specificity Score of 2 input columns.

9.7 vertica_ml_python.learn.model_selection

9.7.1 best_k

```
def best_k(
      X: list,
      input_relation: str,
      cursor,
      n_cluster = (1, 100),
      init = "kmeanspp",
      max_iter: int = 50,
      tol: float = 1e-4,
      elbow_score_stop = 0.8)
```

Find the KMeans K by using a score threshold.

Parameters

- **X:** *<list>*
  List of the predictor columns.
- **input_relation:** *<str>*
  Relation used to train the model.
- **cursor:** *<object>*
  DB cursor.
- **n_cluster:** *<tuple>*, optional
  Tuple representing the number of cluster to start with and to end with.
- **init:** *<str or list>*, optional
  The method used to find the initial cluster centers. The method can be in {random | kmeanspp}.
  It can be also a list with the initial cluster centers to use.
- **max_iter:** *<int>*, optional
  The maximum number of iterations the algorithm performs.
- **tol:** *<float>*, optional
  Determines whether the algorithm has converged. The algorithm is considered converged after no center has moved more than a distance of 'tol' from the previous iteration.

- **elbow_score_stop:** *<float>*, optional
  Stop the Parameters Search when this Elbow score is reached.

Example

```
from vertica_ml_python.learn.model_selection import best_k

best_k(["PetalLengthCm", "PetalWidthCm", "SepalLengthCm", "SepalWidthCm"], "
    iris", cur)

# Output
3
```

9.7.2 cross_validate

Compute the K-Fold cross validation of an estimator.

initialization

```
def cross_validate(
    estimator,
    input_relation: str,
    X: list,
    y: str,
    cv: int = 3,
    pos_label = None,
    cutoff: float = 0.5)
```

Parameters

- **estimator:** *<object>*
  Estimator having a fit method and a DB cursor.
- **input_relation:** *<str>*
  The relation used to test the estimator.
- **X:** *<list>*
  List of the predictor columns.
- **y:** *<str>*
  Response Column.
- **cv:** *<int>*, optional
  Number of folds.
- **pos_label:** *<anytype>*, optional
  The main class in case of classification.
- **cutoff:** *<float>*, optional
  The cutoff in case of classification. It must be in]0,1[

Returns

The `tablesample` type containing different metric to evaluate the estimator (the information will be stored in the `values` attribute). You can convert this object to pandas using the `to_pandas` method or to vDataframe using the `to_vdf` method.

Example

```
from vertica_ml_python.learn.linear_model import LogisticRegression
from vertica_ml_python.learn.model_selection import cross_validate

# We test the estimator
model = Logistic_Regression("logit_titanic", cur)
cross_validate(model, "titanic", ["age", "pclass", "fare"], "survived")

# Output
                           auc                 prc_auc   \\
1-fold      0.7269061583577717      0.636432596278644   \\
2-fold      0.6903612550943709      0.580010812173139   \\
3-fold      0.6922692837465568     0.5612045504938292   \\
avg             0.703178899066         0.592549319649   \\
std            0.0167957786056        0.0319658708149   \\
                     accuracy               log_loss   \\
1-fold      0.704663212435233     0.277918039898239   \\
2-fold      0.708433734939759     0.280386273274253   \\
3-fold      0.690531177829099     0.282475484333873   \\
avg            0.701209375068        0.280259932502   \\
std          0.00770593417198      0.00186271243782   \\
                     precision                  recall   \\
1-fold       0.5984848484848485      0.6528925619834711   \\
2-fold      0.47244094488188976                   0.625   \\
3-fold      0.49242424242424243      0.5909090909090909   \\
avg              0.521116678597          0.622933884298   \\
std             0.0553124960506         0.0253467854264   \\
                      f1-score                     mcc   \\
1-fold      0.6897262557977386     0.37822641657012745   \\
2-fold      0.6642216788916055     0.31190453116529854   \\
3-fold      0.6503716409376787      0.3006795875598417   \\
avg             0.668106525209          0.330270178432   \\
std            0.0162996002176         0.0342184202053   \\
                   informedness              markedness   \\
1-fold      0.38385702898854723      0.37267839687194515   \\
2-fold      0.33369565217391317      0.29153642226882437   \\
3-fold      0.31404958677685957      0.28787878787878785   \\
avg              0.343867422646           0.317364535673   \\
std             0.0293923848748          0.0391412996111   \\
                            csi
1-fold       0.4540229885057471
2-fold      0.36809815950920244
```

```
3-fold        0.3672316384180791
avg              0.396450928811
std             0.0407111308106
```

9.7.3 train_test_split

```
train_test_split(input_relation: str, cursor, test_size: float = 0.33)
```

Build one table and 2 views which can be used to evaluate a model. The table will include all the main relation information with a test column (boolean) which represents if the data belong to the test or train set.

Parameters

- **input_relation:** *<str>*
  The relation used to test the estimator.
- **cursor:** *<object>*
  A DB cursor.
- **test_size:** *<float>*
  Proportion of the test set comparint to the training set.

Returns

A tuple (name of the train view, name of the test view)

9.8 vertica_ml_python.learn.naive_bayes

9.8.1 MultinomialNB

Create a MultinomialNB object by using the Vertica Highly Distributed and Scalable Naive Bayes on the data.

initialization

```
class MultinomialNB(name: str, cursor, alpha: float = 1.0)
```

Parameters

- **name:** *<str>*
  Name of the the model. The model is stored in the DB.
- **cursor:** *<object>*
  DB cursor.
- **alpha:** *<float>*, optional
  A float that specifies use of Laplace smoothing if the event model is categorical, multinomial, or Bernoulli.

Methods

The MultinomialNB object has many methods:

```
# Add the MultinomialNB prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "", cutoff: float = 0.5)

# Compute different metrics to evaluate the model
def classification_report(self, cutoff: float = 0.5, labels = [])

# Draw the confusion matrix of the model
def confusion_matrix(self, pos_label = None, cutoff: float = 0.5)

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1, all_classes:
     bool = False)

# Drop the model from the DB
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Draw the Lift Chart
def lift_chart(self, pos_label = None)

# Draw the PRC Curve
def prc_curve(self, pos_label = None)

# Draw the ROC Curve
def roc_curve(self, pos_label = None)

# Compute the selected metric
def score(self, pos_label = None, cutoff: float = 0.5, method: str = "accuracy
    ")
```

Attributes

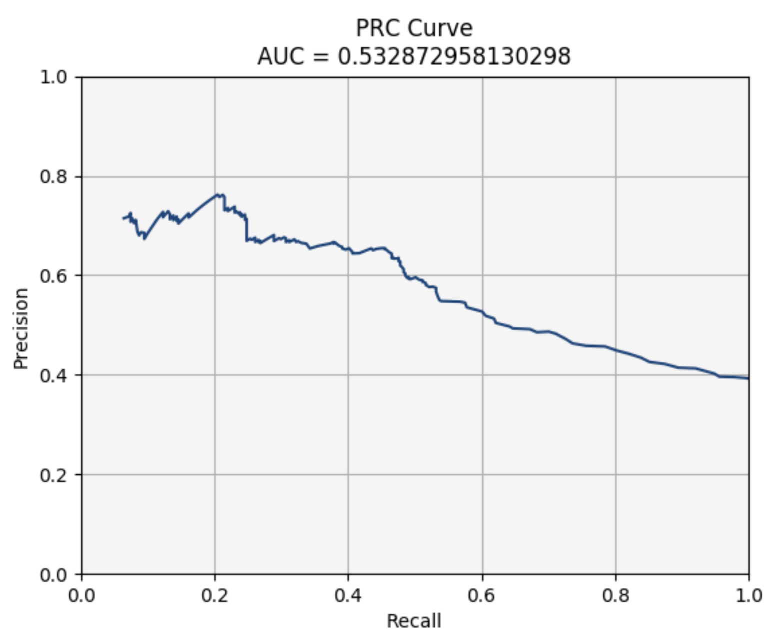
The MultinomialNB object has only one attribute:

```
self.classes # The response column classes
```

Example

```
from vertica_ml_python.learn.naive_bayes import MultinomialNB

# We can build the model
model = MultinomialNB("nb_titanic", cur)
model.fit("titanic", ["age", "pclass", "fare"], "survived")

# We can evaluate the model efficiency
```

```
model.classification_report()

# Output
                                  value
auc                   0.6682737629726706
prc_auc               0.5326937341539878
accuracy                0.53484602917342
log_loss               0.586771724006015
precision            0.20460358056265984
recall                0.7619047619047619
f1-score              0.7020729308518686
mcc                  0.25963840041342523
informedness          0.41285874619207963
markedness           0.16328126651307318
csi                  0.19230769230769232

# We can also plot the PRC curve
model.prc_curve()
```



9.9 vertica_ml_python.learn.neighbors

9.9.1 KNeighborsClassifier

Create a KNeighborsClassifier object by using the K Nearest Neighbors Algorithm. This object is using pure SQL to compute all the distances and final score. It is using CROSS JOIN and may be really expensive in some cases. As KNeighborsClassifier is using the p-distance, it is highly sensible to un-normalized data.

initialization

```python
class KNeighborsClassifier(
  name: str,
  cursor,
  n_neighbors: int = 20,
  p: int = 2)
```

Parameters

- **name:** *<str>*
  Name of the relation created after fitting the model.
- **cursor:** *<object>*
  DB cursor.
- **n_neighbors:** *<int>*, optional
  Number of neighbors.
- **p:** *<int>*, optional
  The p corresponding to the one of the p-distance (distance metric used during the model computation).

Methods

The KNeighborsClassifier object has many methods:

```python
# Compute different metrics to evaluate the model
def classification_report(self, cutoff: float = 0.5, labels = [])

# Draw the confusion matrix of the model
def confusion_matrix(self, pos_label = None, cutoff: float = 0.5)

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1, all_classes:
     bool = False)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Draw the Lift Chart
def lift_chart(self, pos_label = None)

# Draw the PRC Curve
def prc_curve(self, pos_label = None)

# Draw the ROC Curve
def roc_curve(self, pos_label = None)

# Compute the selected metric
def score(self, pos_label = None, cutoff: float = 0.5, method: str = "accuracy
    ")
```

Example

```
from vertica_ml_python.learn.neighbors import KNeighborsClassifier

# We can build the model
model = KNeighborsClassifier(cur)
model.fit("iris", ["SepalLengthCm", "SepalWidthCm"], "Species")

# We can evaluate the model
model.classification_report()

# Output
                          Iris-setosa        Iris-versicolor   \\
auc                              0.99      0.7909756097560976   \\
prc_auc            0.019615384615384597    0.6462689689983612   \\
accuracy              0.980769230769231     0.725274725274725   \\
log_loss             0.0171690390452333     0.233867972918641   \\
precision                        0.98                  0.86   \\
recall                            1.0      0.7049180327868853   \\
f1-score                          0.8       0.734496843668771   \\
mcc                  0.8082903768654761    0.4455621801770265 3   \\
informedness         0.6666666666666665    0.4715846994535519 7   \\
markedness                       0.98      0.4209756097560975   \\
csi                              0.98      0.6323529411764706   \\
                     Iris-virginica
auc                         0.79725
prc_auc            0.6555011655011654
accuracy            0.7333333333333333
log_loss            0.230656742114622
precision                      0.66
recall                        0.825
f1-score           0.7333333333333334
mcc                           0.485
informedness       0.4849999999999999
markedness         0.4849999999999999
csi                0.5789473684210527
```

9.9.2 KNeighborsRegressor

Create a KNeighborsRegressor object by using the K Nearest Neighbors Algorithm. This object is using pure SQL to compute all the distances and final score. It is using CROSS JOIN and may be really expensive in some cases. As KNeighborsRegressor is using the p-distance, it is highly sensible to un-normalized data.

initialization

```
class KNeighborsRegressor(
  name: str,
  cursor,
  n_neighbors: int = 20,
```

```
    p: int = 2)
```

Parameters

- **name:** *<str>*
  Name of the relation created after fitting the model.
- **cursor:** *<object>*
  DB cursor.
- **n_neighbors:** *<int>*, optional
  Number of neighbors.
- **p:** *<int>*, optional
  The p corresponding to the one of the p-distance (distance metric used during the model computation)

Methods

The KNeighborsRegressor object has four main methods:

```
# Save a table or a view in the DB corresponding to the model predictions for
   all the classes
def deploy_to_DB(self, name: str, view: bool = True)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Compute different metrics to evaluate the model
def regression_report(self)

# Compute the selected metric
def score(self, method: str = "r2")
```

Example

```
from vertica_ml_python.learn.neighbors import KNeighborsRegressor

# We can build the model
model = KNeighborsRegressor(cur)
model.fit("iris", ["SepalLengthCm"], "SepalWidthCm")

# We can evaluate the model
model.regression_report()

# Output
                                        value
explained_variance           0.229656749700724
max_error                                 1.32
median_absolute_error                      0.2
mean_absolute_error          0.274933333333333
mean_squared_error           0.145845333333333
```

```
r2                          0.219037147569338
```

9.9.3 NearestCentroid

Create a NearestCentroid object by using the K Nearest Centroid Algorithm. This object is using pure SQL to compute all the distances and final score. As KNeighborsClassifier is using the p-distance, it is highly sensible to un-normalized data.

initialization

```
class NearestCentroid(
  name: str,
  cursor,
  n_neighbors: int = 20,
  p: int = 2)
```

Parameters

- **name:** *<str>*
  Name of the relation created after fitting the model.
- **cursor:** *<object>*
  DB cursor.
- **p:** *<int>*, optional
  The p corresponding to the one of the p-distance (distance metric used during the model computation).

Methods

The NearestCentroid object has many methods:

```
# Compute different metrics to evaluate the model
def classification_report(self, cutoff: float = 0.5, labels = [])

# Draw the confusion matrix of the model
def confusion_matrix(self, pos_label = None, cutoff: float = 0.5)

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1, all_classes:
     bool = False)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Draw the Lift Chart
def lift_chart(self, pos_label = None)

# Draw the PRC Curve
def prc_curve(self, pos_label = None)
```

```
# Draw the ROC Curve
def roc_curve(self, pos_label = None)

# Compute the selected metric
def score(self, pos_label = None, cutoff: float = 0.5, method: str = "accuracy
    ")
```

Attributes

The NearestCentroid object has two main attributes:

```
self.centroids # Information on the different centroids
self.classes # The response column classes
```

Example

```
from vertica_ml_python.learn.neighbors import NearestCentroid

# We can build the model
model = NearestCentroid(cur)
model.fit("iris", ["SepalLengthCm", "SepalWidthCm"], "Species")

# We can evaluate the model
model.classification_report()

# Output
                              Iris-setosa          Iris-versicolor   \\
auc                    0.9646000000000002                   0.7232   \\
prc_auc                 0.901881291162864       0.5053846970363146   \\
accuracy                0.666666666666667        0.666666666666667   \\
log_loss                  0.2202052884694 4        0.265883056465362   \\
precision                             0.0                      0.0   \\
recall                                  0                        0   \\
f1-score                              0.0                      0.0   \\
mcc                                     0                        0   \\
informedness         -0.33333333333333337     -0.33333333333333337   \\
markedness                            0.0                      0.0   \\
csi                                   0.0                      0.0   \\
                           Iris-virginica
auc                                0.7887
prc_auc                0.5563484355454329
accuracy                0.666666666666667
log_loss                0.249463099665484
precision                             0.0
recall                                  0
f1-score                              0.0
mcc                                     0
informedness         -0.33333333333333337
markedness                            0.0
```

```
csi                                 0.0
```

9.9.4 LocalOutlierFactor

Create a LocalOutlierFactor object by using the Local Outlier Factor algorithm as defined by Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng and Jörg Sander. This object is using pure SQL to compute all the distances and final score. It is using CROSS JOIN and may be really expensive in some cases. It will index all the elements of the table in order to be optimal (the CROSS JOIN will happen only with IDs which are integers). As LocalOutlierFactor is using the p-distance, it is highly sensible to un-normalized data.

initialization

```
class LocalOutlierFactor(
  name: str,
  cursor,
  n_neighbors: int = 20,
  p: int = 2)
```

Parameters

- **name:** *<str>*
  Name of the relation created after fitting the model.
- **cursor:** *<object>*
  DB cursor.
- **n_neighbors:** *<int>*, optional
  Number of neighbors to consider when computing the score.
- **p:** *<int>*, optional
  The p corresponding to the one of the p-distance (distance metric used during the model computation).

Methods

The LocalOutlierFactor object has five main methods:

```
# Fit the model with the input columns
def fit(self, input_relation: str, X: list, key_columns: list = [], index = ""
    )

# Print the model information
def info(self)

# Plot the model (only possible if the number of columns <= 3)
def plot(self)

# Create a vDataframe using the final relation
def to_vdf(self)
```

The index parameter (name of the primary key column in the relation) of the 'fit' method is very important as without it an indexed copy of the main relation will be created (it could be really expensive). It is highly recommended to have a primary key column in the main table to avoid unnecessary computations.

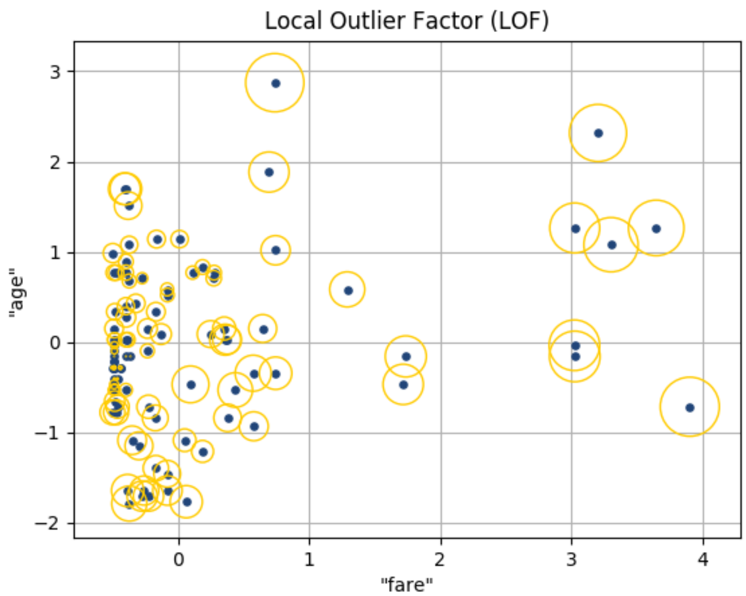
Attributes

The LocalOutlierFactor object has one main attribute:

```
self.n_errors # Number of errors
```

Example

```
from vertica_ml_python import vDataframe
from vertica_ml_python.learn.neighbors import LocalOutlierFactor

# Building a normalized relation of the features, we will use
titanic = vDataframe("titanic", cur)
# We will use a sample of the data in order to have a beautiful plot
titanic.main_relation += " TABLESAMPLE(10)"
titanic = titanic.select(["fare", "age"])
titanic.normalize()
titanic.to_db("titanic_normalize")

# We can build the model
model = LocalOutlierFactor("lof_titanic", cur)
model.fit("titanic_normalize", ["fare", "age"])

# We can see the different information relative to the model
model.info()

# Output
All the LOF scores were computed.

# And Plot the model
model.plot()
```



9.10 vertica_ml_python.learn.plot

9.10.1 elbow

```
def elbow(
        X: list,
        input_relation: str,
        cursor,
        n_cluster = (1, 15),
        init = "kmeanspp",
        max_iter: int = 50,
        tol: float = 1e-4)
```

Draw the Elbow Curve. This curve is helpful to find the number of clusters of a KMeans model.

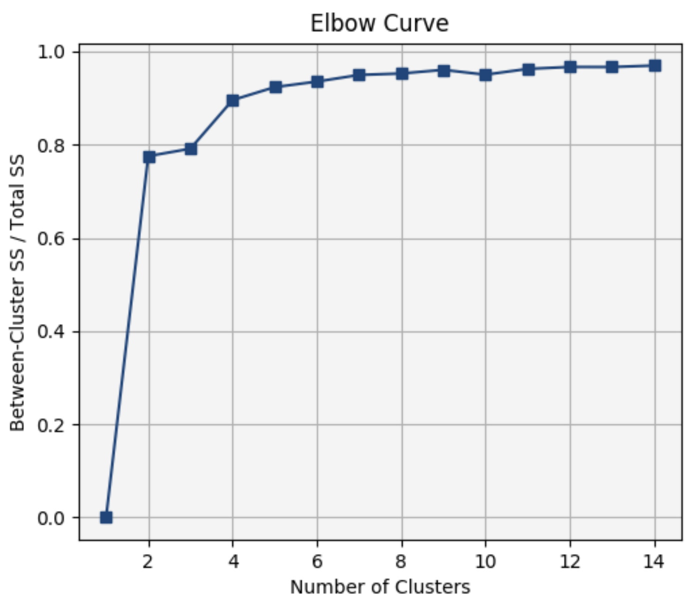
Parameters

- **X:** *<list>*
  List of the predictor columns.
- **input_relation:** *<str>*
  Relation used to train the model.
- **cursor:** *<object>*
  DB cursor.
- **n_cluster:** *<tuple>*, optional
  Tuple representing the number of cluster to start with and to end with.
- **init:** *<str or list>*, optional
  The method used to find the initial cluster centers. The method can be in {random | kmeanspp}.
  It can be also a list with the initial cluster centers to use.

- **max_iter:** *<int>*, optional
  The maximum number of iterations the algorithm performs.
- **tol:** *<float>*, optional
  Determines whether the algorithm has converged. The algorithm is considered converged after no center has moved more than a distance of 'tol' from the previous iteration.

Example

```
from vertica_ml_python.learn.plot import elbow

# We can build the model
elbow(["PetalLengthCm", "SepalWidthCm", "SepalLengthCm"], "iris", cur)
```



9.10.2 lift_chart

```
def lift_chart(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1,
    nbins: int = 1000)
```

Draw the Lift Curve.

Parameters

- **y_true:** *<str>*
  Response Column.

- **y_score:** *<str>*
  Prediction.
- **input_relation:** *<str>*
  Input Relation.
- **cursor:** *<object>*
  DB cursor.
- **pos_label:** *<anytype>*, optional
  Label of the main class.
- **nbins:** *<int>*, optional
  Number of bins used to compute the curve (= number of points).

9.10.3 prc_curve

```
def prc_curve(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
    pos_label = 1,
    nbins: int = 1000)
```

Draw the PRC Curve.

Parameters

- **y_true:** *<str>*
  Response Column.
- **y_score:** *<str>*
  Prediction.
- **input_relation:** *<str>*
  Input Relation.
- **cursor:** *<object>*
  DB cursor.
- **pos_label:** *<anytype>*, optional
  Label of the main class.
- **nbins:** *<int>*, optional
  Number of bins used to compute the curve (= number of points).

9.10.4 roc_curve

```
def roc_curve(
    y_true: str,
    y_score: str,
    input_relation: str,
    cursor,
```

```
    pos_label = 1,
    nbins: int = 1000)
```

Draw the ROC Curve.

Parameters

- **y_true:** *<str>*
  Response Column.
- **y_score:** *<str>*
  Prediction.
- **input_relation:** *<str>*
  Input Relation.
- **cursor:** *<object>*
  DB cursor.
- **pos_label:** *<anytype>*, optional
  Label of the main class.
- **nbins:** *<int>*, optional
  Number of bins used to compute the curve (= number of points).

9.11 vertica_ml_python.learn.preprocessing

9.11.1 Balance

```
def Balance(
    name: str,
    input_relation: str,
    cursor,
    y: str,
    method: str,
    ratio = 0.5)
```

Build a view with an equal distribution of the input data based on the response column.

Parameters

- **name:** *<str>*
  Output view name.
- **input_relation:** *<str>*
  Input Relation.
- **cursor:** *<object>*
  DB cursor.
- **y:** *<str>*
  Response column.
- **method:** *<str>*, optional
  Label of the main class.

- **ratio:** *<float>*, optional
  hybrid_sampling | over_sampling | under_sampling
  hybrid_sampling (default): Performs over-sampling and under-sampling on different classes so each class is equally represented.
  over_sampling: Over-samples on all classes, with the exception of the most majority class, towards the most majority class's cardinality.
  under_sampling: Under-samples on all classes, with the exception of the most minority class, towards the most minority class's cardinality.

9.11.2 CountVectorizer

Create a CountVectorizer object which creates the dictionary of the different text columns. During the process, it will create a text index and compute the number of occurrences.

initialization

```
class CountVectorizer(
    name: str,
    cursor,
    lowercase: bool = True,
    max_df: float = 1.0,
    min_df: float = 0.0,
    max_features: int = -1,
    ignore_special: bool = True,
    max_text_size: int = 2000)
```

Parameters

- **name:** *<str>*
  Name of the text index.
- **cursor:** *<object>*
  DB cursor.
- **lowercase:** *<bool>*, optional
  Convert all the elements to lowercase before processing.
- **max_df:** *<float>*, optional
  Keep the words which represent less than this float in the total dictionary distribution.
- **min_df:** *<float>*, optional
  Keep the words which represent more than this float in the total dictionary distribution.
- **max_features:** *<int>*, optional
  Keep only the top words of the dictionary
- **ignore_special:** *<bool>*, optional
  Ignore all the special characters to build the dictionary.
- **max_text_size:** *<int>*, optional
  The maximum size of the column which is the concatenation of all the text columns during the fitting.

Methods

The CountVectorizer object has 3 methods:

```
# Drop the text index
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list)

# Build a vdf from the output relation
def to_vdf(self)
```

Attributes

The CountVectorizer object has two attributes:

```
self.stop_words # The words not added to the vocabulary because of some
    parameters
self.vocabulary # The final vocabulary
```

Example

```
from vertica_ml_python.learn.preprocessing import CountVectorizer

# We can build the model
model = CountVectorizer("name_voc", cur)
model.fit("titanic", ["Name"])

# We can export the dictionary
model.to_vdf()

# Output
         token                      df     cnt     rnk
0           mr     0.148163100524828421     734       1
1         miss     0.046023415421881308     228       2
2          mrs     0.037343560758982640     185       3
3      william     0.016148566814695196      80       4
4         john     0.013726281792490916      68       5
...        ...                      ...     ...     ...
Name: name_voc, Number of rows: 1841, Number of columns: 4
```

9.11.3 Normalizer

Create a Normalizer object which can be used to normalize the data.

initialization

```
class Normalizer(name: str, cursor, method: str = "zscore")
```

Parameters

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  DB cursor.
- **method:** *<str>*, optional
  The normalization method to use: {minmax | zscore | robust_zscore}

Methods

The Normalizer object has 3 methods:

```
# Drop the model from the DB
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list)

# Build a vdf from the output relation
def to_vdf(self, reverse = False)
```

Attributes

The Normalizer object has only one attribute:

```
self.param # The information about columns used to normalize the data.
```

Example

```
from vertica_ml_python.learn.decomposition import Normalizer

# We can build the model
model = Normalizer("norm_iris", cur)
model.fit("iris", ["SepalLengthCm", "SepalWidthCm"])

# We can look at the model parameters
model.param

# Output
       column_name                avg             std_dev
0    sepallengthcm    5.84333333333333    0.828066127977865
1     sepalwidthcm               3.054    0.433594311362172

# We can export the model
model.to_vdf()

# Output
          SepalLengthCm          SepalWidthCm
0    -0.897673879196763      1.02861128089724
1      -1.13920048346495    -0.12454037930146
```

```
2      -1.38072708773314     0.33672028477802
3      -1.50149038986723     0.10608995273828
4      -1.01843718133086     1.25924161293698
...                    ...                    ...
Name: norm_iris, Number of rows: 150, Number of columns: 2
```

9.11.4 OneHotEncoder

Create a OneHotEncoder object which can be used to encode the data using dummies.

initialization

```
class OneHotEncoder(
    name: str,
    cursor,
    extra_levels: dict = {},
    drop_first: bool = True,
    ignore_null: bool = True)
```

Parameters

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  DB cursor.
- **extra_levels:** *<dict>*, optional
  Additional levels in each category that are not in the input relation.
- **drop_first:** *<bool>*, optional
  If true, treat the first level of the categorical variable as the reference level.
- **ignore_null:** *<bool>*, optional
  If false, Null values in input?columns are treated as a categorical level.

Methods

The OneHotEncoder object has 3 methods:

```
# Drop the model from the DB
def drop(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list)

# Build a vdf from the output relation
def to_vdf(self, reverse = False)
```

Attributes

The OneHotEncoder object has only one attribute:

```
self.param # The information about columns used to get the data dummies.
```

Example

```
from vertica_ml_python.learn.decomposition import OneHotEncoder

# We can build the model
model = OneHotEncoder("dummies_iris", cur)
model.fit("iris", ["Species"])

# We can look at the model parameters
model.param

# Output
     category_name     category_level    category_level_index
0          species        Iris-setosa                       0
1          species    Iris-versicolor                       1
2          species     Iris-virginica                       2

# We can export the model
model.to_vdf()

# Output
         species    species_iris-versicolor    species_iris-virginica
0    Iris-setosa                          0                         0
1    Iris-setosa                          0                         0
2    Iris-setosa                          0                         0
3    Iris-setosa                          0                         0
4    Iris-setosa                          0                         0
...          ...                        ...                       ...
Name: dummies_iris, Number of rows: 150, Number of columns: 3
```

9.12 vertica_ml_python.learn.svm

9.12.1 LinearSVC

Create a LinearSVC object by using the Vertica Highly Distributed and Scalable SVM on the data.

initialization

```
class LinearSVC(
    name: str,
    cursor,
    tol: float = 1e-4,
    C: float = 1.0,
    fit_intercept: bool = True,
    intercept_scaling: float = 1.0,
```

```
    intercept_mode: str = "regularized",
    class_weight: list = [1, 1],
    max_iter: int = 100)
```

Parameters

- **name:** *<str>*
  Name of the relation created after fitting the model.
- **cursor:** *<object>*
  DB cursor.
- **tol:** *<float>*, optional
  Used to control accuracy.
- **C:** *<float>*, optional
  The weight for misclassification cost. The algorithm minimizes the regularization cost and the misclassification cost.
- **fit_intercept:** *<bool>*, optional
  A bool to fit also the intercept.
- **intercept_scaling:** *<float>*, optional
  A float value, serves as the value of a dummy feature whose coefficient Vertica uses to calculate the model intercept. Because the dummy feature is not in the training data, its values are set to a constant, by default set to 1.
- **intercept_mode:** *<str>*, optional
  A string that specifies how to treat the intercept, one of the following: {regularized | unregularized}
- **class_weight:** *<list>*, optional
  Specifies how to determine weights of the two classes. It can be in {None | auto} or a list of 2 elements.
- **max_iter:** *<int>*, optional
  The maximum number of iterations that the algorithm performs.

Methods

The LinearSVC object has many methods:

```
# Add the LinearSVC prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "", cutoff: float = 0.5)

# Compute different metrics to evaluate the model
def classification_report(self, cutoff: float = 0.5)

# Draw the confusion matrix of the model
def confusion_matrix(self, cutoff: float = 0.5)

# Save a table or a view in the DB corresponding to the model predictions for
    all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1)

# Drop the model from the DB
```

```
def drop(self)

# Compute the importance of each feature
def features_importance(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Draw the Lift Chart
def lift_chart(self)

# Plot the SVM if it is possible (The length of X must be lesser of equal to
    3)
def plot(self)

# Draw the PRC Curve
def prc_curve(self)

# Draw the ROC Curve
def roc_curve(self)

# Compute the selected metric
def score(self, cutoff: float = 0.5, method: str = "accuracy")
```

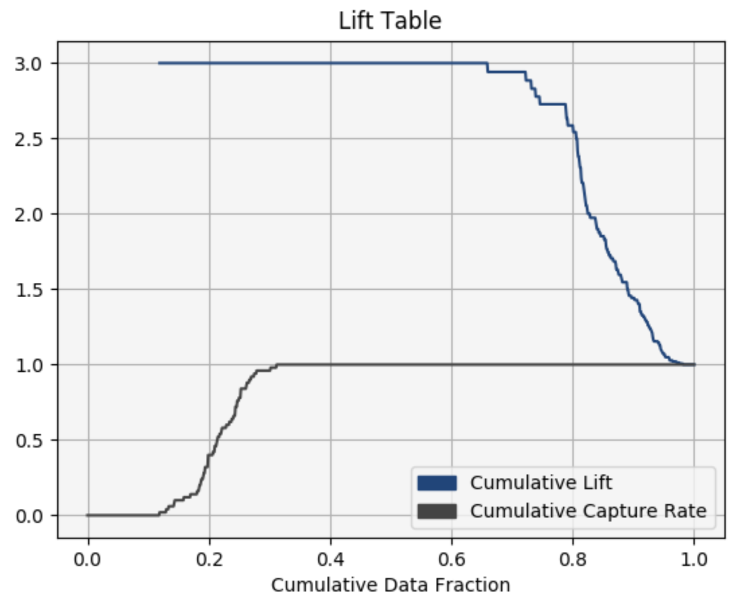
Attributes

The LinearSVC object has only one attribute:

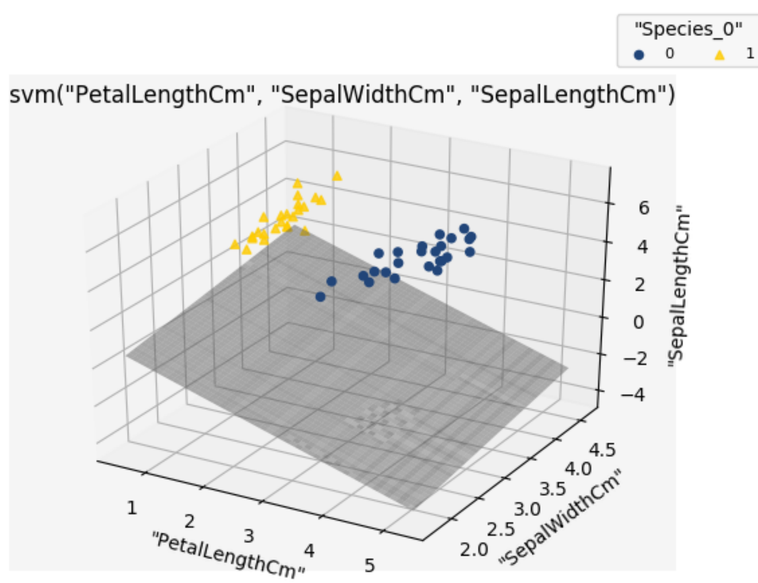
```
self.coef # Informations about the model coefficients
```

Example

```
from vertica_ml_python import vDataframe
from vertica_ml_python.learn.svm import LinearSVC

# We create dummies
iris = vDataframe("iris", cur)
iris["Species"].get_dummies(use_numbers_as_suffix = True)
iris.to_db("iris_dummy")

# We can build the model
model = LinearSVC("svc_iris", cur)
model.fit("iris_dummy", ["PetalLengthCm", "SepalWidthCm", "SepalLengthCm"], "
    Species_0")

# We can draw the lift chart of the model
model.lift_chart()
```




```
# We can also draw the model
model.plot()
```



9.12.2 LinearSVR

Create a LinearSVR object by using the Highly Distributed and Scalable SVM on the data.

initialization

```
class LinearSVR(
```

```
    name: str,
    cursor,
    tol: float = 1e-4,
    C: float = 1.0,
    fit_intercept: bool = True,
    intercept_scaling: float = 1.0,
    intercept_mode: str = "regularized",
    acceptable_error_margin: float = 0.1,
    max_iter: int = 100)
```

Parameters

- **name:** *<str>*
  Name of the model.
- **cursor:** *<object>*
  DB cursor.
- **tol:** *<float>*, optional
  Used to control accuracy.
- **C:** *<float>*, optional
  The weight for misclassification cost. The algorithm minimizes the regularization cost and the misclassification cost.
- **fit_intercept:** *<bool>*, optional
  A bool to fit also the intercept.
- **intercept_scaling:** *<float>*, optional
  A float value, serves as the value of a dummy feature whose coefficient Vertica uses to calculate the model intercept. Because the dummy feature is not in the training data, its values are set to a constant, by default set to 1.
- **intercept_mode:** *<str>*, optional
  A string that specifies how to treat the intercept, one of the following: {regularized | unregularized}
- **acceptable_error_margin:** *<float>*, optional
  Defines the acceptable error margin. Any data points outside this region add a penalty to the cost function.
- **max_iter:** *<int>*, optional
  The maximum number of iterations that the algorithm performs.

Methods

The LinearSVR object has many methods:

```
# Add the LinearSVR prediction in a vDataframe
def add_to_vdf(self, vdf, name: str = "")

# Save a table or a view in the DB corresponding to the model predictions for all the classes
def deploy_to_DB(self, name: str, view: bool = True, cutoff = -1)

# Drop the model from the DB
```

```
def drop(self)

# Compute the importance of each feature
def features_importance(self)

# Fit the model with the input columns
def fit(self, input_relation: str, X: list, y: str, test_relation: str = "")

# Plot the SVM if it is possible (The length of X must be lesser of equal to
    3)
def plot(self)

# Compute different metrics to evaluate the model
def regression_report(self)

# Compute the selected metric
def score(self, method: str = "r2")
```

Attributes

The LinearSVR object has only one attribute:

```
self.coef # Informations about the model coefficients
```

Example

```
from vertica_ml_python.learn.svm import LinearSVR

# We can build the model
model = LinearSVR("svc_iris", cur)
model.fit("iris", ["SepalLengthCm"], "SepalWidthCm")

# We can evaluate the model
model.regression_report()

# Output
                                          value
explained_variance         -0.0357335274222341
max_error                     1.3548634384927
median_absolute_error       0.287106492764341
mean_absolute_error         0.342827242830171
mean_squared_error          0.193424390470397
r2                         -0.035736010600985

# We can also get the model coefficients
model.coef()

# Output
         predictor           coefficient
```

```
0          Intercept       2.71974116342001
1     sepallengthcm     0.0570869119451395
```

9.13 vertica_ml_python.learn.tree

The four following algorithm have the same methods and attributes than the Random Forest models. Decision trees can be seen as Random Forest with one tree and using the entire dataset. Dummy Trees are infinite depth Decision Trees.

9.13.1 DecisionTreeClassifier

```
def DecisionTreeClassifier(
               name: str,
               cursor,
               max_features = "auto",
               max_leaf_nodes: int = 1e9,
               max_depth: int = 100,
               min_samples_leaf: int = 1,
               min_info_gain: float = 0.0,
               nbins: int = 32)
```

Single Decision Tree Classifier.

9.13.2 DecisionTreeRegressor

```
def DecisionTreeRegressor(
              name: str,
              cursor,
              max_features = "auto",
              max_leaf_nodes: int = 1e9,
              max_depth: int = 100,
              min_samples_leaf: int = 1,
              min_info_gain: float = 0.0,
              nbins: int = 32)
```

Single Decision Tree Regressor.

9.13.3 DummyTreeClassifier

```
def DummyTreeClassifier(name: str, cursor)
```

Dummy Tree Classifier. This classifier learns by heart the training data.

9.13.4 DummyTreeRegressor

```
def DummyTreeRegressor(name: str, cursor)
```

Dummy Tree Regressor. This regressor learns by heart the training data.

10 Contact

If you have any question or issue, feel free to write it at `https://github.com/vertica/Vertica-ML-Python/issues`. If you really need a specific function or algorithm and it is feasible using Vertica SQL, I can make your wish come true. Write me at badr.ouali@microfocus.com and I will try to find a solution to your problem.

11 License