Vertica ML Python Workshop

Exercise 0: Basics

Ouali Badr

December 2, 2019

Executive Summary

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

Louis Pasteur

VERTICA ML PYTHON allows the users to use Vertica advanced analytics and Machine Learning with a Python front-end Interface. In this exercise, you'll learn some basics to begin your fantastic Data Science Journey with the API. As a summary:

- Create a Vertica cursor
- Create a Virtual Dataframe
- Reading a CSV file
- Summarize descriptive statistics
- Draw histograms
- Plot cumulative sums and max

Contents

1 Presentation

Setting up correctly the environment is crucial. The purpose of these exercises is to give you basics to let you build your own path. In the Exercise 0, we will learn some basics to start any Data Science project. We will use the Amazon Dataset.

Forest fires are a serious problem for the preservation of the Tropical Forests. Understanding the frequency of forest fires in a time series can help to take action to prevent them. Brazil has the largest rainforest on the planet that is the Amazon rainforest.

The file amazon.csv represents the number of forest fires in Brazil. These time series include the period of approximately 10 years (1998 to 2017). The data were obtained from the official website of the Brazilian government. We will try to explore the data and understand when forest fires are happening.

For this study, we have access to the date, the state (in Brazil) and the number of forest fires each day. Let's look at the functions, we will use during the study.

2 Functions used during the Exercise

2.1 cumsum

Library: vertica_ml_python.vDataframe

Explanation: Cumulative sums (and other cumulative windows functions) are primordial to do easy predictions on seasonal time series. The time series are represented by 'order_by' (a timestamp or a list of different columns which represent a timestamp) and other categorical and numerical columns. The 'by' parameter allows to group by specific columns. If needed, the method 'rolling' allows to do more customised moving windows.

```
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
```

2.2 describe

Library: vertica_ml_python.vDataframe

Explanation: 'describe' is one of the first method to use to have a first look at the data. We can easily understand the variables distribution and differents other information (number of missing elements, the mode...). If the method is set to "numerical", only numerical features will be summarised whereas if it is set to "categorical" all the features will be summarised. The parameter 'unique' is used to also compute the cardinality of each element (an additional query is generated).

```
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
```

2.3 hist

Library: vertica_ml_python.vDataframe[]

Explanation: The histograms are very important elements for a first data exploration. It helps to understand the different variables and the link between all of them. The parameter 'method' can be set to many different aggregations (count | density | avg | min | max | sum) and the parameter 'of' is the column in which you want to apply the aggregation. You'll see many times the parameter 'max_cardinality' which determine if a variable is categorical or not by using a threshold. If the column cardinality is lesser than 'max_cardinality' then the feature is automatically categorical.

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

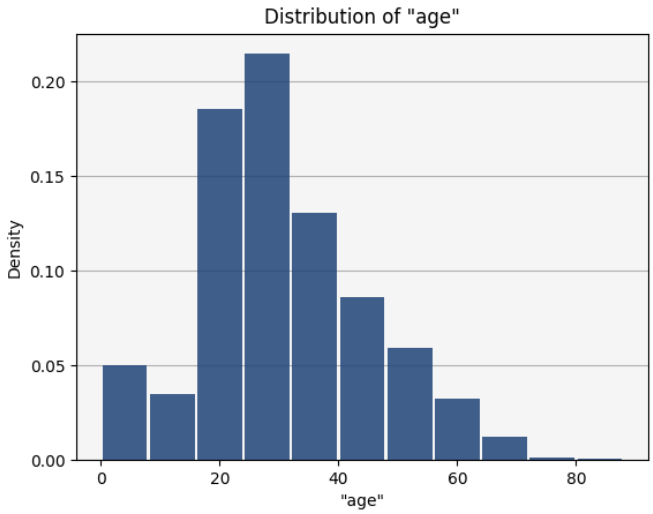
Draw the column histogram.

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()
```



2.4 plot

Library: vertica_ml_python.vDataframe[]

Explanation: Drawing time series can be very useful to understand the data. The time series are represented by 'ts' (a timestamp) and other categorical and numerical columns. 'start_date' and 'end_date' allow the user to filter the data on a specific time range. The 'by' parameter allows to group by a specific column to plot time series representing for example different IDs.

```
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.

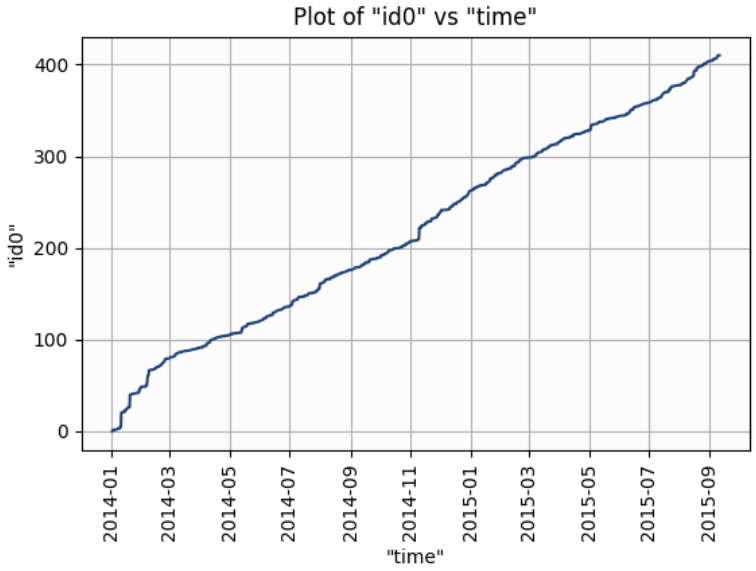
Returns

The parent vDataframe.

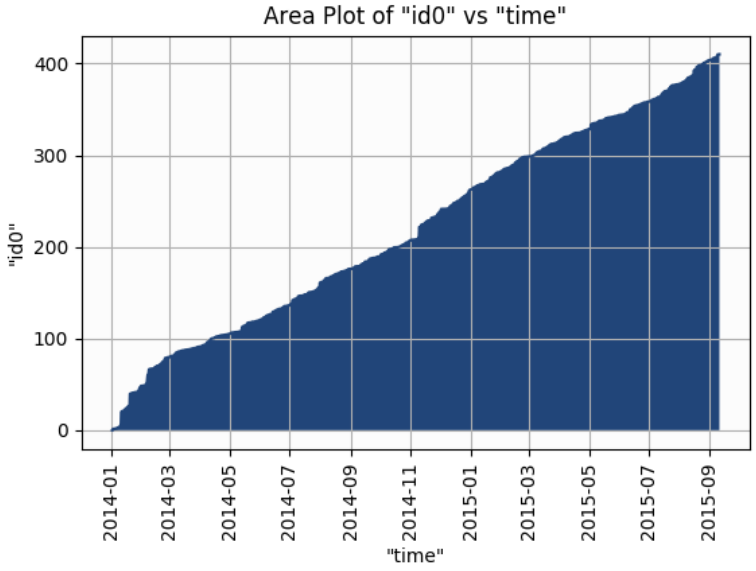
Example

```
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")
```

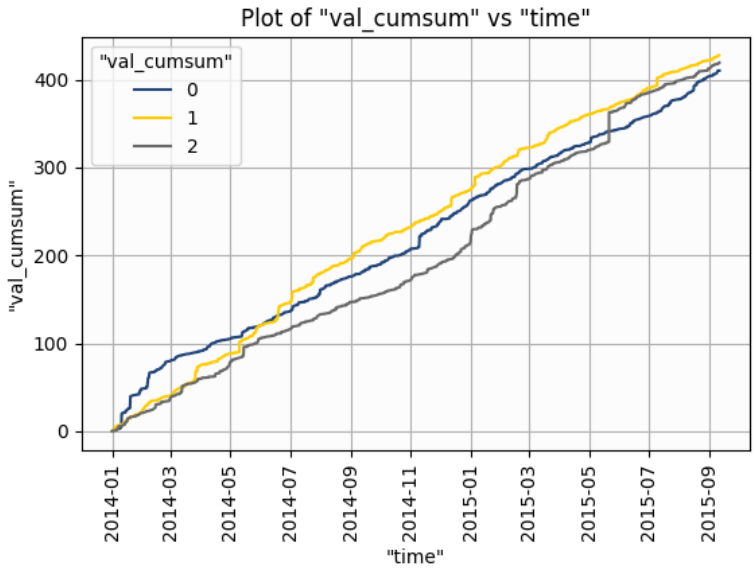



```
sm["id0"].plot(ts = "time", area = True)
```




```
# 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")
```



2.5 read_csv

Library: vertica_ml_python.utilities

Explanation: This function is used to parse automatically CSV files. It is using Flex Tables to identify the data types. Flex Tables will parse all the data, that's why the 'parse_n_lines' parameter can limit the process by building a sample of the CSV file to only parse it in order to identify the data types. If the data volume is too big, you should consider using this parameter to avoid waiting too much. The 'return_dlist' parameter can return to you the guessed types without creating the table. It can be used if you want some customised types for some columns. The 'genSQL' parameter can be used to see the generated SQL. If you already know the data types, please complete the 'header_names' and 'dtype' parameters. Do not forget that if your cursor is not set to 'autocommit', this function could never ends. That's why I recommend 'pyodbc' which is perfectly suitable for Vertica ML Python.

```
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
```

3 Questions

Turn on Jupyter with the 'jupyter notebook' command. Start the notebook exercise0.ipynb and answer the following questions.

- **Question 1:** Compute the histogram of the number of forest fires per year and the one per state. What do you notice ?
- **Question 2:** Compute the cumulative sum of the number of forest fires in Brazil since the start of the dataset. Plot the time series using the 'plot' method. What do you notice?
- **Question 3:** Mato Grosso seems to be subject to a lot of forest fires. Filter the data and find all the possible information on this State.
- **Question 4:** Plot the cumulative sum of the number of forest fires group by state (do the same with the cummax and explain what this graphic reveals).