Vertica ML Python Workshop

Exercise 8: Time Series

Ouali Badr

December 6, 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:

- Compute frequencies
- Use Moving Windows
- Extract Information on timestamps
- Compute a cumulative sum

Contents

1 Presentation

Time Series are everywhere. Any Data which contain the timestamp of each record are time series. The data can be ordered by this variable to explain what happened during the time. Any raw data are time series as we can label the date-time of the action. Non-time series data are the result of aggregations.

Time Series are the most rich type of information you can have in Data Science as the purpose is mainly to predict the future. With time series, you have an unlimited features that you can create:

- **Sessions:** To study users behaviour, we group his activities into what we call ?sessions?. It can help us to understand what happened when he decided to take an action.
- **Moving Windows:** They are the ability to compute aggregations at a successions of actions during a specific time range. It helps to find specific patterns during a time interval.
- **Frequencies:** They are the simplest way to build features from time series. We use direct aggregations on a day/month/year or even at a specific time range to find patterns.

During this study, we will use the 'flights' dataset to understand what could possibly influence the flights delay.

2 Functions used during the Exercise

2.1 groupby

Library: vertica_ml_python.vDataframe

```
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
```

2.2 date_part

Library: vertica_ml_python.vDataframe[]

```
vDataframe[].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 parent vDataframe.

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
```

2.3 rolling

Library: vertica_ml_python.vDataframe

```
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
```

3 Questions

Turn on Jupyter with the 'jupyter notebook' command. Start the notebook exercise8.ipynb and answer to the following questions.

- **Question 1:** Find the average number of flights (departure) per day and per month in the US (You should probably create new features: day/month)
- **Question 2:** A flight delay could be the result of a company mishandling. Compute the number of flights the company has to manage 1 hour preceding each flight and two hours following.
- **Question 3:** Create a new variable is_dep_delay which is equal to 1 if the departure_delay is greater than 10 minutes. Compute the correlation between is_dep_delay and the previously created feature. What do you notice ? Which information is missing to create a more relevant feature ?

- **Question 4:** Compute the cumulative number of delay per company. What do you notice ? How can we solve this problem ?