PyGMQL Documentation

Release 0.1

Luca Nanni

Feb 22, 2018

Contents:

PyGMQL is a python module that enables the user to perform operation on genomic data in a scalable way.

CHAPTER 1

Installation

1.1 Prerequisites

In order to use the library you need to have Java installed in your system. And, in particular, the environment variable JAVA_HOME must be setted to your current Java installation.

1.2 Using the github repository

You can install this library by downloading its source code from the github repository:

```
git clone https://github.com/DEIB-GECO/PyGMQL.git
```

and then using:

```
cd PyGMQL/
pip install -e .
```

This will install the library and its dependencies in your system.

CHAPTER 2

Introduction

In this brief tutorial we will explain the typical workflow of the user of PyGMQL. In the github page of the project you can find a lot more example of usage of the library.

You can use this library both interactively and programmatically. **We strongly suggest to use it inside a Jupyter notebook for the best graphical render and data exploration.**

2.1 Importing the library

To import the library simply type:

```
import gmql as gl
```

If it is the first time you use PyGMQL, the Scala backend program will be downloaded. Therefore we suggest to be connected to the internet the first time you use the library.

2.2 Loading of data

The first thing we want to do with PyGMQL is to load our data. You can do that by calling the `gmql.dataset.loaders.Loader.load_from_path()`. This method loads a reference to a local gmql dataset in memory and creates a `GMQLDataset`. If the dataset in the specified directory is already GMQL standard (has the xml schema file), you only need to do the following:

```
dataset = gl.load_from_path(local_path="path/to/local/dataset")
```

while, if the dataset has no schema, you need to provide it manually. This can be done by creating a custom parser using `BedParser` like in the following:

```
custom_parser = gl.parsers.BedParser(chrPos=0, startPos=1, stopPos=2,
                                     otherPos=[(3, "gene", "string")])
dataset = gl.load_from_path(local_path="path/to/local/dataset", parser=custom_parser)
```

2.3 Writing a GMQL query

Now we have a dataset. We can now perform some GMQL operations on it. For example, we want to select samples that satisfy a specific metadata condition:

```
selected_samples = dataset[ (dataset['cell'] == 'es') | (dataset['tumor'] == 'brca') ]
```

Each operation on a GMQLDataset returns an other GMQLDataset. You can also do operations using two datasets:

```
other_dataset = gl.load_from_path("path/to/other/local/dataset")

union_dataset = dataset.union(other_dataset)                                    # the union
join_dataset = dataset.join(other_dataset, predicate=[gl.MD(10000)])    # a join
```

2.4 Materializing a result

PyGMQL implements a lazy execution strategy. No operation is performed until a materialize operation is requested:

```
result = join_dataset.materialize()
```

If nothing is passed to the materialize operation, all the data are directly loaded in memory without writing the result dataset to the disk. If you want also to save the data for future computation you need to specify the output path:

```
result = join_dataset.materialize("path/to/output/dataset")
```

2.5 The result data structure

When you materialize a result, a `GDataframe` object is returned. This object is a wrapper of two pandas dataframes, one for the regions and one for the metadata. You can access them in the following way:

```
result.meta     # for the metadata
result.regs     # for the regions
```

These dataframes are structured as follows:

- The region dataframe puts in every line a different region. The source sample for the specific region is the index of the line. Each column represent a field of the region data.
- The metadata dataframe has one row for each sample in the dataset. Each column represent a different metadata attribute. Each cell of this dataframe represent the values of a specific metadata for that specific sample. Multiple values are allowed for each cell.

CHAPTER 3

The Genomic data model

As we have said, PyGMQL is a Python interface to the GMQL system. In order to understand how the library works, a little insights on the data model used by GMQL is necessary.

GMQL is based on a representation of the genomic information known as GDM - Genomic Data Model. Datasets are composed of samples , which in turn contains two kinds of data:

- **Region values** (or simply **regions**), aligned w.r.t. a given reference, with specific left-right ends within a chromosome. Regions can store different information regarding the "spot" they mark in a particular sample, such as region length or statistical significance. Regions of the model describe processed data, e.g. mutations, expression or bindings; they have a **schema** , with 5 common attributes (id , chr , left , right , strand) including the id of the region and the region coordinates, along the aligned reference genome, and then arbitrary typed attributes. This provides interoperability across a plethora of genomic data formats
- **Metadata**, storing all the knowledge about the particular sample, are arbitrary attribute-value pairs, independent from any standardization attempt; they trace the data provenance, including biological and clinical aspects

CHAPTER 4

The GMQLDataset

Here we present the functions that can be used on a GMQLDataset.

4.1 Loading functions

You can create a GMQLDataset by loading the data using the following functions:

CHAPTER 5

GDataframe

CHAPTER 6

Remote data management

PyGMQL can be used in two different ways. The first one (and the most intuitive and classical one) is to use it like any other computational library.

The second way is to make the library interact with a remote server. In order to do that you need to create an instance of a *RemoteManager*.

CHAPTER 7

Library settings

The following functions define the behavior of the library

7.1 Logging and progress bars

- genindex
- modindex
- search

CHAPTER 8

Indices and tables

8.1 Dataset structures

GMQLDataset.GMQLDataset
GDataframe.GDataframe

8.2 Dataset loading functions

load_from_path
load_from_remote

8.3 Parsing

BedParser.BedParser

8.4 Aggregates operators

COUNT
SUM
MIN
MAX
AVG
BAG
STD
MEDIAN

Continued on next page

Table 4 – continued from previous page

Q1
Q2
Q3

8.5 Genometric predicates

MD
DLE
DGE
UP
DOWN