qualipy Documentation

Release 0.1.0

Boudewijn Aasman

Aug 01, 2022

CONTENTS

Qualipy is a library designed to track and monitor real-time and retrospective data and provide automated anomaly detection, reporting, and analysis on that data.

It does the following:

- **Provide a library that allows you to:**
  - Create generic aggregate functions
  - Reflect any column that aggregates can be tracked on (Currently Pandas Series, Spark Columns and SQL columns)
  - Execute these aggregate functions as your data pipeline is running
- Track and maintain these aggregate values in a separate location (Either SQlite or Postgres)
- **Generate reports describing your data in real time**
  - Longitudinally describing all batches over time
  - Describe a single batch to understand at a deeper level
- Run automated anomaly detection on all collected aggregates (and has an extendible anomaly detection model)
- **Provide a command line interface to:**
  - Execute anomaly detection
  - Produce anomaly, comparison or batch reports
  - Interact with the historical data

Contents:

CHAPTER

ONE

INSTALLATION AND SETUP

I highly recommend using conda to simplify the installation of fbprophet/pystan, but any package manager will work. fbprophet is not a required library, but it provides the best anomaly detection results.

To create a new virtual environment:

```
$ conda create --name qpy python=3
$ conda activate qpy
```

To install using pip:

```
$ pip install qualipy
$ conda install -c conda-forge pystan
$ conda install -c conda-forge fbprophet
```

To install using git:

```
$ git clone https://github.com/baasman/qualipy
$ cd qualipy
$ pip install .
$ conda install -c conda-forge pystan
$ conda install -c conda-forge fbprophet
```

By default, Qualipy will create a SQLite file to store and maintain all data. However, as data grows larger and more complex, Postgres becomes recommended. This is not a guide on running Postgres, but to get setup easily, docker is recommended.

CHAPTER

TWO

OVERVIEW

2.1 What it looks like

Take a look at the following example to get a view of what a qualipy pipeline looks like:

```
from qualipy.reflect.column import column
from qualipy.reflect.function import function
from qualipy import Qualipy, Project
from qualipy.backends.pandas_backend.pandas_types import FloatType
from qualipy.backends.pandas_backend.dataset import PandasData
from qualipy.datasets import stocks


@function(return_format=float)
def mean(data, column):
    return data[column].mean()


price_column = column(column_name="price", column_type=FloatType(), functions=[mean])


project = Project(project_name="example", config_dir="/tmp/.qualipy")
project.add_column(price_column)

qualipy = Qualipy(project=project)
stocks = PandasData(stocks)
stocks.set_stratify_rule("symbol")
qualipy.set_chunked_dataset(stocks, time_column="date", time_freq="1M")
qualipy.run(autocommit=True)
```

2.2 What just happened?

First, we created a function called ‘mean’, using the function decorator. This establishes a numerical aggregator that returns the mean of a column, in a float format. This could now be applied to any live - numerical data and tracked.

Second, we create a mapping between the column “price” of the stocks data and the *price_column* object. This establishes the mapping, enforces a datatype, and specifies what metrics to track. Many more options are available.

Third, we establish a Project. This project will be the overarching object that persists each batch’s aggregate data.

Now that we've set up the boilerplate of Qualipy, we can get to actually running it on some real data. All we need to do instantiate the Qualipy object and tie it to whatever project we want to track. We also need to instantiate our dataset, in this case a pandas dataset, and optionally define a way to stratify in incoming data. All that's left is to set the current dataset, and run.

CHAPTER

THREE

TUTORIALS AND RECIPES

For beginners getting into using Qualipy, I'd recommend following these tutorials in order

3.1 Tutorial 1

Explore a dataset by collecting aggregates over chunks of a dataset: Example_1.

Concepts:

1. Creating a Pandas function
2. Defining core components in Qualipy
3. Simulate a time series by chunking the input data

3.2 Tutorial 2

Explore a single batch of data using Qualipy: Example_2.

Concepts:

1. Creating a Pandas function
2. Defining core components in Qualipy
3. Profiling a dataset by automatically collecting useful information about a single batch of data

3.3 Tutorial 3

Similar to example 1, but this is a more involved example and includes flat data as a source: Example_3.

Concepts:

1. Creating a Pandas function
2. Using reference names for columns to refer to them at different times
3. Working with numerical and categorical data
4. Simulate a time series by chunking the input data

3.4 Tutorial 4

Like example 3, but creates a new more complex function: Example_4.

Concepts:

1. Creating a Pandas function
2. Creating another Pandas function with an additional argument
3. Specifying the argument in the column definition
4. Using reference names for columns to refer to them at different times
5. Working with numerical and categorical data
6. Simulate a time series by chunking the input data

3.5 Recipe 1

Use this to analyze a pandas dataframe. All you need to fill in is the numeric, categorical, and datetime columns, and the rest should work like magic!

CHAPTER FOUR

USER GUIDE

4.1 Creating Functions

qualipy.reflect.function.**function**(*allowed_arguments: ~typing.Optional[~typing.List[str]] = None, return_format: type = typing.Union[float, str], arguments: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, fail: bool = False, display_name: ~typing.Optional[str] = None, description: ~typing.Optional[str] = None, input_format: type = <class 'float'>, custom_value_return_format: ~typing.Optional[type] = None*) → Callable

Define a function that can be applied to a qualipy dataset

Use this decorator to specify a qualipy function, and describe how it will function when executed. Whatever function this decorator is used for must abide by three rules:

1) The first argument must be *data* - This is the data object you pass to Qualipy
2) **The second argument is the column - This is the name of the column the function**
   is being applied to.
3) **Any arguments as they correspond to *allowed_arguments* - They must contain the same**
   name exactly.

Parameters

- **allowed_arguments** – An optional list that specifies what arguments can be passed to the function at runtime
- **return_format** – Used for rendering purposes on the reporting. Can be either float, int, str, dict, or bool
- **fail** – If this rule returns a boolean, should the process halt given a False?
- **display_name** – This is how the function would be displayed on a report. if not given, it will take the name of the function itself
- **description** – If given, this will be displayed when hovering over the function name in a report

Returns
Any value that corresponds to the appropriate return_format

Example 1 - A simple function with no additional arguments:

```
import qualipy as qpy
@qpy.function(return_format=float)
def mean(data, column):
    return data[column].mean()
```

Per the rules, `data` represents the data passed through, in this case a pandas DataFrame,
`column` is the string name of column is used to access the column from the DataFrame.

Additionally, the method `mean` returns a float value, which is consistent with the `return_format` set in the decorator call.

Example 2 - A simple function with additional arguments:

```
@qpy.function(return_format=int, allowed_arguments=["standard_deviations"])
def std_over_limit(data, column, standard_deviations):
    mean = data[column].mean()
    std = data[column].std()
    data = data[
        (data[column] < (mean - standard_deviations * std))
        | (data[column] > (mean + standard_deviations * std))
    ]
    return data.shape[0]
```

Example 3 - A function when running SQL as backend:

```
@qpy.function(return_format=float)
def mean(data, column):
    return data.engine.execute(
        sa.select([sa.func.avg(sa.column(column))]).select_from(data._table)
    ).scalar()
```

4.2 Creating a mapping

`qualipy.reflect.column.`**`column`**(*column_name: Optional[Union[str, List[str]]] = None*, *column_type=None*, *force_type: bool = False*, *overwrite_type: bool = False*, *null: bool = True*, *force_null: bool = False*, *unique: bool = False*, *is_category: bool = False*, *is_date: bool = False*, *split_on: Optional[str] = None*, *column_stage_collection_name: Optional[str] = None*, *functions: Optional[List[Union[Callable, Dict]]] = None*, *extra_functions: Optional[Dict[str, Dict]] = None*)

This allows us to map to a column of a data object.

This is one of the essential components of Qualipy. Using column `allows` us to map to a specific column of whatever data object we are reflecting, and specify what that column should look like - as well as apply any aggregate functions we've defined.

Note - You must explicitly add it to the Project object in order for it to run.

Parameters

- **`column_name`** – The name of the column in the data object - Generally either the column name in the pandas or SQL table.
- **`column_type`** – Useful if you want to enforce types in a pandas DataFrame. See (link here) DataTypes section for more information.

- **force_type** – If column_type is used, should the type be enforced. Setting this to True means that the entire process will halt if right type is not present.
- **overwrite_type** – This is useful if the aggregate function requires a specific datatype for it to be computed.
- **null** – Can the column contain missing values
- **force_null** – If null is set to False - should the process fail given there are missing values present.
- **unique** – Should uniqueness in the column be enforced.
- **is_category** – Denoting a column as a category has several consequences - including automatically collecting counts for each category.
- **functions** – A list of property defined functions.
- **extra_functions** – If this mapping is used for multiple columns but want a function to be applied to only one of the columns, use this. See example for more information.

Returns
A column object that can be added to a Project. See Project for more details.

Example 1 - Reflect a pandas column with one function:

```
price = qpy.column(column_name="price", column_type=FloatType(), functions=[mean])
```

Here, `price` is the name of the pandas column. We want to column to be of float type, and we're collecting the mean of the price.

Example 2 - Reflect a column, and call a function with arguments:

```
price = qpy.column(
    column_name="price",
    column_type=FloatType(),
    functions=[{"function": std_over_limit, "parameters": {"standard_deviations": 3}}],
)
```

Example 3 - Reflect multiple columns, and call a function on just one of them:

```
num_columns = qpy.column(
    column_name=["price", "some_other_column"],
    column_type=FloatType(),
    functions=[mean],
    extra_functions={
        "price": [
            {"function": std_over_limit, "parameters": {"standard_deviations": 3}},
        ],
    },
)
```

In this scenario, `mean` will be applied to `price`, but `std_over_limit` will only be applied `price`

4.3 Project

`class qualipy.project.Project`(*project_name: str*, *config_dir: str*, *re_init: bool = False*)

The project class points to a specific configuration, and holds all mappings.

It also includes a lot of useful utility functions for working with the management of projects

`__init__`(*project_name: str*, *config_dir: str*, *re_init: bool = False*)

Parameters

- **`project_name`** – The name of the project. This will be important for referencing in report generation later. The project_name can not be changed - as it used internally when storing data
- **`config_dir`** – A path to the configuration directory, as created using the CLI command `qualipy generate-config`. See the (link here)``config`` section for more information

`add_column`(*column: Column*, *name: Optional[str] = None*, *column_stage_collection_name: Optional[str] = None*) → None

Add a mapping to this project

This is the method to use when adding a column mapping to the project. Once added, it will automatically be executed when running the pipeline.

Parameters

- **`column`** – The column object. Can either be created through the function method or class method.
- **`name`** – This is useful when you don't want to run all mappings at once. Often, you'll do analysis on different subsets of the same dataset. Use name to reference it later on and only execute it for a specific subset.

  This name is also essential if you want to analyze the same column, but in a different subset of the data.

Returns

None

Example 1 - Instantiate a project:

```python
import qualipy as qpy

project = qpy.Project(project_name='stocks', config_dir='/tmp/.config')
```

Example 2 - Instantiate a project and add a column to it:

```python
import qualipy as qpy

project = qpy.Project(project_name='stocks', config_dir='/tmp/.config')
# using the price column defined above
project.add_column(column=price, name='price_analysis')
```

4.4 Supported DataSet Types

Currently, there are three different dataset types supported: Pandas, Spark, and SQL

Pandas

class qualipy.backends.pandas_backend.dataset.**PandasData**(*data: DataFrame*)

PandasData must be instantiated when tracking pandas data

__init__(*data: DataFrame*)

Parameters
data – The pandas dataset that we want to track

set_stratify_rule(*column: str*, *values: Optional[List[str]] = None*) → None

Use this when you want to run all functions on separate stratifications

Currently, only equality based stratification is possible. In the future, comparison based stratifications will be available.

Parameters

- **column** – The name of the column you want to stratify on.
- **values** – If you only want to include a subset of values within `column`, specify them here

Returns
None

Example 1 - Setting symbol as a stratification:

```
from qualipy.backends.pandas_backend.dataset import PandasData

stocks = PandasData(stocks)
stocks.set_stratify_rule("symbol")
```

Example 2 - Setting symbol as a stratification and specifying the subset of stocks to analyze:

```
from qualipy.backends.pandas_backend.dataset import PandasData

stocks = PandasData(stocks)
stocks.set_stratify_rule("symbol", values=['IBM', 'AAPL'])
```

SQL

class qualipy.backends.sql_backend.dataset.**SQLData**(*engine: Optional[Engine] = None*, *table_name: Optional[str] = None*, *schema: Optional[str] = None*, *conn_string: Optional[str] = None*, *custom_select_sql: Optional[str] = None*, *create_temp: bool = False*, *backend='sql'*)

This is used when tracking a relational table

__init__(*engine: Optional[Engine] = None*, *table_name: Optional[str] = None*, *schema: Optional[str] = None*, *conn_string: Optional[str] = None*, *custom_select_sql: Optional[str] = None*, *create_temp: bool = False*, *backend='sql'*)

Parameters

- **engine** – A sqlalchemy engine to the database containing the table we want to track

- **table_name** – The name of the table we want to track
- **schema** – The schema the table is in
- **conn_string** – If engine is None, you can just pass the sqlalchemy database connection
- **custom_select_sql** – Must be proper SQL for whatever DB you are using. This will instantiate a temporary table that Qualipy will run against. This is useful if you dont need the entire table, or need to run any joins before running Qualipy. However, often it might be better to just create a view of what you need.

set_custom_where(*custom_where: str*)

Set this when you want a function to run on a subset of the table

Parameters

custom_where – The where portion of a sql statement. This can then be used in a function. See example in the documentation for more information

Example 1 - Instantiating a table:

```
import sqlalchemy as sa
from qualipy.backends.sql_backend.dataset import SQLData

engine = sa.create_engine('sqlite://')
data = SQLData(engine=engine, table_name='my_table')
```

Example 2 - Instantiating a table and setting a custom where clause:

```
import sqlalchemy as sa
from qualipy.backends.sql_backend.dataset import SQLData

engine = sa.create_engine('sqlite://')
data = SQLData(engine=engine, table_name='my_table')
data.set_custom_where("my_col = 'setosa'")
```

4.5 Qualipy

class qualipy.run.**Qualipy**(*project: Project, backend: str = 'pandas', time_of_run: Optional[datetime] = None, batch_name: Optional[str] = None, overwrite_arguments: Optional[dict] = None*)

This is the main entrypoint to Qualipy. This is the object that will actually execute on your data.

__init__(*project: Project, backend: str = 'pandas', time_of_run: Optional[datetime] = None, batch_name: Optional[str] = None, overwrite_arguments: Optional[dict] = None*)

Parameters

- **project** – Your defined qualipy.Project
- **backend** – Can be either "pandas", "sql", or "spark" depending on what kind of data you are tracking
- **time_of_run** – If None, this will be the current datetime. Note, this is very important for analysis, as time_of_run is essentially your x_axis in all time series analysis. Being able to set it to a specific date can be useful when generating retrospective statistics.

- **batch_name** – Useful for comparing specific time points by name during analysis. By default it will take the time_of_run as batch_name

`set_dataset`(*df*, *columns: Optional[List[str]] = None*, *run_name: Optional[str] = None*) → None

This specified the exact subset of data you want to run on.

Use this method when you don't have all of the data (a live process) and want to only run on one batch of data.

Parameters

- **df** – Can be either PandasData, SQLData, or SparkData
- **columns** – If you don't want to run all mappings on this specific subset of data, you can specify just the columns you want to run. Note - this corresponds to the `name` argument when adding a column to a project
- **run_name** – If you're running metrics from a project on many different subsets any iterations of the data, you might want to give each specific subset a name. This is especially necessary when running aggregates on a column where the column name itself stays the same, but the meaning changes based on the subset. By default, this will take the value of '0'

Returns

None

`set_chunked_dataset`(*df*, *columns: Optional[List[str]] = None*, *run_name: Optional[str] = None*, *time_freq: str = '1D'*, *time_column=None*)

This specified the exact subset of data you want to run on.

Use this method when you already have all data available, and want to retrospectively analyze all historical as if it was a live process. Note - There's nothing stopping you from first running this on the available data and then running on a batch-per-batch basis afterwards using regular `set_dataset`.

Parameters

- **df** – Can be either PandasData, SQLData, or SparkData
- **columns** – If you don't want to run all mappings on this specific subset of data, you can specify just the columns you want to run. Note - this corresponds to the `name` argument when adding a column to a project
- **run_name** – If you're running metrics from a project on many different subsets any iterations of the data, you might want to give each specific subset a name. This is especially necessary when running aggregates on a column where the column name itself stays the same, but the meaning changes based on the subset. By default, this will take the value of '0'
- **time_freq** – A pandas-like timeseries frequency term. Use this page to know what you can use: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases (turn to link)
- **time_column** – The time series column qualipy should use to chunk the data

Returns

None

4.6 Data types

There are several data types one can check for, depending on the backend. For pandas, these include

- *DateTimeType*
- *FloatType* - will match against float16-128
- *IntType* - will match against int0-64
- *NumericTypeType* - will match with any numeric subtype
- *ObjectType*
- *BoolType*

For SQL and SPARK backends, these are generally less important as type is usually enforced by the framework itself, reducing the need for type checking.

CHAPTER

FIVE

CONFIGURATION FILE

The configuration file drives a lot of the reporting and anomaly detection in Qualipy. The default config.json file is created upon running `qualipy generate-config` through the CLI.

Some notes on the configuration:

- how to reference metrics…

The following json has all available keys one could set.

```
{
    # The sqlalchemy like string that tells Qualipy where to store all data.
    # By default this is set to a sqlite file within the config directory
    "QUALIPY_DB": "sqlite:////tmp/.qualipy/qualipy.db"",
    # If using a database like postgres that supports schemas, setting this
    # would place all data in that specific schema
    "SCHEMA": "",
    # the name of the project were configuring. This corresponds to your Qualipy
    # pipeline.
    "example_project": {
        # This is where you specify anomaly specific settings
        "ANOMALY_ARGS": {
            # Each anomaly score corresponds to a standardized value. In general,
            # anything over 1 is considered an anomaly, but this could be used to
            # control the severity of the outliers
            "importance_level": 1.3,
            # This is used to set "rules" for any specific column. See what rules
            # are available to use **here (set this)
            "specific": {
                # to reference an aggregate, use run_name + column_name + metric_name +␣
↪arguments (if any)
                "rows_my_column_count_": {
                    # "increasing" is just an example of a function that checks whether
                    # or not the aggregate is always increasing. This might be useful
                    # when you're inspecting the total size of a database
                    "increasing": {
                        # Can be turned on and off
                        "use": true,
                        # Since this is not a machine learning based approach, you have
                        # to set your own severity level when using custom rules
                        "severity": 3
                    }
                },
```

(continues on next page)

(continued from previous page)

```
            }
        },
        # What anomaly model to use. See the Anomaly Detection guide for different
        # options
        "ANOMALY_MODEL": "prophet",
        # Date format to use on reports
        "DATE_FORMAT": "%Y-%m-%d",
        # Minimum severity level to set for filtering out numerical
        # anomalies on the anomaly report
        "NUM_SEVERITY_LEVEL": 1,
        # Minimum severity level to set for filtering out categorical
        # anomalies on the anomaly report
        "CAT_SEVERITY_LEVEL": 1,
        # Useful for categorizing anomalies based on certain thresholds
        "SEVERITY_LEVELS": {
            "low": 1.5,
            "medium": 2.5,
            "high": 10
        },
        # The following section controls the plots on the anomaly report
        "VISUALIZATION": {
            # Controls the visualizations that are displayed in the anomaly report. There
            # are 5 different categories of data to be displayed. Each one of them has␣
↪their
            # own section

            # Since Qualipy by default gathers raw row counts for each data input, this␣
↪section
            # will show show the overal trend of data size over time
            "row_counts": {

                # Include this if you want to view the counts of the most recent batch.
                "include_bar_of_latest": {
                    "use": true,
                    "diff": true,
                    "show_by_default": true
                },
                # Include this if you want to get a summary overview of the row counts
                "include_summary": {
                    "use": true,
                    "show_by_default": true
                }
            },

            # This section is for viewing all metrics that return a numerical data type,
            # such as float and int
            "trend": {
                "include_bar_of_latest": {
                    "use": true,
                    # You can use this to only include certain metrics
                    "variables": [
                        "measurement_concept_id_measurement_number_of_unique_",
```

(continues on next page)

(continued from previous page)

```
                        "drug_concept_id_drug_number_of_unique_",
                    ],
                    "diff": false,
                    "show_by_default": true
                },
                "include_summary": {
                    "use": true,
                    "show_by_default": true
                },
                # Specify an sst to add a layer to the plot that include_summary
                # change point detection. The value refers to how far to look back
                "sst": 3,
                # Set this to true if each batch should have a point. Note, this
                # can look unappealling with a large number of batches
                "point": true,
                # Set this to include a rolling mean for each trend
                "n_steps": 10,
                # Set this if you want to include a layer in the plot that shows
                # the difference from a previous value
                "add_diff": {
                    # Set this to determine how far to look back
                    "shift": 1
                }
            },
            # Add this to visualize all categorical variables (those returning dicts
            # with counts).
            "proportion": {
            }
            # This section includes analysis on the missingness of the data
            "missing": {
                # By default, it will only show data that contains any actual missing␣
↪data.
                # To also show data without any missingness, set this to True
                "include_0": true,
                "include_bar_of_latest": {
                    "use": true,
                    "diff": false
                }
            },

        },

        # This section is for customizing the metric names and hover-over descriptions,
        # in order to potentially make them more human-readable
        "DISPLAY_NAMES": {
            # This default list is automatically populated by the function name
            # and description from the function definition
            "DEFAULT": {
                "number_of_unique": {
                    "display_name": "number_of_unique_values",
                    "description": "A total count of the number of unique values in the␣
↪batch"
```

(continues on next page)

(continued from previous page)

```
                }
            },
            "CUSTOM": {
                "random_function": {
                    "display_name": "Random Function",
                    "description": "Description of random_function"
                }
            },
        }
    },
}
```

CHAPTER

SIX

CLI USER GUIDE

CHAPTER SEVEN

ANOMALY DETECTION

Qualipy trains a separate anomaly and forecasting model for each aggregate that's generated in your project. Each of these models are stored in the `models` directory of the configuration directory, and can be deployed in any situation. It's generally up to the user on how to schedule and deploy the training and reporting of anomalies, but Qualipy provides the functionalities to configure each model, fine-tune it's sensitivity to outliers, and store all anomalies for a project. It also provides a variety of anomaly reports to understand and traverse through all anomalies in a project.

7.1 How to use

In Qualipy, anomaly models can be controlled through the CLI and configuration file

First, we should cover the different available options in the configuration file to set up our models. Note, it might be useful to first have a general understanding of the configuration layout **here**

The configuration for the anomaly models can be set within a the project specific config. To specify which model we want to deploy, we must first set the `ANOMALY_MODEL` key:

```
...
"my_project": {
    "ANOMALY_MODEL": "prophet"
}
...
```

This means that Qualipy will train a separate `prophet` model for each metric. To further configure our prophet models, we must dive into the `ANOMALY_ARGS`. There are just a few options to set here.

Option	Meaning
importance_level	Sets the minimum required importance for the observations to be anomalous. As a rule of thumb, any value greater than 1 is considered to be anomalous. Any value greater 3 is considered to be very anomalous. By default, it is set to one.
specific	An area to set additional rules on a metric by metric basis. See more information here…

Currently, there are three models implemented within Qualipy: prophet, std, and isolationForest. More info about each of these can be found in the sections below.

7.2 Prophet

7.3 isolationForest

7.4 StandardDeviation

CHAPTER EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX