TUTORIAL

How To Test Your Data With Great Expectations

System Tools Python Git

By Sam Bail
Published on October 15, 2021

The author selected the Diversity in Tech Fund to receive a donation as part of the Write for DOnations program.

Introduction

In this tutorial, you will set up a local deployment of Great Expectations, an open source data validation and documentation library written in Python. Data validation is crucial to ensuring that the data you process in your pipelines is correct and free of any data quality issues that might occur due to errors such as incorrect inputs or transformation bugs. Great Expectations allows you to establish assertions about your data called *Expectations*, and validate any data using those Expectations.

When you're finished, you'll be able to connect Great Expectations to your data, create a suite of Expectations, validate a batch of data using those Expectations, and generate a data quality report with the results of your validation.

Prerequisites

To complete this tutorial, you will need:

- A local development environment for Python 3.6 or above. You can follow the tutorial for your operating system in this series: How To Install and Set Up a Local Programming Environment for Python 3. It is recommended to use a local programming environment to avoid problems with connecting to the browser.
- Some familiarity with Python. You can look at the How to code in Python eBook to learn more.
- Some familiarity with Jupyter notebooks.
- A working installation of Git.
- A web browser like Firefox or Chrome.

Step 1 — Installing Great Expectations and Initializing a Great Expectations Project

In this step, you will install the Great Expectations package in your local Python environment, download the sample data you'll use in this tutorial, and initialize a Great Expectations project.

To begin, open a terminal and make sure to activate your virtual Python environment. Install the Great Expectations Python package and command-line tool (CLI) with the following command:

CONTENTS

```
$ pip install great_expectations==0.13.35
```

Note: This tutorial was developed for Great Expectations version 0.13.35 and may not be applicable to other versions.

In order to have access to the example data repository, run the following git command to clone the directory and change into it as your working directory:

```
$ git clone https://github.com/do-community/great_expectations_tutorial
$ cd great_expectations_tutorial
```

The repository only contains one folder called `data`, which contains two example CSV files with data that you will use in this tutorial. Take a look at the contents of the `data` directory:

```
$ ls data
```

You'll see the following output:

```
Output
yellow_tripdata_sample_2019-01.csv      yellow_tripdata_sample_2019-02.csv
```

Great Expectations works with many different types of data, such as connections to relational databases, Spark dataframes, and various file formats. For the purpose of this tutorial, you will use these CSV files containing a small set of taxi ride data to get started.

Finally, initialize your directory as a Great Expectations project by running the following command. Make sure to use the `--v3-api` flag, as this will switch you to using the most recent API of the package:

```
$ great_expectations --v3-api init
```

When asked `OK to proceed? [Y/n]:`, press `ENTER` to proceed.

This will create a folder called `great_expectations`, which contains the basic configuration for your Great Expectations project, also called the *Data Context*. You can inspect the contents of the folder:

```
$ ls great_expectations
```

You will see the first level of files and subdirectories that were created inside the `great_expectations` folder:

```
Output
checkpoints             great_expectations.yml  plugins
expectations            notebooks               uncommitted
```

The folders store all the relevant content for your Great Expectations setup. The `great_expectations.yml` file contains all important configuration information. Feel free to explore the folders and configuration file a little more before moving on to the next step in the tutorial.

In the next step, you will add a Datasource to point Great Expectations at your data.

Step 2 — Adding a Datasource

In this step, you will configure a Datasource in Great Expectations, which allows you to automatically create data assertions called *Expectations* as well as validate data with the tool.

While in your project directory, run the following command:

```
$ great_expectations --v3-api datasource new
```

You will see the following output. Enter the options shown when prompted to configure a file-based Datasource for the `data` directory:

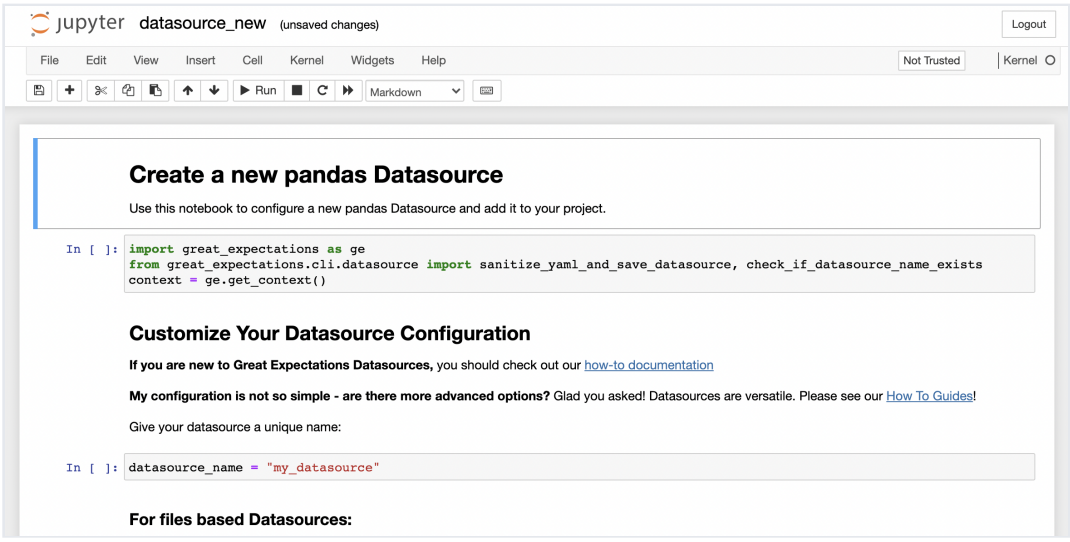
```
Output
What data would you like Great Expectations to connect to?
    1. Files on a filesystem (for processing with Pandas or Spark)
```

```
    1. Files on a filesystem (for processing with Pandas or Spark)
    2. Relational database (SQL)
: 1

What are you processing your files with?
    1. Pandas
    2. PySpark
: 1

Enter the path of the root directory where the data files are stored. If files are on local disk
: data
```

After confirming the directory path with `ENTER`, Great Expectations will open a *Jupyter notebook* in your web browser, which allows you to complete the configuration of the Datasource and store it to your Data Context. The following screenshot shows the first few cells of the notebook:



The notebook contains several pre-populated cells of Python code to configure your Datasource. You can modify the settings for the Datasource, such as the name, if you like. However, for the purpose of this tutorial, you'll leave everything as-is and execute all cells using the `Cell > Run All` menu option. If run successfully, the last cell output will look as follows:

```
Output
[{'data_connectors': {'default_inferred_data_connector_name': {'module_name': 'great_expectation
    'base_directory': '../data',
    'class_name': 'InferredAssetFilesystemDataConnector',
    'default_regex': {'group_names': ['data_asset_name'], 'pattern': '(.*)'}},
   'default_runtime_data_connector_name': {'module_name': 'great_expectations.datasource.data_co
    'class_name': 'RuntimeDataConnector',
    'batch_identifiers': ['default_identifier_name']}},
  'module_name': 'great_expectations.datasource',
  'class_name': 'Datasource',
  'execution_engine': {'module_name': 'great_expectations.execution_engine',
   'class_name': 'PandasExecutionEngine'},
  'name': 'my_datasource'}]
```

This shows that you have added a new Datasource called `my_datasource` to your Data Context. Feel free to read through the instructions in the notebook to learn more about the different configuration options before moving on to the next step.

> **Warning:** Before moving forward, close the browser tab with the notebook, return to your terminal, and press `CTRL+C` to shut down the running notebook server before proceeding.

You have now successfully set up a Datasource that points at the `data` directory, which will allow you to access the CSV files in the directory through Great Expectations. In the next step, you will use one of these CSV files in your Datasource to automatically generate Expectations with a profiler.

Step 3 — Creating an Expectation Suite With an Automated Profiler

In this step of the tutorial, you will use the built-in Profiler to create a set of Expectations based on some existing data. For this purpose, let's take a closer look at the sample data that you

downloaded:

> The files `yellow_tripdata_sample_2019-01.csv` and `yellow_tripdata_sample_2019-02.csv` contain taxi ride data from January and February 2019, respectively.
> This tutorial assumes that you know the January data is correct, and that you want to ensure that any subsequent data files match the January data in terms of number or rows, columns, and the distributions of certain column values.

For this purpose, you will create Expectations (data assertions) based on certain properties of the January data and then, in a later step, use those Expectations to validate the February data. Let's get started by creating an Expectation Suite, which is a set of Expectations that are grouped together:

```
$ great_expectations --v3-api suite new
```

By selecting the options shown in the output below, you specify that you would like to use a profiler to generate Expectations automatically, using the `yellow_tripdata_sample_2019-01.csv` data file as an input. Enter the name `my_suite` as the Expectation Suite name when prompted and press `ENTER` at the end when asked `Would you like to proceed? [Y/n]`:

```
Output
Using v3 (Batch Request) API
How would you like to create your Expectation Suite?
    1. Manually, without interacting with a sample batch of data (default)
    2. Interactively, with a sample batch of data
    3. Automatically, using a profiler
: 3

A batch of data is required to edit the suite - let's help you to specify it.

Which data asset (accessible by data connector "my_datasource_example_data_connector") would you
    1. yellow_tripdata_sample_2019-01.csv
    2. yellow_tripdata_sample_2019-02.csv
: 1

Name the new Expectation Suite [yellow_tripdata_sample_2019-01.csv.warning]: my_suite

When you run this notebook, Great Expectations will store these expectations in a new Expectatio

  <path_to_project>/great_expectations_tutorial/great_expectations/expectations/my_suite.json

Would you like to proceed? [Y/n]: <press ENTER>
```

This will open another Jupyter notebook that lets you complete the configuration of your Expectation Suite. The notebook contains a fair amount of code to configure the built-in profiler, which looks at the CSV file you selected and creates certain types of Expectations for each column in the file based on what it finds in the data.

Scroll down to the second code cell in the notebook, which contains a list of `ignored_columns`. By default, the profiler will ignore all columns, so let's comment out some of them to make sure the profiler creates Expectations for them. Modify the code so it looks like this:

```
ignored_columns = [
#     "vendor_id"
# ,    "pickup_datetime"
# ,    "dropoff_datetime"
# ,    "passenger_count"
    "trip_distance"
,    "rate_code_id"
,    "store_and_fwd_flag"
,    "pickup_location_id"
,    "dropoff_location_id"
,    "payment_type"
,    "fare_amount"
,    "extra"
,    "mta_tax"
,    "tip_amount"
,    "tolls_amount"
,    "improvement_surcharge"
,    "total_amount"
,    "congestion_surcharge"
,]
```

Make sure to remove the comma before `"trip_distance"`. By commenting out the columns

Make sure to remove the comma before `trip_distance`. By commenting out the columns `vendor_id`, `pickup_datetime`, `dropoff_datetime`, and `passenger_count`, you are telling the profiler to generate Expectations for those columns. In addition, the profiler will also generate *table-level Expectations*, such as the number and names of columns in your data, and the number of rows. Once again, execute all cells in the notebook by using the `Cell > Run All` menu option.

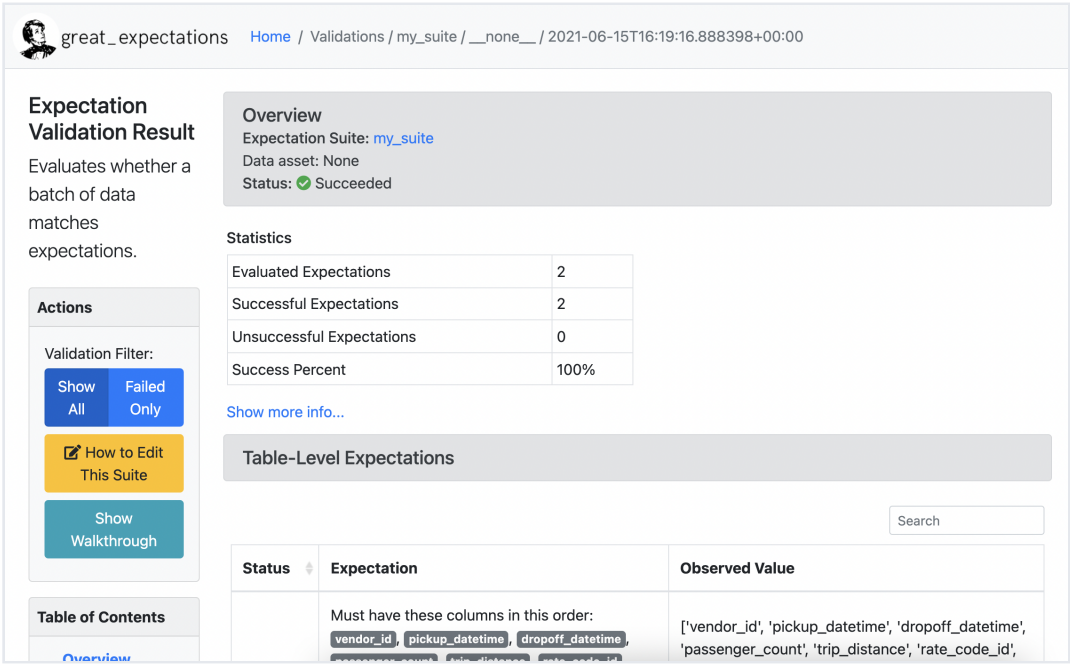
When executing all cells in this notebook, two things happen:

> The code creates an Expectation Suite using the automated profiler and the `yellow_tripdata_sample_2019-01.csv` file you told it to use.
> The last cell in the notebook is also configured to run validation and open a new browser window with *Data Docs*, which is a data quality report.

In the next step, you will take a closer look at the Data Docs that were opened in the new browser window.

Step 4 — Exploring Data Docs

In this step of the tutorial, you will inspect the Data Docs that Great Expectations generated and learn how to interpret the different pieces of information. Go to the browser window that just opened and take a look at the page, shown in the screenshot below.



At the top of the page, you will see a box titled **Overview**, which contains some information about the validation you just ran using your newly created Expectation Suite `my_suite`. It will tell you `Status: Succeeded` and show some basic statistics about how many Expectations were run. If you scroll further down, you will see a section titled **Table-Level Expectations**. It contains two rows of Expectations, showing the Status, Expectation, and Observed Value for each row. Below the table Expectations, you will see the column-level Expectations for each of the columns you commented out in the notebook.

Let's focus on one specific Expectation: The `passenger_count` column has an Expectation stating "values must belong to this set: `1 2 3 4 5 6`." which is marked with a green checkmark and has an Observed Value of "0% unexpected". This is telling you that the profiler looked at the values in the `passenger_count` column in the January CSV file and detected only the values 1 through 6, meaning that all taxi rides had between 1 and 6 passengers. Great Expectations then created an Expectation for this fact. The last cell in the notebook then triggered validation of the January CSV file and it found no unexpected values. This is spuriously true, since the same data that was used to create the Expectation was also the data used for validation.

In this step, you reviewed the Data Docs and observed the `passenger_count` column for its Expectation. In the next step, you'll see how you can validate a different batch of data.

Step 5 — Creating a Checkpoint and Running Validation

In the final step of this tutorial, you will create a new Checkpoint, which bundles an Expectation Suite and a batch of data to execute validation of that data. After creating the Checkpoint, you will then run it to validate the February taxi data CSV file and see whether the file passed the

Expectations you previously created. To begin, return to your terminal and stop the Jupyter notebook by pressing `CTRL+C` if it is still running. The following command will start the workflow to create a new Checkpoint called `my_checkpoint`:

```
$ great_expectations --v3-api checkpoint new my_checkpoint
```

This will open a Jupyter notebook with some pre-populated code to configure the Checkpoint. The second code cell in the notebook will have a random `data_asset_name` pre-populated from your existing Datasource, which will be one of the two CSV files in the `data` directory you've seen earlier. Ensure that the `data_asset_name` is `yellow_tripdata_sample_2019-02.csv` and modify the code if needed to use the correct filename.

```
my_checkpoint_name = "my_checkpoint" # This was populated from your CLI command.

yaml_config = f"""
name: {my_checkpoint_name}
config_version: 1.0
class_name: SimpleCheckpoint
run_name_template: "%Y%m%d-%H%M%S-my-run-name-template"
validations:
  - batch_request:
      datasource_name: my_datasource
      data_connector_name: default_inferred_data_connector_name
      data_asset_name: yellow_tripdata_sample_2019-02.csv
      data_connector_query:
        index: -1
    expectation_suite_name: my_suite
"""
print(yaml_config)
"""
```

This configuration snippet configures a new Checkpoint, which reads the data asset `yellow_tripdata_sample_2019-02.csv`, i.e., your February CSV file, and validates it using the Expectation Suite `my_suite`. Confirm that you modified the code correctly, then execute all cells in the notebook. This will save the new Checkpoint to your Data Context.

Finally, in order to run this new Checkpoint and validate the February data, scroll down to the last cell in the notebook. Uncomment the code in the cell to look as follows:

```
context.run_checkpoint(checkpoint_name=my_checkpoint_name)
context.open_data_docs()
```

Select the cell and run it using the `Cell > Run Cells` menu option or the `SHIFT+ENTER` keyboard shortcut. This will open Data Docs in a new browser tab.

On the Validation Results overview page, click on the topmost run to navigate to the Validation Result details page. The Validation Result details page will look very similar to the page you saw in the previous step, but it will now show that the Expectation Suite failed, validating the new CSV file. Scroll through the page to see which Expectations have a red X next to them, marking them as failed.

Find the Expectation on the `passenger_count` column you looked at in the previous step: "values must belong to this set: `1 2 3 4 5 6`". You will notice that it now shows up as failed and highlights that `1579 unexpected values found. ≈15.79% of 10000 total rows`. The row also displays a sample of the unexpected values that were found in the column, namely the value `0`. This means that the February taxi ride data suddenly introduced the unexpected value `0` as in the `passenger_counts` column, which seems like a potential data bug. By running the Checkpoint, you validated the new data with your Expectation Suite and detected this issue.

Note that each time you execute the `run_checkpoint` method in the last notebook cell, you kick off another validation run. In a production data pipeline environment, you would call the `run_checkpoint` command outside of a notebook whenever you're processing a new batch of data to ensure that the new data passes all validations.

Conclusion

In this article you created a first local deployment of the Great Expectations framework for data validation. You initialized a Great Expectations Data Context, created a new file-based Datasource, and automatically generated an Expectation Suite using the built-in profiler. You then created a Checkpoint to run validation against a new batch of data, and inspected the Data Docs to view the validation results.

This tutorial only taught you the basics of Great Expectations. The package contains more options for configuring Datasources to connect to other types of data, for example relational databases. It also comes with a powerful mechanism to automatically recognize new batches of data based on pattern-matching in the tablename or filename, which allows you to only configure a Checkpoint once to validate any future data inputs. You can learn more about Great Expectations in the official documentation.

About the authors

Sam Bail

Data professional based in NYC

Still looking for an answer?

Ask a question

Search for more help

RELATED

How To Set Up a Masterless Puppet Environment on Ubuntu 14.04

Tutorial

How to Use Reprepro for a Secure Package Repository on Ubuntu 14.04

Tutorial

Comments

Leave a comment

Leave a comment ...

Sign in to Comment

GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.

HOLLIE'S HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

Featured on Community Kubernetes Course Learn Python 3 Machine Learning in Python Getting started with Go Intro to Kubernetes

DigitalOcean Products Virtual Machines Managed Databases Managed Kubernetes Block Storage Object Storage Marketplace VPC Load Balancers