Setting your data expectations - Data profiling and testing with the Great Expectations library & Databricks

Rich Louden

2020-01-20

## Introduction - Why Data Quality matters and the Great Expectations library

Over the last decade or so companies have been striving to make better use of their data. The use cases for such projects have generally fallen under two categories, improve operational efficiency or drive customer sales / behaviour. However, in order to utilise this data it must first be piped from source systems (CRM, ordering, POS etc) into somewhere with greater redundancy. In addition to this simpler goal, it must also be manipulated into a format that is acceptable to the people analysing said data!

The pipelines that complete these operations are often complex, incorporating numerous source systems with different schemas, update times etc. This leads to the development of numerous functions that work within an ETL flow, scheduled by a tool such as [Prefect](https://docs.prefect.io/core/) or [Airflow](https://airflow.Apache.org/).

So with the crucial nature of this data in mind, how do we ensure that what gets pulled though our flow is going to be of use to those at the end of the pipeline? That something hasn’t been misentered or corrupted in the source systems? Well in most software the role of unit / integration testing would help, however if your unit test expects a data frame and to return a dataframe, as a simple example, this may pass whilst the data within said dataframe is riddled with NULL values and bad quality data. As such, a good addition to more standard testing is to actually test what data those source systems are providing, such as in the example below, which is where a fairly recent library, [Great Expectations](https://github.com/great-expectations) (GE), can be a real help! (Though as is usually the case, other DQ testing libraries exist that can be explored, such as [deequ](https://github.com/awslabs/deequ) and [bulwark](https://github.com/ZaxR/bulwark))

To summarise the library, GE works in addition to unit / integration testing by profiling data sources in order to build a set of “expectations” around each column based on the type, which can then be pruned / added to by the user. For example, on the dataframe shown below we have and ID column that you may expect to never be NULL, an Animal column that should always be a string and a cost that should always be a float.

import pandas as pd  
  
df = pd.DataFrame(zip([1, 2, 3, 4, 5], ["Cat", "Dog", "Cat", "Rabbit", "Dog"], [70, 65.00, 120.50, 20.75, 1000.00]), columns = ["ID", "Animal", "Treatment\_Cost"])  
  
print(df)

## ID Animal Treatment\_Cost  
## 0 1 Cat 70.00  
## 1 2 Dog 65.00  
## 2 3 Cat 120.50  
## 3 4 Rabbit 20.75  
## 4 5 Dog 1000.00

An example flow of a GE pipeline would involve initial profiling of data, adaption of the produced expectations and continual validation of new data against said expectations. In addition to a better understanding of the source and processed data, this process can help visualise it though auto generated html, which can be deployed as a static website. This nice addition allows the user to deploy a very quick data quality dashboard, an example of which from the package’s docs is shown below.

## Getting started with GE on databricks

As eluded to in the title of this post, I’ve been utilising GE on the well known Spark platform Databricks, as we’ve been using this platform with a number of clients in order to do distributed computation of large datasets. However, whilst this post is based around Spark, GE can work with other datatypes such as CSVs (via Pandas Dataframes) and Relational Databases (via SQL Alchemy), as shown in their docs [here](https://docs.greatexpectations.io/en/latest/getting_started/typical_workflow.html).

So, in order to start using GE in DataBricks you must first follow the initialisation [instructions](https://docs.greatexpectations.io/en/latest/getting_started/cli_init.html) on your local machine, after downloading from PyPy both locally and on your Databricks cluster. During set up choose option 1 regarding data sources and then 2 for pyspark, which will give you an error unless you have pyspark installed locally, however this doesnt matter. If you now check the directory you initiated within youll find a great\_expectations folder and a great\_expectations.yml file which you can open and edit. In this file add the following code, so that it matches the image shown below.

datasources:  
 spark:  
 class\_name: SparkDFDatasource  
 generators:  
 passthrough:  
 class\_name: DatabricksTableBatchKwargsGenerator

Once this is saved you can copy the local files over into the DataBricks File System (DBFS), using the DataBricks CLI, as shown below and documented [here](https://docs.databricks.com/dev-tools/cli/dbfs-cli.html). This involves first making a directory in the DBFS and then copying over the files.

dbfs mkdirs your\_ge\_dir  
  
#Note this \ is for Windows, change to / for linux  
dbfs cp -r .\your\_local\_folder dbfs:/your\_ge\_dir

Once this is done you can check the files have copied by opening a notebook, and using the %fs cell magic to check the contents of your dbfs, using the below code.

%fs  
  
ls your\_new\_dir/

Now that you have the initilisation set up and the library on your cluster you can set your GE context (which tells the profiler what kind of data to expect) and set your initial expectations, which I’ll document in two different ways, manual profiling and full profiling, the latter involving the building of the above mentioned data docs.

### Setting your expectations - Setup

To do either of the above mentioned methods there is a standard set up, which is shown below. This sets up your data context and builds a list of your data assets in a specific database.

#Loading the package and any additional elements, such as the basic profiler  
  
import great\_expectations as ge  
from great\_expectations.datasource.generator.databricks\_generator import DatabricksTableBatchKwargsGenerator  
from great\_expectations.datasource.sparkdf\_datasource import SparkDFDatasource  
from great\_expectations.dataset.sparkdf\_dataset import SparkDFDataset  
from great\_expectations.profile.basic\_dataset\_profiler import BasicDatasetProfiler  
import json  
  
DATABASE = "database"  
  
context = ge.data\_context.DataContext("/dbfs/your\_ge\_dir/great\_expectations/")  
  
Datasource = SparkDFDatasource()  
  
generator = DatabricksTableGenerator(name = 'default', datasource = Datasource, database = DATABASE)  
  
data\_asset\_tuples = test.get\_available\_data\_asset\_names()["names"]  
  
data\_assets = [i[0] for i in data\_asset\_tuples]

### Method 1: Manual Profiling

The first step to manual profiling is to load the required data, convert this to a GE recognisable spark dataframe object and then create an empty expectations file.

data = spark.table(database + "." + data\_assets\_paths[0])  
  
spark\_data = SparkDFDataset(data)  
  
data\_asset\_name = data\_assets\_paths[0]  
  
context.create\_expectation\_suite(data\_asset\_name, "Test")

Once you have your empty file you can manually add in the relevant expectations for the desired columns, and profile the batch of data via the validation operator and save out the new expectation suite (in this scenario also saving any failed expectations).

spark\_data.expect\_column\_values\_to\_not\_be\_null("PROJECT\_ID")  
spark\_data.expect\_column\_values\_to\_be\_in\_set("DELIVERY\_GROUP\_LATEST", set(['IP Legacy', 'IP AM SP&C']))  
  
spark\_data.validate()  
  
context.save\_expectation\_suite(discard\_failed\_expectations = False)

### Method 2: Full Profiling

As opposed to manual profiling, full profiling utilises the inbuilt data profiler that is packaged with GE in order to build a full expectation suite based on the column types, and then validate the data against all of these expectations. A workflow for such a method might be to profile all the tables within a database, maintain these expectations and then validate any updates from the raw data that is appended to these tables. This example is what is shown below, which utilises a couple of custom functions which can be applied to each table within the data asset list created during set up.

# First build the functions to produce the expectations for the database and also  
# to run the validations   
  
def build\_expectations(database, asset\_name, context):  
   
 exp\_suite = database + "." + asset\_name + "." + str(datetime.today().strftime("%d-%m-%Y")) + "\_expectations"  
   
 data = spark.table(database + "." + asset\_name)  
   
 spark\_data = SparkDFDataset(data)  
   
 context.create\_expectation\_suite(exp\_suite, overwrite\_existing = True)  
   
 profile = spark\_data.profile(BasicDatasetProfiler)  
   
 context.save\_expectation\_suite(profile)  
   
 sqlContext.uncacheTable(database + "." + asset\_name)  
   
   
def validate\_expectations(database, asset\_name, context):  
   
 val\_suite = database + "." + asset\_name + "." + str(datetime.today().strftime("%d-%m-%Y")) + "\_validation.json"  
   
 data = spark.table(database + "." + asset\_name)  
   
 spark\_data = SparkDFDataset(data)  
   
 validation = spark\_data.validate()  
   
 val\_json = json.loads(str(test))   
   
 with open(f"/dbfs/getest/expectations{val\_suite}", 'w') as f:  
 json.dump(val\_json, f)  
   
 sqlContext.uncacheTable(database + "." + asset\_name)

# Create the expectations and then validate, storing the results of the validation and updating the data docs  
  
  
  
for i, j in zip(data\_assets, data\_assets\_paths):  
 build\_expectations(DATABASE, i, context)  
  
  
for i, j in zip():  
 validate\_expectations(DATABASE, i, context)

The first function should produce an expectations file for each table, which can then be validated against to build a new data docs page. Examples of the json outputs from both functions are shown below.

## Wrap up

So, hopefully this post has shown you a potential new way to more simply test the quality of what’s moving within your data pipeline, rather than just the functions that make it up! In addition to the enhanced testing that GE provides, I believe that the incorporation of the static Data Docs data quality pages could be a real help to organisations looking to quickly understand the quality of their data!