CONTENTS

Translab Technologies

Translab Technologies is the premier digital transformation partner for organizations looking to embrace the future. We offer a range of IT services that help organization innovate faster, develop new revenue streams and expand to new markets.
We provide a gamut of IT services that range from cloud to DevOps, from database to analytics, from performance tuning to managed services. We have been associated with more than 150 transformational journeys for multinational enterprises and upcoming start-ups across geographies and industries.

Translab Technologies enables Digital Transformation for enterprises worldwide by providing seamless customer experience, business agility and actionable insights. Utilizing disruptive technologies, like Big Data, Analytics, Internet of Things, Automation, Mobility and Cloud, we offer solutions that have applications across industry sectors.
Mission of the company is to help organizations unlock their potential and shape their future together and the Vision of the company is to give organizations the power to maximize their potential and go from disrupted to disruptor. Translab Technologies Private Limited is an ISO certified Organization.

2. Data Validation Platform

We are designing the platform that validates the data using great expectation library and create reports after validating the data.

2.1 Data Validation:

Data validation is the practice of checking the integrity, accuracy and structure of data before it is used for a business operation. Data validation operation results can provide data used for data analytics, business intelligence or training a machine learning model. It can also be used to ensure the integrity of data for financial accounting or regulatory compliance.

Data can be examined as part of a validation process in a variety of ways, including data type, constraint, structured, consistency and code validation. Each type of data validation is designed to make sure the data meets the requirements to be useful.

Data validation is related to data quality. Data validation can be a component to measure data quality, which ensures that a given data set is supplied with information sources that are of the highest quality, authoritative and accurate.

Data validation is also used as part of application workflows, including spell checking and rules for strong password creation.

2.2 Why Data Validation is important

Data validation is also important for data to be useful for an organization or for a specific application operation. For example, if data is not in the right format to be consumed by a system, then the data can't be used easily, if at all.

As data moves from one location to another, different needs for the data arise based on the context for how the data is being used. Data validation ensures that

the data is correct for specific contexts. The right type of data validation makes the data useful.

By removing data mistakes from every project to guarantee that the data is not corrupted, data validation ensures that the dataset is accurate, clean, and comprehensive.

Extraction, transformation, and loading, or ETL, is the process that involves moving a source database to a target data warehouse. To increase the value of the data warehouse and the information housed within, data validation must be done.

2.3 Impact of Data Validation

Inaccurate and incomplete data may lead the end-users to lose trust in data. Data validation is essentially a part of the **ETL process** (Extract, Transform, and Load), which involves moving the source database to the target data warehouse. In doing so, performing data validation is required to enhance the value of the data warehouse and the information stored there.

Various data validation testing tools, such as Grafana, MySql, InfluxDB, and Prometheus, are available for data validation.

As Data Validation goes step-by-step:

1. **Analyzing data:**

It is crucial to analyze data to understand the business requirements. This step includes identifying the data analysis technique required, understanding the dataset, performing data analysis, and processing the results from the data analysis techniques.

2. Validating database:

This step ensures that the available data in the database is relevant. It compares the source and target based on the data field.

3. Sampling:

Sampling involves testing the process on sample data instead of complete data to check if the data is correct or not. Only if the sampled data is accurate validation is performed on the entire dataset. This step saves time as well as resources.

4. Comparison:

This step handles insufficient data correctly and compares the output result to match the expected results

3. Data Validation Libraries

3.1 Soda Core Library:

1.Soda Core is an open-source tool that provides the necessary capabilities to ensure data validation. And even though the tool itself is also written in Python (similarly to Great Expectations), it tackles data validation in a different way: As a developer, you're simply expected to deliver a set of YAML configuration files that tell Soda how to connect to your data warehouse and what checks you want to run on your different tables.

2. This approach scales very conveniently when managing hundreds of tables with different owners and maintainers. Initially, Soda required one YAML file per table, but now with the release of SodaCL you can leverage loops and custom Soda syntax within the YAML configuration to optimize how you define the metrics/checks.

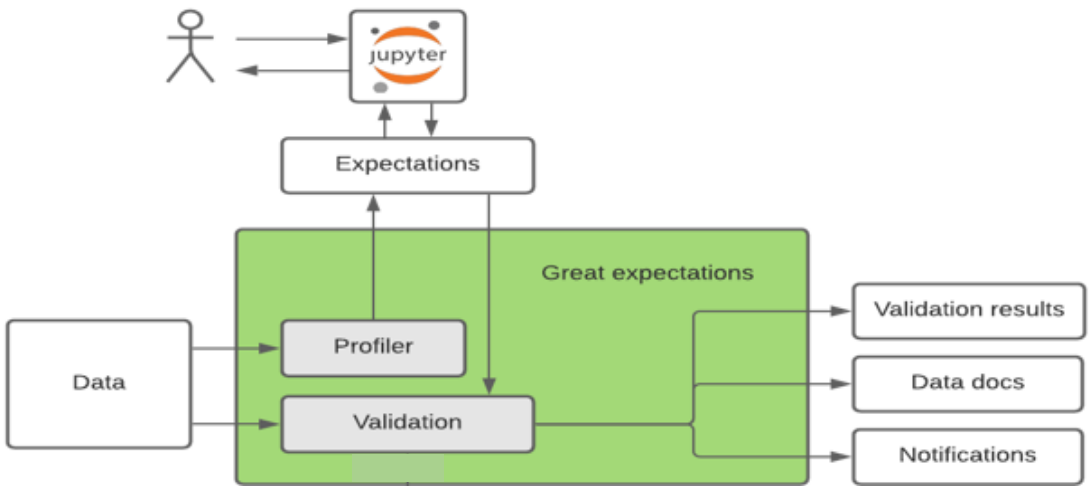
3.Soda mainly prioritizes CLI interactions to run the checks (with a wide range of commands and options), but it also offers a rich Python library — which opens the door to custom usage and leveraging the output of its checks directly within a Python application.

3.2 Great Expectation library:

Great Expectation is arguably the tool that defined the current standard of what should be expected from a data validation tool: You define your checks (or expectations) and how/when you want to run them, and then your data validation component takes care of the rest. Pretty neat.

In the past four years, the tool expanded in every way possible: an integrations list that's getting longer and longer, data profiling capabilities, and built-in data documentation. And the cherry on top is that Great Expectation is a Python library, and your expectations are simply Python functions.

Through quality assurance checks, documentation, and profiling, Great Expectations aids data teams in developing a shared knowledge of their data.



3.3 Types of validations

1.Custom Expectation : Custom Expectations are extensions to the core functionality of Great Expectations. Many Custom Expectations may be fit for a very specific purpose, or be at lower levels of stability and feature maturity than the core library.

2. Native Validation:

3.4 Installation:

Installing PySpark and the Great Expectations Library comes first.

!pip install pyspark

!pip install great_expectations

Native Validation of data:

We validate the email column, as well as other columns, to see if they include null values or unique values, and to see if they contain a specific datatype.

```
df.expect_column_values_to_be_of_type('age','int')
```


```
{
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "result": {
    "observed_value": "int64"
  },
  "success": false
}
```

```
df.expect_column_values_to_be_of_type('name','str')
```


```
{
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "result": {
    "element_count": 100,
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0,
    "partial_unexpected_list": []
  },
  "success": true
}
```


Categorical columns

```
df.expect_column_values_to_be_in_set(
    column = "gender",
    value_set = ["Male","Female"]
)
```


```
{
  "success": true,
  "result": {
    "element_count": 100,
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0,
    "partial_unexpected_list": []
  },
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "meta": {}
}
```

Custom Validation

We performed custom email column validation.

```
# Databricks notebook source
import json
import re  # regular expressions

# !!! This giant block of imports should be something simpler, such as:
from great_expectations import *
from great_expectations.execution_engine import (
    PandasExecutionEngine,
    SparkDFExecutionEngine,
    SqlAlchemyExecutionEngine,
)
from great_expectations.expectations.expectation import (
ColumnMapExpectation,
    Expectation,
    ExpectationConfiguration,
)
from great_expectations.expectations.metrics import (
    ColumnMapMetricProvider,
    column_condition_partial,
)
from great_expectations.expectations.registry import (
    _registered_expectations,
    _registered_metrics,
```

```
    _registered_renderers,
)
from great_expectations.expectations.util import render_evaluation_parameter_string
from great_expectations.render.renderer.renderer import renderer
from great_expectations.render.types import RenderedStringTemplateContent
from great_expectations.render.util import num_to_str, substitute_none_for_missing
from great_expectations.validator.validator import Validator

EMAIL_REGEX = r"[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,7}$"


class ColumnValuesContainValidEmail(ColumnMapMetricProvider):
    # This is the id string that will be used to reference your metric.
    condition_metric_name = "column_values.valid_email"
    condition_value_keys = ()

    # This method defines the business logic for evaluating your metric when using a PandasExecutionEngine
    @column_condition_partial(engine=PandasExecutionEngine)
    def _pandas(cls, column, **kwargs):
        def matches_email_regex(x):
            if re.match(EMAIL_REGEX, str(x)):
                return True
            return False

        return column.apply(lambda x: matches_email_regex(x) if x else False)
```

```
    # This method defines the business logic for evaluating your metric when using a SqlAlchemyExecutionEngine
    #     @column_condition_partial(engine=SqlAlchemyExecutionEngine)
    #     def _sqlalchemy(cls, column, _dialect, **kwargs):
    #         return column.in_([3])

    # This method defines the business logic for evaluating your metric when using a SparkDFExecutionEngine
    @column_condition_partial(engine=SparkDFExecutionEngine)
    def _spark(cls, column, **kwargs):
        return column.rlike(EMAIL_REGEX)


# This class defines the Expectation itself
# The main business logic for calculation lives here.
class ExpectColumnValuesToContainValidEmail(ColumnMapExpectation):
    # These examples will be shown in the public gallery, and also executed as unit tests for your Expectation
    examples = [
        {
            "data": {
                "fail_case_1": ["a123@something", "a123@something.", "a123."],
                "fail_case_2": ["aaaa.a123.co", "aaaa.a123.", "aaaa.a123.com"],
                "fail_case_3": ["aaaa@a123.e", "aaaa@a123.a", "aaaa@a123.d"],
                "fail_case_4": ["@a123.com", "@a123.io", "@a123.eu"],
                "pass_case_1": [
                    "a123@something.com",
                    "vinod.km@something.au",
```

```
      "this@better.work",
    ],
    "pass_case_2": [
      "example@website.dom",
      "ex.ample@example.ex",
      "great@expectations.email",
    ],
    "valid_emails": [
      "Janedoe@company.org",
      "someone123@stuff.net",
      "mycompany@mycompany.com",
    ],
    "bad_emails": ["Hello, world!", "Sophia", "this should fail"],
  },
  "tests": [
    {
      "title": "negative_test_for_no_domain_name",
      "exact_match_out": False,
      "include_in_gallery": True,
      "in": {"column": "fail_case_1"},
      "out": {
        "success": False,
        "unexpected_index_list": [0, 1, 2],
        "unexpected_list": [
          "a123@something",
          "a123@something.",
          "a123.",
```

```
        ],
      },
    },
    {
      "title": "negative_test_for_no_at_symbol",
      "exact_match_out": False,
      "include_in_gallery": True,
      "in": {"column": "fail_case_2"},
      "out": {
        "success": False,
        "unexpected_index_list": [0, 1, 2],
        "unexpected_list": [
          "aaaa.a123.co",
          "aaaa.a123.",
          "aaaa.a123.com",
        ],
      },
    },
    {
      "title": "negative_test_for_ending_with_one_character",
      "exact_match_out": False,
      "include_in_gallery": True,
      "in": {"column": "fail_case_3"},
      "out": {
        "success": False,
        "unexpected_index_list": [0, 1, 2],
        "unexpected_list": [
```

```
            "aaaa@a123.e",
            "aaaa@a123.a",
            "aaaa@a123.d",
        ],
    },
},
{
    "title": "negative_test_for_emails_with_no_leading_string",
    "exact_match_out": False,
    "include_in_gallery": True,
    "in": {"column": "fail_case_4"},
    "out": {
        "success": False,
        "unexpected_index_list": [0, 1, 2],
        "unexpected_list": [
            "aaaa@a123.e",
            "aaaa@a123.a",
            "aaaa@a123.d",
        ],
    },
},
{
    "title": "pass_test",
    "exact_match_out": False,
    "include_in_gallery": True,
    "in": {"column": "pass_case_1"},
    "out": {
```

```
      "success": True,
      "unexpected_index_list": [],
      "unexpected_list": [],
    },
  },
  {
    "title": "pass_test",
    "exact_match_out": False,
    "include_in_gallery": True,
    "in": {"column": "pass_case_2"},
    "out": {
      "success": True,
      "unexpected_index_list": [],
      "unexpected_list": [],
    },
  },
  {
    "title": "valid_emails",
    "exact_match_out": False,
    "include_in_gallery": True,
    "in": {"column": "valid_emails"},
    "out": {
      "success": True,
      "unexpected_index_list": [],
      "unexpected_list": [],
    },
  },
```

```
        {
          "title": "invalid_emails",
          "exact_match_out": False,
          "include_in_gallery": True,
          "in": {"column": "bad_emails"},
          "out": {
            "success": False,
            "unexpected_index_list": [0, 1, 2],
            "unexpected_list": [
              "Hello, world!",
              "Sophia",
              "this should fail",
            ],
          },
        },
      ],
    }
]

# This dictionary contains metadata for display in the public gallery
library_metadata = {
    "maturity": "experimental",
    "tags": ["experimental", "column map expectation"],
    "contributors": [  # Github
      "@aworld1",
      "@enagola",
      "@spencerhardwick",
```

```
        "@vinodkri1",
        "@degulati",
        "@ljohnston931",
        "@rexboyce",
        "@lodeous",
        "@sophiarawlings",
        "@vtdangg",
    ],
}


# This is the id string of the Metric used by this Expectation.
# For most Expectations, it will be the same as the `condition_metric_name` defined in your Metric class above.
map_metric = "column_values.valid_email"


# This is a list of parameter names that can affect whether the Expectation evaluates to True or False
# Please see {some doc} for more information about domain and success keys, and other arguments to Expectations
success_keys = ()


# This dictionary contains default values for any parameters that should have default values
default_kwarg_values = {}


# This method defines a question Renderer
# For more info on Renderers, see {some doc}
# !!! This example renderer should render RenderedStringTemplateContent, not just a string
```

```
#     @classmethod
#     @renderer(renderer_type="renderer.question")
#     def _question_renderer(
#         cls, configuration, result=None, language=None, runtime_configuration=None
#     ):
#         column = configuration.kwargs.get("column")
#         mostly = configuration.kwargs.get("mostly")

#         return f'Do at least {mostly * 100}% of values in column "{column}" equal 3?'

# This method defines an answer Renderer
# !!! This example renderer should render RenderedStringTemplateContent, not just a string
#     @classmethod
#     @renderer(renderer_type="renderer.answer")
#     def _answer_renderer(
#         cls, configuration=None, result=None, language=None, runtime_configuration=None
#     ):
#         column = result.expectation_config.kwargs.get("column")
#         mostly = result.expectation_config.kwargs.get("mostly")
#         regex = result.expectation_config.kwargs.get("regex")
#         if result.success:
#             return f'At least {mostly * 100}% of values in column "{column}" equal 3.'
#         else:
#             return f'Less than {mostly * 100}% of values in column "{column}" equal 3.'

# This method defines a prescriptive Renderer
```

```
#     @classmethod
#     @renderer(renderer_type="renderer.prescriptive")
#     @render_evaluation_parameter_string
#     def _prescriptive_renderer(
#         cls,
#         configuration=None,
#         result=None,
#         language=None,
#         runtime_configuration=None,
#         **kwargs,
#     ):
# !!! This example renderer should be shorter
#         runtime_configuration = runtime_configuration or {}
#         include_column_name = runtime_configuration.get("include_column_name", True)
#         include_column_name = (
#             include_column_name if include_column_name is not None else True
#         )
#         styling = runtime_configuration.get("styling")
#         params = substitute_none_for_missing(
#             configuration.kwargs,
#             ["column", "regex", "mostly", "row_condition", "condition_parser"],
#         )

#         template_str = "values must be equal to 3"
#         if params["mostly"] is not None:
#             params["mostly_pct"] = num_to_str(
#                 params["mostly"] * 100, precision=15, no_scientific=True
```

```
#         )
#         # params["mostly_pct"] = "{:.14f}".format(params["mostly"]*100).rstrip("0").rstrip(".")
#         template_str += ", at least $mostly_pct % of the time."
#     else:
#         template_str += "."

#     if include_column_name:
#         template_str = "$column " + template_str

#     if params["row_condition"] is not None:
#         (
#             conditional_template_str,
#             conditional_params,
#         ) = parse_row_condition_string_pandas_engine(params["row_condition"])
#         template_str = conditional_template_str + ", then " + template_str
#         params.update(conditional_params)

#     return [
#         RenderedStringTemplateContent(
#             **{
#                 "content_block_type": "string_template",
#                 "string_template": {
#                     "template": template_str,
#                     "params": params,
#                     "styling": styling,
#                 },
#             }
```

)

]

if _name_ == "_main_":

ExpectColumnValuesToContainValidEmail().print_diagnostic_checklist()

```
1  validator.expect_column_values_to_contain_valid_email(column="email")
```

▸ (5) Spark Jobs

Loading widget. This should take less than 30 seconds.


```
Out[38]: {
  "result": {
    "element_count": 100,
    "unexpected_count": 1,
    "unexpected_percent": 1.0,
    "partial_unexpected_list": [
      "bocojheb@ulpeti.co.uk"
    ],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 1.0,
    "unexpected_percent_nonmissing": 1.0
  },
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "meta": {},
  "success": false
}
```