PyDeequ

Release 0.0.4

Calvin Wang, Chris Ghyzel, Joan Aoanan, Veronika Megler

Sep 21, 2023

CONTENTS:

CHAPTER

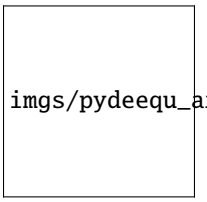
ONE

PYDEEQU

PyDeequ is a Python API for Deequ, a library built on top of Apache Spark for defining "unit tests for data", which measure data quality in large datasets. PyDeequ is written to support usage of Deequ in Python.

There are 4 main components of Deequ, and they are:

- Metrics Computation:
  - `Profiles` leverages Analyzers to analyze each column of a dataset.
  - `Analyzers` serve here as a foundational module that computes metrics for data profiling and validation at scale.
- Constraint Suggestion:
  - Specify rules for various groups of Analyzers to be run over a dataset to return back a collection of constraints suggested to run in a Verification Suite.
- Constraint Verification:
  - Perform data validation on a dataset with respect to various constraints set by you.
- Metrics Repository
  - Allows for persistence and tracking of Deequ runs over time.



1.1 Announcements

- **NEW!!!** 1.1.0 release of Python Deequ has been published to PYPI https://pypi.org/project/pydeequ/. This release brings many recency upgrades including support up to Spark 3.3.0! Any feedbacks are welcome through github issues.
- With PyDeequ v0.1.8+, we now officially support Spark3 ! Just make sure you have an environment variable `SPARK_VERSION` to specify your Spark version!
- We've release a blogpost on integrating PyDeequ onto AWS leveraging services such as AWS Glue, Athena, and SageMaker! Check it out: Monitor data quality in your data lake using PyDeequ and AWS Glue.

- Check out the PyDeequ Release Announcement Blogpost with a tutorial walkthrough the Amazon Reviews dataset!
- Join the PyDeequ community on PyDeequ Slack to chat with the devs!

1.2 Quickstart

The following will quickstart you with some basic usage. For more in-depth examples, take a look in the `tutorials/` directory for executable Jupyter notebooks of each module. For documentation on supported interfaces, view the `documentation`.

1.2.1 Installation

You can install PyDeequ via pip.

```
pip install pydeequ
```

1.2.2 Set up a PySpark session

```
from pyspark.sql import SparkSession, Row
import pydeequ

spark = (SparkSession
    .builder
    .config("spark.jars.packages", pydeequ.deequ_maven_coord)
    .config("spark.jars.excludes", pydeequ.f2j_maven_coord)
    .getOrCreate())

df = spark.sparkContext.parallelize([
            Row(a="foo", b=1, c=5),
            Row(a="bar", b=2, c=6),
            Row(a="baz", b=3, c=None)]).toDF()
```

1.2.3 Analyzers

```
from pydeequ.analyzers import *

analysisResult = AnalysisRunner(spark) \
                    .onData(df) \
                    .addAnalyzer(Size()) \
                    .addAnalyzer(Completeness("b")) \
                    .run()

analysisResult_df = AnalyzerContext.successMetricsAsDataFrame(spark, analysisResult)
analysisResult_df.show()
```

1.2.4 Profile

```
from pydeequ.profiles import *

result = ColumnProfilerRunner(spark) \
    .onData(df) \
    .run()

for col, profile in result.profiles.items():
    print(profile)
```

1.2.5 Constraint Suggestions

```
from pydeequ.suggestions import *

suggestionResult = ConstraintSuggestionRunner(spark) \
             .onData(df) \
             .addConstraintRule(DEFAULT()) \
             .run()

# Constraint Suggestions in JSON format
print(suggestionResult)
```

1.2.6 Constraint Verification

```
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Warning, "Review Check")

checkResult = VerificationSuite(spark) \
    .onData(df) \
    .addCheck(
        check.hasSize(lambda x: x >= 3) \
        .hasMin("b", lambda x: x == 0) \
        .isComplete("c")  \
        .isUnique("a")  \
        .isContainedIn("a", ["foo", "bar", "baz"]) \
        .isNonNegative("b")) \
    .run()

checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
checkResult_df.show()
```

1.2.7 Repository

Save to a Metrics Repository by adding the `useRepository()` and `saveOrAppendResult()` calls to your Analysis Runner.

```
from pydeequ.repository import *
from pydeequ.analyzers import *

metrics_file = FileSystemMetricsRepository.helper_metrics_file(spark, 'metrics.json')
repository = FileSystemMetricsRepository(spark, metrics_file)
key_tags = {'tag': 'pydeequ hello world'}
resultKey = ResultKey(spark, ResultKey.current_milli_time(), key_tags)

analysisResult = AnalysisRunner(spark) \
    .onData(df) \
    .addAnalyzer(ApproxCountDistinct('b')) \
    .useRepository(repository) \
    .saveOrAppendResult(resultKey) \
    .run()
```

To load previous runs, use the `repository` object to load previous results back in.

```
result_metrep_df = repository.load() \
    .before(ResultKey.current_milli_time()) \
    .forAnalyzers([ApproxCountDistinct('b')]) \
    .getSuccessMetricsAsDataFrame()
```

1.2.8 Wrapping up

After you've ran your jobs with PyDeequ, be sure to shut down your Spark session to prevent any hanging processes.

```
spark.sparkContext._gateway.shutdown_callback_server()
spark.stop()
```

1.3 Contributing

Please refer to the contributing doc for how to contribute to PyDeequ.

1.4 License

This library is licensed under the Apache 2.0 License.

1.5 Contributing Developer Setup

1. Setup *SDKMAN*
2. Setup *Java*
3. Setup *Apache Spark*
4. Install *Poetry*
5. Run *tests locally*

1.5.1 Setup SDKMAN

SDKMAN is a tool for managing parallel Versions of multiple Software Development Kits on any Unix based system. It provides a convenient command line interface for installing, switching, removing and listing Candidates. SDKMAN! installs smoothly on Mac OSX, Linux, WSL, Cygwin, etc… Support Bash and ZSH shells. See documentation on the SDKMAN! website.

Open your favourite terminal and enter the following:

```
$ curl -s https://get.sdkman.io | bash
If the environment needs tweaking for SDKMAN to be installed,
the installer will prompt you accordingly and ask you to restart.

Next, open a new terminal or enter:

$ source "$HOME/.sdkman/bin/sdkman-init.sh"

Lastly, run the following code snippet to ensure that installation succeeded:

$ sdk version
```

1.5.2 Setup Java

Install Java Now open favourite terminal and enter the following:

```
List the AdoptOpenJDK OpenJDK versions
$ sdk list java

To install For Java 11
$ sdk install java 11.0.10.hs-adpt

To install For Java 11
$ sdk install java 8.0.292.hs-adpt
```

1.5.3 Setup Apache Spark

Install Java Now open favourite terminal and enter the following:

```
List the Apache Spark versions:
$ sdk list spark

To install For Spark 3
$ sdk install spark 3.0.2
```

1.5.4 Poetry

Poetry Commands

```
poetry install

poetry update

# --tree: List the dependencies as a tree.
# --latest (-l): Show the latest version.
# --outdated (-o): Show the latest version but only for packages that are outdated.
poetry show -o
```

1.6 Running Tests Locally

Take a look at tests in `tests/dataquality` and `tests/jobs`

```
$ poetry run pytest
```

CHAPTER

TWO

APIS

The PyDeequ SDK consists of a variety of modules:

2.1 Core APIs

2.1.1 Analyzers

Analyzers file for all the different analyzers classes in Deequ

class pydeequ.analyzers.**AnalysisRunBuilder**(*spark_session: SparkSession*, *df: DataFrame*)

Bases: `object`

Low level class for running analyzers module. This is meant to be called by AnalysisRunner.

Parameters

- **SparkSession** (*spark_session*) – SparkSession
- **df** (*DataFrame*) – DataFrame to run the Analysis on.

addAnalyzer(*analyzer: _AnalyzerObject*)

Adds a single analyzer to the current Analyzer run.

Parameters
analyzer – Adds an analyzer strategy to the run.

Return self
for further chained method calls.

run()

Run the Analysis.

Returns
self: Runs the AnalysisRunBuilder.

saveOrAppendResult(*resultKey:* ResultKey)

A shortcut to save the results of the run or append them to existing results in the metrics repository.

Parameters
resultKey (ResultKey) – The result key to identify the current run

Returns
self

useRepository(*repository:* MetricsRepository)

Set a metrics repository associated with the current data to enable features like reusing previously computed results and storing the results of the current run.

Parameters
repository (MetricsRepository) – A metrics repository to store and load results associated with the run

Returns
self

class pydeequ.analyzers.**AnalysisRunner**(*spark_session: SparkSession*)

Bases: object

Runs a set of analyzers on the data at hand and optimizes the resulting computations to minimize the number of scans over the data. Additionally, the internal states of the computation can be stored and aggregated with existing states to enable incremental computations.

Parameters
SparkSession (*spark_session*) – SparkSession

onData(*df*)

Starting point to construct an AnalysisRun. :param dataFrame df: tabular data on which the checks should be verified :return: new AnalysisRunBuilder object

class pydeequ.analyzers.**AnalyzerContext**

Bases: object

The result returned from AnalysisRunner and Analysis.

classmethod successMetricsAsDataFrame(*spark_session: SparkSession*, *analyzerContext*, *forAnalyzers: Optional[list] = None*, *pandas: bool = False*)

Get the Analysis Run as a DataFrame.

Parameters

- **spark_session** (*SparkSession*) – SparkSession
- **analyzerContext** (AnalyzerContext) – Analysis Run
- **forAnalyzers** (*list*) – Subset of Analyzers from the Analysis Run

Return DataFrame
DataFrame of Analysis Run

classmethod successMetricsAsJson(*spark_session: SparkSession*, *analyzerContext*, *forAnalyzers: Optional[list] = None*)

Get the Analysis Run as a JSON.

Parameters

- **spark_session** (*SparkSession*) – SparkSession
- **analyzerContext** (AnalyzerContext) – Analysis Run
- **forAnalyzers** (*list*) – Subset of Analyzers from the Analysis Run

:return JSON : JSON output of Analysis Run

class pydeequ.analyzers.**ApproxCountDistinct**(*column: str*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Computes the approximate count distinctness of a column with HyperLogLogPlusPlus.

Parameters

- **column** (*str*) – Column to compute this aggregation on.
- **where** (*str*) – Additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**ApproxQuantile**(*column: str*, *quantile: float*, *relativeError: float = 0.01*, *where=None*)

Bases: _AnalyzerObject

Computes the Approximate Quantile of a column. The allowed relative error compared to the exact quantile can be configured with the *relativeError* parameter.

Parameters

- **column** (*str*) – The column in the DataFrame for which the approximate quantile is analyzed.
- **quantile** (*float [0,1]*) – The computed quantile. It must be within the interval [0, 1], where 0.5 would be the median.
- **relativeError** (*float [0,1]*) – Relative target precision to achieve in the quantile computation. A *relativeError* = 0.0 would yield the exact quantile while increasing the computational load.
- **where** (*str*) – Additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**ApproxQuantiles**(*column*, *quantiles*, *relativeError=0.01*)

Bases: _AnalyzerObject

Computes the approximate quantiles of a column. The allowed relative error compared to the exact quantile can be configured with *relativeError* parameter.

Parameters

- **column** (*str*) – Column in DataFrame for which the approximate quantile is analyzed.
- **quantiles** (*List[float[0,1]]*) – Computed Quantiles. Must be in the interval [0, 1], where 0.5 would be the median.
- **relativeError** (*float [0,1]*) – Relative target precision to achieve in the quantile computation. A *relativeError* = 0.0 would yield the exact quantile while increasing the computational load.

class pydeequ.analyzers.**Completeness**(*column*, *where=None*)

Bases: _AnalyzerObject

Completeness is the fraction of non-null values in a column.

Parameters

- **column** (*str*) – Column in DataFrame for which Completeness is analyzed.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Compliance**(*instance*, *predicate*, *where=None*)

Bases: _AnalyzerObject

Compliance measures the fraction of rows that complies with the given column constraint. E.g if the constraint is "att1>3" and data frame has 5 rows with att1 column value greater than 3 and 10 rows under 3; a DoubleMetric would be returned with 0.33 value.

Parameters

- **instance** (*str*) – Unlike other column analyzers (e.g completeness) this analyzer can not infer to the metric instance name from column name. Also the constraint given here can be referring to multiple columns, so metric instance name should be provided,describing what the analysis being done for.
- **predicate** (*str*) – SQL-predicate to apply per row
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Correlation**(*column1*, *column2*, *where=None*)

Bases: _AnalyzerObject

Computes the pearson correlation coefficient between the two given columns.

Parameters

- **column1** (*str*) – First column in the DataFrame for which the Correlation is analyzed.
- **column2** (*str*) – Second column in the DataFrame for which the Correlation is analyzed.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**CountDistinct**(*columns*)

Bases: _AnalyzerObject

Counts the distinct elements in the column(s).

Parameters

columns (*List[str]*) – Column(s) in the DataFrame for which distinctness is analyzed.

class pydeequ.analyzers.**DataType**(*column*, *where=None*)

Bases: _AnalyzerObject

Data Type Analyzer. Returns the datatypes of column

Parameters

- **column** (*str*) – Column in the DataFrame for which data type is analyzed.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**DataTypeInstances**(*value*)

Bases: Enum

An enum class that types columns to scala datatypes

Boolean = 'Boolean'

Fractional = 'Fractional'

Integral = 'Integral'

String = 'String'

Unknown = 'Unknown'

class pydeequ.analyzers.**Distinctness**(*columns*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Count the distinctness of elements in column(s). Distinctness is the fraction of distinct values of a column(s).

Parameters

- **columns** (*str OR list[str]*) – Column(s) in the DataFrame for which data type is to be analyzed. The column is expected to be a str for single column or list[str] for multiple columns.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Entropy**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Entropy is a measure of the level of information contained in a message. Given the probability distribution over values in a column, it describes how many bits are required to identify a value.

Parameters

- **column** (*str*) – Column in DataFrame for which entropy is calculated.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Histogram**(*column*, *binningUdf=None*, *maxDetailBins: Optional[int] = None*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Histogram is the summary of values in a column of a DataFrame. It groups the column's values then calculates the number of rows with that specific value and the fraction of the value.

Parameters

- **column** (*str*) – Column in DataFrame to do histogram analysis.
- **binningUdf** (*lambda expr*) – Optional binning function to run before grouping to re-categorize the column values.For example to turn a numerical value to a categorical value binning functions might be used.
- **maxDetailBins** (*int*) – Histogram details is only provided for N column values with top counts. MaxBins sets the N. This limit does not affect what is being returned as number of bins.It always returns the distinct value count.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**KLLParameters**(*spark_session: SparkSession*, *sketchSize: int*, *shrinkingFactor: float*, *numberOfBuckets: int*)

Bases: object

Parameter definition for KLL Sketches.

Parameters

- **sketchSize** (*int*) – size of kll sketch.
- **shrinkingFactor** (*float*) – shrinking factor of kll sketch.
- **numberOfBuckets** (*int*) – number of buckets.

class pydeequ.analyzers.**KLLSketch**(*column: str*, *kllParameters:* KLLParameters)

Bases: _AnalyzerObject

The KLL Sketch analyzer.

Parameters

- **column** (*str*) – Column in DataFrame to do histogram analysis.
- **kllParameters** (KLLParameters) – parameters of KLL Sketch

class pydeequ.analyzers.**MaxLength**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

MaxLength Analyzer. Get Max length of a str type column.

Parameters

- **column** (*str*) – column in DataFrame to find the maximum length. Column is expected to be a str type.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Maximum**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Get the maximum of a numeric column.

class pydeequ.analyzers.**Mean**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Mean Analyzer. Get mean of a column

Parameters

- **column** (*str*) – Column in DataFrame to find the mean.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**MinLength**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Get the minimum length of a column

Parameters

column (*str*) – Column in DataFrame to find the minimum Length. Column is expected to be a str type.

:param str where : additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Minimum**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Count the distinct elements in a single or multiple columns

Parameters

- **column** (*str*) – Column in DataFrame to find the minimum value.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**MutualInformation**(*columns*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Describes how much information about one column can be inferred from another column.

Parameters

- **columns** (*list[str]*) – Columns in DataFrame for mutual information analysis.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**PatternMatch**(*column*, *pattern_regex: str*, **pattern_groupNames*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

PatternMatch is a measure of the fraction of rows that complies with a given column regex constraint.

E.g A sample dataFrame column has five rows that contain a credit card number and 10 rows that do not. According to regex, using the constraint Patterns.CREDITCARD returns a doubleMetric .33 value.

Parameters

- **column** (*str*) – Column in DataFrame to check pattern.
- **pattern_regex** (*str*) – pattern regex
- **pattern_groupNames** (*str*) – groupNames for pattern regex
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Size**(*where: Optional[str] = None*)

Bases: _AnalyzerObject

Size is the number of rows in a DataFrame.

Parameters

where (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**StandardDeviation**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Calculates the Standard Deviation of column

Parameters

- **column** (*str*) – Column in DataFrame for standard deviation calculation.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Sum**(*column*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Calculates the sum of a column

Parameters

- **column** (*str*) – Column in DataFrame to calculate the sum.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**UniqueValueRatio**(*columns*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Calculates the ratio of uniqueness.

Parameters

- **columns** (*list[str]*) – Columns in DataFrame to find unique value ratio.
- **where** (*str*) – additional filter to apply before the analyzer is run.

class pydeequ.analyzers.**Uniqueness**(*columns*, *where: Optional[str] = None*)

Bases: _AnalyzerObject

Uniqueness is the fraction of unique values of column(s), values that occur exactly once.

Parameters

- **columns** (*list[str]*) – Columns in DataFrame to find uniqueness.

- **where** (*str*) – additional filter to apply before the analyzer is run.

2.1.2 Anomaly Detection

2.1.3 Checks

class pydeequ.checks.**Check**(*spark_session: SparkSession*, *level:* CheckLevel, *description: str*, *constraints: Optional[list] = None*)

Bases: `object`

A class representing a list of constraints that can be applied to a given
[[org.apache.spark.sql.DataFrame]]. In order to run the checks, use the *run* method. You can also use VerificationSuite.run to run your checks along with other Checks and Analysis objects. When run with VerificationSuite, Analyzers required by multiple checks/analysis blocks is optimized to run once.

addConstraint(*constraint*)

Returns a new Check object with the given constraints added to the constraints list. :param Constraint constraint: new constraint to be added. :return: new Check object

addConstraints(*constraints: list*)

addFilterableContstraint(*creationFunc*)

Adds a constraint that can subsequently be replaced with a filtered version :param creationFunc: :return:

areAnyComplete(*columns*, *hint=None*)

Creates a constraint that asserts any completion in the combined set of columns.

Parameters

- **columns** (*list[str]*) – Columns in Data Frame to run the assertion on.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
areAnyComplete self: A Check.scala object that asserts completion in the columns.

areComplete(*columns*, *hint=None*)

Creates a constraint that asserts completion in combined set of columns.

Parameters

- **columns** (*list[str]*) – Columns in Data Frame to run the assertion on.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
areComplete self: A Check.scala object that asserts completion in the columns.

containsCreditCardNumber(*column*, *assertion=None*, *hint=None*)

Check to run against the compliance of a column against a Credit Card pattern.

Parameters

- **column** (*str*) – Column in DataFrame to be checked. The column is expected to be a string type.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*hint*) – A hint that states why a constraint could have failed.

Returns
containsCreditCardNumber self: A Check object that runs the compliance on the column.

containsEmail(*column*, *assertion=None*, *hint=None*)

Check to run against the compliance of a column against an e-mail pattern.

Parameters

- **column** (*str*) – The Column in DataFrame to be checked. The column is expected to be a string datatype.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
containsCreditCardNumber self: A Check object that runs the compliance on the column.

containsSocialSecurityNumber(*column*, *assertion=None*, *hint=None*)

Check to run against the compliance of a column against the Social security number pattern for the US.

Parameters

- **column** (*str*) – The Column in DataFrame to be checked. The column is expected to be a string datatype.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
containsSocialSecurityNumber self: A Check object that runs the compliance on the column.

containsURL(*column*, *assertion=None*, *hint=None*)

Check to run against the compliance of a column against an e-mail pattern.

Parameters

- **column** (*str*) – The Column in DataFrame to be checked. The column is expected to be a string datatype.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
containsURL self: A Check object that runs the compliance on the column.

evaluate(*context*)

Evaluate this check on computed metrics.

Parameters
context – result of the metrics computation

Returns
evaluate self: A Check object that evaluates the check.

hasApproxCountDistinct(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the approximate count distinct of the given column

Parameters

- **column** (*str*) – Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function with an int or float parameter.

- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasApproxCountDistinct self: A Check object that asserts the count distinct of the column.

hasApproxQuantile(*column*, *quantile*, *assertion*, *hint=None*)

Creates a constraint that asserts on an approximated quantile

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on
- **quantile** (*float*) – Quantile to run the assertion on.
- **assertion** (*lambda*) – A function that accepts the computed quantile as an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasApproxQuantile self: A Check object that asserts the approximated quantile in the column.

hasCompleteness(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts column completion. Uses the given history selection strategy to retrieve historical completeness values on this column from the history provider.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasCompleteness self: A Check object that implements the assertion.

hasCorrelation(*columnA*, *columnB*, *assertion*, *hint=None*)

Creates a constraint that asserts on the pearson correlation between two columns.

Parameters

- **columnA** (*str*) – First column in Data Frame which calculates the correlation.
- **columnB** (*str*) – Second column in Data Frame which calculates the correlation.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasCorrelation self: A Check object that asserts the correlation calculation in the columns.

hasDataType(*column*, *datatype:* ConstrainableDataTypes, *assertion=None*, *hint=None*)

Check to run against the fraction of rows that conform to the given data type.

Parameters

- **column** (*str*) – The Column in DataFrame to be checked.
- **datatype** (ConstrainableDataTypes) – Data type that the columns should be compared against
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasDataType self: A Check object that runs the compliance on the column.

hasDistinctness(*columns*, *assertion*, *hint=None*)

Creates a constraint on the distinctness in a single or combined set of key columns. Distinctness is the fraction of distinct values of a column(s).

Parameters
columns (*list[str]*) – Column(s) in Data Frame to run the assertion on.

:param lambda assertion : A function that accepts an int or float parameter. :param str hint: A hint that states why a constraint could have failed. :return: hasDistinctness self: A Check object that asserts distinctness in the columns.

hasEntropy(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on a column entropy. Entropy is a measure of the level of information contained in a message.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasEntropy self: A Check object that asserts the entropy in the column.

hasHistogramValues(*column*, *assertion*, *binningUdf*, *maxBins*, *hint=None*)

Creates a constraint that asserts on column's value distribution.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter as a distribution input parameter.
- **binningUDF** (*str*) – An optional binning function.
- **maxBins** (*str*) – Histogram details is only provided for N column values with top counts. MaxBins sets the N.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasHistogramValues self: A Check object that asserts the column's value distribution in the column.

hasMax(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the maximum of the column. The column contains either a long, int or float datatype.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on. The column is expected to be an int, long or float type.
- **assertion** (*lambda*) – A function which accepts an int or float parameter that discerns the maximum value of the column.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMax self: A Check object that asserts maximum of the column.

hasMaxLength(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the maximum length of a string datatype column

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on. The column is expected to be a string type.
- **assertion** (*lanmbda*) – A function which accepts an int or float parameter that discerns the maximum length of the string.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMaxLength self : A Check object that asserts maxLength of the column.

hasMean(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the mean of the column

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function with an int or float parameter. The parameter is the mean.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMean self: A Check object that asserts the mean of the column.

hasMin(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the minimum of a column. The column is contains either a long, int or float datatype.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on. The column is expected to be an int, long or float type.
- **assertion** (*lambda*) – A function which accepts an int or float parameter that discerns the minumum value of the column.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMaxLength self: A Check object that asserts the minumum of the column.

hasMinLength(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the minimum length of a string datatype column.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on. The column is expected to be a string type.
- **assertion** (*lambda*) – A function which accepts the int or float parameter that discerns the minimum length of the string.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMinLength self: A Check object that asserts minLength of the column.

hasMutualInformation(*columnA*, *columnB*, *assertion*, *hint=None*)

Creates a constraint that asserts on a mutual information between two columns. Mutual Information describes how much information about one column can be inferred from another.

Parameters

- **columnA** (*str*) – First column in Data Frame which calculates the mutual information.
- **columnB** (*str*) – Second column in Data Frame which calculates the mutual information.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMutualInformation self: A Check object that asserts the mutual information in the columns.

hasNumberOfDistinctValues(*column*, *assertion*, *binningUdf*, *maxBins*, *hint=None*)

Creates a constraint that asserts on the number of distinct values a column has.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **binningUDF** (*lambda*) – An optional binning function.
- **maxBins** (*int*) – Histogram details is only provided for N column values with top counts. MaxBins sets the N.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasNumberOfDistinctValues self: A Check object that asserts distinctness in the column.

hasPattern(*column*, *pattern*, *assertion=None*, *name=None*, *hint=None*)

Checks for pattern compliance. Given a column name and a regular expression, defines a Check on the average compliance of the column's values to the regular expression.

Parameters

- **column** (*str*) – Column in DataFrame to be checked
- **pattern** (*Regex*) – A name that summarizes the current check and the metrics for the analysis being done.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **name** (*str*) – A name for the pattern constraint.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasPattern self: A Check object that runs the condition on the column.

hasSize(*assertion*, *hint=None*)

Creates a constraint that calculates the data frame size and runs the assertion on it.

Parameters

- **assertion** (*lambda*) – Refers to a data frame size. The given function can include comparisons and conjunction or disjunction statements.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasSize self: A Check.scala object that implements the assertion on the column.

hasStandardDeviation(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the standard deviation of the column

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function with an int or float parameter. The parameter is the standard deviation.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMean self: A Check object that asserts the std deviation of the column.

hasSum(*column*, *assertion*, *hint=None*)

Creates a constraint that asserts on the sum of the column

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function with an int or float parameter. The parameter is the sum.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasMean self: A Check object that asserts the sum of the column.

hasUniqueValueRatio(*columns*, *assertion*, *hint=None*)

Creates a constraint on the unique value ratio in a single or combined set of key columns.

Parameters

- **columns** (*list[str]*) – Column(s) in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasUniqueValueRatio self: A Check object that asserts the unique value ratio in the columns.

hasUniqueness(*columns*, *assertion*, *hint=None*)

Creates a constraint that asserts any uniqueness in a single or combined set of key columns. Uniqueness is the fraction of unique values of a column(s) values that occur exactly once.

Parameters

- **columns** (*list[str]*) – Column(s) in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
hasUniqueness self: A Check object that asserts uniqueness in the columns.

haveAnyCompleteness(*columns*, *assertion*, *hint=None*)

Creates a constraint that asserts on any completion in the combined set of columns.

Parameters

- **columns** (*list[str]*) – Columns in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
haveAnyCompleteness self: A Check.scala object that asserts completion in the columns.

haveCompleteness(*columns*, *assertion*, *hint=None*)

Creates a constraint that asserts on completed rows in a combined set of columns.

Parameters

- **columns** (*list[str]*) – Columns in Data Frame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
haveCompleteness self: A Check.scala object that implements the assertion on the columns.

isComplete(*column*, *hint=None*)

Creates a constraint that asserts on a column completion.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **hint** (*str*) – A hint that discerns why a constraint could have failed.

Returns
isComplete self:A Check.scala object that asserts on a column completion.

isContainedIn(*column*, *allowed_values*)

Asserts that every non-null value in a column is contained in a set of predefined values

Parameters

- **column** (*str*) – Column in DataFrame to run the assertion on.
- **allowed_values** (*list[str]*) – A function that accepts allowed values for the column.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
isContainedIn self: A Check object that runs the assertion on the columns.

isGreaterThan(*columnA*, *columnB*, *assertion=None*, *hint=None*)

Asserts that, in each row, the value of columnA is greater than the value of columnB

Parameters

- **columnA** (*str*) – Column in DataFrame to run the assertion on.
- **columnB** (*str*) – Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
isGreaterThan self: A Check object that runs the assertion on the columns.

`isGreaterThanOrEqualTo`(*columnA*, *columnB*, *assertion=None*, *hint=None*)

Asserts that, in each row, the value of columnA is greather than or equal to the value of columnB

Parameters

- **columnA** (*str*) – Column in DataFrame to run the assertion on.
- **columnB** (*str*) – Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
isGreaterThanOrEqualTo self: A Check object that runs the assertion on the columns.

`isLessThan`(*columnA*, *columnB*, *assertion=None*, *hint=None*)

Asserts that, in each row, the value of columnA is less than the value of columnB

Parameters

- **columnA** (*str*) – Column in DataFrame to run the assertion on.
- **columnB** (*str*) – Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
isLessThan self : A Check object that checks the assertion on the columns.

`isLessThanOrEqualTo`(*columnA*, *columnB*, *assertion=None*, *hint=None*)

Asserts that, in each row, the value of columnA is less than or equal to the value of columnB.

Parameters

- **columnA** (*str*) – Column in DataFrame to run the assertion on.
- **columnB** (*str*) – Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function that accepts an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
isLessThanOrEqualTo self (isLessThanOrEqualTo): A Check object that checks the assertion on the columns.

`isNonNegative`(*column*, *assertion=None*, *hint=None*)

Creates a constraint which asserts that a column contains no negative values.

Parameters

- **column** (*str*) – The Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
self (isNonNegative): A Check object that runs the compliance on the column.

isPositive(*column*, *assertion=None*, *hint=None*)

Creates a constraint which asserts that a column contains no negative values and is greater than 0.

Parameters

- **column** (*str*) – The Column in DataFrame to run the assertion on.
- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns

isNonNegative self: A Check object that runs the assertion on the column.

isPrimaryKey(*column*, **columns*, *hint=None*)

Creates a constraint that asserts on a column(s) primary key characteristics. Currently only checks uniqueness, but reserved for primary key checks if there is another assertion to run on primary key columns.

how does column and columns differ :param str column: Column in Data Frame to run the assertion on. :param str hint: A hint that states why a constraint could have failed. :param list[str] columns: Columns to run the assertion on. :return: isPrimaryKey self: A Check.scala object that asserts completion in the columns.

isUnique(*column*, *hint=None*)

Creates a constraint that asserts on a column uniqueness

Parameters

- **columns** (*list[str]*) – Columns in Data Frame to run the assertion on.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns

isUnique self: A Check.scala object that asserts uniqueness in the column.

kllSketchSatisfies(*column*, *assertion*, *kllParameters=None*, *hint=None*)

Creates a constraint that asserts on column's sketch size.

Parameters

- **column** (*str*) – Column in Data Frame to run the assertion on.
- **assertion** (*Lambda(BucketDistribution)*) – A function that accepts an int or float parameter as a distribution input parameter.
- **kllParameters** (KLLParameters) – Parameters of KLL sketch
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns

kllSketchSatisfies self: A Check object that asserts the column's sketch size in the column.

requiredAnalyzers()

satisfies(*columnCondition*, *constraintName*, *assertion=None*, *hint=None*)

Creates a constraint that runs the given condition on the data frame.

Parameters

- **columnCondition** (*str*) – Data frame column which is a combination of expression and the column name. It has to comply with Spark SQL syntax. Can be written in an exact same way with conditions inside the *WHERE* clause.
- **constraintName** (*str*) – A name that summarizes the check being made. This name is being used to name the metrics for the analysis being done.

- **assertion** (*lambda*) – A function with an int or float parameter.
- **hint** (*str*) – A hint that states why a constraint could have failed.

Returns
satisfies self: A Check object that runs the condition on the data frame.

class pydeequ.checks.**CheckLevel**(*value*)

Bases: Enum

An enumeration.

Error = 'Error'

Warning = 'Warning'

class pydeequ.checks.**CheckResult**

Bases: object

class pydeequ.checks.**CheckStatus**(*value*)

Bases: Enum

An enumeration.

Error = 'Error'

Success = 'Success'

Warning = 'Warning'

class pydeequ.checks.**ConstrainableDataTypes**(*value*)

Bases: Enum

An enumeration.

Boolean = 'Boolean'

Fractional = 'Fractional'

Integral = 'Integral'

Null = 'Null'

Numeric = 'Numeric'

String = 'String'

2.1.4 Profiles

2.1.5 Repository

Repository file for all the different metrics repository classes in Deequ

Author: Calvin Wang

class pydeequ.repository.**FileSystemMetricsRepository**(*spark_session: SparkSession*, *path: Optional[str] = None*)

Bases: *MetricsRepository*

High level FileSystemMetricsRepository Interface

Parameters

- **spark_session** – SparkSession
- **path** – The location of the file metrics repository.

class pydeequ.repository.**InMemoryMetricsRepository**(*spark_session: SparkSession*)

Bases: *MetricsRepository*

High level InMemoryMetricsRepository Interface

class pydeequ.repository.**MetricsRepository**

Bases: object

Base class for Metrics Repository

after(*dateTime: int*)

Only look at AnalysisResults with a result key with a greater value :param dateTime: The minimum dateTime of AnalysisResults to look at

before(*dateTime: int*)

Only look at AnalysisResults with a result key with a smaller value :param dateTime: The maximum dateTime of AnalysisResults to look at

forAnalyzers(*analyzers: list*)

Choose all metrics that you want to load :param analyzers: List of analyers who's resulting metrics you want to load

getSuccessMetricsAsDataFrame(*withTags: Optional[list] = None*, *pandas: bool = False*)

Get the AnalysisResult as DataFrame :param withTags: List of tags to filter previous Metrics Repository runs with

getSuccessMetricsAsJson(*withTags: Optional[list] = None*)

Get the AnalysisResult as JSON :param withTags: List of tags to filter previous Metrics Repository runs with

classmethod helper_metrics_file(*spark_session: SparkSession*, *filename: str = 'metrics.json'*)

Helper method to create the metrics file for storage

load()

Get a builder class to construct a loading query to get AnalysisResults

withTagValues(*tagValues: dict*)

Filter out results that don't have specific values for specific tags :param tagValues: Dict with tag names and the corresponding values to filter for

class pydeequ.repository.**ResultKey**(*spark_session: SparkSession*, *dataSetDate: Optional[int] = None*, *tags: Optional[dict] = None*)

Bases: object

Information that uniquely identifies a AnalysisResult

Parameters

- **spark_session** – SparkSession
- **dataSetDate** – Date of the result key
- **tags** – A map with additional annotations

static current_milli_time()

Get current time in milliseconds # TODO: Consider putting this into scala_utils? Or general utils?

2.1.6 Scala Utilities

A collection of utility functions and classes for manipulating with scala objects anc classes through py4j

class pydeequ.scala_utils.**PythonCallback**(*gateway*)

Bases: object

class pydeequ.scala_utils.**ScalaFunction1**(*gateway*, *lambda_function*)

Bases: *PythonCallback*

Implements scala.Function1 interface so we can pass lambda functions to Check https://www.scala-lang.org/api/current/scala/Function1.html

class Java

Bases: object

scala.Function1: a function that takes one argument

implements = ['scala.Function1']

apply(*arg*)

Implements the apply function

class pydeequ.scala_utils.**ScalaFunction2**(*gateway*, *lambda_function*)

Bases: *PythonCallback*

Implements scala.Function2 interface https://www.scala-lang.org/api/current/scala/Function2.html

class Java

Bases: object

scala.Function2: a function that takes two arguments

implements = ['scala.Function2']

apply(*t1*, *t2*)

Implements the apply function

pydeequ.scala_utils.**get_or_else_none**(*scala_option*)

pydeequ.scala_utils.**java_list_to_python_list**(*java_list: str*, *datatype*)

pydeequ.scala_utils.**scala_get_default_argument**(*java_object*, *argument_idx: int*) → JavaObject

pydeequ.scala_utils.**scala_map_to_dict**(*jvm*, *scala_map*)

pydeequ.scala_utils.**scala_map_to_java_map**(*jvm*, *scala_map*)

pydeequ.scala_utils.**to_scala_map**(*spark_session*, *d*)

Convert a dict into a JVM Map. Args:

spark_session: Spark session d: Python dictionary

Returns:
Scala map

pydeequ.scala_utils.**to_scala_seq**(*jvm*, *iterable*)

Helper method to take an iterable and turn it into a Scala sequence Args:

jvm: Spark session's JVM iterable: your iterable

Returns:
Scala sequence

2.1.7 Suggestions

2.1.8 Verification

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

I

J

K

L

M

N

O

P

R

S

T

U

W