Schemaflow Documentation

Release 0.1.0

Jorge C. Leitao

Feb 02, 2019

Contents:

This is a package to write robust pipelines for data science and data engineering in Python 3. Thanks for checking it out.

A major challenge in creating a robust data pipeline is guaranteeing interoperability between pipes. Data transformations often change the underlying data representation (e.g. change column type, add columns, convert PySpark DataFrame to Pandas or H2O DataFrames). This makes it difficult to track what exactly is going on at a certain point of the pipeline, which often requires running the whole pipeline until that point to debug a certain pipe.

This package declares a simple API to define data transformations that know what schema they require to run, what schema they return, and what states they depend on.

Under this API, you define a *Pipe* as follows (an example):

```
from pipeline import pipe, types


class MyPipe(pipe.Pipe):
    requirements = {'sklearn'}

    fit_requires = {
        # (arbitrary items, arbitrary features)
        'x': types.Array(np.float64, shape=(None, None)),
        'y': types.List(float)
    }

    transform_requires = {
        'x': types.List(float)
    }

    fit_parameters = {
        'gamma': float
    }

    # parameter assigned in fit; the pipe's state
    fitted_parameters = {
        'a': float
    }

    # type and key of transformed data
    transform_modifies = {
        'b': float
    }

    def fit(self, data, parameters=None):
        # accesses data['x'], data['y'] and parameters['gamma']; expects the types␣
↪defined above
        # assigns a float to self['a']

    def transform(self, data):
        # assigns a float to data['b']
        return data
```

Without reading nor executing `fit` and `transform`, we know how `data` will flow through this pipe:

1. it requires an `'x'` and `'y'` and a parameter `gamma` in fit
2. it is stateful through `a`
3. it transforms `data['b']`.

This allows to check whether a `Pipeline` is consistent **without** executing `fit` or `transform` of *any* pipe.

Specifically, you can execute the pipe using the traditional fit-transform idiom,

```
p = MyPipe()
p.fit(train_data, {'gamma': 1.0})
result = p.transform(test_data)
```

but also check whether the data format that you pass is consistent with its requirements:

```
p = MyPipe()
exceptions_fit = p.check_fit({'x': 1}, {'gamma': 1.0})
assert len(exceptions_fit) > 0

exceptions_transform = p.check_transform({'x': 1})
assert len(exceptions_transform) > 0
```

which does not execute `fit` nor `transform`.

The biggest advantage of this declaration is that when the pipes are used within a pipeline, **Schemaflow** can compute how the schema flows and therefore know the schema flow of a Pipeline:

```
p = schemaflow.pipeline.Pipeline([
    ('fix_ids', PipeA()),
    ('join_tables_with_fix', PipeB()),
    ('featurize', Featurize1_Pipeline()),
    ('model', Model1_Pipeline()),
    ('export_metrics', Export_results_PDF_Pipeline()),
    ('export_metrics', PushResultsToCache())
])

print(p.transform_modifies)
```

I.e. because we know how each Pipe modifies the schema, we can compute how the schema flows through it and therefore obtain what are the dependencies of `p` and what it transforms.

CHAPTER 1

When to use this package

Use it when you are fairly certain that:

- there is the need for a complex data pipeline (e.g. more than 1 data source and different data types)
- the data transforms are expensive (e.g. Spark, Hive, SQL)
- your data pipeline aims to be maintainable and reusable (e.g. production code)

1.1 Pipe

class schemaflow.pipe.**Pipe**

A Pipe represents a stateful data transformation.

Data in this context consists of a Python dictionary whose each value is a type with some representation of data, either in-memory (e.g. `float`, `pandas.DataFrame`) of remote (e.g. `pyspark.sql.DataFrame`, `sqlalchemy`).

A *`Pipe`* is defined by:

- a method *`transform()`* that:
  - uses the keys *`transform_modifies`* from `data`
  - uses the *`state`*
  - modifies the keys in *`transform_modifies`* in `data`
- a method *`fit()`* that:
  - uses (training) keys *`fit_requires`* from `data`
  - uses (passed) *`fit_parameters`*
  - modifies the keys *`fitted_parameters`* in *`state`*
- a set of *`requirements`* (a set of package names, e.g. `{'pandas'}`) of the transformation

All *transform_modifies* and *fit_requires* have a *Type* that can be used to check that the Pipe's input is consistent, with

- *check_fit()*
- *check_transform()*

The existence of the requirements can be checked using

- *check_requirements()*

The rational is that you can run `check_*` with access only to the data's schema. This is specially important when the schemaflow is an expensive operation.

requirements = set()
set of packages required by the Pipe.

fit_requires = {}
the data schema required in *fit()*; a dictionary `str`: *Type*.

transform_requires = {}
the data schema required in *transform()*; a dictionary `str`: `Type`.

fit_parameters = {}
parameters' schema passed to *fit()*

fitted_parameters = {}
schema of the parameters assigned in *fit()*

transform_modifies = {}
type and key of *transform()*

state = None
A dictionary with the states of the Pipe. Use [] operator to access and modify it.

check_requirements
Checks for requirements.

Returns a list of exceptions with missing requirements

check_fit(*data: dict*, *parameters: dict = None*, *raise_: bool = False*)
Checks that a given data has a valid schema to be used in *fit()*.

Parameters

- **data** – a dictionary with either `(str, Type)` or `(str, instance)`
- **parameters** – a dictionary with either `(str, Type)` or `(str, instance)`
- **raise** – whether it should raise the first found exception or list them all (default: list them)

Returns a list of (subclasses of) `PipelineError` with all failed checks.

check_transform(*data: dict*, *raise_: bool = False*)
Checks that a given data has a valid schema to be used in *transform()*.

Parameters

- **data** – a dictionary with either `(str, Type)` or `(str, instance)`
- **raise** – whether it should raise the first found exception or list them all (default: list them)

Returns a list of (subclasses of) *schemaflow.exceptions.SchemaFlowError* with all missing arguments.

transform_schema(*schema: dict*)
Transforms the `schema` into a new schema based on *transform_modifies*.

Parameters **schema** – a dictionary of pairs `str` *Type*.

Returns the new schema.

fit(*data: dict*, *parameters: dict = None*)
Modifies the instance's *state*.

Parameters

- **data** – a dictionary of pairs `(str, object)`.
- **parameters** – a dictionary of pairs `(str, object)`.

Returns `None`

transform(*data: dict*)
Modifies the data keys identified in *transform_modifies*.

Parameters **data** – a dictionary of pairs `(str, object)`.

Returns the modified data

1.2 Pipeline

class schemaflow.pipeline.**Pipeline**(*pipes*)
A list of *Pipe*'s that are applied sequentially.

Pipeline is a *Pipe* and can be part of another *Pipeline*.

Parameters **pipes** – a `list` or `OrderedDict` of *Pipe*. If passed as a list, you can either pass pipes or tuples `(name, Pipe)`.

pipes = None
An `OrderedDict` whose keys are the pipe's names or `str(index)` where `index` is the pipe's position in the sequence and the values are *Pipe*'s.

fit_requires
The data schema required in *fit()*.

transform_requires
The data schema required in *transform()*.

transform_modifies
The schema modifications that this Pipeline apply in `transform`.

When a key is modified more than once, changes are appended as a list.

fitted_parameters
Parameters assigned to fit of each pipe.

Returns a dictionary with the pipe's name and their respective *fitted_parameters*.

requirements
Set of packages required by the Pipeline. The union of all *requirements* of all pipes in the Pipeline.

check_transform(*data: dict = None*, *raise_: bool = False*)
Checks that a given data has a valid schema to be used in *transform()*.

Parameters

- **data** – a dictionary with either `(str, Type)` or `(str, instance)`

- **raise** – whether it should raise the first found exception or list them all (default: list them)

Returns a list of (subclasses of) *schemaflow.exceptions.SchemaFlowError* with all missing arguments.

check_fit (*data: dict*, *parameters: dict = None*, *raise_: bool = False*)
Checks that a given data has a valid schema to be used in *fit()*.

Parameters

- **data** – a dictionary with either `(str, Type)` or `(str, instance)`
- **parameters** – a dictionary with either `(str, Type)` or `(str, instance)`
- **raise** – whether it should raise the first found exception or list them all (default: list them)

Returns a list of (subclasses of) `PipelineError` with all failed checks.

transform (*data: dict*)
Applies each of *transform()* sequentially into `data`.

Parameters **data** – a dictionary of pairs `str, object`.

Returns the transformed data.

transform_schema (*schema: dict*)
Transforms the `schema` into a new schema based on *transform_modifies*.

Parameters **schema** – a dictionary of pairs `str` *Type*.

Returns the new schema.

fit (*data: dict*, *parameters: dict = None*)
Fits the *pipes* in sequence: `p1.fit, p1.transform, p2.fit, p2.transform, ..., pN.transform`.

Parameters

- **data** – a dictionary of pairs `(str, object)`.
- **parameters** – a dictionary `{pipe_name: {str: object}}`, where each of its value is the parameters to be passed to the respective's pipe named `pipe_name`.

Returns `None`

logged_transform (*data: dict*)
Performs the same operation as *transform()* while logging the schema on each intermediary step.

It also logs schema inconsistencies as errors. Specifically, for each pipe, it checks if its input data is consistent with its `transform_requires`, and whether its output data is consistent with its `transform_modifies`.

This greatly helps the Pipeline developer to identify problems in the pipeline.

Parameters **data** – a dictionary of pairs `str` *Type*.

Returns the transformed data.

logged_fit (*data: dict*, *parameters: dict = None*)
Performs the same operation as *fit()* while logging the schema on each intermediary step.

It also logs schema inconsistencies as errors. Specifically, for each pipe, it checks if its input data is consistent with its `fit_requires`, and whether its state changes is consistent with its `fitted_parameters`.

This greatly helps the Pipeline developer to identify problems in the pipeline.

Parameters

- **data** – a dictionary of pairs `(str, object)`.
- **parameters** – a dictionary of pairs `(str, object)`.

Returns `None`

1.3 Types

class schemaflow.types.**Type**

The base type of all types. Used to declare new types to be used in *schemaflow.pipe.Pipe*.

The class attribute *requirements* (a set of strings) is used to define if using this type has package requirements (e.g. *numpy*).

requirements = {}

set of packages required for this type to be usable.

classmethod base_type()

A class property that returns the underlying type of this Type. :return:

classmethod requirements_fulfilled()

Returns whether this Type has its requirements fulfilled.

Returns bool

check_schema(*instance: object*, *raise_: bool = False*)

Checks that the instance has the correct type and schema (composite types).

Parameters

- **instance** – a datum in either its representation form or on its schema form.
- **raise** –

Returns a list of exceptions

class schemaflow.types.**PySparkDataFrame**(*schema: dict*)

Representation of a pyspark.sql.DataFrame. Requires `pyspark`.

classmethod base_type()

A class property that returns the underlying type of this Type. :return:

class schemaflow.types.**PandasDataFrame**(*schema: dict*)

Representation of a pandas.DataFrame. Requires `pandas`.

classmethod base_type()

A class property that returns the underlying type of this Type. :return:

class schemaflow.types.**List**(*items_type*)

class schemaflow.types.**Tuple**(*items_type*)

classmethod base_type()

A class property that returns the underlying type of this Type. :return:

class schemaflow.types.**Array**(*items_type: type*, *shape=None*)

Representation of a numpy.array. Requires `numpy`.

classmethod base_type()
A class property that returns the underlying type of this Type. :return:

1.4 Exceptions

exception schemaflow.exceptions.**SchemaFlowError**(*locations: list = None*)
The base exception of Pipeline

exception schemaflow.exceptions.**NotFittedError**(*pipe*, *key*, *locations: list = None*)
SchemaFlowError raised when someone tries to access a non-fitted parameter.

exception schemaflow.exceptions.**MissingRequirement**(*object_type: type*, *requirement: str*, *locations: list = None*)
SchemaFlowError raised when a requirement is missing

exception schemaflow.exceptions.**WrongSchema**(*expected_columns*, *passed_columns*, *locations: list = None*)
SchemaFlowError raised when the schema of a datum is wrong (e.g. wrong shape)

exception schemaflow.exceptions.**WrongParameter**(*expected_columns*, *passed_columns*, *locations: list = None*)
SchemaFlowError raised when unexpected parameters are passed to *fit*.

exception schemaflow.exceptions.**WrongType**(*expected_type*, *base_type*, *locations: list = None*)
SchemaFlowError raised when the type of the datum is wrong

exception schemaflow.exceptions.**WrongShape**(*expected_shape*, *shape*, *locations: list = None*)
SchemaFlowError raised when the shape of the datum is wrong

Python Module Index

S

Index

W