Typedframe

Alexander Reshytko

Sep 07, 2023

CONTENTS:

Typed wrappers over pandas DataFrames with schema validation.

`TypedDataFrame` is a lightweight wrapper over pandas `DataFrame` that provides runtime schema validation and can be used to establish strong data contracts between interfaces in your Python code.

The goal of the library is to reveal and make explicit all unclear or forgotten assumptions about your DataFrame.

CHAPTER

ONE

QUICKSTART

Install typedframe library:

```
pip install typedframe
```

Assume an overly simplified preprocessing code like this:

```
def preprocess(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    c1_min, c1_max = df['col1'].min(), df['col1'].max()
    df['col1'] = 0 if c1_min == c1_max else (df['col1'] - c1_min) / (c1_max - c1_min)
    df['month'] = df['date'].dt.month
    df['comment'] = df['comment'].str.lower()
    return df
```

To add `typedframe` schema support for this transformation we will define two schema classes - for the input and for the output:

```
import numpy as np
from typedframe import TypedDataFrame, DATE_TIME_DTYPE

class MyRawData(TypedDataFrame):
    schema = {
        'col1': np.float64,
        'date': DATE_TIME_DTYPE,
        'comment': str,
    }


class PreprocessedData(MyRawData):
    schema = {
        'month': np.int8
    }
```

Then let's modify the `preprocess` function to take a typed wrapper `MyRawData` as input and return `PreprocessedData`:

```
def preprocess(data: MyRawData) -> PreprocessedData:
    df = data.df.copy()
    c1_min, c1_max = df['col1'].min(), df['col1'].max()
    df['col1'] = 0 if c1_min == c1_max else (df['col1'] - c1_min) / (c1_max - c1_min)
    df['month'] = df['date'].dt.month
```

(continues on next page)

(continued from previous page)

```
    df['comment'] = df['comment'].str.lower()
    return PreprocessedData.convert(df)
```

As you can see the actual DataFrame can be accessed via the `.df` attribute of the Typed DataFrame.

Now clients of the `preprocess` function can easily check what are the inputs and outputs without the need to look at its internals. And if there are some unforseen changes in the data an exception will be thrown before the actual function will be invoked.

Let's check:

```
import pandas as pd

df = pd.DataFrame({
  'col1': [0.1, 0.2],
  'date': ['2021-01-01', '2022-01-01'],
  'comment': ['foo', 'bar']
})
df.date = pd.to_datetime(df.date)

bad_df = pd.DataFrame({
  'col1': [1, 2],
  'comment': ['foo', 'bar']
})

df2 = preprocess(MyRawData(df))
df3 = preprocess(MyRawData(bad_df))
```

The first call was successful. But when we've tried to pass a wrong dataframe as input we've got the following error:

```
AssertionError: Dataframe doesn't match schema
Actual: {'col1': dtype('int64'), 'comment': dtype('O')}
Expected: {'col1': <class 'numpy.float64'>, 'date': dtype('<M8[ns]'), 'comment': <class
↪'object'>}
Difference: {('col1', <class 'numpy.float64'>), ('date', dtype('<M8[ns]'))}
```

CHAPTER

TWO

PROBLEMS WITH PANDAS DATAFRAME

Let's return the initial code example above. What's the problem here?

```
def preprocess(df: pd.DataFrame) -> pd.DataFrame:
```

Even when we have added type hints to our function, the user doesn't really know how he can use it. He must dig inside the code of the function to find out things like expected columns and their types. This violates on of the core software development principles - the encapsulation.

Pandas DataFrame is an open data type. It introduces a lot of implicit assumptions about the data. Let's explore some examples where one can easily overlook these implicit assumptions:

2.1 Required columns and data types:

```
df.grouby('state')['income'].mean()
```

The dataframe is expected to have `state` and `income` columns. `income` column must have a numeric type.

2.2 Index name and type

```
df.reset_index(inplace=True)
x = df['my_index']
```

It is expected that a dataframe has a named index with a name `my_index`.

2.3 Categorical columns

```
df3 = pd.merge(df1, df2, on='categorical_col')
```

The result above will differ based on whether a `categorical_col` in `df1` and `df2` has exactly the same set of categories or not.

All these scenarios above can lead to a variety of subtle bugs in our pipeline.

CHAPTER

THREE

THE CONCEPT OF TYPED DATAFRAME

A Typed DataFrame is a minimalistic wrapper on top of your pandas DataFrame. You create it by creating a subclass of a `TypedDataFrame` and defining `schema` static variable. Then you can wrap your DataFrame in it by passing it to your Typed DataFrame constructor. The constructor will do a runtime schema validation and the original dataframe can be accessed through `df` attribute of a wrapper.

This wrapper serves 2 purposes:

- Formal explicit documentation about dataframe assumptions. You can use your Typed DataFrame schema definition as a form of documentation to communicate your data interfaces to others. This works very well especially in combination with Python type hints.
- Runtime schema validation. In case of any data contracts violation you'll get an exception explaining the exact reason. If you guard your pipeline with such Typed DataFrames you'll be able to catch errors early - closer to the root causes.

CHAPTER

FOUR

FEATURES

4.1 Required Schema

You can define the required schema by passing a dictionary to a static variable `schema` of a `TypeFrame` subclass. The dictionary defines the mapping from a column name to a dtype:

```
class MyTable(TypedDataFrame):
   schema = {
      "col1": str,
      "col2": np.int32,
      "col3": ('foo', 'bar')
   }
```

4.2 Schema Inheritance

You can inherit one Typed DataFrame from another one.

The semantics of the inheritance relation is the same as with class methods and attributes in classic OOP. I.e. if Typed DataFrame A is a subclass of a Typed DataFrame B, all the schema requirements for B must also be held for A. In case of any conflicts, the schema defined in A takes a precedence.

```
class MyDataFrame(TypedDataFrame):
    schema = {
        'int_field': np.int16,
        'float_field': np.float64,
        'bool_field': bool,
        'str_field': str,
        'obj_field': object
    }


class InheritedDataFrame(MyDataFrame):
    schema = {
        'new_field': np.int64
    }
```

4.2.1 Multiple Inheritance

Multiple Inheritance is allowed. It has a "union" semantics.

```
class Root(TypedDataFrame):

    schema = {
        'root': bool
    }


class Left(Root):
    schema = {
        'left': bool
    }


class Right(Root):
    schema = {
        'root': object,
        'right': bool
    }


class Down(Left, Right):
    pass
```

4.3 Index Schema

You can specify schema for the index of the DataFrame. It's defined as a tuple of a dtype and a name which you assign to an `index_schema` static variable:

```
class IndexDataFrame(TypedDataFrame):
    schema = {
        'foo': bool
    }

    index_schema = ('bar', np.int32)
```

4.4 Optional Schema

You can specify optional columns in a schema definition. Optional column types will be checked only if present in a DataFrame. In case some optional column (or all of them) is missing no validation error will be raised. Besides that all columns from optional schema that are missing in a dataframe will be added with NaN values.

```
class DataFrameWithOptional(TypedDataFrame):
    schema = {
        'required': bool
    }
```

(continues on next page)

(continued from previous page)

```
    optional = {
        'optional': bool
    }
```

4.5 Convert Method

`TypedDataFrame` provides a convenient `convert` classmethod that tries to convert a given DataFrame to be compliant with a schema.

```
class IndexDataFrame(TypedDataFrame):
    schema = {
        'foo': bool
    }

    index_schema = ('bar', DATE_TIME_DTYPE)

df = pd.DataFrame({'foo': [True, False]},
                  index=pd.Series(['2021-06-03', '2021-05-31']))
data = IndexDataFrame.convert(df)
```

CHAPTER

FIVE

SUPPORTED TYPES

5.1 Integers

`np.int16`, `np.int32`, `np.int64`, etc.

5.2 Floats

`np.float16`, `np.float32`, `np.float64`, etc.

5.3 Boolean

`bool`

5.4 Python objects

`str`, `dict`, `list`, `object`

WARNING: no actual check is performed for Python objects. They are all considered to be of the same type `object`.

5.5 Categorical

Categorical dtype is specified as a tuple of categories. To avoid common categorical pitfalls categorical types are required to have an exact schema with all categories enumerated in the exact order.

```
class MyTable(TypedDataFrame):
   schema = {
      "col": ('foo', 'bar')
   }

df = pd.DataFrame({"col": ['foo', 'foo', 'bar']})
df.col = pd.Categorical(df.col, categories=('foo', 'bar'), ordered=True)
data = MyTable(df)
```

5.6 DateTime

`np.dtype('datetime64[ns]')`

`typedframe` library provides an alias for that also: `DATE_TIME_DTYPE`

5.6.1 UTC DateTime

`pd.DatetimeTZDtype('ns', pytz.UTC)`

`typedframe` library provides an alias for that also: `UTC_DATE_TIME_DTYPE`

CHAPTER

SIX

BEST PRACTICES TO USE TYPED DATAFRAME

What are the best places to use Typed DataFrame wrappers in your codebase?

Our experience with `typedframe` library in a number of projects has shown the following scenarios where it's use was justified the most:

6.1 Team Borders

Typed DataFrame helps to establish data contracts between teams. It also helps to spot the errors caused by miscommunication or inconsistent system evolution early. Whenever some dataset is being passed between teams it makes sense to define a Typed DataFrame class with its specification.

6.2 Public Functions and Methods

Typed DataFrame work especially well in combination with Python type hints. So a good place to use it is when you have a public function or method that takes as an argument / returns some pandas DataFrame.

6.3 Sources and Sinks of Data Pipelines

It is a good practice to provide schema definitions and runtime validation at the beginning and at the end of data pipelines. I.e. right after you read from the external storage and before you write to it. This is where Typed DataFrames can also be used.

CHAPTER

SEVEN

SIMILAR PROJECTS

- Great Expectations. It's a much more feature-rich library which allows data teams to do a lot of assertions about the data. `typedframe` is a more light-weight library which can be considered as a thin extension layer on top of pandas DataFrame.
- Marshmallow. A library for Python objects serialization and deserialization with schema validation. It's not integrated with pandas or numpy and focuses only on Python classes and builtin objects.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search