Castle Leonard
March 18, 2022
DATA 515

Data Versioner Project Report

Product Motivation

In the wild west of a Jupyter notebook any exploratory or experimental data task inevitably leads to skipping around in the notebook while tweaking and rerunning individual cells to satisfy some curiosity or derive some new metric and before too long your past discoveries and variations have disappeared. And even if you determinedly keep your notebook perfectly linear, the notebook soon becomes too long to scroll through and rediscovering a particular section starts to become the larger portion of the task.

To tackle this issue of limited real estate and reluctance to reign in one's free spirited and creative processes, data versioner is just the tool. It is a python package that offers git-like tracking of each notable stage, discovery, and variation of your data so your experiments and findings are preserved even as your notebook evolves.

Functional Spec

This tool is tailored towards data professionals who favor the quick and easy option of an interactive notebook, such as Jupyter, along with the familiarity of pandas for a frictionless data analysis or model development experience.

The primary use case is to unobtrusively facilitate users in documenting any promising outputs and findings as they go. This way every question asked or variation tried, however long ago, will be right after your fingertips the moment it becomes relevant again.

Component Spec

The package is made up of two main modules - the interface, dataversioner.py, and the data structure, committree.py.

The dataversioner.py module contains the DataVersioner class which is the layer between user's requests, potentially invalid inputs, and preconceived notions and the generic workings of a tree data structure. Its primary functions are validating user inputs and conveying requests to the data structure to update the tree or get information about prior commits.

The committree.py module contains the CommitTree class which manages the tree relationships including the successors of nodes (also referred to as commits), the root and current nodes, and furnishes details about the tree structure itself and accesses the information

of individual nodes on request. The data structure itself is actually the work of two dictionaries with the commit name as the key - one that tracks the successors of each node and therefore is traversable as a tree, in this case depth first, and the other holds each of the objects associated with those nodes. This choice enables O(1) look up times but does come with the restriction that each commit must have a unique name, which the CommitTree class expects and the DataVersioner interface guarantees.

Within the CommitTree class there is the DataCommit inner class which captures the minimum relevant information associated with a commit, including the name, a longer form message, a timestamp, and the data itself which can be any data type but we will assume is a pandas dataframe.

Design Decisions

Separation of concerns of interface and data structure

> The interfaces alone should need to worry about providing convenience understanding user expectations from prior experience in data processing and with versioning tools. Meanwhile the data object should be generalized and modular so as to be leveraged by the interface, but otherwise only care about functioning as a generic tree without concerning itself with unclean user input or the convenience or usefulness of its methods.

Editing and deleting data commits is not supported!

> The DataCommit class has only getter methods and by default returns a copy of the recorded data. Returning the original is available so as not to wastefully copy, but is only used for display or comparison purposes. This design choice was based on the idea of a scientific record as immutable, such as a physical lab notebook (references similarly in Joel Grus's Why I don't like Jupyter Notebooks) and reflected in github itself.

Commit outside dataframes

> The commit() method of the DataVersioner class will by default use the .data attribute as the data to be committed or, optionally, it can be overridden by passing in another dataframe. This allows the versioner to be out of the way and unobtrusive during analysis rather than adding the step of accessing the attribute of another class every time you need to reference that dataframe (imagine more than one filter and this could be repeated multiple times in one operation). This would become tiresome and clunky very quickly, so the alternative exists to work with you data as usual and then pass in the updated versions as ready.

Comparison

Sparkora (https://github.com/Spratiher9/Sparkora) is, in the creator's own words, "a Python library designed to automate the painful parts of exploratory data analysis in Apache Spark. The library contains convenient functions for data cleaning, feature selection & extraction,

visualization (databricks native), partitioning data for model validation, and versioning transformations of data."

Although deemphasized in the description, the data versioning is central to this tool's design. Similarly to dataversioner, it takes a dataframe as input to initialize and makes that data available as an attribute, however for pyspark rather than pandas. But very differently from dataversioner it offers automatic logging of changes to the dataframe, but only if the changes are performed through its own limited set of data manipulation functions. Additionally these functions have new APIs and patterns of behavior to become familiar with, and in the case of applying lambda functions, run a risk of being quite inefficient if certain pandas user defined functions are passed. And lastly, the self-documenting will be incomplete if and when functionality is needed outside of what has been implemented. Rather than follow this pattern, I opted to prioritize enabling the user to self document by adding a message field for further details, which sparkora does not have.

Additionally the "snapshot" mechanism seems to act more as organizer for data, much like files in a filing cabinet, than a versioner, since it returns the recorded dataframe, not a copy, foregoing strict record keeping. It also doesn't offer any protections around data lose when switching to another data snapshot with use_snapshot(). Alternatively, I require a default parameter, protect_data to be explicitly set to False before unsaved changes will be discarded. The filing cabinet method also doesn't track the lineage of datasets, so the context of their relationships to each other is unknown.

Extensibility

This design would readily extend to allow writing and restoring data-versioner objects with just the addition of serializer and deserializer of each of the DataCommit and CommitTree classes into protobuf or json. This would be quite valuable for returning to analysis over multiple sessions or for reference.

Other easy extensions would be to expand the DataCommit class to capture more information, such as comments of additional thoughts or context relating to that version of the data that may surface in later analysis.

A slightly trickier extension, but a valuable one, would be to have the DataVersioner class as generic base class, which could initialized into different subclasses based on the data type of the data object, so with little work pandas, pyspark, koalas, numpy could each be supported with limited additional functionality needed.