Report date 30/10/2024

Paper report:

**Overview:**

The paper introduces to problem of bugs that lead to data distribution error and skews during the data engineering and preprocessing pipeline in ML scenarios. Mlinspect is the introduced tool to revise ML pipelines and spot data distribution issues such as an IDE may spot code bugs. These kinds of bugs are manually fixed and spotted but a more complete and automatic system is needed.

Several data distribution bugs are listed such as: join operations that might change some user groups representation in the data distribution, dropped columns that should instead be used to measure model fairness, selections that may change proportions in group of data, usage of illegal features, using embedding for values that do not have an embedding…

To spot these kinds of bugs the authors have proposed mlinspect: a runtime program for lineage based inspection of python scripts. The tool does not require manual code instrumentation or annotation, it also supports scikit-learn and pandas operators and objects.

**DAG creation:**

Mlinspect extracts a DAG of the pipeline (based on a pipeline execution, not on all possible conditional branches) representing the pipeline.

Mlinspects uses monkey patching to add a DAG node whenever a supported relevant function call is made, it exploits the AST (Abstract Syntax Tree) from the Python parser to add the monkey patches.

The DAG is built after a code execution, meaning that if a piece of code is not executed due to conditional branches, that part of code will not be represented in the DAG.

The paper also discusses an algebraic definition of mlinspect DAG.

The DAG consists of a set of nodes, each node can be a different operator type, with the operator types being: Data Source, Unary Map, N-ary Map, Unary resampling, Join, Sink.

**Inspections and checks:**

Mlinspects supports scikit-learn and pandas built pipelines and analyzes those libraries method calls and objects via their respective backend.

Mlinspect relies on inspections and checks over the inspections. Inspections are snapshots/metadata of the data during different stages of the pipeline, namely, before and after every operator. Based on the checks requested by the ussr mlinspect will deploy given inspections. Inspections can be implemented with normal python code or with operations on dataframes, the latter method being less expressive but more efficient.

Most of the checks and inspections require annotations to be brought along the pipeline, such as data lineage or for instance the group a tuple belongs to even after the attribute that describes the group gets dropped.

**Mlinspect execution:**

* Preparation: determines the inspection that we need to execute in order to be able to verify the checks requested by the user.
* Instrumentation: Based on which inspections need to be used mlinspect patches all and only the relevant function calls. Mlinspect is also able to deal with nested pipelines and patch the indirect function calls. This part consists in instrumenting and making the code able to be logged. The instrumentation and monkey patching is accomplished via the Python AST of the user program.
* Execution: the DAG is built during the pipeline execution and thanks to the monkey patched functions. The necessary nodes are created and input, output and annotations are added to the node (the operator). Some inspections will be implemented by running Python code while others (more efficient) can be integrated in the main pipeline, the former ones must be implemented as dataframe operations.
* Results: the annotated results of each operators are stored together with the output object of the operator: to accomplish this, mlinspect uses monkey patches to add field to standard python objects or creates wrappers to wrap class (like numpy or pandas) objects and add the fields to store the annotation of that output. In this way the annotation will be stored as long the output data is stored, once the garbage collector will remove the operator output, also its annotations will be deleted.

**Experimental metrics and scenarios:**

In the experiments they measure:

* Runtime overhead (ms) of:
  + No mlinspect usage
  + Mlinspect code instrumentation
  + 1 inspection
  + 2 inspections
  + 3 inspections
* Runtime overhead (ms) of:
  + Empty inspection
  + Materialize inspection
  + Lineage
  + Histogram of 1 column
  + Histogram of 2 columns
  + Histogram of 3 columns
* Overhead (ms) of optimized vs non optimized inspections
* Overhead (ms) of no instrumentation, legacy instrumentation, improved instrumentation, improved instrumentation with code location tracking
* Exploratory study with questionnaire with expert after training and mlinspect demo usage

The scenarios they work with:

* Dataset and pipeline available on their repo but I couldn't find where they mention the dataset source in the paper.
* The tests have been run on their code only (tests available on the repo)

**Benchmarked against:**

During the quantitative benchmarking they've testes the tool overhead without comparing with similar tools.

During the qualitative benchmark they've compared mlinspect against:

* MLFlow
* noWorkflow
* Other minor tools they've mentioned but not fully compared against

**Limitations/future works:**

* Assisting data scientist in the mlinspect output analysis
* Additional backend for other ML libraries (Tensorflow, Apache SparkML, Beam)
* Complex to work in distributed systems such as when working with Beam and Spark operators
* Dataframe algebra may not be enough to also work with tensor operators

**Repo:**

<https://github.com/stefan-grafberger/mlinspect>

**Authors:**

**BIFOLD**

Stefan, Sebastian, Paul Groth and Julia Stoyanovich

**Year:**

2021