**Provenance for a Digital Pathology Research Pipeline**

This repository contains data, code, and other supplementary documents demonstrating application of the Provenance Backbone concept and underlying provenance model to a use case from digital pathology domain. Content of this repository consists of two parts:

1. A textual description of the research pipeline and application of the proposed provenance model usage (“Supplementary text” folder),
2. Implementation of provenance generation for computational parts of the use case (“XYZ” folder) -- a machine learning (ML) workflow used for cancer detection research. The repository also contains a portion of the ML workflow necessary to run the example. The implementation is described in this readme. file in two parts: a) part describing implementation of the ML workflow; b) and part describing provenance generation based on existing logs generated during the workflow execution and additional configuration file.

**The Machine Learning Workflow**

The ML workflow is implemented as a set of python scripts, and works with units called Experiments. In this context, an experiment consists of three parts – input data (WSI) preprocessing, AI model training, and AI model testing. An Experiment defines a logic of a job to be run using a configuration file. A configuration file is a nested JSON file describing:

* **Definitions –** defining what components (Data, Generator, Model, Callbacks, etc) are to be used in the experiment, and
* **Configurations –** defining parameters of the components.

Sample configuration files can be found in rationai/config/ directory. The workflow can be run using the provided Makefile files:

**Slide conversion**

make -f Makefile.convert run CONFIG\_FILE=rationai/config/prov\_converter\_config.json

**Execution of an experiment**

make -f Makefile.experiment run TRAIN\_CONFIG=rationai/config/prov\_train\_config.json TEST\_CONFIG=rationai/config/prov\_test\_config.json EVAL\_CONFIG=rationai/config/prov\_eval\_config.json EID\_PREFIX=PROV

alternatively, each experiment can be run individually

make -f Makefile.experiment setup train TRAIN\_CONFIG=rationai/config/prov\_train\_config.json EID\_PREFIX=PROV-TRAIN

Each makefile call creates a new experiment directory <EID\_PREFIX>-<EID\_HASH>, where EID\_PREFIX can be set during the Makefile call for easier experiment identification, and EID\_HASH is generated randomly to minimize experiment overwriting.

**Preprocessing (xml\_annot\_patcher.py)**

The preprocessing script prepares the WSIs to be processed by the ML workflow – splits the WSIs into two datasets and partitions each WSI into smaller regions, called patches, which are filtered and labeled. This preprocessing script can process several directories of histopathological slides using the openslide-python package. Each slide is the processed in the following manner:

1. A binary background mask is generated using Otsu's Thresholding method.
2. If an XML annotation file is provided a binary label mask is generated by drawing polygons on a canvas.
3. A sliding window technique is then applied on a background mask to generate patches. If a patch contains less tissue than a pre-defined threshold, the patch is discarded.
4. If a patch is not filtered by a background filter, it is assigned label according to the binary label mask.
5. Information about the patch (coordinates, label) is the added to a pandas table.
6. After all patches of a slide are processed, slide metadata (slide filepath, annotation filepath, etc) are added to the pandas table and the entire table is inserted into an index file (pandas HDFStore file).

**Training (slide\_train.py)**

Training script implements the ML model training. The training script first builds a data generator, and then divides training set represented as an index file into two disjunct sets: training set and validation set. The generator behaves as following:

1. A sampling structure is built from the contents of an index file.
2. During the training, the Generator samples a patch entry from the Sampler and passes it to an Extractor.
3. The Extractor accesses the correct slide and extracts an RGBA image from the coordinates within the sampled entry.
4. The extracted image is then augmented (if necessary) and normalized before being passed back to the Generator.
5. The Generator repeats this process for each sampled entry in a batch before passing the batch to the Model.

For both the training and the validation set a Generator is constructed. During the training the trained model iterates between two modes:

* **Training mode** - the model updates its own parameters (weights) based on how well it manages to predict a correct label for the patches.
* **Validation mode** - the model tracks its performance on the validation dataset, which has not been provided to the ML model before. It uses this information to create periodic checkpoints on every improvement or to stop the training process prematurely.

**Predictions (slide\_test.py)**

The script loads a previously trained model and executes it to create predictions for test slides (slides used neither for training nor validation of the model). The predictions for each slide are appended to its corresponding table as to new column and saved to disk as a new predictions HDFStore file.

**Eval (slide\_eval.py)**

During evaluation Evaluator objects are used to calculate metrics of interest (Accuracy, Precision, Recall, etc). Generator uses different Extractor during evaluation. Instead of accessing slides and retrieving images, the Extractor retrieves only those columns from the HDFStore tables that are required by the Evaluators.

**Provenance Generation**

Due to the heavy focus on configuration-driven approach, a significant portion of the experiments can be documented either by processing to the configuration file or referring to a github repository. This leaves us with information that is a result of a random process (splitting, sampling, checkpoints). For this example we have decided for a simple logging approach. We export key-value pairs of interest into a structured JSON log to be translated into finalized provenance information, which is expressed in accordance to the proposed provenance model.

In the resulting provenance, we document the following information about each step of the workflow:

* **Preprocessing** - no special logging is needed as the entire process is deterministic. As such only the configuration file, github repository URL, and the output file are necessary for provenanace generation. Only a hashed content of the output dataset will be presented in the final provenance.
* **Training** - in order to validate reproducibility of an experiment we log the states of the following objects: Datasource (hashed content of data split sets), Generator (hashed sampled entries for each epoch), Model (training and validation metric at the end of an epoch; checkpoints).
* **Predictions** - the states of a Datasource and the output file (predictions).
* **Evaluations** - the states of a Datasource and the results of Evaluators.

The corresponding log files and configuration files of an exemplary run of the ML workflow can be found in [outputs/experiment\_logs](https://gitlab.ics.muni.cz/rationai/crc_ml/histopat/-/tree/new-auto-provenance/outputs/experiment_logs).

**Logging**

During a regular run of an experiment, a structured JSON log is being constructed using a custom SummaryWriter object. Only a single copy with a given name can exist at any given time. Retrieveing a SummaryWriter object with the same name from multiple locations results in the same object similarly to standard logging.Logger.

Any key-value pair that we wish to keep track of must be set using the SummaryWriter .set() or .add() functions. The utility package rationai.utils.provenance contains additional helpful functions, for example for generating SHA256 hashes of pandas tables, pandas HDFStore, filepaths and directories.

log = SummaryWriter.getLogger('provenance')

log.set('level1', 'level2, 'level3', value='value')

log.set('level1', 'key', value=5)

log.to\_json(filepath)

# {

# 'level1': {

# 'level2': {

# 'level3': 'value'

# },

# 'key': 5

# }

# }

**Generation**

In order to parse the logs and generate resulting provenance according to the proposed model, we can call the Makefile.provenance file.

**Provenance Graph Generation**

make -f Makefile.provenance run TRAIN\_LOG=experiments/8c85b9321e00eeac082da2c3/prov\_train.log TEST\_LOG=experiments/8c85b9321e00eeac082da2c3/prov\_test.log EVAL\_LOG=experiments/8c85b9321e00eeac082da2c3/prov\_eval.log PREPROC\_LOG=data/prov\_preprocess.log

The result of this command are three provenance graphs serialized into a graphical format: prov-preprocessing, prov-training, and prov-evaluation.

The resulting provenance graphs serialized into a graphical format can be found in [outputs/provenance\_graphs](https://gitlab.ics.muni.cz/rationai/crc_ml/histopat/-/tree/new-auto-provenance/outputs/provenance_graphs). The underlying library for provenance handling would enable serialization of provenance into PROV-O (RDF), PROV-XML and PROV-JSON formats.