A.5 Developer Documentation

This is the developer documentation of the Rule Extraction Assistant (REA), contained in this repository. It includes: - What are the modules? - How do they interact? - Which software patterns have been used? - What software-design choices have been made - and why?

Project Structure

- the source code can be found in `40_Realisation/99_Final_System`
- The content of the folder is structured as follows:

Root

Name	Purpose
`Pipfile`	Pipfile to be used with pipenv to create the venv necessary for development
`requirements.txt`	List of required python packages. Can be used with the venv module
`.pre-commit-config.yaml`	Configuration for pre-commit hooks

Experiments folder

Name	Purpose
`experiments`	All experiments/studies conducted using the rea tool
`experiments/datasets`	Datasets used for experiments

Units Tests

Name	Purpose
`test`	Unit tests for the rea python module
`test/resources`	some resources (data, configuration files, ...) used for the unit tests

REA Program

Name	Purpose
`rea`	The python code of rea
`rea/rea.py`	Main python class and cli
`rea/configuration.py`	Json configuration reader/validator
`rea/data`	Data loading and preprocessing module
`rea/model`	Neural network training module
`rea/extraction`	Rule extraction module
`rea/extraction/alpa`	ALPA rule extraction algorithm
`rea/extraction/dnnre`	DNNRE rule extraction algorithm (implemented by sumaiyah)
`rea/rules`	DNF rule representation in python (implemented by sumaiyah)
`rea/evaluation`	Module and functions for the evaluation of extracted rules (fidelity, comprehensibility, accuracy, ...)

Dependencies

For the execution/building of the project, `Python 3.9` is required.

The following python packages are dependencies of rea: - `pandas` - `numpy` - `scikit-learn` - `category_encoders` - `tensorflow` - `rpy2` - `Jinja2`

The following non-python software is needed: - `R` (programming language/interpreter) - R `C50` package

The following are dependencies only necessary for development: - `pre-commit` - `flake8` - `jupyter`

3rd Party Code

The code for rea was based on a previous NEidI project implementation of Dnnre. However, except for the general pipeline structure and style of configuration, there is not much left from this implementation.

The code for dnnre, the rule representation and the rule evaluation is taken from sumaiyah. Some modifications were made, especially the evaluation code was heavily modified. The files were also moved to the proper modules. The documentation of the modifications can be found at the top of the respective files. We note that the repository of sumaiyah has no license attached, which prohibits the use of his code for non-private purposes in a legal sense. If this project was to be made public, the affected part would need to be replaced or the aforementioned author contacted to provide an (open-source) license (like Apache-2.0).

Software Design

REA follows a modular structure, where each module works independently to produce certain output files from given input files (which usually are ouputs of other modules), forming a pipeline-like data-flow. The pipleine can be configured using a file in `json` format following a certain structure/schema.

This will explain the software design, for information on how to use the modules/configuration, refer to the user documentation.

Pipeline Structure

The following modules are available: - `data` - `model` - `extraction` - `evaluation`

The **data** module is the foundation of the pipeline. It will load a provided dataset into a pandas dataframe and apply pre-processing steps, such as one-hot encoding for labels and weight of evidence encoding for categorical features. At the moment, only csv(`.csv`) or hdf5 (`.hdf5`, `.h5`) formats are supported. It is however easy to add support for another pandas-supported format by modifying the loading procedure in `data.py`.

The **model** module uses `tensorflow`s `keras` API to construct and train feed-forward or convolutional neural networks on a provided dataset. The constructed neural networks are general purpose and thus the ability to tailor them to a specific problem is limited. Basic parameters, like the number of hidden layers, kernel size, but also the learning rate, exponential decay confifguration and also dropout factor are exposed in the configuration file. For cases that need very well constructed and trained networks, it is recommended to supply a pre-trained model to the pipeline. This module can however be used for experimenting with the tool, wehere automatic creation of networks is advantageous.

The **extraction** module uses either the dnnre or alpa algorithm to extract rules from a trained tensorflow model and the provided data. It also collects some metrics, for example the execution time and memory consumption, on this process for later evaluation.

The **evaluation** module calculates some metrics on generated rules, the neural network and the extraction algorithm metrics and compiles them into markdown reports for the training and test datasets respectively.

Pipeline Configuration

To describe a (reproducible) pipeline run, configuration file(s) need to be supplied for each run. The minimal configuration accepted is the global section with the seed and logging level. A minimal *useful* configuration additonally contains at least of invoking the data module and possibly one of the three other modules. Each module has its own section in the configuration file, which follows the json format. In each section, the input and output paths are specified. The contents of the configuration file can be split into multiple files, allowing the combination and re-use of different module configurations. For example, one might want to only execute the data module once and then train two different networks on the pre-processed dataset.

API vs CLI

We provide a CLI, which accepts a list of configuration files and executes the specified pipeline run. Different runs can be achieved by executing the CLI multiple times with different configuration file(s). The generated output files of each module can then also be used by other programs (provided that they can read the format). Some (advanced) examples for the usage of the CLI can be found in the experiments folder.

Additionally, it is also possible to use the tool through its API by importing the rea module in other python projects. It is also possible to replace existing modules with custom implementations by inheriting from on of the module superclasses.

Development

Prerequisites

- Python 3.9
- R with the C50 library
  - to install, invoke the `R` REPL (root privileges might be necessary to write to `usr/lib`)
  - type `install.packages("C50")`
  - for this, you may need `build-essentials` (debian) or `gcc` and `gfortran`

Contributing

1. create a virtual python environment, for example `pipenv install --python 3.9`
2. activate the environment, for example `pipenv shell`
3. run `pre-commit install` to add the commit hooks to your git hooks
4. run `pre-commit run --all-files` once to apply all hooks for the first time
5. develop, develop, develop
6. stage and commit your changes; if you get an error during pre-commit, you need to stage the changes by pre-commit and commit again
7. push to the remote
8. goto (4)

Tests

Unit tests for the source code are provided in the `tests` folder. You can use the `run_all.sh` script to run them all. We use the standard `unittest` API provided by Python.

API Documentation

Docstrings are provided in the source code, so you can use `pydoc` or the tools of your IDE to read their documentation. Beyond that, for developers or other very interested individuals, a lot of source-code comments are provided for potentially unclear/complex sections.

In-Depth Information

`data` Folder

The `Data` class implements the preprocessing. Modules which use `Data` for data preprocessing inherit the `ProcessingModule` base class. Data uses `scikit-learn` to split input data into test and training sets and `sklearn.preprocessing.LabelEncoder` for one hot encoding of the class labels. To handle nominal or ordinal input data we added an interface to scikit-learns `category_encoders` package. This allows the use of numerous encodings, such as One Hot, Helmert, Leave One Out or ordinal. More encoding functions can be easily added (dictionary `Data.cat_encoder_methods`). As default encoding method for nominal attributes we implemented a custom, Numpy-based version of Weight Of Evidence encoding which can even handle non-binary classes. The implementation follows *Transformation of nominal features into numeric in supervised multi-class problems based on the weight of evidence parameter* (http://dx.doi.org/10.15439/2015F90).

`model` Folder

The `Model`class uses Tensorflow and Keras to generate an artificial neural network. There is support for Feed-Forward and Convolutional networks with associated parameters. The module adds a `keras.layers.Dropout` layer before the output layer per default to avoid overfitting. Convolutional networks contain of a series of `Conv2D` and `MaxPool2D(2, 2)` layers. Some learning parameters (learning rate, exponential decay, ...) are also exposed for configuration. Generally, the default parameters are chose to support common use-cases and need to be adapted using the configuration if necessary. When training the model, a callback is used to always only store (and potentially overwrite) the most accurate model in terms of validation accuracy. For this, a validation split is made on the fly by using the `validation_split` property of the `model.fit` function. Finally, some reports over the training history are generated to support an iterative training process.

`extraction` Folder

The `Extraction` class wraps rule extraction methods. It assures that time and memory usage of the algorithms are recorded in a comparable manner (`Extraction.metrics`). We support DNNRE and ALPA. More algorithms can be added by a corresponding folder and another `Extraction` instance method. The present rule representation in the `rules` folder should be used for a comparable evaluation. Memory measurements might exhibit some strange behaviour that could not be explained. Most likely, this occurs when using a GPU with tensorflow.

`DNNRE` The DNNRE implementation was completely taken over from the previous group, which in turn took over the code from sumaiyah. This is why the current DNNRE code only supports Feed-Forward neural networks.

`ALPA` Our Alpa implementation grew out of a review of the Java code for Weka . The program flow in the loop of the `alpa` method as well as the auxiliary methods are based on it. We eliminated some inefficiencies and peculiarities of the original implementation (like redundant network predictions). Since a pure Python implementation would produce a high computational overhead, we decided to implement the computationally intensive operations using the Numpy library. This was especially necessary for the nearest neighbor calculation in `Alpa.get_nearest` to match the quadratic effort. As a whitebox, we used the C5.0 decision tree, since it is also used in DNNRE and is considered one of the best. We use the same R interface to train the decision tree and generate rules from it. The `R` interface in turn uses a single threaded C implementation, which is freely available under the GNU General Public License. The computational effort within the Alpa loop mainly originates from training, and classifying with, the whitebox instances. To make the best whitebox instance reusable in the `Evaluation`, we save it with `pickle`. Other whiteboxes could be added analogously to `alpa_c5.py`. A function for model generation and a function for classifying new data would have to be implemented.

`rules` Folder

This folder contains the parts of the code of the sumaiyah DNNRE implementation which we use for a common basis of evaluation and rule extraction. The intention is to create rules with different extraction algorithms in the same format, so that the evaluation is not dependent on the method used.

`evaluation` Folder

The `evaluate_rules` folder contains the parts of the code of the sumaiyah DNNRE implementation which we use for a common evaluation of the extracted rules. We have made some parts more efficient by using Numpy operations and fixed bugs. Furthermore, we added functions for pretty printing of rules and attribute counting for evaluation purposes. The latter is especially intended for creating heatmaps when evaluating image data. The DNNRE `predict` function in `evalue_rules/predict.py` has a high Python overhead and takes a long time to predict high dimensional data such as images. This is why we added the possibility of using a prediction instance in `Evaluation.load_predict_instance`. We advise to use this feature for evaluation rules that are extracted with the alpa algorithm. For instance, only with the help of the `R C5.0` prediction instance we were able to predict the MNIST dataset with rules in reasonable time. The `Evaluation` class uses a `Jinja2` template to generate Markdown files with the evaluation output. The template is stored in `template/eval_templ.md`.

`configuration.py`

This file contains classes for easy access to and renaming of dictionary keys. This is useful for autocompletion and also for preventing KeyErrors. The `Configuration` class has validation methods for each of the REA modules. They are intended to be used before the modules are run. This is to catch as many usage errors as possible before the pipeline is actually executed.

`rea.py`

The `REA` class combines the modules into the pipeline. It also implements the two user interfaces (Cli and API). The CLI flags are generated in `__main__.py`, so that the `rea` module can be used with `python -m rea` command.

Pre-Commit

Pre-Commit is used for consistent collaboration. It ensures a unified code-style and enforces other constraints, like a filesize limit.