MIALab Documentation

Release 0.3.1

Fabian Balsiger, Alain Jungo, Yannick Suter, Elias Ruefenacht, Yo

Nov 20, 2023

GETTING STARTED

Welcome to the medical image analysis laboratory (MIALab). This repository contains all code you will need to get started with classical medical image analysis.

During the MIALab you will work on the task of brain tissue segmentation from magnetic resonance (MR) images (see data). We have set up an entire pipeline to solve this task, specifically:

- Pre-processing
- Registration
- Feature extraction
- Voxel-wise tissue classification
- Post-processing
- Evaluation

During the laboratory you will get to know the entire pipeline and investigate one of these pipeline elements in-depth (see pipeline). You will get to know and to use various libraries and software tools needed in the daily life as biomedical engineer or researcher in the medical image analysis domain (see tools).

Enjoy!

Found a bug or do you have suggestions? Open an issue or better submit a pull request.

CHAPTER

ONE

INSTALLATION

To start with the installation, download the Anaconda installer for your operating system with Python >= 3.8.

1.1 Initial Steps

These steps need to be performed for all operating systems.

1. Create a Github account (The free account is sufficient).
2. Fork the MIALab repository
   - Go to the MIALab repository
   - Fork the MIALab repository by clicking on the *Fork* button in the right upper corner
   - Follow the instructions of Github
   - Go to your MIALab fork (You are at the right location if the text in the left upper corner is of structure *[yourgithubaccount] / MIALab*)
   - Click on the green *Clone* button and copy the URL (`https://github.com/[yourgithubaccount]/MIALab.git`) shown. You will later use it for cloning your fork to your local machine and probably to UBELIX.
3. Continue with the operating system specific installation instructions

1.2 Operating System Specific Installation Steps

Select your operating system to get the corresponding installation steps:

The installation has been tested on Windows 10.

1. git installation
   - Download git and install
2. Clone your MIALab repository fork
   - Open "Git Bash"
   - :bash:`cd \path\to\where\you\want\the\code`
   - Clone the MIALab repository fork using the URL from the *Initial Steps*
   - :bash:`git clone https://github.com/[yourgithubaccount]/MIALab.git`
3. Anaconda installation

- Follow the instructions on the official website

4. Verify the installation
   - Open "Anaconda Prompt"
   - :bash:`conda list`, which should list all installed Anaconda packages
5. Create a new Python 3.8 environment with the name mialab (confirm with `y` when promoted during creation)
   - :bash:`conda create -n mialab python=3.8`
6. Activate the environment by
   - :bash:`conda activate mialab`
7. Install all required packages for the MIALab
   - :bash:`cd \path\to\MIALab\repository`
   - :bash:`pip install -r requirements.txt` will install all required packages
8. Execute the hello world to verify the installation
   - :bash:`python .\bin\hello_world.py`
9. Run Sphinx to create the documentation
   - :bash:`sphinx-build -b html .\docs .\docs\_build`
   - The documentation is now available under `.\docs\_build\index.html`

Run the following commands in the terminal (tested on Ubuntu 16.04 LTS and 18.04 LTS).

1. git installation
   - `sudo apt-get install git`
2. Clone your MIALab repository fork
   - `cd /path/to/where/you/want/the/code`
   - Clone the MIALab repository fork using the URL from the *Initial Steps*
   - `git clone https://github.com/[yourgithubaccount]/MIALab.git`
3. Run Anaconda installation script
   - Follow the instructions on the official website
   - No need to install the GUI packages
4. Verify the installation
   - `conda list`, which should list all installed Anaconda packages
5. Create a new Python 3.8 environment with the name mialab (confirm with `y` when promoted during creation)
   - `conda create -n mialab python=3.8`
6. Activate the environment by
   - `conda activate mialab`
7. Install all required packages for the MIALab
   - `cd /path/to/MIALab/repository`
   - `pip install -r requirements.txt` will install all required packages
8. Execute the hello world to verify the installation

- `python ./bin/hello_world.py`

9. Run Sphinx to create the documentation
    - `sphinx-build -b html ./docs ./docs/_build`
    - The documentation is now available under `./docs/_build/index.html`

The installation has not been tested.

1. git installation
    - Download git and install
2. Clone your MIALab repository fork
    - `cd /path/to/where/you/want/the/code`
    - Clone the MIALab repository fork using the URL from the *Initial Steps*
    - `git clone https://github.com/[yourgithubaccount]/MIALab.git`
3. Anaconda installation
    - Follow the instructions on the official website
4. Verify the installation
    - `conda list`, which should list all installed Anaconda packages
5. Create a new Python 3.8 environment with the name mialab (confirm with `y` when promoted during creation)
    - `conda create -n mialab python=3.8`
6. Activate the environment by
    - `conda activate mialab`
7. Install all required packages for the MIALab
    - `cd /path/to/MIALab/repository`
    - `pip install -r requirements.txt` will install all required packages
8. Execute the hello world to verify the installation
    - `python ./bin/hello_world.py`
9. Run Sphinx to create the documentation
    - `sphinx-build -b html ./docs ./docs/_build`
    - The documentation is now available under `./docs/_build/index.html`

CHAPTER

TWO

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

We recommend to use JetBrains PyCharm as IDE to program in Python. The community edition is open-source and sufficient for our purposes. Follow the instructions to install PyCharm.

To open the MIALab as project and to configure the Python interpreter do the following:

1. Launch PyCharm
2. Click Open (or File > Open)
    1. In the dialog navigate to `</path/to/where/you/have/the/code>/MIALab`
    2. Click OK
    3. MIALab is now open as PyCharm project (PyCharm created the `.idea` directory)
3. Click File > Settings… to open the settings dialog
    1. Navigate to Project: MIALab > Python Interpreter
    2. Select the Python interpreter `</path/to/your/anaconda/installation>/envs/mialab/bin/python` (on Linux and macOS) or `<\path\to\your\anaconda\installation>\envs\mialab\python.exe` (on Windows)

        Hint: If the interpreter is not available in the combo box, click the gear icon and choose Add Local and navigate the the files above

    3. Confirm by clicking OK
4. Open the `hello_world.py` (`bin` directory) in the navigator
    1. Right click in the editor > Run 'hello_world'
    2. Runs the hello_world and adds a configuration (see top right corner) to the project
    3. You can add configurations manually under Run > Edit Configurations…

You can watch the getting started videos to get accustomed with the interface.

2.1 Additional Configuration

To change the **docstring** format to Google do the following:

1. Click File > Settings… to open the settings dialog
2. Navigate to Tools > Python Integrated Tools
3. Select Google in the Docstring format dropbox
4. Click OK

To add a configuration for the **Sphinx documentation** do the following:

1. Click Run > Edit Configurations…
2. Click Add New Configuration (plus icon) > Python docs > Sphinx task
3. Edit the following
    1. Name (e.g. `docs`)
    2. Input to `</path/to/where/you/have/the/code>/MIALab/docs` (on Linux and macOS) or `<\path\to\where\you\have\the\code>\MIALab\docs` (on Windows)
    3. Output to `</path/to/where/you/have/the/code>/MIALab/docs/_build/html` (on Linux and macOS) or `<\path\to\where\you\have\the\code>\MIALab\docs\_build\html` (on Windows)
4. Click OK

CHAPTER

THREE

TOOLS

This list summarizes software tools and libraries we are going to use or we can recommend to use.

3.1 Python

- We use Python 3.8 or newer inside an Anaconda environment.

3.1.1 Libraries

- Numpy: Fundamental package for scientific computing with Python (a good & short tutorial)
- scikit-learn: An open-source software library for machine learning
- SimpleITK: Suite of software tools for image analysis (examples)
- matplotlib: Plotting library
- seaborn: Statistical data visualization
- pymia: pymia - generic and modular code for medical image analysis (developed by your lecturers!)
- Sphinx: A Python documentation generator

3.1.2 Integrated development environment (IDE)

- PyCharm: Strongly recommended (Community and Professional version are free for students) [Win, Mac, Linux]

3.2 Git

- git: required base of the version control system [Win, Mac, Linux]
- GitHub: free git repository hosting service

3.2.1 GUI Clients

- SourceTree: [Win, Mac]
- GitHub Desktop: [Win, Mac]
- GitKraken: [Win, Mac, Linux]
- Git Extensions: [Win]
- git-gui: Rudimentary GUI that comes with git [Win, Mac, Linux]
- or use the git client contained in PyCharm

3.3 Image Viewer

- ITK-SNAP: Simple tool to check results and do manual corrections [Win, Mac, Linux]
- 3D Slicer: More advanced tool [Win, Mac, Linux]
- ParaView: Visualization software [Win, Mac, Linux]

3.4 LaTeX

- LaTeX Guide: Installation, basics, elements, etc.
- Cheat Sheet

3.4.1 Editors

- Overleaf: Online LaTeX editor, collaborative writing, and publishing tool
- TeXstudio: Desktop editor [Win, Mac, Linux]
- Texmaker: Desktop editor [Win, Mac, Linux]
- or pick any (list of tex editors)

3.4.2 Distributions

(not needed with Overleaf)

- TeX Live: [Win, Linux]
- MiKTeX: [Win]
- MacTeX: [Mac]

3.5 Literature

3.5.1 Search

- Google Scholar
- Scopus
- Web of Science

3.5.2 Management

- Mendeley: Tool to organize your references (and create your bibliography)
- Mendeley Desktop: Client for Mendeley [Win, Mac, Linux]

3.6 Organization

- Trello: Tool to organize and prioritize your projects
- Jira: Tool to manage large projects
- Asana: Task management tool

3.7 Server Transactions

3.7.1 SSH Clients

- Putty: [Win, Linux]
- Termius: [Win, Linux, Mac]
- or pick any (list of SSH clients)

3.7.2 SFTP Clients

- WinSCP: [Win]
- FileZilla: [Win, Linux, Mac]
- or pick any (list of SFTP clients)

CHAPTER

FOUR

UBELIX HPC

The UBELIX (University of Bern Linux Cluster) is a HPC cluster of the University of Bern. During the MIALab course you can use UBELIX for computing your experiments. Beside this short guide, we recommend reading the official documentation of UBELIX.

Important: The access to the UBELIX HPC is only granted to students officially enrolled at the University of Bern.

4.1 Activation & Installation

The general activation and installation procedure is independent on your operating system. If you need assistance, please consult your lecturer.

1. Request UBELIX access with your student account
2. Install a SSH client on your machine (see *SSH Clients*)
3. Install a SFTP client on your machine (see *SFTP Clients*)
4. Wait until you get the confirmation from the UniBE IT Service department that your access request is approved
5. After receiving the account activation confirmation, establish a VPN connection to the university network
6. Login to UBELIX with your SSH client via `[campusaccount]@submit.unibe.ch` to validate that the account is working
7. Configure your SFTP client for UBELIX
    - File protocol: `SFTP`
    - Port: `22`
    - Host name: `submit.unibe.ch`
    - User name: `[campusaccount]`
    - Password: `[yoursecretpassword]`

Tip: Check the documentation of your SSH and SFTP client to enable autologin (e.g. Putty Autologin) or make configurations.

4.2 Project Handling

We expect you to work on your local machine and execute the experiments on UBELIX. To deploy your local code and its changes to UBELIX we recommend you to use Github. If you setup your MIALab fork correctly you can update the code on UBELIX by console without loosing information.

Important: It's crucial that you work on your own fork of the MIALab repository! You need to fork the MIALab repository before proceeding with the next steps.

Warning: Make sure that you do not add large-size files (>200kB, e.g. images, results) to your remote Github repository! Copy them manually from your local machine to UBELIX. For ignoring the appropriate folders / files modify your `.gitignore` file.

4.2.1 Clone Your Github Repository to UBELIX

This procedure needs to be performed once in order to clone the remote repository to UBELIX as a local repository.

Important: Make sure that you do **not** clone the original MIALab repository (`https://github.com/ubern-mia/MIALab.git`)! Your remote repository URL should have the following pattern: `https://github.com/[yourgithubaccount]/MIALab.git`.

---

1. Login via SSH to UBELIX (`[campusaccount]@submit.unibe.ch`)
2. Create a new directory for the MIALab: `mkdir MIALab`
3. Change to your new directory: `cd MIALab`
4. Clone your remote repository: `git clone https://github.com/[yourgithubaccount]/MIALab.git`
5. Login via SFTP to UBELIX
6. Upload the images and additional large files (>200kB) manually to the correct directories on UBELIX

4.2.2 Update Your Local UBELIX Repository from Github

This procedure needs to be performed when you want to update your code on UBELIX from Github.

Important: Make sure that you commit and push your changes on your local machine to Github before updating the UBELIX repository.

---

1. Login via SSH to UBELIX (`[campusaccount]@submit.unibe.ch`)
2. Change to your MIALab base directory (e.g. `./MIALab`): `cd MIALab`
3. Update the local UBELIX repository from Github: `git pull origin master`

Important: If you have multiple branches on Github modify the update command appropriately.

4.2.3 Setup Your UBELIX Python Environment

This procedure needs to be performed once before the first computation on UBELIX and after the cloning of your MIALab fork onto UBELIX. For detailed information we refer to the official UBELIX Python documentation.

Note: If you prefer an automatic generation of your Python environment you can run `bash scripts/create_ubelix_env.sh` from your MIALab directory on UBELIX and skip the following steps.

---

1. Login via SSH to UBELIX (`[campusaccount]@submit.unibe.ch`)
2. Load the Python module: `module load Anaconda3`
3. Prepare the environment for Python: `eval "$(conda shell.bash hook)"`
   - This command needs to be executed after each `module load Anaconda3`
   - Do **not** run `conda init` because it hardcodes environment variables and you need to rework the `.bashrc` file.
4. Create a new Python 3.8 environment with the name mialab (confirm with `y` when promoted during creation): `conda create -n mialab python=3.8`
5. Activate your new environment: `conda activate mialab`
6. Change to your MIALab base directory (e.g. `./MIALab`): `cd MIALab`
7. Install the dependencies of MIALab: `pip install -r requirements.txt`

Important: If you require additional Python packages later in your project you can add them to your `requirements.txt` file and re-execute the steps 2 - 3 and 5 - 7 in the previous procedure.

4.2.4 Transfer Large-Size Data from UBELIX to your Local Machine

This procedure is typically used after an experiment is finished and when you need to analyze the results locally on your machine.

1. Login via SFTP to UBELIX
2. Navigate to the appropriate directory
3. Copy the files to your local machine by drag-and-drop

4.3 Computation Job Handling

The UBELIX contains a job scheduler (SLURM) to assign computational resources to jobs and to handle priorities. The job scheduler is responsible that the necessary resources are available during the execution of the jobs and that no aborts are generated due to unavailable resources.

All normally privileged users on UBELIX have exclusively access to the submission node (`submit.unibe.ch`) where they submit their computational jobs via a job script. Writing a job script can be challenging at the beginning of your HPC life. Therefore, we prepared a template job script for you below. If you need any further assistance, consult the official UBELIX documentation or ask a lecturer.

4.3.1 Writing A Job Script

The job script specifies the resources you require for your computation. Because the experiments you will do in this course require more or less similar resources we prepared a template job script for you.

```
#!/bin/bash

# SLURM Settings
#SBATCH --job-name="GIVE_IT_A_NAME"
#SBATCH --time=24:00:00
#SBATCH --mem-per-cpu=128G
#SBATCH --partition=epyc2
#SBATCH --qos=job_epyc2
#SBATCH --mail-user=your.name@students.unibe.ch
#SBATCH --mail-type=ALL
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err

# Load Anaconda3
module load Anaconda3
eval "$(conda shell.bash hook)"

# Load your environment
conda activate mialab

# Run your code
srun python3 main_example_file.py
```

Important: Do **not** use the GPU partition if you do not use specific libraries with GPU support! Your code does not magically speed-up when running on a GPU partition. Furthermore, MIALab as it is does not make use of GPUs!

4.3.2 Submitting & Controlling A Job

The following procedure needs to be performed whenever you want to submit a computation job.

1. Write a job script or modify an existing one
2. Copy the job script to the correct location using the SFTP client
3. Submit the computation job by `sbatch [yourjobscriptname].sh`

Important: Be aware of the paths inside the job script! Use relative paths from the location of the job script.

Additional Useful Commands

- Monitor your jobs via `squeue --me`
- Cancel a job via `scancel [yourjobid]`

Important: Cancel jobs which contain errors such that other users can use the allocated resources.

CHAPTER

FIVE

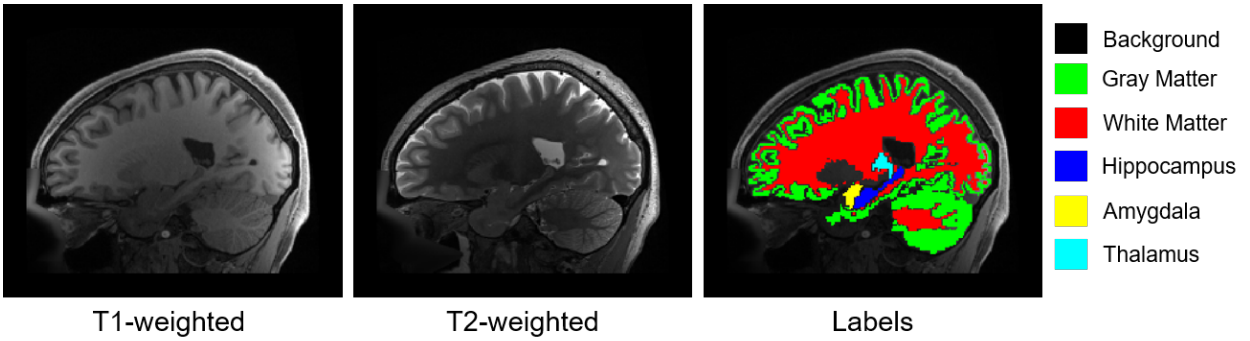
CLINICAL BACKGROUND

In the MIALab, we are segmenting structures of the human brain. We are thus focusing on the most prominent medical imaging analysis (MIA) task, segmentation, and do it in the most prominent area in MIA, the human brain, on magnetic resonance (MR) images. Segmenting brain structures from MR images is important, e.g., for the tracking of progression in neurodegenerative diseases by the atrophy of brain tissue[1]. Performing the segmentation task manually is very time-consuming, user-dependent, and costly[2]. Think about being a neuroradiologist who needs to segment the brain of every scanned patient. This is why we aim for an automated approach based on machine learning (ML).

The aim of the pipeline is to classify each voxel of a brain MR image in one of the following classes:

- 0: Background (or any other structures than the one listed below)
- 1: Cortical and cerebellar white matter
- 2: Cerebral and cerebellar cortex / grey matter
- 3: Hippocampus
- 4: Amygdala
- 5: Thalamus

An example sagittal image slice is shown in the figure below, where the label image (reference segmentation referred to as ground truth or simply labels) is shown next to the two available MR images (T1-weighted and T2-weighted).



[1] Pereira, S., Pinto, A., Oliveira, J., Mendrik, A. M., Correia, J. H., Silva, C. A.: Automatic brain tissue segmentation in MR images using Random Forests and Conditional Random Fields. Journal of Neuroscience Methods 270, 111-123, (2016). https://doi.org/10.1016/j.jneumeth.2016.06.017

[2] Porz, N., Bauer, S., Pica, A., Schucht, P., Beck, J., Verma, R.K., Slotboom, J., Reyes, M., Wiest, R.: Multi-Modal Glioblastoma Segmentation: Man versus Machine. PLoS ONE 9(5), (2014). https://doi.org/10.1371/journal.pone.0096873

5.1 References

CHAPTER

SIX

DATA

6.1 Medical Images

The dataset consists of 3 tesla head MR images of 30 unrelated healthy subjects from the Human Connectome Project (HCP) dataset of healthy volunteers[2]. For each subject, the following data is available:

- T1-weighted (T1w) MR image volume, not skull-stripped (but defaced for anonymization[3]), with a bias field correction
- T2-weighted (T2w) MR image volume, processed the same way as the T1w image
- Both modalities in native T1w subject-space
- The ground truth label map and brain mask in native subject-space
- Affine transformation to align the images to the atlas (see below)

The ground truth labels are generated by FreeSurfer 5.3 (e.g.,[4]) and are not manual expert annotations. As you will see when opening some example label maps, the automated labelling is not perfect. This is a common problem in the MIA domain, often real expert annotations are sparse and a "silver-standard" ground truth has to be used.

6.2 Atlas

The MR image and label files with `mni` prefix are registered to the MNI152 atlas using nonlinear FNIRT.

- T1-weighted atlas image: `mni_icbm152_t1_tal_nlin_sym_09a.nii.gz`
- T2-weighted atlas image: `mni_icbm152_t2_tal_nlin_sym_09a.nii.gz`
- Brain mask: `mni_icbm152_t1_tal_nlin_sym_09a_mask.nii.gz`

Add these files to the `./data/atlas/` directory.

[2] Van Essen, D.C., Smith, S.M., Barch, D.M., Behrens, T.E., Yacoub, E., Ugurbil, K. and Wu-Minn HCP Consortium, 2013. The WU-Minn human connectome project: an overview. Neuroimage, 80, pp.62-79.

[3] Milchenko, M. and Marcus, D., 2013. Obscuring surface anatomy in volumetric imaging data. Neuroinformatics, 11(1), pp.65-75.

[4] Fischl, B., Salat, D.H., Busa, E., Albert, M., Dieterich, M., Haselgrove, C., Van Der Kouwe, A., Killiany, R., Kennedy, D., Klaveness, S. and Montillo, A., 2002. Whole brain segmentation: automated labeling of neuroanatomical structures in the human brain. Neuron, 33(3), pp.341-355.

6.3 Random Forest Toy Example

The get a feeling of what a random forest, the type of machine learning classifier used to classify voxels in the brain tissues at interest, does, toy example data are provided. The toy example data files in the data directory (`exp1_n2.txt`, ...) are taken from the Sherwood library[1].

6.4 References

[1] Microsoft Research, Sherwood C++ and C# code library for decision forests, 2012. [Online]. Available: http://research.microsoft.com/en-us/downloads/52d5b9c3-a638-42a1-94a5-d549e2251728/. [Accessed: 08-Sep-2020].

CHAPTER

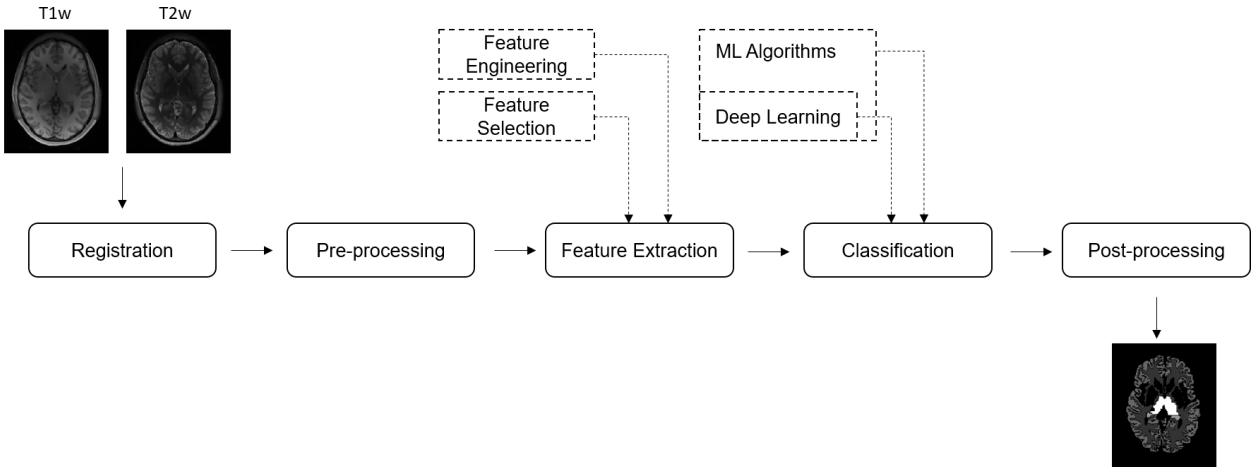
SEVEN

PIPELINE

The figure below shows our medical image analysis (MIA) pipeline with its single steps. Our pipeline has as input two magnetic resonance (MR) image slices (i.e., a T1-weighted (T1w) image slice and a T2-weighted (T2w) image slice) and a segmentation of the brain into the structures described previously (see *Clinical Background*). The pipeline itself consists of the following steps:

- Registration, which aims at aligning the two MR images
- Pre-processing, which aims at improving the image quality for our machine learning algorithm
- Feature extraction, which aims to extract meaningful features from the MR images for the subsequent classification
- Classification, which performs a voxel-wise tissue classification using the extracted features
- Post-processing, which aims to improve the classification

The dashed boxes indicate pre-steps or selections that influence a step. The provided experiments (see *Pre-processing* and others) correspond to boxes in the figure. Additionally, we will also have a look at the evaluation of such a pipeline.



An in-depth description of the concept of the pipeline with references for further reading can be found in[1].

[1] Pereira, S., Pinto, A., Oliveira, J., Mendrik, A. M., Correia, J. H., Silva, C. A.: Automatic brain tissue segmentation in MR images using Random Forests and Conditional Random Fields. Journal of Neuroscience Methods 270, 111-123, (2016). https://doi.org/10.1016/j.jneumeth.2016.06.017

7.1 References

CHAPTER

EIGHT

PRE-PROCESSING

Investigate the influence of pre-processing on the segmentation performance.

- Image smoothing / noise reduction
- Image normalization
- Histogram matching
- Skull stripping (separate the brain from the skull and other surrounding structures)

8.1 Materials

- `pymia.filtering.preprocessing`
- medpy.filter.IntensityRangeStandardization.IntensityRangeStandardization
- L. G. Nyúl, J. K. Udupa, and X. Zhang, New variants of a method of MRI scale standardization, IEEE Trans. Med. Imaging, vol. 19, no. 2, pp. 143–50, Feb. 2000.
- J.-P. Bergeest and F. Jäger, A Comparison of Five Methods for Signal Intensity Standardization in MRI, in Bildverarbeitung für die Medizin 2008, Berlin Heidelberg: Springer, 2008, pp. 36–40.
- scikit-learn: Pre-processing data

CHAPTER

NINE

REGISTRATION

What is the optimal setting to register the images to an atlas?

- Transformation type
- Metric type
- Optimizer type
- Deep learning for image registration

9.1 Materials

- `pymia.filtering.registration`
- P. Viola and W. M. I. Wells, Alignment by maximization of mutual information, Proc. IEEE Int. Conf. Comput. Vis., vol. 24, no. 2, pp. 16–23, 1995.
- P. Cattin and V. Roth, Biomedical Image Analysis, 2016. [Online]. Available: https://miac.unibas.ch/BIA/ [Accessed: 08-Sep-2020].
- M.-M. Rohé, M. Datar, T. Heimann, M. Sermesant, and X. Pennec, "SVF-Net: Learning Deformable Image Registration Using Shape Matching," in Medical Image Computing and Computer Assisted Intervention MICCAI 2017: 20th International Conference, Quebec City, QC, Canada, September 11-13, 2017, Proceedings, Part I, Springer International Publishing, 2017, pp. 266–274.
- SimpleITK Notebooks: See chapters 60-67
- ITK Software Guide, Book 2: In C++ but with a thorough description

9.2 Tools

There exist various tools for registration besides the implemented code for registration:

- 3D Slicer: Open source software which also includes registration.
- ANTs: Advanced Normalization Tools, which come with registration algorithms.
- NiftyReg: Rigid, affine and non-linear registration of medical images.
- SimpleElastix: An extension of SimpleITK.

CHAPTER

TEN

POST-PROCESSING

Can we leverage the segmentation performance by post-processing?

- Morphological operators
- Dense conditional random field (CRF)
- Manual user interaction (e.g., brushing)

10.1 Materials

- `mialab.filtering.postprocessing`, e.g. use `DenseCRF`
- P. Krähenbühl and V. Koltun, Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials, Advances in Neural Information Processing Systems, vol. 24, pp. 109-117, 2011.
- S. Nowozin and C. H. Lampert, Structured Learning and Prediction in Computer Vision, Foundations and Trends in Computer Graphics and Vision, vol. 6, pp. 185-365, 2010.
- P. Cattin, Image Segmentation, 2016. [Online]. Available: https://www.miac.unibas.ch/SIP/pdf/SIP-07-Segmentation.pdf [Accessed: 08-Sep-2020], see chapter 6 - Mathematical Morphology

10.1.1 Evaluation

Which metrics are suitable for our task? What is the influence of the validation procedure on the results?

- Metric types
- Influence of e.g. small structures
- Influence of validation procedure

10.2 Materials

- See `pymia.evaluation` package
- A. A. Taha and A. Hanbury, Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool, BMC Med. Imaging, vol. 15, no. 1, pp. 1–28, 2015.
- Cross-validation in machine learning

CHAPTER

ELEVEN

FEATURE ENGINEERING

What features could be used to improve our model?

- Investigate other features
  - Hemisphere feature
  - Filter banks
  - Histogram of oriented gradients (HOGs)
- 2-D / 3-D differences

11.1 Materials

- scikit-image feature module

11.1.1 Feature Selection

Can we reduce the number of features to decrease the model complexity and the computational burden.

- Decision forest feature importance
- Principal component analysis (PCA)
- Mutual information based feature selection

11.2 Materials

- scikit-learn: Dimensionality reduction
- Parallelized Mutual Information based Feature Selection
- H. Peng, F. Long, and C. Ding, Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 27, no. 8, pp. 1226-38, 2005.

CHAPTER

TWELVE

MACHINE LEARNING

Do other machine learning algorithms perform better on our task? Can we improve the segmentation performance by parameter tuning?

- Overfitting
- Parameter tuning (tree depth, forest size)
- Support Vector Machine (SVM)
- Variants of decision forests (e.g., gradient boosted trees)

12.1 Materials

- scikit-learn: Supervised learning
- A. Criminisi and J. Shotton, Decision Forests for Computer Vision and Medical Image Analysis, 1st ed. London: Springer, 2013.
- R. S. Olson, W. La Cava, Z. Mustahsan, A. Varik, and J. H. Moore, Data-driven Advice for Applying Machine Learning to Bioinformatics Problems, Aug. 2017.

12.1.1 Deep Learning

Deep learning has gained much attention in the last years outperforming methods such as decision forests. What is the performance of a deep learning method on our task?

- Implement a deep learning method

12.2 Materials

- Generic U-Net Tensorflow implementation for image segmentation
- U-Net PyTorch implementation for brain MRI
- O. Ronneberger, P. Fischer, and T. Brox, U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015.

CHAPTER

THIRTEEN

DATA (MIALAB.DATA PACKAGE)

Provides data representation.

13.1 Structure (mialab.data.structure module)

The data structure module holds model classes.

class mialab.data.structure.**BrainImage**(*id_: str*, *path: str*, *images: dict*, *transformation: SimpleITK.Transform*)

Represents a brain image.

__init__(*id_: str*, *path: str*, *images: dict*, *transformation: SimpleITK.Transform*)

Initializes a new instance of the BrainImage class.

Parameters

- **id** (*str*) – An identifier.
- **path** (*str*) – Full path to the image directory.
- **images** (*dict*) – The images, where the key is a *BrainImageTypes* and the value is a SimpleITK image.

class mialab.data.structure.**BrainImageTypes**(*value*)

Represents human readable image types.

BrainMask = 4

The brain mask image.

GroundTruth = 3

The ground truth image.

RegistrationTransform = 5

The registration transformation

T1w = 1

The T1-weighted image.

T2w = 2

The T2-weighted image.

CHAPTER

FOURTEEN

FILTERING AND MANIPULATION (MIALAB.FILTERING PACKAGE)

This package contains various image filters and image manipulation functions.

14.1 Pre-processing (mialab.filtering.preprocessing module)

The pre-processing module contains classes for image pre-processing.

Image pre-processing aims to improve the image quality (image intensities) for subsequent pipeline steps.

class mialab.filtering.preprocessing.**ImageNormalization**(**args: Any*, ***kwargs: Any*)

Represents a normalization filter.

__init__()

Initializes a new instance of the ImageNormalization class.

execute(*image: SimpleITK.Image*, *params: pymia.filtering.filter.FilterParams | None = None*) → SimpleITK.Image

Executes a normalization on an image.

Parameters

- **image** (*sitk.Image*) – The image.
- **params** (*FilterParams*) – The parameters (unused).

Returns
The normalized image.

Return type
sitk.Image

class mialab.filtering.preprocessing.**ImageRegistration**(**args: Any*, ***kwargs: Any*)

Represents a registration filter.

__init__()

Initializes a new instance of the ImageRegistration class.

execute(*image: SimpleITK.Image*, *params:* ImageRegistrationParameters *| None = None*) → SimpleITK.Image

Registers an image.

Parameters

- **image** (*sitk.Image*) – The image.
- **params** (ImageRegistrationParameters) – The registration parameters.

Returns
The registered image.

Return type
sitk.Image

class mialab.filtering.preprocessing.**ImageRegistrationParameters**(**args: Any*, ***kwargs: Any*)

Image registration parameters.

__init__(*atlas: SimpleITK.Image*, *transformation: SimpleITK.Transform*, *is_ground_truth: bool = False*)

Initializes a new instance of the ImageRegistrationParameters

Parameters

- **atlas** (*sitk.Image*) – The atlas image.
- **transformation** (*sitk.Transform*) – The transformation for registration.
- **is_ground_truth** (*bool*) – Indicates weather the registration is performed on the ground truth or not.

class mialab.filtering.preprocessing.**SkullStripping**(**args: Any*, ***kwargs: Any*)

Represents a skull-stripping filter.

__init__()

Initializes a new instance of the SkullStripping class.

execute(*image: SimpleITK.Image*, *params:* SkullStrippingParameters | *None = None*) → SimpleITK.Image

Executes a skull stripping on an image.

Parameters

- **image** (*sitk.Image*) – The image.
- **params** (SkullStrippingParameters) – The parameters with the brain mask.

Returns
The normalized image.

Return type
sitk.Image

class mialab.filtering.preprocessing.**SkullStrippingParameters**(**args: Any*, ***kwargs: Any*)

Skull-stripping parameters.

__init__(*img_mask: SimpleITK.Image*)

Initializes a new instance of the SkullStrippingParameters

Parameters
img_mask (*sitk.Image*) – The brain mask image.

14.2 Feature extraction (mialab.filtering.feature_extraction module)

The feature extraction module contains classes for feature extraction.

class mialab.filtering.feature_extraction.**AtlasCoordinates**(**args: Any*, ***kwargs: Any*)

Represents an atlas coordinates feature extractor.

__init__()

Initializes a new instance of the AtlasCoordinates class.

execute(*image: SimpleITK.Image*, *params: pymia.filtering.filter.FilterParams | None = None*) → SimpleITK.Image

Executes a atlas coordinates feature extractor on an image.

Parameters

- **image** (*sitk.Image*) – The image.
- **params** (*fltr.FilterParams*) – The parameters (unused).

Returns
The atlas coordinates image (a vector image with 3 components, which represent the physical x, y, z coordinates in mm).

Return type
sitk.Image

Raises
ValueError – If image is not 3-D.

class mialab.filtering.feature_extraction.**NeighborhoodFeatureExtractor**(**args: Any*, ***kwargs: Any*)

Represents a feature extractor filter, which works on a neighborhood.

__init__(*kernel=(3*, *3*, *3)*, *function_=<function first_order_texture_features_function>*)

Initializes a new instance of the NeighborhoodFeatureExtractor class.

execute(*image: SimpleITK.Image*, *params: pymia.filtering.filter.FilterParams | None = None*) → SimpleITK.Image

Executes a neighborhood feature extractor on an image.

Parameters

- **image** (*sitk.Image*) – The image.
- **params** (*fltr.FilterParams*) – The parameters (unused).

Returns
The normalized image.

Return type
sitk.Image

Raises
ValueError – If image is not 3-D.

class mialab.filtering.feature_extraction.**RandomizedTrainingMaskGenerator**

Represents a training mask generator.

A training mask is an image with intensity values 0 and 1, where 1 represents masked. Such a mask can be used to sample voxels for training.

static get_mask(*ground_truth: SimpleITK.Image*, *ground_truth_labels: list*, *label_percentages: list*, *background_mask: SimpleITK.Image | None = None*) → SimpleITK.Image

Gets a training mask.

Parameters

- **ground_truth** (*sitk.Image*) – The ground truth image.
- **ground_truth_labels** (*list of int*) – The ground truth labels, where 0=background, 1=label1, 2=label2, ..., e.g. [0, 1]
- **label_percentages** (*list of float*) – The percentage of voxels of a corresponding label to extract as mask, e.g. [0.2, 0.2].
- **background_mask** (*sitk.Image*) – A mask, where intensity 0 indicates voxels to exclude independent of the
- **label.** –

Returns
The training mask.

Return type
sitk.Image

mialab.filtering.feature_extraction.**first_order_texture_features_function**(*values*)

Calculates first-order texture features.

Parameters
values (*np.array*) – The values to calculate the first-order texture features from.

Returns

A vector containing the first-order texture features:

- mean
- variance
- sigma
- skewness
- kurtosis
- entropy
- energy
- snr
- min
- max
- range
- percentile10th
- percentile25th

- percentile50th
- percentile75th
- percentile90th

Return type
np.array

14.3 Post-processing (mialab.filtering.postprocessing module)

The post-processing module contains classes for image filtering mostly applied after a classification.

Image post-processing aims to alter images such that they depict a desired representation.

class mialab.filtering.postprocessing.**ImagePostProcessing**(**args: Any*, ***kwargs: Any*)

Represents a post-processing filter.

__init__()

Initializes a new instance of the ImagePostProcessing class.

execute(*image: SimpleITK.Image*, *params: pymia.filtering.filter.FilterParams | None = None*) → SimpleITK.Image

Registers an image.

Parameters

- **image** (*sitk.Image*) – The image.
- **params** (*FilterParams*) – The parameters.

Returns
The post-processed image.

Return type
sitk.Image

CHAPTER

FIFTEEN

UTILITIES (MIALAB.UTILITIES PACKAGE)

This package contains various classes and functions for the pipeline construction and execution.

15.1 The file access module (mialab.utilities.file_access_utilities)

This modules contains utility functions and classes for the access of the file system.

class mialab.utilities.file_access_utilities.**BrainImageFilePathGenerator**

Represents a brain image file path generator.

The generator is used to convert a human readable image identifier to an image file path, which allows to load the image.

__init__()

Initializes a new instance of the BrainImageFilePathGenerator class.

static get_full_file_path(*id_: str*, *root_dir: str*, *file_key*, *file_extension: str*) → str

Gets the full file path for an image.

Parameters

- **id** (*str*) – The image identification.
- **root_dir** (*str*) – The image' root directory.
- **file_key** (*object*) – A human readable identifier used to identify the image.
- **file_extension** (*str*) – The image' file extension.

Returns
The images' full file path.

Return type
str

class mialab.utilities.file_access_utilities.**DataDirectoryFilter**

Represents a data directory filter.

The filter is used to

__init__()

Initializes a new instance of the DataDirectoryFilter class.

static filter_directories(*dirs: List[str]*) → List[str]

Filters a list of directories.

Parameters
dirs (*List[str]*) – A list of directories.

Returns
The filtered list of directories.

Return type
List[str]

class mialab.utilities.file_access_utilities.**DirectoryFilter**

Represents an abstract directory filter.

This class is used in *FileSystemDataCrawler* to filter a list of directories.

abstract static filter_directories(*dirs: List[str]*) → List[str]

Filters a list of directories.

Parameters
dirs (*List[str]*) – A list of directories.

Returns
The filtered list of directories.

Return type
List[str]

class mialab.utilities.file_access_utilities.**FilePathGenerator**

Represents an abstract file path generator.

This class is used in *FileSystemDataCrawler* to convert a human readable data identifier to an data file path, which allows to load the data.

abstract static get_full_file_path(*id_: str*, *root_dir: str*, *file_key*, *file_extension: str*) → str

Gets the full file path for a data file.

Parameters

- **id** (*str*) – The data's identification.
- **root_dir** (*str*) – The data file's root directory.
- **file_key** (*object*) – A human readable identifier used to identify the data file.
- **file_extension** (*str*) – The data's file extension.

Returns
The data's full file path.

Return type
str

class mialab.utilities.file_access_utilities.**FileSystemDataCrawler**(*root_dir: str*, *file_keys: list*, *file_path_generator:* FilePathGenerator, *dir_filter:* DirectoryFilter | *None* = *None*, *file_extension: str* = *'.nii.gz'*)

Represents a file system data crawler.

Examples

Suppose we have the following directory structure:

```
/path/to/root_dir
    ./Patient1
        ./Image.mha
        ./GroundTruth.mha
        ./some_text_file.txt
    ./Patient2
        ./Image.mha
        ./GroundTruth.mha
        ./GroundTruthRater2.mha
    ./Atlas
        ./Atlas.mha
```

We can use the following code to load the images *Image.mha* and *GroundTruth.mha* in the directories *Patient1* and *Patient2*:

```
>>> class MyImgType(enum.Enum):
>>>     T1 = 1
>>>     GroundTruth = 2
>>>
>>> class MyFilePathGenerator(FilePathGenerator):
>>>     @staticmethod
>>>     def get_full_file_path(_id: str, root_dir: str, file_key, file_extension:␣
↪str) -> str:
>>>         if file_key == MyImgType.T1:
>>>             file_name = 'Image'
>>>         elif file_key == MyImgType.GroundTruth:
>>>             file_name = 'GroundTruth'
>>>         else:
>>>             raise ValueError('Unknown key')
>>>
>>>         return os.path.join(root_dir, file_name + file_extension)
>>>
>>> class MyDirFilter(DirectoryFilter):
>>>     @staticmethod
>>>     def filter_directories(dirs: t.List[str]) -> t.List[str]:
>>>         return sorted([dir_ for dir_ in dirs if dir_.lower().__contains__(
↪'patient')])
>>>
>>> crawler = FileSystemDataCrawler('/path/to/root_dir',
>>>                                 [MyImgType.T1, MyImgType.GroundTruth],
>>>                                 MyFilePathGenerator(),
>>>                                 MyDirFilter(),
>>>                                 '.mha')
>>> for id_, path in crawler.data.items():
>>>     print(id_, path)
Patient1 {'Patient1': '/path/to/root_dir/Patient1',
          <MyImgType.T1: 1>: '/path/to/root_dir/Patient1/Image.mha',
          <MyImgType.GroundTruth: 2>: '/path/to/root_dir/Patient1/GroundTruth.mha'}
Patient2 {'Patient2': '/path/to/root_dir/Patient2',
          <MyImgType.T1: 1>: '/path/to/root_dir/Patient2/Image.mha',
```

(continues on next page)

(continued from previous page)

```
        <MyImgType.GroundTruth: 2>: '/path/to/root_dir/Patient2/GroundTruth.mha'}
```

__init__(*root_dir: str*, *file_keys: list*, *file_path_generator:* FilePathGenerator, *dir_filter:* DirectoryFilter | *None = None*, *file_extension: str = '.nii.gz'*)

Initializes a new instance of the FileSystemDataCrawler class.

Parameters

- **root_dir** (*str*) – The path to the root directory, which contains subdirectories with the data.
- **file_keys** (*list*) – A list of objects, which represent human readable data identifiers (one identifier for each data file to crawl).
- **file_path_generator** (FilePathGenerator) – A file path generator, which converts a human readable data identifier to an data file path.
- **dir_filter** (DirectoryFilter) – A directory filter, which filters a list of directories.
- **file_extension** (*str*) – The data file extension (with or without dot).

15.2 The multi processor module (mialab.utilities.multi_processor)

Module for the management of multi-process function calls.

class mialab.utilities.multi_processor.**BrainImageToPicklableBridge**

A BrainImage to *PicklableBrainImage* bridge.

static convert(*brain_image:* BrainImage) → *PicklableBrainImage*

Converts a BrainImage to *PicklableBrainImage*.

Parameters
brain_image (BrainImage) – A brain image.

Returns
The pickable brain image.

Return type
PicklableBrainImage

class mialab.utilities.multi_processor.**DefaultPickleHelper**

Default pickle helper class

make_params_picklable(*params*)

Default function called to ensure that all parameters can be pickled before transferred to the new process. To be overwritten if non-picklable parameters are contained in params.

Parameters
params (*tuple*) – Parameters to be rendered picklable.

Returns
The modified parameters.

Return type
tuple

make_return_value_picklable(*ret_val*)

Default function called to ensure that all return values `ret_val` can be pickled before transferring back to the original process. To be overwritten if non-picklable objects are contained in `ret_val`.

Parameters
ret_val – Return values of the function executed in another process.

Returns
The modified return values.

recover_params(*params*)

Default function called to recover (from the pickle state) the original parameters in another process. To be overwritten if non-picklable parameters are contained in `params`.

Parameters
params (*tuple*) – Parameters to be recovered.

Returns
The recovered parameters.

Return type
tuple

recover_return_value(*ret_val*)

Default function called to ensure that all return values `ret_val` can be pickled before transferring back to the original process. To be overwritten if non-picklable objects are contained in `ret_val`.

Parameters
ret_val – Return values of the function executed in another process.

Returns
The modified return values.

class mialab.utilities.multi_processor.**MultiProcessor**

Class managing multiprocessing

static run(*fn: callable*, *param_list: iter*, *fn_kwargs: dict | None = None*, *pickle_helper_cls: type = <class 'mialab.utilities.multi_processor.DefaultPickleHelper'>*)

Executes the function `fn` in parallel (different processes) for each parameter in the parameter list.

Parameters

- **fn** (*callable*) – Function to be executed in another process.
- **param_list** (*List[tuple]*) – List containing the parameters for each `fn` call.
- **fn_kwargs** (*dict*) – kwargs for the `fn` function call.
- **pickle_helper_cls** – Class responsible for the pickling of the parameters

Returns
A list of all return values of the `fn` calls

Return type
list

class mialab.utilities.multi_processor.**PicklableAffineTransform**(*transform: SimpleITK.Transform*)

Represents a transformation that can be pickled.

get_sitk_transformation()

class mialab.utilities.multi_processor.**PicklableBrainImage**(*id_: str*, *path: str*, *np_images: dict*, *image_properties: pymia.data.conversion.ImageProperties*, *transform: SimpleITK.Transform*)

Represents a brain image that can be pickled.

__init__(*id_: str*, *path: str*, *np_images: dict*, *image_properties: pymia.data.conversion.ImageProperties*, *transform: SimpleITK.Transform*)

Initializes a new instance of the `BrainImage` class.

Parameters

- **id** (*str*) – An identifier.
- **path** (*str*) – Full path to the image directory.
- **np_images** (*dict*) – The images, where the key is a `BrainImageTypes` and the value is a numpy image.

class mialab.utilities.multi_processor.**PicklableToBrainImageBridge**

A *PicklableBrainImage* to `BrainImage` bridge.

static convert(*picklable_brain_image:* PicklableBrainImage) → *BrainImage*

Converts a *PicklableBrainImage* to `BrainImage`.

Parameters
picklable_brain_image (PicklableBrainImage) – A pickable brain image.

Returns
The brain image.

Return type
BrainImage

class mialab.utilities.multi_processor.**PostProcessingPickleHelper**

Post-processing pickle helper class

make_params_picklable(*params: Tuple[*BrainImage*, SimpleITK.Image, SimpleITK.Image, dict]*)

Ensures that all post-processing parameters can be pickled before transferred to the new process.

Parameters
params (*tuple*) – Post-processing parameters to be rendered picklable.

Returns
The modified post-processing parameters.

Return type
tuple

make_return_value_picklable(*ret_val: SimpleITK.Image*) → Tuple[ndarray, pymia.data.conversion.ImageProperties]

Ensures that all post-processing return values `ret_val` can be pickled before transferring back to the original process.

Parameters
ret_val (*sitk.Image*) – Return values of the post-processing function executed in another process.

Returns
The modified post-processing return values.

recover_params(*params: Tuple[*PicklableBrainImage*, ndarray, ndarray, dict]*)

Recovers (from the pickle state) the original post-processing parameters in another process.

Parameters
params (*tuple*) – Post-processing parameters to be recovered.

Returns
The recovered post-processing parameters.

Return type
tuple

recover_return_value(*ret_val: Tuple[ndarray, pymia.data.conversion.ImageProperties]*) → SimpleITK.Image

Recovers (from the pickle state) the original post-processing return values.

Parameters
ret_val – Post-processing return values to be recovered.

Returns
The recovered post-processing return values.

Return type
sitk.Image

class mialab.utilities.multi_processor.**PreProcessingPickleHelper**

Pre-processing pickle helper class

make_return_value_picklable(*ret_val:* BrainImage) → *PicklableBrainImage*

Ensures that all pre-processing return values `ret_val` can be pickled before transferring back to the original process.

Parameters
ret_val (BrainImage) – Return values of the pre-processing function executed in another process.

Returns
The modified pre-processing return values.

Return type
PicklableBrainImage

recover_return_value(*ret_val:* PicklableBrainImage) → *BrainImage*

Recovers (from the pickle state) the original pre-processing return values.

Parameters
ret_val (PicklableBrainImage) – Pre-processing return values to be recovered.

Returns
The recovered pre-processing return values.

Return type
BrainImage

15.3 The pipeline module (mialab.utilities.pipeline_utilities)

This module contains utility classes and functions.

class mialab.utilities.pipeline_utilities.**FeatureExtractor**(*img:* BrainImage, ***kwargs*)

Represents a feature extractor.

__init__(*img:* BrainImage, ***kwargs*)

Initializes a new instance of the FeatureExtractor class.

Parameters
img (structure.BrainImage) – The image to extract features from.

execute() → *BrainImage*

Extracts features from an image.

Returns
The image with extracted features.

Return type
structure.BrainImage

class mialab.utilities.pipeline_utilities.**FeatureImageTypes**(*value*)

Represents the feature image types.

ATLAS_COORD = 1

T1w_GRADIENT_INTENSITY = 3

T1w_INTENSITY = 2

T2w_GRADIENT_INTENSITY = 5

T2w_INTENSITY = 4

mialab.utilities.pipeline_utilities.**init_evaluator**() → pymia.evaluation.evaluator.Evaluator

Initializes an evaluator.

Returns
An evaluator.

Return type
eval.Evaluator

mialab.utilities.pipeline_utilities.**load_atlas_images**(*directory: str*)

Loads the T1 and T2 atlas images.

Parameters
directory (*str*) – The atlas data directory.

mialab.utilities.pipeline_utilities.**post_process**(*img:* BrainImage, *segmentation: SimpleITK.Image*, *probability: SimpleITK.Image*, ***kwargs*) → SimpleITK.Image

Post-processes a segmentation.

Parameters

- **img** (structure.BrainImage) – The image.
- **segmentation** (*sitk.Image*) – The segmentation (label image).

- **probability** (*sitk.Image*) – The probabilities images (a vector image).

Returns
The post-processed image.

Return type
sitk.Image

mialab.utilities.pipeline_utilities.**post_process_batch**(*brain_images: List[BrainImage]*, *segmentations: List[SimpleITK.Image]*, *probabilities: List[SimpleITK.Image]*, *post_process_params: dict | None = None*, *multi_process: bool = True*) → List[SimpleITK.Image]

Post-processes a batch of images.

Parameters

- **brain_images** (*List[structure.BrainImageTypes]*) – Original images that were used for the prediction.
- **segmentations** (*List[sitk.Image]*) – The predicted segmentation.
- **probabilities** (*List[sitk.Image]*) – The prediction probabilities.
- **post_process_params** (*dict*) – Post-processing parameters.
- **multi_process** (*bool*) – Whether to use the parallel processing on multiple cores or to run sequentially.

Returns
List of post-processed images

Return type
List[sitk.Image]

mialab.utilities.pipeline_utilities.**pre_process**(*id_: str*, *paths: dict*, ***kwargs*) → *BrainImage*

Loads and processes an image.

The processing includes:

- Registration
- Pre-processing
- Feature extraction

Parameters

- **id** (*str*) – An image identifier.
- **paths** (*dict*) – A dict, where the keys are an image identifier of type structure.BrainImageTypes and the values are paths to the images.

Return type
(*structure.BrainImage*)

mialab.utilities.pipeline_utilities.**pre_process_batch**(*data_batch: Dict[BrainImageTypes, BrainImage]*, *pre_process_params: dict | None = None*, *multi_process: bool = True*) → List[*BrainImage*]

Loads and pre-processes a batch of images.

The pre-processing includes:

- Registration
- Pre-processing
- Feature extraction

Parameters

- **data_batch** (*Dict[structure.BrainImageTypes, structure.BrainImage]*) – Batch of images to be processed.
- **pre_process_params** (*dict*) – Pre-processing parameters.
- **multi_process** (*bool*) – Whether to use the parallel processing on multiple cores or to run sequentially.

Returns
A list of images.

Return type
List[*structure.BrainImage*]

CHAPTER

SIXTEEN

INDICES

- genindex
- modindex

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

R

S

T