Technische Universität München
Fakultät für Informatik
Prof. Nassir Navab

WS 15/16
Exercise Sheet 1
October 10, 2015

Exercises in Machine learning in Medical Imaging

Exercise 1 Set up your GIT account

This practical course contains several coding exercises that must be carried on using a computer. In order to facilitate the developement of your algorithm and ensure fair evaluation we decided to use GIT and provide you with a repository that is shared with us, where you can keep your code throughout the whole duration of the MLMI course.

Please create an account on bitbucket (www.bitbucket.org) in order to be able to participate to the exercises. An account on bitbucket.org is FREE and if you use your TUM mail address to register, you will have even more advantages than the ones who register using other email address.

Each of you should:

- Create one account on bitbucket preferably using your TUM mail address.
- Send me an email with your own username on bitbucket (send it to: fausto.milletari@gmail.com).
- Accept the invitation to the repository I created for you and that will be used throughout the course.
- Checkout and pull from the master branch of the repository, that contains a skeleton code for the exercise we are going to solve today.

If you have windows or MacOs please download and familiarise also with sourcetree, which is a client for git. If you have linux please find, download and familiarise with a git client of your choice.

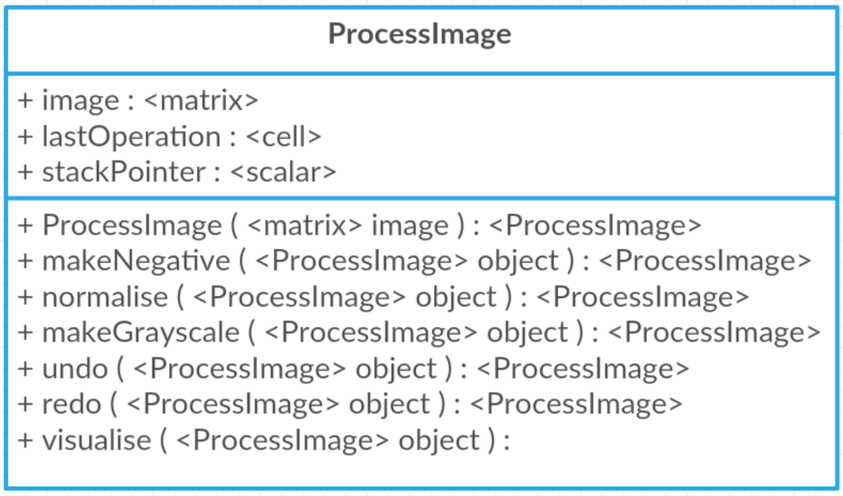
During your exercises you should often commit your code and push it to the repository. If the meaning of the words 'commit' and 'push' is unclear for you, please feel free to ask any tutor.

Exercise 2 Introduction to MATLAB

This is an introductive exercise that will not be graded and therefore will not contribute to your final score. We recommend you to solve the exercise and to submit it using GIT.

We will implement a very simple class to perform 'image processing'. The class diagram is shown in Figure 1 and it should be implemented in matlab. For your convenience you have been provided with a file containing the definitions of all interfaces you will be using in your class and a few lines of code that are going to help you with some of the functionalities that need to be implemented. Please DO NOT CHANGE any of the interfaces and do not declare any of its members private: the whole class is being tested automatically to assess the quality of your implementation.

In particular:



ProcessImage

+ image : <matrix>
+ lastOperation : <cell>
+ stackPointer : <scalar>

+ ProcessImage (<matrix> image) : <ProcessImage>
+ makeNegative (<ProcessImage> object) : <ProcessImage>
+ normalise (<ProcessImage> object) : <ProcessImage>
+ makeGrayscale (<ProcessImage> object) : <ProcessImage>
+ undo (<ProcessImage> object) : <ProcessImage>
+ redo (<ProcessImage> object) : <ProcessImage>
+ visualise (<ProcessImage> object) :

Figure 1: Your implementation of the class ProcessImage should follow this diagram

- The function ProcessImage is the constructor of the class. It takes only one argument: the image we want to process. It returns one argument: an initialised object whose type is ProcessImage. Please be sure to assign the image passed as argument to the class property *image*, to initialise *lastOperation* as a empty cell and to initialise *stackPointer* to 1. Be sure you are normalising the image to have only values comprised between 0 and 1.

- The function *makeNegative* subtracts 1 to the image, which is memorised in *obj.image*, and stores its absolute value in the variable *obj.image* itself. Please refer to the implementation you are already provided it to understand how to code the other member functions which are in some extent similar to this one.

- The function *normalise* subtracts the mean of the image to the image itself and divides the result by its standard deviation.

- The function *makeGrayscale* checks that the image has three channels and replaces it by:

$$0.21 * obj.image(:,:,1) + 0.72 * obj.image(:,:,2) + 0.07 * obj.image(:,:,3); \qquad (1)$$

- The function *undo* un-does the operation pointed by the *stackPointer*. Please be sure the *stackPointer* is always consistent with the content of the *lastOperation* variable.

- The function *redo* re-does the operation pointed by the *stackPointer*. Please be sure the *stackPointer* is always consistent with the content of the *lastOperation* variable.

- The function *visualise* shows the current content of the variable *image* via the function imshow.