Automating PLORAS - SYSTEM MANUAL

Contents

Introduction

This section is dedicated to explaining how our solution works and what is needed to modify in order to add extra functionalities. We have decided to divide our system into separate subsystems, each providing different sets of functionalities. In doing so it is much easier to change one particular part of the product without having to go through unrelated code in order to fix some unexpected dependencies. We have separated our system into sections responsible for the front end, the processing of images in MATLAB and also for the generation of predictions for a particular patient.

1 Front End

For this part of the project we have decided to use Django framework, which we have justified in the Research section of our project website. To summarise, it allowed for much easier communication with the other parts of the system.

Due to the way Django was created, most of the important functionalities are located in the views.py file that is responsible for validating input data as well as redirecting users to appropriate sections based on that. Methods that need to be mentioned here are scan_data(request) that use checkNumberOfFiles(zipFile) to check if the uploaded zip file is valid. Currently it checks if a user uploads a zip file with patient scans, it contains precisely 352 files which is the case with the BUCNI standard. However, we are aware that in the future that the system might be further developed and that part might need to be changed in order to accept scans from different machines.

Furthermore we have decided that the most appropriate location for storing scans uploaded by users as well as all intermediate steps would be in '/home/ploras/Patients/'. We have specified that location in the views.py as well as models.py. In order to change that, those files would need to be modified. Furthermore a potential new directory would require changing permission settings as well as ownership on it. This would need to be owned by the user apache and have the group as apache too. It would also need to have read, write and execution permissions on it. Finally Security-Enhanced Linux would require to have 'httpd_sys_rw_content_t' settings on the directory in order to allow the apache service to save files there.

When a user uploads a new set of scans, all the related information are

put into the user_interface_scandata table, which contains the handedness of the patient, date of birth, scan, and so on. It also holds a field for prediction that is going to be provided by the prediction script. However, what is more important, at same moment this data is introduced, the scan is also put into the user_interface_jobqueue table that is used by the processing script in order to pull jobs that still requires processing. That will be described in the appropriate section below.

Moreover, when accessing our local database we need to create objects for different tables. Their attributes are defined in the models.py file. Apart from the database, which is located in db.sqlite3 file, appropriate model in models.py needs to be redefined. It is important that the apache service is restarted after every such change.

The HTML and CSS are what give the application its design. It is consistent with the rest of the PLORAS website through the use of the UCL Indigo theme which can be seen in the base.html file. This acts as a template in which all the other HTML files are based in. The CSS can be found in the static folder for use with the HTML files. A 404.HTML page has been set up to ensure there are no broken webpages but instead they redirect to a custom error page. Each page in the application has an explanatory file name such as login.HTML. The code is a mixture of HTML with some links towards the Python files that allow the interactive content to be added. An example of this can be seen in the results.HTML page in which the HTML code merely serves the purpose of setting up the layout of the content.

2 MATLAB Processing

At the other end of the system we have MATLAB scripts that automate the required processing that was previously done manually using SPM8 and ALI. We have created a separate script file for each of the steps during the processing, so it is very easy to remove, add or change it.

To control them we have a single master script: PLORAS_ProcessingControl.m that calls all other functions in an appropriate order. It also defines two debug variables: processRawScans and generateSpreadsheet that are really only used if something goes wrong. For example, if one would not want to process all scans all over again or would not want to generate the spreadsheet with percentage damage of given regions. In order to add or remove a certain step, one just has to follow the pattern, taking for example lesion identification:

```
%LESION IDENTIFICATION:
try
    cd(scriptDirectory);
    PLORAS_AutomatedLesionIdentification(processedScans,processedSca-
    ns);
catch exception
    display(getReport(exception));
    return;
end
```

For each step, apart from the Dicom Import, we have used an inbuilt ALI batch system, so that for each step, it is clearly visible what kind of variables and constants are used. Taking for example Lesion Identification, we have:

matlabbatch{1}.spm.tools.ali.lesion_definition.step4directory = {outputDirectory};
matlabbatch{1}.spm.tools.ali.lesion_definition.step4binary_size = 100;

It is clear that the binary_size constant (that was declared inside ALI) is 100, and the output directory was set in the script to outputDirectory (which was declared in the master control script).

For generating the mentioned above spreadsheet, we have written appropriate functions in PLORAS_AutomatedTableCreation.m. We utilise here the GUI application written by Tom Hope. Our part is only responsible for creating appropriate inputs and then redirecting them to the application, which we have modified to create a .csv file rather than .xls, because it is easier to work on during the prediction generation process.

Furthermore each of the functions were clearly described in the header of each file, where their inputs and attributes are explained.

3 Prediction Generation

Finally, we have the prediction generation script that is located inside the script that is also responsible for connecting the MATLAB back end with the Django front end. The script here is being managed by a crontab scheduler. It is being called every 10 minutes to check whether there is something in the previously mentioned user_interface_jobqueue table. If it detects a

job, it sets a flag on it so that if another crontab instance is started before the previous one finished, it would skip it. It also checks whether the job it pulled has a 'failed' flag, so that it would try to pull another one in the queue. When the job is pulled, a MATLAB instance is created and then searchpaths are added. This is so that it would be able to access spm8 functions as well as ones created by us. If the scripts are moved, that part needs to be updated:

```
eng.addpath('/usr/local/MATLAB/R2015b/toolbox/spm8',nargout=0)
eng.addpath('/var/www/Team-32-System-Engineering-Project/Matlab-
AutomationScripts',nargout=0)
eng.addpath('/var/www/Team-32-System-Engineering-Project/Matlab-
AutomationScripts/Spreadsheet',nargout=0)
```

Where eng is the MATLAB engine object created few lines above.

Furthermore, we have introduced two kinds of logs for the process. When given processing fails or finishes, it appends this information to jobQueue.log together with the timestamp. While given processing is in progress, the information on the current progress is located in detailedLog.log. Rather than appending it, we overwrite it every time due to the fact that this file would grow in size really quickly as SPM generates a lot of text.

4 Prediction Script

This portion of code which can be found in runProcessing.py, is used for producing the prediction output given the processed scores for damage in the regions of interest. What follows is an explanation of what each section does, and how to adjust it to suit your needs and any changes.

Line 31-38: read_patient_scores

This function takes the scores for damage to the regions of interest out of the file that MATLAB writes them into, and imports them to be used within the script. The only change you need worry about here is on line 36 in which the tokens 'scores[2:7]' need to be adjusted if the number of examined regions of interest changes. By default there were 5 regions of interest, so in order to increase this to 6 regions that statement would need adjusting to 'scores[2:8]'.

Line 40-41: find_sem_mem_score

This function is not used due to the fact that we cannot be sure that the patient is already in the database - it defaults to assuming there is no damage to the semantic memory of the patient and a message is given with the prediction saying that semantic memory damage will affect the quality of the prediction.

Line 43-44: is_bilateral

This function is intended to find if the patient has bilateral damage, as with find_sem_mem_score it has not been implemented because we cannot ensure the patient is in the database.

Line 46-95: generate_prediction_string

This function is the one that does all the work. The first line (line 47) is there to add a blank element to the start of the list so that the regions begin counting from 1 rather than 0 for ease of maintainability as constantly subtracting 1 mentally from each region number could easily lead to slip ups. Lines 51-57 are where the different possible predictions lie. Each string is assigned to a variable name mainly for readability, when adding new prediction possibilities I would recommend doing the same. From here onwards to the end of the function the predictions themselves are generated, based on the original set of predictions the first 'stage' is from line 59-71, as you can see the logic is formed purely of if statements; the script will only fall through to stage 2 if the patient does not fit into any of the categories from the first stage. The same applies for stage 2, and stage 3 will always return a string no matter what input it receives.