DoCure

The Smart Way to Maintain your Health Records...

TECHNICAL REFERENCE MANUAL

Contents

About

Health is an important aspect of every life. For all the necessary records related to health, medical records are employed. Traditionally these are in the form of paper and can often be misplaced, or suffer from tear and damage DoCure aims at resolving this problem by using an Electronic Health Record system that scans and stores the medical report data on the database for anytime access from the user. Users can directly upload their report in the form of images or PDFs, that are processed and the data is stored in the database. When the user wants to access this data, they can directly see the data as well as several analysis that are available on the platform. The proposed approach is designed keeping in mind the different technological intricacies involved to provide a fast, secure, and most importantly a very user-friendly application that can be operated with ease.

DoCure

General Overview of Source Code

The DoCure Web Application is built using the Django Framework v3.1.7. Django is a high-level Python Web framework that allows rapid development and clean, pragmatic design. It is fast, secure and scalable.

The OCR used for medical report images are the Pytesseract OCR engine and Nanonets OCR Engine.

The data extraction activities are primarily supported by a deep learning model which is configured using Spacy models along with regular expressions.

The code is managed on GitHub which helps in version control and collaboration between developers. Further, it is hosted on Heroku PaaS which includes database connectivity of Heroku dbSQLite3. GitHub Repository can be found here –

https://github.com/Tirth11/Hack_DoCure

(It's a private repository and will require prior access before initiating work). The documentation further assumes that you are working on Windows OS (Windows 10+) and using Visual Studio Code as your editor. You can use settings of your own but the instruction steps may defer and need to figure them by yourself. There are several references attached in this document. However, trusting/using the references is solely at your discretion.

Source code contributors:

Anushka Darade

Tirth Thaker

Dhairya Umrania

Utsav Parekh

Getting Started

With necessary prior permissions, the project can be cloned using this link:

https://github.com/smart-girll/DoCure

After cloning the Git Repository, unzip the folder to a working directory where you want to work with this project.

1. Installing Virtual Environment

Install a virtual environment to work with the project. This can be done using this reference. In your integrated terminal type:

```
pip install virtualenv
```

Post that create your virtual environment using:

```
virtualenv your_env_name
```

After that enable the virtual environment using script:

```
your_env_name/Scripts/activate.bat
```

2. Installing Dependencies

After setting up the Virtual Environment, the first task is to install all the libraries, extensions and dependencies that is required to run the application. For this run the following line inside your virtual environment:

```
pip install –r requirements.txt
```

This will install all the dependencies in your working environment. In case any new dependencies are installed, use the following command to save them in requirements.txt

```
pip freeze > requirements.txt
```

DON'T ADD ANYTHING MANUALLY TO THE REQUIREMENTS.TXT FILE OR EDIT IT IN ANY WAY WITHOUT USING SHELL COMMANDS.

3. Understanding the File Directory Structure

FOLDER/FILE NAME	DESCRIPTION
MEDIA	Includes media files such as logo, user profile images, etc.
DOCURE	This includes the core Django project files such as settings.py amongst others. For using additional services like emailing, configuring database connectivity, hosting configuration, communication server/protocols, this is where changes need to be made.
HOME	All templates for pages on the web application can be found here. Django models, views, urls, etc. files are present here.
STATIC	Static files including .css, .js, images used inside the application, are saved in the static folder.
PROCFILE	Heroku apps include a Procfile that specifies the commands that are executed by the app on startup. Details for hosting the web service is written here.
MANAGE.PY	The file is the driver code which calls settings in miniproject and initiates the server for using the web application.
REQUIREMENTS.TXT	Includes dependencies which are necessary to run the program.
RUNTIME.TXT	Includes the python's runtime environment information.
.GITIGNORE, DB.SQLITE3	These files are the configuration files for either GitHub or Heroku PaaS which includes instructions to not include certain file types which may occupy unnecessary storage.

4. Initial Test Run on Localhost Server:

Type the following command in the terminal to initiate Localhost Server:

```
python manage.py runserver
```

This will initialize the server for gevent which will load files/configurations to run the application.

<u>NOTE:</u> THE CODE NEEDS SEVERAL CHANGES BEFORE HOSTING IT ON LOCALHOST SINCE SERVER CONFIGURATION FILES, PYTESSERACT FILES AND OTHER NECESSARY FILES DIFFER WHEN HOSTED ON HEROKU PAAS.

5. Steps to Host on Heroku PaaS (Django application with Postgres as Backend)

Step 1: Install and configure the Heroku CLI

Step 2: Create and activate the virtual environment (Done already)

Step 3: Install and add the following dependencies:

```
pip install django gunicorn whitenoise dj-database-url psycopg2
```

Step 4: Create a Procfile with the following contents:

```
web: gunicorn DoCure.wsgi --log-file –
```

Step 5: Freeze the requirements.txt file using the command given before

Step 6: Create a runtime.txt file with any of the supported runtimes by Heroku

Step 7: Initialize/connect to a git repository and add the .gitignore file.

git init

touch .gitignore

--Add following files—

*.log *.pot *.pyc __pycache__/ local_settings.py db.sqlite3 venv/ git add . git commit –m “message”

Step 8: Login to Heroku terminal

heroku login

heroku create <nameOfApp>

Step 9: Now let’s modify the settings.py file.

• Modify allowed hosts by adding nameofapp.herokuapp.com

ALLOWED_HOSTS = ['0.0.0.0', 'localhost', '127.0.0.1', 'nameofapp.herokuapp.com']

Set environment variables (SECRET_KEY and others)

SECRET_KEY = os.environ.get('SECRET_KEY')

EMAIL_HOST_USER = os.environ.get('EMAIL_HOST_USER')

EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_HOST_PASSWORD') and running in the terminal heroku config:set SECRET_KEY=secretkey

Set DEBUG = False and you could use heroku logs for debugging.

• Modify INSTALLED_APPS by adding whitenoise.runserver_nostatic , also MIDDLEWARE by adding 'whitenoise.middleware.WhiteNoiseMiddleware', and add STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

• Add import dj_database_url at the top. After DATABASES add db_from_env = dj_database_url.config(conn_max_age=600)
DATABASES['default'].update(db_from_env)

- Also ensure you file has following variables STATIC_URL , STATIC_ROOT , STATICFILES_DIRS accordingly. Just for example my file has STATIC_URL = '/static/'#location where django collect all static files

STATIC_ROOT = os.path.join(BASE_DIR,'static')# location where you will store your static files STATICFILES_DIRS = [os.path.join(BASE_DIR,'project_name/static')]

- Also ensure the media file variables.

MEDIA_ROOT = os.path.join(BASE_DIR,'media')

MEDIA_URL = '/media/'

- Commit and save changes in git by git add . and git commit -m "change settings

Step 10: Adding and configuring dbSQLite3

Now lets push local database to herokuDB

Step 11: Disable Collectstatic and push the files to Heroku heroku config:

set DISABLE_COLLECTSTATIC=1 git push heroku master

Step 12: Open the deployed app heroku open

The application is deployed at: https://docure-ehr.herokuapp.com/

Making Changes & Review

1. Adding Functionality for more report types

To add usage for more reports, we have prepared a pipeline which can be followed. The first step is to collect reports of the type you desire to add, there is no fixed number for the number of reports, but with more reports, the text extraction model will be trained better. After collecting the reports, the raw text of the reports need to be extracted. For that we have provided a jupyter notebook, which contains the pytesseract OCR, which is extracting the raw text from the report images and storing the text in .txt files that are named according to the number in which they appear in the image folder, and then they are stored in a directory of your choice. If you want to use some other OCR, you can use that too.

After the raw text has been extracted from the images, the txt files have to be checked if the data has been correctly identified because there could be cases where the text has not been detected correctly due various reasons such as bad image quality, bad lighting, OCR mistakes, incorrect interpretation, etc. The txt files with bad text must be filtered out and only those which are correctly identified must be kept.

Next step is to train the text mining model. For this we have used Spacy, this can be found in the cbc_ml.ipynb jupyter file. But before training the model, a rule based dictionary approach must be done to get the desired text data entities and their spans in corresponding text, because spacy needs the training data in the format:

{text, {entities: (label1, span), (label2, span)}

To do this first the parameters of the report that are to be extracted must be listed down and for each separate entity a JSON file must be created with all the names that the entity could take according to the user.

Then open the cbc_ml.ipynb jupyter file and read the JSON files to get the entities and their patterns, the stems of the words in the JSON files are modified a bit to cover more patterns for the dictionary approach.

After all the patterns are mapped out, they have to be put in the following format:

[{label1: entity1, pattern: pattern1},

{label1: entity2, pattern: pattern2},

…]

This is done using the spacy entityruler which can be found in cbc.ml_ipynb jupyter file.

After the entity ruler has been used to map out the patterns, the dictionary approach can be used to get the entities and their spans from the text corpus collected from the report images.

Once the entities and their spans are collected they are to be put in the format that we defined previously: {text, {entities: (label1, span), (label2, span)}

This can be done using the function provided in the jupyter file.

Once all the txt files from the text corpus have been parsed, the data will be created in the format mentioned above. This data can be split into training, validation and testing datasets. The training and validation datasets need to be converted into .spacy format, which can be done using the save_data function in the jupyter file.

After the training and validation datasets have been saved in the .spacy format, the training of the model has to be done on the command line.

The commands and steps for training can be referenced from the following site:

https://spacy.io/usage/training

Here, a Spacy NER base.cfg file must be downloaded, and the paths of the training and validation datasets must be provided.

The base.cfg file must be converted to a config.cfg file using the following command on command line:

```
python -m spacy init fill-config base_config.cfg config.cfg
```

After this, the training must be done using the following command:

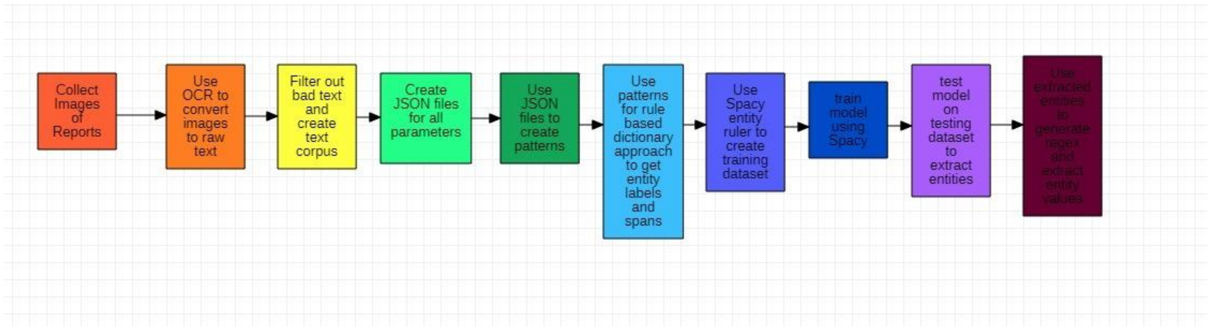
```
python -m spacy train config.cfg --output ./output
```

Once the model is trained, the training scores will be visible in the command prompt, and the trained model will be saved to the folder named "output".

That model can be called and tested on the testing dataset. The function for generating the confusion matrix and calculating the accuracy on the testing data is given in the jupyter file.

After looking at the metrics, if the model is well trained, the next step is to use those extracted entities to create a regex expressions to extract their corresponding values. The code for generating the regex is given in the jupyter file. Once the regex for the specific parameter is created the value of that parameter from the report can be extracted and stored in a variable.

This pipeline can be repeated for as many report types as required provided the data is sufficiently available.


Collect Images of Reports
Use OCR to convert images to raw text
Filter out bad text and create text corpus
Create JSON files for all parameters
Use JSON files to create patterns
Use patterns for rule based dictionary approach to get entity labels and spans
Use Spacy entity ruler to create training dataset
train model using Spacy
test model on testing dataset to extract entities
Use extracted entities to generate regex and extract entity values

Final Comments

NOTE FOR REVIEW OF CHANGES AND PUSHING/COMITTING CHANGES:

All changes made to code needs a detailed review before testing it on Localhost. Further, if the logic works, the code needs to be tested as per procedure before pushing this code on GitHub repository's main branch. If everything works well and clears test cases, then the code can be pushed to GitHub repository and then after making server changes can be pushed to Heroku. To successfully close the update, a test procedure needs to be carried out for the hosted application as well.