Training of OCR and NER models for Medicine name identification

Shubham Miglani

North Carolina State University

Abstract

This project investigates the task of medicine name recognition for Drug Label using Optical Character Recognition and Named entity recognition on device. The current method used by Wizeview involves using cloud services but using these cloud-based services can have a significant delay so the OCR and NER models were required to run on device. For the NER task, first dataset generation was discussed. After that various models such as Memory tagger, Random forest, Sequence tagging using LSTMs were trained and their performance and issues discussed. Neural network models were found ineffective for this task due to a class imbalance and lack of sentence structure problem whereas a memory or dictionary-based model were found more effective. For the OCR task, first dataset generation and training tesseract OCR was discussed. Various factors affecting the efficiency of the OCR such as configurations parameters, preprocessing techniques were analyzed, and their results discussed. The final model had an accuracy of 76% with an average time of 1 second per image as compared to the baseline tesseract with 50% accuracy and an average time of 2.24 second per image.

Table of Contents

Chapter 1 - Introduction

This project is an extension of the work being currently done by the Wizeview with focus on medication management. The current workflow used by Wizeview for medication management is as follows. The image clicked by the Wizeview app on a mobile device of a medication label is sent to the medicines API [1] developed by them. The medicines API accepts an input image URL along with other information regarding patient and visit ids. The information inside the API is processed in two steps.

- First Amazon's Rekognition service is called which is used to convert the image to text.
- This text is fed into an IBM Watson model for entity recognition which predicts the following entities: medicine name, form, strength, dosage, quantity, quantity on hand, refills, fill data, expiry date.

The processed information from the API is used to display data on the Wizeview portal to be accessed for later use. While experimenting with the medicines API [1] calls, I found that average time for the information to be processed was around 8.57s. This is a significant delay and problem with using microservices. If the information is required to be updated instantly on the device, it cannot work with a framework like this.

To reduce this significant time, it was decided to implement the image to text classification models on the device only. So, the task was split into development of two models. The first one being OCR for image to text conversion and the second one being classification of this text into various entities.

Chapter 2 - Named Entity Recognition

2.1 Dataset Generation

Named Entity Recognition is a part of natural language processing and is used for retrieving information. The purpose of NER is to find the entity or the class of the words. These entities or classes for example can be, name of a person, location.

The first task for NER was to create a dataset. There are various formats available but the reference from [3] was taken and the dataset was created using the Inside-outside-beginning tagging format (B-I-O) which is as follows:

John	lives	in	New	York
B-PER	O	O	B-LOC	I-LOC

Where B represents the beginning, I represent Inside, O represents outside. The PER is used for a person, LOC for location. For our purposes, the label B-name and I-name was used for labelling medicine names. Using this labelling process, the following Data Creation process was followed:

1. Wizeview had a data set of 17 images for front side medicine labels which were taken as the reference dataset.
2. From this data, 17 sentences using Amazon's Rekognition OCR output were extracted and labelled in B-I-O tagging format.
3. Datasets of medicine names were found [2] and combined with the current dataset of Wizeview of medicine names to generate a list of medicine names.

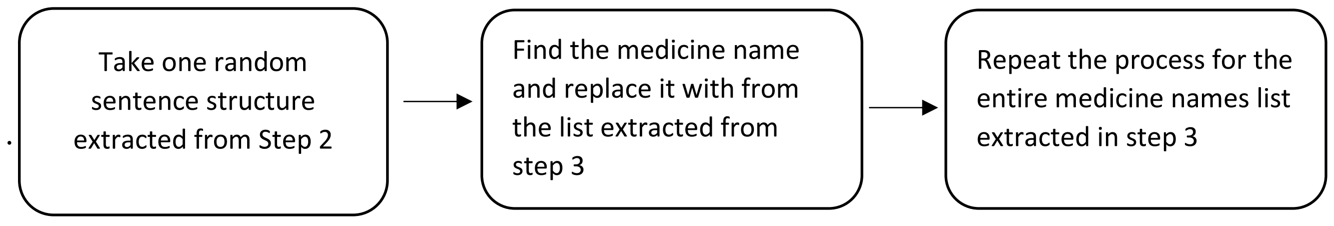

Take one random sentence structure extracted from Step 2
Find the medicine name and replace it with from the list extracted from step 3
Repeat the process for the entire medicine names list extracted in step 3

Fig 1: NER Dataset Creation

Using the above approach, three kinds of dataset set were generated:

1) Unique: The medicine names list was not repeated with the sentence structures selected randomly. This was necessary as to compare how the model performs with unrepeated medicine name values that the model may not have seen before.
2) Repetitive: The medicine names list was repeated 10 times randomly with the sentence structures selected randomly as well. This was to check how the model performs with repeated data and to check whether creating additional data helps improve the model performance or not.
3) AWS Bucket: 50 images were downloaded from the WizeView AWS bucket database. This was necessary to check how the model performs on the data it has not seen before. The medicine name might still be repeated, but the sentence structure was different for all these images. Also, the AWS bucket had only single medicine name present so the I-name class performance analysis is not done in AWS Bucket.

Table 1*:* NER Dataset Distribution

Dataset	**Total Sentences**	**Training Data**	**Test Data**
Repetitive	37410	30056	7354
Unique	3741	3006	735
AWS Bucket	50	0	50

Note that, in the report, the model trained on repetitive data is termed as Model_R whereas the model trained on unique data is termed as Model_U.

2.2 Performance analysis

For performance analysis, calculating accuracy was not the correct criteria as we have a significant class imbalance. For example, if the sentence contains 50 words with 1 medicine name. Even if I predict ‘O’ for every word, the accuracy would be 98%. Hence class wise results are discussed with focus on Precision, Recall, F1-score. The definitions of these are taken from [4] and are as follows:

- Precision: It is defined as the ratio of True Positives divided by the total number of true positives and false positive. It is used to determine how accurate the model is out of those predicted positive, how many of them are actual positive. For example, in medicine name entity detection, a false positive would mean that a word that is not a medicine name has been identified as one and might give erroneous data.

- Recall: It is defined as the ratio of True Positives divided by total actual positives (True positive plus False Negative). Applying the same understanding as before, this would be able to measure if a medicine name is predicted as ‘O’.
- F1 score: It is a function of Precision and Recall and is used when a good balance between precision and recall is required. As discussed earlier, calculating accuracy does not help so F1 score is used when there is an uneven class distribution.

2.3 Model training

There were a variety of models trained and check for this NER task.

2.3.1 Memory tagger

The first one was a memory tagger [14] which is just used to remember the most common named entity for every word and predict that. In case if the word is unknown, an ‘O’ is predicted. The first model discussed is trained on Repetitive data.

Table 2: Memory Tagger Model_R Results

Test Set				
	Precision	**Recall**	**F1-score**	**support**
B-name	0.96	0.96	0.96	7354
I-name	0.94	0.87	0.90	4389
O	1.00	1.00	1.00	289097
AWS Bucket				
B-name	0.68	0.79	0.73	66
O	1.00	0.95	0.97	2030

The followings are the results for the model trained on the Unique Dataset.

Table 3: Memory tagger Model_U Results

Test Set				
	Precision	**Recall**	**F1-score**	**support**
B-name	0.84	0.25	0.38	735
I-name	0.91	0.66	0.76	452
O	0.98	1	0.99	28507
AWS Bucket				
B-name	0.55	0.45	0.50	66
O	0.99	0.95	0.97	2030

As can be clearly seen, having repetitive data helps the memory tagger increase the efficiency as the F1-score on the AWS Bucket was 0.73 for Model_R whereas for Model_U, the F1 score decreased to 0.50.

2.3.2 Random Forest Classifier

A simple Random Forest Classifier [14] was used. First, the data was converted to a simple feature vector for every word. The features are defined as below:

1) Is Title
2) Is Lower
3) Is Upper
4) Length of word
5) Has digit
6) Has alphanumeric characters

For every word, a feature vector was generated which was used further for classification using a random forest. The number of trees used in the random forest is 20. No significant difference in the performance was observed by tuning the number of trees parameter. The results for Model_R are shown in the table below.

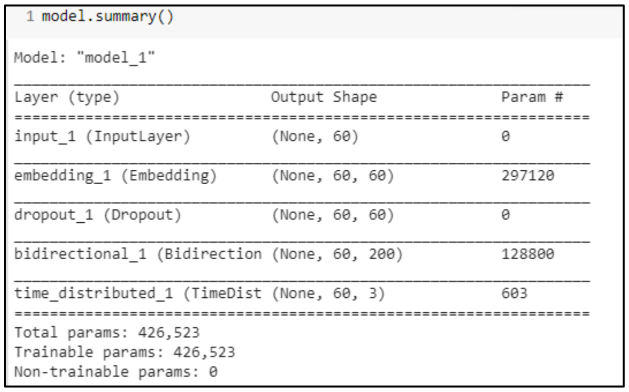
Table 4: Random Forest Model_R Results

Test Set				
	Precision	**Recall**	**F1-score**	**support**
B-name	0.79	0.40	0.53	7354
I-name	1.00	0.18	0.31	4389
O	0.97	1.00	0.99	289097
AWS Bucket				
B-name	0.16	0.47	0.24	66
O	0.98	0.92	0.95	2030

It was found that the performance for the Model_R and Model_U were quite similar so the results for Model_U have not been shown. As compared to the memory tagger, the random forest model performs worse as the F1-score decreases significantly especially for the AWS Bucket test set. This can of course be attributed to the feature selection process.

2.3.3 Sequence tagging using LSTM

LSTM model for sequence tagging was used in this. The summary of the model is as shown below:

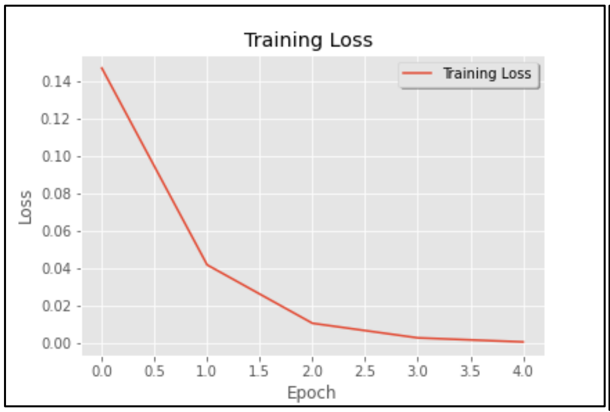


```
model.summary()

Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 60)                0
_________________________________________________________________
embedding_1 (Embedding)      (None, 60, 60)            297120
_________________________________________________________________
dropout_1 (Dropout)          (None, 60, 60)            0
_________________________________________________________________
bidirectional_1 (Bidirection (None, 60, 200)           128800
_________________________________________________________________
time_distributed_1 (TimeDist (None, 60, 3)             603
=================================================================
Total params: 426,523
Trainable params: 426,523
Non-trainable params: 0
```

Fig 2: LSTM Model Summary

The validation loss and training loss plots are also shown:



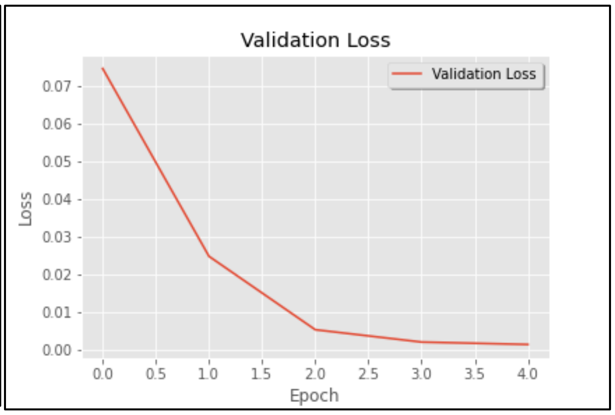


Fig 3: Training and Validation loss

The followings are the results for the model trained on the Unique Dataset.

Table 5: LSTM Model_U Results

Test Set				
	Precision	**Recall**	**F1-score**	**support**
B-name	1	1	1	7354
I-name	1	0.99	1	4389
O	1	1	1	289097
AWS Bucket				
B-name	0.57	0.12	0.20	66
O	0.97	0.97	0.97	2030

As can be seen, the model overfits a lot on the test data which has the same sentence distribution as the training data and the F1-score is very high but for sentence structure the model hasn't seen before, as in the AWS Bucket, the model performs poorly. Model_R performs even worse and results are not shown.

Causes:

1) The LSTM model overfits on the sentence structure very quickly. Even after adding and increasing regularization (Dropout, Weight decay) [12], the model overfits on the dataset quickly. Even with randomized sentences data, this model performs badly on the AWS bucket dataset.

2) This can be attributed to the fact that these models are used to learn the sequence structure of the words, but the OCR output is not necessarily a sentence, but just a collection of words.
3) There can also be another reason which is class imbalance [13]. As seen above, in the testing data, the tag ‘O’ appears 289097 times whereas the tag ‘B-name’ appears 7354 only so the models just learns to predict ‘O’ for every word.
4) Another factor observed was, during training the words and tags are converted to ids. Each word is represented by a unique id. While testing, if the word does not occur in the dictionary of the model, then the id for it would be taken as 0. The predicted values for these words were found to be a major source of the problem. Thus, the network was not performing well on any new words which was the purpose of doing NER.

2.3.4 Model selection

There were other models trained as well including a combination of LSTM and CRFs, enhancing LSTMs with character embeddings, BERT but the results were found similarly unsatisfactory. The memory tagger model was selected for the NER task for this reason as it has the highest F1-score and is simpler to train and implement.

Chapter 3 – Optical Character Recognition

3.1 Introduction

OCR stands for Optical Character Recognition which simply defined is the conversion of scanned, printed, or handwritten text [1] into text understood by machines for further processing and analysis which in this case would be for entity recognition. There are many examples of OCR applications ranging from depositing checks through mobile apps, or for reading your card information while making payments or such as in the case of Wizeview, for extracting the text from an image of a medication label.

There is a wide variety of OCR software's available today with many free and open source options which can be used directly on the devices such as Google's Tesseract, OCRopus, Kraken to many cloud services being offered such as Amazon's Rekognition, Textract, Microsoft Azure computer vision, Google cloud vision etc. The cloud services provide excellent performance but using them for real time applications is a problem due to the time delay in receiving back the information from the service. The best

3.2 Tesseract-OCR

Tesseract is an **open source** [11] Optical character recognition engine under Apache License 2.0. It helps to convert text from documents or images to machine legible text.

3.2.1 Setup and Installation

There are numerous ways to install tesseract as described in the documentation but for our purposes, training tools were required to be installed as well. So, the following instructions from [17] were followed. The installation was done on Ubuntu 18.04.

```
//Install the required packages
sudo apt-get install automake ca-certificates g++ git libtool libleptonica-dev
make pkg-config
sudo apt-get install --no-install-recommends asciidoc docbook-xsl xsltproc
sudo apt-get install libpango1.0-dev
sudo apt-get install libicu-dev
sudo apt-get install libcairo2-de


//Clone the master branch:
git clone https://github.com/tesseract-ocr/tesseract.git

//Then run the following commands
cd tesseract
    ./autogen.sh
    ./configure
    make
    sudo make install
    sudo ldconfig
    make training
    sudo make training-install
```

After the installation is finished, a traineddata file for a language is needed to recognize the text in images. Tesseract provides three kinds of files:

1) tessdata - `https://github.com/tesseract-ocr/tessdata`
2) tessdata_best - `https://github.com/tesseract-ocr/tessdata_best`
3) tessdata_fast - `https://github.com/tesseract-ocr/tessdata_fast`

The files in tessdata supports the legacy engine as well along with the LSTM based model. The eng.traineddata file was downloaded from tessdata_best repository. This downloaded file has to be copied in the tessdata configs directory which for Linux environments is /usr/local/share/tessdata.

3.2.2 Synthetic data from fonts for Data generation and model training

The process for generating synthetic data for fonts that require to be trained on is as follows:

1. Required repository download

Create a new project folder and run the following commands

```
git clone https://github.com/tesseract-ocr/tesseract.git
git clone https://github.com/tesseract-ocr/langdata_lstm.git
```

The first and second command clones the main repository of the tesseract along with the language data required for generating training data.

```
cd tesseract/tessdata
wget https://github.com/tesseract-
ocr/tessdata_best/raw/master/eng.traineddata
```

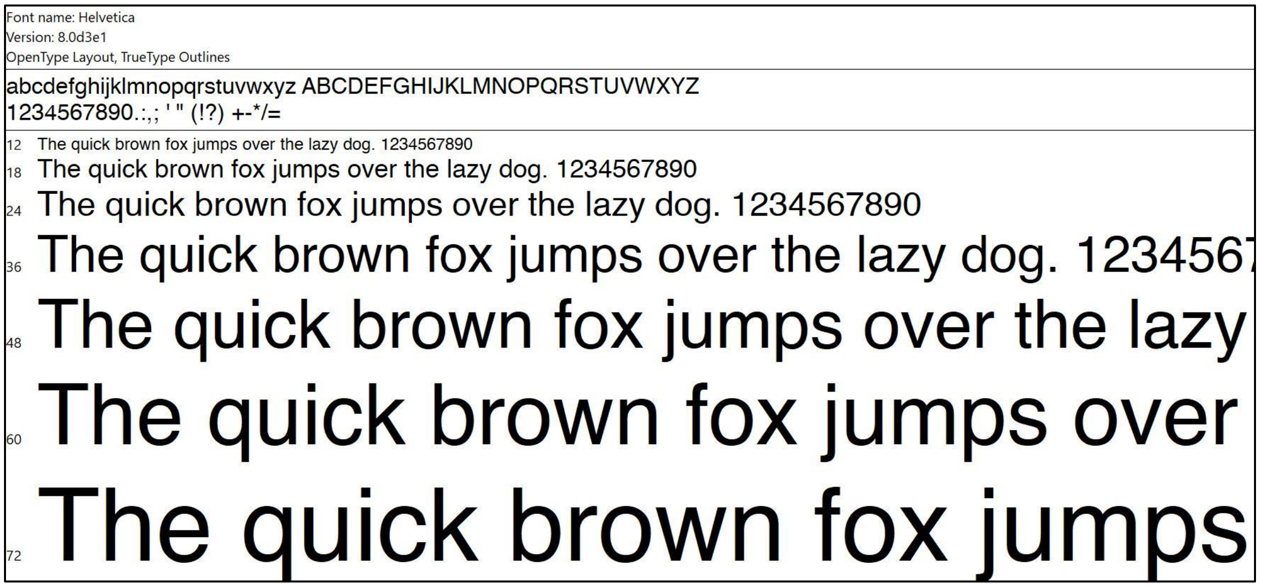
The third and fourth command are used to copy the eng.traineddata file from the tessdata_best repository as explained earlier.

```
mkdir fonts output train
```

After, that three new folders are created which are going to be used later.

2. Font identification and data generation

The second step is to identify the font that is required to be trained on. It is clearly identified in [6] that sans serif fonts are used for drug labels. Sans serif fonts are defined as the fonts that does not have extending features called ‘serifs’ thus the name, sans serif [7]. The most common sans serif fonts used according to [5] are Arial and Verdana are used for medication labels.
After identification, the tesseract requires ttf (TrueType font) files.
An example of ttf file for Helvetica font is like this:


Font name: Helvetica
Version: 8.0d3e1
OpenType Layout, TrueType Outlines

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890.:,; ' " (!?) +-*/=

12 The quick brown fox jumps over the lazy dog. 1234567890
18 The quick brown fox jumps over the lazy dog. 1234567890
24 The quick brown fox jumps over the lazy dog. 1234567890
36 The quick brown fox jumps over the lazy dog. 123456
48 The quick brown fox jumps over the lazy
60 The quick brown fox jumps over
72 The quick brown fox jumps

Fig 4: Font File

The files can be downloaded easily and required to be placed in the fonts folder created earlier. After placing the font files, a shell script *generate_training_data.sh* is written which contains:

```
rm -rf train/*
tesstrain.sh --fonts_dir fonts\
      --fontlist 'Arial' 'Verdana'\
      --lang eng \
      --linedata_only \
      --langdata_dir langdata_lstm \
      --tessdata_dir tesseract/tessdata \
      --save_box_tiff \
      --maxpages 400 \
      --output_dir train
```

The explanation for the script is as follows:

1) The shell script *tesstrain.sh* is provided by the tesseract training tools installed earlier. Most of the arguments are self-explanatory.
2) The fonts specified in the *fontlist* argument can be modified as to what is the requirement. For examples, for adding Helvetica font, the type font file for Helvetica should be placed in the fonts folder and the argument modified to:

```
--fontlist 'Arial' 'Verdana' 'Helvetica'
```

3) The *maxpages* argument by default is set to 4000 pages which can take a huge amount of time for data generation. During experimentation, it was found that the number 400 seemed to work pretty well without overfitting too much.

After running this shell script, file with extension *tif* will be generated in the train folder which contains randomly generated lines of text of the fonts specified. This file will have pages equal to the number which were specified during the data generation. The files with extensions *box* and *lstmf* are also created in the train folder which are used by the tesseract for training.

The current model downloaded from the tessdata_best repository can be evaluated on the generated training data. Run the following commands from the project directory:

```
combine_tessdata -e tesseract/tessdata/eng.traineddata eng.lstm
lstmeval --model eng.lstm \
      --traineddata tesseract/tessdata/eng.traineddata \
      --eval_listfile train/eng.training_files.txt
```

The *combine_tessdata* extracts the model from the traineddata file downloaded earlier. This model file (with lstm extension) is used to evaluate the model using *lstmeval*.
The character error rate: 0.876 and word error rate: 3.276

3. Fine tuning tesseract

For fine tuning tesseract, a scrip *finetune.sh* is created which contains the following commands:

```
OMP_THREAD_LIMIT=8 lstmtraining \
      --continue_from eng.lstm \
      --model_output output/engmed \
      --traineddata tesseract/tessdata/eng.traineddata \
      --train_listfile train/eng.training_files.txt \
      --max_iterations 2500
```

1) The *continue from* argument is the model which is used as a base for the fine tuning to start which in this case is the *eng.lstm* file generated from the *combine_tessdata* command earlier.
2) The *model_output* argument specifies the output directory and model name to be used which in our case would be *engmed.*
3) The *traineddata* and *train_listfile* argument are used to specify the traineddata file and the training files respectively.

4) The max iterations is the maximum number of iterations for which the model is to be trained. 2500 was found to be a good number after experimentation.

After the script is run, the output folder will generate a bunch of checkpoint files. Then we can use the following *combine.sh* shell script to generate the traineddata file from this checkpoint file.

```
lstmtraining --stop_training \
      --continue_from output/engmed_checkpoint \
      --traineddata tesseract/tessdata/eng.traineddata \
      --model_output output/engmed.traineddata
```

The explanation for the script is as follows:

1) The *stop_training* argument is used to specify to stop the training and generate the traineddata file.
2) The *continue_from* argument is used to specify the model that was just trained.
3) The *model_output* command specifies the output folder directory and model name.

This will generate the *engmed.traineddata* which can be used for evaluation and model deployment. To use this file for evaluation, run the following command:

```
lstmeval --model output/engmed_checkpoint \
      --traineddata tesseract/tessdata/eng.traineddata \
      --eval_listfile train/eng.training_files.txt
```

The final directory structure will be as follows:

```
>> combine.sh
>> eng.lstm
>> finetune.sh
>> fonts/
>> generate_training_data.sh
>> langdata_lstm/
>> output/
>> tesseract/
>> train/
```

3.2.3 Image data for Data generation and model training

The previous method of generating training data using synthetic fonts is effective for training on additional new fonts or improving the performance for fonts which are in the data. The sans serif fonts required for this task contain a huge list of fonts. Arial and Verdana were just taken as examples. But after viewing and analyzing the data, it was found that the fonts used have a lot of variance.

As mentioned earlier, Tesseract has two ways of training. The first one using synthetic font data discussed earlier and second one using image data.

The image data required by tesseract has to be in the form of line images and their transcriptions (ground-truth). Rules for them are as follows:

- Images must be TIFF and have the extension .tif or PNG and have the extension .png, .bin.png or .nrm.png.
- Transcriptions must be single-line plain text and have the same name as the line image but with the image extension replaced by .gt.txt.

Annotating images such as this can be a time-consuming task but two methods were used to automate the process to reduce the time for annotation.

1. Dataset generation using hocr-tools.

- ➢ The first step is to install the hocr-tools. Assuming pip is installed on the linux machine, use the following command. Else for reference, use [18]

```
sudo pip install hocr-tools
```

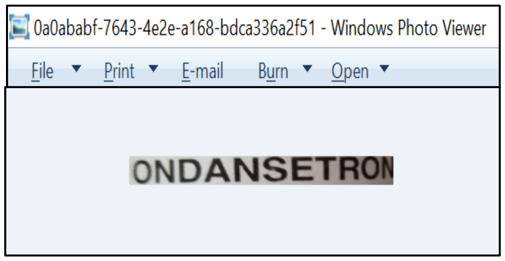
- ➢ Create a new project folder and clone the tesseract repository.

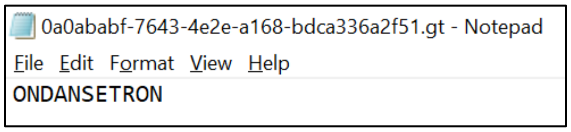
```
git clone https://github.com/tesseract-ocr/tesseract.git
```

- ➢ After cloning the directory create a new folder *myfiles* and place all the images in this folder that are required to be processed generating training data
- ➢ After that, create the following script *hocr_output.sh* which has been referred from [8].

```
#!/bin/bash
SOURCE="myfiles/"
lang=eng
set -- "$SOURCE"*.jpg
for img_file; do
    echo -e  "\r\n File: $img_file"
    OMP_THREAD_LIMIT=1 tesseract --tessdata-dir tesseract/tessdata
"${img_file}" "${img_file%.*}"  --psm 6  --oem 1  -l $lang -c
page_separator='' hocr
    PYTHONIOENCODING=UTF-8 ./hocr-extract-images -b myfiles/ -p
"${img_file%.*}"-%03d.exp0.tif  "${img_file%.*}".hocr
done
rename s/exp0.txt/exp0-gt.txt/ ./myfiles/*exp0.txt
```

- The summary for the script is given as follows:
    - SOURCE is used as myfiles folder where the images for generating training data are saved. Please note that the image files used for our purpose was jpg hence it has been defined has such
    - Tesseract is called which performs OCR on the image, divides it into line texts and generates hocr output.
    - The hocr output is used to extract the images from the original image that was identified by the tesseract.
    - Both image file and transcription file with the naming rules mentioned above are saved using this format.
- An example of this process is shown in the figure below. The image generated from the hocr can be seen below along with .gt.txt file containing the transcription.




0a0ababf-7643-4e2e-a168-bdca336a2f51.gt - Notepad
File Edit Format View Help

ONDANSETRON

Fig 5: Image data generation using hocr

- For some of these outputs, the ground truth generated from the script might be different which requires the data to be manually checked and corrected.

2. Dataset generation using bounding box API

   A bounding box API was created which is used to return the bounding box coordinates of the medicine name found. The bounding box API has four functions defined:

   1) *generateurl()*

   Input: Its argument is a picture

   Output: generates a url for the photo by using a base 64 encoded version of the photo

   2) *medicinename():*

   Input: arguments include: pictureid, patient id, visit id, picture url, picture number, tag

   Output: Generates a dictionary with the following keys: name, form, strength, dosage, quantity, quantity_on_hand, refulls, fillDate, expireDate. The name is returned as an output of the function currently

   3) *Detect_labels():*

   Input: arguments include a picture

   Output: uses aws 'Rekognition' to get OCR output with the following keys: DetectedText, Type, Confidence, Geometry. The entire dictionary is returned as the output of the function

   4) *Boundingbox():*

   Input: arguments include a picture

   Process:

   1) Calls *medicinename()* function to get the medicine name

   2) Searches for the medicine name returned in the output of *Detect_labels()* function. This string comparison is made case insensitive as the OCR output is in upper case characters.

   3) After finding the medicine name in *Detect_labels(),* the coordinates are

returned as an output of the function along with the medicine name.

Output: medicine name, height, width, top, left

This API was used for dataset generation. The idea was inspired from the above *hocr_output.sh.* The Bounding Box API can be used in the following way:

- Wizeview has an abundant dataset of images from their visits recorded till now.
- Those images passed through the Bounding Box API would return the medicine name along with its coordinates.
- Using those coordinates, the medicine name was cropped out of the image to create the image file and the medicine name was written in a text file to create the transcription. The naming conventions discussed earlier were followed for the above task.
- Python script for this task is shown below

```
entries = os.listdir(path)
for i,e in enumerate(entries):
  img_name = e
  m,t,c =boundingbox(path+img_name,20000+i)
  if c:
    img = Image.open(path+'froms3try2/'+img_name)
    imgWidth, imgHeight = img.size
    left = imgWidth*c['Left']
    top = imgHeight*c['Top']
    width = imgWidth*c['Width']
    height = imgHeight*c['Height']

    img.crop((left,top,left+width,top+height)).save(path+'medOCRs3/'
+img_name.split('.')[0]+'.tif')
    f=open(path+'medOCRs3/'+img_name.split('.')[0]+'.gt.txt','w+')
    f.write(str(m))
    f.close()
    print(i,m)
```

Using the script, 269 image data was generated and combined with the earlier generated data to be used in training.

3. Training using image data

The reference for this has been taken from [9]

Create a new project folder and run the following command.

```
git clone https://github.com/tesseract-ocr/tesstrain.git
git clone https://github.com/tesseract-ocr/tesseract.git
```

Place ground truth consisting of line images and transcriptions in the folder *data/MODEL_NAME-ground-truth* and further run the following command to download the eng.traineddata file from the tessdata_best repository

```
wget https://github.com/tesseract-
ocr/tessdata_best/raw/master/eng.traineddata
```

After this, to start training run the following command

```
make training MODEL_NAME=foo START_MODEL=eng MAX_ITERATIONS 1000
PSM=11 TESSDATA='data'
```

The arguments are clearly explained in [9] and can be referred from there for further changes.

After the training is finished a traineddata file is generated which can be used for model deployment and evaluation. The repository also provides a way to create additional traineddata files from intermittent checkpoints using the command below

```
make traineddata
```

Using this any of the checkpoints can be used for model deployment and evaluation.

3.2.4 Factors for Performance analysis

For performance analysis, three models were used:

1) tessdata_best
2) tessdata_fast
3) trained model (using image data)

Performance was measured on a test data set of 50 images. These 50 images were passed through the OCR model and used the OCR model to predict the medicine name. Both time (for the complete OCR and NER process) and accuracy data was recorded for the performance analysis. The baseline tesseract gives an accuracy of 50% with 112.3s

The following factors were studied for performance:

3.2.4.1 Configuration parameters

Tesseract has a variety of configuration parameters which can help improve the performance.

1. Thread Limit: Thread limit is a factor which depends on the machine where the tesseract is running. The configuration variable OMP_THREAD_LIMIT can be set to manually specify the number of cores. The following are the results for tessdata_fast.

   Table 6: Thread Limit analysis

Thread Limit	Time (s)	Accuracy (%)
1	61	50
2	54	50
4	47.2	50

   As can be clearly seen, the Thread Limit parameter decreased the time significantly without causing a change in the accuracy. This was only tested on the tessdata_fast file as the results for other models were same.
   An additional parameter recommened in the Tesseract FAQ is tessedit_do_invert. Putting this parameter to 0 decreased the time from 47.2 seconds to 40 seconds.

2. Page segmentation mode (PSM): It is used to define how tesseract should treat the text in the image being processed. For example, if the image contains a single character or a block of text, you want to specify the corresponding PSM so that you can improve

accuracy. There are 13 PSM modes defined in the tesseract documentation. After checking manually, three modes 6,11 and 12 were found to have the best performance. Their description as of in the documentation is as below:

PSM 6: Assume a single uniform block of text

PSM 12: Sparse Text. Find as much text as possible in no particular order

PSM 13: Sparse text with OSD

The results of this parameter are discussed further in the report.

3.2.4.2 Preprocessing of Images

Tesseract does various image processing operations internally before doing the actual OCR. It was necessary to analyze these images first before going for a preprocessing method. The processed image can be observed by using the configuration variable *tessedit_write_images* to True.

For the right image, it can be seen that the preprocessing method of Tesseract does pretty well as the text is highlighted very cleraly but for the left image, the preprocessing method doesn’t do a good job. It removes the out of foucs part which might have critical text and also the text is not clearly highlighted which causes problems in the OCR.

Based on this and observing other preprocessed image outputs, it was decided to check the performance of the OCR task based on applying various preprocessing methods.

The preprocessing techniques checked were:

1) Rescaling: As mentioned in the official documentation, tesseract works best on images which have atleast 300 DPI. As the images being processed through the OCR may be of different sizes, it was checked whether the resizing process helped the OCR. The image can be rescaled to either a fixed size, a percentage scaling can be done where if the image is of size (m,n), it can be rescaled to (2m,2n). There are various interpolation methods for rescaling as well such as linear, cubic, area which were analyzed as well.

2) Binarization: It is the process of converting an image to black and white [19]. Tesseract does this internally using the Otsu algorithm but the results may not be optimal for images with varying brightness. There are a variety of Binarization techniques:
   Simple Thresholding: In Simple thresholding, a thresholding function is used. If a pixel is

smaller than the threshold, its value is set to 0 otherwise it is increased to the maximum value.

Adaptive Thresholding: This is useful for images with different lighting conditions in different areas. The algorithm determines the threshold for a pixel based on a small region around it. So different thresholds are obtained for different regions of the same image.

3.2.5 Results and Discussion

Thread limit is kep as 4 and tessedit_do_invert configuration variable is kept as 0. For analyzing the performance, the following parameters were considered:

Table 7: Parameters Checked

Parameter	**Values**
Model type	Tessdata_best, tessdata_fast, trained model
PSM	Mode 6, 11, 12
Resize	Scaled, No, Fixed (1191x2000)
Resize Scale	1.5, 2, 2.5
Resize method	Linear, Cubic, Area

All these parameters were run in a loop, giving a total possible 118 configurations. The results for each parameter are summarized below:

1) Model type: For model, the average accuracy for the three different types of models are plotted below. The time taken for the trained model is similar to the best but the accuracy increases signficantly (around 6% increase). It should be also noted that the highest accuracy reported was for the trained model (64%) while the tessdata_fast as the name suggests takes the least time on average.

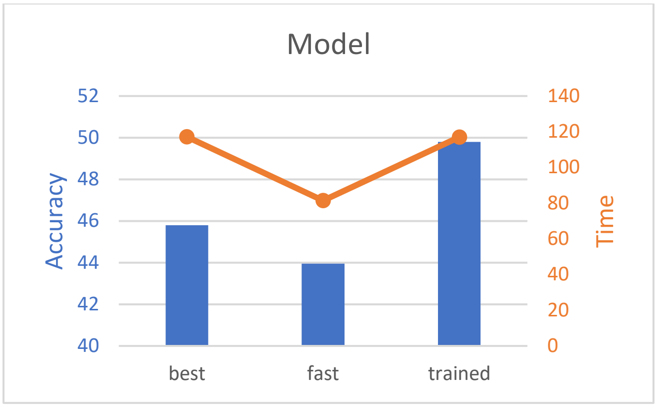


Fig 7: Accuracy and time vs Model type

2) PSM Mode: As is clearly visible, PSM mode 6 provides the best combination of accuracy and time.

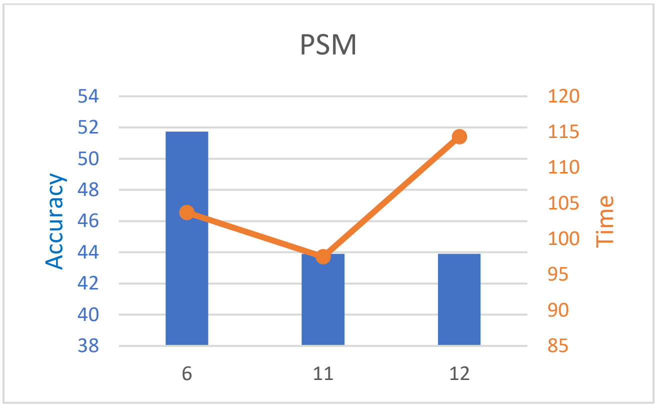


Fig 8: Accuracy and time vs PSM Mode

3) Resize: The Fixed resize option didn't provide good accuracy. Although in this result it seems resizing is not a good option, it is further analyzed in the next parameter.

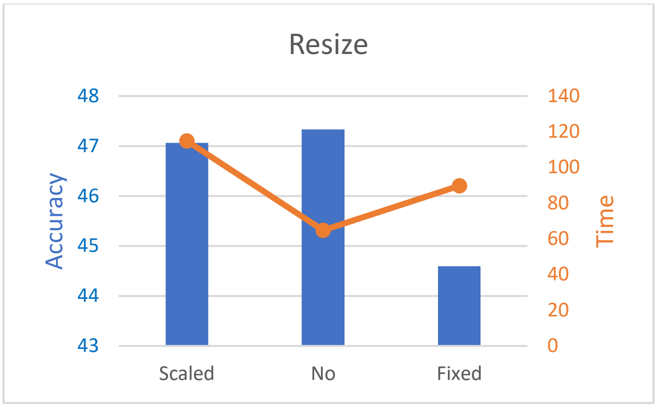


Fig 9: Accuracy and time vs Resize option

4) Resize Factor: It can be clearly seen from the below data, that resizing to 1.5 times increases the accuracy significantly without a lot of compromise in the time.

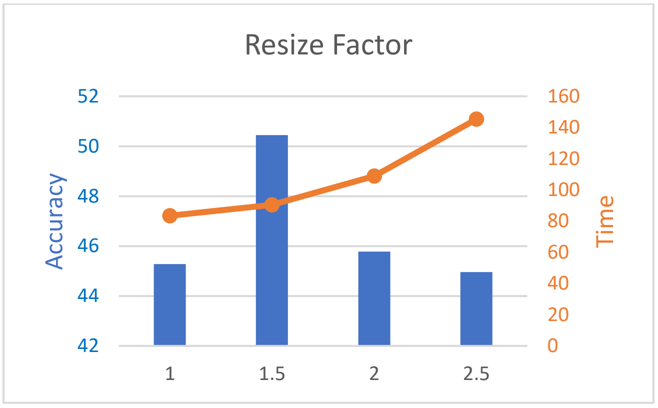


Fig 10: Accuracy and time vs Resize Factor

5) Resize Method: The linear method provides the best combination of time and accuracy as can be seen from the chart below.

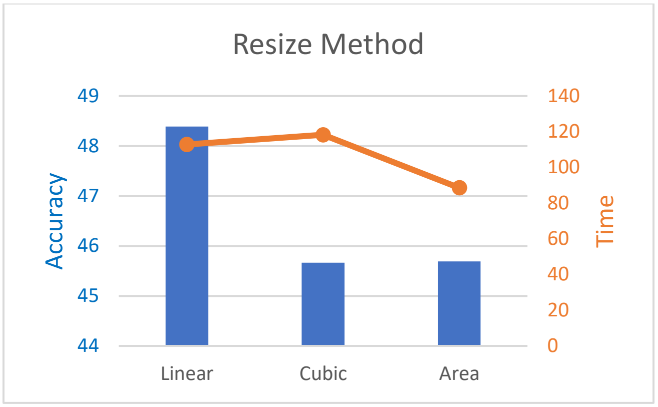


Fig 11: Accuracy and time vs Resize Method

Based on the above analysis, two best model parameters were found:

Table 8: Best Models

Parameter	**Model 1**	**Model 2**
Model type	tessdata_fast	trained model
PSM	Mode 6	Mode 6
Resize	Scaled	Scaled
Resize Scale	1.5	1.5
Resize method	Linear	Linear
Accuracy	58	62
Time	62.5	100.97

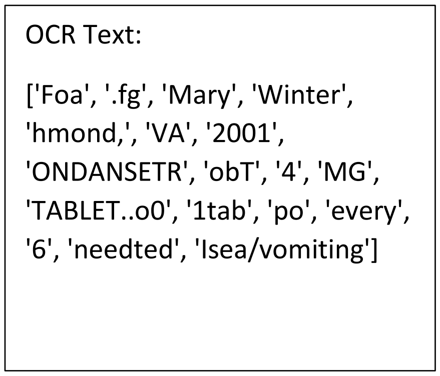
For Binarization, the following results were obtained:

Table 9: Binarization Analysis

Thresholding	**Model 1 accuracy**	**Model 1 time**	**Model 2 accuracy**	**Model 2 time**
THRESH_BINARY	42	34.4	42	71
THRESH_BINARY_INV	12	34.06	2	72.2
THRESH_BIN_TRUNC	46	68.46	56	197.62
ADAPTIVE_THRESH_MEAN_C	46	25.81	46	60.27
ADAPTIVE_THRESH_GAUSSIAN	28	23.50	32	51.11

As can be clearly observed from the above data, none of the binarization techniques helped in improving the performance so they were not used but it gives an interesting insight that with binarization, the content for tesseract to recognize especially with THRESH_BINARY decreases thereby decreasing the time for prediction.

For further analysis, the incorrectly classified medicine name images were extracted and analyzed. One of the images is shown below:



```
OCR Text:

['Foa', '.fg', 'Mary', 'Winter',
'hmond,', 'VA', '2001',
'ONDANSETR', 'obT', '4', 'MG',
'TABLET..o0', '1tab', 'po', 'every',
'6', 'needted', 'Isea/vomiting']
```

Fig 12: OCR Output

As can be observed, the OCR does well on the text in the middle but has problems in the text in the curvature. Also, the OCR clearly identifies ‘ONDANSETR’ missing the last two characters. The NER memory tagger was manually modified to handle two such following situations:

1) Extra word added to the medicine name at the start or the end such as if the name is advil, [advil is predicted by the OCR model.
2) The predicted OCR word is not complete for example instead of advil, only dvil is predicted by the OCR output.

These two situations were manually handled in the program

3.2.6 Model selection

The final model results are as shown:

Table 10: Final Model Results

Parameter	Accuracy	Time
Baseline	50	112.3
Model 1	76	49.9
Model 2	80	74.4

The accuracy increase in the Model 1 with just preprocessing is very significant. It shows that preprocessing is more important than model training for tesseract. Although, model training still had higher accuracy and should be used if higher accuracy is highly desirable.

The Model 1 also performed significantly faster than Model 2 and the Baseline. The better the image is processed, the lesser time it takes for tesseract to process the image as it helps to remove erroneous data.

Chapter 4 - Conclusion and Recommendations

- OCR accuracy improvement

Preprocessing was found to be a greater factor for improving OCR accuracy rather than model training. The current issues with OCR preprocessing now are related to the curvature part of the label bottle. Further analysis or image processing techniques could be checked so that this area can be analyzed, and its output improved further.

Tesseract provides option to add a dictionary to its model so that the words can be recognized better. It is also recommended to use this option to include the medicine name list as a dictionary to the tesseract model.

- OCR training

Using image data for model training is a better option than using the fonts option due to a wide variety of fonts on medication labels. Amazon's Rekognition API can be used on the AWS bucket to get the bounding boxes and extract the line images, these images can be further labelled and used for training OCR.

The Model 2 was trained using tessdata_best model but the same process can be repeated using the tessdata_fast model which should give the increased accuracy as we are getting now with decreased time.

- Additional NER entities

The NER task involves not just predicting the medicine name but other parameters such as expiry date, form, strength etc. as well. While including these parameters can solve the class imbalance problem for NER for training neural network models, it is still recommended to use rule-based engines such as regular expressions for these tasks as they are highly accurate and will be quicker as well in their prediction. The NER accuracy is highly dependent on the OCR accuracy, so it is recommended to focus on improving OCR accuracy first.

- iOS Development

For iOS development, there are plenty of tutorials such as [16] on deploying tesseract on iOS while using the trained data file for deployment. The process for deployment would be as follows:

1) Deploy the trained data file along with tesseract OCR as mentioned in the instructions in the above tutorial
2) Perform the same image processing and the configuration parameters as for the final model obtained
3) Deploy memory tagger using a simple text file search

References

[1] Website https://api-docs.wizeview.com/

[2] ARCHANA A. SHINDE, D**.** 2012.Text Pre-processing and Text Segmentation for OCR. International Journal of Computer Science Engineering and Technology, pp. 810-812.

[3] Che, W., Wang, M., Manning, C. D., & Liu, T. (2013, June). Named entity recognition with bilingual constraints. In Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 52-62).

[4] Goutte, C., & Gaussier, E. (2005, March). A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In European Conference on Information Retrieval (pp. 345-359). Springer, Berlin, Heidelberg.

[5] Website https://www.afb.org/blindness-and-low-vision/your-rights/rx-label-enable-campaign/summary-recommendations-pharmacists

[6] Leat SJ, Ahrens K, Krishnamoorthy A, Gold D, Rojas-Fernandez CH. The legibility of prescription medication labelling in Canada: Moving from pharmacy-centred to patient-centred labels. Can Pharm J (Ott). 2014;147(3):179-187. doi:10.1177/1715163514530094

[7] Website https://en.wikipedia.org/

[8] Website https://github.com/tesseract-ocr/tesstrain/issues/7#issuecomment-419714852

[9] Website https://github.com/tesseract-ocr/tesstrain

[10] Website https://tesseract-ocr.github.io/tessdoc/ImproveQuality

[11] Smith, R. (2007, September). An overview of the Tesseract OCR engine. In Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) (Vol. 2, pp. 629-633). IEEE.

[12] Merity, S., Keskar, N. S., & Socher, R. (2017). Regularizing and optimizing LSTM language models. arXiv preprint arXiv:1708.02182.

[13] Bingel, J., & Haider, T. (2014, May). Named entity tagging a very large unbalanced corpus: training and evaluating NE classifiers. In *LREC* (pp. 2578-2583).

[14] https://www.depends-on-the-definition.com/

[15] Website https://tesseract-ocr.github.io/

[16] Website https://www.raywenderlich.com/2010498-tesseract-ocr-tutorial-for-ios

[17] Website https://tesseract-ocr.github.io/tessdoc/Compiling

[18] Website https://github.com/tmbdev/hocr-tools

[19] https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html