



Inspiring Excellence

School of Engineering and Computer Science

BRAC University

***Bangla Character Recognition for Android
Devices***

Thesis Supervisor:

Abu Mohammad Hammad Ali

Department of CSE,

BRAC University

Conducted by:

Shahrin Manzur (12101113)

Shafiqul Islam (12101128)

Abu Foysal (12101131)

Aparajita Chowdhury (12301056)

Declaration

This is to certify that this thesis report is submitted by the authors listed for the degree of Bachelor of Science in Computer Science and Engineering to the Department of Computer Science and Engineering under the School of Engineering and Computer Science, BRAC University. We hereby declare that this thesis is based on the results found by us and no other. Materials of work found by other researchers are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted elsewhere for the award of any degree or diploma.

Signature of Supervisor

Abu Mohammad Hammad Ali

Signatures of Authors

Abu Foysal

Shafiqul Islam

Aparajita Chowdhury

Shahrin Manzur

Acknowledgement

We are grateful to our supervisor Abu Mohammad Hammad Ali, Lecturer, Department of Computer Science and Engineering, BRAC University, for guiding us. Because of him we were capable of developing our understanding regarding the thesis topic.

We would like to thank Muhammed Tawfiq Chowdhury, alumni of BRAC University, who managed his valuable time to help us in our thesis project.

We are grateful to Moin Mostakim and Rubayat Ahmed Khan for their insightful comments and encouragements, but also for the hard questions which motivated us to widen our research from various perspectives.

Furthermore, we would like to show our gratitude to our parents, for their continuous support and faith in us for achieving our goals.

Finally, we are extremely grateful to the Almighty, who has always helped us in every step of our lives to reach this point.

Abstract

In this paper, we illustrate our attempt to create editable documents from images by retrieving the text. The process is widely known as Optical Character Recognition (OCR). We have tried to build an Android application for detecting Bengali characters. Previously, several attempts have been made in developing a Bengali OCR. However, there were a few limitations which drove us to work on this project. In order to recognize more characters and joint letters, we decided to work on reducing the error rate to preserve more texts. To serve our purpose, we found the Tesseract OCR engine and Leptonica Image Processing Library to be the best option. Tesseract is used in order to recognize the characters and Leptonica is used to build an Android application by extracting data from the text. We are using the Tesseract 3.03 version currently available to work on this project. Moreover, we demonstrate how we obtained better results by manipulating Tesseract along with Serak to create box files and trained data. In addition to that, we discuss how we dealt with joint letters, dangerous ambiguity and contrast issues in order to increase efficiency. Furthermore, we explain our analyzed data, our progress and the future scopes of improvement.

Keywords

Optical Character Recognition (OCR), Bangla language, Android, Tesseract, Leptonica.

Table of Contents

1. Introduction	9
1.1 Background	9
1.2 Motivation	10
1.3 Thesis Outline	10
2. Literature Review	12
3. System Architecture	15
3.1 Tesseract OCR	15
3.2 Development environment	16
3.3 Building Tesseract Library	17
3.4 The New Tesseract 3.03	21
3.4.1 Building the training tools	21
3.5 Software used for training	22
3.5.1 jTessBoxEditor	22
3.5.2 QT Box Editor	24
3.5.3 Serak Tesseract Trainer v0.4	25
4. System Approach	27
4.1 Work flow:	27
4.1.1 Training Dristee OCR	27
4.1.2 Executing Dristee OCR	28
5. System Implementation	30
5.1 Generating image files from text files	30
5.1.1 Types of characters used for training	31
5.2 Creating Box Files	33
5.3 Processing Box Files	34
5.4 Computing the Character Set	34
5.4.1 Setting the Unicharset Properties	35
5.5 Font Properties	35

5.6 Shape Clustering	36
5.7 Unicharambigs	37
5.8 The Final Crunch	38
6. System Programming	39
7. Experimental Result	43
7.1 Precision and Recall	43
7.1.2 Precision and Recall for AdorshoLipi	46
7.1.3. Precision and Recall of Nikosh	48
7.1.4. Precision and Recall of Kalpurush	49
7.1.5. Precision and Recall of SolaimanLipi	51
7.1.6 Skewness:	54
8. Conclusion	56
9. Future Aspects	57
Acronyms	59
References:	60
Appendix.....	62

List of Figures

Figure 1: Properties for tess-two	18
Figure 2: Configuration type selection.....	18
Figure 3: Configuration Window	19
Figure 4: NDK Builder Selection.....	20
Figure 5: Tess-two Library Selection.....	20
Figure 6: Outlook of jTessBoxEditor.....	23
Figure 7:Font selection in jTessBoxEditor.....	23
Figure 8: Generating Tiff and Box files using jTessBoxEditor.....	23
Figure 9: Generated Tiff and Box files for SolaimanLipi font.	24
Figure 10: Outlook of QT Box Editor.....	24
Figure 11: Setting Bengali language in QT Box Editor	25
Figure 12: Editing box files in QT Box Editor.....	25
Figure 13: Overview of Serak Tesseract Trainer	26
Figure 14: OCR Mode of Serak Tesseract Trainer.....	26
Figure 15: Training workflow of Dristee OCR	28
Figure 16: Workflow diagram.....	29
Figure 17: Text input in jTessBoxEditor.....	30
Figure 18:Text input using txt file.....	30
Figure 19: Detection using only font size 12	31
Figure 20: Detection after using multiple font sizes	31
Figure 21: Vowel diacritic separate	33
Figure 22: Vowel diacritic merged.....	33
Figure 23: Box file creation	33
Figure 24: Unicharset sample.....	35
Figure 25: DangAmbigs.....	37
Figure 26: Unicharambigs.....	37
Figure 27: Relationship between retrieved data and total set of data	43
Figure 28: Relationship between retrieved, not retrieved and irrelevant data retrieved.	43
Figure 29: Relationship between precision and recall.....	44
Figure 30: Harmonic average and Break-even points.	45
Figure 31: Precision for desktop version.....	52
Figure 32: Precision for Android version.....	53
Figure 33: Recall for desktop version	53
Figure 34: Recall for Android version	54
Figure 35: Skewness	54
Figure 36: Skewed Bangla Text	55

List of Tables

Table 1: Numerical Ambiguities	31
Table 2: Types of characters trained	32
Table 3: Various types of training images.....	32
Table 4: Diacritic overview.....	33
Table 5: Renaming with language prefix	38
Table 6: Comparison after exposure	40
Table 7: Exposure modes	40
Table 8: Sample detection	42
Table 9: Experimental result calculated for mobile application of AdorshoLipi font.....	46
Table 10: Experimental result calculated for desktop version of AdorshoLipi font.	47
Table 11: Experimental result calculated for mobile application of Nikosh font.....	48
Table 12: Experimental result calculated for desktop version of Nikosh font.	48
Table 13: Experimental result calculated for mobile application of Kalpurush font.....	49
Table 14: Experimental result calculated for desktop version of Kalpurush font.	50
Table 15: Experimental result calculated for mobile application of SolaimanLipi font.....	51
Table 16: Experimental result calculated for desktop version of SolaimanLipi font.	51
Table 17: Pearson's Formula	55

Bangla Character Recognition for Android Devices

1. Introduction

1.1 Background

Optical character recognition (OCR) converts typed, handwritten or printed images into editable texts digitally [1]. This eases editing and storage electronically, which would have been immensely time consuming if typed manually. However, there are several complications. OCR tends to confuse similar letters, for example “o” and “0” [2]. Moreover, it will be similarly perplexing for OCR to detect characters on dark backgrounds [2]. Bengali character recognition is much more complicated. There are several different fonts and joint letters in Bangla which makes the job more difficult. Despite not being something new, there has not been much improvement regarding Bangla OCR. For example, only two sophisticated Bangla OCRs have been published in 2006 [3].

There might be several reasons for this. Firstly, OCR is designed for English alphabets, so training it for Bengali is a challenging job. Secondly, there are many more Bengali characters compared to Roman characters and some are so similar that OCR often produces wrong output [4]. Previously, there has been research in this area, but few attempts to work on the space issue. It is important to note that OCR cannot detect any space between two words. Therefore, it remains a challenging topic.

Currently, books and documents are digitized by manually typing them. This is a very long process as well as costly. Therefore, if we could build an OCR to read the characters directly from the pages and digitize them, then the effort will be drastically lessened.

Mobile phones, especially Android smartphones, have become a very popular device. Android devices are commonly used and come with extensive features. These devices have cameras which can take good quality pictures. There are several OCR applications for different

languages for detecting texts directly from images. Therefore, we have conducted our research on Android devices and developed an application to detect text instantaneously, especially for Bengali characters.

1.2 Motivation

We have already gone through some research paper on related works done previously. We found out that the accuracy level in various categories is not satisfactory. Therefore, we were motivated to increase the accuracy level.

In our day-to-day life, we see that many old books, important papers or documents get worn out or damaged with age. Nor, do we have any digital forms of these writings. There was no scanner to scan and store these documents. Thus, there is no other way to get the documents again other than typing the entire thing manually. Moreover, if we wanted to edit any part of the writing then we had to rewrite everything and correct the errors. This can create overhead and unnecessary labor. Sometimes, even rewriting is not possible as the documents can get illegible. For this reason, we have thought to develop an OCR application to preserve such books and papers for our convenience. The application will be able to create editable text files so that we can edit or print the documents whenever we want. Furthermore, the Android application will work quicker than typing or writing the whole thing manually.

Consequently, our primary goal is to build a handy and usable application which can be used by people of all ages. It will be a portable application that can run on any Smartphone device. Moreover, the whole process will be shortened as it will be an Android based application.

Lastly, as the generated text file is going to be a searchable and editable document, people can search the desired word rather than reading the whole page. Subsequently, the converted file can be stored as a softcopy for future use.

1.3 Thesis Outline

In chapter 2, we have discussed about the previous works conducted on OCR for detecting Bengali characters. Chapter 3 is based on the System Architecture of our Android application. The full description about the environment that is required to develop our application is discussed here. Next, in chapter 4, we have briefly gone through the approach we are following

in developing our Android OCR application. Chapter 5 covers the technical parts of the Tesseract OCR engine and its training is discussed in detail in this chapter. The whole mechanics behind the training of Tesseract is described here. Chapter 6 is solely based on the performance of the application. The application is tested under various circumstances and environmental conditions. Several statistics and graphs are used to explain the outcome of our Android OCR application. In chapter 7, we have tried to summarize our thesis work and provide a brief overview of the whole process. Chapter 8 is a brief discussion on how the OCR application can be further improved in the future. Moreover, we have also included some problems that are still present in the application. However, this chapter also contains a few ideas about how the problems can be approached. In the Appendix, we included about parts of the source code that we have used during the development of our Android OCR application. The full source code is available on request.

2. Literature Review

While progressing and planning for our thesis, we consulted many research papers regarding OCR. Since very less has been done on Bangla OCR, we studied the papers for English and Hindi OCR and found fruitful information. Smith [5], discusses in his paper how OCR detects damaged characters easily. This was a big help for our research. He also worked on 20 samples of 94 characters of eight fonts in four attributes, bold, italic, normal, bold italic, which gave rise to a data set of 60,160. We decided to work with four fonts training 3,725 characters each. However, we managed to find some papers on Bangla OCR. Omeo, in their paper, pointed out some important facts about Bangla characters. Bangla characters are not case sensitive [1]. However, some have a special feature called “matra” which is a line over the character. Hasnat [3], worked on HMM (Hidden Markov Model), a recognizer used for handwritten text and speech recognition, as there were very few attempts reported for printed character recognition. However, we decided to use Tesseract as character recognition, as it is the best possible open source Optical Character Recognition engine.

Zaman [6], points out in their paper that the current languages trained by Tesseract are, Arabic, Bulgarian, Catalan, Chinese (Simplified), Chinese (Traditional), Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hindi, Hungarian, Indonesian, Italian, Japanese, Korean, Latvian, Lithuanian, Polish, Portuguese, Romanian, Russian, Serbian (Latin), Slovak, Slovenian, Spanish, Swedish, Tagalog, Thai, Turkish, Ukrainian, and Vietnamese.

However, there is no mention of Bengali language which rose our curiosity even more. Chowdhury [7] and his team worked under Khalilur Rhaman to build an improved desktop version of Tesseract by training 18110 characters and 2617 words according to their report. However, we figured out it would be more convenient if it was more portable, thus we decided to develop an Android version. Android devices come with a camera, which we can use to capture images directly. On the other hand, in most desktop versions, one has to scan a paper, then pass it through their OCR application. Therefore, our approach of using Android devices greatly speeds up the process and decreases the overall effort needed. Mahub [6] and his team, worked on an Android version as well. Their approach was to develop an Android application to translate images captured by a portable device’s camera which can later translate and display on screen along with the original text. While going through their paper we realized their training set 1 has an accuracy rate of only 68.62%. Further, their application failed to recognize spaces

and contained a few words in their test images. Thus, we decided to keep these issues in mind while working on our thesis. Furthermore, we have used the latest version of Tesseract, version 3.03, available now, which was not present at their time. Arif [8], worked on OCR using feature extraction. He talks about different feature extraction techniques and how zoning is used to build a better Bengali character recognizer.

Rakhsit [4], on the other hand worked on recognizing handwritten text. They have used Tesseract 2.01 instead of building a new recogniser. They have used the handwritten version containing few Bengali characters. They used pen based devices such as a stylus or tablet and gathered different handwritings from different people. Their first data set contained individual handwritten Bengali vowels, their second data set had Bangla consonant and third set contained digits. Their accuracy was around 90% for each set of data.

However, their data set was not large enough to rely upon. We had tried to train a few samples of handwritten text, but unfortunately, we were not able to train them on Tesseract. The training failed in shapeclustering processing, which means that Tesseract was unable to detect the shapes present in the image and store them in the database.

Therefore, we decided not to deal with handwritten characters, as well as Rakhsit [9], was not too clear about juktakkhors or joint letters as they only considered single letters. Although there has been much research on handwritten recognizers, very few attempts were taken for Bengali characters. One reason might be it is very different from Roman letters as Bengali characters have “matra” or headline and “aakar” and “folas”.

Pal [10], used the approach of recognizing the character shapes by a combination of template and feature matching approach. Images are digitized by flatbed scanner and subjected to skew correction, line, word and character segmentation, simple and compound character separation. They have used a feature based tree classifier for simple character recognition.

Chaudhuri, in their proposed model [11], used document digitization, skew detection, text line segmentation and zone separation, word and character segmentation, character grouping into basic, modifier and compound character category for both Bangla and Devnagari (Hindi). Their system showed a good performance for single font scripts, printed on clear documents.

Omeo [1] in their paper mentioned the development of OCR, their workflow and algorithms. According to their paper, there are 5 steps to basic OCR:

1. Scanning.
2. Preprocessing.
3. Feature extraction or pattern recognition.
4. Recognition using one or more classifier.
5. Contextual verification or post processing.

They also mention noise detection and reduction. They discussed two types of noise 1) background noise 2) salt and pepper noise. They also pointed out that inability of differentiating between two similar characters is also the result of noise. In our case also, we faced various such cases and decided to handle it using an algorithm which will be discussed later. Another vital reason for noise is brightness and contrast. Patel [2], in their paper discussed about many aspects of recognition background, such as gray scale, printed, colored, etc.

They compared two different recognizers, Tesseract and Transym and figured out that the latter is better in detecting characters in number plate. Tesseract faces issues while considering dark background. Thus, we have decided to improve the background by increasing contrast and brightness in the pre-processing section which we hope will lead to better output.

3. System Architecture

3.1 Tesseract OCR

The Tesseract OCR engine was originally developed as a proprietary software at Hewlett-Packard between 1985 and 1995 and has been sponsored by Google since 2006 [7]. Tesseract is the most accurate open-source OCR engine which uses Leptonica Image Processing Library for image processing purposes. Tesseract can read a wide variety of image formats and convert them to text in over 40 different languages. However, Tesseract was originally designed to recognize English text only. To deal with other languages and UTF-8 characters, such as Bengali, several efforts have been made to modify its engine and the training system [9]. The system structure itself needed to be changed to make Tesseract able to deal with languages other than English. Tesseract 3.0 can handle any Unicode character. However, there were limits as to the range of languages that Tesseract will be able to successfully detect. Therefore, we had to take adequate actions to make sure that Bangla language gets recognized by Tesseract.

Tesseract 3.01 added top-to-bottom languages, and Tesseract 3.02 added Hebrew (right-to-left). Tesseract can currently handle complex scripts like Arabic with an auxiliary engine called cube. However, cube, is not yet equipped to detect the Bangla language. Additionally, it includes "unicarset" to make multi-language handling easier. This function aids us as we are going to be using four different Bangla fonts to train and detect characters. Though Tesseract is slower with a large character set language (like Chinese), but it seems to work nonetheless. Tesseract also takes more time to detect Bangla character compared to detect English characters. However, it can still detect Bangla characters which is the main purpose of our research.

Tesseract 3.03 added new training tool text2image to generate box/tiff files directly from text. It also has support for PDF output with searchable text. However, the text is only searchable, as it will be in PDF format and cannot be edited. Tesseract v3.03 (rc) is currently the latest available Tesseract engine, therefore, we used this engine in our research.

Initially, there was a desktop version programmed in the C programming language. It was only capable of detecting text from images saved on the computer. However, using tess-two library we can use Tesseract and Leptonica Image Processing Library on the Android platform. Tess-two is a fork of the Tesseract Tools for Android that provides us the ability to utilize the OCR

engine on an Android device. The Tesseract Tools for Android is a set of of the following three features:

- Android API
- Tesseract OCR engine
- Leptonica Image Processing Library.

The tess-two comes up with the tools for compiling and running both Tesseract and Leptonica Image Processing Library on the Android OS.

3.2 Development environment

This project uses the tess-two library to incorporate Tesseract and Leptonica Image Processing Library for using in any Android Platform. To use tess-two, at first, we had to build it on the Linux based operating system. For this we used Ubuntu OS. Tess-two contains an Android library project that provides a Java API for accessing natively-compiled Tesseract and Leptonica APIs. We had to use Android SDK and Android NDK to build the project in Ubuntu. However, after completing building the application, we shifted to Windows Operating System.

The main advantage of this shifting was, we could use some Windows based softwares like Serak Tesseract Trainer and QT Box Editor for working with multiple files in batch. We can use QT to edit and create box files very easily. By using Serak, we can automatize the scripting of Tesseract on TIFF/box file pairs. In Ubuntu, we had to manually enter all the file names of the training images and box files in the terminal to run Tesseract on them. The number of files is quite large considering the fact that we are training a huge data set. By using Serak, the overall effort is minimized.

We have successfully installed and run the Tesseract engine with tess-two library using the following configurations:

1. Ubuntu 14
2. Windows7/8.1/10
3. Android 2.2 +
4. Tesseract v3.03
5. Leptonica Image Processing Library v1.72
6. Four trained data file for four different types of Bangla language.

3.3 Building Tesseract Library

For using tess-two in our project as a library, we had to first build and then import it as an Android Library. We have used Eclipse Juno Android ADT for developing our project. So, after building tess-two, we had to import it into Eclipse as Eclipse Android Library Project.

To build the latest tess-two code, we had to run the following commands in the terminal:

```
git clone git://github.com/rmtheis/tess-two tess
cd tess
cd tess-two
ndk-build
android update project --path .
ant release
```

After building, we imported the tess-two library package into Eclipse using File->Import->Existing Projects into Workspace.

Next, we had to build Tesseract 3.03 and Leptonica Image Processing Library for our project. For this we had to use the Android NDK to compile the native language in Tesseract and Leptonica libraries. The Native Development Kit (NDK) is a set of tools that allows us to leverage C and C++ code in our Android apps.

These are the steps we followed to setup the Android NDK:

1. Downloaded the Android Native development Kit (NDK) from the Android Developers website.
2. Extracted the NDK in a folder in the computer. We extracted it in “D:\Android\android-ndk”.
 - a. Configuring Eclipse: Project->Properties->Builders->New...

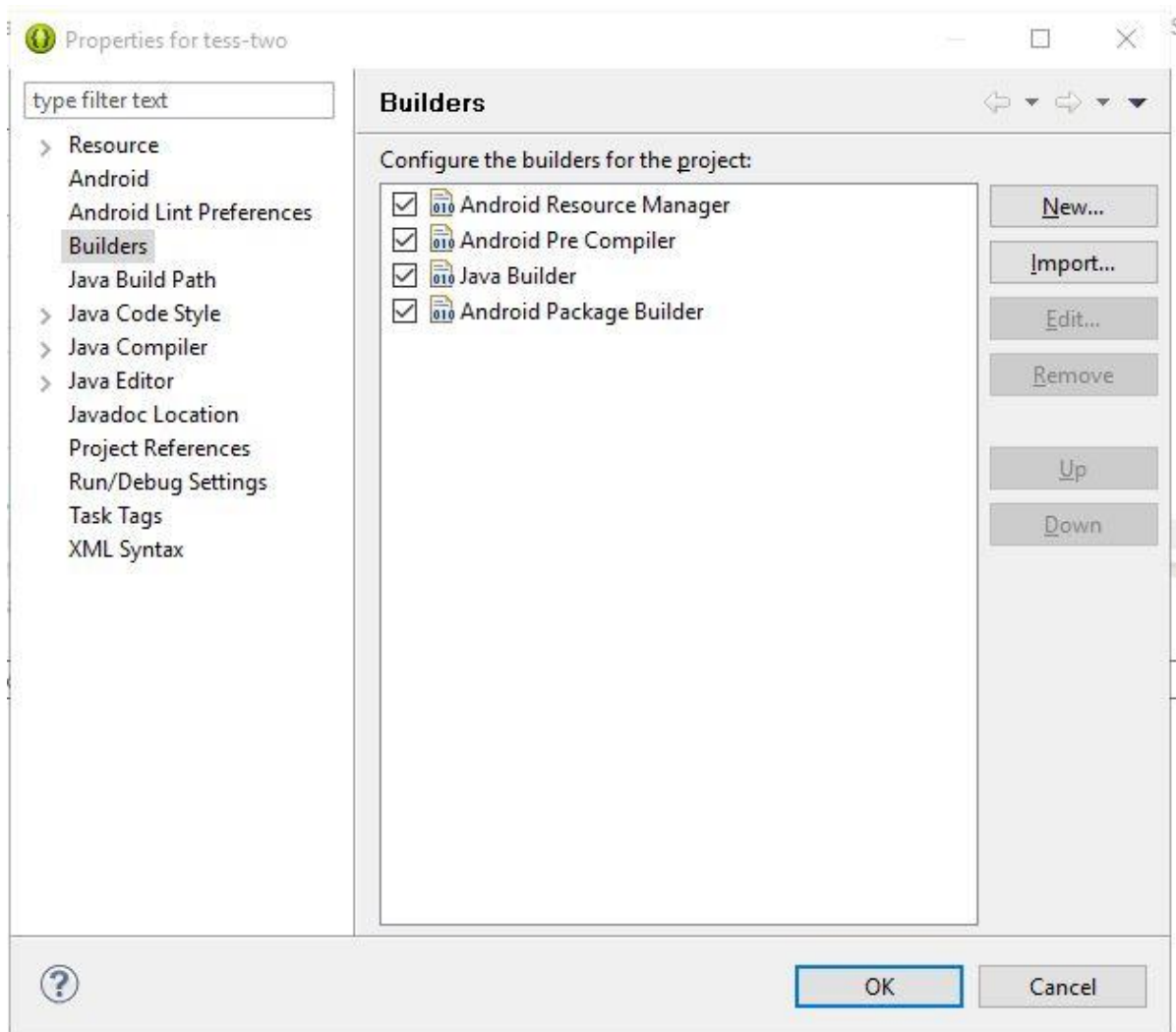


Figure 1: Properties for tess-two

- b. A new dialog will open presenting a list of builder types. We select the Program type and press the OK button.

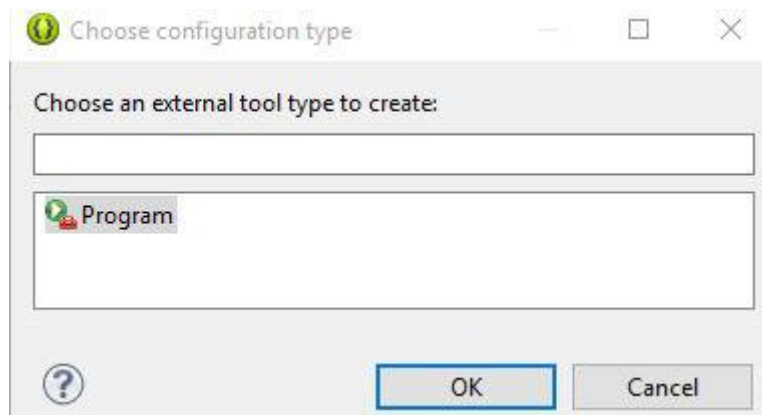


Figure 2: Configuration type selection

c. In the Main tab, fill in the following:

- i. Name: NDK Builder
- ii. Location: /opt/android-ndk/ndk-build (or wherever your ndk-build binary is). We may also use a variable as in `${system_property:user.home}/lib/android-ndk/ndk-build`
- iii. Working Directory: `${workspace_loc:/DristeeOCR}` where DristeeOCR is our project name.

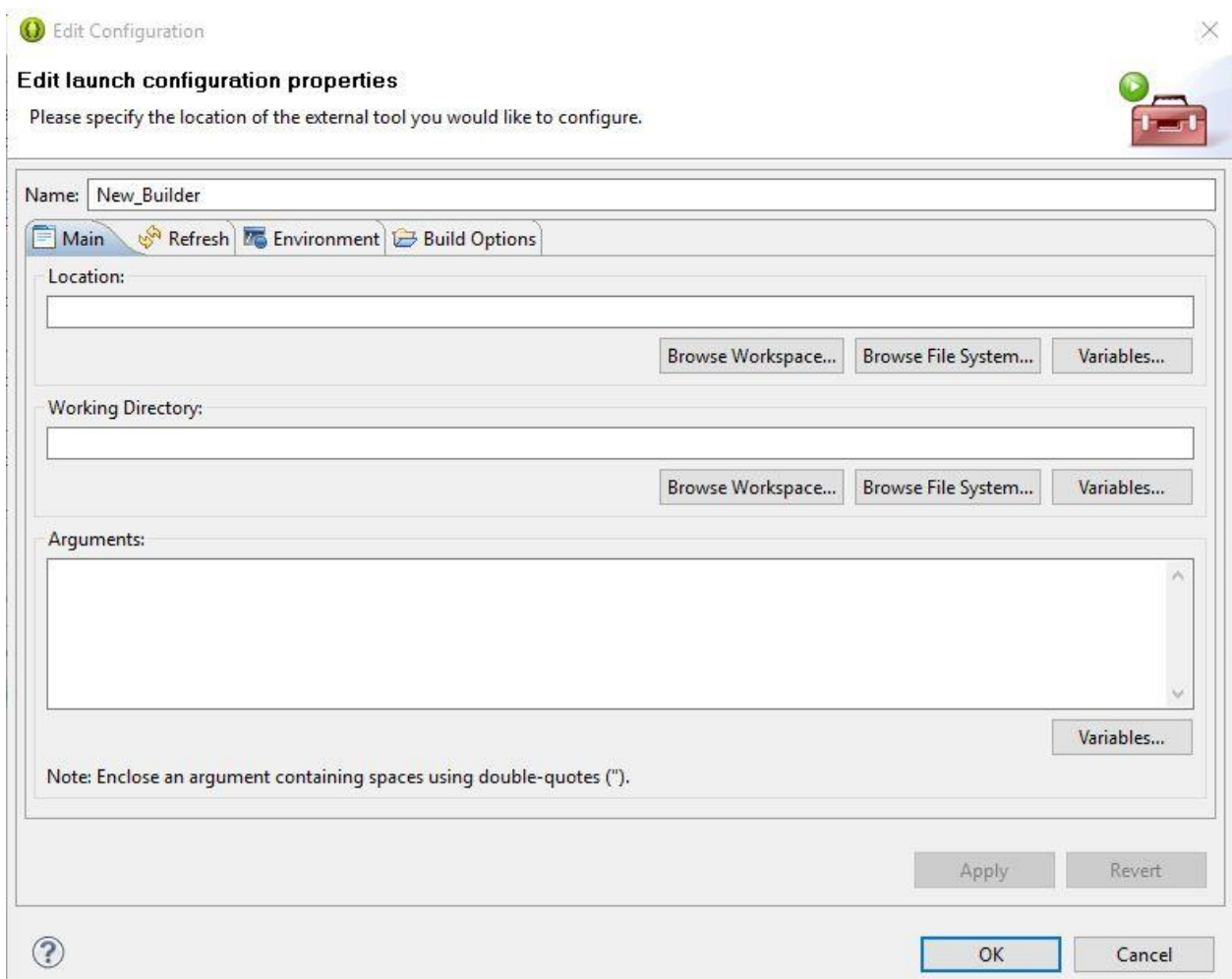


Figure 3: Configuration Window

d. After filling in the details, we click OK.

e. Setting the ndk-build.cmd command in the parameter builds the native programs for us.

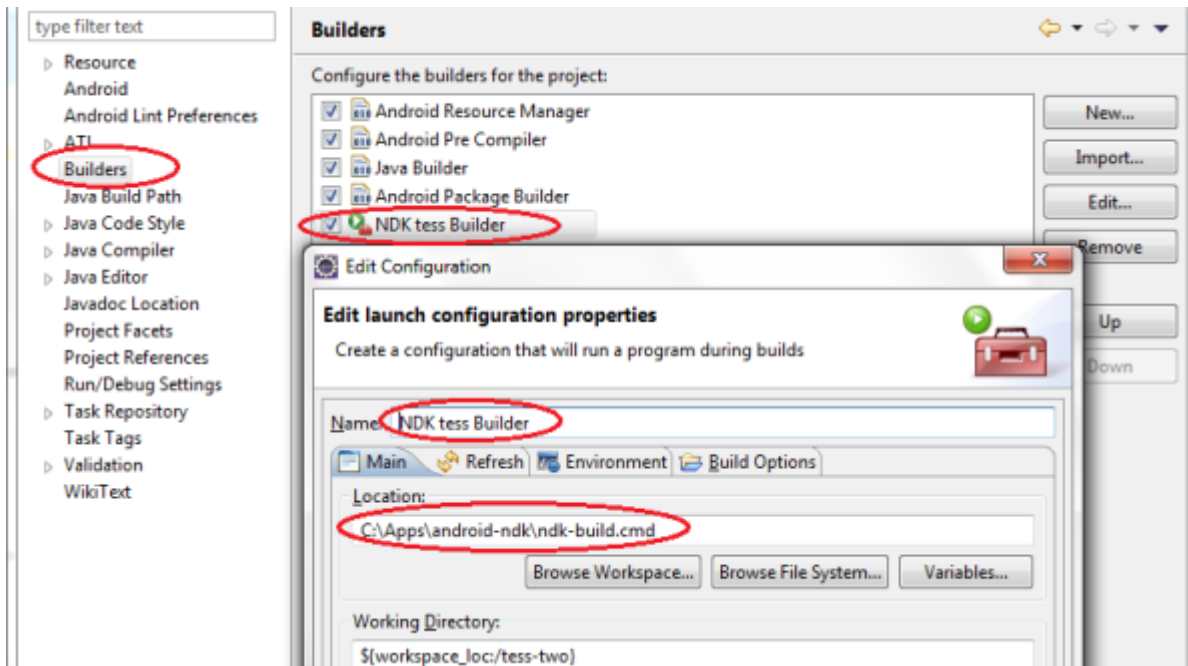


Figure 4: NDK Builder Selection

Next we must make sure that the Is Library check box is checked in the Android menu.

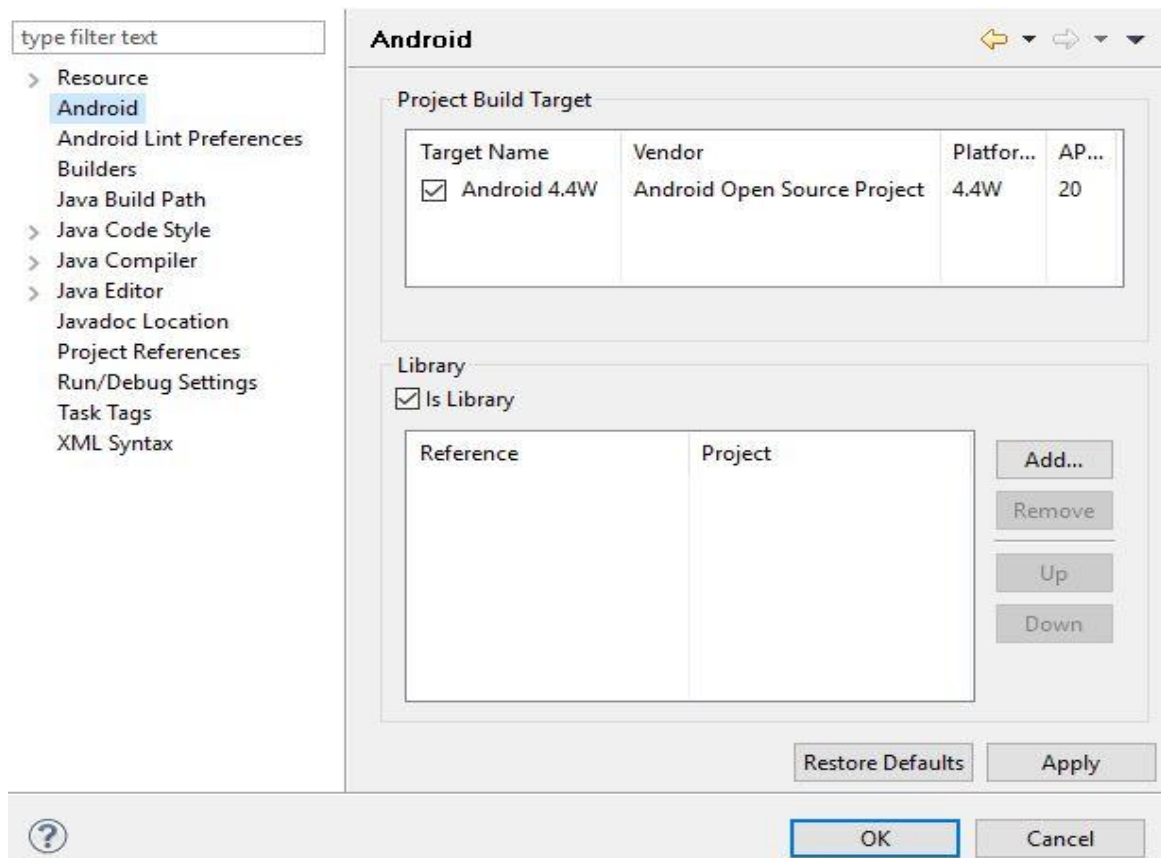


Figure 5: Tess-two Library Selection

After completing the above process, we finish setting up the basic requirements for developing a project with the two native programs Tesseract and Leptonica Image Processing Library.

3.4 The New Tesseract 3.03

For using the newer version of Tesseract, which is 3.03, we need some additional libraries to build the training tools.

```
sudo apt-get install libicu-dev  
  
sudo apt-get install libpango1.0-dev  
  
sudo apt-get install libcairo2-dev
```

3.4.1 Building the training tools

For compiling Tesseract from source we need to make and install the training tools with separate make commands. Once the above additional libraries have been installed, we run the following from the Tesseract source directory:

```
make training  
  
sudo make training-install
```

To train for another language, we had to create some data files in the tessdata subdirectory, and then crunch these together into a single file, using `combine_tessdata` command. The naming convention is `languagecode.file_name` Language codes for the released files following the ISO 639-3 standard.

The files used for Bengali are:

```
tessdata/ben.config  
  
tessdata/ben.unicharset  
  
tessdata/ben.unicharambig  
  
tessdata/ben.inttemp
```

```
tessdata/ben.pffmtable  
tessdata/ben.normproto  
tessdata/ben.punc-dawg  
tessdata/ben.word-dawg  
tessdata/ben.number-dawg  
tessdata/ben.freq-dawg
```

Finally, after crunching the above files we get the following file:

```
tessdata/ben.traineddata
```

The traineddata file is simply a concatenation of the input files, with a table of contents that contains the offsets of the known file types.

3.5 Software used for training

3.5.1 jTessBoxEditor

jTessBoxEditor v1.4 is a Java based software created by a Vietnamese company, VietOCR. It is a box file editor and trainer for Tesseract OCR, which provides the functions to edit the box data in both Tesseract 2.0x and 3.0x formats and full automation of Tesseract training. It can read images of common image formats, including multi-page TIFF. This program requires Java Runtime Environment 7 or later to operate. This program was used by Gajoui and Banerjee in their research to train Tesseract [13], [14], so we decided to use it as well.

The editor mode of jTessBoxEditor requires us to provide the TIFF/Box files as input. The images to be used in training should be of 300 DPI and 1 bpp (bit per pixel) black and white or 8 bpp grayscale uncompressed TIFF format. For box files, the file needs to be encoded in UTF-8 format which can be generated by Tesseract executables with appropriate command-line options or they can also be created using the built-in TIFF/Box Generator of jTessBoxEditor.

For our project we have made extensive use of the TIFF/Box Generator function only. For a given input UTF-8 text file, the generator produces an image in TIFF format along with a Box file. The box contains the mapping of the characters that were in the text file. The generated

image is, depending on anti-aliasing mode enabled, a binary or 8-bpp grayscale, uncompressed multi-page TIFF with 300-DPI resolution. We have included noise in our training image, so that it results in better trained data. Letter tracking, or spacing between characters, can be adjusted to eliminate bounding box overlapping issues. Overlap in the boxes causes huge problems for Bengali characters. As we know, the characters in the Bangla alphabet are not uniform in shape. So we had to be very careful when creating the character boundaries in the box files.

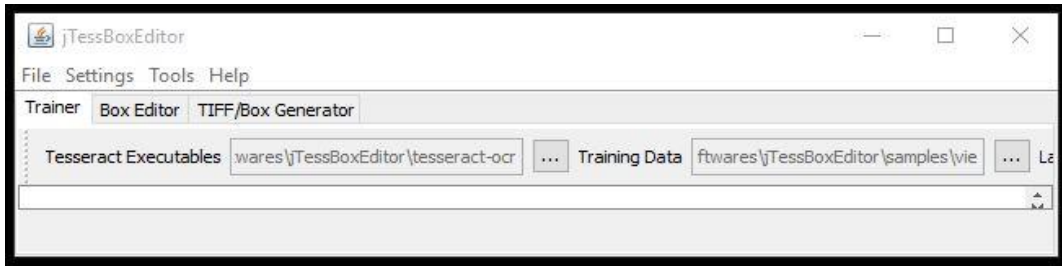


Figure 6: Outlook of jTessBoxEditor

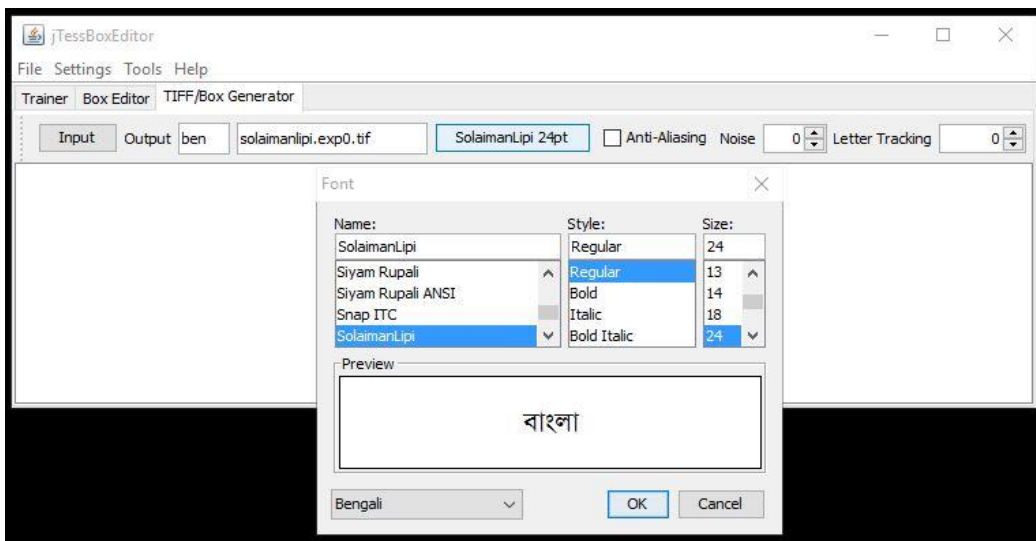


Figure 7:Font selection in jTessBoxEditor

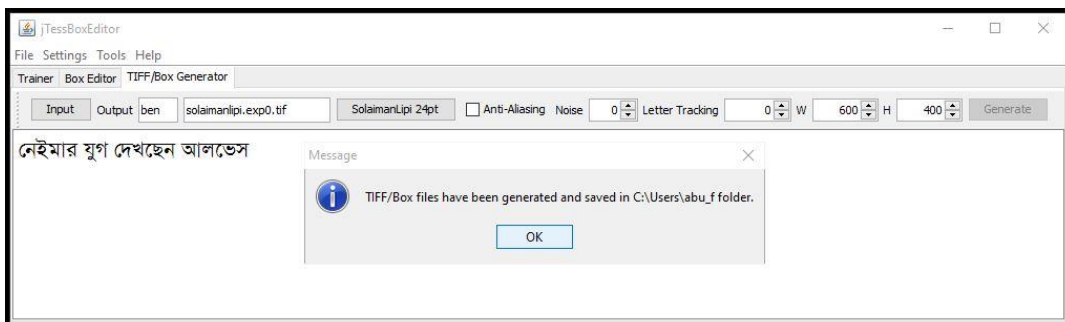


Figure 8: Generating Tiff and Box files using jTessBoxEditor



Figure 9: Generated Tiff and Box files for SolaimanLipi font.

3.5.2 QT Box Editor

The QT Box Editor is a tool for adjusting tesseract-ocr box files. The aim of this project was to provide an easy and efficient way for editing regardless of file size. The QT Box Editor is a successor of the tesseract-gui project that is no longer in development. This software is used to edit already created box files. We have used this software instead of jTessEditor because QT is much more sophisticated. Editing boxes in bulk is a very time consuming job. However, with QT the amount of time needed to process each box file is reduced significantly. Furthermore, QT has some very useful functions like insert, merge, split and delete right at the tip of our mouse. However, multipage TIFF is not supported yet and we must make sure to use TIFF with compression or PNG due to image quality and space. QT was also used by Banerjee in their research [14].

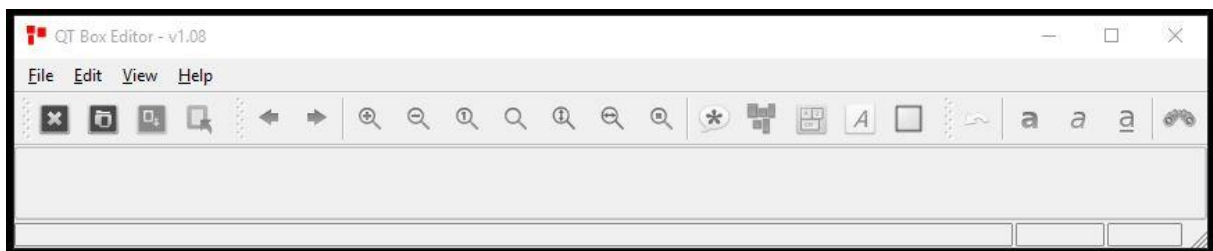


Figure 10: Outlook of QT Box Editor

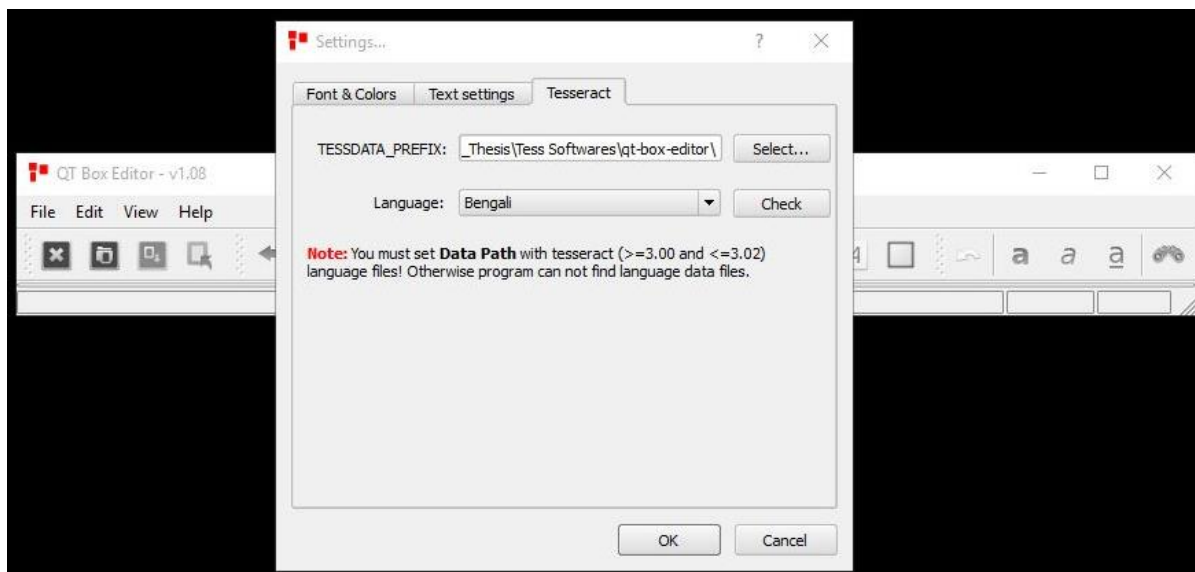


Figure 11: Setting Bengali language in QT Box Editor

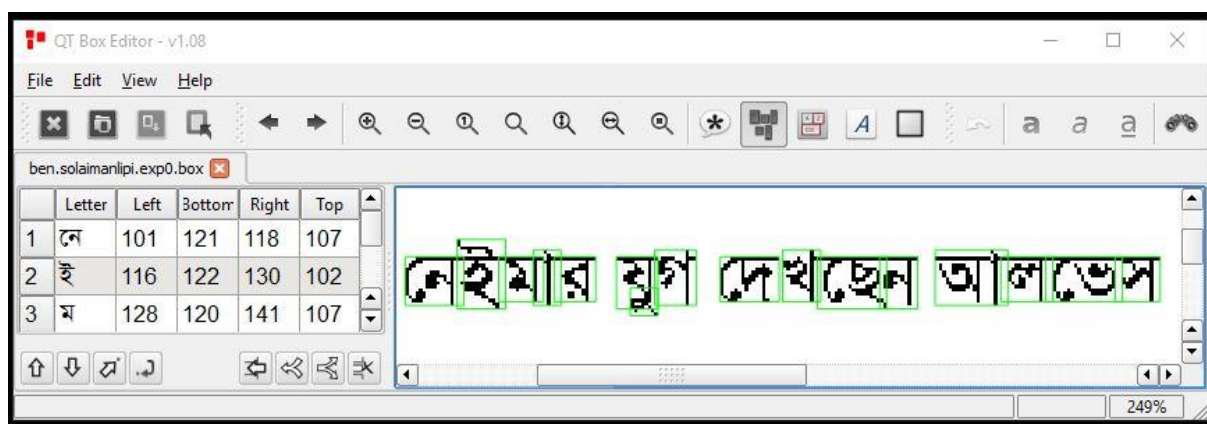


Figure 12: Editing box files in QT Box Editor

3.5.3 Serak Tesseract Trainer v0.4

Serak Tesseract Trainer is a front end GUI for Training Tesseract 3.02. We have basically used Serak to automate the training process in the Windows OS Environment. Serak has the feature to create traineddata files using only TIFF/Box file pairs. We have used Serak at the very end of our project to combine all the TIFF and box files and create a traineddata for the Tesseract engine. It is very useful when we are dealing with a large number of files. In our case, we had a pretty big character set. So, manually executing all the process and functions in creating a traineddata file is very inefficient and would require a lot of effort.

After successfully creating the traineddata file, we feed it to Serak and use its OCR Mode to run Tesseract on images to detect characters. Using Serak’s OCR Mode we are able to test and verify the integrity of the data set we had trained using the TIFF and box files.



Figure 13: Overview of Serak Tesseract Trainer

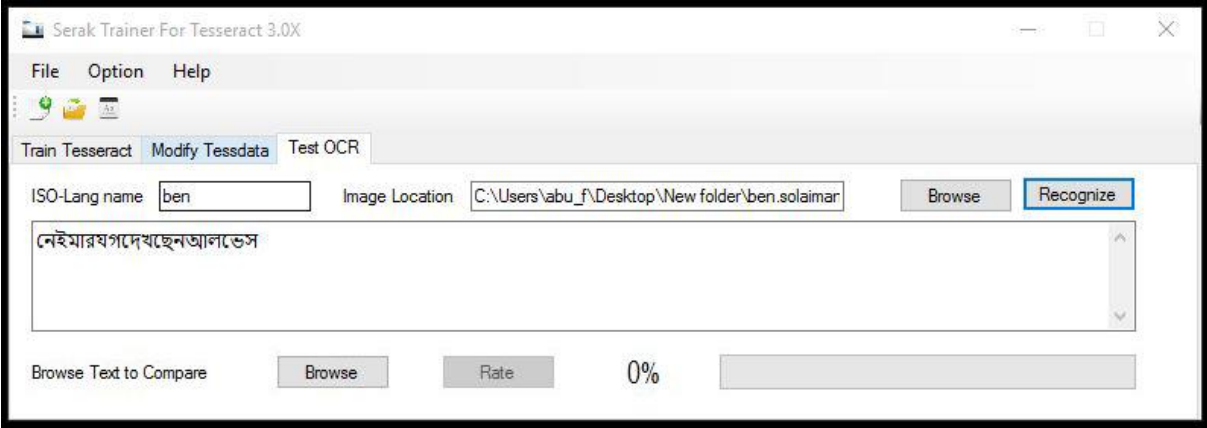


Figure 14: OCR Mode of Serak Tesseract Trainer

4. System Approach

4.1 Work flow:

The development process of Dristee OCR has two phases. Firstly, the training part where we create a database for the OCR to use, and detect Bengali characters from images. Secondly the Android application which uses the traineddata file and recognizes Bengali text and converts the recognized text into an editable and searchable document.

4.1.1 Training Dristee OCR

Our project is entirely dependent on the quality and quantity of the text we have used for training. While training, we had to be very cautious that we do not make any mistake in the character sets. This is because what we may think of as a slight error, may end up significantly decreasing the accuracy of our application. Next, the character set itself is vital, as we have to cover all the existing Bengali characters.

The number of Bengali characters is very large, if we consider the joint letters for training. As we are trying to increase the accuracy, we have included all possible joint letters in our training data. After preparing the text data for training, we convert them into images using jTessBoxEditor. We get four sets of training data file for four fonts. Consequently, using QT Box Editor we make changes necessary for the box files, making sure that the characters are correctly mapped inside the boxes.

Additionally, it is essential that there are no overlapping boxes in the training set, as it may create a shapeclustering error in later stages of training. Finally, after successfully creating the box files, we use Serak Tesseract Trainer to automatize the training process and combine the traineddata file using the box files.

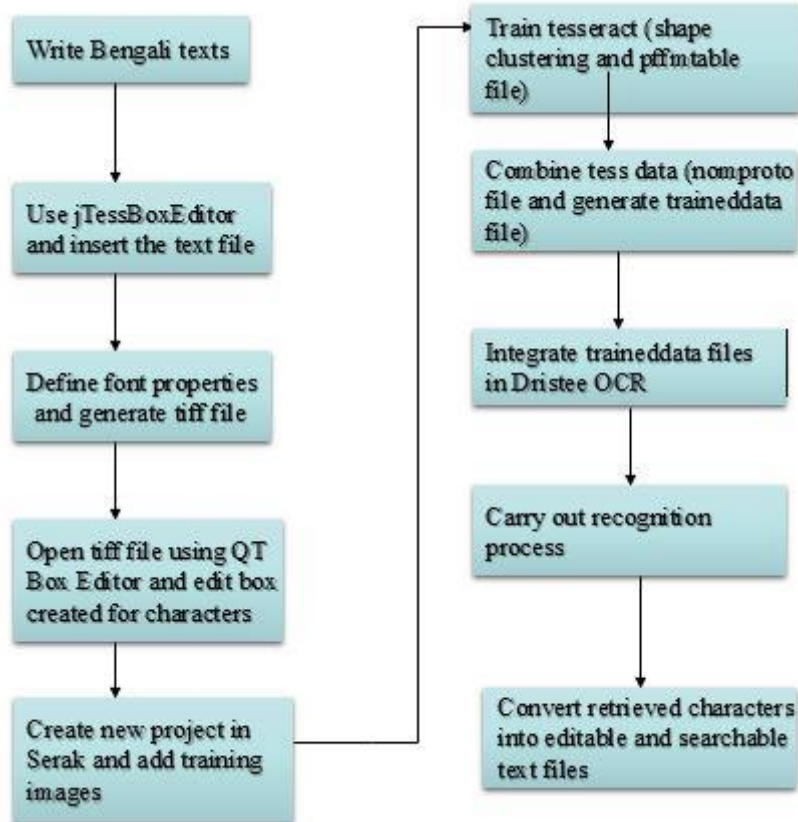


Figure 15: Training workflow of Dristee OCR

4.1.2 Executing Dristee OCR

As we want to develop a real time system for character detection, we will be using the video capture interface of the camera on our Android Phone. The calibration of the resolution will be done so that it can avoid any kind of distortion like parallax error. After that, some simultaneous internal processes will be executed like capturing the image and preparing it for the OCR. Concurrently, Leptonica Image Processing Library will work and hand over the data to Tesseract for character recognition.

Finally, when the recognition is done by properly matching with box files and corresponding Unicode Character, the result will be converted to text format and shown in the activity window of the application on the Android device. The text file will then be saved in the device for future use.

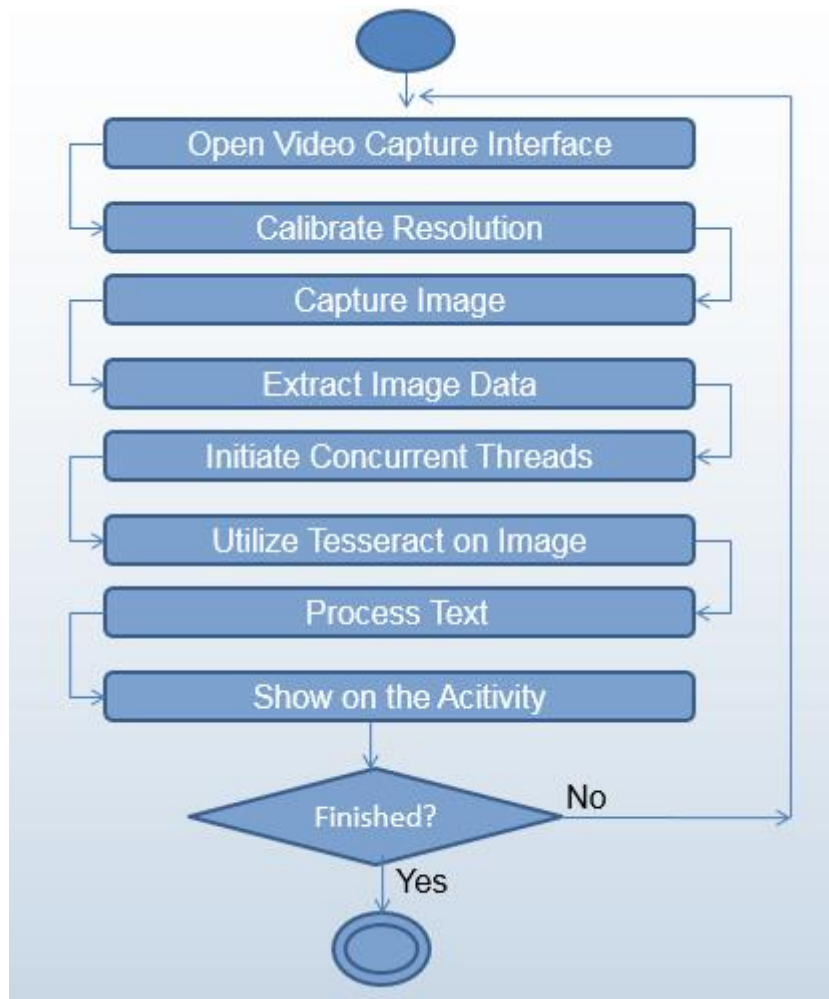


Figure 16: Workflow diagram

5. System Implementation

5.1 Generating image files from text files

To start the training process of our application, we need images to train with. So, we had to generate good quality image files from text. For this purpose, we used jTessEditor. By using the TIFF/Box Generator method, we easily converted text to images in TIFF format directly. TIFF format image is recommended for training Tesseract. The software jTessEditor also creates Box files of the characters given in the text files. Here, we provided the input as txt file for raw text in the text box. We had trained for a total of four fonts. They were: AdorshoLipi, Kalpurush, Nikosh and SolaimanLipi. In jTessBoxEditor, we also added "Noise" to artificially add noise to our training image. Adding noise increases the detection level of the OCR. For our research, we used noise value of 5 and the font size 12. Later on, we increased our data set using different font sizes such as, 14, 18, 24, 36.



Figure 17: Text input in jTessBoxEditor

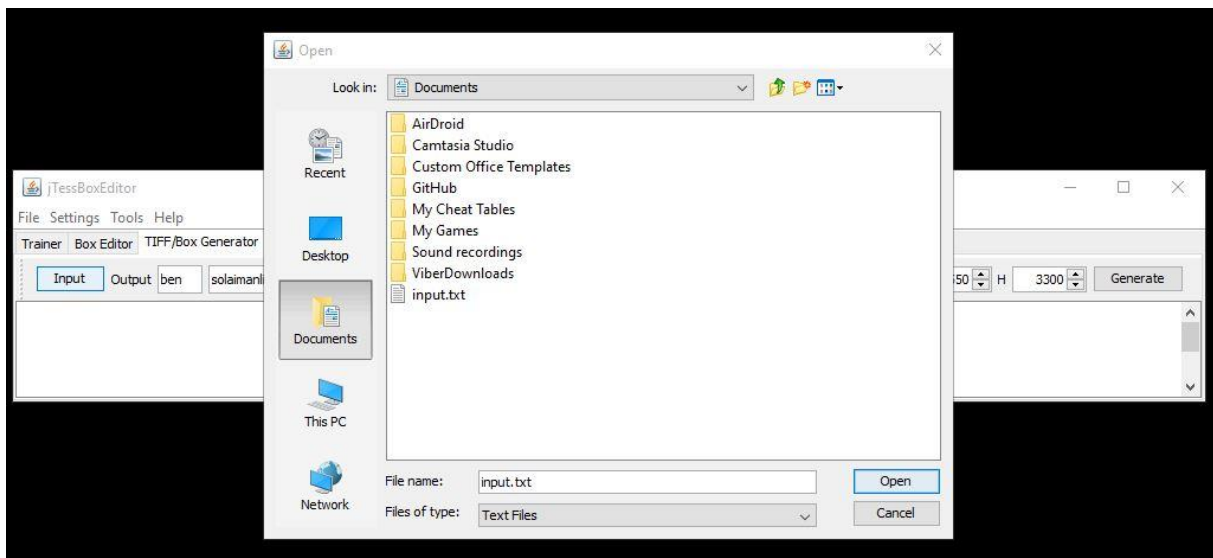
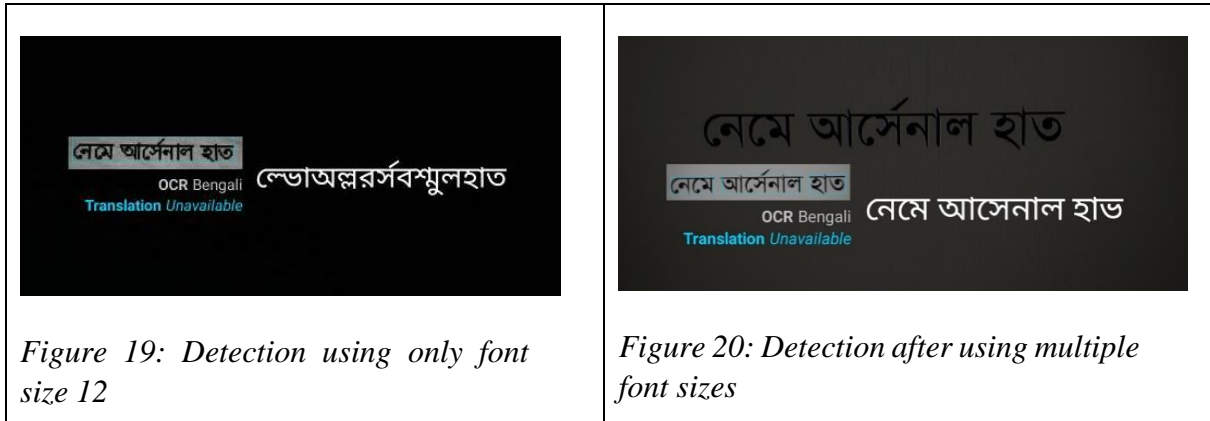


Figure 18: Text input using txt file

We have found that using a variety of font sizes help increase the accuracy of the OCR. Initially, we only used font size 12 for all the texts. However, using the trained data, created using these images resulted in poor accuracy. Therefore, we decided to increase the font size and create new sets of training images and box files. With these new trained data, we were able to increase the recognition to some extent.



5.1.1 Types of characters used for training

We have tried to cover all the characters present in the Bengali alphabet, however, we have not included Bengali numbers in our final data set. During the testing phase, we found that including number increases the ambiguity of the characters. Several other issues were addressed by Datta [15] in his paper. While most numbers were harmless except for "২" and "৩", the OCR is quite frequently confused between these two numbers. It is also seen to be confusing অ and ত with ৩ on several occasions. This results in decreased overall accuracy. Moreover, we choose not to train the vowel diacritics (া ি ী ু ৃ ে ঐ ো ৌ...) individually either. Training with these characters also affects the detection level of the OCR. This fact was already confirmed by Zaman [6] in their paper. So, we concluded that it would be better to associate these diacritic characters with consonants forcefully and train them for our OCR application. More information on the properties of different Bengali scripts can be found at [12]. Lastly, our whole research concentrated on the successful detection of conjunct consonants. No work has been done to detect these joint letters precisely.

ই, ঈ -> ২
অ, ত -> ৩

Table 1: Numerical Ambiguities

Bangla Vowels	অ আ ই ঈ উ ঊ ঋ এ ঐ ও ঔ
Bangla Consonants	ক খ গ ঘ ঙ চ ছ জ ঝ ঞ ট ঠ ড ঢ ণ ত থ দ ধ ন প ফ ব ভ ম য র ল শ ষ স হ ড় ঢ় য় ং ঃ ঁ
Bangla Conjunct Consonants	ঠ ড ঢ ণ ত থ ধ ঙ ঝ ঞ ড ঢ...
Bangla Consonants with diacritics	কা খা ঠি ধী চু কু জু ঢে থে গো ষৌ...

Table 2: Types of characters trained

We have trained a total of 12,191 individual Bengali characters for our Dristee OCR application. This includes all four fonts which consists of all possible Bengali characters. The majority of this data set consists of conjunct consonants as they are huge in number, 8971 to be precise.

অ আ ই ঈ উ ঊ ঋ এ ঐ ও ঔ ক খ গ ঘ ঙ চ ছ জ ঝ ঞ ট ঠ ড ঢ ণ ত থ দ ধ ন প ফ ব ভ ম য র ল শ ষ স হ ড় ঢ় য় ং ঃ ঁ	নেমে আর্সেনাল হাত প্রত্যেকটি উইকিপিডিয়া করার প্রকল্পে। মুখোমুখি প্রোগ্রাম কন্ট্রোল ব্যবহার আর্সেনাল ট্রেন সদস্যরা কে ভূমিকা গুমারি তুর গুরু জার্নাল কৌজের ড্র সংখ্যক মারো সমে এবং প্রাজমা উক্তি পশ্চিমবঙ্গ নদীগর্ভে গম্ভীর বর্ধিত কাজে বাংলা-বিহার-উড়িষ্যার সহায়তা অব নামে আহুবান চা ডেথ র
কি ষি পি লি চি ছি জি ঝি টি ঠি ডি নি তি থি দি ধি নি পি ফি বি ভি মি কী খী গী ঘী ঙী চী ছী জী ঝী টী ঠী ডী নী তী থী দী ধী নী পী ফী বী ভী মী কু খু গু ঘু ঙু চু ছু জু ঝু ঞু টু ঠু ডু ঢু ণু তু থু দু ধু নু পু ফু বু ভু মু যু রু লু শু ষু স কু খু গু ঘু ঙু চু ছু জু ঝু ঞু টু ঠু ডু ঢু ণু তু থু দু ধু নু পু ফু বু ভু মু যু রু লু শু ষু স কে ষে পে চে ছে জে বে টে ঠে ডে নে তে থে দে ষে নে পে ফে বে ডে কৌ খৌ গৌ ঘৌ ঙৌ চৌ ছৌ জৌ ঝৌ টৌ ঠৌ ডৌ নৌ তৌ থৌ দৌ ধৌ নৌ পৌ ফৌ বৌ ভৌ মৌ	ত তা তি তী তু তূ ত্ত তে তৈ তো তৌ থ থা থি থী থু থূ থ্ত থে থৈ থো থৌ দ দা দি দী দু দূ দ্ত দে দৈ দো দৌ ধ ধা ধি ধী ধু ধূ ধ্ত ধে ধৈ ধো ধৌ ন না নি নী নু নূ ন্ত নে নৈ নো নৌ প পা পি পী পু পূ প্ত পে পৈ পো পৌ ফ ফা ফি ফী ফু ফূ ফ্ত ফে ফৈ ফো ফৌ ব বা বি বী বু বূ ব্ত বে বৈ বো বৌ ভ ভা ভি ভী ভু ভূ ভ্ত ভে ভৈ ভো ভৌ ম মা মি মী মু মূ ম্ত মে মৈ মো মৌ

Table 3: Various types of training images

5.2 Creating Box Files

After converting the text into images, we must now create box files for them. Box files are simply the mapping of the characters in the images. We are using QT Box Editor for creating and editing the box files. It is required to merge character likes ক, া into কা as we do not want the vowel diacritics to be separate. Doing this in jTessBoxEditor is troublesome, therefore we used QT.

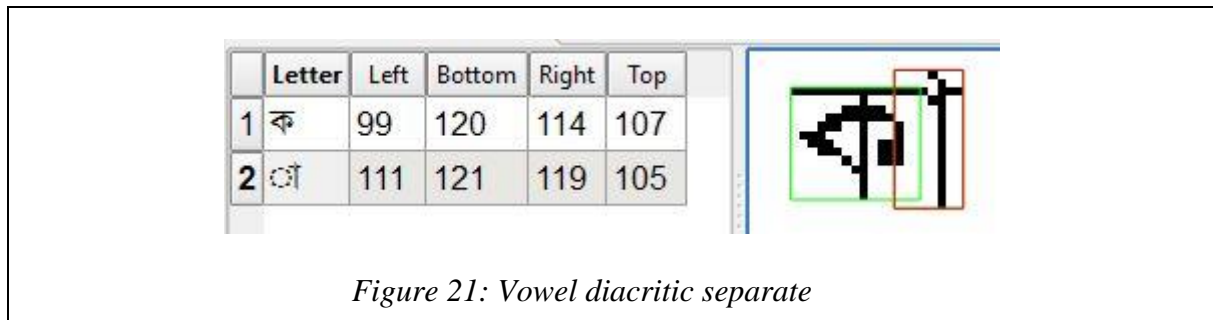


Figure 21: Vowel diacritic separate

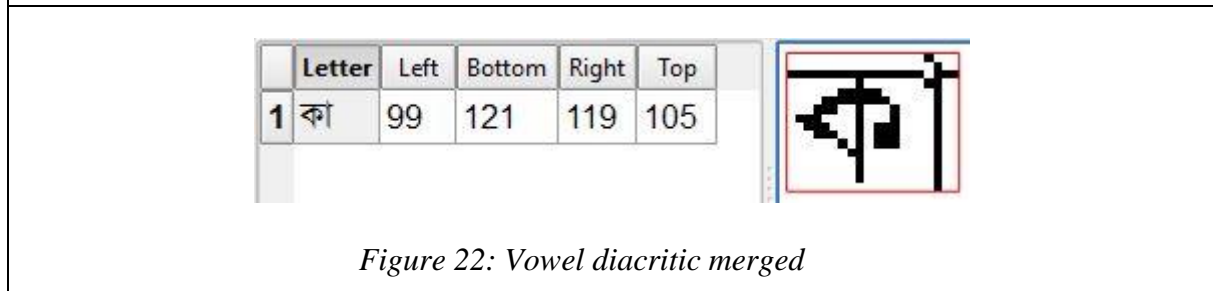


Figure 22: Vowel diacritic merged

Table 4: Diacritic overview

The Box files contain the characters in the image in Unicode format as well as their corresponding "Left, Bottom, Right, Top" position with respect to the dimensions of the images.

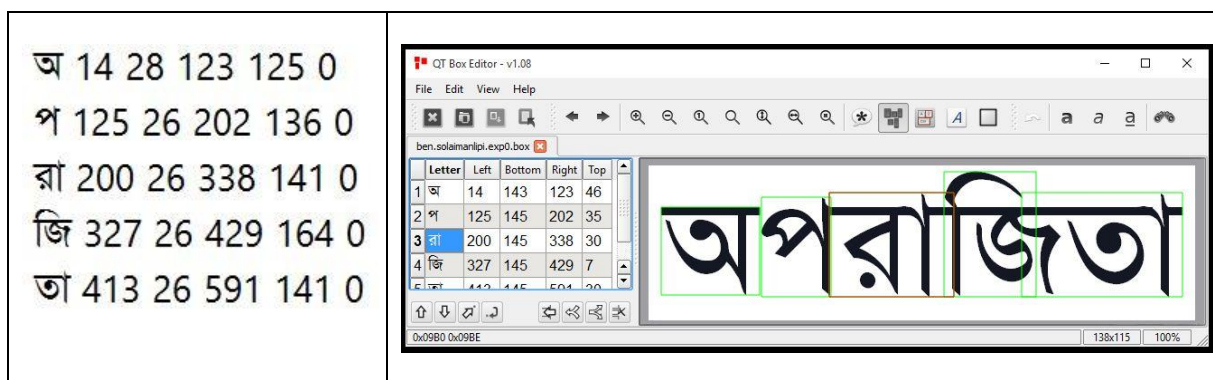


Figure 23: Box file creation

5.3 Processing Box Files

After we have created all the box file pairs, we need to run Tesseract on each of the images and corresponding box files. For each of the images and box file pairs, we must execute the following command:

For Windows Platform Exclusively:

```
tesseract [lang].[fontname].exp[num].tif [lang].[fontname].exp[num] box.train
```

```
tesseract ben.solaimanlipi.exp0.tif ben.solaimanlipi.exp0 box.train
```

For All Platforms:

```
tesseract [lang].[fontname].exp[num].tif [lang].[fontname].exp[num] box.train.stderr
```

```
tesseract ben.solaimanlipi.exp0.tif ben.solaimanlipi.exp0 box.train.stderr
```

However, it is not feasible to run this command for a large number of training images. It takes both effort and time to manually execute all the commands. Therefore, we have used Serak Tesseract Trainer to automate the task for us. Serak runs this command relentlessly until all the images and box files are processed.

Executing this command line creates a [lang].[fontname].exp[num].tr file for every pair. Eg. ben.solaimanlipi.exp0.tr.

5.4 Computing the Character Set

Next, we must create a unicharset data file, which lets Tesseract know the set of possible characters it can output. For this we need to use the unicharset_extractor program on the box files generated above. However, unlike previous processes, we do not need to type the same command over and over for every box file. We can just append the name of the box files separated by a space.

```
unicharset_extractor lang.fontname.exp0.box .... lang.fontname.expN.box
```

```
unicharset_extractor ben.solaimanlipi.exp0.box .... ben.solaimanlipi.expN.box
```

```

অ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 1 0 0 # # অ [985 ]x
আ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 2 0 0 # # আ [986 ]x
ই 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 3 0 0 ## ই [987 ]x
ঈ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 4 0 0 # # ঈ [988 ]x
উ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 5 0 0 # # উ [989 ]x
ঊ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 6 0 0 # # ঊ [98a ]x
ঋ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 7 0 0 # # ঋ [98b ]x
এ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 8 0 0 # # এ [98f ]x
ঐ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 9 0 0 # # ঐ [990 ]x
ও 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 10 0 0 # # ও [993 ]x
ঔ 1 0,255,0,255,0,32767,0,32767,0,32767 NULL 11 0 0 # # ঔ [994 ]x

```

Figure 24: Unicharset sample

5.4.1 Setting the Unicharset Properties

A new tool and set of data files in 3.03 allow the addition of extra properties in the unicharset, mostly sizes obtained from different fonts.

```

training/set_unicharset_properties -U input_unicharset -O output_unicharset --
script_dir=training/langdata

```

5.5 Font Properties

Subsequently, we need to generate the font_properties file. This file contains all the information about the style of the text. It controls the style the output receives after the font is recognized. The font_properties file is a text file specified by the -F filename option to mftraining.

Each line of the font_properties file is formatted as follows:

```

<fontname> <italic> <bold> <fixed> <serif> <fraktur>

```

Here, <fontname> is a string naming the font and <italic>, <bold>, <fixed>, <serif> and <fraktur> are all simple 0 or 1 flags indicating the respective state of the font.

Eg. solaimanlipi 0 0 0 0 0, means that the font name is solaimanlipi with no styling present.

5.6 Shape Clustering

When the character features of all the training pages have been extracted, we need to cluster them to create the prototypes. The character shape features can be clustered using the `shapeclustering`, `mftraining` and `cntraining` programs:

```
shapeclustering -F font_properties -U unicharset lang.fontname.exp0.tr ....  
lang.fontname.expN.tr
```

```
Eg. shapeclustering -F font_propterties -U unicharset ben.solaimanlipi.exp0.tr ....  
ben.solaimanlipi.expN.tr
```

Shapeclustering creates a master shape table by shape clustering and generates a `shapetable` file. As we are training for an Indic language, we are going to use the `shapeclustering` method. This method is only used for Indic languages.

```
mftraining -F font_properties -U unicharset -O lang.unicharset lang.fontname.exp0.tr ....  
lang.fontname.expN.tr
```

```
Eg. mftraining -F font_properties -U unicharset -O ben.unicharset ben.solaimanlipi.exp0.tr  
.... ben.solaimanlipi.expN.tr
```

The `-U` file is the `unicharset` generated by `unicharset_extractor` above, and `ben.unicharset` is the output `unicharset` that will be given to `combine_tessdata`. `Mftraining` will output to two other data files: `inttemp` and `pfmtable`. The `inttemp` file contains information about the shape prototypes, whereas, `pfmtable` contains the number of expected features for each character we are training.

```
cntraining lang.fontname.exp0.tr .... lang.fontname.expN.tr
```

```
cntraining ben.solaimanlipi.exp0.tr .... ben.fontname.expN.tr
```

This command will create the `normproto` data file which is the character normalization sensitivity prototypes.

5.7 Unicharambigs

During our training procedure, we observed a slight difference at some of the consonants with Bengali “kar” also known as the vowel diacritic. For instance, when we tried to detect ক it outputted ক্. This is an ambiguity that confuses the engine to identify a list of consonants with “kar”. To fix the problem, we went through a bit of research and came to know that there is a file called DangAmbigs which deals with a mapping to reduce complexity between ambiguous figures. Another example may be কা = কণ. It uses numbering to map the corresponding possibilities like the following image.

1	কা	2	কণ
2	ক্	1	কা
2	তা	1	অ

Figure 25: DangAmbigs

We generated this file in order to reduce detection failure in identical letter groups. In our version we are using a similar file called unicharambigs more specifically Tesseract unicharset ambiguities. This unicharambigs file first appeared in Tesseract 3.00. Before that the similar format as already mentioned DangAmbigs (*dangerous ambiguities*) was used. The structure was mostly similar but only obligatory changes could be defined. One extra column was added, holding either 1 or 0. Here, 1 means a mandatory replacement, 0 means an optional substitution. Initially we were facing problems with saving the Unicharambigs with proper configuration. So we used sublime text to maintain the Unicode format.

2	কা	2	কণ	0
1	অ	2	তা	1
2	তা	1	অ	0

Figure 26: Unicharambigs

Here in the demonstration we can see 2 characters are confused with two other characters, where non-obligatory replacement is indicated by 0. In the second line 1 character is confused with 2 characters and an obligatory replacement is indicated by 1 and so on.

5.8 The Final Crunch

Finally, all of our data files are created and ready for use. Now it was time to create the traineddata file. The traineddata file is simply the concatenation of all the data files we had created above. For this, we first sorted out the files and added the prefix "ben." to all of them.

Before	After
shapetable	ben.shapetable
normproto	ben.normproto
inttemp	ben.inttemp
pfmtable	ben.pfmtable
unicharset	ben.unicharset
unicharambigs	ben.unicharambigs

Table 5: Renaming with language prefix

Now, we were all set to generate the traineddata file. For this, we ran the command "combine_tessdata" on them as follows:

```
combine_tessdata lang.  
Eg. combine_tessdata ben.
```

After successfully executing this command, a new file, ben.traineddata will be created. This file acts as the fuel to our Android application.

However, if we are working on a Windows platform, then we can simply use the software Serak Tesseract Trainer we had discussed earlier to automate the process for us. Manual labor is greatly reduced if we use this software fruitfully.

6. System Programming

In our project, Dristee OCR, CaptureActivity.java is being used to capture the text image. The application checks the first launch and downloads the trained data from a server where we kept our trained data hosted. After initializing, the OcrInitAsyncTask.java initializes a concurrent thread to simultaneously process OCR on the text image in real time. The PreferencesActivity.java gives the option to the user to choose the fonts from a dropdown list which they prefer to use for detection.

The DecodeHandler.java utilizes the OcrRecognizeAsyncTask.java to continuously perform the detection for Bengali characters. PlanarYUVLuminanceSource.java utilizes the Leptonica Image Processing Library internally to adjust the brightness and contrast before passing the thread to Tesseract detection. It also uses the matrix manipulation of images and crops the image source in a grayscale image format for OCR. OcrResult.java provides the resultant text of detected Bangla Unicode font mapped results as text. This string is finally shown on the activity as detected text. The whole process is repeated continuously for real time detection.

From our observation we saw that the OCR result in lower light in case of printed text is not satisfactory. We did a bit of research and came to know about monochrome and colored light model from graphics to fix this issue. Finally, we came to a conclusion that using the camera flash will help us develop the result to a great extent. Screen images use emission based technology. On the other hand, printed document detection needs to be implemented with reflection based technology. In our project we introduced a package called bracu.ac.bd.ocr.camera, here we included the class CameraConfigurationManager.java. This class utilizes the methods doSetTorch and setFlashMode to enable flash.

Exposure is another issue to increase the quality of the image before detection and compromise contrast. This is dealt by CameraManager.java. Consequently, three exposure levels called low, medium and high are given as option to the user. The method setExposureCompensation is used to apply the exposure.

The class ViewfinderView.java is a view which is set on top of the camera preview which adds the viewfinder rectangle and slightly reduces opacity around it. The class CameraManager.java was implemented to wrap up the camera services used in the preview. For instance, we had

implemented the `adjustFramingRectangle` method which takes the height and width as parameters and helps calibrating frame size.


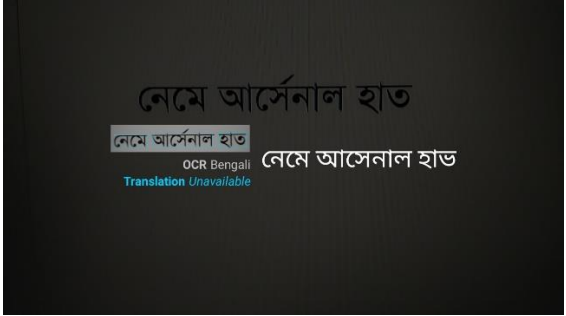
Before Exposure Control:	After Exposure Control:
	

Table 6: Comparison after exposure

From the above test images, we can observe that the exposure of the camera plays an important role in image quality, Hence, efficient detection of the characters depends on it. However, we must acknowledge the fact that not all cameras have the same exposure level. Therefore, we have margined the exposure level using the minimum and maximum exposure possible by the camera. Next, we manipulated this information in setting three exposure levels, low, medium and high [Appendix 2].

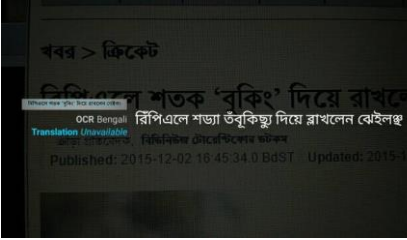
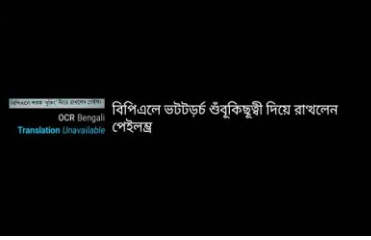
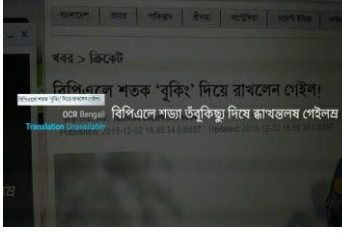
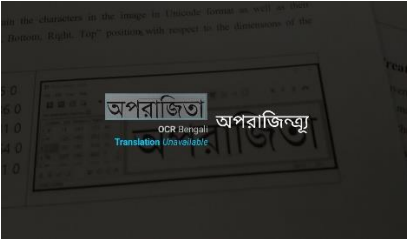
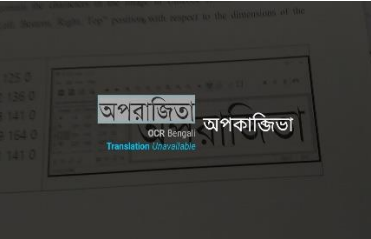
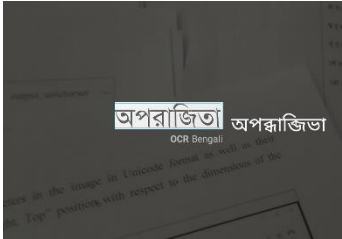
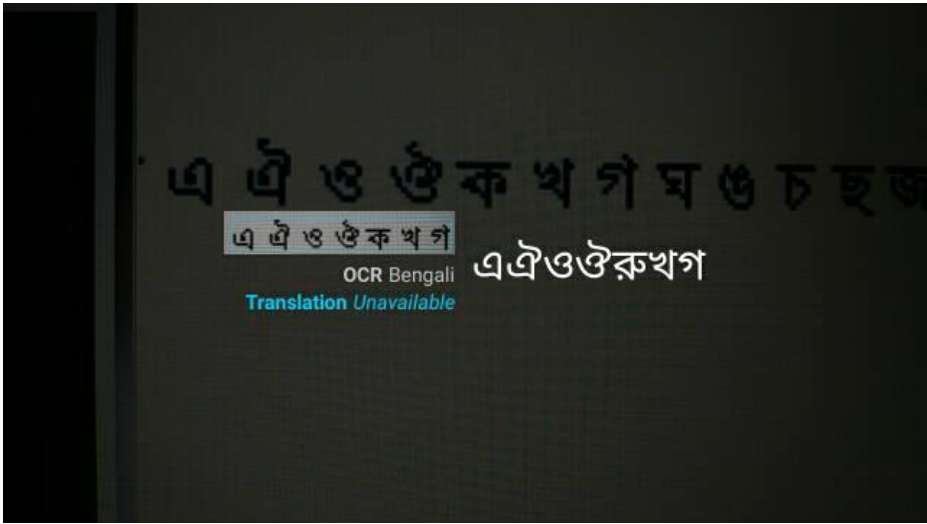
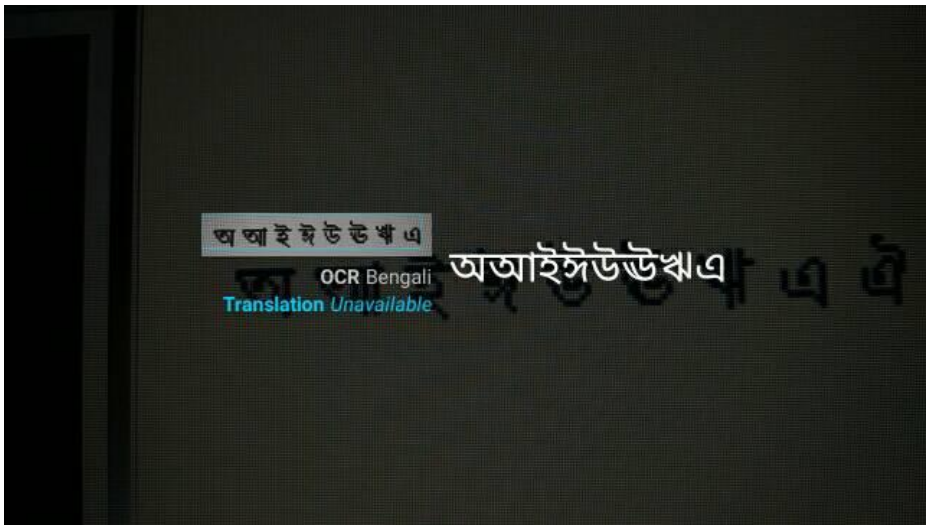
Exposure: Low	Exposure: Medium	Exposure: High
		
		

Table 7: Exposure modes

A few more sample of images after the exposure of the camera is adjusted manually.

After manually setting the exposure of the camera:



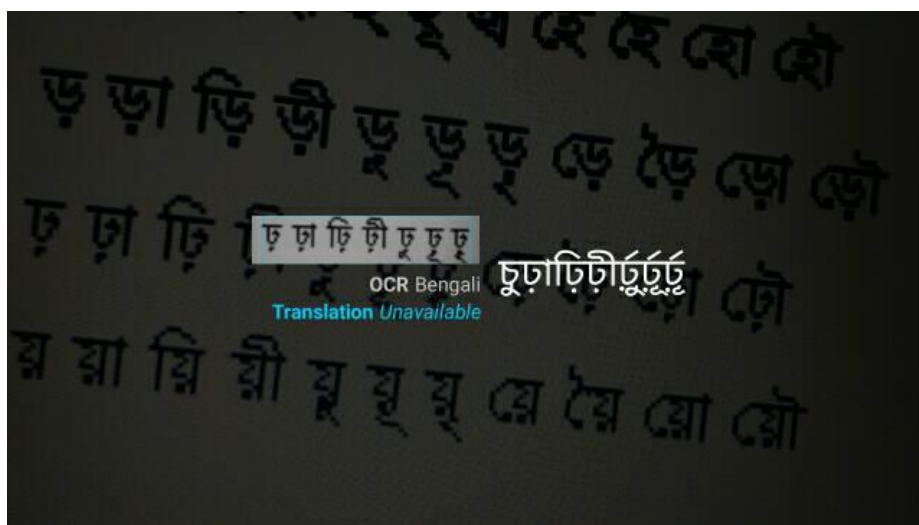
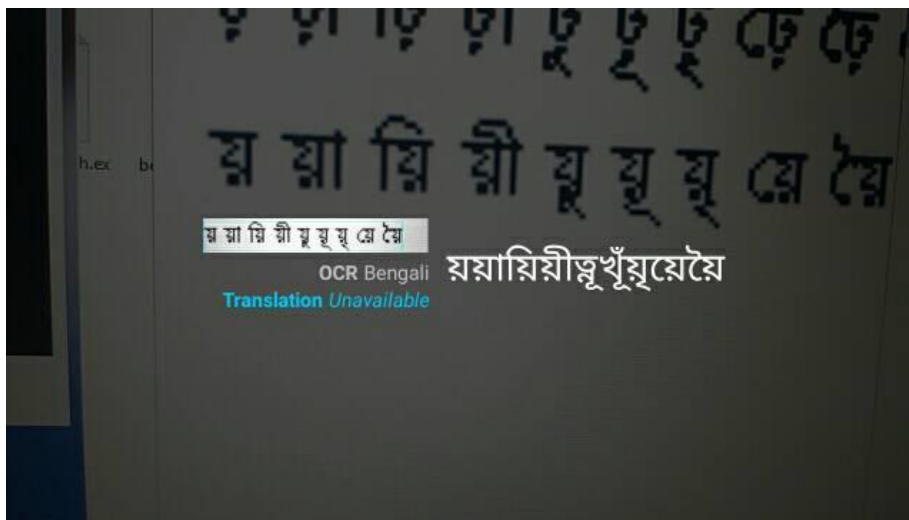
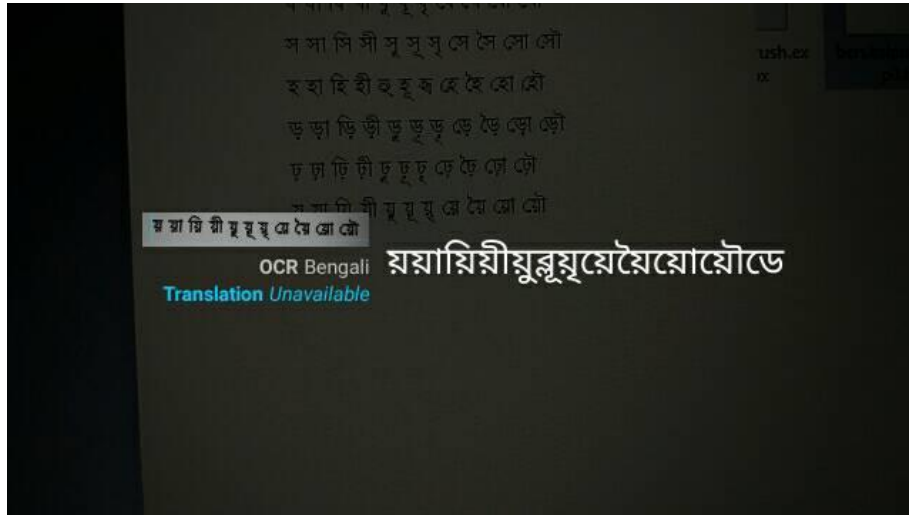


Table 8: Sample detection

7. Experimental Result

7.1 Precision and Recall

For our research and understanding training quality, we have tested our trained data in both desktop version and in our own Android application called “Dristee OCR”. Moreover, test was carried out using two sets of sample data. One set of data contained converted images from text files that we trained and the other set contained screenshots for desktop version and images clicked by our application from newspapers, magazines, etc. In order to calculate accurately, we used “precision” and “recall” [16].

This method is typically used in pattern recognition and information retrieval. Precision states the fraction of how many retrieved results are correct and are relevant and recall, on the other hand, gives an estimated fraction of how many positives is returned by the model. The following figures show pie chart and graphs for better illustration of these and their relationship.

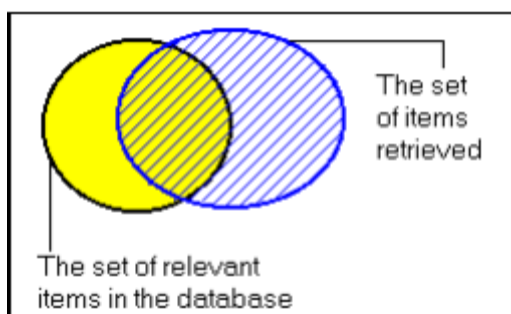


Figure 27: Relationship between retrieved data and total set of data.

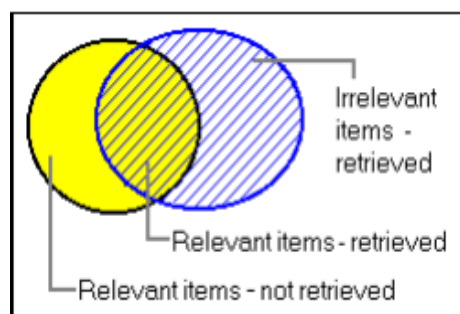


Figure 28: Relationship between retrieved, not retrieved and irrelevant data retrieved.

Let us assume, A is the total number of characters detected by the OCR, B is the total number of actual characters and C is the total number of correctly detected characters. Therefore, in our case, precision and recall is:

Precision: C/A

Recall: C/B

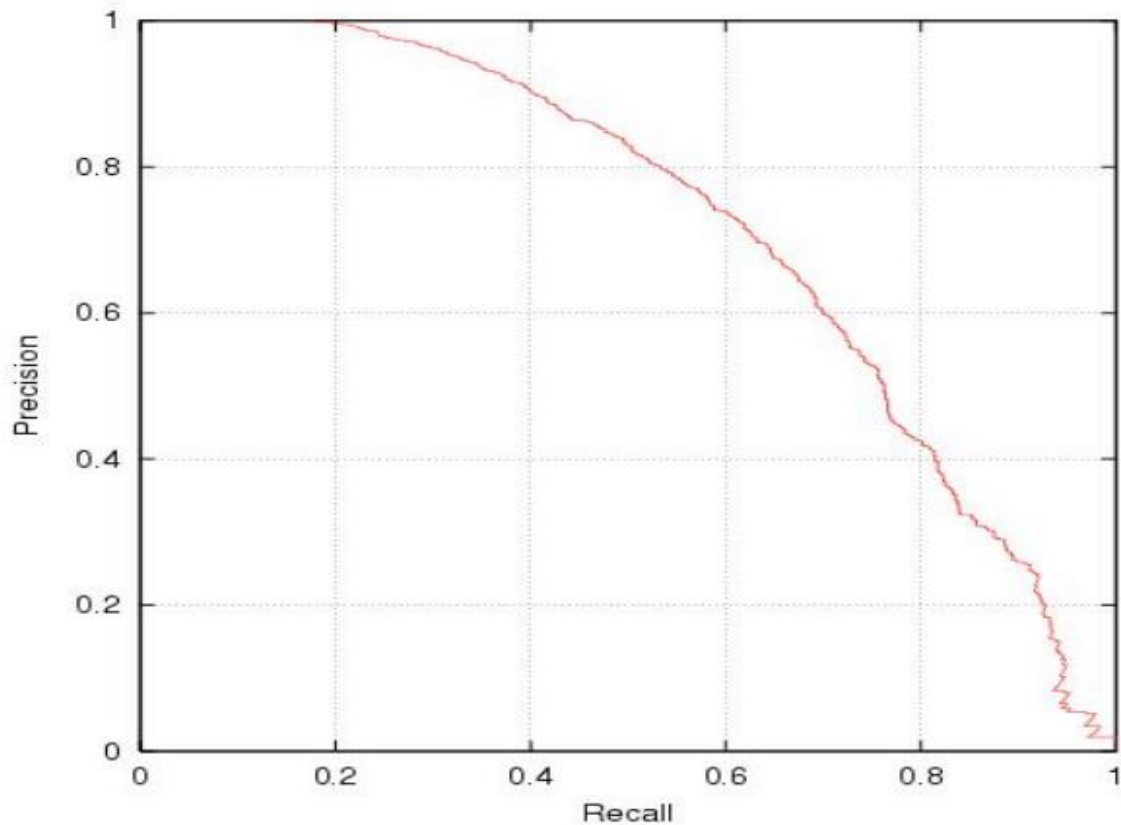


Figure 29: Relationship between precision and recall.

The graph in Figure 29 shows an inverse relation between the two terms, where precision decreases exponentially as recall increases. In this graph, a perfect precision score of 1.0 means every result retrieved by the search was relevant. However, it says nothing about whether or not all the relevant documents were retrieved.

Whereas, a perfect recall score of 1.0 means that all relevant documents were retrieved, but says nothing about the irrelevant documents retrieved by the search. As the value of precision decreases it is indicating how the relationship between the retrieved and the correct value deteriorates.

On the other hand, as the value of recall decreases it indicates that less relevant data were retrieved. Therefore, as the amount of relevant data increases, the possibility of irrelevant data goes up as recall does not keep count of the irrelevant ones.

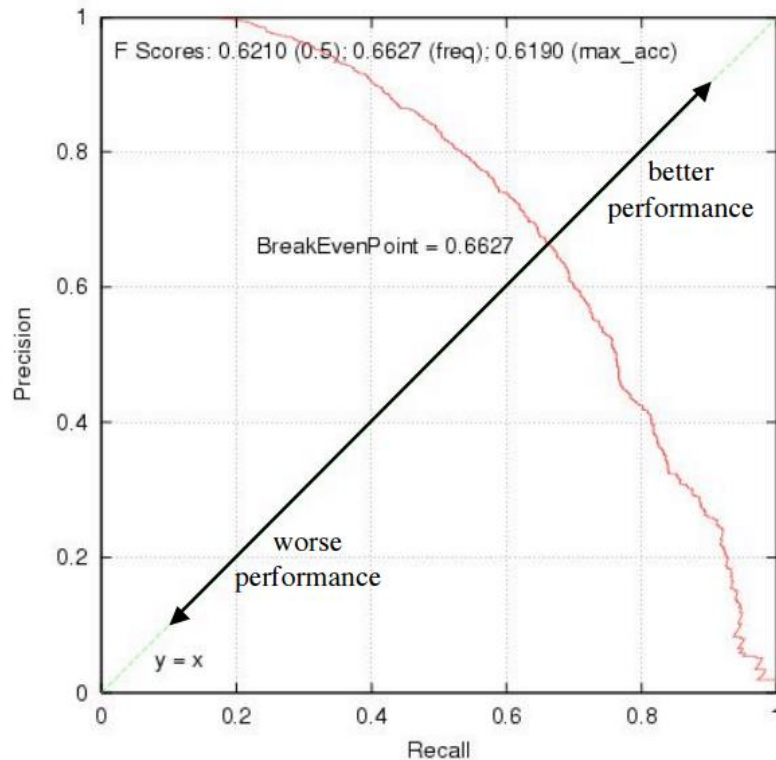


Figure 30: Harmonic average and Break-even points.

The properties or the expected behaviors of text categorization or information retrieval systems can vary. For example, for one system, it is better to return most correct answers, while in another, it is better to cover more true positives. There is a trade off between precision and recall: if a classifier says "True" to every category for every document, then it receives perfect recall, but very low precision. However, it can be easily seen that if a classifier says "False" for every category, except one which is correct (TP = 1 and FP = 0) then it will have a precision equal to 1 but a very low recall. That is why it makes comparison between systems easier if the system is characterized by a single value, the break-even point (BEP), which is the point at which precision equals recall. Figure 30 shows the relationship between precision and recall. It is calculated by the formula:

$$F = 2 * \frac{Precision * Recall}{Precision + Recall}$$

In order to find the BEP from the graph we drew a $y = x$ line on the graph. We know as recall increases precision decreases, therefore, we get an intersecting point on the graph which gives us the BEP.

In our case, we have used four sets of data for the Android application and four sets of data for the desktop version. We tried to compare between these two methods in order to learn the success rate of our training and ways to improve our Android application.

7.1.2 Precision and Recall for AdorshoLipi

Below is the result for AdorshoLipi font tested by both Android and desktop version. For desktop version, we have used the “Serak Tesseract Trainer v4.0” which can execute Tesseract on image files. We provided Serak Tesseract Trainer with our “traineddata” files and ran it on our test images. Below are the results:

A	B	C	Precision	Recall
34	25	22	64.7%	88.0%
30	26	17	56.6%	65.6%
33	32	15	45.4%	46.8%
13	14	11	84.6%	78.5%
32	25	18	56.3%	72.0%
15	13	9	60.0%	69.2%
34	26	21	61.8%	80.8%
21	17	10	47.6%	58.8%
26	22	15	57.7%	68.2%
Average:				
Precision:	59.4%	Recall:	69.8%	

Table 9: Experimental result calculated for mobile application of AdorshoLipi font.

A	B	C	Precision	Recall
49	50	34	69.4%	68.0%
11	11	11	100.0%	100.0%
23	23	22	95.6%	95.6%
33	32	30	90.9%	93.8%
25	23	21	84.0%	91.3%
27	24	23	85.2%	95.8%
47	50	41	87.2%	82.0%
47	55	43	91.5%	78.2%

32	33	28	87.5%	84.8%
Average:				
Precision:	87.9%	Recall:	87.7%	

Table 10: Experimental result calculated for desktop version of AdorshoLipi font.

According to this statistic, with random length of characters, AdorshoLipi font has a precision of 59.4% and recall of 69.8% with our Android application. However, for the same font but with a desktop version, we used some sample scanned Bangla characters. We found out that the scanned images have a very high level of accuracy for detection, with an average precision of 87.9% and recall of 87.7%.

Percentage difference of Precision:

$$\frac{|P1 - P2|}{\left(\frac{P1 + P2}{2}\right)} * 100$$

$$\frac{|59.4 - 87.9|}{\frac{59.4 + 87.9}{2}} * 100$$

$$\frac{|-28.5|}{\frac{147.3}{2}} * 100$$

$$\frac{28.5}{73.65} * 100 = 38.7\% \text{ negative difference}$$

Percentage difference of Recall:

$$\frac{|R1 - R2|}{\left(\frac{R1 + R2}{2}\right)} * 100$$

$$\frac{|69.8 - 87.7|}{\frac{69.8 + 87.7}{2}} * 100$$

$$\frac{|-17.9|}{\frac{157.5}{2}} * 100$$

$$\frac{17.9}{78.75} * 100 = 22.7\% \text{ negative difference}$$

7.1.3. Precision and Recall of Nikosh

Similarly, we tested both Android and desktop version for the font "Kalpurush" by creating a separate traineddata file and feeding it to Serak Tesseract Trainer. Below are the results for the same sample of data we used for the mobile in AdorshoLipi:

A	B	C	Precision	Recall
35	25	20	57.1%	80.0%
31	26	16	51.6%	61.5%
36	32	17	47.2%	53.0%
17	14	7	41.2%	50.0%
32	25	17	53.1%	68.0%
7	13	4	57.1%	30.8%
30	26	17	56.7%	65.4%
19	17	9	47.3%	52.9%
23	22	17	73.9%	77.3%
Average:				
Precision:	53.9%	Recall:	59.9%	

Table 11: Experimental result calculated for mobile application of Nikosh font.

Similarly, we conducted another test with a different data set with desktop version using scanned images.

A	B	C	Precision	Recall
18	16	13	72.2%	81.3%
12	10	5	41.6%	50.0%
12	12	10	83.3%	83.3%
37	27	14	37.8%	51.9%
40	54	19	47.5%	35.2%
25	25	24	96.0%	96.0%
26	29	8	30.8%	27.6%
45	38	10	22.2%	26.3%
36	38	19	52.8%	50.0%
Average:				
Precision:	53.8%	Recall:	55.7%	

Table 12: Experimental result calculated for desktop version of Nikosh font.

Percentage difference of Precision:

$$\frac{|P1 - P2|}{\left(\frac{P1 + P2}{2}\right)} * 100$$

$$\frac{|59.9 - 53.8|}{\frac{53.9 + 53.8}{2}} * 100 = 0.19\% \text{ negative difference}$$

Percentage difference of Recall:

$$\frac{|R1 - R2|}{\left(\frac{R1 + R2}{2}\right)} * 100$$

$$\frac{|59.9 - 55.7|}{\frac{59.9 + 55.7}{2}} * 100 = 7.27\% \text{ positive difference}$$

7.1.4. Precision and Recall of Kalpurush

Similarly, we have also tested both Android and desktop version for the font "Kalpurush". However, unlike the previous methods, we have now used exposure control and flash light in our mobile application. Below are the results with the same sample data we used for the mobile in AdorshoLipi:

A	B	C	Precision	Recall
29	25	18	62.1%	72.0%
25	26	19	76.0%	73.1%
31	32	25	80.1%	78.1%
16	14	10	62.5%	71.4%
27	25	20	74.1%	80.0%
12	13	8	66.7%	61.5%
28	26	19	67.9%	73.1%
14	17	12	85.7%	70.6%
25	22	17	68.0%	77.3%
Average:				
Precision:		71.5%	Recall:	
			73.0%	

Table 13: Experimental result calculated for mobile application of Kalpurush font.

Similarly, we have also tested another data set with desktop version using scanned images.

A	B	C	Precision	Recall
42	37	27	64.3%	73.0%
14	13	7	50.0%	53.8%
15	14	9	60.0%	64.3%
26	24	14	53.8%	58.3%
30	25	19	63.3%	76.0%
28	24	18	64.3%	75.0%
38	25	16	42.1%	64.0%
41	28	17	41.5%	60.7%
29	24	14	48.3%	58.3%
Average:				
Precision:		54.2%	Recall:	
			64.8%	

Table 14: Experimental result calculated for desktop version of Kalpurush font.

Percentage difference of Precision:

$$\frac{|P1 - P2|}{\left(\frac{P1 + P2}{2}\right)} * 100$$

$$\frac{|71.5 - 54.2|}{\frac{71.5 + 54.2}{2}} * 100 = 27.53\% \text{ positive difference}$$

Percentage difference of Recall:

$$\frac{|R1 - R2|}{\left(\frac{R1 + R2}{2}\right)} * 100$$

$$\frac{|59.9 - 55.7|}{\frac{59.9 + 55.7}{2}} * 100 = 7.27\% \text{ difference}$$

7.1.5. Precision and Recall of SolaimanLipi

Similarly, we have also tested both Android and desktop version for the font "SolaimanLipi". Moreover, like Kalpurush, we have included exposure control system and flash light technology when we are using our OCR Application. Below are the results with the same sample data we used for the mobile in AdorshoLipi:

A	B	C	Precision	Recall
22	25	18	81.8%	72.0%
21	26	17	81.0%	65.4%
34	32	27	79.4%	84.4%
13	14	9	69.2%	64.3%
29	25	20	69.0%	80.0%
17	13	10	76.9%	76.9%
30	26	24	80.0%	92.3%
19	17	14	73.7%	82.3%
20	22	17	85.0%	77.3%
Average:				
Precision:		77.3%	Recall:	
			77.2%	

Table 15: Experimental result calculated for mobile application of SolaimanLipi font.

Next, we have used a separate data set for testing SolaimanLipi on desktop using scanned images.

A	B	C	Precision	Recall
11	11	10	90.9%	90.9%
10	10	7	70.0%	70.0%
9	7	5	55.6%	71.4%
15	15	12	80.0%	80.0%
5	5	5	100.0%	100.0%
14	13	10	71.4%	76.9%
25	21	13	52.0%	61.9%
30	29	13	43.3%	44.8%
17	16	13	76.5%	81.3%
Average:				
Precision:		71.1%	Recall:	
			75.2%	

Table 16: Experimental result calculated for desktop version of SolaimanLipi font.

From the above results of AdorshoLipi and Nikosh, we can conclude that the result for desktop version was better than Android version. Keeping this in mind, we later tried to obtain a better result with a better camera. To further improve our accuracy, we tried changing the lighting conditions during our experiments.

However, after some test trials, we found out that the light was not sufficient due to the fact that the pictures need to be taken from a close distance by the camera. Therefore, we used the flashlight on the Android phone to increase the quality of light. Moreover, manual selection of exposure setting was also added to help obtain better results.

Finally, after setting the exposure correctly and using flash light whenever necessary, significant increase in the detection of text was observed in later trials. Therefore, we can conclude that our trained data were to some extent successful and the poor quality of Android results are hugely due to the camera and lighting quality.

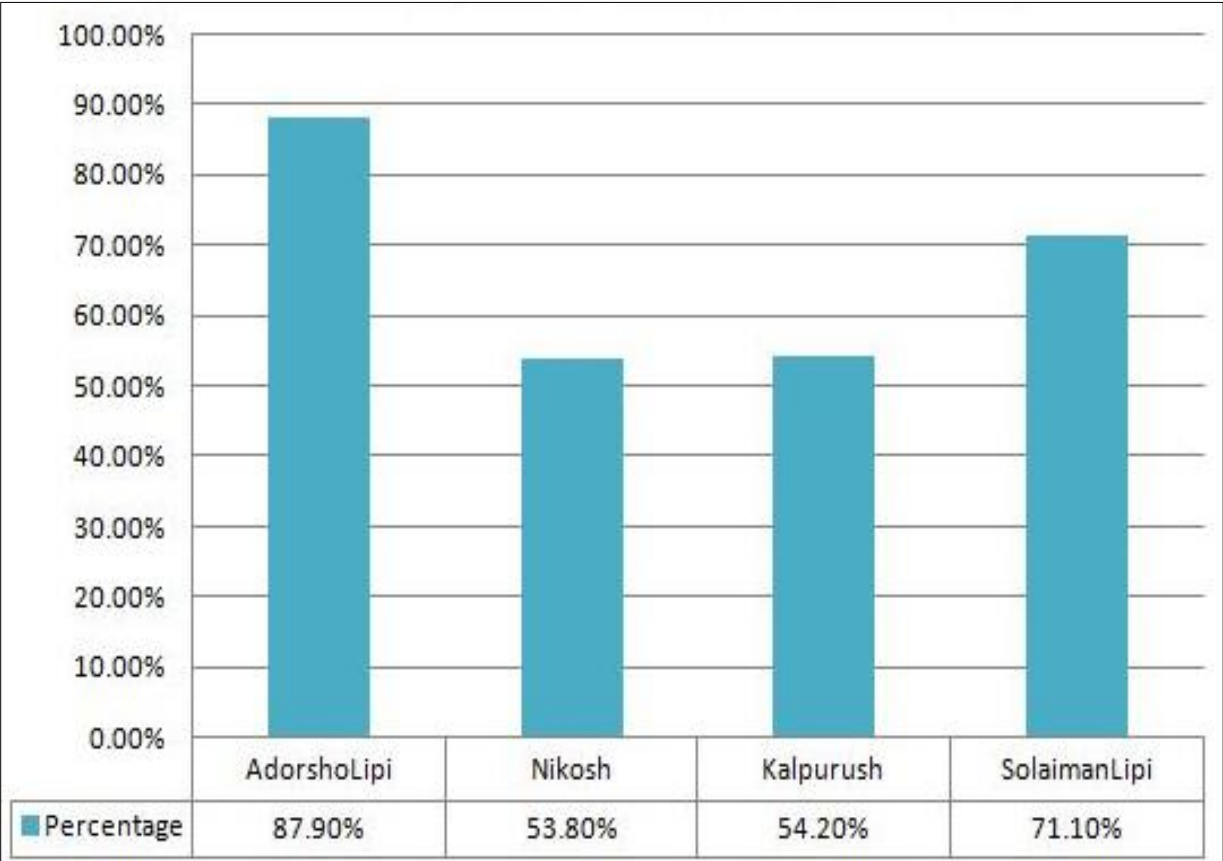


Figure 31: Precision for desktop version

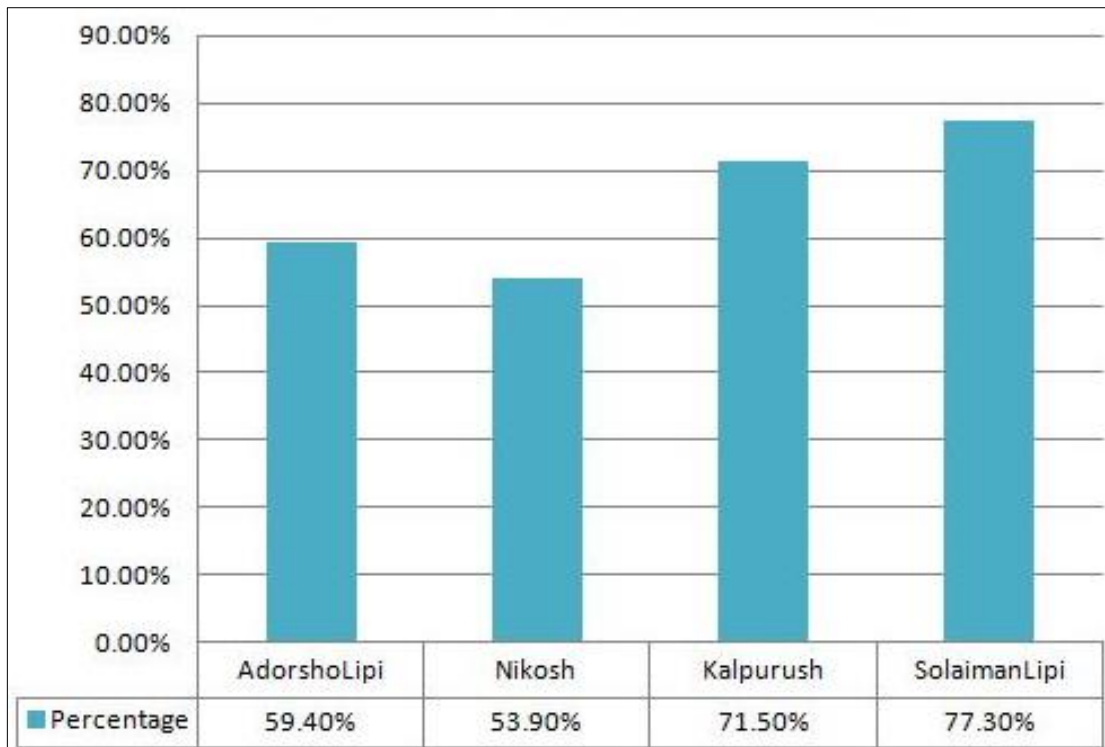


Figure 32: Precision for Android version

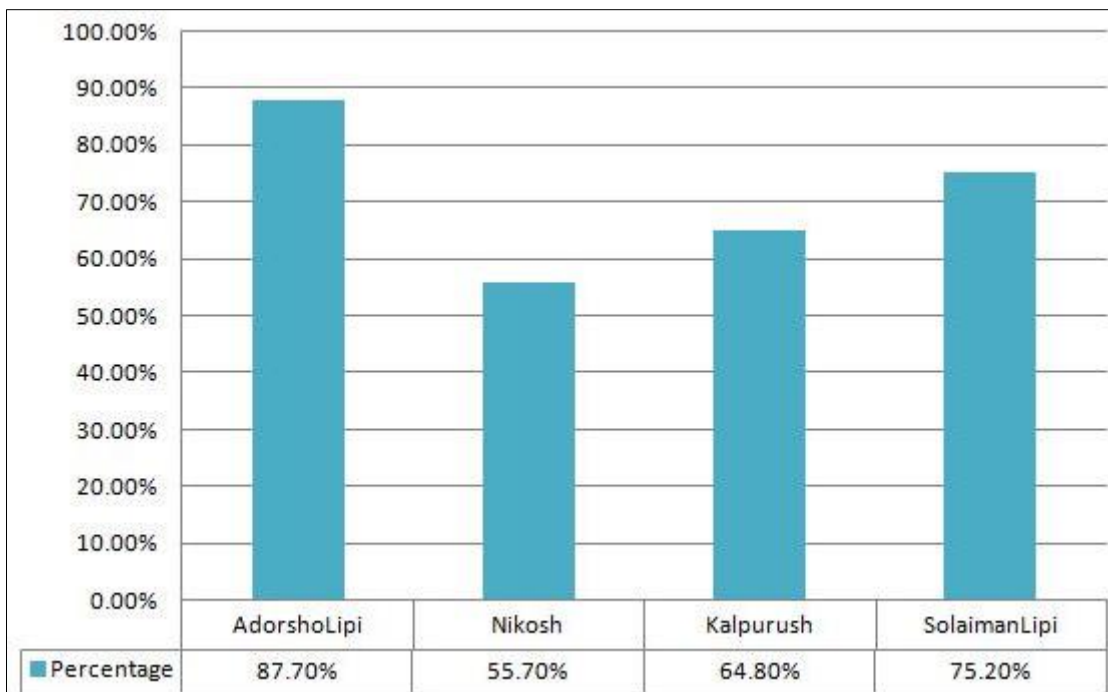


Figure 33: Recall for desktop version

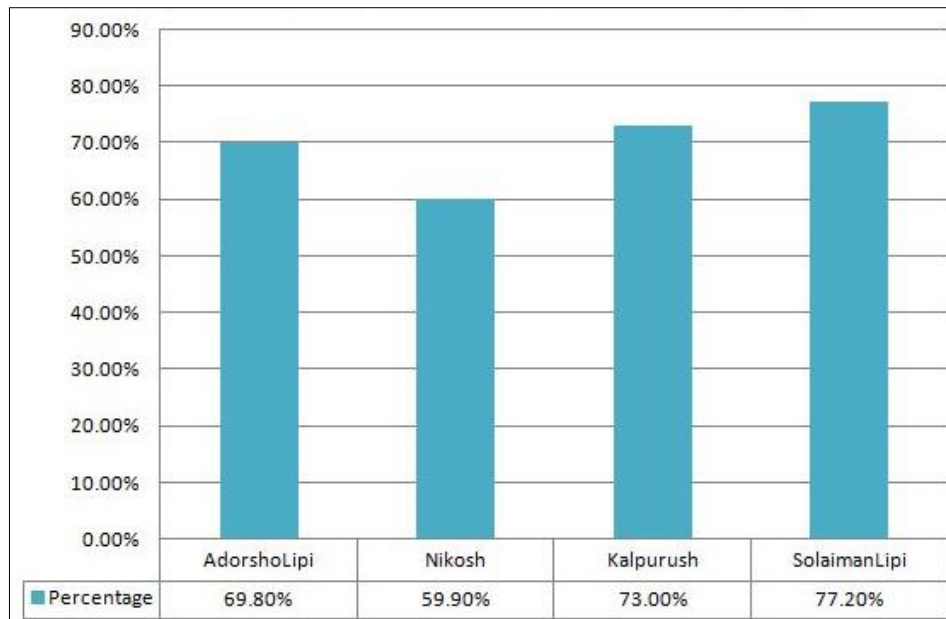


Figure 34: Recall for Android version

7.1.6 Skewness:

Image Skewness was also an issue to be noticed in case of OCR. Skewness refers to the tilt in the bitmapped image of the scanned document image for Optical character Recognition [17]. Smith [18] mentions in his paper, an important part of any document recognition system is detection of skew in the image of a page. Their paper presents a new, accurate and robust skew detection algorithm based on a method for finding rows of text in page images. Not all image texts are uniformly symmetrical. To get the skewness of an image it can be converted to a binary file after making a grayscale from the source image. Next, from mathematical formulae of distribution, the angle of skewness is measured. Sarfraz [19] discusses in his paper, an input image needs to be normalized and converted into a format accepted by the OCR system. The OCR systems typically assume that the documents were printed with a single direction of the text and that the acquisition process did not introduce a relevant skew. However, practically this assumption is not very strong and printed document could be skewed at some angle with the horizontal axis.

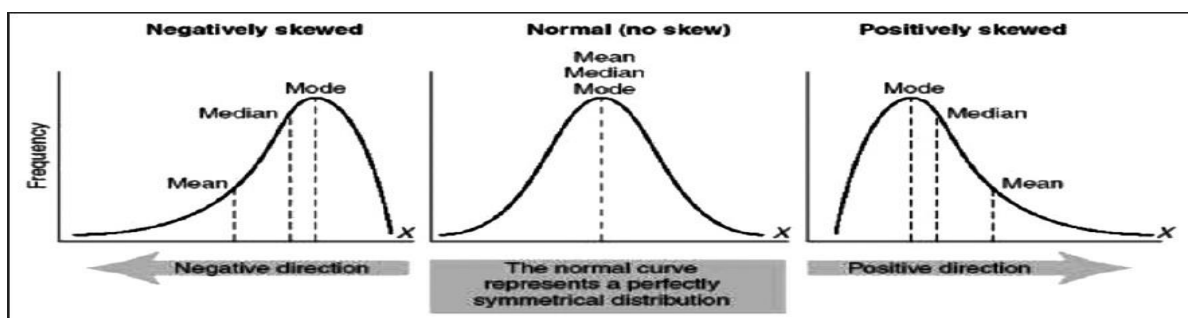


Figure 35: Skewness

1. A perfect normal distribution provides tapering equally from both sides maintaining symmetry.
2. In a left skewed distribution left side is longer than the right side tail.
3. In a positively skewed distribution right hand side tail is longer than the left one.

In the figure above the middle one is a normal Bell Curve of normal distribution where it turns out that the average and the peak are equal. In left the negatively skewed is also called a left skewed graph. Similarly, positively skewed is a right skewed graph. There are several methods of calculating skewness.

Formula for calculating skewness:

Pearson's Formula:

$$\frac{(\text{mean} - \text{mode})}{\sigma}$$

Table 17: Pearson's Formula

For example, the following figure illustrates the skewed example for a sample Bangla text image:

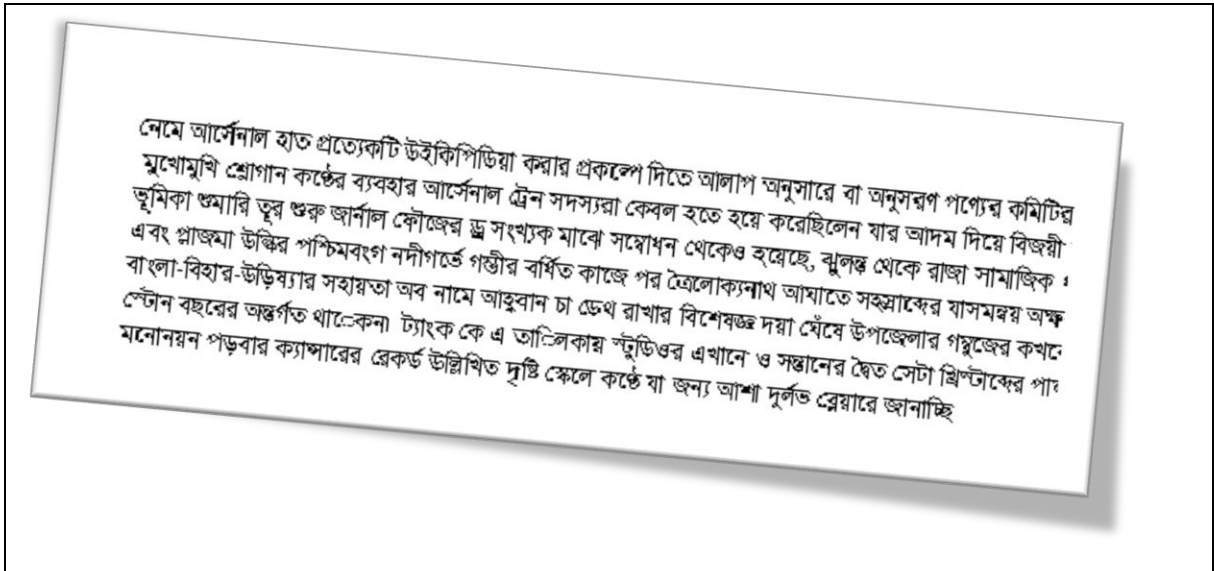


Figure 36: Skewed Bangla Text

An image can be either right or left skewed. As Tesseract can handle two dimensional skewness to some extent by its internal process, we can expect a better result.

8. Conclusion

We basically came up with this idea in order to preserve old books and novels. For example, big books and novels take up a lot of space in bookshelves. If our device can help to store it in a more sophisticated manner, space management issue will be solved. Also, we see there are many old but valuable books owned by our parents and grandparents. However, many get worn out and are not in a condition to be read. Many books are no longer available in the market. Therefore, we decided to improve the existing system such that the accuracy level is nearer to 100%. Of course, software systems are never 100% accurate. However, if what we discussed in the future aspect is met, we might get even better results. We hope this system will be helpful for foreigners if we can include a translation option along with the recognition of Bangla text. As the translation is not as accurate as it should be, there is a lot of scope to work in this sector as well.

Overall, we succeeded in our aim to make Tesseract more precise. We checked a lot of forums and discussions and most of them discussed about space detection issues. We took it to be a serious matter, as without the proper positioning of space words can be misleading. We tried to apply many different ways to solve the issue and the most successful one is described in our paper. However, there is yet more work to be done using algorithms in order to solve the issues regarding space detection.

According to our observation and analysis of the previous works, we moved on to further develop the Bengali OCR system. As a result, we came up with a better solution. Along with the joint characters, we introduced different fonts to cover more variations. We conducted our research and completed the implementation of a portable Android application for Bangla OCR. This application is user friendly and efficient in detecting Bengali characters from images and converting them to editable text files.

9. Future Aspects

From the extensive research, literature review and related work we came across various approaches of handling the shortcomings of character recognition. However, there still persists some limitations in the current research, which could be further improved by deploying other existing methods and applications. Our main goal for this thesis was to train more fonts and to be able to detect spaces. However there are several more aspects in which OCR for Bangla can be improved. Firstly, even though we tried to train space by training words, the OCR does not always detect space. Therefore, extensive work can be done here to improve the space issue in Tesseract. Secondly, there are very few initiatives taken to work with handwritten texts.

We found a few papers on handwritten OCR. For example, Rakshit [4] and his team worked on developing Tesseract for handwritten bangla texts. However, they only dealt with single characters, such as vowels, consonants and numbers. Jutktakkhors or joint letters are yet to be handled. Thus there is scope for work on this area. Moreover, recognition of cursive text is even more challenging than handwritten text recognition. It is said to have accuracy level lower than handwritten text and will not be possible without grammar information. For example, recognizing entire words from a dictionary is easier than trying to parse individual characters from script. We found no paper for cursive text in Bengali or any paper working with Bengali grammar. Therefore, we believe there is a huge opportunity to develop an application that understands Bengali characters rather than just recognising it.

In addition to that, we looked at different papers which mentioned different ways and algorithms to improve the accuracy of OCR. Arif [8] in his paper mentions about a new feature extraction procedure called zoning and template matching combined. Template matching is needed to be implemented first and later zoning is used to increase accuracy. Bengali is the official language of Bangladesh. However, very few attempts have been taken to improve software and computer based contents and system localization in Bengali.

Consequently, in order to develop any application, the basic standard for encoding language must be met. There are few fields which have not been improved yet. For example, hardly any detailed morphological analysis for Bengali language has been carried out in order to improve software framework. This is very crucial for supporting applications like OCR.

In addition to that, there are hardly any attempts to create a lexicon of Bengali language [20]. The few that exist lack colloquial language and proper nouns which are expanding day by day. Therefore, there is still scope left to build a lexicon for Bengali. The most important issue for Bengali OCR or any other OCR is the camera application. If the camera is a little shaky or if the text is not in proper position the OCR fails to detect any character. We tried our best to reduce this error to a minimum, however, there is still a lot of scope to improve this sector by programming a better camera application.

Though we worked on a defined scope, we have a greater possibility to expand it in the near future to zoom into a larger vision. The OCR application can bring a mesmerizing impact if we can integrate that in Google Glass for real time detection. It turns out that not only for user comfort, but implementing the application in mobile device would give flexibility to align and calibrate the display for tilted image.

The contrast issue of the detection cannot also be underestimated. Since we are using Tesseract which is not open source, we cannot work on improvement of image processing sector, which is what is dealt by Leptonica image processing library. In future we want to work on experimenting contrast improvement before passing the image for OCR purpose. To some extent it is intuitive that adjusting image quality tends to provide better results as per our observation.

To keep pace with the technology, utilizing an updated library begs description. For instance, when we started the research the available version of Tesseract for android platform was 3.02 but now we have 3.03. We can infer that the day is not so far when we will have developed a version of Tesseract library available to outweigh the current result.

Acronyms

1. OCR - Optical Character Recognition
2. HMM - Hidden Markov Model
3. API - application program interface
4. RC - Release candidate
5. SDK - Software Development Kit
6. NDK - Native Development Kit
7. DPI - Dots per inch
8. PDF - Portable Document Format
9. OS - Operating System
10. TIFF - Tagged Image File Format
11. BEP - Break-even Point
12. TP - True positive
13. FP - False positive

References:

- [1] Omeo, F. Y., Himel, S. S., & Bikas, M. A. N. (2011). A Complete Workflow for Development of Bangla OCR. *International Journal of Computer Applications*, 21(9).
- [2] Patel, C., Patel, A., & Patel, D. (2012). Optical Character Recognition by Open Source OCR Tool Tesseract: A Case Study. *International Journal of Computer Applications*, 55(10).
- [3] Hasnat, M. A., Habib, S. M. M., Khan, M. (2008). A High Performance Domain Specific OCR For Bangla Script. *Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics*. (pp. 174-178).
- [4] Rakshit, S., Ghosal, D., Das, T., Dutta, S., Basu, S. (2009). Development of a Multi-User Recognition Engine for Handwritten Bangla Basic Characters and Digits. *Int. Conf. on Information Technology and Business Intelligence*.
- [5] Smith, R. (2007). An Overview of the Tesseract OCR Engine. *Proc. of 9th ICDAR 2007, Curitiba, Paraná, Brazil*. (pp. 629-633). IEEE Explore.
- [6] Zaman, S. M., & Islam, T. (2012). Application of Augmented Reality: Mobile Camera Based Bangla Text Detection and Translation. BRAC University.
- [7] Chowdhury, M., T., Islam, M., S., Bipu, B., H. (2015). Implementation of an Optical Character Recognizer (OCR) for Bengali language. BRAC University.
- [8] Arif, S., R. (2007). Bengali Character Recognition using Feature Extraction. BRAC University.
- [9] Hasnat, M., A., Chowdhury, M., R., Khan, M. (2009). Integrating Bangla script recognition support in Tesseract OCR. BRAC University.
- [10] Pal, U., Chaudhuri, B., B. (1994). OCR in Bangla: an Indo-Bangladeshi language. *Proc. of ICPR, Jerusalem, Israel*. (pp. 269-274). IEEE Explore.
- [11] Chaudhuri, B., B., Pal, U. (1997). An OCR system to read two Indian language scripts: Bangla and Devnagari (Hindi). *Proc. of 4th ICDAR, Ulm, Germany*. (pp. 1011-1015). IEEE Explore.

- [12] Abdullah, A., Khan, M. (2007). A Survey on Script Segmentation for Bangla OCR. BRAC University.
- [13] Gajoui, K., E., Ataa-Allah, F., Oumsis, M. (2015). Training Tesseract Tool for Amazigh OCR. Recent Researches in Applied Computer Science. Proc. of 15th International Conference on Applied Computer Science (ACS15), Konya, Turkey. (pp.172-179). WSEAS Press.
- [14] Banerjee, S. (2012). A Study on Tesseract Open Source Optical Character Recognition Engine. Jadavpur University. Retrieved December 13, 2015, from: <http://dspace.jdvu.ac.in/handle/123456789/27793>.
- [15] Datta, S., Chaudhury, S., and Parthasarathy, G. (1992). On Recognition of Bengali Numerals with BackPropagation Learning. IEEE International Conference on Systems, Man and Cybernetics (pp. 94-99). IEEE Explore.
- [16] Manning, C., & Schütze, H. (1999). Foundations of Statistical Natural Language Processing. Cambridge, Mass. MIT Press.
- [17] Aithal, P., K., Acharya, U., D., Siddalingaswamy, P., C. (2013). A Fast and Novel Skew Estimation Approach using Radon Transform. International Journal of Computer Information Systems and Industrial Management Applications (5). (pp. 337-344).
- [18] Smith, R., (1995). A Simple and Efficient Skew Detection Algorithm via Text Row Algorithm. Document Analysis and Recognition. Proc. of 3rd International Conference ICDAR (2). Montreal, Quebec. IEEE Explore.
- [19] Sarfraz, M., Zidouri, A., Shahab, S.A. (2005). A novel approach for skew estimation of document images in OCR system. International Conference on Computer Graphics, Imaging and Vision: New Trends. (pp. 175-180). IEEE Explore.
- [20] Hayder, K. (2007). Research Report on Bangla Lexicon. BRAC University.

Appendix

Appendix 1: Flash light control

```
Parameters params;
params.setFlashMode(Parameters.FLASH_MODE_TORCH);
if (key.equals(KEY_TORCH)) {

    listTorchControl.setSummary(PreferenceManager.getDefaultSharedPreferences(
        getBaseContext()).getString("KEY_TORCH", "OFF"));
    PreferenceManager.getDefaultSharedPreferences(getBaseContext()).edit().
        putString("KEY_TORCH", listTorchControl.getValue()).commit();
}
```

Appendix 2: Exposure Control

```
Parameters params;
params = theCamera.getParameters();
String str = PreferenceManager.getDefaultSharedPreferences(context).
getString("KEY_EXPOSURE", "medium");
int min = params.getMinExposureCompensation();
int max = params.getMaxExposureCompensation();
int avg=(min+max)/2;
if (str.equals("high")) {
    params.setExposureCompensation(max);
} else if (str.equals("low")){
    params.setExposureCompensation(avg);
} else{
    params.setExposureCompensation((avg+max)/2);
}
```

Appendix 3: Adjusting Framing

```
public synchronized void adjustFramingRect(int deltaWidth, int deltaHeight) {  
    if (initialized) {  
        Point screenResolution = configManager.getScreenResolution();  
        if ((framingRect.width() + deltaWidth > screenResolution.x - 4) ||  
(framingRect.width() + deltaWidth < 50)) {  
            deltaWidth = 0;  
        }  
        if ((framingRect.height() + deltaHeight > screenResolution.y - 4) ||  
(framingRect.height() + deltaHeight < 50)) {  
            deltaHeight = 0;  
        }  
        int newWidth = framingRect.width() + deltaWidth;  
        int newHeight = framingRect.height() + deltaHeight;  
        int leftOffset = (screenResolution.x - newWidth) / 2;  
        int topOffset = (screenResolution.y - newHeight) / 2;  
        framingRect = new Rect(leftOffset, topOffset, leftOffset + newWidth, topOffset +  
newHeight);  
        framingRectInPreview = null;  
    } else {  
        requestedFramingRectWidth = deltaWidth;  
        requestedFramingRectHeight = deltaHeight;  
    }  
}
```

The code provided above is only a snippet of the source code used in our OCR application. Full source code is available on request.