Hand-written Character Recognition Agent

Amir Amine, John Habib and Victoria Harb

Abstract – This report discusses the implementation of an intelligent agent that recognizes hand-written characters and classifies them accordingly based on their label (Capital letter, lower-case letter, or number), and finally based on what letter or number that character is.

– – – – – – – – – – ◆ – – – – – – – – – –

Contents

Figures

No table of figures entries found.

Tables

No table of figures entries found.

1 Introduction

This report walks through the implementation of a hand-written character recognition agent. The agent follows the following steps: Pre-processing, feature extraction, classification (training and evaluation), and prediction. The details of these steps are well documented throughout this report.

2 Background

Handwritten character recognition is gaining an increasing importance because of the demand for creating a paperless world and digitization and for automatically processing large volumes of data Handwritten character recognition is challenging due to its divergence in writing styles (due to differences in age, gender, and education) and visual similarity between characters. In addition, handwriting recognition is more complex than printed handwriting due to its extreme variability, variability of shapes, spacing between words and characters, line fluctuations. This recognition system recognizes handwritten characters on filled dataset forms, and other historical reports, such as digitization of historical manuscript, bank data processing, signature verification for banking or forensic, zip code recognition, etc.

Handwritten character recognition systems can be divided into offline and online. Offline is processing scanned images of handwritten characters, while online entails the processing of characters in real time while writing is taking place. In the implementation described in this project, we are only tackling an offline character recognition system.

3 Proposal

3.1 Overview

This project presents a hand-written character recognition agent, written in python with the aid of different external libraries (Scikit-learn [1] and OpenCV[2]). The agent gets as an input an image. That character is then processed, features are extracted from the image and the image is classified using SVM classifier which is trained offline on a set of data.

3.2 Pre-processing

Every image input needs to be pre-processed to make classification more accurate. Pre-processing consists of various consecutive steps to convert the image of the character into the most recognizable format. The steps of pre-processing are: Importing the image, converting it to grayscale, thresholding the image (binary image), removing noise, getting the bounded box, resizing to 64x64 to normalize the image, and finally skeletonizing the image, using Scikit-Image and OpenCV external libraries.

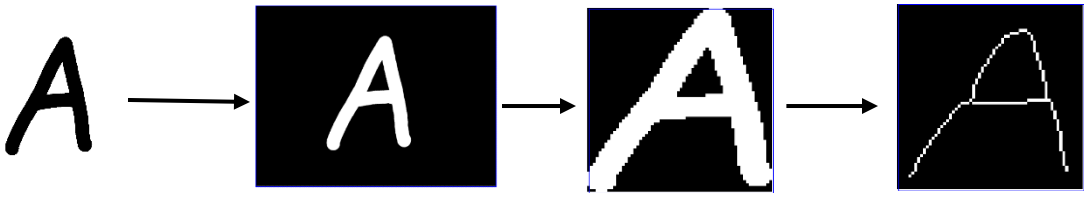

Figure 1-a: Imported Image

Figure 1-b: Binary Image

Figure 1-c: Normalized Image (64x64)

Figure 1-d: Skeletonized Image

OpenCV was used for importing, gray-scaling, thresholding and cropping to bounded rectangle of the image

Scikit-Image was used for image normalization (resizing to 64x64) and skeletonizing the image.

After experimentations, we found out that resizing the image to 500x500 pixels, after cropping to bounded box, and downscaling to 64x64 pixels produces a better result than directly resizing the image to 64x64, which will distort the image and remove pixel values.
The image pre-processing is done in one python definition (**image_processing()**) which returns the skeletonized image, which is an **array** of 0s and 1s with 0s representing the black values (background) and 1s representing the white values (character).

3.3 Feature extraction

We pre-processed the image to a skeletonized image to easily extract features. In this project, two methods of feature extraction are utilized: Zoning and Histogram of Gradients.

3.3.1 Zoning

The image is divided into 4x4 zones (16 equal zones), each zone containing 256 values of 0s and 1s (pixels). In each zone, we calculate the number of white pixels (value = 1), knowing that the white pixels represent the character, and the black pixels represent the background. The number of white pixels of each zones are appended into one feature vector (zone_ fv). Each image has its own zone feature vector of 16 features each.

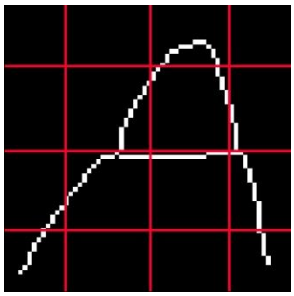

Figure 2-a: 16 equal zones

Figure 2-b: Zone 5

Figure2-c: Zone 10

```
[0, 0, 18, 0, 0, 13, 4, 15, 6, 18, 16, 20, 12, 0, 0, 10]
```

Figure 2-d: Zone Feature Vector

3.3.2 Histogram of gradients

Histogram of Gradients describe the structure and texture of the image by computing local gradients in x direction and y direction. The dimensionality of the feature vector is affected by the pixels_per_cells and pixels_per_block parameters. In our project, we chose the pixels_per_cells to be (8,8) which will divide the image into (64/8)x(64/8) cells = 8x8 cells = 64 cells. We chose the pixels_per_block to be (2,2) which will group every 2x2 cells (4 cells) together in one block. Gradients are computed in each cell, normalized with respect to the block the cell belongs to, and concatenated with the Histogram of Gradients feature vector (hog_fv).

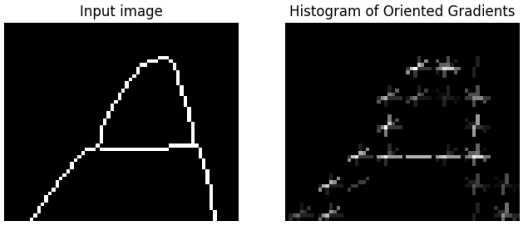

Figure 3-a: Histogram of Gradients of Letter 'A'

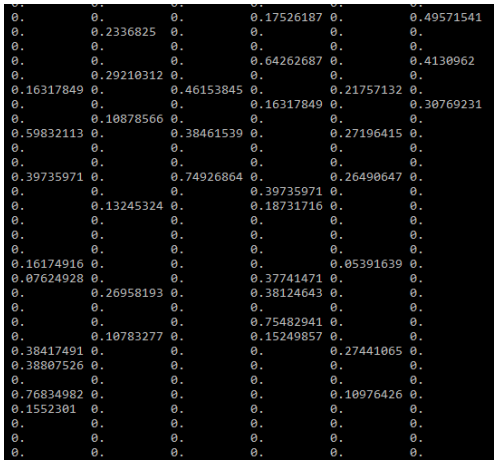

Figure 3-b: Section of the Histogram of Gradients feature vector of the same Letter 'A' in figure 3-a

3.3.3 Additional Features

We extracted additional features from the image such as **Projection Histogram, Mean Projection Histogram, Profile Intensity (along a line), and Edges.** After experimentations, we found out that the best features which produced accurate results were Zoning and Histogram of Gradients. Moreover, adding more features to the training data resulted in an overfitting model. Therefore, reducing the number of features to only two solved the overfitting challenge.

3.4 Classification

The classifier we used is a multi-class SVM classifier from the Scikit-Learn library. This classifier is an ensemble of two-classes SVM classifiers built on the equation $\frac{n_{classes} \times (n_{classes}-1)}{2}$ with each classifier training data from two classes. After experimentations, we decided that SVM would be our best non-parametric learner. Compared with KNN, Decision Trees, and Bagged Classifier of SVM, KNN, and Decision Trees, SVM produced the highest accuracy when trained with the data of the extracted feature vectors.

3.4.1 Training

3.4.1.1 Data

The training data is obtained from Kaggle [3]. It consists of 3410 images divided into 64 classes (0-9), (A-Z) and (a-z). We renamed the training data **training_dataset_1** for the sake of convenience of our program.

3.4.1.2 Validation

Before splitting the data into training data and testing data, we shuffled the data so that we make sure the model passes through all the data each time we re-train the model. The data was split into 80% training data and 20% using cross-validation algorithm. The model produced a 79% accuracy with 538 correct labels out of 682.

3.4.1.3 Evaluation

The whole **training_dataset_1** was tested on the model and produced an accuracy of 84%. We assume that the evaluation accuracy is higher than the validation accuracy because the model was not fitted to a percentage of the training data since the algorithm that splits the data is cross validation.

3.4.2 Prediction

We prepared a set of inputs written by ourselves. These inputs were pre-processed to produce their own feature vectors and were given to the model to predict. With 23 inputs, the model predicted 10 correct labels which yields an accuracy of 56%.

4 GUI interface

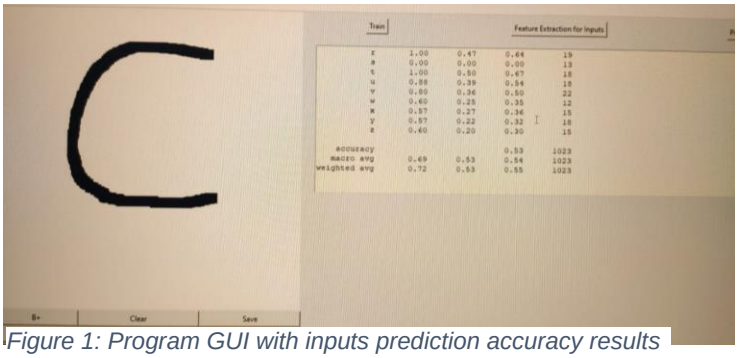

Figure 1: Program GUI with inputs prediction accuracy results

For the GUI, we used the Tkinter library in Python along with the Pillow external library that adds support for opening, manipulating, and saving many different image file formats. In the left frame, we've created a canvas which is used for drawing pictures. Additionally, we've added a button frame at the bottom left corner which has the following buttons: B+ (increases the brush width), B- (decreases the brush width), clear (clears the drawing), Change Color (changes color). Moreover, we added a "Label" dropdown menu that allows the user of choosing the correct label for their drawn character. We also created a "Save" button which

saves the drawn image in the “Inputs/Img” folder and automatically adds the image to the “inputs.csv” spreadsheet using the csv module in python. In the right frame, we used 3 buttons which are the core functionality of our GUI. The first button “Train” runs the evaluation over the training data, the second button “Feature extraction for Inputs” runs the feature extraction for the inputs, and the third button “Predict” runs the prediction over the inputs. Finally, we integrated a text box which outputs the result of the script that is currently running, similar

to the output console in the IDE.

5 Limitations

Similarities:

There are letters and numbers that could be written similarly that even humans cannot differentiate between them, unless they were a part of a context (e.g., a sentence). Therefore, this limitation produces inaccuracies in the model.

I	Capitalized letter i	1	Number 1
o	Letter o	0	Number 0
s	Letter s	5	Number 5
g	Letter g	9	Number 9

Cursive:

There exist many ways people can write letters using cursive handwriting, which might not be accounted for and could possibly confuse the program. Our image processing does not account for slant correction.

SVM:

Since we are using SVM as a classifier, we can’t tackle large datasets because the training complexity of SVM is very high.

Text from pictures:

The training data consists of images with a white background and not taken using a camera picture of a piece of paper. Therefore, certain types of noise might not be correctly removed for e.g.: shadows, dark background color, etc.). This might produce a case of overfitting.

Language:

The program is specific for Latin numerals and Arabic number. Therefore, it is incapable of determining different type of letters and numbers from other languages.

6 Possible improvements

In the training process we used cross validation. Therefore, to increase our accuracy we had to shuffle the data every time we re-train the model to achieve higher accuracy. Therefore, it would have been more efficient to utilize the K-fold cross validation, which improves the accuracy organically by training it only once, for a desired K value. Moreover, In our design, we only had the model predict the labels on a set of inputs and not on each input individually. Our implementation can be further improved by outputting the prediction of each input image individually Additionally, Adding more training data is needed to avoid overfitting and improve accuracy. Finally, hyper-tuning the model is an improvement to consider to further improve accuracy.

6 References

[1] https://scikit-learn.org/

[2] https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

[3] https://www.kaggle.com/datasets/dhruvildave/english-handwritten-characters-dataset