

Image2Code

Ibrahim Ali, Omar Wael, Ahmed Hamed, Abdel Rahman Hashim
*Cairo university

Abstract—In technical interviews, interviewees are sometimes asked to write code on paper. The interviewer can see the general logic of the code but cannot actually run it and prove its correctness. In our exams, all the code is answered on paper and the instructors need a way to test the written code. Our project addresses these problems. First, we apply preprocessing to an image (either captured by a camera or a screen-shot), Second, we apply segmentation on the line, word and character levels, Then we classify the segmented characters using a MLP classifier.

I. INTRODUCTION

Tasks automation has been an ultimate goal for a while. We are willing to automate many tasks that the machine can do. One of the great fields that really needs automation (as it wastes too much time and energy) is the handwriting recognition field. In which field the humans only tries to observe the written text to extract some piece of information. Those humans were replaced in some banks and postal offices by AI systems that can detect the given digits (e.g. bank account numbers, post address, ...etc) with an accuracy higher than the average persons. This field is still growing and developing, and were trying to contribute to this enhancement. Here you basically try to detect the handwritten code, which if done with high accuracy can automate many tasks that takes more time. And its also a portal to higher levels of handwriting recognition of complex characters. The current available methods depend on datasets which contain pen strokes during writing, which achieves pretty high accuracy. However, here we try to avoid this, as in too many cases you dont have and cannot acquire these strokes (especially in the old scanned files), therefore, were using pure image processing techniques, AI models and post processing algorithms to fix some mis-classified characters. In this paper, well walk you through the stages in which the scanned handwritten code passes, and well demonstrate each stage and the techniques used to achieve the overall goal. Well also provide links to resources and the used datasets in this project. Weve developed some simple GUI application to assist to try this research as a product and observe the input-output relationship. Finally, there are further improvements that can be made to increase the system stability, accuracy and precision.

II. PREPROCESSING STAGE

A. Noise reduction and binarization

The gray-scale source image may contain noisy areas that should be eliminated and it may need contrast enhancement. In the first step, we try to extract the paper from the surrounding background by detecting the four corners of the paper using canny edge detection after blurring and getting the largest

contour in image that must have 4 points which represents the text paper. If the image is merely the text paper without any background, The algorithm simply applies the second step. In the second step, Wiener filter with window size of 3 is applied on the current preprocessed text paper and then Sauvola adaptive thresholding is applied using $k = 0.2$ [1].

B. Skewness correction

In this stage we try to calculate the skew angle of the tilted text inside the paper in order to align it with the horizontal axis as much as possible. After binarization, we try to fit the rectangle with the minimum area around the whole text, then after finding this rectangle, we calculate the angle between it and the vertical axis. By finding this angle we can easily multiply the whole image by a rotation matrix constructed using this angle (Fig. 1).

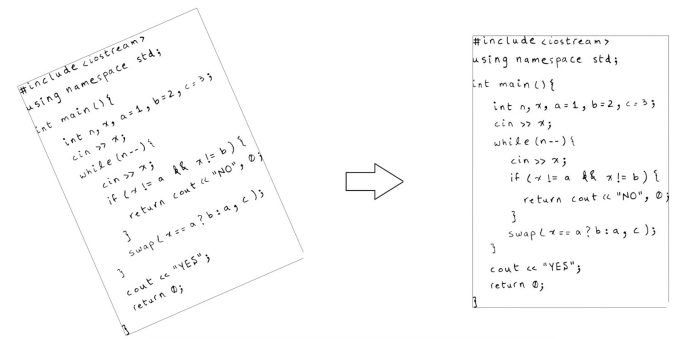


Fig. 1. Paper skew correction

This technique requires the text paragraph to be approximately in rectangular shape, otherwise the minimum bounding rectangle may be titled, which results in wrong calculated angle as shown below (Fig. 2).

So in order to get consistent results, we switched the skew angle calculations from the bounding box technique explained previously to Hough line transform technique. In Hough line transform technique we calculate the skew angle as the average angle of all promising lines results from Hough transform (Fig. 3).

III. LINE SEGMENTATION

We assume that the lines are well separated between each other which made the line segmentation easy for us.

We have tried to segment lines using a statistical approach [2] but the results were not accurate. The paper uses the following approach to segment lines:

- 1) OTSU binarization.

```

#include <iostream>
Using namespace std;

int main() {

    cout << "Hello Image processing course";
    return 0;
}

```

Fig. 2. Wrong paper skew correction

```

#include <iostream>
using namespace std;
bool isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= n; ++i)
        if (n % i == 0)
            return false;
    return true;
}
int main() {
    int x;
    cin >> x;
    cout << isPrime(x);
    return 0;
}

```

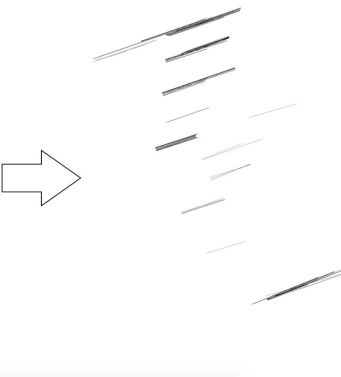


Fig. 3. Determining skew angle using probabilistic hough transform

- 2) Get initial candidate lines using y-axis histogram projection profile and using adaptive threshold between valleys.
- 3) Apply line drawing algorithm and make decisions about regions that hit the lines using Bivariate Gaussian Distribution.

The paper at the beginning apply binarization using OTSU thresholding. Then, the paper is divided into 20 vertical chunks and the y projection histogram profile is calculated for each chunk and we get the peaks for each chunk histogram. The valleys is the local minimum between 2 peaks. We get the initial line candidates by recursively connecting each valley in the nth with its nearest chunk in the n-1th chunk. We start from the 5th chunk till the 1st chunk. After we have computed the initial lines we repair them using the Bivariate Gaussian probabilities, when a line hit a component in the image we calculate the probability that this component belongs to the above line region or the below one then, a decision is made and we correct the line in turn. The equations used for calculating the probabilities are:

- Get the line region Bivariate distribution mean and covariance (Fig. 4, 5).
- The probability that a pixel in a component belongs to a line region (Fig. 6).
- The probability of a component is in a region (Fig. 7).

$$\vec{\mu}_{A(N+1)} = \frac{N-1}{N} \cdot \vec{\mu}_{A(N)} + \frac{1}{N} \cdot p_{N+1}$$

Fig. 4. Mean vector of a line region.

$$\Sigma_{A(N+1)} = \frac{N-2}{N-1} \cdot \Sigma_{A(N+1)} + \frac{1}{N-1} \cdot (p_{N+1} - \vec{\mu}_{A(N+1)})(p_{N+1} - \vec{\mu}_{A(N+1)})^T$$

Fig. 5. Covariance matrix of a line region.

The problem with this approach was the code has a different nature than the normal handwritten text; the special characters made it difficult for line segmentation to perform as proposed in the paper. The following three example show how it has succeeded and somehow failed to segment some lines (Fig. 8, 9, 10).

IV. WORD SEGMENTATION

In our first trials we did a naive algorithm to detect words. The basic idea of this algorithm is to search for white spaces that are greater than a certain constant threshold. In fact this technique results in good output for printed codes where the words are uniformly distributed with clear white spaces between them. But due the fact that we are willing to detect handwritten codes, not just printed ones, and that human handwriting most times vary in font size and spaces among different lines, so we began to think how to enhance this algorithm by calculate this threshold dynamically instead of giving it a constant value. The best output of our trials results from the following calculations:

- 1) Calculate the average characters width.
- 2) Calculate the average spaces width that are greater than the number calculated in step (1).
- 3) Calculate the threshold as the average of the two numbers calculated in the previous two steps.

These calculations are performed for every each line individually, so each line has its own threshold that is calculated dynamically as mentioned above. The good point of this dynamic approach is that it can handle fonts of different sizes and spaces without the need to change the threshold manually.

V. CHAR SEGMENTATION

In character segmentation stage, we receive a word image as the input in order to break it down into characters that forms the word. In order to do so, we run a basic flood-fill algorithm (with 8 neighbours) to give a unique label for every connected components (i.e. the characters). What was challenging in this stage is that some of the characters in our alphabet are composed of more than one connected components. Those characters, like i j % ; ! ? = , require us to do a further step after finding the connected components, that is the merging step. In the merging step we try to merge different components that most likely form single character from our alphabet. In order to do so we tried different techniques. One technique

$$P(p_i|\mu, \Sigma) = |2\pi\Sigma|^{\frac{1}{2}}(p_i - \mu)\Sigma^{-1}(p_i - \mu)^T$$

Fig. 6. probability that a pixel belongs to a line region

$$P(C|\vec{\mu}_A, \Sigma_A) = P(p_1|\vec{\mu}_A, \Sigma_A) \cdot P(p_2|\vec{\mu}_A, \Sigma_A, p_1) \dots P(p_T|\vec{\mu}_A, \Sigma_A, p_1, p_2, \dots, p_{T-1})$$

Fig. 7. probability that a component belongs to a line region

was to merge two neighbour components when one of them is above the other (i.e. when they are in different y-level). This technique was so good in merging most of the characters but it fails in detecting % because these characters have components in the same y-level. Another technique was to merge two neighbour components when one of them is fully contained in the other. This technique was not so good because in most handwritten cases, because the components forming the character will not be fully contained in each other, instead, they will share only a some common width. We try to enhance the previous technique by using a threshold percentage, so instead of checking a component to be 100% contained in another one, we check if a component is P% contained in another one, this enhancement results in slightly better output. Our best outputs results from the following technique: By looping through components from left to right, we only merge neighbour components whenever the difference between the original component width and the width after merging is smaller than a certain threshold. And we use the same threshold calculated in word segmentation stage.

VI. CLASSIFICATION

The core of this project relays on handwritten characters recognition, which basically means classification. C++ is a symbol-rich programming language, it uses almost every visible character in the ASCII table, which turns to be a very sophisticated problem to solve. The basic course weve followed to overcome the problems is described next.

We started by looking for a suitable and reliable digits-characters dataset (0-9 a-z A-Z), after doing some research we selected the EMNIST (Extended MNIST) data, which contains the required set. The EMNIST comes with different styles (ByClass, ByMerge, Balanced, ...etc). After some trials we decided to neglect the upper-case characters as we needed to reduce the output classes from the neural network. Then we just selected the required characters from the ByClass category.

After selecting this part of the dataset, we tried to find handwritten special-characters dataset to use, but we couldnt find any possible dataset for the complete set. We found dataset related to mathematics symbols on Kaggle and we selected some of these symbols (e.g. brackets and basic math operations), but we still couldnt find many other symbols. We then followed another approach, we generated the remaining special characters using some of the handwritten fonts available online for free (we examined these fonts in depth to make sure that

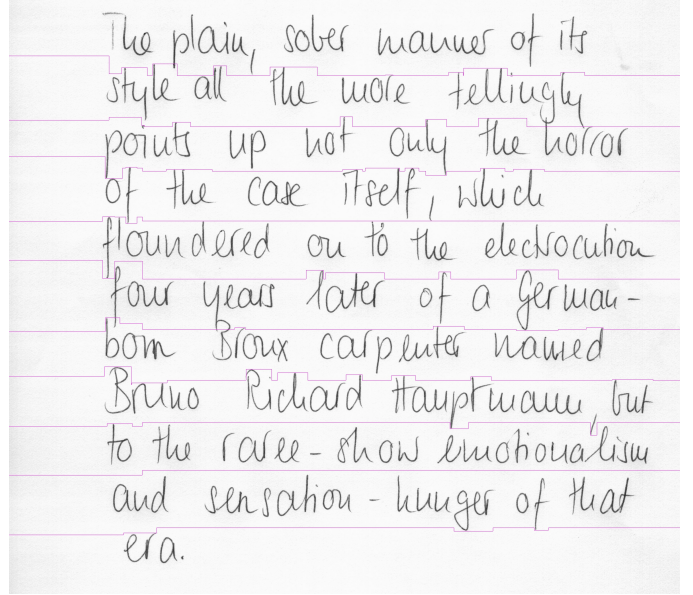


Fig. 8. Correct line segmentation for ordinary text page.

they are somehow close to normal handwriting). This resulted in a 60 samples/special-character only.

Now comes the classification part, where we had to select the best classifier for our case. The classifiers were working pretty good on the normal digits-characters dataset (as mentioned in the EMNIST paper), after merging the special characters dataset with this original dataset the classifiers started to struggle. Here we will examine our trails:

- 1) KNN: being one of the most widely used handwriting classification methods, we gave KNN our first shot, the results on the training set wasnt more than 65%, and on the operational dataset the accuracy was pretty low (25%).
- 2) SVM: the SVM model didnt give results more than what the KNN did.
- 3) Adaptive boosting:
 - NN: we implemented a special voting-NN to classify the character as special character or normal digit-character, then we applied the adaptive boosting algorithm on this voting neural network to achieve better results and give higher weights for the misclassified characters, and we tried to balance the whole dataset, but the classifier didnt enhance a lot.
 - Decision Tree: applying the same algorithm using decision tree didnt enhance too, which highlights the main problem to be the dataset.
- 4) NN: we built a basic feedforward neural network from scratch using python, it uses the MSE function and sigmoid function as its activation function. The neural network accuracy was a little higher than the previous methods. However, it was lower than what can be accepted as image2code program. The main challenges

```

#include <bits/stdc++.h>
using namespace std;
const int N = 100100;
int n, m, u, v, i, vis[N]; long long ans;
pair<int, int> p[N];
vector<int> edges[N];
void dfs (int u) {
    if (vis[u]++) return;
    for (int v : edges[u]) dfs(v);
}
int main () {
    cin >> n >> m;
    for (i = 1; i <= n; ++i) {
        cin >> p[i].first; p[i].second = i;
    }
    sort (p+1, p+n+1);
    while (m--) {
        cin >> u >> v;
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    for (i = 1; i <= n; ++i) {
        u = p[i].second;
        if (!vis[u]) {
            dfs(u);
            ans += p[i].first;
        }
    }
    cout << ans; return 0;
}

```

Fig. 9. The other method of line segmentation failed here to segment some lines.

were that the special characters dataset isn't the same as the original dataset (quantity and some other visual features) and the fact that many special characters are actually composing many other characters which may confuse the neural network especially when the dataset isn't similar.

Trying to balance dataset, we used the EMNIST balanced dataset and the special characters dataset combined, but this didn't do us any good.

After so many trials we figured out that the best options we have now is to separate the special characters classifier and the digits-characters classifier. This way we could train two different models to classify each category. After laying down this assumption (that special characters can be distinguished from other characters by their writing color-red-), we tried to enhance each neural network of the two to classify with a good accuracy.

Enhancing the digits-characters dataset was mainly done in the post-processing phase. However, to enhance the special-characters dataset, we used a data augmentation named Augmentor to generate more samples (using rotation only) for the special-characters that got low confidence value in the neural network output). We have

```

#include <iostream>
#include <stdio.h>
using namespace std;
int main() {
    int Prob, X, Y;
    cin >> Prob >> X >> Y;
    for (int i = 0; i < X; i++) {
        cin >> Prob;
        X += Prob;
        Y /= Y; i++;
    }
    cout << "Helloworld" << endl;
    return 0;
}

```

Fig. 10. The other method of line segmentation succeeded here.

```

int main () {
    cin >> n >> m;
    for (i = 1; i <= n; ++i) {
        cin >> p[i].first; p[i].second = i;
    }
    sort (p+1, p+n+1);
}

```

Fig. 11. Dynamic threshold approach

also dilated the symbols provided by the Kaggle math dataset such that they come close to the actual written characters thickness.

We have finally trained the 2 neural networks using a (784, 300, #classes) neurons, after getting the output, we used another layer of post-processing to make use of the previous knowledge of C++ dictionary.

VII. POST-PROCESSING

As we have a prior knowledge that the given input is a C++ code, we could make use of this to increase the accuracy (85

- 1) Predicting the special characters depending on the previous or next ones (++ && --).
- 2) We've also assumed that the user cannot name a function or variable using digits, we can use this piece of information to replace the hard-to-classify characters (9-g, S-5, ...etc)

```

int main () {
    cin >> n >> m;
    for (i = 1; i <= n; ++i) {
        cin >> p[i].first; p[i].second = i;
    }
    sort (p+1, p+n+1);
}

```

Fig. 12. Constant threshold approach (with T=35)



Fig. 13. Character segmentation (before and after merging components)
Notice how the (x and y) are not considered a single character after merging although they share a common width.

- 3) We also replace some words by the C++ keywords if some matching criteria is satisfied.

REFERENCES

- [1] B. Gatos, I. Pratikakis, S.J. Perantonis, "Adaptive degraded document image binarization," *Computational Intelligence Laboratory, Institute of Informatics and Telecommunications, National Center for Scientific Research Demokritos, 153 10 Athens, Greece*
- [2] Manivannan Arivazhagan, Harish Srinivasan and Sargur Srihari, "A Statistical approach to line segmentation in handwritten documents", *Center of Excellence for Document Analysis and Recognition (CEDAR) University at Buffalo, State University of New York*
- [3] Mohammad Imrul Jubair, Prianka Banik, "A Simplified Method for Handwritten Character Recognition from Document Image"
- [4] Holger Schwenk, Yoshua Bengio "Adaptive Boosting of Neural Networks for Character Recognition"
- [5] <http://neuralnetworksanddeeplearning.com/chap1.html>
- [6] <http://yann.lecun.com/exdb/mnist/>
- [7] <https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/>