Writer Identifier

CMP450

Team Members:

- Radwa Samy Khattab 1 – 18
- Salma Ibrahim 1 – 26
- Mariam Hesham 2 – 21
- Nehal Abdelkader 2 – 28

Delivered to

Dr. AbdelMoneim Bayoumi

Eng. Hussein Fadl

Table of Contents

I. Chosen Language & Pre-installed Libraries

We used Python3 as our programming language.

We also used multiple modules from Python in this project.

List of needed libraries:

1. NumPy
2. OpenCV
3. Pillow
4. SciKit-Learn

To install these libraries using Conda:

1. `conda install numpy`
2. `conda install -c conda-forge opencv`
3. `conda install -c anaconda pillow`
4. `conda install -c anaconda scikit-learn`

To easy install them, simply run the bash file "requirments.sh"

II. How to Run

Copy folder "data" with the test cases (00, 01, 02…etc.) into the project folder. Which would contain all needed python files.

Then run "**`python WriterIdentifier.py`**".

After it finishes, you'd find two files generated; "**time.txt**" and "**results.txt**".

The time file has the time taken of each test case, starting before preprocessing till after getting the prediction from the model.

The results file has the prediction of each test case, which is the author ID (author folder) name.

III. Project Pipeline

There're two python files in the project, that aren't exactly necessary if you will run the project on ready prepared data distributed into test cases.

But since we are using IAM dataset, we first ran seperateImages.py file which separated images depending on whose author it belongs to using the provided metadata in "forms.txt" in the dataset.

Then we needed to generate some test cases, so we ran "generateTestCases.py" which generated 100 test cases folders, each case has 3 author folders, each having 2 forms belonging to that author, and finally one test image which belong to one of the authors, but not one of the 2 images for training. We used this test cases in our analysis.

Our main project starts from “WriterIdentifier.py” python file, which imports functions from 3 other files; “preprocessing.py”, “featureExtraction.py” and “model.py”.

We start by reading all test cases in “data” folder, and looping on them.

For each test case:

- Read Images of authors and test image
- Calculate start time
- Call preprocess function to divide the form images into lines and return these new images
- Call features function to extract the features in all given images and return feature vectors in form of: X training set, Y training labels, and X_test having features of the test lines.
- Call model function which trains on X and Y, then predicts the author of X_test
- Calculate end time and duration
- Write prediction and time in their perspective files results.txt and time.txt

IV. Preprocessing Module

In preprocessing module, we loop on the images for each author, then for each, we extract the lines in this form, and finally store separate images each representing a line. These images are what we work on in the Feature Extraction Module.

To do this, we first threshold the image and convert it to binary. With biased limits to remove noise and just leave the handwriting only.

Then we dilate the image with kernel/window of 19 horizontal pixel for 10 iterations. This result in merging all words into one line. Which makes it easier to get one outer contour including the line text. And we crop the white spaces before first word and after last word.

These cropped images are then saved in grayscale values.

V. Feature Extraction

Generally, we loop on images—lines—and for each line we extract the features in it depending on the provided method to the function—explained in following trials—and returns:

- X: feature vector of shape NxM
  - N: number of data/images we extracted features from
  - M: size of feature vector (depends on which feature we extracted)
- Y: labels of authors, of size N having corresponding author label of each line
- X_test: feature vector of size KxM
  - K: number of lines in test image

Based on the trials results—in analysis section—we finalized the module to extracting Edge direction feature—explained in B (second trial) below.

A. First Trial—White spacing & Baselines

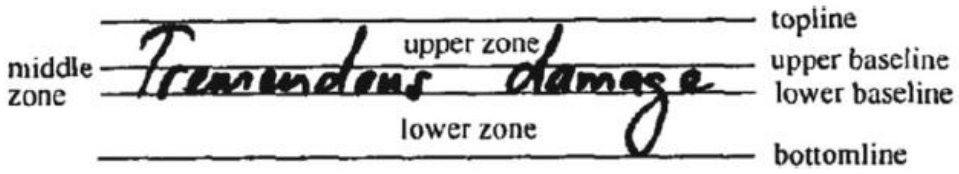
First try we extracted 4 features only from each line:

1. White spacing between each two words, then taking median of these values
2. Ratio between topline and upper baseline—explained below
3. Ratio between upper baseline and lower baseline
4. Ration between lower baseline and bottom line

Topline, upper-baseline, lower-baseline and bottom line are 4 extracted features from each line.

We considered this method from paper "*Text-Independent Handwriting Classification Using Line and Texture-Based Features"*.

The following figure explains each line.



To get these lines:

We got the histogram of black pixels in each line. The topline is the index of the first row having any black pixel. Similarly, the bottom line is the index of the last row having a black pixel.

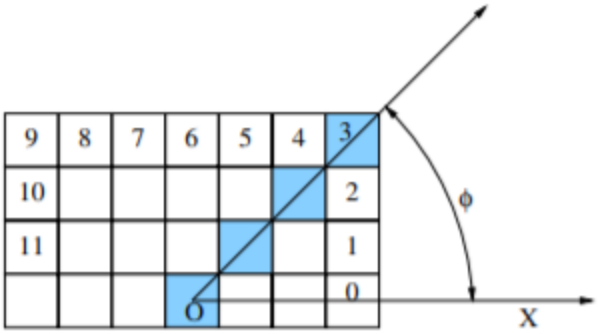
To get both baselines, we calculated the average number of black pixels in each row, then the upper-baseline is the first row having black pixels ≥ average, and lower-baseline is the last row having black pixels ≥ average.

B. Second Trial—Edge-direction (selected)

In this trial we tried a feature called "Edge-direction Distribution". We considered this idea based on paper "*Writer Identification Using Edge-Based Directional Features*".

Feature extraction starts with conventional edge detection (we used canny) that generates a binary image. We then consider each edge pixel in the middle of a square neighborhood and we check in all directions emerging from the central pixel and ending on the periphery of the neighborhood for the presence of an entire edge fragment.

All the verified instances are counted into a histogram. We used n=12 directions which is also the number of bins in the histogram.



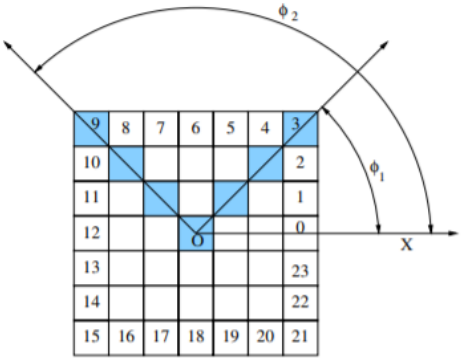
C. Third Trial—Edge-Hinge direction

In this trial we used a feature called "Edge-hinge distribution". This idea is taken from same pervious paper.

The method of feature extraction is similar to the one previously described, but it has added complexity. The central idea is to consider in the neighborhood, not one, but two edge fragments emerging from the central pixel.

This time the directions are 2n = 24 and from the total number of combinations of two angles we will consider only the non-redundant ones ($\emptyset_2 > \emptyset_1$) and we will also eliminate the cases when the ending pixels have a common side. Therefore, the final number of combinations

$C(2n, 2) = n(2n - 3)$ and accordingly, our hinge feature vector will have 252 components.



D. Forth Trial—LBP

In this trail we used a texture feature LBP (local binary pattern) to represent the handwritten feature. We worked on a gray scale image of each and we did like threshold for each pixel but it depends on its 8-neighbor pixel value.

$$LBP_{P,R}(x, y) = \sum_{i=0}^{P-1} s(g_i - g_c)2^i$$

We calculate LBP value for each black pixel only by applying this equation which represents the relation between this pixel and its neighbors. LBP is a decimal value and s function returns 1 if the pixel is darker than its neighbor, and 0 if it's brighter.

Then get histogram for this LBP values for each line this will be our feature vector with length of 256.

E. Fifth Trial—Edge Direction

Same as Second Trial, but we modified the selection model.

VI. Selected Model

We used KNN model with K = 3.

KNN model votes the test feature vector depending on the nearest k-feature vectors.

For example, if k = 3 and after knowing the closest 3 feature vectors to our test case, we found that 2 belong to class 1, and 1 belong to class 2, then KNN model predicts the test case to be class 1—highest number of votes.

Since we have multiple lines in the test image, we predict each line separately and store its prediction. Then our final prediction for the image is the most frequent prediction in lines.

VII. Performance Analysis

Using the 100 test cases we generated—described in project pipeline section—to test each of our trials and calculate its accuracy.

We tried running the program multiple times as the trials mentioned in Feature Extraction section. Here we will mention the accuracy we got in each trial.

Note: We noticed that Edge-direction features took the least time too among these trials.

A. First Trial—White spacing & Baselines

Using KNN selection model, with k = 3.

Accuracy = 69%

B. Second Trial—Edge direction (selected)

Using KNN selection model, with k = 3.

Accuracy = 95%

C. Third Trial—Edge-Hinge direction

Using KNN selection model, with k = 3.

Accuracy = 93%

D. Forth Trial—LBP

Using KNN selection model, with k = 3.

Accuracy = 93%

E. Fifth Trial—Edge direction

Using KNN selection model, with k = 5.

Accuracy = 94%

After these trial runs, we decided to go with **Edge-direction** for our features and **KNN** model with **K = 3** as our model.

VIII. Team Load Distribution

Each wrote parts of the document depending on their work load.

In the end, we worked together in Writer Identifier (Merging all code in a one file to run)

Member	Work
Radwa Samy	• Researching • Preprocessing Module • White Spacing feature • Model
Salma Ibrahim	• Researching • Edge direction feature • Edge-hinge direction feature • LBP feature
Mariam Hesham	• Researching • Baselines feature • LBP feature • Model
Nehal AbdelKader	• Researching • Edge direction feature • Edge-hinge direction feature