UNIVERSITI MALAYA

Faculty of Computer Science and Information Technology

WIX3001 SOFT COMPUTING

ALTERNATIVE ASSESSMENT

SEMESTER 2 SESSION 2023/2024

LECTURER: DR LIEW WEI SHIUNG

NO	NAME	MATRIC NUMBER
1.	QUAH JUN CHUAN	22004851
2.	TER ZHEN HUANG	22004736
3.	CHENG YEE ERN	22004791
4.	RACHEL LIM	22052702

Table of Contents

1. Introduction

Handwriting analysis and recognition are pivotal in various applications, spanning fields and ranging from forensic science and investigations, psychology to personalized user authentication, and more recently, computer science. This is mainly because the unique characteristics of an individual's handwriting can be used as a biometric identifier, similar to fingerprints or facial recognition.

This project aims to explore the identification of individuals based on their unique handwriting using a dataset inspired by the MNIST dataset (Mikalaichaly, 2018) but labeled with the names of each group member who wrote the characters. The goal is to leverage the distinct characteristics of each person's handwriting to create a reliable classification model. This project employs various primary soft computing methods and techniques, specifically fuzzy logic, neural networks, and evolutionary computing. These methods offer diverse approaches to solving the classification problem, each bringing unique strengths and insights. The integration of these methods not only enhances the accuracy of handwriting recognition but also provides a comprehensive understanding of the underlying features that differentiate individual handwriting styles.

By developing a robust system that can accurately classify handwriting samples, the implementation of this project not only highlights the practical applications of soft computing methods but also underscores the importance of interdisciplinary approaches in solving complex classification problems.

2. Problem Statement

Unlike traditional handwriting recognition systems that classify characters or words, our goal is to classify the writer. This task presents several challenges, including variations in writing style, pressure, slant, and other individual nuances. The dataset must be carefully collected and preprocessed to capture these variations effectively. Additionally, the models must be capable of generalizing well to new, unseen samples of handwriting, necessitating robust training and validation procedures. Our approach involves using soft computing techniques, which are well-suited to handle the imprecision and uncertainty inherent in handwriting data. Moreover, implementing soft computing techniques helps us in achieving the final goal of this project.

Below are some examples of research problems that could be addressed through the project:

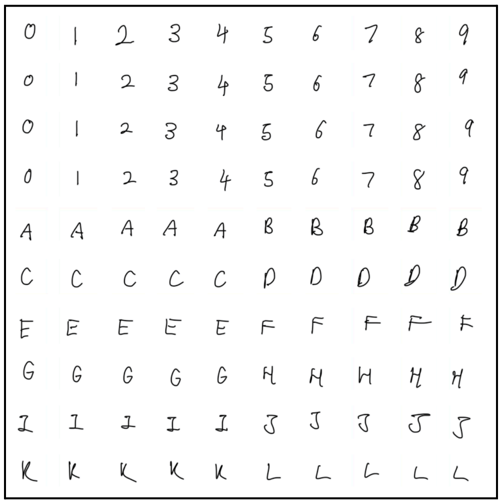
I. How do soft computing techniques outperform traditional methods in approaching handwriting analysis?
II. Which preprocessing techniques yield optimal enhancement of features critical for distinguishing various handwriting styles?
III. Among soft computing techniques or their combinations, which demonstrate superior performance in identifying the user based on their handwriting?

3. Dataset

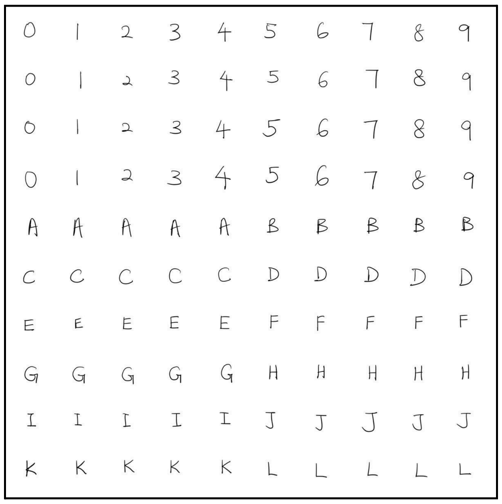
3.1 Dataset Collection

Our dataset is constructed and collected from the handwritten digits and alphabets of our own group members. Within our dataset, we had digits 0-9 written four times each (10 x 4 = 40) and alphabets A-Z written five times each in both uppercase (26 x 5 = 130) and lowercase format (26 x 5 = 130) which makes up to 300 characters per member. As we have four members, our dataset has 1200 characters. The handwritten dataset is written and extracted in picture (.jpg or .png) format. Snippets of the datasets from each member are as shown below where each picture contains 100 handwriting.

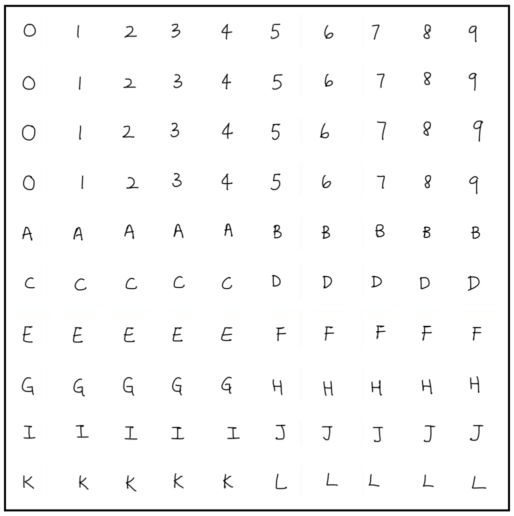
Handwriting Examples:



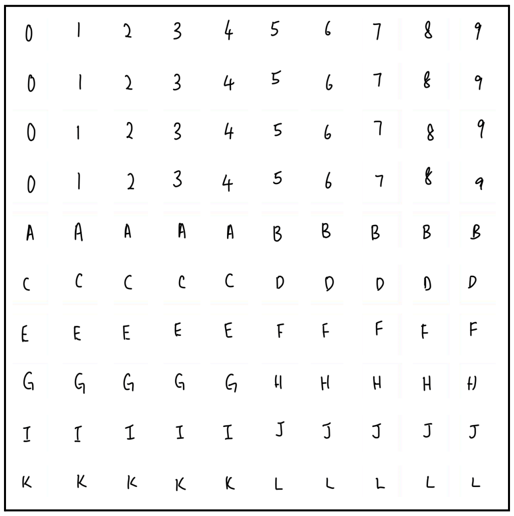
Member 1: Yee Ern



Member 2: Jason (Zhen Huang)



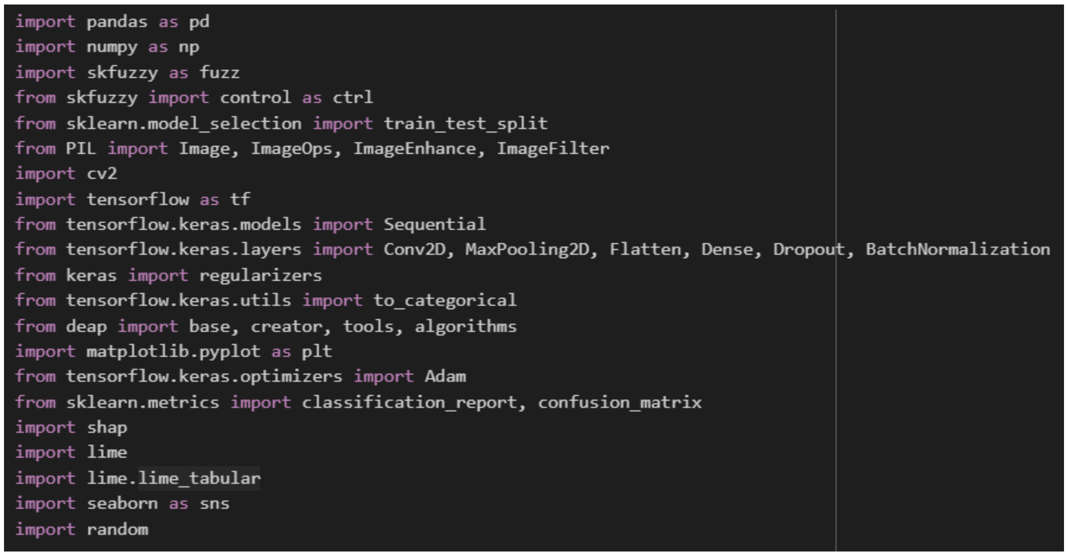
Member 3: Jun Chuan



Member 4: Rachel

3.2 About Dataset - Preprocessing Methods

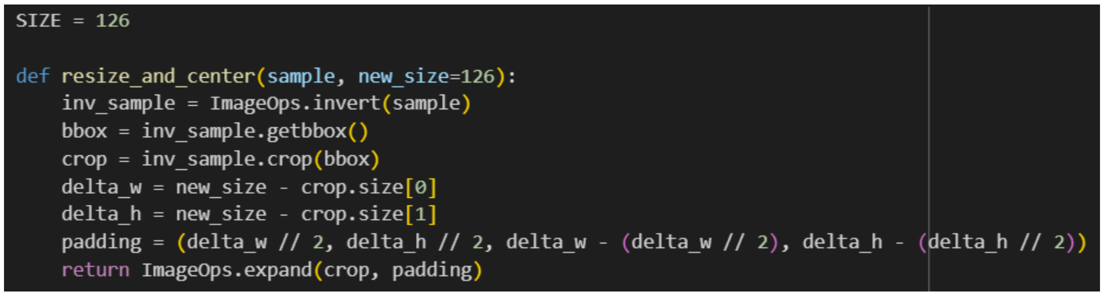
3.2.1 Image Preprocessing and Segmentation



```
import pandas as pd
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
from sklearn.model_selection import train_test_split
from PIL import Image, ImageOps, ImageEnhance, ImageFilter
import cv2
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from keras import regularizers
from tensorflow.keras.utils import to_categorical
from deap import base, creator, tools, algorithms
import matplotlib.pyplot as plt
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import classification_report, confusion_matrix
import shap
import lime
import lime.lime_tabular
import seaborn as sns
import random
```

Diagram 3.2.1.1

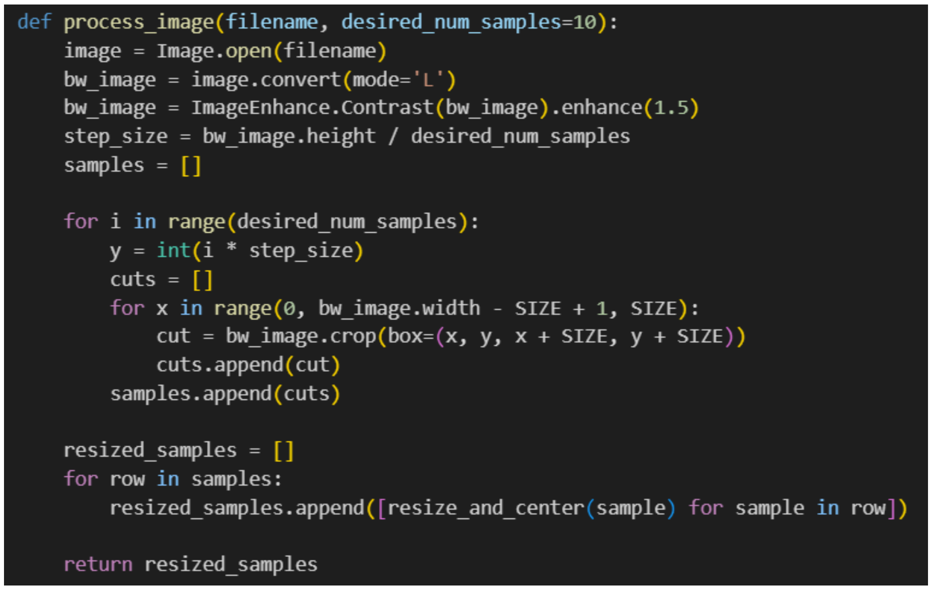
Firstly, we imported necessary libraries such as numpy and pandas; matplotlib.pyplot and seaborn for visualization; skfuzzy, cv2, tensorflow, deap, keras, shap and lime to implement soft computing techniques; sklearn for splitting, training and testing data; random for random generation of values or data; PIL for images processing.



```
SIZE = 126

def resize_and_center(sample, new_size=126):
    inv_sample = ImageOps.invert(sample)
    bbox = inv_sample.getbbox()
    crop = inv_sample.crop(bbox)
    delta_w = new_size - crop.size[0]
    delta_h = new_size - crop.size[1]
    padding = (delta_w // 2, delta_h // 2, delta_w - (delta_w // 2), delta_h - (delta_h // 2))
    return ImageOps.expand(crop, padding)
```

Diagram 3.2.1.2

We first set the SIZE to 126 (in the unit of pixels) which is the crop size for image segments. Next, we processed the images through resizing, centering and converting to prepare image data for further analysis and classification by standardizing their size and format. We used the ‘resize_and_center’ function to invert the image and crop it to the bounding box (‘bbox’) to ensure the background is uniform. The cropped image is then centered within a ‘new_size x new_size’ square by adding padding.

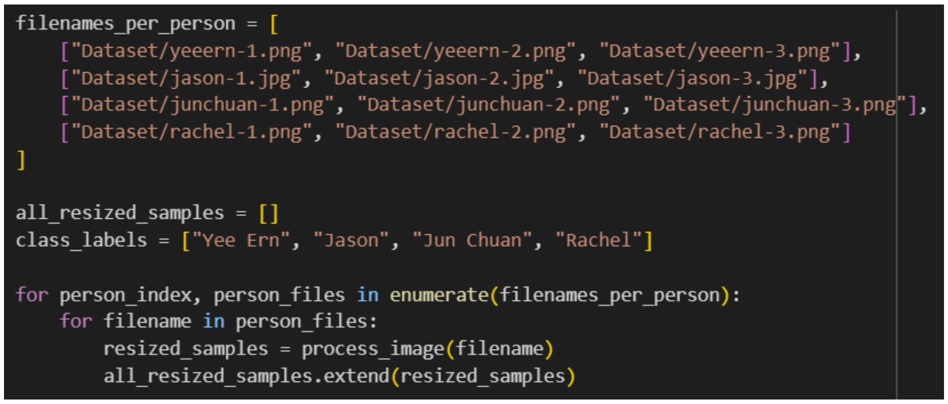


```python
def process_image(filename, desired_num_samples=10):
    image = Image.open(filename)
    bw_image = image.convert(mode='L')
    bw_image = ImageEnhance.Contrast(bw_image).enhance(1.5)
    step_size = bw_image.height / desired_num_samples
    samples = []

    for i in range(desired_num_samples):
        y = int(i * step_size)
        cuts = []
        for x in range(0, bw_image.width - SIZE + 1, SIZE):
            cut = bw_image.crop(box=(x, y, x + SIZE, y + SIZE))
            cuts.append(cut)
        samples.append(cuts)

    resized_samples = []
    for row in samples:
        resized_samples.append([resize_and_center(sample) for sample in row])

    return resized_samples
```

Diagram 3.2.1.3

In the process_image function, we receive 2 parameters named ‘filename’ and ‘desire_num_samples’ set as 10. We opened the input image and converted it to grayscale using the convert (mode='L') method. Then, we enhanced the contrast of the image to make features more distinguishable and segmented the image vertically into ‘desired_num_samples’ (set as 10) equal parts. For each segment, the image is sliced horizontally into non-overlapping patches of size 126x126 pixels. We also called upon the resize_and_center function to process each patch to ensure uniform size and centering. Lastly, a list of resized patches is returned for further processing.

3.2.2 Dataset Preparation

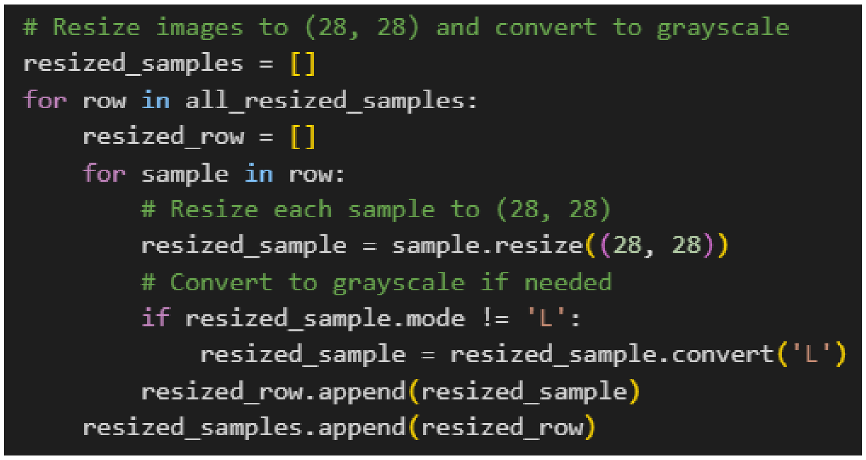


```python
filenames_per_person = [
    ["Dataset/yeeern-1.png", "Dataset/yeeern-2.png", "Dataset/yeeern-3.png"],
    ["Dataset/jason-1.jpg", "Dataset/jason-2.jpg", "Dataset/jason-3.jpg"],
    ["Dataset/junchuan-1.png", "Dataset/junchuan-2.png", "Dataset/junchuan-3.png"],
    ["Dataset/rachel-1.png", "Dataset/rachel-2.png", "Dataset/rachel-3.png"]
]

all_resized_samples = []
class_labels = ["Yee Ern", "Jason", "Jun Chuan", "Rachel"]

for person_index, person_files in enumerate(filenames_per_person):
    for filename in person_files:
        resized_samples = process_image(filename)
        all_resized_samples.extend(resized_samples)
```

Diagram 3.2.2.1

We constructed a filenames_per_person list which contains lists of image filenames for each person, with each sublist corresponding to a different individual. We declared an empty list (‘all_resized_samples’) for future use and labeled a list of ‘class_labels’ having the names of each member. We performed an image processing loop that iterates through each person's images. For each image, we called the ‘process_image’ function to obtain resized patches and collected all processed patches into ‘all_resized_samples’.

3.2.3 Final Resizing and Conversion

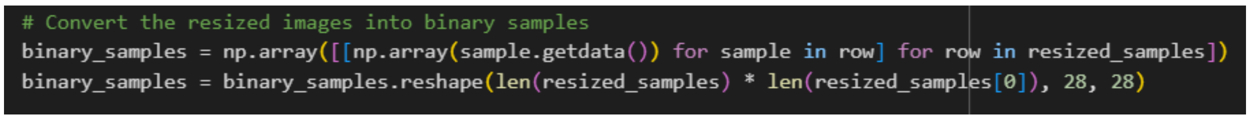


```
# Resize images to (28, 28) and convert to grayscale
resized_samples = []
for row in all_resized_samples:
    resized_row = []
    for sample in row:
        # Resize each sample to (28, 28)
        resized_sample = sample.resize((28, 28))
        # Convert to grayscale if needed
        if resized_sample.mode != 'L':
            resized_sample = resized_sample.convert('L')
        resized_row.append(resized_sample)
    resized_samples.append(resized_row)
```

Diagram 3.2.3.1

We performed final resizing and conversion by iterating through each patch and resizing them to 28x28 pixels, a common size for image recognition tasks. We ensured each patch is in grayscale mode, whereby if it is not in grayscale mode yet, we convert it into grayscale using the .convert(‘L’) method. We then collected these resized patches and added them into a list named ‘resized_samples’.

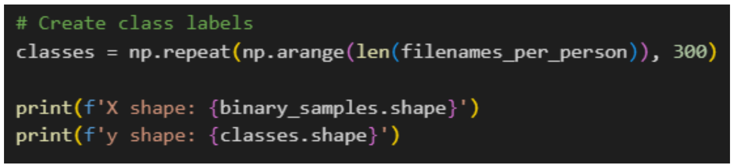
3.2.4 Data Conversion and Storage



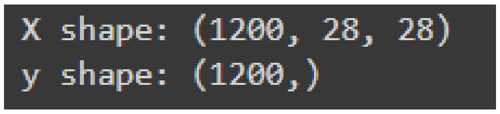
```
# Convert the resized images into binary samples
binary_samples = np.array([[np.array(sample.getdata()) for sample in row] for row in resized_samples])
binary_samples = binary_samples.reshape(len(resized_samples) * len(resized_samples[0]), 28, 28)
```

Diagram 3.2.4.1

We performed binary sample conversion by converting the list of resized images into a NumPy array and reshaping it to the appropriate dimensions for machine learning models using the .reshape() method. Each image is flattened into a one-dimensional array of pixel values, and the result is a 3D array where each 2D slice is an image of size 28x28 pixels.



```
# Create class labels
classes = np.repeat(np.arange(len(filenames_per_person)), 300)

print(f'X shape: {binary_samples.shape}')
print(f'y shape: {classes.shape}')
```

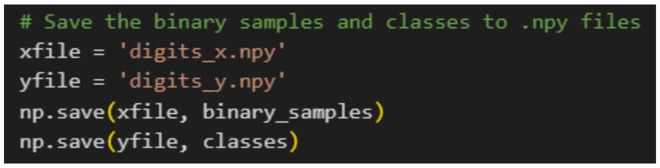


```
X shape: (1200, 28, 28)
y shape: (1200,)
```

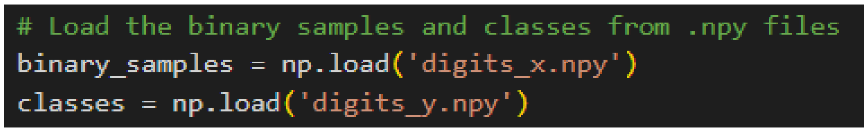
Diagram 3.2.4.2

We created a NumPy array named classes with repeated indices corresponding to each person's label ('filenames_per_person'), where each person's set of 300 images (provided 100 images from each of 3 files per person) gets the same label.

3.2.5 Saving, Loading and Inspecting Data



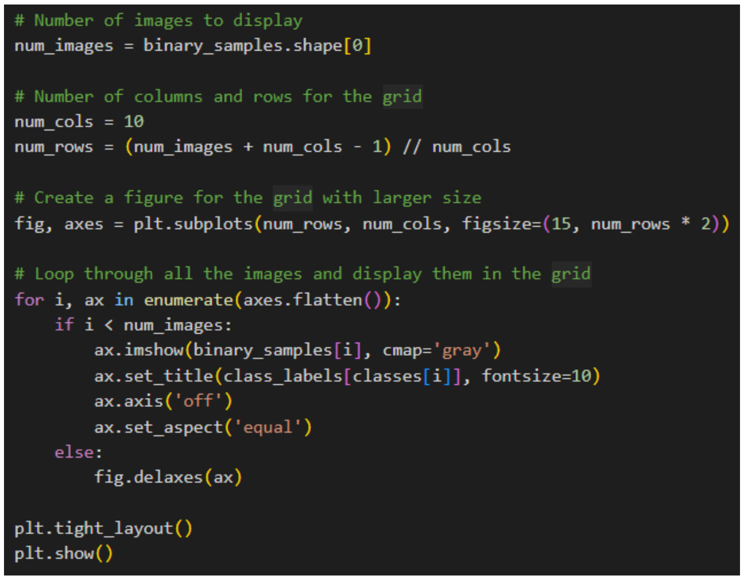
```python
# Save the binary samples and classes to .npy files
xfile = 'digits_x.npy'
yfile = 'digits_y.npy'
np.save(xfile, binary_samples)
np.save(yfile, classes)
```



```python
# Load the binary samples and classes from .npy files
binary_samples = np.load('digits_x.npy')
classes = np.load('digits_y.npy')
```

Diagram 3.2.5.1

We proceeded with saving the binary samples ('digits_x.npy') and class labels ('digits_y.npy') into '.npy' files using the 'np.save' method. We then reloaded the saved arrays of binary samples and class labels using the 'np.load()' method for further use.



```python
# Number of images to display
num_images = binary_samples.shape[0]

# Number of columns and rows for the grid
num_cols = 10
num_rows = (num_images + num_cols - 1) // num_cols

# Create a figure for the grid with larger size
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, num_rows * 2))

# Loop through all the images and display them in the grid
for i, ax in enumerate(axes.flatten()):
    if i < num_images:
        ax.imshow(binary_samples[i], cmap='gray')
        ax.set_title(class_labels[classes[i]], fontsize=10)
        ax.axis('off')
        ax.set_aspect('equal')
    else:
        fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

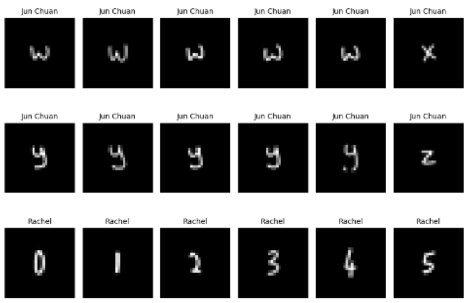
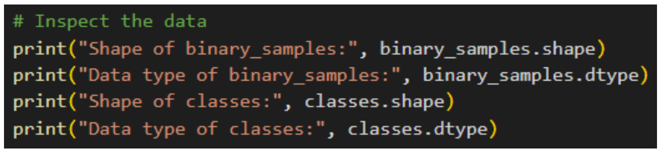
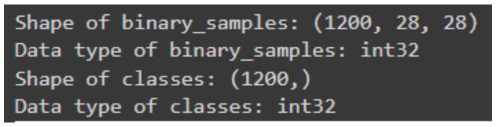


Diagram 3.2.5.2

We determined the number of images to display and set up a grid for visualization by initializing the number of columns and rows for the grid. We created a loop that iterates through the images, displaying each in the grid with corresponding labels (names). Lastly, we ensure that the layout is tidy using the 'ply.tight_layout' method before showing the visualization. The figure above shows a snippet of the example output.



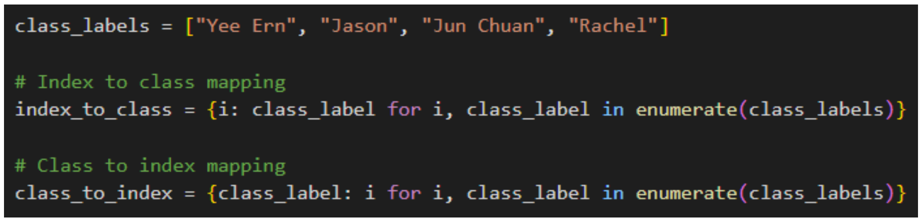
```
# Inspect the data
print("Shape of binary_samples:", binary_samples.shape)
print("Data type of binary_samples:", binary_samples.dtype)
print("Shape of classes:", classes.shape)
print("Data type of classes:", classes.dtype)
```



```
Shape of binary_samples: (1200, 28, 28)
Data type of binary_samples: int32
Shape of classes: (1200,)
Data type of classes: int32
```

Diagram 3.2.5.3

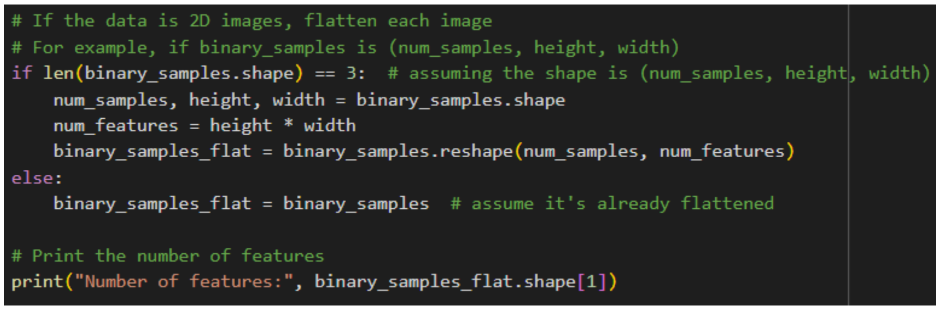
The above print statements output the shape and data type of four loaded arrays: 'binary_samples.shape', 'binary_samples.dtype; 'classes.shape', 'classes.dtype'.



```
class_labels = ["Yee Ern", "Jason", "Jun Chuan", "Rachel"]

# Index to class mapping
index_to_class = {i: class_label for i, class_label in enumerate(class_labels)}

# Class to index mapping
class_to_index = {class_label: i for i, class_label in enumerate(class_labels)}
```

Diagram 3.2.5.4

We created a 'class_labels' list that contains the names of the classes in the dataset, where each name corresponds to a member's name. We created two dictionary comprehensions with them creating a mapping from indices to class labels and class labels to indices respectively. Both dictionaries uses 'enumerate(class_labels)' that generates pairs of index and class label with results in 'index_to_class' dictionary for looking up the class label given an index and 'class_to_index' dictionary for looking up the index given a class label.



```
# If the data is 2D images, flatten each image
# For example, if binary_samples is (num_samples, height, width)
if len(binary_samples.shape) == 3:  # assuming the shape is (num_samples, height, width)
    num_samples, height, width = binary_samples.shape
    num_features = height * width
    binary_samples_flat = binary_samples.reshape(num_samples, num_features)
else:
    binary_samples_flat = binary_samples  # assume it's already flattened

# Print the number of features
print("Number of features:", binary_samples_flat.shape[1])
```

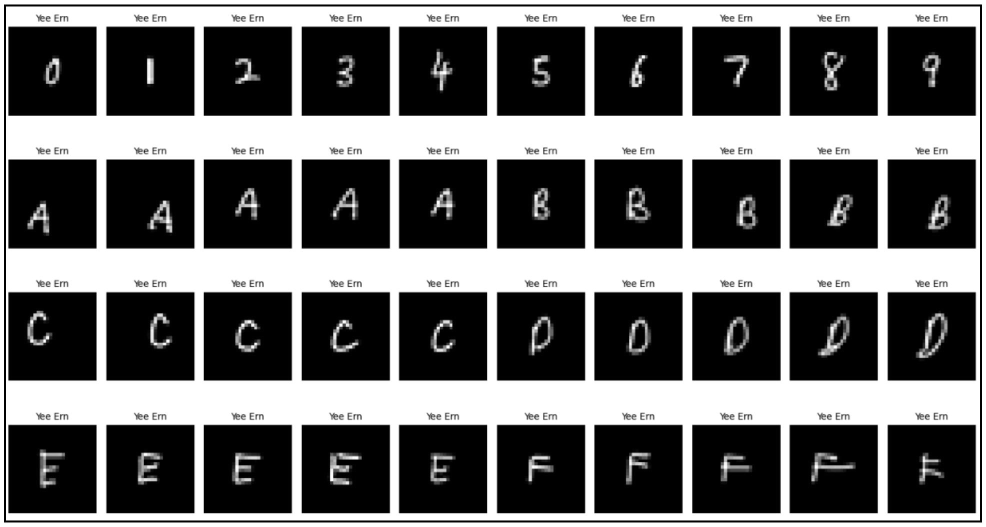
Diagram 3.2.5.5

If the length of the shape is 3 (indicating that the shape is '(num_samples, height, width)', which means the data consists of 2D images), we flatten the binary samples from 3D array into a 2D image by having 'num_samples' and 'num_features' (calculated from 'height' multiplied by 'width') arrays where each row represents a flattened image that will be stored in 'binary_samples_flat'. Lastly, we printed the number of features in the flattened dataset to verify that the flattening process was successful and to understand the dimensionality of the input data.

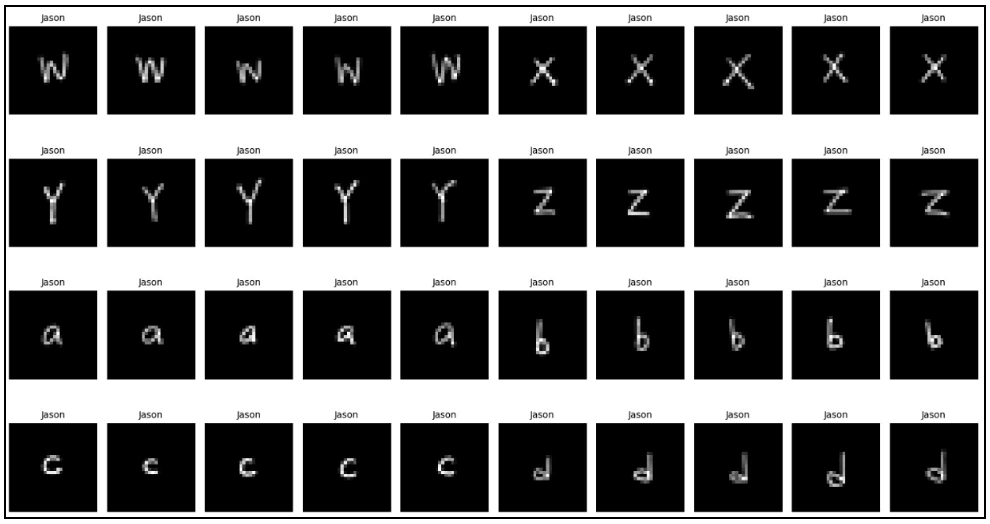
3.3 Examples of Handwritten Characters by each member

Provided are the snippets of images processed originated from each member using the lines of codes above done by Python programming language.

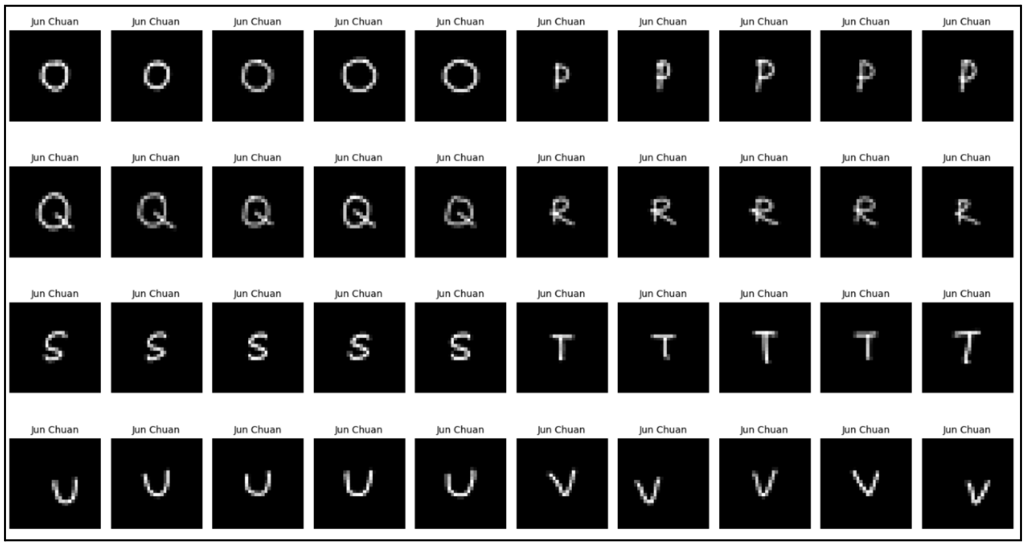
Member 1: Yee Ern



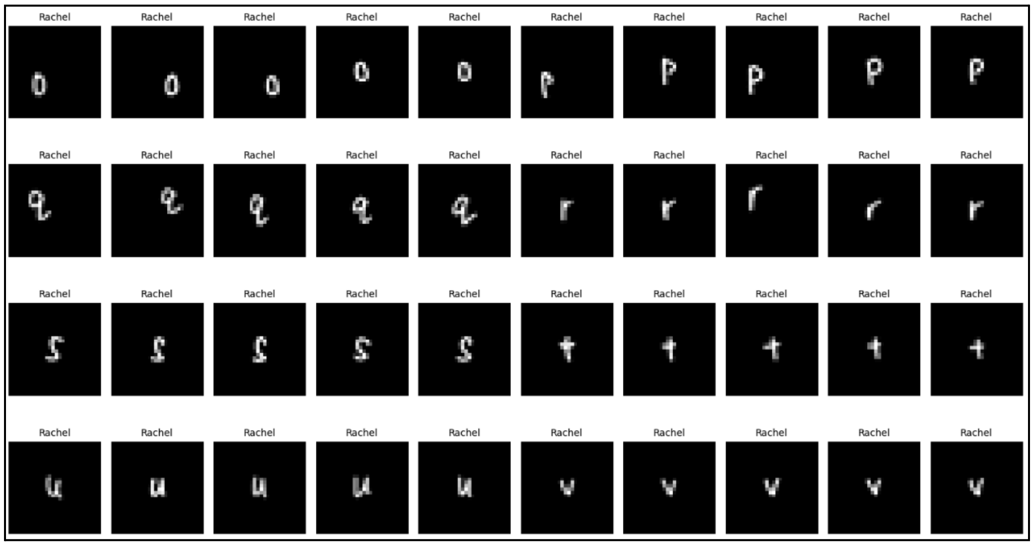
Member 2: Zhen Huang (named as Jason)



Member 3: Jun Chuan



Member 4: Rachel



4. Fuzzy System

We utilize fuzzy systems to further process the binary images to be fed into the neural network, enabling more flexible and human-like decision-making.

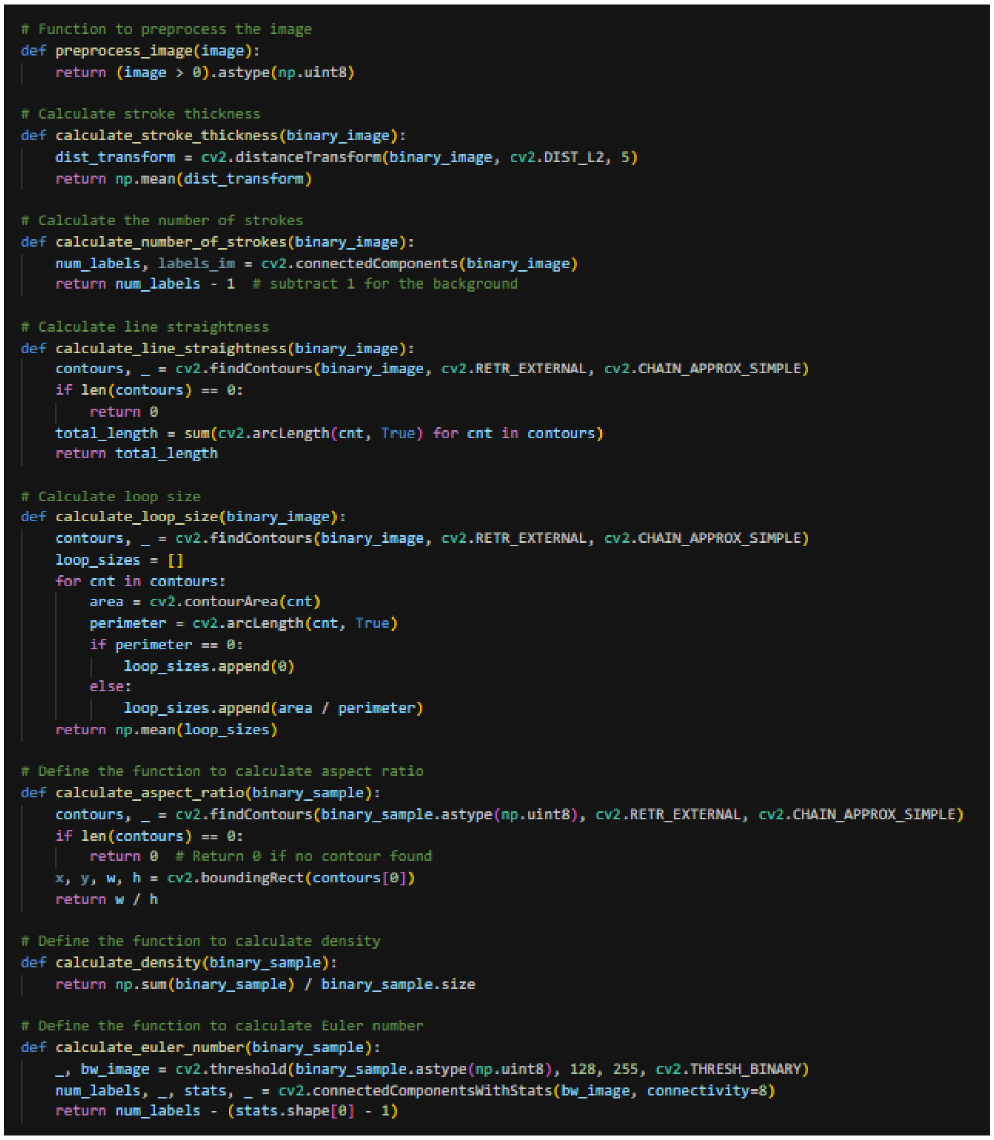
Fuzzy System 1: Feature-based Input Value Extraction

This fuzzy system is designed to extract input values from the features present in the binary images using fuzzy logic. By employing fuzzy logic, which accommodates uncertainty and imprecision, this system gains insights into various characteristics embedded in the images. These characteristics may include stroke thickness, curvature, intensity gradients, or any other relevant features crucial for handwriting analysis. By extracting input values based on fuzzy rules, the system captures nuanced information from the binary images, enriching the dataset with detailed features that enhance the neural network's ability to discern subtle differences in handwriting styles.

Fuzzy System 2: Direct Image Enhancement and Character Detection

Unlike the first fuzzy system, this approach directly applies fuzzy logic to enhance the quality of binary images and improve character detection. By leveraging fuzzy rules, this system enhances image brightness, sharpens edges, and character presence, thereby enhancing the overall quality of the images. These enhancements are crucial for ensuring that the neural network receives high-quality inputs, leading to more accurate character detection and classification. Additionally, fuzzy logic aids in character segmentation and boundary detection, which are essential steps in accurately identifying individual characters within the images.

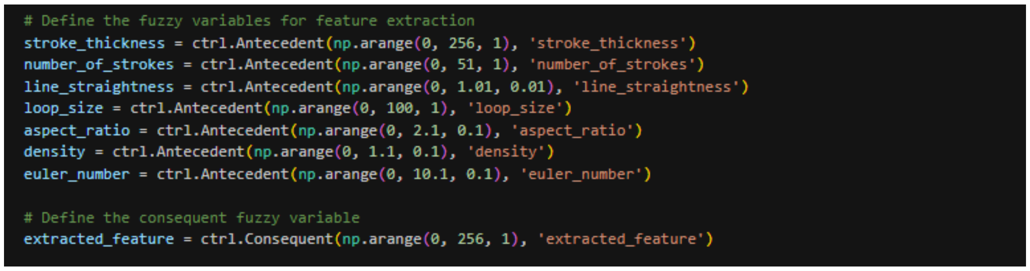
4.1 Fuzzy System 1



```python
# Function to preprocess the image
def preprocess_image(image):
    return (image > 0).astype(np.uint8)

# Calculate stroke thickness
def calculate_stroke_thickness(binary_image):
    dist_transform = cv2.distanceTransform(binary_image, cv2.DIST_L2, 5)
    return np.mean(dist_transform)

# Calculate the number of strokes
def calculate_number_of_strokes(binary_image):
    num_labels, labels_im = cv2.connectedComponents(binary_image)
    return num_labels - 1  # subtract 1 for the background

# Calculate line straightness
def calculate_line_straightness(binary_image):
    contours, _ = cv2.findContours(binary_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(contours) == 0:
        return 0
    total_length = sum(cv2.arcLength(cnt, True) for cnt in contours)
    return total_length

# Calculate loop size
def calculate_loop_size(binary_image):
    contours, _ = cv2.findContours(binary_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    loop_sizes = []
    for cnt in contours:
        area = cv2.contourArea(cnt)
        perimeter = cv2.arcLength(cnt, True)
        if perimeter == 0:
            loop_sizes.append(0)
        else:
            loop_sizes.append(area / perimeter)
    return np.mean(loop_sizes)

# Define the function to calculate aspect ratio
def calculate_aspect_ratio(binary_sample):
    contours, _ = cv2.findContours(binary_sample.astype(np.uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(contours) == 0:
        return 0  # Return 0 if no contour found
    x, y, w, h = cv2.boundingRect(contours[0])
    return w / h

# Define the function to calculate density
def calculate_density(binary_sample):
    return np.sum(binary_sample) / binary_sample.size

# Define the function to calculate Euler number
def calculate_euler_number(binary_sample):
    _, bw_image = cv2.threshold(binary_sample.astype(np.uint8), 128, 255, cv2.THRESH_BINARY)
    num_labels, _, stats, _ = cv2.connectedComponentsWithStats(bw_image, connectivity=8)
    return num_labels - (stats.shape[0] - 1)
```

Diagram 4.1.1

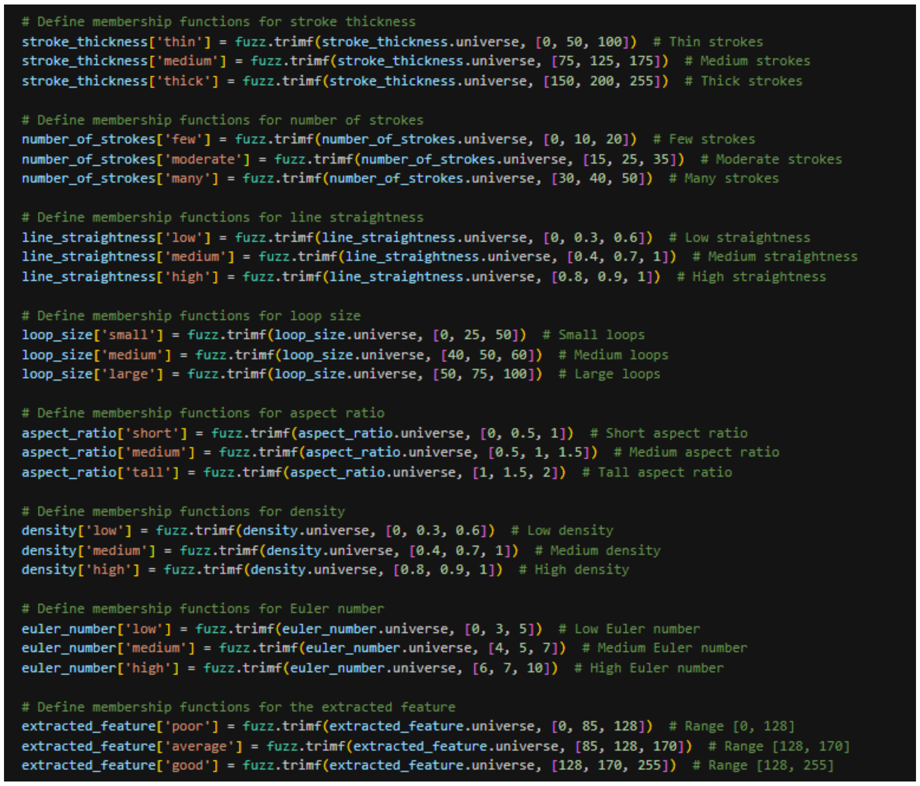
We start by defining feature calculation functions to extract 7 usable features from the binary images: stroke thickness, number of strokes, line straightness, loop size, aspect ratio, density and Euler number. These metrics capture important characteristics of the handwriting, including stroke morphology, spatial arrangement, and complexity. The extracted features serve as informative inputs for further analysis, such as handwriting recognition or writer identification tasks. Overall, these functions enable the transformation of raw image data into quantitative descriptors, aiding in the understanding and classification of handwritten content.



```python
# Define the fuzzy variables for feature extraction
stroke_thickness = ctrl.Antecedent(np.arange(0, 256, 1), 'stroke_thickness')
number_of_strokes = ctrl.Antecedent(np.arange(0, 51, 1), 'number_of_strokes')
line_straightness = ctrl.Antecedent(np.arange(0, 1.01, 0.01), 'line_straightness')
loop_size = ctrl.Antecedent(np.arange(0, 100, 1), 'loop_size')
aspect_ratio = ctrl.Antecedent(np.arange(0, 2.1, 0.1), 'aspect_ratio')
density = ctrl.Antecedent(np.arange(0, 1.1, 0.1), 'density')
euler_number = ctrl.Antecedent(np.arange(0, 10.1, 0.1), 'euler_number')

# Define the consequent fuzzy variable
extracted_feature = ctrl.Consequent(np.arange(0, 256, 1), 'extracted_feature')
```

Diagram 4.1.2

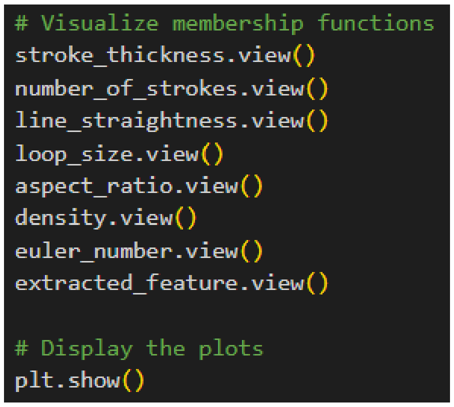
Next, we create fuzzy variables for each feature using the skfuzzy.control.Antecedent and skfuzzy.control.Consequent classes. The antecedents represent the input variable for the fuzzy logic system while the consequent represents the output variable for the fuzzy logic system. These variables represent linguistic terms related to the feature values, such as 'low', 'medium', and 'high'.



```python
# Define membership functions for stroke thickness
stroke_thickness['thin'] = fuzz.trimf(stroke_thickness.universe, [0, 50, 100])  # Thin strokes
stroke_thickness['medium'] = fuzz.trimf(stroke_thickness.universe, [75, 125, 175])  # Medium strokes
stroke_thickness['thick'] = fuzz.trimf(stroke_thickness.universe, [150, 200, 255])  # Thick strokes

# Define membership functions for number of strokes
number_of_strokes['few'] = fuzz.trimf(number_of_strokes.universe, [0, 10, 20])  # Few strokes
number_of_strokes['moderate'] = fuzz.trimf(number_of_strokes.universe, [15, 25, 35])  # Moderate strokes
number_of_strokes['many'] = fuzz.trimf(number_of_strokes.universe, [30, 40, 50])  # Many strokes

# Define membership functions for line straightness
line_straightness['low'] = fuzz.trimf(line_straightness.universe, [0, 0.3, 0.6])  # Low straightness
line_straightness['medium'] = fuzz.trimf(line_straightness.universe, [0.4, 0.7, 1])  # Medium straightness
line_straightness['high'] = fuzz.trimf(line_straightness.universe, [0.8, 0.9, 1])  # High straightness

# Define membership functions for loop size
loop_size['small'] = fuzz.trimf(loop_size.universe, [0, 25, 50])  # Small loops
loop_size['medium'] = fuzz.trimf(loop_size.universe, [40, 50, 60])  # Medium loops
loop_size['large'] = fuzz.trimf(loop_size.universe, [50, 75, 100])  # Large loops

# Define membership functions for aspect ratio
aspect_ratio['short'] = fuzz.trimf(aspect_ratio.universe, [0, 0.5, 1])  # Short aspect ratio
aspect_ratio['medium'] = fuzz.trimf(aspect_ratio.universe, [0.5, 1, 1.5])  # Medium aspect ratio
aspect_ratio['tall'] = fuzz.trimf(aspect_ratio.universe, [1, 1.5, 2])  # Tall aspect ratio

# Define membership functions for density
density['low'] = fuzz.trimf(density.universe, [0, 0.3, 0.6])  # Low density
density['medium'] = fuzz.trimf(density.universe, [0.4, 0.7, 1])  # Medium density
density['high'] = fuzz.trimf(density.universe, [0.8, 0.9, 1])  # High density

# Define membership functions for Euler number
euler_number['low'] = fuzz.trimf(euler_number.universe, [0, 3, 5])  # Low Euler number
euler_number['medium'] = fuzz.trimf(euler_number.universe, [4, 5, 7])  # Medium Euler number
euler_number['high'] = fuzz.trimf(euler_number.universe, [6, 7, 10])  # High Euler number

# Define membership functions for the extracted feature
extracted_feature['poor'] = fuzz.trimf(extracted_feature.universe, [0, 85, 128])  # Range [0, 128]
extracted_feature['average'] = fuzz.trimf(extracted_feature.universe, [85, 128, 170])  # Range [128, 170]
extracted_feature['good'] = fuzz.trimf(extracted_feature.universe, [128, 170, 255])  # Range [128, 255]
```

Diagram 4.1.3

We allocate membership functions to both input and output fuzzification variables by utilizing manually specified values stored in arrays. Each feature is divided into three membership functions: 'low', 'medium', and 'high', all utilizing triangle membership functions.



```
# Visualize membership functions
stroke_thickness.view()
number_of_strokes.view()
line_straightness.view()
loop_size.view()
aspect_ratio.view()
density.view()
euler_number.view()
extracted_feature.view()

# Display the plots
plt.show()
```

Diagram 4.1.4

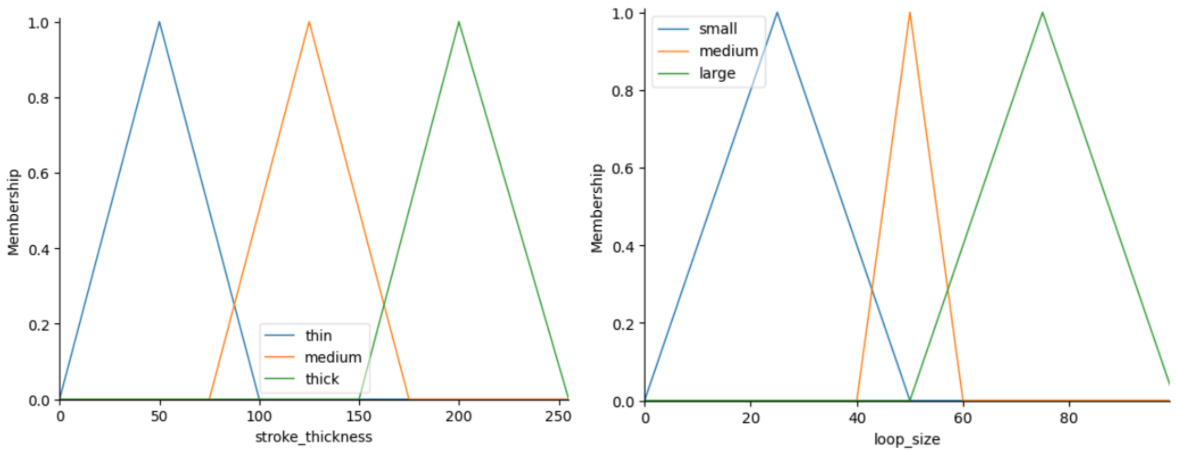
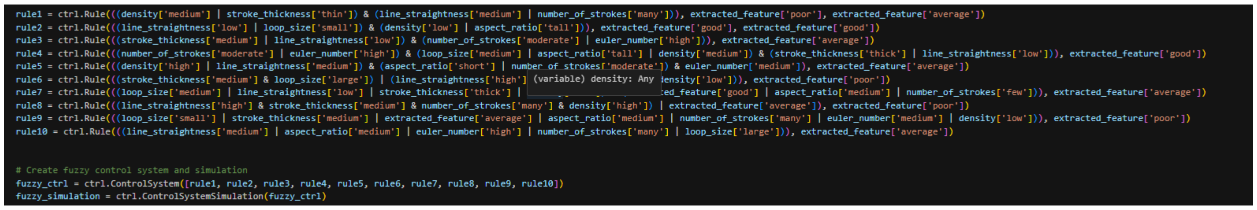


Diagram 4.1.5

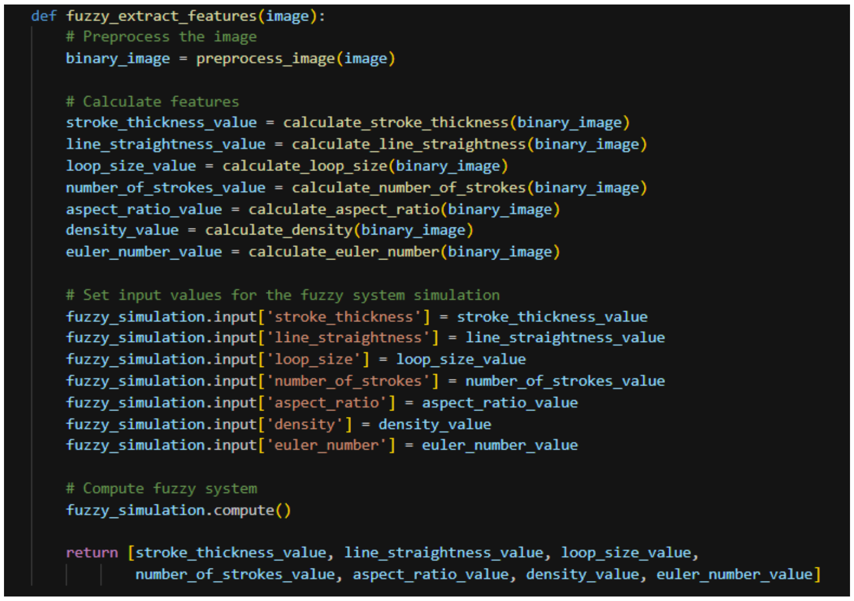
The triangular membership functions of each fuzzy variable are visualized. Some instances of the visualization outputs are screenshot and attached as above.



```
rule1 = ctrl.Rule(((density['medium'] | stroke_thickness['thin']) & (line_straightness['medium'] | number_of_strokes['many'])), extracted_feature['poor'], extracted_feature['average'])
rule2 = ctrl.Rule(((line_straightness['low'] | loop_size['small']) & (density['low'] | aspect_ratio['tall'])), extracted_feature['good'], extracted_feature['good'])
rule3 = ctrl.Rule(((stroke_thickness['medium'] | line_straightness['low']) & (number_of_strokes['moderate'] | euler_number['high'])), extracted_feature['average'])
rule4 = ctrl.Rule(((number_of_strokes['moderate'] | euler_number['high']) & (loop_size['medium'] | aspect_ratio['tall'] | density['medium']) & (stroke_thickness['thick'] | line_straightness['low'])), extracted_feature['good'])
rule5 = ctrl.Rule(((density['high'] | line_straightness['medium']) & (aspect_ratio['short'] | number_of_strokes['moderate']) & euler_number['medium']), extracted_feature['average'])
rule6 = ctrl.Rule(((stroke_thickness['medium'] & loop_size['large']) | (line_straightness['high'] (variable) density: Any ensity['low'])), extracted_feature['poor'])
rule7 = ctrl.Rule(((loop_size['medium'] | line_straightness['low'] | stroke_thickness['thick'] | ed_feature['good'] | aspect_ratio['medium'] | number_of_strokes['few'])), extracted_feature['average'])
rule8 = ctrl.Rule(((line_straightness['high'] & stroke_thickness['medium'] & number_of_strokes['many'] & density['high']) | extracted_feature['average']), extracted_feature['poor'])
rule9 = ctrl.Rule(((loop_size['small'] | stroke_thickness['medium'] | extracted_feature['average'] | aspect_ratio['medium'] | number_of_strokes['many'] | euler_number['medium'] | density['low'])), extracted_feature['poor'])
rule10 = ctrl.Rule(((line_straightness['medium'] | aspect_ratio['medium'] | euler_number['high'] | number_of_strokes['many'] | loop_size['large'])), extracted_feature['average'])


# Create fuzzy control system and simulation
fuzzy_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8, rule9, rule10])
fuzzy_simulation = ctrl.ControlSystemSimulation(fuzzy_ctrl)
```

Diagram 4.1.6

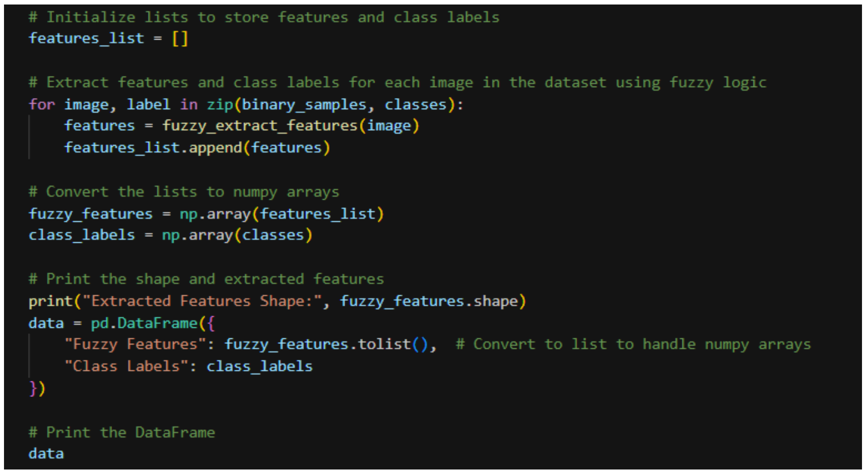
10 different fuzzy rules are defined based on the relationship between the input features and the output feature. Next, the fuzzy control system and its simulation are created.



```python
def fuzzy_extract_features(image):
    # Preprocess the image
    binary_image = preprocess_image(image)

    # Calculate features
    stroke_thickness_value = calculate_stroke_thickness(binary_image)
    line_straightness_value = calculate_line_straightness(binary_image)
    loop_size_value = calculate_loop_size(binary_image)
    number_of_strokes_value = calculate_number_of_strokes(binary_image)
    aspect_ratio_value = calculate_aspect_ratio(binary_image)
    density_value = calculate_density(binary_image)
    euler_number_value = calculate_euler_number(binary_image)

    # Set input values for the fuzzy system simulation
    fuzzy_simulation.input['stroke_thickness'] = stroke_thickness_value
    fuzzy_simulation.input['line_straightness'] = line_straightness_value
    fuzzy_simulation.input['loop_size'] = loop_size_value
    fuzzy_simulation.input['number_of_strokes'] = number_of_strokes_value
    fuzzy_simulation.input['aspect_ratio'] = aspect_ratio_value
    fuzzy_simulation.input['density'] = density_value
    fuzzy_simulation.input['euler_number'] = euler_number_value

    # Compute fuzzy system
    fuzzy_simulation.compute()

    return [stroke_thickness_value, line_straightness_value, loop_size_value,
            number_of_strokes_value, aspect_ratio_value, density_value, euler_number_value]
```

Diagram 4.1.7

All the function above are integrated into a single function to process an image, calculate all the features, sets these value as inputs for fuzzy control system, and computes the fuzzy output

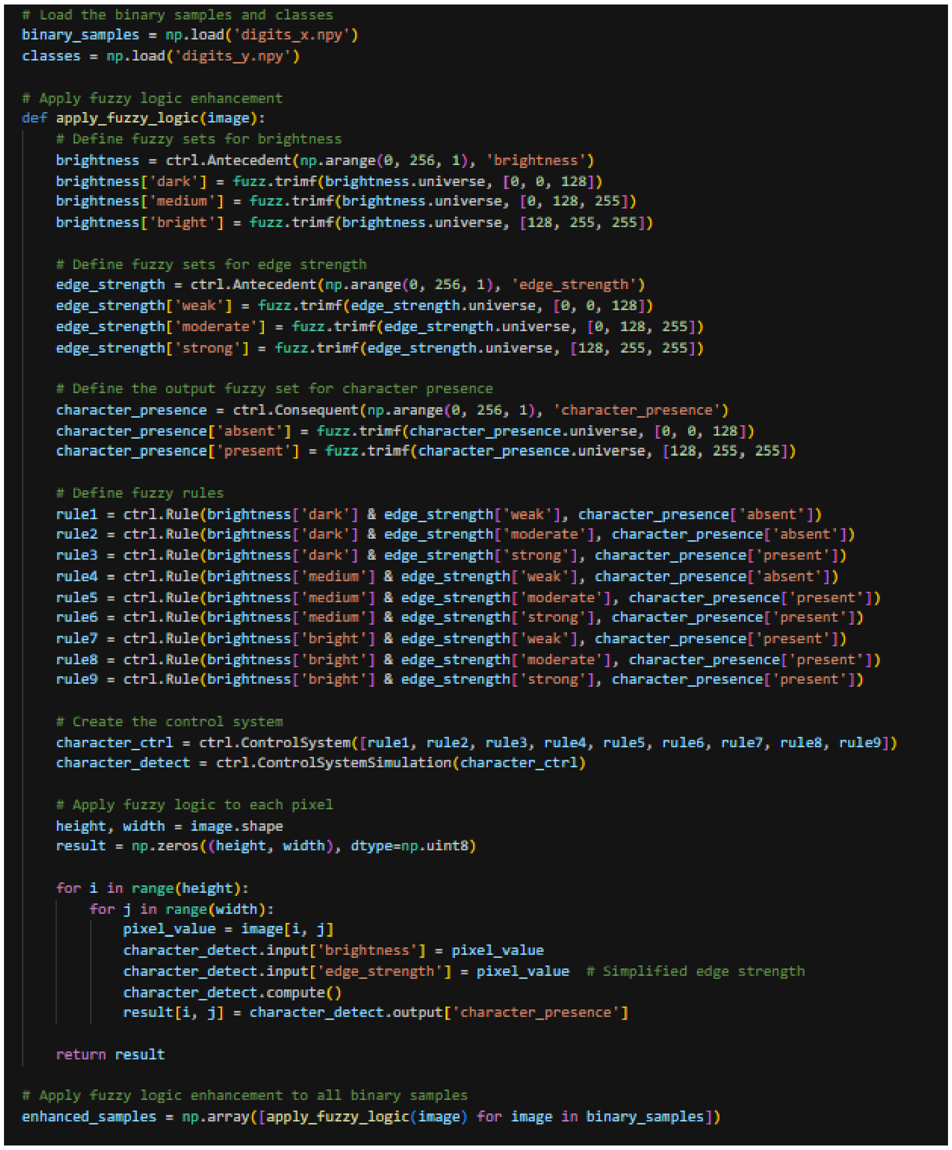


```python
# Initialize lists to store features and class labels
features_list = []

# Extract features and class labels for each image in the dataset using fuzzy logic
for image, label in zip(binary_samples, classes):
    features = fuzzy_extract_features(image)
    features_list.append(features)

# Convert the lists to numpy arrays
fuzzy_features = np.array(features_list)
class_labels = np.array(classes)

# Print the shape and extracted features
print("Extracted Features Shape:", fuzzy_features.shape)
data = pd.DataFrame({
    "Fuzzy Features": fuzzy_features.tolist(),  # Convert to list to handle numpy arrays
    "Class Labels": class_labels
})

# Print the DataFrame
data
```

Diagram 4.1.8

Features are extracted for all images in the dataset using the defined fuzzy extraction function. The features and class labels are together stored in a DataFrame.

4.2 Fuzzy System 2

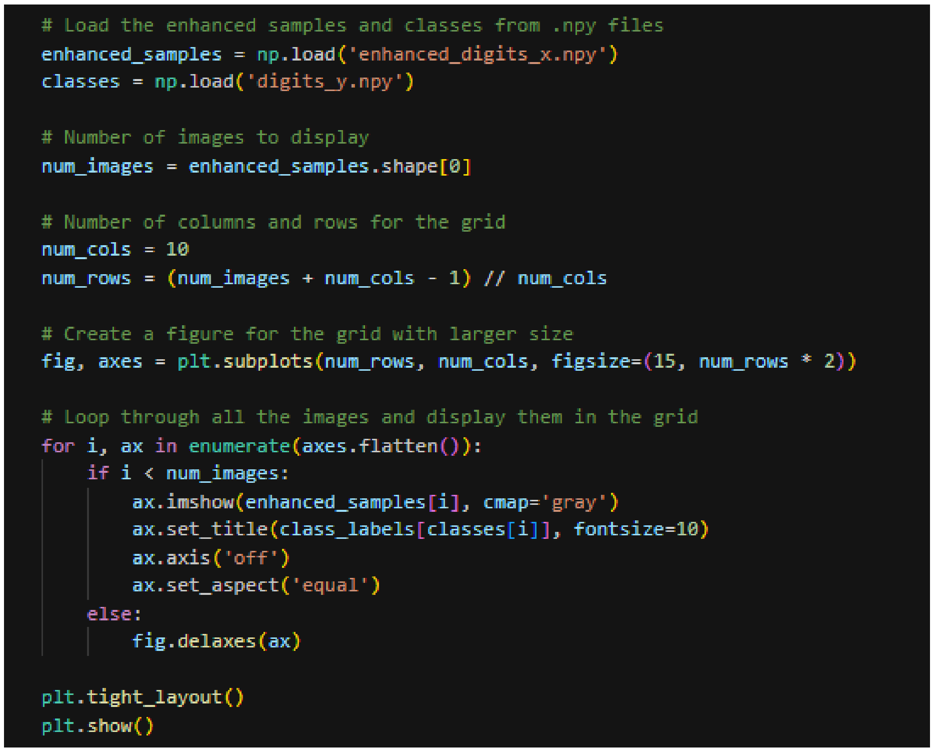


```python
# Load the binary samples and classes
binary_samples = np.load('digits_x.npy')
classes = np.load('digits_y.npy')

# Apply fuzzy logic enhancement
def apply_fuzzy_logic(image):
    # Define fuzzy sets for brightness
    brightness = ctrl.Antecedent(np.arange(0, 256, 1), 'brightness')
    brightness['dark'] = fuzz.trimf(brightness.universe, [0, 0, 128])
    brightness['medium'] = fuzz.trimf(brightness.universe, [0, 128, 255])
    brightness['bright'] = fuzz.trimf(brightness.universe, [128, 255, 255])

    # Define fuzzy sets for edge strength
    edge_strength = ctrl.Antecedent(np.arange(0, 256, 1), 'edge_strength')
    edge_strength['weak'] = fuzz.trimf(edge_strength.universe, [0, 0, 128])
    edge_strength['moderate'] = fuzz.trimf(edge_strength.universe, [0, 128, 255])
    edge_strength['strong'] = fuzz.trimf(edge_strength.universe, [128, 255, 255])

    # Define the output fuzzy set for character presence
    character_presence = ctrl.Consequent(np.arange(0, 256, 1), 'character_presence')
    character_presence['absent'] = fuzz.trimf(character_presence.universe, [0, 0, 128])
    character_presence['present'] = fuzz.trimf(character_presence.universe, [128, 255, 255])

    # Define fuzzy rules
    rule1 = ctrl.Rule(brightness['dark'] & edge_strength['weak'], character_presence['absent'])
    rule2 = ctrl.Rule(brightness['dark'] & edge_strength['moderate'], character_presence['absent'])
    rule3 = ctrl.Rule(brightness['dark'] & edge_strength['strong'], character_presence['present'])
    rule4 = ctrl.Rule(brightness['medium'] & edge_strength['weak'], character_presence['absent'])
    rule5 = ctrl.Rule(brightness['medium'] & edge_strength['moderate'], character_presence['present'])
    rule6 = ctrl.Rule(brightness['medium'] & edge_strength['strong'], character_presence['present'])
    rule7 = ctrl.Rule(brightness['bright'] & edge_strength['weak'], character_presence['present'])
    rule8 = ctrl.Rule(brightness['bright'] & edge_strength['moderate'], character_presence['present'])
    rule9 = ctrl.Rule(brightness['bright'] & edge_strength['strong'], character_presence['present'])

    # Create the control system
    character_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8, rule9])
    character_detect = ctrl.ControlSystemSimulation(character_ctrl)

    # Apply fuzzy logic to each pixel
    height, width = image.shape
    result = np.zeros((height, width), dtype=np.uint8)

    for i in range(height):
        for j in range(width):
            pixel_value = image[i, j]
            character_detect.input['brightness'] = pixel_value
            character_detect.input['edge_strength'] = pixel_value  # Simplified edge strength
            character_detect.compute()
            result[i, j] = character_detect.output['character_presence']

    return result

# Apply fuzzy logic enhancement to all binary samples
enhanced_samples = np.array([apply_fuzzy_logic(image) for image in binary_samples])
```

Diagram 4.2.1

In this approach, we apply fuzzy logic to enhance the input binary image. This code defines fuzzy variables for brightness and edge strength, along with the output variable for character presence. These variables represent linguistic terms related to the image properties, such as 'dark', 'medium', and 'bright'. Membership functions are allocated to both input variables based on manually specified values stored in arrays. Each variable is divided into three membership functions: 'weak', 'moderate', and 'strong', represented by triangle membership functions.

Next, 9 different fuzzy rules are defined to determine the presence of characters based on brightness and edge strength. A fuzzy control system is then created, and simulation is performed for each pixel in the image. This system represents the logic of how brightness and edge strength are processed to enhance the binary image and detect characters.



```python
# Load the enhanced samples and classes from .npy files
enhanced_samples = np.load('enhanced_digits_x.npy')
classes = np.load('digits_y.npy')

# Number of images to display
num_images = enhanced_samples.shape[0]

# Number of columns and rows for the grid
num_cols = 10
num_rows = (num_images + num_cols - 1) // num_cols

# Create a figure for the grid with larger size
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, num_rows * 2))

# Loop through all the images and display them in the grid
for i, ax in enumerate(axes.flatten()):
    if i < num_images:
        ax.imshow(enhanced_samples[i], cmap='gray')
        ax.set_title(class_labels[classes[i]], fontsize=10)
        ax.axis('off')
        ax.set_aspect('equal')
    else:
        fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

Diagram 4.2.2

Fuzzy logic enhancement is applied to all binary samples in the dataset and saved to a .npy file. Then, we display multiple enhanced images in a grid, with their corresponding class labels for better visualization.

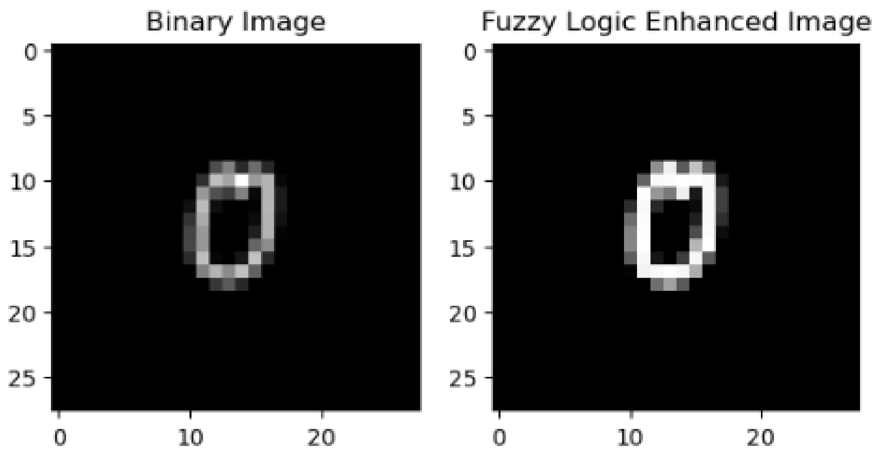


Diagram 4.2.3

As we can see from the visualization above, the binary image becomes clearer after enhancement with fuzzy logic. This will definitely facilitate handwriting identification by the neural network.

5. Neural Network

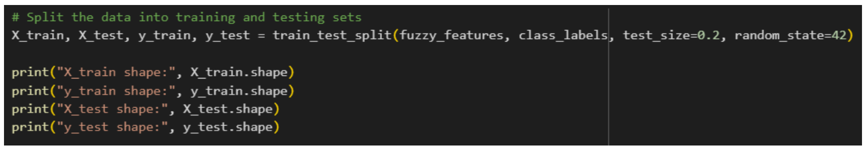
After preprocessing the binary images using fuzzy systems, the refined inputs are then fed into the neural network for further processing and classification. By incorporating fuzzy logic into the preprocessing stage, we not only enhance the quality of input data but also provide the neural network with richer, more informative features extracted from the images.

Neural Network 1: Dense Neural Network (DNN)

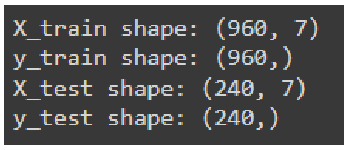
Neural network 1 concentrates on classifying the extracted feature values obtained from the fuzzy system. This model operates by ingesting the raw features derived from the fuzzy logic preprocessing stage. By training on these feature representations, the DNN becomes adept at discerning intricate patterns and relationships present within the data, thereby enabling accurate predictions concerning the identity or characteristics of the handwritten input.

Neural Network 2: Convolutional Neural Network (CNN)

CNN directly classifies the fuzzy-enhanced binary images without explicitly extracting features. With its capacity to learn hierarchical representations of data, CNNs are well-suited for image classification tasks. By feeding the fuzzy-enhanced binary images directly into the CNN, the model learns to discern intricate patterns and features within the images, ultimately making predictions based on these learned representations. This approach bypasses the explicit feature extraction step, allowing the CNN to automatically learn relevant features directly from the input data.



```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(fuzzy_features, class_labels, test_size=0.2, random_state=42)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```



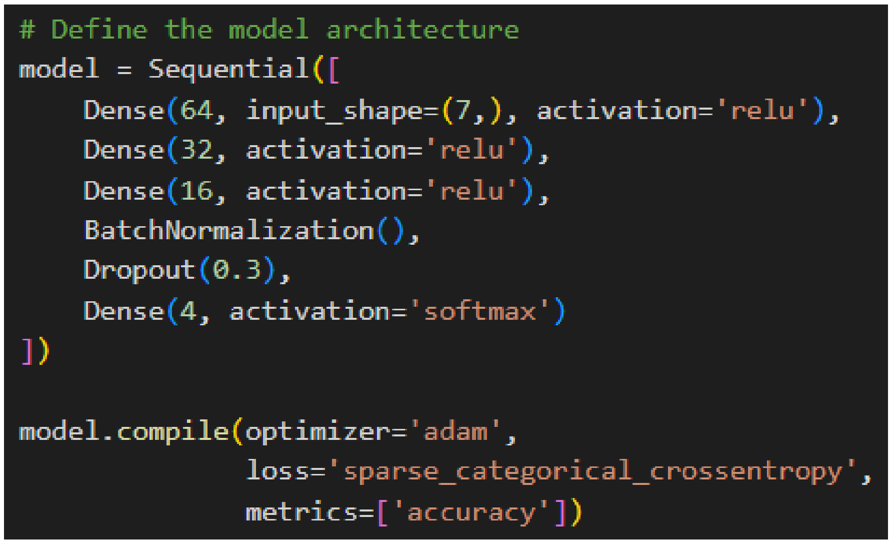
```
X_train shape: (960, 7)
y_train shape: (960,)
X_test shape: (240, 7)
y_test shape: (240,)
```

Diagram 5.1

We first split our data into training and testing sets with a 80-20 ratio and set the random state as 42. We then printed out the shapes for variables X_train, y_train, X_test and y_test.

5.1 Model Architecture

5.1.1 Model 1: DNN



```
# Define the model architecture
model = Sequential([
    Dense(64, input_shape=(7,), activation='relu'),
    Dense(32, activation='relu'),
    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(0.3),
    Dense(4, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

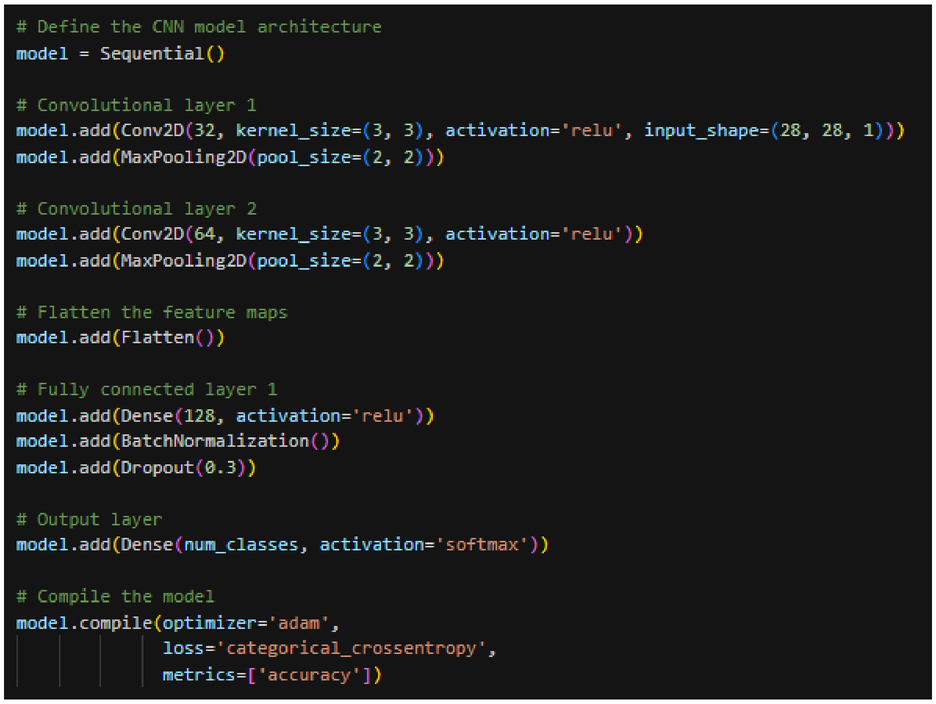
Diagram 5.1.1.1

To build a neural network algorithm, we first defined the Dense Neural Network's model architecture using the Sequential model from the Keras library, which is part of TensorFlow. To explain, Sequential model is a stack of linear layers where each layer has precisely one input tensor and one output tensor. The model consists of several layers, namely the input layer that accepts data with a shape of '(7,)', which means each input sample has 7 features; first dense layer, second dense layer and third dense layer with 64, 32 and 16 neurons each while using the ReLU activation function which introduces non-linearity to the model.

Furthermore, batch normalization layer is implemented whereby the inputs are normalized to the next layer; dropout layer which sets 30% of its inputs to zero during each update cycle to avoid overfitting; and, output layer that has 4 neurons with a softmax activation function which produces a probability distribution over 4 possible cases.

We compiled the model using the Adam optimizer for training, sparse categorical cross-entropy loss function, which is appropriate for multi-class classification tasks and specified the use of the accuracy metric to evaluate the model's performance.

5.1.2 Model 2: CNN



```python
# Define the CNN model architecture
model = Sequential()

# Convolutional layer 1
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Convolutional layer 2
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the feature maps
model.add(Flatten())

# Fully connected layer 1
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))

# Output layer
model.add(Dense(num_classes, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

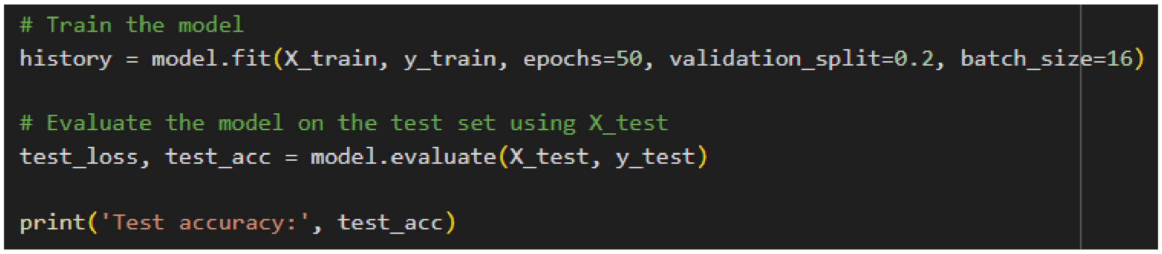
Diagram 5.1.2.1

To build a neural network algorithm, we first defined the Convolutional Neural Network's model architecture using the Sequential model from the Keras library, which is part of TensorFlow. To explain, Sequential model is a stack of linear layers where each layer has precisely one input tensor and one output tensor. The model consists of several layers, namely the input layer that accepts data with a shape of '(28,28,1)', indicating 28 x 28 pixel grayscale images; first Conv2D layer with 32 filters, kernel size of 3x3, and ReLU activation, MaxPooling2D layer with a pool size of 2x2, Second Conv2D Layer with 64 filters, kernel size of 3x3, and ReLU activation, MaxPooling2D Layer with a pool size of 2x2, flatten later and finally dense later.

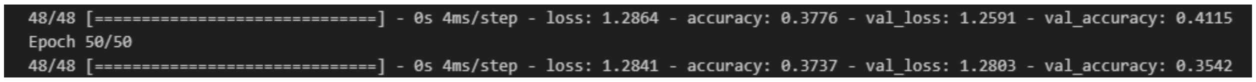
Furthermore, batch normalization layer is implemented whereby the inputs are normalized to the next layer; dropout layer which sets 30% of its inputs to zero during each update cycle to avoid overfitting; and, output layer that has 4 neurons with a softmax activation function which produces a probability distribution over 4 possible cases.

We compiled the model using the Adam optimizer for training, sparse categorical cross-entropy loss function, which is appropriate for multi-class classification tasks and specified the use of the accuracy metric to evaluate the model's performance.

5.2 Result Analysis



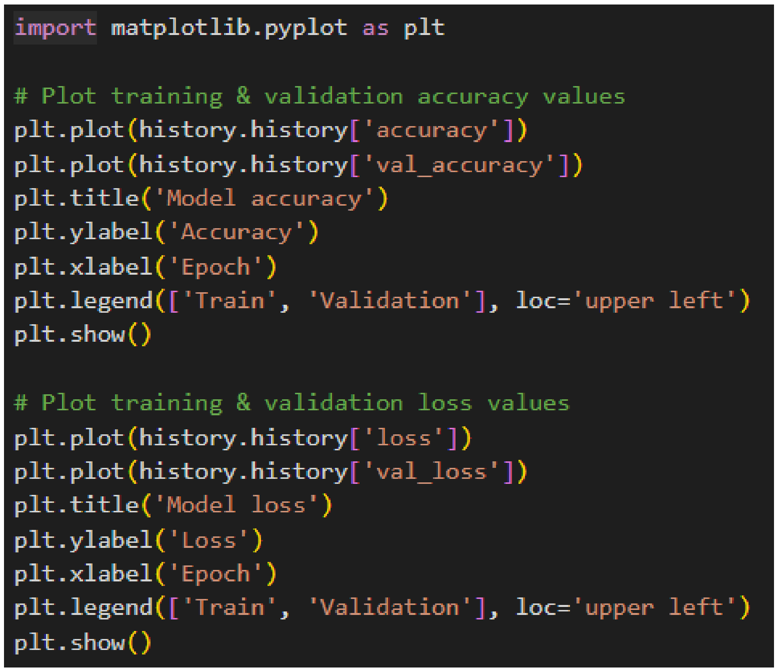
```python
# Train the model
history = model.fit(X_train, y_train, epochs=50, validation_split=0.2, batch_size=16)

# Evaluate the model on the test set using X_test
test_loss, test_acc = model.evaluate(X_test, y_test)

print('Test accuracy:', test_acc)
```



```
48/48 [==============================] - 0s 4ms/step - loss: 1.2864 - accuracy: 0.3776 - val_loss: 1.2591 - val_accuracy: 0.4115
Epoch 50/50
48/48 [==============================] - 0s 4ms/step - loss: 1.2841 - accuracy: 0.3737 - val_loss: 1.2803 - val_accuracy: 0.3542
```

Diagram 5.2.1

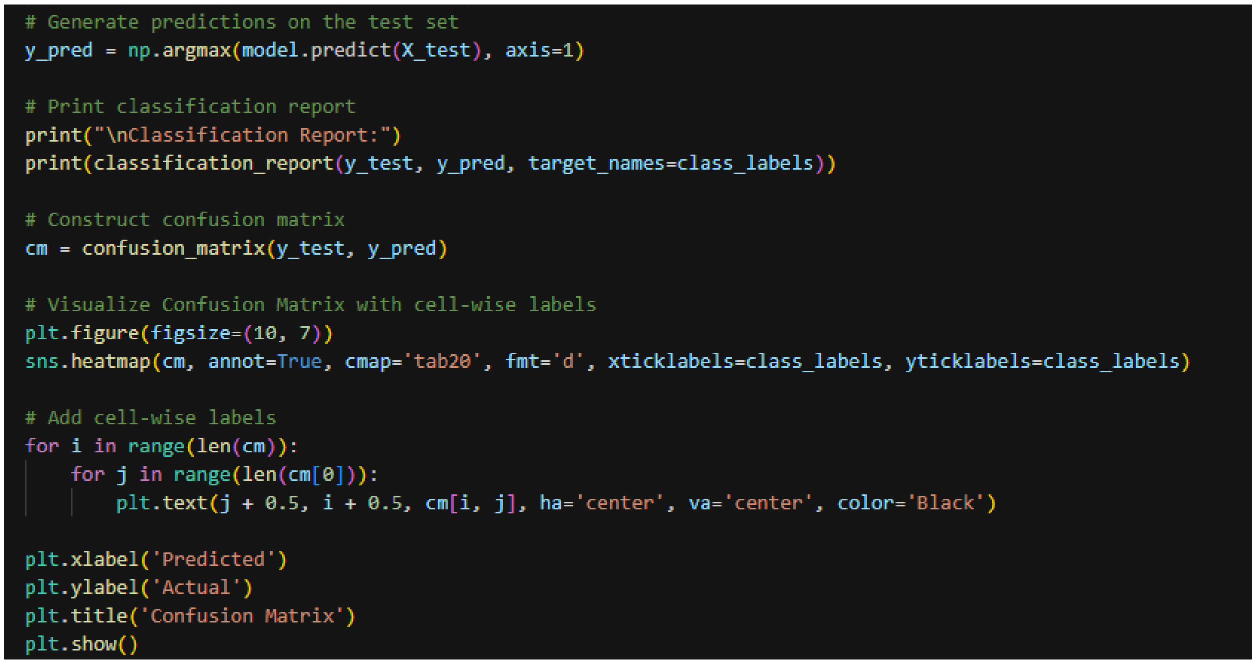
We trained the model on X_train and y_train datasets and included a validation split of 20%, whereby 20% of training data is used as a validation set to monitor the model's performance on unseen data. The training process ran for 50 epochs and the batch size was set to 16. In other words, the model will iterate over the training dataset for 50 times and update its weights after processing every 16 samples. After training, the model's performance is evaluated on the test set ('X_test' and 'y_test') and we used 'test_loss' and 'tess_acc' to store the loss and accuracy on the test set.



```python
import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

Diagram 5.2.2

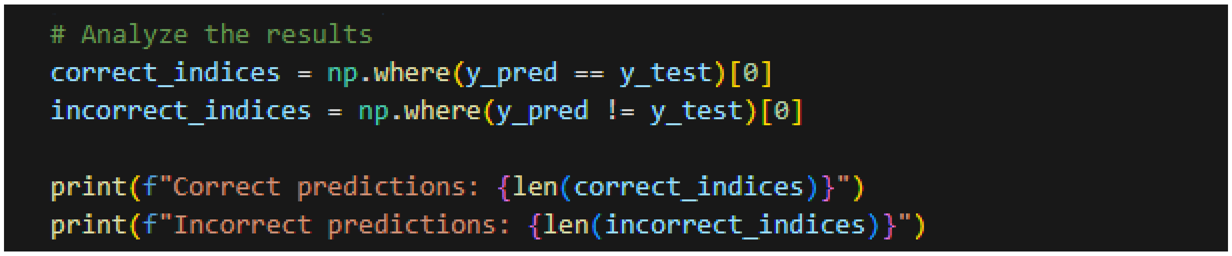
To visualize the accuracy and loss values, we plotted the graphs of model accuracy and model loss using matplotlib.pyplot library with both training and validation data.



```
# Generate predictions on the test set
y_pred = np.argmax(model.predict(X_test), axis=1)

# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=class_labels))

# Construct confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Visualize Confusion Matrix with cell-wise labels
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, cmap='tab20', fmt='d', xticklabels=class_labels, yticklabels=class_labels)

# Add cell-wise labels
for i in range(len(cm)):
    for j in range(len(cm[0])):
        plt.text(j + 0.5, i + 0.5, cm[i, j], ha='center', va='center', color='Black')

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

Diagram 5.2.3

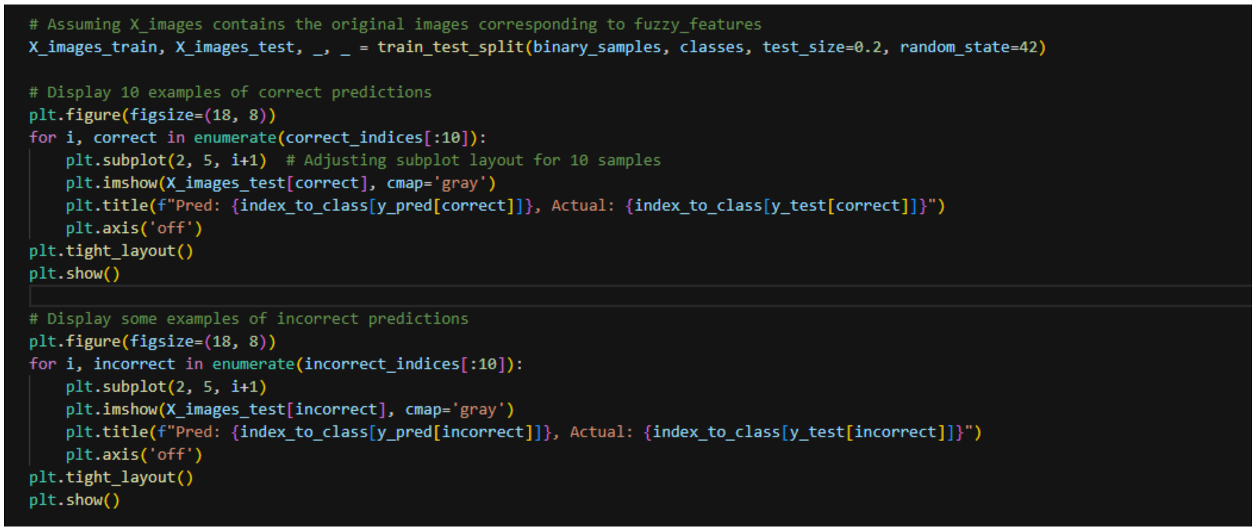
Next, we predict on the test set using a trained neural network model and prints a classification report showing various evaluation metrics such as precision, recall, and F1-score for each class label. Additionally, we constructs a confusion matrix to visualize the performance of the model in classifying different classes. The confusion matrix provides a detailed breakdown of the predicted versus actual class labels, facilitating an understanding of the model's strengths and weaknesses in classification tasks. By visualizing the confusion matrix with cell-wise labels, it becomes easier to interpret and analyze the model's performance across different classes, aiding in fine-tuning and improving the model's accuracy.



```
# Analyze the results
correct_indices = np.where(y_pred == y_test)[0]
incorrect_indices = np.where(y_pred != y_test)[0]

print(f"Correct predictions: {len(correct_indices)}")
print(f"Incorrect predictions: {len(incorrect_indices)}")
```

Diagram 5.2.4

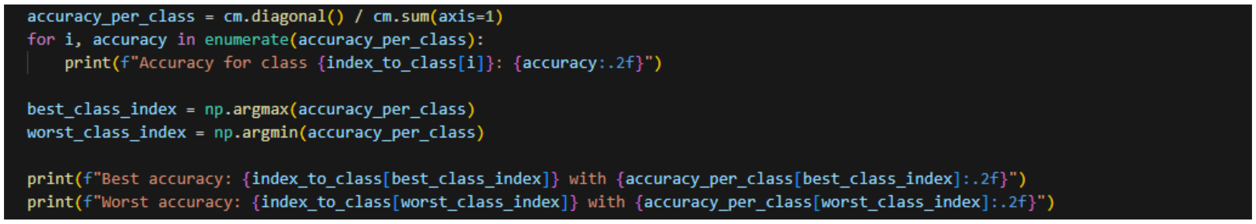
In this analysis, we examine the results of our model predictions on the test set. We segregate the indices of correct predictions and incorrect predictions to assess the overall performance of our model. By counting the number of correct and incorrect predictions, we gain insights into our model's accuracy. Correct predictions signify instances where our model accurately classified the handwritten samples, while incorrect predictions indicate misclassifications. This evaluation allows us to gauge our model's effectiveness and identify potential areas for improvement.



```python
# Assuming X_images contains the original images corresponding to fuzzy_features
X_images_train, X_images_test, _, _ = train_test_split(binary_samples, classes, test_size=0.2, random_state=42)

# Display 10 examples of correct predictions
plt.figure(figsize=(18, 8))
for i, correct in enumerate(correct_indices[:10]):
    plt.subplot(2, 5, i+1)  # Adjusting subplot layout for 10 samples
    plt.imshow(X_images_test[correct], cmap='gray')
    plt.title(f"Pred: {index_to_class[y_pred[correct]]}, Actual: {index_to_class[y_test[correct]]}")
    plt.axis('off')
plt.tight_layout()
plt.show()

# Display some examples of incorrect predictions
plt.figure(figsize=(18, 8))
for i, incorrect in enumerate(incorrect_indices[:10]):
    plt.subplot(2, 5, i+1)
    plt.imshow(X_images_test[incorrect], cmap='gray')
    plt.title(f"Pred: {index_to_class[y_pred[incorrect]]}, Actual: {index_to_class[y_test[incorrect]]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Diagram 5.2.5

We split the original images into training and testing sets, aligning them with the fuzzy_features and their corresponding classes. This division ensures that our visualization accurately reflects the performance of the model on unseen data. Subsequently, we display 10 examples of correct predictions made by our model on the test set. Each example showcases an image alongside its predicted and actual class labels. Additionally, we present some examples of incorrect predictions, highlighting instances where our model misclassified the handwritten samples. These visualizations offer valuable insights into the model's performance and aid in understanding its strengths and weaknesses.



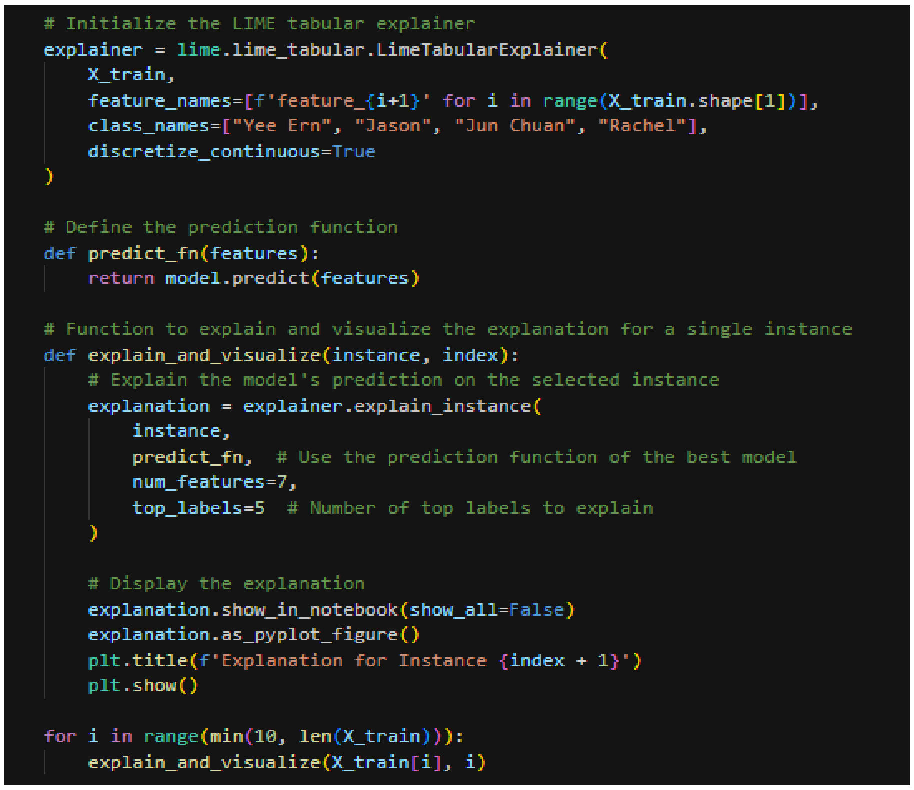
```python
accuracy_per_class = cm.diagonal() / cm.sum(axis=1)
for i, accuracy in enumerate(accuracy_per_class):
    print(f"Accuracy for class {index_to_class[i]}: {accuracy:.2f}")

best_class_index = np.argmax(accuracy_per_class)
worst_class_index = np.argmin(accuracy_per_class)

print(f"Best accuracy: {index_to_class[best_class_index]} with {accuracy_per_class[best_class_index]:.2f}")
print(f"Worst accuracy: {index_to_class[worst_class_index]} with {accuracy_per_class[worst_class_index]:.2f}")
```

Diagram 5.2.6

We calculate the accuracy for each class based on the confusion matrix. The accuracy per class is computed by dividing the diagonal elements of the confusion matrix (representing correct predictions for each class) by the sum of the corresponding row (total instances of that class). We then iterate through each class and print its accuracy. Additionally, we identify the class with the highest accuracy and the class with the lowest accuracy, along with their respective accuracy values. This analysis provides insights into the model's performance across different classes, highlighting which classes it performs well on and which classes it struggles with.

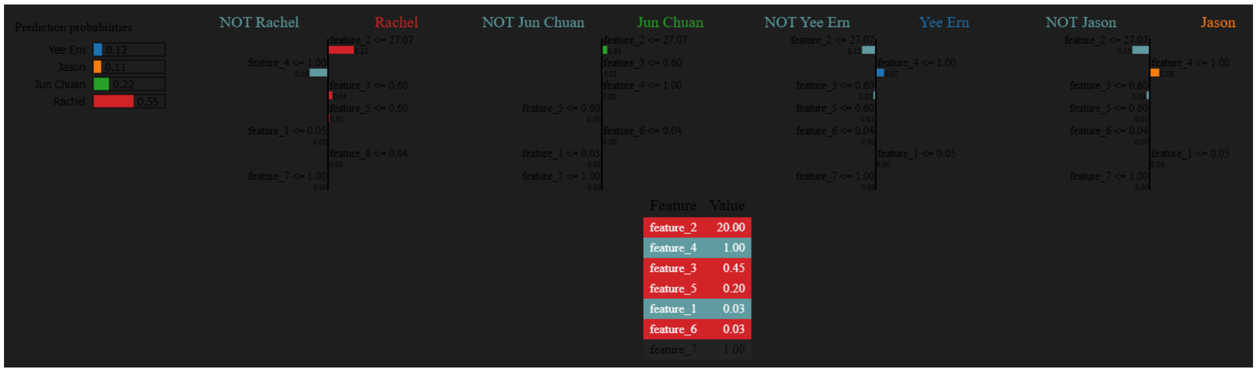
5.3 Model Information Visualization

5.3.1 Model 1: LIME



```python
# Initialize the LIME tabular explainer
explainer = lime.lime_tabular.LimeTabularExplainer(
    X_train,
    feature_names=[f'feature_{i+1}' for i in range(X_train.shape[1])],
    class_names=["Yee Ern", "Jason", "Jun Chuan", "Rachel"],
    discretize_continuous=True
)

# Define the prediction function
def predict_fn(features):
    return model.predict(features)

# Function to explain and visualize the explanation for a single instance
def explain_and_visualize(instance, index):
    # Explain the model's prediction on the selected instance
    explanation = explainer.explain_instance(
        instance,
        predict_fn,  # Use the prediction function of the best model
        num_features=7,
        top_labels=5  # Number of top labels to explain
    )

    # Display the explanation
    explanation.show_in_notebook(show_all=False)
    explanation.as_pyplot_figure()
    plt.title(f'Explanation for Instance {index + 1}')
    plt.show()

for i in range(min(10, len(X_train))):
    explain_and_visualize(X_train[i], i)
```

Diagram 5.3.1.1



Feature	Value
feature_2	20.00
feature_4	1.00
feature_3	0.45
feature_5	0.20
feature_1	0.03
feature_6	0.03
feature_7	1.00

Diagram 5.3.1.2

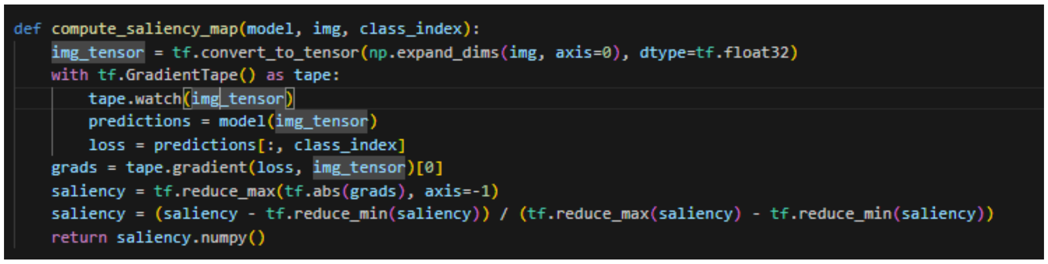
The provided code initializes the LIME (*notebook.community*, n.d.), also known as Local Interpretable Model-agnostic Explanations tabular explainer, which is utilized to provide explanations for individual predictions made by the model. In this setup, the explainer is configured with the training data (X_train), feature names, class names (e.g., "Yee Ern", "Jason", "Jun Chuan", "Rachel"), and the option to discretize continuous features.

Additionally, a prediction function (predict_fn) is defined, which takes features as input and returns the model's predictions. This function is essential for LIME to generate explanations based on the model's behavior.

The explain_and_visualize function is then defined to explain and visualize the explanation for a single instance. Inside this function, LIME's explain_instance method is used to generate an explanation for the selected instance. The explanation is then displayed using the show_in_notebook method, and a pyplot figure is created to visualize the explanation.

Finally, for the first ten instances in the training data (or fewer if the dataset is smaller than ten instances), explanations are generated and visualized using the explain_and_visualize function. Each explanation provides insights into the model's decision-making process for the corresponding instance, aiding in the interpretation and understanding of the model's behavior.

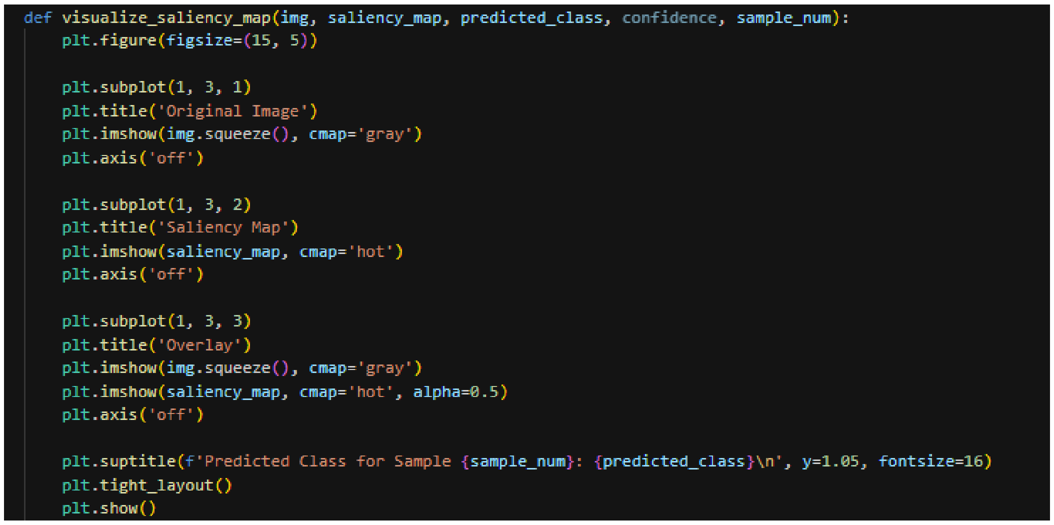
5.3.2 Model 2: Saliency Map



```python
def compute_saliency_map(model, img, class_index):
    img_tensor = tf.convert_to_tensor(np.expand_dims(img, axis=0), dtype=tf.float32)
    with tf.GradientTape() as tape:
        tape.watch(img_tensor)
        predictions = model(img_tensor)
        loss = predictions[:, class_index]
    grads = tape.gradient(loss, img_tensor)[0]
    saliency = tf.reduce_max(tf.abs(grads), axis=-1)
    saliency = (saliency - tf.reduce_min(saliency)) / (tf.reduce_max(saliency) - tf.reduce_min(saliency))
    return saliency.numpy()
```

Diagram 5.3.2.1

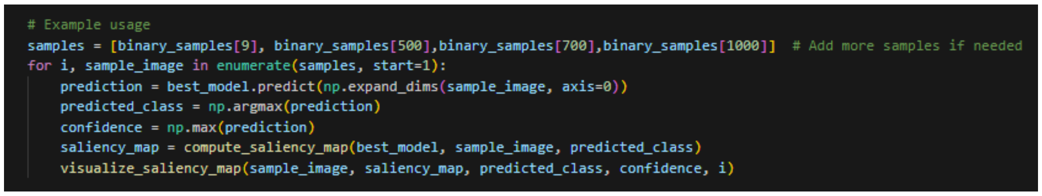
This code is designed to compute and visualize saliency maps for given image samples using a trained neural network model. Saliency maps help in understanding which parts of an input image are most influential in the model's prediction. The process starts by defining a function compute_saliency_map which takes the model, an image, and the target class index as inputs. This function converts the image into a tensor, computes the gradients of the predicted class score with respect to the input image, and then derives the saliency map by taking the maximum absolute gradient across the color channels. The saliency map is normalized to a range of [0, 1] to enhance visualization.



```python
def visualize_saliency_map(img, saliency_map, predicted_class, confidence, sample_num):
    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)
    plt.title('Original Image')
    plt.imshow(img.squeeze(), cmap='gray')
    plt.axis('off')

    plt.subplot(1, 3, 2)
    plt.title('Saliency Map')
    plt.imshow(saliency_map, cmap='hot')
    plt.axis('off')

    plt.subplot(1, 3, 3)
    plt.title('Overlay')
    plt.imshow(img.squeeze(), cmap='gray')
    plt.imshow(saliency_map, cmap='hot', alpha=0.5)
    plt.axis('off')

    plt.suptitle(f'Predicted Class for Sample {sample_num}: {predicted_class}\n', y=1.05, fontsize=16)
    plt.tight_layout()
    plt.show()
```

Diagram 5.3.2.2

Another function, visualize_saliency_map, is defined to display the original image, the saliency map, and an overlay of both. This function creates a figure with three subplots: the original image in grayscale, the saliency map in a 'hot' colormap, and an overlay of the saliency map on the original image. A title is added to indicate the predicted class and the sample number.



```python
# Example usage
samples = [binary_samples[9], binary_samples[500],binary_samples[700],binary_samples[1000]]  # Add more samples if needed
for i, sample_image in enumerate(samples, start=1):
    prediction = best_model.predict(np.expand_dims(sample_image, axis=0))
    predicted_class = np.argmax(prediction)
    confidence = np.max(prediction)
    saliency_map = compute_saliency_map(best_model, sample_image, predicted_class)
    visualize_saliency_map(sample_image, saliency_map, predicted_class, confidence, i)
```

Diagram 5.3.2.3

In the example usage section, a few image samples from binary samples are selected. For each sample, the model predicts the class probabilities, identifies the predicted class, and computes the saliency map. The results are then visualized, showing the original image, the saliency map, and their overlay. This process provides insights into the areas of the image that the model focuses on for making its predictions, thereby offering an intuitive understanding of the model's decision-making process.

6. Evolutionary Algorithm

We employ evolutionary algorithms to optimize the performance of the neural network by tuning its parameters.

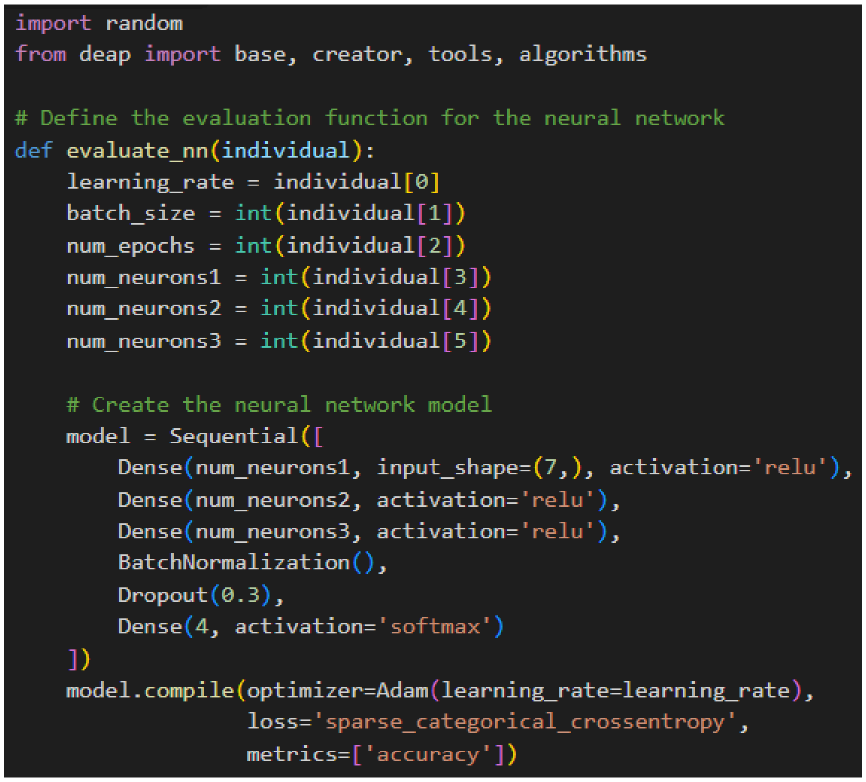
Evolutionary Algorithm 1: Optimizing Parameters in DNN

This evolutionary algorithm focuses on optimizing the parameters of the Dense Neural Network (DNN). By leveraging genetic algorithms, we iteratively evolve a population of potential solutions, with each solution representing a set of DNN parameters. Through a process of selection, crossover, and mutation, the algorithm explores the parameter space to find configurations that maximize the performance of the DNN in classifying handwriting samples. During each generation, individuals with higher fitness scores, indicating better performance on a validation dataset, are more likely to be selected for reproduction. This iterative process continues until convergence or a predefined stopping criterion is met, resulting in a set of optimized parameters for the DNN model.

Evolutionary Algorithm 2: Optimizing Parameters in CNN

Similarly, this evolutionary algorithm targets the optimization of parameters in the Convolutional Neural Network (CNN). Genetic algorithms are utilized to evolve a population of CNN parameter configurations, with the goal of maximizing the network's performance in classifying fuzzy-enhanced binary images. The algorithm iteratively explores the parameter space, with individuals representing different sets of CNN parameters. Through the process of selection, crossover, and mutation, the algorithm identifies and refines parameter configurations that lead to improved classification accuracy. Fitness evaluation is based on the performance of the CNN model on a validation dataset, with individuals exhibiting superior performance being prioritized for reproduction in subsequent generations. This iterative optimization process continues until satisfactory performance is achieved or a convergence criterion is met, yielding optimized parameters for the CNN model.

6.1 Evolutionary Algorithm 1

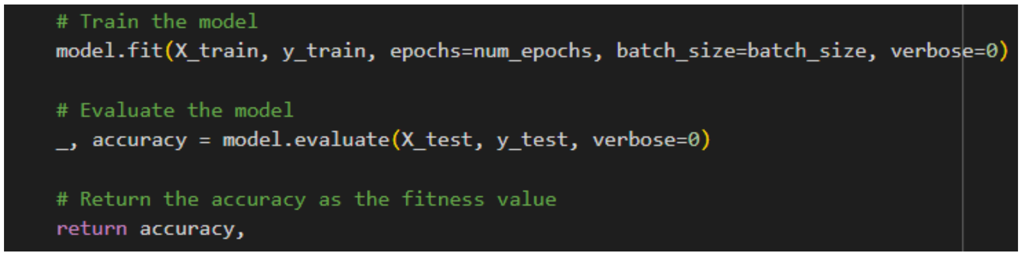


```python
import random
from deap import base, creator, tools, algorithms

# Define the evaluation function for the neural network
def evaluate_nn(individual):
    learning_rate = individual[0]
    batch_size = int(individual[1])
    num_epochs = int(individual[2])
    num_neurons1 = int(individual[3])
    num_neurons2 = int(individual[4])
    num_neurons3 = int(individual[5])

    # Create the neural network model
    model = Sequential([
        Dense(num_neurons1, input_shape=(7,), activation='relu'),
        Dense(num_neurons2, activation='relu'),
        Dense(num_neurons3, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),
        Dense(4, activation='softmax')
    ])
    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
```

Diagram 6.1.1

The neural network is built, trained and evaluated based on hyperparameters provided by an evolutionary algorithm using DEAP.

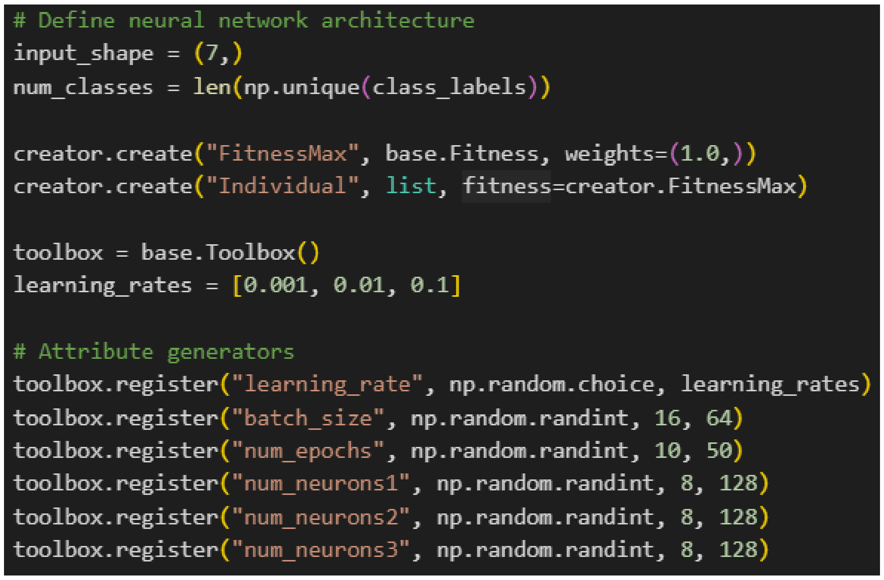
We first defined the evaluate_nn function by extracting hyperparameters from the 'individual' list: learning rate, batch size, number of epochs, and the number of neurons in three dense layers. We then created a neural network model similar to the structure of neural network approach 1 where a Sequential model with three dense layers, batch normalization, dropout and an output layer with softmax activation is built. The model is compiled using the Adam optimizer with a specified learning rate and sparse categorical cross-entropy loss.



```python
    # Train the model
    model.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, verbose=0)

    # Evaluate the model
    _, accuracy = model.evaluate(X_test, y_test, verbose=0)

    # Return the accuracy as the fitness value
    return accuracy,
```

Diagram 6.1.2

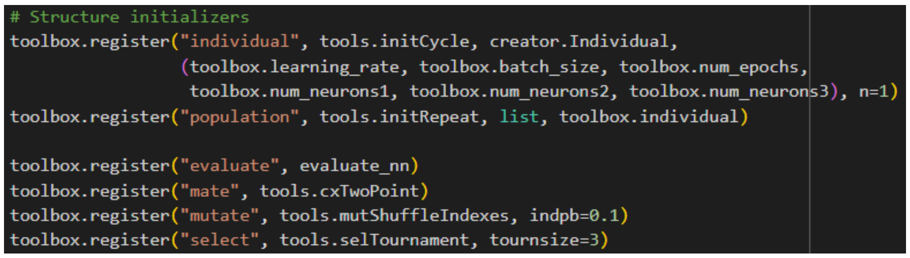
The model is trained on the training data ('X_train' and 'y_train') using the predefined number of epochs and batch size with the verbose set to 0 to suppress output. The model is then evaluated on the test data (‘X_test’, ‘y_test’) to obtain the accuracy. The function lastly returned the accuracy which serves as the fitness value for the genetic algorithm.



```
# Define neural network architecture
input_shape = (7,)
num_classes = len(np.unique(class_labels))

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
learning_rates = [0.001, 0.01, 0.1]

# Attribute generators
toolbox.register("learning_rate", np.random.choice, learning_rates)
toolbox.register("batch_size", np.random.randint, 16, 64)
toolbox.register("num_epochs", np.random.randint, 10, 50)
toolbox.register("num_neurons1", np.random.randint, 8, 128)
toolbox.register("num_neurons2", np.random.randint, 8, 128)
toolbox.register("num_neurons3", np.random.randint, 8, 128)
```

Diagram 6.1.3

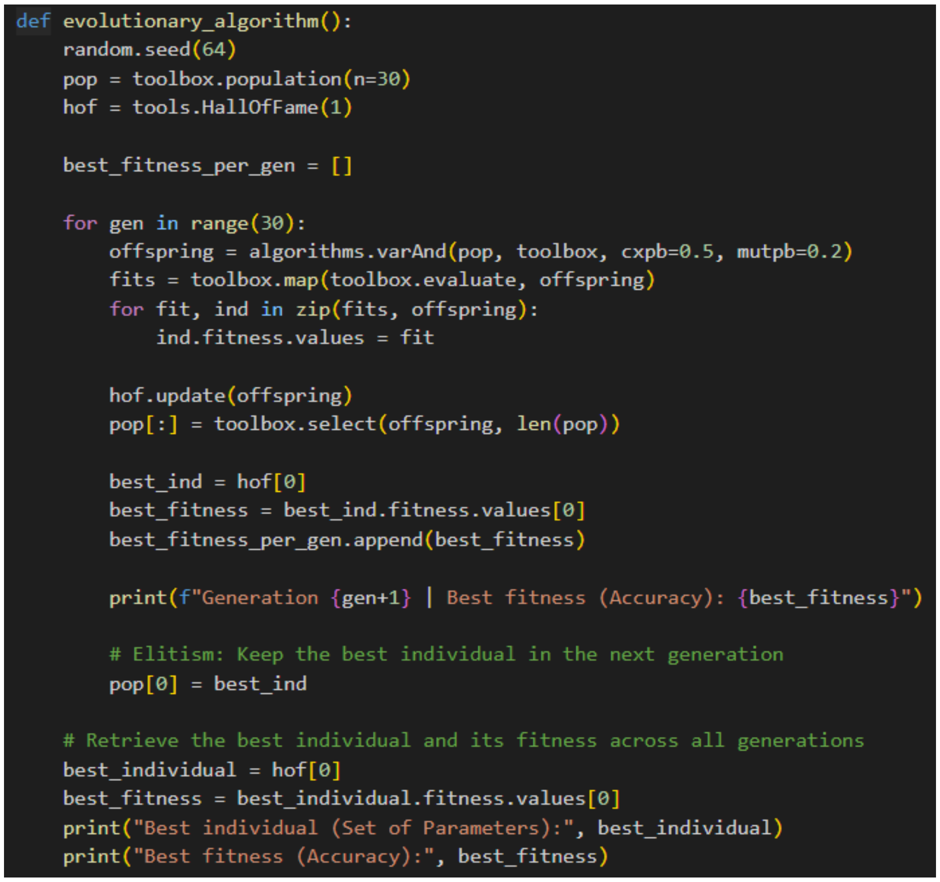
We first defined the neural network parameters by specifying the input shape as 7 input features and calculated the number of unique classes in ‘class_labels’ which are stored in the variable ‘num_classes’. Next, we continued with DEAP configuration where we utilized the ‘creator.create()’ method to define a fitness class to maximize accuracy and an individual as a list with a fitness attribute. We set up the toolbox to store the evolutionary algorithm’s operators and declare a list of possible learning rates (0.001, 0.01, 0.1) for the neural network. We then registered functions using the ‘toolbox.register()’ method to generate values for each hyperparameter aforementioned with specified weights and ranges.



```
# Structure initializers
toolbox.register("individual", tools.initCycle, creator.Individual,
                 (toolbox.learning_rate, toolbox.batch_size, toolbox.num_epochs,
                  toolbox.num_neurons1, toolbox.num_neurons2, toolbox.num_neurons3), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("evaluate", evaluate_nn)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=3)
```

Diagram 6.1.4

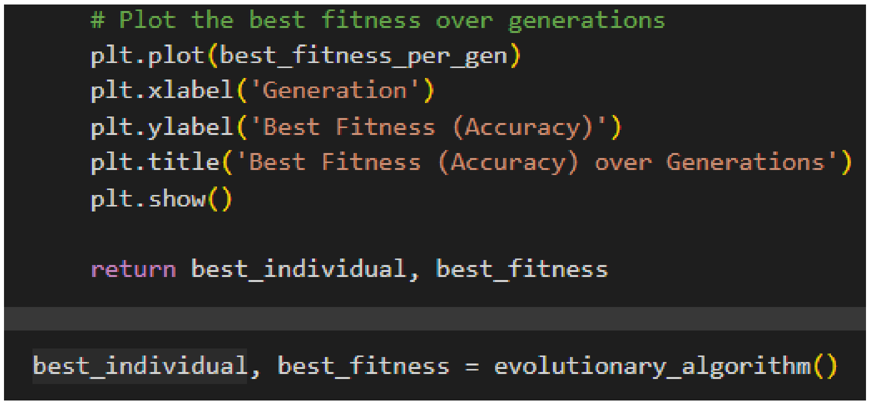
In this section, we initialized the individual by cycling through the provided attribute generators aforementioned by using ‘tools.initCycle’ and ‘creator.Individual’ and initialized the population by repeating the individual creation process using ‘tools.initRepeat’, creating ‘list’ as the type of container to store the population and ‘toolbox.individual’ function to generate an individual. We dived into the evolutionary algorithm components where we used ‘evaluate_nn’, a previously defined function that evaluates the fitness of an individual; crossover operator that performed two-point crossover on two individuals using ‘tools.cxTwoPoint’; mutation operator that shuffles the indices of an individual’s attributes using ‘tools.mutShuffleIndexes’; and, a selection operator that selects individuals based on tournament selection using ‘tools.selTournament’.



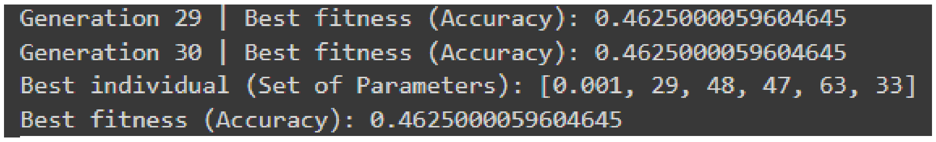
```python
def evolutionary_algorithm():
    random.seed(64)
    pop = toolbox.population(n=30)
    hof = tools.HallOfFame(1)

    best_fitness_per_gen = []

    for gen in range(30):
        offspring = algorithms.varAnd(pop, toolbox, cxpb=0.5, mutpb=0.2)
        fits = toolbox.map(toolbox.evaluate, offspring)
        for fit, ind in zip(fits, offspring):
            ind.fitness.values = fit

        hof.update(offspring)
        pop[:] = toolbox.select(offspring, len(pop))

        best_ind = hof[0]
        best_fitness = best_ind.fitness.values[0]
        best_fitness_per_gen.append(best_fitness)

        print(f"Generation {gen+1} | Best fitness (Accuracy): {best_fitness}")

        # Elitism: Keep the best individual in the next generation
        pop[0] = best_ind

    # Retrieve the best individual and its fitness across all generations
    best_individual = hof[0]
    best_fitness = best_individual.fitness.values[0]
    print("Best individual (Set of Parameters):", best_individual)
    print("Best fitness (Accuracy):", best_fitness)
```

Diagram 6.1.5

We created an evolutionary_algorithm function that set the random seed of 64 for reproducibility, initialized a population of 30 individuals and created a Hall of Fame to keep track of the best individual found. We initialized a list named ‘best_fitness_per_gen’ to keep track of the best fitness values per generation. We ran the loop for 30 generations where we generated offspring using crossover of 0.5 probability and mutation of 0.2 probability and evaluated the fitness of the offspring. Within the loop, we assigned the calculated fitness values, updated the Hall of Fame with the best individual, selected the next generation from the offspring before tracking the best individual and used elitism to ensure the best individual is carried over to the next generation. At the end, we retrieved and printed the best individual and its fitness from the Hall of Fame after all generations.



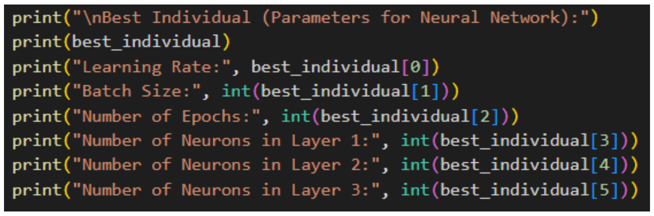
```
    # Plot the best fitness over generations
    plt.plot(best_fitness_per_gen)
    plt.xlabel('Generation')
    plt.ylabel('Best Fitness (Accuracy)')
    plt.title('Best Fitness (Accuracy) over Generations')
    plt.show()

    return best_individual, best_fitness

best_individual, best_fitness = evolutionary_algorithm()
```



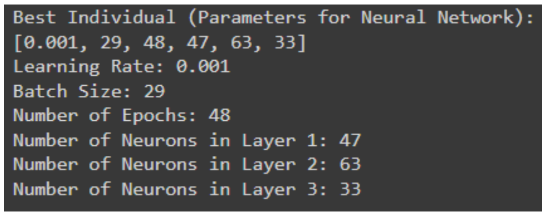
```
Generation 29 | Best fitness (Accuracy): 0.4625000059604645
Generation 30 | Best fitness (Accuracy): 0.4625000059604645
Best individual (Set of Parameters): [0.001, 29, 48, 47, 63, 33]
Best fitness (Accuracy): 0.4625000059604645
```

Diagram 6.1.6

We then plotted and displayed the visualization of the best fitness (in terms of accuracy) over generations, in which the function lastly returned the best individual and its fitness found. Lastly, we called the function ‘evolutionary_algorithm()’ to perform the evolutionary optimization while storing the best set of hyperparameters found and the best fitness (accuracy) associated with the best individual.



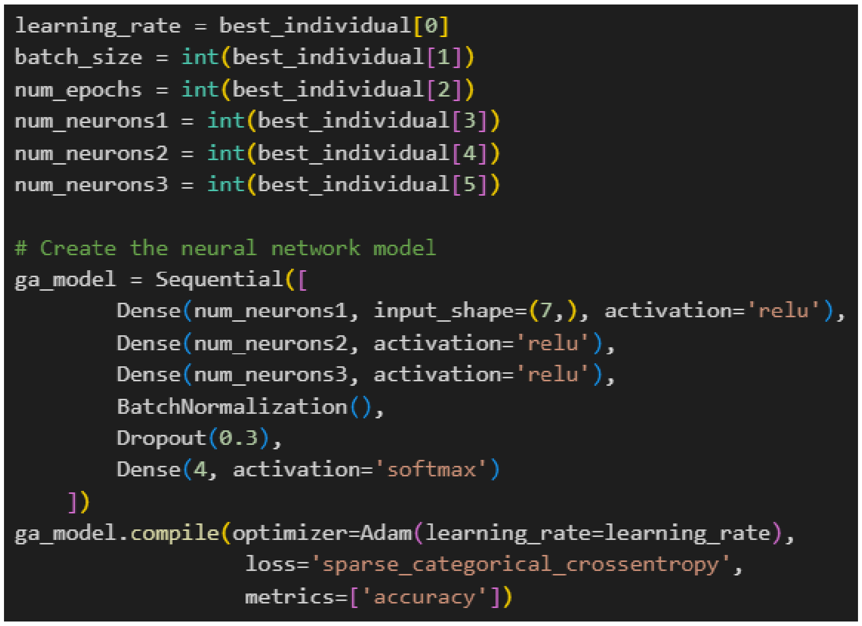
```
print("\nBest Individual (Parameters for Neural Network):")
print(best_individual)
print("Learning Rate:", best_individual[0])
print("Batch Size:", int(best_individual[1]))
print("Number of Epochs:", int(best_individual[2]))
print("Number of Neurons in Layer 1:", int(best_individual[3]))
print("Number of Neurons in Layer 2:", int(best_individual[4]))
print("Number of Neurons in Layer 3:", int(best_individual[5]))
```



```
Best Individual (Parameters for Neural Network):
[0.001, 29, 48, 47, 63, 33]
Learning Rate: 0.001
Batch Size: 29
Number of Epochs: 48
Number of Neurons in Layer 1: 47
Number of Neurons in Layer 2: 63
Number of Neurons in Layer 3: 33
```

Diagram 6.1.7

We printed out the best individual and its fitness details such as its learning rate, batch size, number of epochs and the number of neurons in each layer.



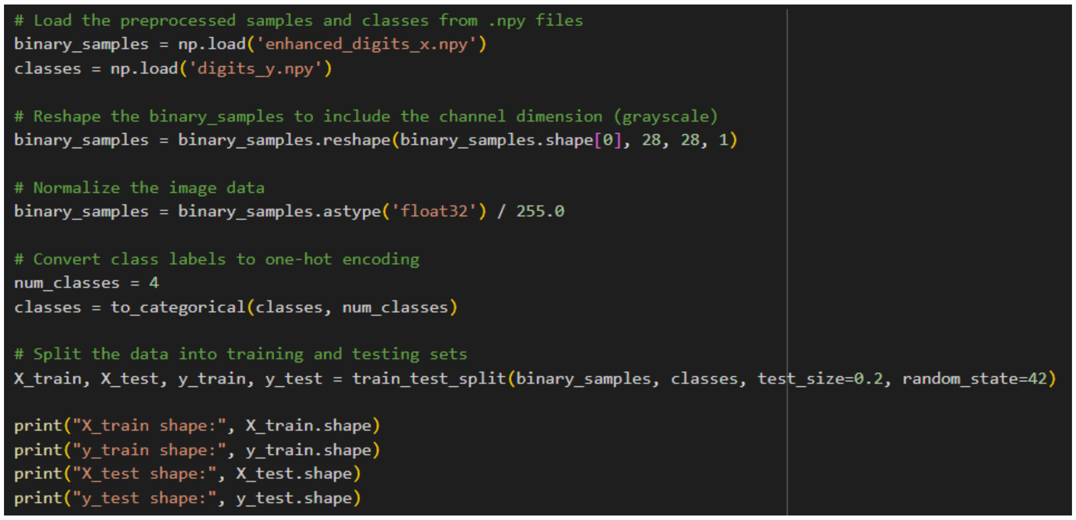
```python
learning_rate = best_individual[0]
batch_size = int(best_individual[1])
num_epochs = int(best_individual[2])
num_neurons1 = int(best_individual[3])
num_neurons2 = int(best_individual[4])
num_neurons3 = int(best_individual[5])

# Create the neural network model
ga_model = Sequential([
        Dense(num_neurons1, input_shape=(7,), activation='relu'),
        Dense(num_neurons2, activation='relu'),
        Dense(num_neurons3, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),
        Dense(4, activation='softmax')
    ])
ga_model.compile(optimizer=Adam(learning_rate=learning_rate),
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])
```

Diagram 6.1.8

After the implementation of the genetic algorithm, we tried building a neural network model using the best hyperparameters identified by the evolutionary algorithm based on the results we obtained previously. We first extracted the best hyperparameters: the optimal fitness for 'learning_rate', 'batch_size', 'num_epochs', 'num_neurons1', 'num_neurons2' and 'num_neurons3'. We created a neural network sequential model with three dense layers implementing ReLU activation function and 7 features for each input sample. Similar to our approach in the neural network, we added a batch normalization layer, a dropout layer with a dropout rate of 0.3 and an output layer with 4 neurons implementing softmax activation. We compiled the model using Adam optimizer and sparse categorical cross-entropy as loss function while monitoring the accuracy during training and evaluation.

As we progressed to plotting, visualizing and testing the model accuracy and model loss, we found out the result analyzed was similar to the approach of a neural network without implementing an evolutionary algorithm, or to be exact, genetic algorithm as discussed previously.

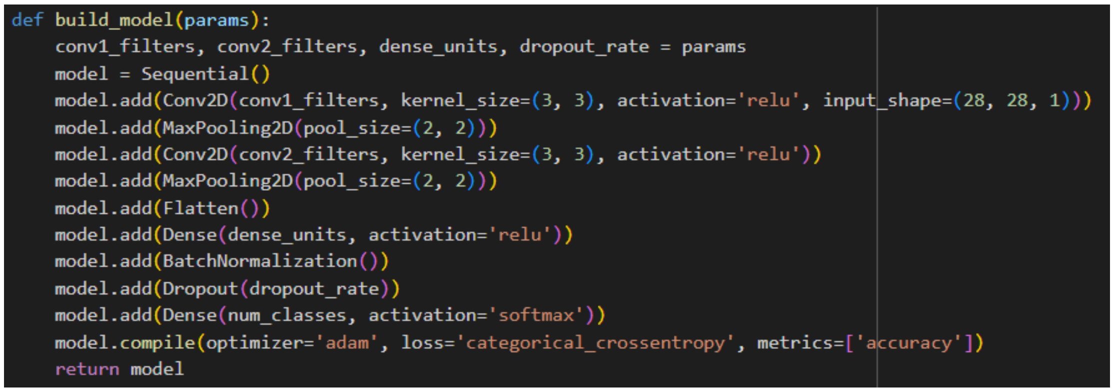
6.2 Evolutionary Algorithm 2



```
# Load the preprocessed samples and classes from .npy files
binary_samples = np.load('enhanced_digits_x.npy')
classes = np.load('digits_y.npy')

# Reshape the binary_samples to include the channel dimension (grayscale)
binary_samples = binary_samples.reshape(binary_samples.shape[0], 28, 28, 1)

# Normalize the image data
binary_samples = binary_samples.astype('float32') / 255.0

# Convert class labels to one-hot encoding
num_classes = 4
classes = to_categorical(classes, num_classes)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(binary_samples, classes, test_size=0.2, random_state=42)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

Diagram 6.2.1

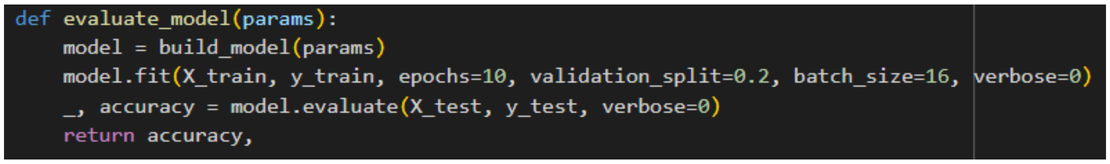
Before we begin the Evolutionary Algorithm, we started off by loading the enhanced images that we obtained earlier from the fuzzy system , together with their corresponding class labels. Specifically, binary_samples contain the image data of digits, and classes contain the associated labels. Next, the images in binary_samples are reshaped to include a channel dimension, converting them into 4D arrays suitable for input into a convolutional neural network (CNN). This step is particularly necessary because the images are grayscale, and the CNN expects an input shape of (height, width, channels). The images are then normalized by converting their data type to float32 and scaling the pixel values to a range between 0 and 1, which helps in faster and more efficient training of the neural network. Following this, the class labels are converted to one-hot encoded vectors using the to_categorical function, preparing them for classification tasks where the model predicts probabilities for each class. With a specified num_classes of 4, each label is transformed into a vector of length 4. Finally, the dataset is split into training and testing sets using the train_test_split function, with 80% of the data allocated for training and 20% for testing. The random_state parameter ensures reproducibility of the split. The shapes of the resulting arrays (X_train, X_test, y_train, y_test) are printed to confirm the successful execution of these preprocessing steps.



```
def build_model(params):
    conv1_filters, conv2_filters, dense_units, dropout_rate = params
    model = Sequential()
    model.add(Conv2D(conv1_filters, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(conv2_filters, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(dense_units, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Diagram 6.2.2

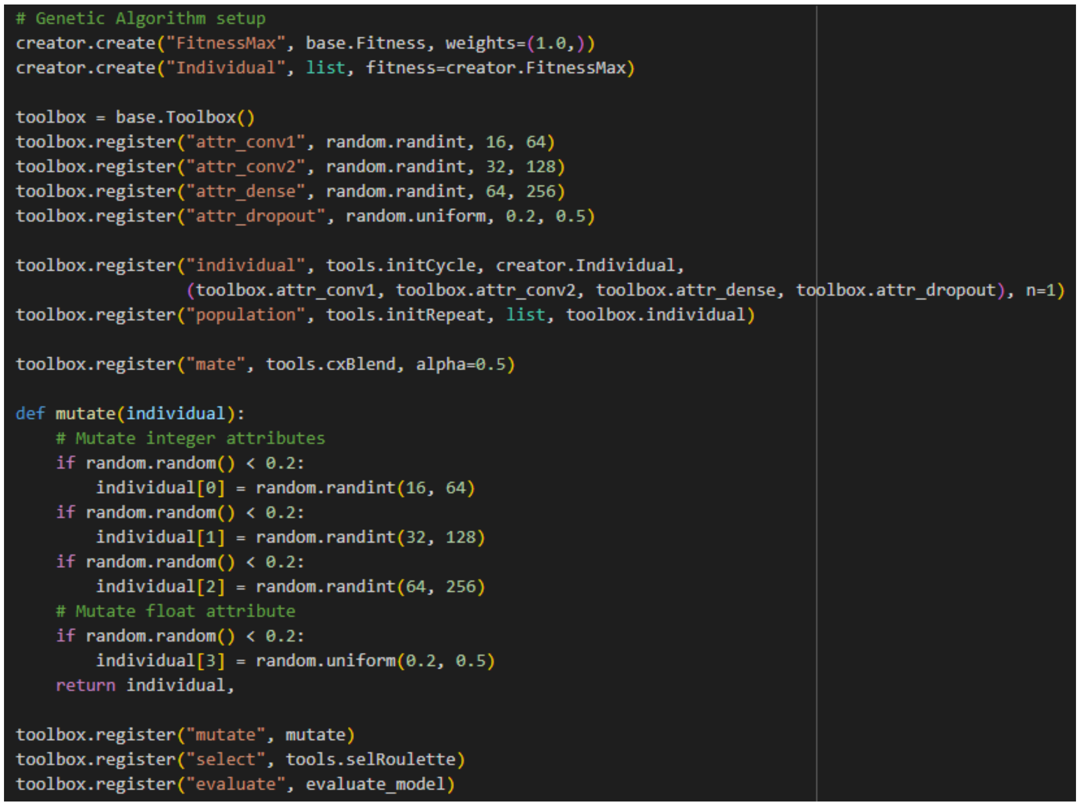
Then, we define two functions, build_model and evaluate_model, which are used to create and assess a convolutional neural network (CNN) in a genetic algorithm framework. The build_model function constructs a CNN based on parameters for the number of filters in the convolutional layers (conv1_filters and conv2_filters), the number of units in the dense layer (dense_units), and the dropout rate (dropout_rate). The model includes convolutional and max-pooling layers, followed by flattening, a dense layer with ReLU activation, batch normalization, dropout, and a softmax output layer for classification. It is compiled with the Adam optimizer and categorical cross entropy loss.



```python
def evaluate_model(params):
    model = build_model(params)
    model.fit(X_train, y_train, epochs=10, validation_split=0.2, batch_size=16, verbose=0)
    _, accuracy = model.evaluate(X_test, y_test, verbose=0)
    return accuracy,
```

Diagram 6.2.3

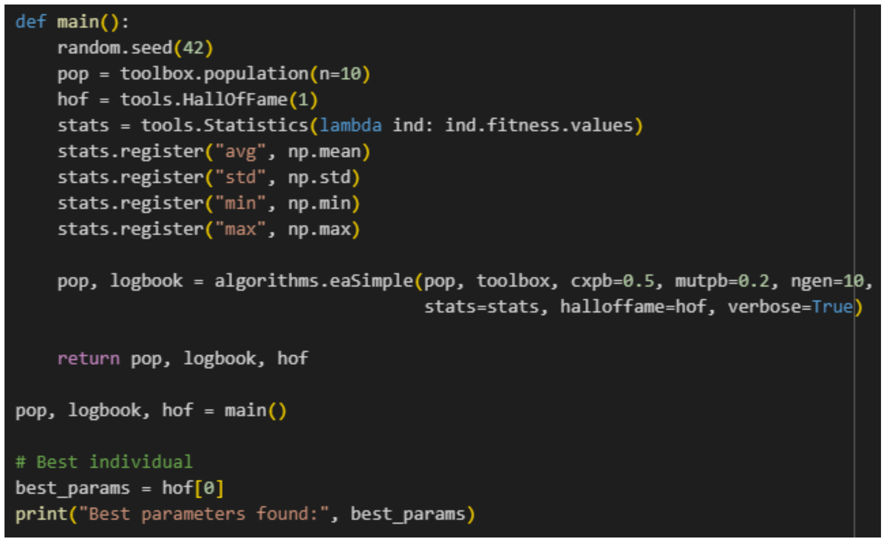
The evaluate_model function acts as a fitness function which builds the model with build_model, trains it on the training data (X_train and y_train) for 10 epochs, and evaluates its accuracy on the test data (X_test and y_test). This setup allows for the automated optimization of model hyperparameters based on accuracy within the genetic algorithm.



```python
# Genetic Algorithm setup
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_conv1", random.randint, 16, 64)
toolbox.register("attr_conv2", random.randint, 32, 128)
toolbox.register("attr_dense", random.randint, 64, 256)
toolbox.register("attr_dropout", random.uniform, 0.2, 0.5)

toolbox.register("individual", tools.initCycle, creator.Individual,
                 (toolbox.attr_conv1, toolbox.attr_conv2, toolbox.attr_dense, toolbox.attr_dropout), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("mate", tools.cxBlend, alpha=0.5)

def mutate(individual):
    # Mutate integer attributes
    if random.random() < 0.2:
        individual[0] = random.randint(16, 64)
    if random.random() < 0.2:
        individual[1] = random.randint(32, 128)
    if random.random() < 0.2:
        individual[2] = random.randint(64, 256)
    # Mutate float attribute
    if random.random() < 0.2:
        individual[3] = random.uniform(0.2, 0.5)
    return individual,

toolbox.register("mutate", mutate)
toolbox.register("select", tools.selRoulette)
toolbox.register("evaluate", evaluate_model)
```

Diagram 6.2.4

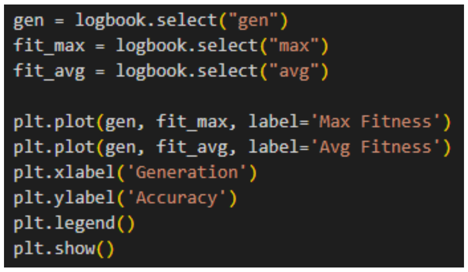
The code shows the setup of the genetic algorithm to optimize CNN hyperparameters. It begins by defining a maximization fitness function FitnessMax and an Individual class inheriting from a list. The toolbox is configured to generate hyperparameters: attr_conv1 (16-64 filters), attr_conv2 (32-128 filters), attr_dense (64-256 units), and attr_dropout (0.2-0.5 dropout rate). Individuals are created using these attributes, and populations are initialized from these individuals. Crossover (mate) uses the blend method (cxBlend). The mutate function randomly changes hyperparameters with a 20% probability for each. Selection uses roulette wheel (selRoulette), and evaluate is linked to the evaluate_model function. This setup allows the genetic algorithm to optimize hyperparameters by evolving them over generations to maximize model accuracy.



```python
def main():
    random.seed(42)
    pop = toolbox.population(n=10)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)

    pop, logbook = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=10,
                                       stats=stats, halloffame=hof, verbose=True)

    return pop, logbook, hof

pop, logbook, hof = main()

# Best individual
best_params = hof[0]
print("Best parameters found:", best_params)
```

Diagram 6.2.5

The main function sets up and runs the genetic algorithm. It starts by setting a random seed for reproducibility and initializes a population of 10 individuals using toolbox.population. A HallOfFame object (hof) is created to store the best individual. Statistics are collected using a Statistics object, which tracks the average, standard deviation, minimum, and maximum fitness values. The eaSimple algorithm is then used to evolve the population over 10 generations (ngen=10) with crossover (cxpb=0.5) and mutation (mutpb=0.2) probabilities. The evolution process outputs the final population (pop), a logbook of statistics (logbook), and the best individual found (hof). The best hyperparameters are printed from the HallOfFame.



```python
gen = logbook.select("gen")
fit_max = logbook.select("max")
fit_avg = logbook.select("avg")

plt.plot(gen, fit_max, label='Max Fitness')
plt.plot(gen, fit_avg, label='Avg Fitness')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Diagram 6.2.6

Then we plot the values to visualize the evolution of fitness over generations. The x-axis represents generations, while the y-axis represents accuracy. Two lines are plotted: one for maximum fitness and one for average fitness.

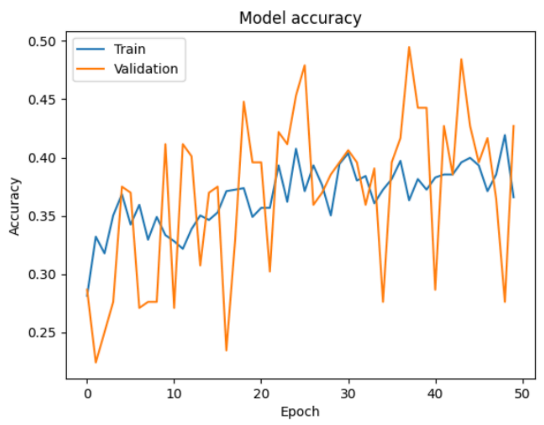
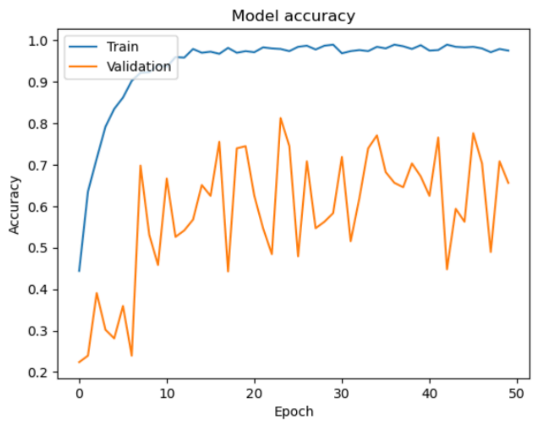
After implementing the genetic algorithm, we utilized the optimal hyperparameters identified by the evolutionary algorithm for 'conv1_filters', 'conv2_filters', 'dense_units', and 'dropout_rate'. These parameters guided the creation of a convolutional neural network (CNN) model with two convolutional layers, each employing the ReLU activation function. The convolutional layers were followed by max-pooling layers to reduce spatial dimensions. The model also included a dense layer with ReLU activation, batch normalization, dropout with the optimal rate, and an output layer with 4 neurons utilizing softmax activation. During compilation, we employed the Adam optimizer and used categorical cross-entropy as the loss function, monitoring accuracy throughout the training and evaluation stages.

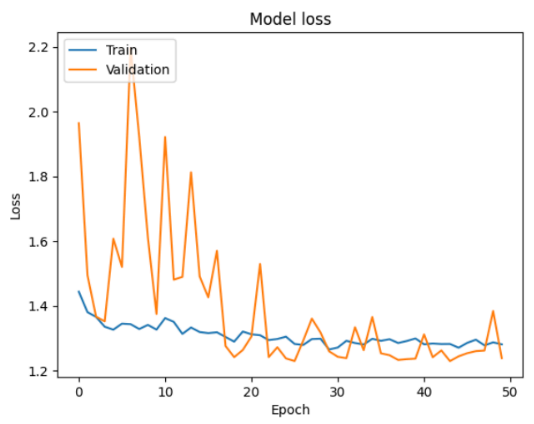
7. Results and Discussion

Set 1	**Set 2**
Fuzzy System 1: Feature-based Input Value Extraction ● This system employs fuzzy logic to extract features from input data, specifically binary images of handwritten digits. ● These features, such as stroke thickness, number of strokes, line straightness, etc., are then used as inputs to a neural network for classification.	Fuzzy System 2: Direct Image Enhancement and Character Detection ● In contrast to Set 1, this fuzzy system directly enhances input binary images using fuzzy logic techniques, focusing on brightness and edge strength to detect characters. ● Instead of extracting features explicitly, the fuzzy system enhances the input images based on predefined rules to improve the visibility of characters for classification.
Neural Network 1: Dense Neural Network (DNN) ● After feature extraction by the fuzzy system, the processed features are fed into a dense neural network for classification. ● The DNN consists of multiple densely connected layers, each with a specified number of neurons. ● This architecture allows the network to learn complex patterns and relationships among the extracted features, ultimately making predictions about the class labels of the input images.	Neural Network 2: Convolutional Neural Network (CNN) ● The enhanced images from Fuzzy System 2 are then processed by a CNN for classification. ● CNNs are well-suited for image classification tasks due to their ability to learn spatial hierarchies of features through convolutional and pooling layers. ● By leveraging the enhanced images, the CNN can extract relevant features and make accurate predictions about the class labels of the input images.
Evolutionary Algorithm 1: Optimizing Parameters in DNN ● Evolutionary algorithms, such as genetic algorithms or particle swarm optimization, are employed to optimize the parameters of the DNN. ● These algorithms explore the parameter space, adjusting weights	Evolutionary Algorithm 2: Optimizing Parameters in CNN ● Similar to Set 1 but with different settings of selection, crossover, and mutation, an evolutionary algorithm is employed to optimize the parameters of the CNN.

and biases to minimize the classification error or loss function. ● By fine-tuning the DNN parameters, the algorithm aims to improve the network's performance and generalization ability.	● By adjusting the weights and biases of the convolutional and fully connected layers, the algorithm aims to improve the CNN's performance in classifying the enhanced images. ● This optimization process helps fine-tune the CNN to better capture relevant features and improve classification accuracy.

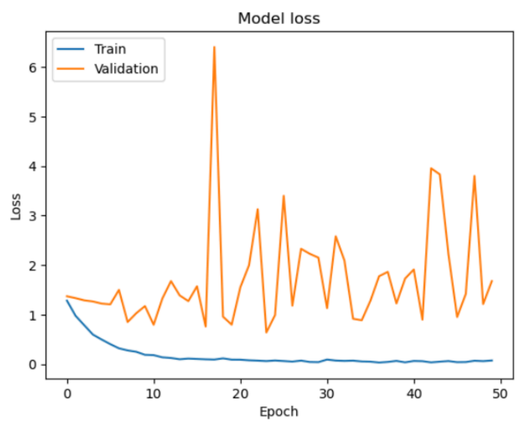
7.1 Neural Network

Set 1	**Set 2**
Model Accuracy Curve	
 ● Training accuracy line increase with fluctuation ● Training accuracy is overlapping with validation line The model might be overfitting to the training data, as it is adapting too closely to the noise or specific patterns within the training set.	 ● Training accuracy line increases steadily ● Training accuracy line is always higher than validation accuracy line The model's performance on the training data consistently improves across epochs. Additionally, the model is learning well from the training data but may not generalize optimally to unseen data.
Model Loss Curve	



- Validation loss line decreases with fluctuation
- Training loss line decreases gradually

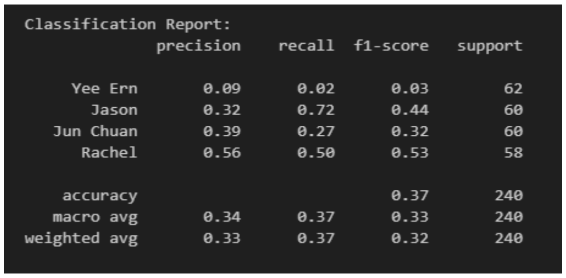
The model’s performance on the training data can be said to be improving gradually and slowly despite small improvement as the training loss only decreases a little over each epoch.



- Training loss line decreases steadily
- Training loss line is always lower than validation loss line

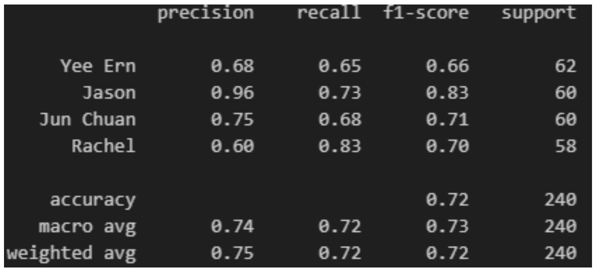
The model's performance on the training data is said to have improved across epochs as training loss decreases. Additionally, the model is learning well from the training data but may not generalize optimally to unseen data.

Classification Report



```
Classification Report:
              precision    recall  f1-score   support

     Yee Ern       0.09      0.02      0.03        62
       Jason       0.32      0.72      0.44        60
   Jun Chuan       0.39      0.27      0.32        60
      Rachel       0.56      0.50      0.53        58

    accuracy                           0.37       240
   macro avg       0.34      0.37      0.33       240
weighted avg       0.33      0.37      0.32       240
```

- Accuracy of 37%
- Lower precision, recall, and f1-scores across all classes.
- The "Jason" class has a relatively higher precision but lower recall, indicating that while it identifies "Jason" correctly, it misses many instances of this class.
- Performs poorly in identifying the "Yee Ern" class, with a low precision, recall, and f1-score



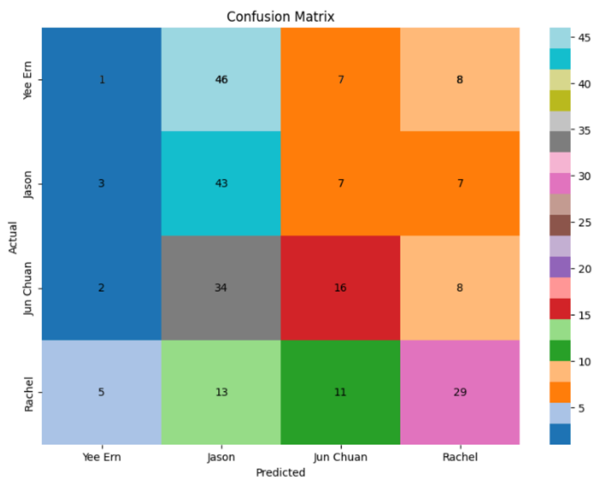
```
              precision    recall  f1-score   support

     Yee Ern       0.68      0.65      0.66        62
       Jason       0.96      0.73      0.83        60
   Jun Chuan       0.75      0.68      0.71        60
      Rachel       0.60      0.83      0.70        58

    accuracy                           0.72       240
   macro avg       0.74      0.72      0.73       240
weighted avg       0.75      0.72      0.72       240
```

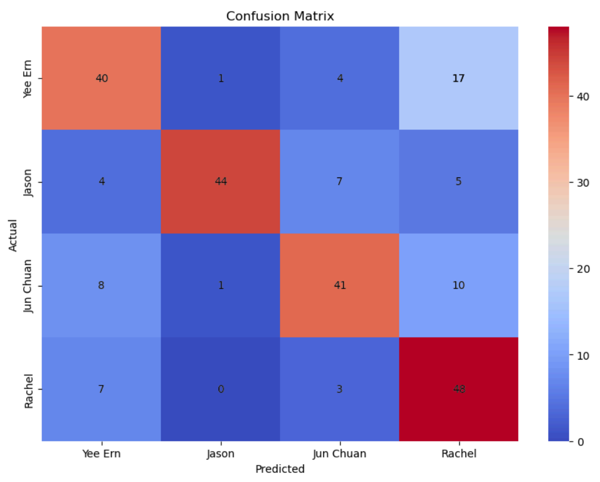
- Accuracy of 70%.
- Higher precision, recall, and f1-scores across all classes compared to the first model.
- Performs exceptionally well in identifying the "Jason" class, with high precision, recall, and f1-score.
- The "Yee Ern" class also shows considerable improvement compared to the first model.

Overall, the model's performance is relatively low, with low scores across most metrics.

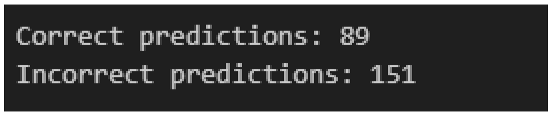
Overall, the second model outperforms the first model significantly, with better accuracy and overall performance metrics.

Confusion Matrix

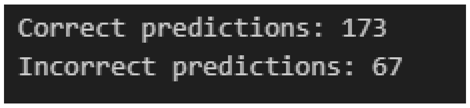




Label Prediction

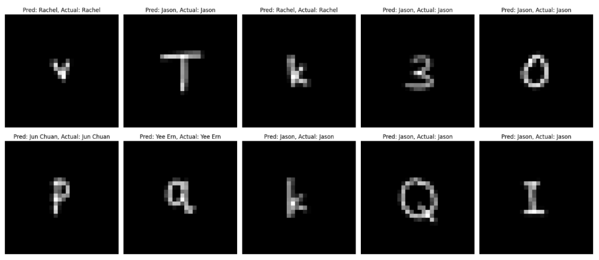


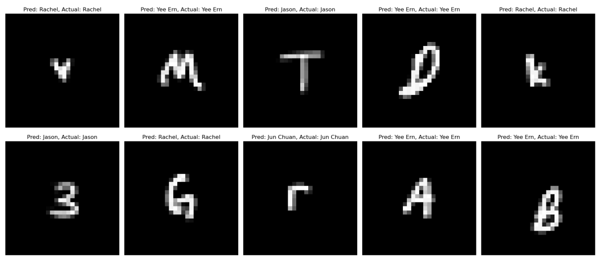
```
Correct predictions: 89
Incorrect predictions: 151
```



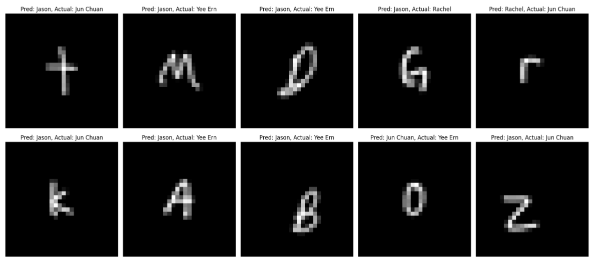
```
Correct predictions: 173
Incorrect predictions: 67
```

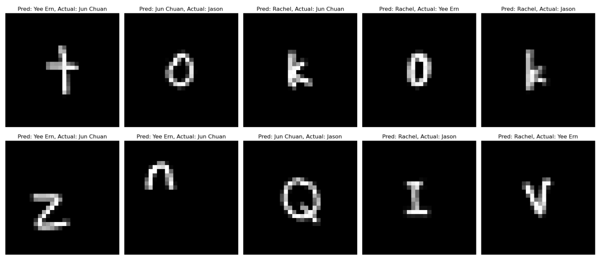
Correctly Predicted Labels Visualization


Pred: Rachel, Actual: Rachel
Pred: Jason, Actual: Jason
Pred: Rachel, Actual: Rachel
Pred: Jason, Actual: Jason
Pred: Jason, Actual: Jason
Pred: Jun Chuan, Actual: Jun Chuan
Pred: Yee Ern, Actual: Yee Ern
Pred: Jason, Actual: Jason
Pred: Jason, Actual: Jason
Pred: Jason, Actual: Jason

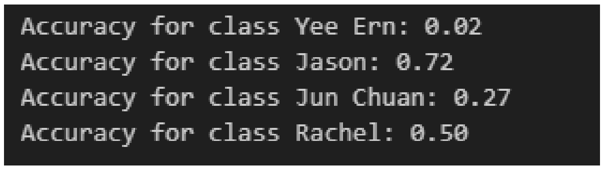

Pred: Rachel, Actual: Rachel
Pred: Yee Ern, Actual: Yee Ern
Pred: Jason, Actual: Jason
Pred: Yee Ern, Actual: Yee Ern
Pred: Rachel, Actual: Rachel
Pred: Jason, Actual: Jason
Pred: Rachel, Actual: Rachel
Pred: Jun Chuan, Actual: Jun Chuan
Pred: Yee Ern, Actual: Yee Ern
Pred: Yee Ern, Actual: Yee Ern

Wrongly Predicted Labels Visualization

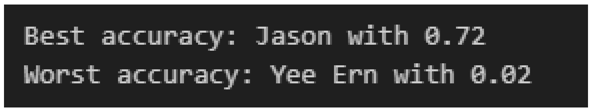

Pred: Jason, Actual: Jun Chuan
Pred: Jason, Actual: Yee Ern
Pred: Jason, Actual: Yee Ern
Pred: Jason, Actual: Rachel
Pred: Rachel, Actual: Jun Chuan
Pred: Jason, Actual: Jun Chuan
Pred: Jason, Actual: Yee Ern
Pred: Jason, Actual: Yee Ern
Pred: Jun Chuan, Actual: Yee Ern
Pred: Jason, Actual: Jun Chuan


Pred: Yee Ern, Actual: Jun Chuan
Pred: Jun Chuan, Actual: Jason
Pred: Rachel, Actual: Jun Chuan
Pred: Rachel, Actual: Yee Ern
Pred: Rachel, Actual: Jason
Pred: Yee Ern, Actual: Jun Chuan
Pred: Yee Ern, Actual: Jun Chuan
Pred: Jun Chuan, Actual: Jason
Pred: Rachel, Actual: Jason
Pred: Rachel, Actual: Yee Ern

Accuracy by Class

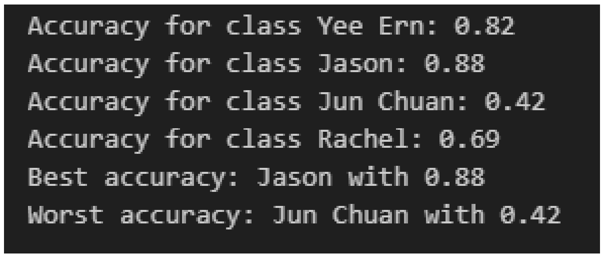


```
Accuracy for class Yee Ern: 0.02
Accuracy for class Jason: 0.72
Accuracy for class Jun Chuan: 0.27
Accuracy for class Rachel: 0.50
```



```
Best accuracy: Jason with 0.72
Worst accuracy: Yee Ern with 0.02
```

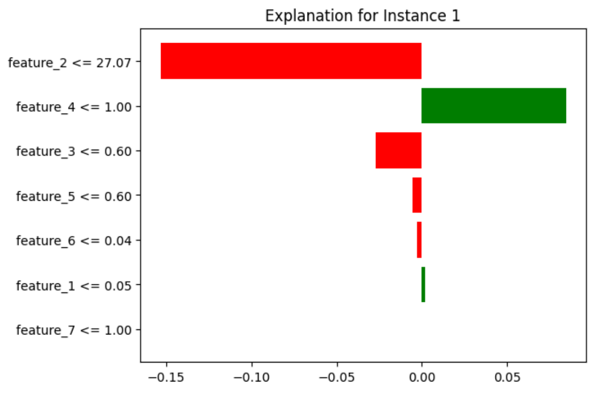
The accuracy out of the four labels ranges from 0.02 ('Yee Ern' class label) to 0.72 ('Jason' class label).



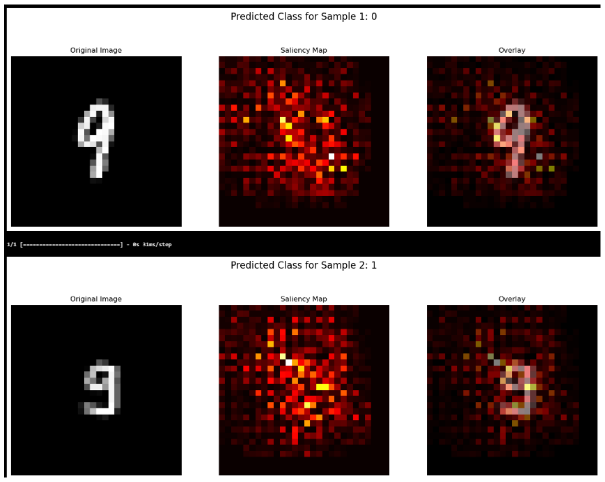
```
Accuracy for class Yee Ern: 0.82
Accuracy for class Jason: 0.88
Accuracy for class Jun Chuan: 0.42
Accuracy for class Rachel: 0.69
Best accuracy: Jason with 0.88
Worst accuracy: Jun Chuan with 0.42
```

The accuracy out of the four labels ranges from 0.42 ('Jun Chuan' class label) to 0.82 ('Jason' class label).

Model Information Visualization



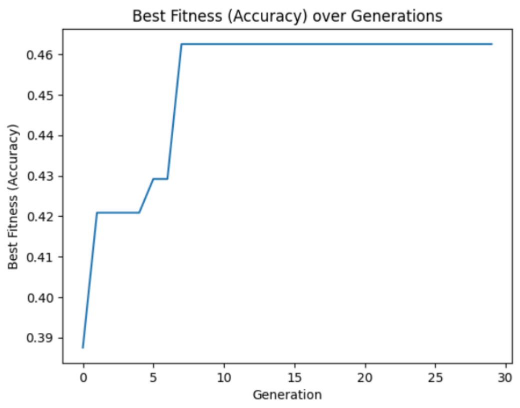
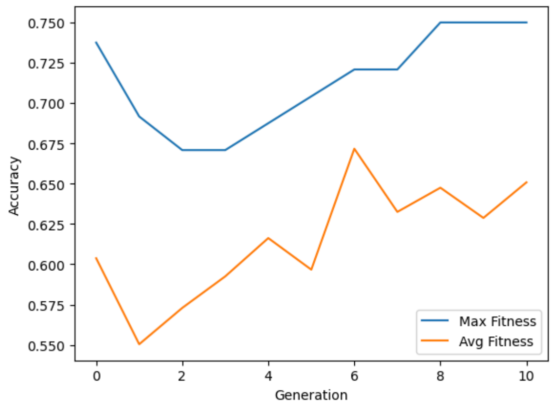
LIME (*notebook.community*, n.d.), also known as Local Interpretable Model-agnostic Explanations tabular explainer, which is utilized to provide explanations for individual predictions made by the model. In this setup, the explainer is configured with the training data (X_train), feature names, class names (e.g., "Yee Ern", "Jason", "Jun Chuan", "Rachel"), and the option to discretize continuous features. Each explanation provides insights into the model's decision-making process for the corresponding instance, aiding in the interpretation and understanding of the model's behavior.

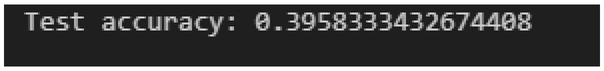


```
1/1 [==============================] - 0s 31ms/step
```

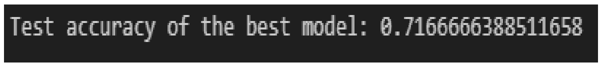
The image showcases saliency maps for two samples from a dataset.The saliency map visualizes areas most influential in the model's decision-making process, with brighter regions (yellow/red) indicating higher importance.The overlays combine the original images with their respective saliency maps, showing which parts influence the model's predictions.These visualizations provide insight into the model's internal workings, illustrating how it interprets different parts of the input images to make predictions.

7.2 Evolutionary Algorithm

Set 1	**Set 2**
Best Fitness over Generation	
 In the first few generations (Generation 0 to Generation 8), there is a sharp increase in the fitness score (0.39 to 0.46). This behavior aligns with the expected performance of optimization algorithms, where each generation aims to improve upon the previous one. The initial boost suggests that the algorithm successfully explores the solution space and identifies better solutions. Around generation 8, the fitness score levels off at approximately 0.46. This plateau indicates that further improvements are not significant. The lack of progress suggests that the algorithm might be trapped in a local optimum or that the problem space has inherent limitations, causing it to converge prematurely to a suboptimal solution. Nevertheless, it still demonstrates a good evolutionary algorithm with the increasing fitness value across the 50 generations.	 The graph shows that the genetic algorithm improves maximum model accuracy over ten generations, with initial fluctuations and a steady increase, while average accuracy varies due to exploration and refinement of solutions. The maximum fitness decreases initially, reaching its lowest point in the third generation, likely due to exploration and the introduction of new genetic material. From the third generation onwards, the maximum fitness steadily increases, stabilizing around the eighth generation. The average fitness fluctuates throughout, reflecting the algorithm's balance between exploring new solutions and refining the best ones.
Best Accuracy	

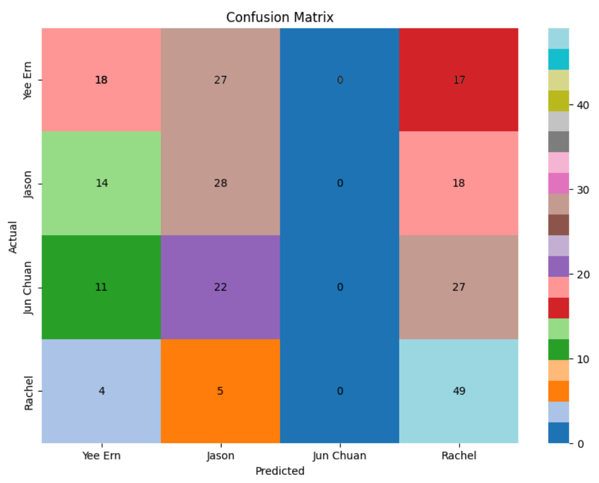


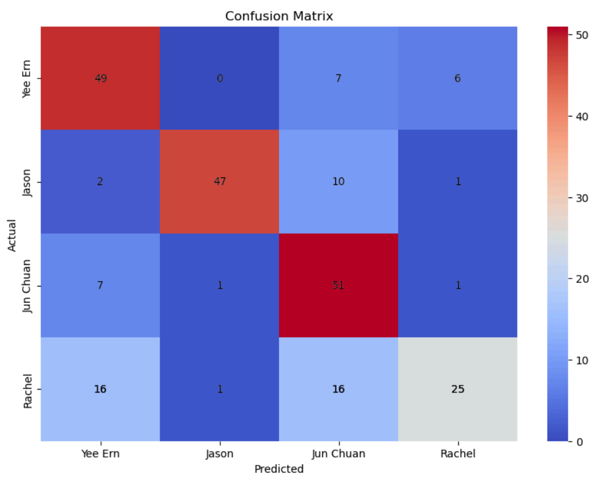
```
Test accuracy: 0.39583333432674408
```



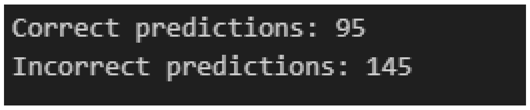
```
Test accuracy of the best model: 0.7166666388511658
```

Confusion Matrix

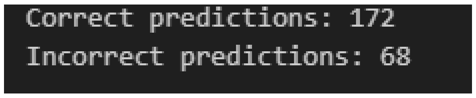




Label Prediction

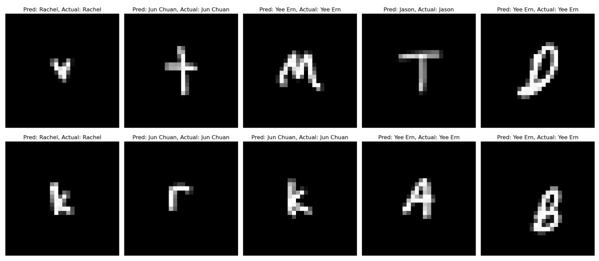


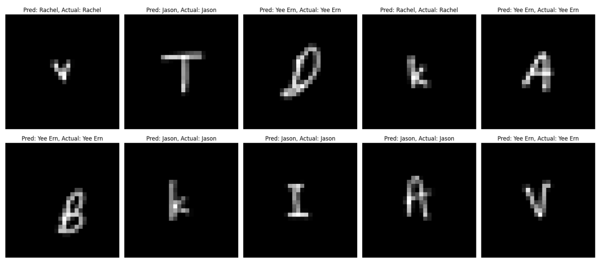
```
Correct predictions: 95
Incorrect predictions: 145
```



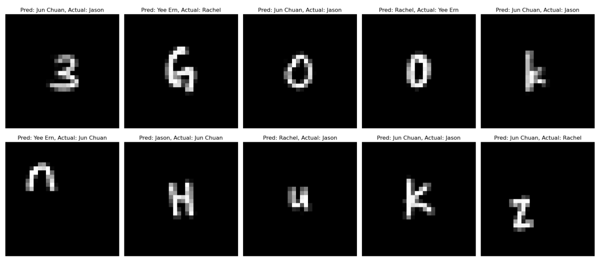
```
Correct predictions: 172
Incorrect predictions: 68
```

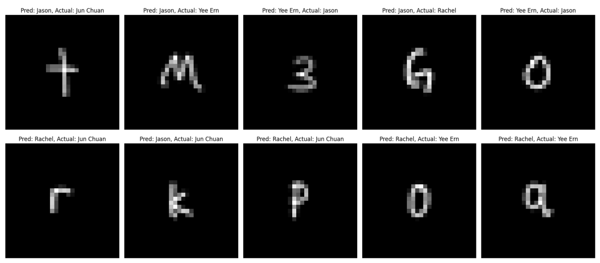
Correctly Predicted Labels Visualization


Pred: Rachel, Actual: Rachel
Pred: Jun Chuan, Actual: Jun Chuan
Pred: Yee Ern, Actual: Yee Ern
Pred: Jason, Actual: Jason
Pred: Yee Ern, Actual: Yee Ern
Pred: Rachel, Actual: Rachel
Pred: Jun Chuan, Actual: Jun Chuan
Pred: Jun Chuan, Actual: Jun Chuan
Pred: Yee Ern, Actual: Yee Ern
Pred: Yee Ern, Actual: Yee Ern


Pred: Rachel, Actual: Rachel
Pred: Jason, Actual: Jason
Pred: Yee Ern, Actual: Yee Ern
Pred: Rachel, Actual: Rachel
Pred: Yee Ern, Actual: Yee Ern
Pred: Yee Ern, Actual: Yee Ern
Pred: Jason, Actual: Jason
Pred: Jason, Actual: Jason
Pred: Jason, Actual: Jason
Pred: Yee Ern, Actual: Yee Ern

Wrongly Predicted Labels Visualization


Pred: Jun Chuan, Actual: Jason
Pred: Yee Ern, Actual: Rachel
Pred: Jun Chuan, Actual: Jason
Pred: Rachel, Actual: Yee Ern
Pred: Jun Chuan, Actual: Jason
Pred: Yee Ern, Actual: Jun Chuan
Pred: Jason, Actual: Jun Chuan
Pred: Rachel, Actual: Jason
Pred: Jun Chuan, Actual: Jason
Pred: Jun Chuan, Actual: Rachel


Pred: Jason, Actual: Jun Chuan
Pred: Jason, Actual: Yee Ern
Pred: Yee Ern, Actual: Jason
Pred: Jason, Actual: Rachel
Pred: Yee Ern, Actual: Jason
Pred: Rachel, Actual: Jun Chuan
Pred: Jason, Actual: Jun Chuan
Pred: Rachel, Actual: Jun Chuan
Pred: Rachel, Actual: Yee Ern
Pred: Rachel, Actual: Yee Ern

After applying genetic algorithms (GA) to optimize the parameters of both neural network models, there is a noticeable improvement in the performance of both sets. However, despite this enhancement, Set 2 continues to outperform Set 1. This superiority can be attributed to the inherent advantages of Set 2's approach, where the Convolutional Neural Network (CNN) directly classifies fuzzy-enhanced binary images without explicitly extracting features. By leveraging the entire image for training and incorporating advanced image enhancement techniques, Set 2 captures more comprehensive and discriminative information, leading to superior classification accuracy. The CNN's ability to automatically learn relevant features from the enhanced images contributes to its robustness and effectiveness in identifying individual handwriting styles, ultimately surpassing the performance of the Dense Neural Network (DNN) in Set 1.

8. Conclusion

In conclusion, Set 2 outperforms Set 1 due to its more holistic approach to feature extraction and utilization. In Set 1, the Fuzzy System 1 extracts input values based on a limited set of features, and only these features are used for training the Dense Neural Network (DNN). However, in Set 2, Fuzzy System 2 directly enhances the images and detects characters, allowing the Convolutional Neural Network (CNN) in Neural Network 2 to utilize the entire image for training. By enhancing the images and considering the complete information present in them, Set 2 achieves better performance compared to Set 1. This highlights the importance of leveraging advanced techniques like image enhancement and utilizing the full information available in the data for improving model performance in complex tasks such as handwriting recognition.

For limitations, the dataset may lack diversity in terms of handwriting styles, as it primarily reflects the characteristics of the individuals involved in creating it. This limitation can hinder the model's ability to generalize to handwriting styles that differ significantly from those present in the dataset. The dataset may inadvertently contain biases or inconsistencies, reflecting individual preferences or idiosyncrasies in handwriting. These biases could impact the model's performance and introduce inaccuracies in classification, especially when encountering handwriting styles not represented in the dataset.

In future iterations, several enhancements can elevate the handwriting recognition system's performance and applicability. Augmenting the dataset with diverse handwriting samples and implementing advanced preprocessing techniques like noise reduction and data augmentation can refine input quality and enrich feature extraction. Exploring varied neural network architectures, employing ensemble learning, and incorporating domain adaptation techniques can enhance model robustness and adaptation to diverse handwriting styles and contexts. Continuous evaluation, interdisciplinary collaboration, and ethical considerations remain pivotal, ensuring the system's integrity, effectiveness, and alignment with user needs and societal norms. By embracing these avenues, the handwriting recognition system can evolve into a more accurate, versatile, and ethically sound tool, extending its utility across forensic, authentication, and document processing domains.

9. References

Features Extraction Using Fuzzy-logic-based Object Model. (n.d.). Www.witpress.com. https://www.witpress.com/elibrary/wit-transactions-on-information-and-communication-technologies/6/10964

Mikalaichaly. (2018, October 8). *DIY MNIST Dataset*. Kaggle. https://www.kaggle.com/code/mikalaichaly/diy-mnist-dataset

notebook.community. (n.d.). LIME. https://notebook.community/marcotcr/lime/doc/notebooks/Tutorial%20-%20MNIST%20and%20RF

Witold Pedrycz, Kandel, A., & Zhang, Y.-Q. (1998). Neurofuzzy Systems. *Springer EBooks*, 311–380. https://doi.org/10.1007/978-1-4615-5505-6_10