Peer Evaluation System UI/UX

We worked on the signature recognition model and developed a system to detect whether a signature is real or forged.

The code for the model and the output screenshots are attached below: -

- **Python code: -**
  1. **For signature Classification -** The below model gets failed so we have changed the approach which is in part 2.

```
import os
import cv2
import numpy as np
from google.colab import drive
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Dropout, BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

drive.mount('/content/drive')


# Define image dimensions
IMG_WIDTH = 256
IMG_HEIGHT = 256

# Load images from the dataset directory
def load_images(folder):
    images = []
    labels = []
    person_labels = {}
```

```
    # Iterate over files in the dataset folder
    for filename in os.listdir(folder):
        if 'original' in filename:  # Only process original signatures
            img_path = os.path.join(folder, filename)
            img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
            img = cv2.resize(img, (IMG_WIDTH, IMG_HEIGHT))
            img = img / 255.0  # Normalize pixel values to [0,1]
            images.append(img)

            # Extract person number from filename
            person_id = filename.split('_')[1]  # Assuming the filename is
original_Per_imgno

            # Assign a unique label to each person
            if person_id not in person_labels:
                person_labels[person_id] = len(person_labels)

            labels.append(person_labels[person_id])

    return np.array(images), np.array(labels)


# Load dataset
dataset_folder = '/content/drive/MyDrive/Sign_Data/Real'  # Update with your
dataset folder
X, y = load_images(dataset_folder)



# Reshape images for CNN input
X = X.reshape(-1, IMG_WIDTH, IMG_HEIGHT, 1)

# Encode labels (e.g., Person_1, Person_2 to numerical labels)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# One-hot encode the labels
y_onehot = to_categorical(y_encoded)
```

```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2,
random_state=42)

# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# CNN model
model = Sequential()

# Convolutional layer 1
model.add(Conv2D(32, (3, 3), input_shape=(IMG_WIDTH, IMG_HEIGHT, 1),
activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Convolutional layer 2
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Convolutional layer 3
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Convolutional layer 4 (new layer for additional complexity)
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(BatchNormalization())
```

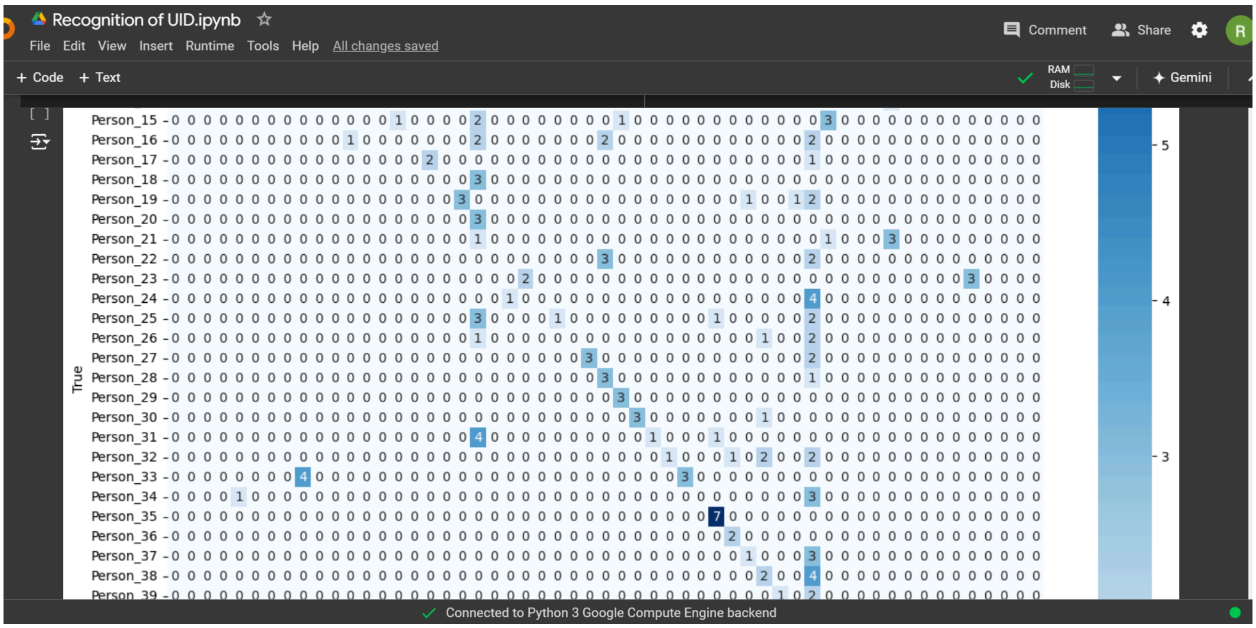
```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Convolutional layer 5 (new layer for additional complexity)
model.add(Conv2D(512, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Flatten layer
model.add(Flatten())

# Dense layer
model.add(Dense(512, activation='relu'))  # Increased number of units for better
learning capacity
model.add(Dropout(0.2))  # Increased dropout to reduce overfitting

num_classes = len(np.unique(y_encoded))  # Number of unique persons in your
dataset
# Output layer (softmax for multi-class classification)
model.add(Dense(num_classes, activation='softmax'))

# Compile model
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='categorical_crossentropy', metrics=['accuracy'])

# Define early stopping callback
early_stopping = EarlyStopping(
    monitor='loss',
    patience=5,  # Increased patience to allow the model more time to improve
    restore_best_weights=True
)


# Fit the model
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=32),
    validation_data=(X_test, y_test),
    epochs=50,
    callbacks=[early_stopping]
```

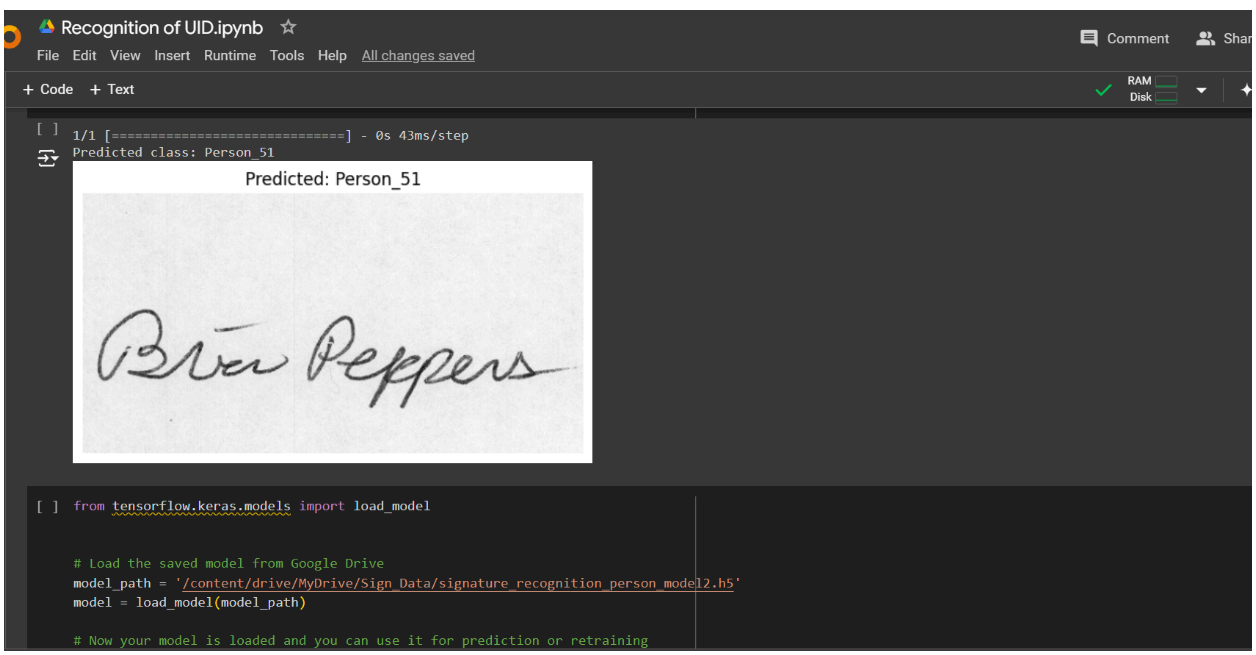
```
)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)

print(f'Test Accuracy: {accuracy * 100:.2f}%')

# Save the model
model.save('/content/drive/MyDrive/Sign_Data/signature_recognition_person_mo
del3.h5')


# Ensure that the y_test and y_pred are one-hot encoded or properly encoded
before applying argmax
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

# If label_encoder is not used, we'll define the class names based on unique labels
in the dataset
class_names = [f'Person_{i}' for i in range(1, num_classes + 1)]  # Assuming
`num_classes` gives the number of persons

# Print classification report
print(classification_report(y_true_classes, y_pred_classes,
target_names=class_names))

# Confusion Matrix
cm = confusion_matrix(y_true_classes, y_pred_classes)

# Plot Confusion Matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()


# Function to load and preprocess a single image
```

```
def preprocess_image(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img = cv2.resize(img, (IMG_WIDTH, IMG_HEIGHT))  # Resize to match the
input dimensions
    img = img / 255.0  # Normalize pixel values to [0,1]
    img = img.reshape(1, IMG_WIDTH, IMG_HEIGHT, 1)  # Reshape to match
model input shape
    return img

# Path to the image you want to predict
image_path = '/content/drive/MyDrive/Sign_Data/Real/original_18_16.png'  #
Provide the path to your image

# Preprocess the image
single_image = preprocess_image(image_path)

# Predict the class
prediction = model.predict(single_image)

# Get the predicted class index
predicted_class_index = np.argmax(prediction)

# Assuming your class names are formatted as Person_1, Person_2, ...,
Person_num_classes
class_names = [f'Person_{i}' for i in range(1, num_classes + 1)]

# Get the predicted class name
predicted_class_name = class_names[predicted_class_index]

# Output the predicted class
print(f'Predicted class: {predicted_class_name}')

# Optionally, display the image and the prediction
plt.imshow(cv2.imread(image_path, cv2.IMREAD_GRAYSCALE), cmap='gray')
plt.title(f'Predicted: {predicted_class_name}')
plt.axis('off')
plt.show()

from tensorflow.keras.models import load_model
```

```
# Load the saved model from Google Drive
model_path =
'/content/drive/MyDrive/Sign_Data/signature_recognition_person_model2.h5'
model = load_model(model_path)

# Now your model is loaded and you can use it for prediction or retraining

# Function to load and preprocess a single image
def preprocess_image(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img = cv2.resize(img, (IMG_WIDTH, IMG_HEIGHT))  # Resize to match the
input dimensions
    img = img / 255.0  # Normalize pixel values to [0,1]
    img = img.reshape(1, IMG_WIDTH, IMG_HEIGHT, 1)  # Reshape to match
model input shape
    return img

# Path to the image you want to predict
image_path = '/content/drive/MyDrive/Sign_Data/Test/original_2_21.png'  #
Provide the path to your image

# Preprocess the image
single_image = preprocess_image(image_path)

# Predict the class
prediction = model.predict(single_image)

# Get the predicted class index
predicted_class_index = np.argmax(prediction)

# Assuming your class names are formatted as Person_1, Person_2, ...,
Person_num_classes
class_names = [f'Person_{i}' for i in range(1, num_classes + 1)]

# Get the predicted class name
predicted_class_name = class_names[predicted_class_index]

# Output the predicted class
print(f'Predicted class: {predicted_class_name}')
```

# Optionally, display the image and the prediction
plt.imshow(cv2.imread(image_path, cv2.IMREAD_GRAYSCALE), cmap='gray')
plt.title(f'Predicted: {predicted_class_name}')
plt.axis('off')
plt.show()

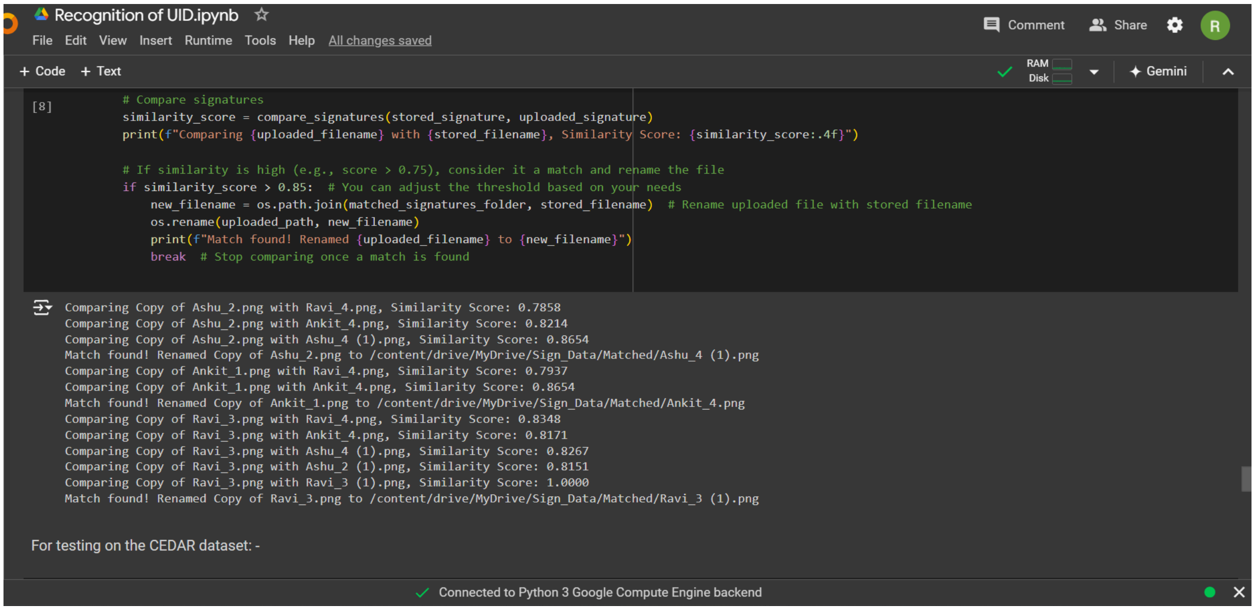


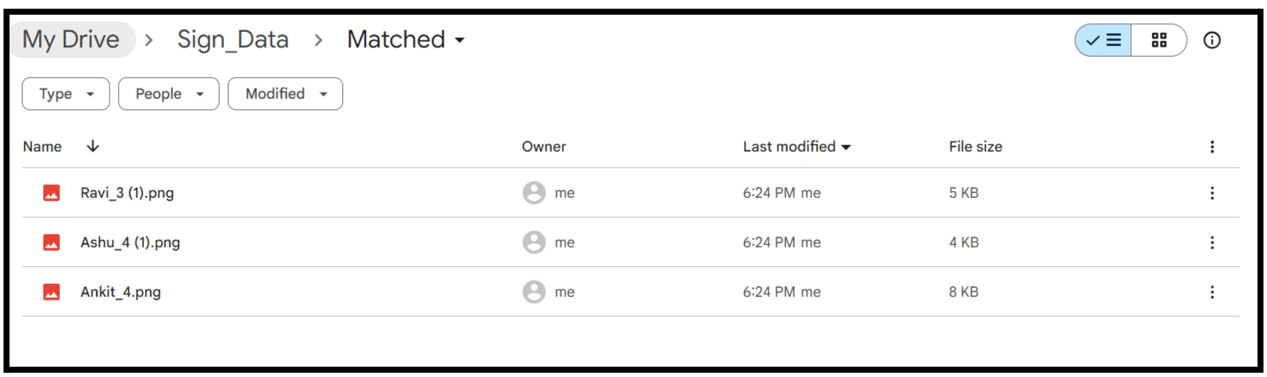


2. **For signature mapping -** Approach 2 using OpenCV and other comparison techniques.

```
import os
import cv2
import numpy as np
from skimage.metrics import structural_similarity as ssim
from scipy.ndimage import center_of_mass
import matplotlib.pyplot as plt

IMG_WIDTH = 256
IMG_HEIGHT = 256

# Function to preprocess image (resize and normalize)
def preprocess_image(image):
    image = cv2.resize(image, (IMG_WIDTH, IMG_HEIGHT))  # Resize
    image = image / 255.0  # Normalize pixel values
    return image

# Align signatures by centering the image based on the center of mass
def align_images(image):
    cy, cx = center_of_mass(image)
    height, width = image.shape
    shift_x = int(width / 2 - cx)
    shift_y = int(height / 2 - cy)
    translation_matrix = np.float32([[1, 0, shift_x], [0, 1, shift_y]])
    aligned_image = cv2.warpAffine(image, translation_matrix, (width, height))
    return aligned_image

# Compare signatures using SSIM with the correct data range
def compare_signatures(stored_sig, uploaded_sig):
    aligned_stored_sig = align_images(stored_sig)
    aligned_uploaded_sig = align_images(uploaded_sig)
    score, _ = ssim(aligned_stored_sig, aligned_uploaded_sig, full=True,
data_range=1.0)
    return score

# Folder paths
```

```
stored_signatures_folder = '/content/drive/MyDrive/Sign_Data/Rtest/'  # Stored
signature images
uploaded_signatures_folder = '/content/drive/MyDrive/Sign_Data/Matched/'  #
Uploaded signature images
matched_signatures_folder = '/content/drive/MyDrive/Sign_Data/Matched/'  #
Folder to store renamed matching images

# Iterate through each uploaded image
for uploaded_filename in os.listdir(uploaded_signatures_folder):
    uploaded_path = os.path.join(uploaded_signatures_folder, uploaded_filename)

    # Load the uploaded signature
    uploaded_signature = cv2.imread(uploaded_path,
cv2.IMREAD_GRAYSCALE)
    uploaded_signature = preprocess_image(uploaded_signature)

    # Compare with each stored signature
    for stored_filename in os.listdir(stored_signatures_folder):
        stored_path = os.path.join(stored_signatures_folder, stored_filename)

        # Load the stored signature
        stored_signature = cv2.imread(stored_path, cv2.IMREAD_GRAYSCALE)
        stored_signature = preprocess_image(stored_signature)

        # Compare signatures
        similarity_score = compare_signatures(stored_signature,
uploaded_signature)
        print(f"Comparing {uploaded_filename} with {stored_filename}, Similarity
Score: {similarity_score:.4f}")

        # If similarity is high (e.g., score > 0.75), consider it a match and rename the
file
        if similarity_score > 0.85:  # You can adjust the threshold based on your
needs
            new_filename = os.path.join(matched_signatures_folder, stored_filename)
# Rename uploaded file with stored filename
            os.rename(uploaded_path, new_filename)
            print(f"Match found! Renamed {uploaded_filename} to {new_filename}")
            break  # Stop comparing once a match is found
```


Recognition of UID.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Gemini
[8]
# Compare signatures
similarity_score = compare_signatures(stored_signature, uploaded_signature)
print(f"Comparing {uploaded_filename} with {stored_filename}, Similarity Score: {similarity_score:.4f}")
# If similarity is high (e.g., score > 0.75), consider it a match and rename the file
if similarity_score > 0.85: # You can adjust the threshold based on your needs
new_filename = os.path.join(matched_signatures_folder, stored_filename) # Rename uploaded file with stored filename
os.rename(uploaded_path, new_filename)
print(f"Match found! Renamed {uploaded_filename} to {new_filename}")
break # Stop comparing once a match is found
Comparing Copy of Ashu_2.png with Ravi_4.png, Similarity Score: 0.7858
Comparing Copy of Ashu_2.png with Ankit_4.png, Similarity Score: 0.8214
Comparing Copy of Ashu_2.png with Ashu_4 (1).png, Similarity Score: 0.8654
Match found! Renamed Copy of Ashu_2.png to /content/drive/MyDrive/Sign_Data/Matched/Ashu_4 (1).png
Comparing Copy of Ankit_1.png with Ravi_4.png, Similarity Score: 0.7937
Comparing Copy of Ankit_1.png with Ankit_4.png, Similarity Score: 0.8654
Match found! Renamed Copy of Ankit_1.png to /content/drive/MyDrive/Sign_Data/Matched/Ankit_4.png
Comparing Copy of Ravi_3.png with Ravi_4.png, Similarity Score: 0.8348
Comparing Copy of Ravi_3.png with Ankit_4.png, Similarity Score: 0.8171
Comparing Copy of Ravi_3.png with Ashu_4 (1).png, Similarity Score: 0.8267
Comparing Copy of Ravi_3.png with Ashu_2 (1).png, Similarity Score: 0.8151
Comparing Copy of Ravi_3.png with Ravi_3 (1).png, Similarity Score: 1.0000
Match found! Renamed Copy of Ravi_3.png to /content/drive/MyDrive/Sign_Data/Matched/Ravi_3 (1).png
For testing on the CEDAR dataset: -
Connected to Python 3 Google Compute Engine backend


My Drive › Sign_Data › Matched
Type
People
Modified
Name
Owner
Last modified
File size
Ravi_3 (1).png
me
6:24 PM me
5 KB
Ashu_4 (1).png
me
6:24 PM me
4 KB
Ankit_4.png
me
6:24 PM me
8 KB