SYMBIOSIS INSTITUTE OF TECHNOLOGY, PUNE – 412115

(A CONSTITUENT OF SYMBIOSIS INTERNATIONAL (DEEMED UNIVERSITY))

Web and Mobile Application Development Project Report

Project Report on:

Vocal Shield: Audio Censoring Mobile Application

Using NLP and Deep Learning

Submitted By:

Dev Bhanushali	22070126032
Abhishek Sinha	22070126005
Harsh Kotadiya	22070126040
Aditi Dhavle	22070126006
Amrut Ghadge	22070126007

UNDER THE GUIDANCE OF
Prof. Kalyani Kadam
Assistant Professor (AIML)

Department of Artificial Intelligence and Machine Learning

SYMBIOSIS INSTITUTE OF TECHNOLOGY, PUNE

Table of Content

Problem Statement

To develop a mobile application that converts **typed Hindi text** into **handwritten-style images** using **deep learning** and image translation techniques. The application integrates a backend powered by the **Pix2Pix architecture**, which processes the typed text and generates corresponding handwritten text images. These images are then aligned in a paragraph-like format to mimic natural handwriting.

The project addresses the need for digitized tools to generate authentic-looking Hindi handwritten text, which can be applied in educational tools, creative projects, or documentation systems. The goal is to provide a seamless, user-friendly platform for generating high-quality handwritten Hindi text images.

Introduction

In today's digital era, the demand for tools that bridge the gap between traditional and digital formats has grown significantly. The ability to generate authentic handwritten text images from digital inputs not only preserves the essence of human expression but also addresses practical applications in education, design, and documentation. Handwritten text retains an aesthetic and personal touch, often preferred for projects like greeting cards, certificates, or creative content. However, producing consistent and natural-looking handwritten text manually can be time-consuming and impractical.

Hindi, being one of the most widely spoken languages in the world, has a rich cultural and linguistic heritage. Despite its prevalence, tools for Hindi language processing often lag behind in terms of innovation compared to those available for global languages like English. The unique structure of Hindi characters, coupled with matras (diacritical marks), poses challenges in generating natural handwriting. Conventional solutions are either restricted to a single font style or fail to deliver the fluidity and natural appearance of authentic handwriting.

This project addresses these challenges by developing a mobile application that converts typed Hindi text into images of handwritten text. The application leverages advanced image-to-image translation techniques, powered by the **Pix2Pix architecture**, to learn and replicate the intricate patterns of Hindi handwriting. Users can input their text, specify parameters such as font size and canvas width, and receive visually appealing handwritten outputs arranged in a paragraph-like format.

The application not only serves practical needs but also opens new avenues for creative and personalized use cases. By integrating deep learning with a user-friendly mobile interface, this solution democratizes access to high-quality Hindi handwritten text generation, providing an innovative tool that combines tradition with modern technology.

(This just serves as an example)

Details of the platform

The Hindi Handwriting Text Generator Application is a Flutter-based software solution designed to convert typed Hindi text into handwritten-style images. The system leverages a combination of state management techniques (Cubit architecture) and singleton design patterns for API handling and data storage. The application aims to streamline the generation and display of handwritten Hindi text, allowing users to input typed Hindi sentences and customize attributes such as font size and canvas width.

Key features include:

- Real-time generation of handwritten text images.
- User-friendly interface with customizable input options.
- Efficient state management with Cubits for robust and predictable behavior.
- Storage and preview of generated images.
- Capability to save generated images to local storage or the gallery.

The backend system, developed using **Quart** and **PyTorch** is designed to leverage a **trained Pix2Pix model** to convert typed text into synthesized handwritten text. The system accepts a sentence as input, processes it word-by-word, generates corresponding handwritten-style images, aligns these words into a natural paragraph layout, and returns the resulting image to the client.

1. Application:

1) Application Architecture

The application is built on the **Flutter framework**, utilizing the following architectural components:

- **State Management**: Implemented using **Cubit** (part of the Flutter Bloc library), ensuring the separation of UI and business logic.
- **API Handling**: A singleton service (`ApiService`) is responsible for interfacing with a backend server to retrieve the handwritten text images.
- **Data Storage**: Another singleton service (`StorageService`) manages the temporary storage of image data for display across different parts of the application.

2) Backend Integration

The app communicates with a backend server through the ApiService class, which sends user inputs to a Flask-based API endpoint (`http://<backend-ip>:5000/predict`). The API returns an image of the handwritten Hindi text as raw bytes, which are then converted into a previewable format within the Flutter application. Key features of the API handling logic include:

- Dio Package: Utilized for HTTP requests with advanced configurations like response type and error handling.

- Error Handling: Implements robust error management for network issues and server errors, ensuring user feedback in case of failures.
- Singleton Pattern: Ensures a single instance of the API handler is used throughout the app, improving efficiency and reducing redundancy.

3) User Interface

The user interface (UI) of the application is designed with simplicity and functionality in mind. Below are the key screens:

1. Conversion Screen

The `ConversionScreen` serves as the primary user interface for text-to-handwriting conversion. Key elements include:

- **Input Fields**:
  - A text field for entering the Hindi sentence (content).
  - Numeric fields for specifying the font size and canvas width (`fontSize` and `canvasWidth`), with input validation using formatters.
- **Action Button**:
  - A button labeled **"Convert Text"** triggers the Cubit to send a request to the API and retrieve the handwritten text image.
  - Displays a loading indicator during the API call.
- **Preview Section**:
  - If an image is successfully generated, it is displayed in a preview area.
  - A button labeled **"View Image"** allows navigation to the image viewer screen.

```dart
Widget queryButton() {
    return TextButton(
      onPressed: () async {
        await context.read<ConversionCubit>().getPrediction(
            content.text != '' ? content.text + " " * 100 : "हमारे " * 100,
            fontSize.text.isNotEmpty ? int.parse(fontSize.text) : 80,
            canvasWidth.text.isNotEmpty ? int.parse(canvasWidth.text) : 2000);
      },
      style: const ButtonStyle(
          shape: WidgetStatePropertyAll(RoundedRectangleBorder(
              borderRadius: BorderRadius.all(Radius.circular(15)))),
          backgroundColor: WidgetStatePropertyAll(Colors.black87),
          foregroundColor: WidgetStatePropertyAll(Colors.white)),
      child: Row(
        children: [
          Expanded(
            child: Container(
              height: 60,
              alignment: Alignment.center,
              child: BlocConsumer<ConversionCubit, ConversionState>(
                listener: (context, state) {
                  if (state is ConversionErrorState) {
                    showSnackbarMessage(context, false, state.message);
                  }
                },
                builder: (context, state) {
                  if (state is ConversionLoadingState) {
                    return Padding(
                      padding: const EdgeInsets.all(8.0),
                      child: const Center(child: CircularProgressIndicator()),
                    );
                  } else {
                    return const Center(
                        child: Text(
                      "Convert Text",
                      style: TextStyle(fontSize: 18),
                    ));
                  }
                },
              ),
            ),
          ),
        ],
      ),
    );
  }

  Widget ImagePreview() {
    return BlocBuilder<ConversionCubit, ConversionState>(
      builder: (context, state) {
        return StorageService().predictedImage != null
            ? Column(children: [
                Row(
                  children: [
                    Expanded(
                      child: Container(
                        decoration: BoxDecoration(
                            borderRadius: BorderRadius.circular(15),
                            color: Colors.black),
                        height: 300,
                        child: Image.memory(
                          StorageService().predictedImage!,
                          fit: BoxFit.contain,
                        ),
                      ),
                    ),
                  ],
                ),
                SizedBox(height: 12),
                TextButton(
                    onPressed: () => Navigator.push(
                        context,
                        MaterialPageRoute(
                            builder: (context) => const ImageScreen())),
                    child: Text(
                      "View Image",
                      style: TextStyle(color: Colors.black),
                    ))
              ])
            : Container();
      },
    );
  }
}

```

```dart
class ConversionScreen extends StatefulWidget {
  const ConversionScreen({super.key});

  @override
  State<ConversionScreen> createState() => _ConversionScreenState();
}

class _ConversionScreenState extends State<ConversionScreen> {
  @override
  void initState() {
    super.initState();
    print("Screen has init");
  }

  TextEditingController content = TextEditingController();
  TextEditingController canvasWidth = TextEditingController();
  TextEditingController fontSize = TextEditingController();

  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          centerTitle: true,
          title: const Text("Hindi HT Generator"),
          backgroundColor: Colors.black,
          foregroundColor: Colors.white,
          elevation: 3,
          shadowColor: Colors.black,
        ),
        body: SafeArea(
          child: Container(
            padding: const EdgeInsets.symmetric(horizontal: 20),
            decoration: const BoxDecoration(color: Colors.white70),
            child: Column(
              children: [
                queryBody(),
                Padding(
                  padding: const EdgeInsets.fromLTRB(30, 10, 30, 10),
                  child: queryButton(),
                ),
                // StorageService().predictedImage != null
                //     ? Column(children: [
                //         Container(
                //           color: Colors.amber,
                //           height: 200,
                //           child: Image.memory(StorageService().predictedImage!),
                //         ),
                //         TextButton(
                //             onPressed: () => Navigator.push(
                //                 context,
                //                 MaterialPageRoute(
                //                     builder: (context) => const ImageScreen())),
                //             child: Text("View Image"))
                //       ])
                //     : Container()
              ],
            ),
          ),
        ));
  }

  Widget queryBody() {
    return Expanded(
      child: ListView(
        physics: const BouncingScrollPhysics(
            parent: AlwaysScrollableScrollPhysics()),
        children: [
          Column(
            mainAxisAlignment: MainAxisAlignment.start,
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              const SizedBox(height: 30),
              const Padding(
                  padding: EdgeInsets.fromLTRB(18, 0, 0, 0),
                  child: Text("Enter Text Content",
                      style: TextStyle(color: Colors.black),
                      textAlign: TextAlign.left)),
              CustomTextInput(
                  controller: content,
                  hintText: "Type your text here",
                  maxLines: 10),
              const SizedBox(height: 30),
              const Padding(
                  padding: EdgeInsets.fromLTRB(18, 0, 0, 0),
                  child: Text("Enter Font Size",
                      style: TextStyle(color: Colors.black),
                      textAlign: TextAlign.left)),
              CustomTextInput(
                  controller: fontSize,
                  hintText: "Default: 80",
                  inputFormatters: [FilteringTextInputFormatter.digitsOnly]),
              const SizedBox(height: 30),
              const Padding(
                  padding: EdgeInsets.fromLTRB(18, 0, 0, 0),
                  child: Text("Enter Canvas Width",
                      style: TextStyle(color: Colors.black),
                      textAlign: TextAlign.left)),
              CustomTextInput(
                  controller: canvasWidth,
                  hintText: "Default: 3000",
                  inputFormatters: [FilteringTextInputFormatter.digitsOnly]),
              const SizedBox(height: 30),
              ImagePreview(),
            ],
          ),
        ],
      ),
    );
  }
```

2. Image Screen

The `ImageScreen` allows users to view and interact with the generated image. Features include:

- **PhotoView**:
  - Displays the image in full screen, with pinch-to-zoom functionality for enhanced user experience.
- **Save Image**:
  - A button to save the image to the app's local storage and the device's gallery using the gal package.

```dart
class ImageScreen extends StatelessWidget {
  const ImageScreen({super.key});

  void saveImage() async {
    final directory = await getApplicationDocumentsDirectory();
    final path =
        '${directory.path}/image_${DateTime.now().millisecondsSinceEpoch}.png';
    final file = io.File(path);

    final imageData = StorageService().predictedImage;
    if (imageData != null) {
      await file.writeAsBytes(imageData);
      print('Image saved to $path');
      await Gal.putImage(file.path);
      print('Image saved to gallery!');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.black,
        foregroundColor: Colors.white,
        elevation: 3,
        shadowColor: Colors.black,
        leading: IconButton(
            onPressed: () {
              Navigator.pop(context);
            },
            icon: Icon(Icons.arrow_back_ios)),
      ),
      body: SafeArea(
        child: Column(children: [
          // Align(
          //   alignment: Alignment.topLeft,
          //   child: IconButton(
          //       onPressed: () => Navigator.pop(context),
          //       icon: const Icon(Icons.close)),
          // ),
          Expanded(
            child: PhotoView(
              imageProvider: MemoryImage(StorageService()
                  .predictedImage!), // Display the image from Uint8List
              backgroundDecoration: BoxDecoration(
                  color: Colors.black), // Optional: black background
            ),
          ),
          SizedBox(height: 10),
          TextButton(
              onPressed: () async {
                saveImage();
              },
              child: Text(
                "Save Image",
                style: TextStyle(color: const Color.fromARGB(255, 165, 144, 144)),
              )),
          SizedBox(height: 10),
        ]),
      ),
    );
  }
}
```

4) State Management with Cubits

The `ConversionCubit` is the core state manager, ensuring the app transitions smoothly between different states. The following states are defined:

- **ConversionInitialState**: Default state when the app is idle.
- **ConversionLoadingState**: Indicates an ongoing API call, displaying a loading spinner.
- **ConversionErrorState**: Triggered when an error occurs, showing an error message.
- **ConversionSuccess**: (Implicit) Managed through the preview of the generated image.

The **Cubit logic** for getPrediction ensures that:

- A request can only be triggered when the _canTriggerActions flag is true.
- State transitions occur predictably to provide a smooth user experience.
- Errors are handled gracefully, with messages displayed using a custom showSnackbarMessage.

```dart
class ConversionCubit extends Cubit<ConversionState> {
  ConversionCubit() : super(const ConversionInitialState());
  bool _canTriggerActions = true;

  Future<void> getPrediction(
      String sentence, int fontSize, int canvasWidth) async {
    if (!_canTriggerActions) return;
    _canTriggerActions = false;
    emit(const ConversionLoadingState());

    try {
      Uint8List? response =
          await ApiService().getPrediction(sentence, fontSize, canvasWidth);
      //   print(response.statusMessage);print(response.statusMessage);
      if (response == null) return;
      StorageService service = StorageService();
      service.predictedImage = response;
      if (service.predictedImage!.isNotEmpty) {
        print(service.predictedImage!);
        // print(response.data["message"].toString());
        // Navigator.push(context,
        //     MaterialPageRoute(builder: (context) => const PDFScreen()));
      } else {
        String message = "Response Image bytes are corrupted";
        print(message.toString());
        emit(ConversionErrorState(message));
      }
    } catch (e) {
      print(e.toString());
      emit(ConversionErrorState(e.toString()));
    }
    emit(const ConversionInitialState());
    _canTriggerActions = true;
  }
}
```

```dart
part of 'conversion_cubit.dart';

@immutable
abstract class ConversionState {
  const ConversionState();
}

class ConversionInitialState extends ConversionState {
  const ConversionInitialState();
}

class ConversionLoadingState extends ConversionState {
  const ConversionLoadingState();
}

class ConversionErrorState extends ConversionState {
  final String message;
  const ConversionErrorState(this.message);
}

```

5) Backend Communication

The `ApiService` class encapsulates backend communication with the following responsibilities:

- **POST Requests**: Sends typed text, font size, and canvas width as JSON payload to the backend API.
- **Image Retrieval**: Receives the generated handwritten image as raw bytes (`Uint8List`) for further processing.
- **Error Handling**: Differentiates between network errors (handled using `DioError`) and server-side errors (status codes other than 200).

The class uses a singleton pattern to ensure:

- Only one instance of the API handler is created.
- Centralized management of API calls for scalability and maintainability.

6) Storage Management

The `StorageService` class manages temporary storage for the generated image. Its responsibilities include:

- Holding the predicted image (`Uint8List`).
- Enabling smooth transitions between the **ConversionScreen** and **ImageScreen** by maintaining image state.

```dart
class ApiService {
  static final ApiService _instance = ApiService._internal();
  factory ApiService() => _instance;
  ApiService._internal();

  final _client = Dio(BaseOptions(
    baseUrl: "http://10.24.84.215:5000/predict",
    // baseUrl: "http://127.0.0.1:5000/predict",
  ));

  Future<Uint8List?> getPrediction(
      String sentence, int fontSize, int canvasWidth) async {
    try {
      final response = await _client.post(
        '',
        data: {
          'sentence': sentence,
          'font_size': fontSize,
          'canvas_width': canvasWidth,
        },
        options: Options(
          responseType: ResponseType.bytes, // Receive image as bytes
        ),
      );

      if (response.statusCode == 200) {
        print("data recieved");
        return response.data; // Returning the image data as Uint8List
      } else {
        throw Exception(
          "Error: ${response.statusCode} - ${response.statusMessage}",
        );
      }
    } on DioError catch (e) {
      throw Exception("Request failed with error: ${e.message}");
    }
  }
}
```

```dart
import 'dart:typed_data';

class StorageService {
  static final StorageService _instance = StorageService._internal();
  factory StorageService() => _instance;
  StorageService._internal();

  Uint8List? predictedImage = null;
}
```

2. Backend:

1) System Overview and Architecture

The backend is implemented using the Quart web framework, which handles HTTP POST requests and responds with dynamically generated images. Key components include:

- **Trained Pix2Pix Generator Model**: Converts input font-based word images to a handwritten-style format.
- **Image Preprocessing and Transformation Pipelines**: Handles the creation, resizing, and padding of input images to ensure compatibility with the generator.
- **Canvas Construction and Text Alignment**: Assembles individual word images into a properly aligned paragraph-like format.

The backend is designed to be modular, extensible, and efficient, ensuring low-latency responses for input text processing.

2) System Overview and Architecture

Step 1: API Request Handling

The /predict API endpoint accepts a **POST request** containing the following JSON keys:

- sentence: The input sentence to be converted into handwritten text.
- font_size: The desired font size for the generated text.
- canvas_width: The width of the output canvas.

Upon receiving a request:

- The JSON payload is validated to ensure required keys are present.
- The sentence is preprocessed to remove non-Hindi characters and split into individual words.

```python
app = Quart(__name__)
app = cors(app)

@app.route('/predict', methods=['POST'])
async def predict():
    data = await request.get_json()  # Retrieve JSON data from the request

    if not data or any(key not in data for key in ['sentence', 'font_size', 'canvas_width']):
        return jsonify({"error": "Request Body must contain 'sentence', 'font_size' and 'canvas_width'"}), 400

    sentence = data['sentence']
    font_size = data['font_size']
    canvas_width = data['canvas_width']

    try:
        # Generate image from the given word
        processed_image = place_words_on_canvas(font_size, sentence, canvas_width, debug=False)

        # Convert the processed image to bytes for response
        img_byte_arr = io.BytesIO()
        processed_image.save(img_byte_arr, format='PNG')
        img_byte_arr = img_byte_arr.getvalue()

        # Return the image in the response
        return Response(img_byte_arr, mimetype='image/png')

    except Exception as e:
        print(e)
        return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    app.run(debug=True, host="127.0.0.1", port=5000)
```

Step 2: Text-to-Image Conversion

For each word in the input sentence:

1. **Font-based Rendering**:
    - The word is rendered into an image using the **Pillow (PIL)** library with a predefined font (Amiko-Regular.ttf).
    - A bounding box is calculated for the word to crop unnecessary whitespace.
2. **Resizing and Padding**:
    - The cropped word image is resized while maintaining the aspect ratio to fit within a **256x256** pixel boundary.
    - Additional padding ensures consistent alignment on the canvas.
3. **Pix2Pix Handwriting Style Conversion**:
    - The preprocessed word image is normalized and converted into a PyTorch tensor.
    - The tensor is passed through the **Pix2Pix generator**, which is a Convolutional Neural Network trained on paired font-to-handwriting datasets.
    - The output tensor is converted back into an image, producing a handwritten-style version of the word.

```python
def text_to_image(text, font_path, height=100, padding=10):
    font = ImageFont.truetype(font_path, size=90, layout_engine=ImageFont.Layout.RAQM)
    temp_image = Image.new("L", (1, 1), color=255)
    draw = ImageDraw.Draw(temp_image)
    text_bbox = draw.textbbox((0, 0), text, font=font)
    text_width = text_bbox[2] - text_bbox[0]
    text_height = text_bbox[3] - text_bbox[1]
    image_width = text_width + 2 * padding
    image_height = height
    image = Image.new("L", (image_width, image_height), color=255)
    draw = ImageDraw.Draw(image)
    text_position = (padding, (image_height - text_height) // 2)
    draw.text(text_position, text, font=font, fill=0)
    return image

def resize_and_pad(image, target_size=(256, 256)):
    original_size = image.size
    ratio = min(target_size[0] / original_size[0], target_size[1] / original_size[1])
    new_size = (int(original_size[0] * ratio), int(original_size[1] * ratio))
    resized_image = image.resize(new_size, Image.Resampling.LANCZOS)
    new_image = Image.new("L", target_size, color=255)
    top_left_x = (target_size[0] - new_size[0] + 10) // 2  # Horizontal center
    top_left_y = (target_size[1] - new_size[1]) // 2  # Vertical center
    new_image.paste(resized_image, (top_left_x, top_left_y))
    return new_image

def createImg(word, font_path):
    image = text_to_image(word, font_path)
    image = resize_and_pad(image)
    return image
```

```python
def convertImg(img: Image) -> Image:
    img = np.asarray(img)
    img = transform_only_input(image=img)["image"]
    img = img.unsqueeze(0)
    x = img.to(DEVICE)
    with torch.amp.autocast('cuda'):
        y_fake = gen(x)
    y_fake_np = (y_fake[0].detach().cpu().numpy()[0] * 255).clip(0, 255).astype(np.uint8)
    return y_fake_np

def crop_and_resize(img_array, target_height):
    # Step 1: Find the bounding box of non-white regions
    # Assuming white is represented as 255 and black text as 0
    _, binarized_image = cv2.threshold(img_array, 128, 255, cv2.THRESH_BINARY)  # Adjust threshold as needed

    coords = np.column_stack(np.where(binarized_image < 255))  # Find non-white pixels
    if coords.size == 0:
        # If there is no black content, return the original or a blank target-sized image
        return np.full((target_height, int(target_height * img_array.shape[1] / img_array.shape[0])), 255, dtype=np.uint8)

    # Get the bounding box for the non-white pixels
    y_min, x_min = coords.min(axis=0)
    y_max, x_max = coords.max(axis=0)

    # Crop the image using the bounding box
    cropped_image = img_array[y_min:y_max + 1, x_min:x_max + 1]

    # Step 2: Resize while maintaining aspect ratio based on target height
    h, w = cropped_image.shape
    aspect_ratio = w / h
    target_width = int(target_height * aspect_ratio)

    # Resize the cropped image to the target height
    resized_image = cv2.resize(cropped_image, (target_width, target_height), interpolation=cv2.INTER_AREA)

    return resized_image

def getWordImage(word, max_height=256):
    min_height = round((max_height * 200) / 256)
    image = createImg(word, font_path)
    image = convertImg(image)
    contains_matras_above_shirorekha = any(char in ['ि', 'ै', 'ी', 'े', 'ू'] for char in word)
    image = crop_and_resize(image, max_height if contains_matras_above_shirorekha else min_height)
    return image
```

Step 3: Paragraph Assembly

Once all words are converted into handwritten-style images:

1. **Canvas Initialization**:
    - An initial blank canvas is created with the specified canvas_width and a default height.
    - Dynamic resizing is enabled to expand the canvas as needed.
2. **Word Placement**:
    - Words are placed row by row with calculated x and y offsets to mimic natural paragraph alignment.
    - Words containing **matras (diacritical marks)** above the Shirorekha (Hindi script's horizontal line) are vertically adjusted for proper placement.
    - Randomized spacing between words and lines adds variability, replicating real-world handwriting characteristics.
3. **Dynamic Canvas Adjustment**:
    - If a word exceeds the current line width, it is placed on the next line.
    - The canvas height is extended dynamically to accommodate additional lines.

Step 4: Response Preparation

- The final canvas containing the handwritten-style text is converted into a **PNG image**.
- The image is sent as a **binary response** with the appropriate MIME type (image/png).

3) Key Functions and Utilities

- **text_to_image**: Converts a word into a grayscale image using a specified font and height.
- **resize_and_pad**: Resizes the input word image to fit within a 256x256 frame, preserving its aspect ratio, and pads it with a white background.
- **convertImg**: Applies the Pix2Pix generator model to convert the preprocessed image into a handwritten style.
- **crop_and_resize**: Crops extraneous whitespace around the word and adjusts the height of the word image based on the presence of matras.
- **place_words_on_canvas**: Aligns word images into a paragraph-like format, dynamically resizing the canvas as needed.

```python
def filter_hindi_letters(sentence):
    # Regular expression to match only Hindi letters and spaces
    hindi_letters_only = re.sub(r'[^\u0900-\u097F\s]+', '', sentence)
    return hindi_letters_only

def place_words_on_canvas(font_size, sentence, canvas_width, debug=False):

    words = filter_hindi_letters(sentence).split()
    word_images = []
    for word in words:
        word_images.append(np.asarray(getWordImage(word, max_height=font_size)))

    # Start with an initial canvas height

    def setRatio(line_height, num):
        return round(line_height * (num / 256))


    line_height = font_size
    x_offset, y_offset = setRatio(line_height, 60), setRatio(line_height, 60)
    idx = 0

    canvas = Image.new('L', (canvas_width, line_height + setRatio(line_height, 120)), 255)  # 'L' mode for grayscale

    for word_img in word_images:
        contains_matras_above_shirorekha = any(char in ['ि', 'े', 'ी', 'ै', 'ू'] for char in words[idx])
        word_pil = Image.fromarray(word_img)
        # if debug:
        #     word_pil = add_green_borders(word_pil)
        word_width, word_height = word_pil.size

        # Check if the word exceeds the current canvas width and needs a new line
        if x_offset + word_width > canvas_width:
            x_offset = setRatio(line_height, 60)
            y_offset += line_height + np.random.randint(setRatio(line_height, 30), setRatio(line_height, 60))

            # Dynamically expand the canvas height if the new line goes beyond the current canvas height
            if y_offset + line_height > canvas.height:
                new_height = canvas.height + line_height + np.random.randint(setRatio(line_height, 30), setRatio(line_height, 60)) + setRatio(line_height, 56)
                new_canvas = Image.new('L', (canvas_width, new_height), 255)
                new_canvas.paste(canvas, (0, 0))
                canvas = new_canvas

        # Paste the word image onto the canvas
        canvas.paste(word_pil, (x_offset, y_offset if contains_matras_above_shirorekha else y_offset + setRatio(line_height, 56)))
        x_offset += word_width + np.random.randint(setRatio(line_height, 45), setRatio(line_height, 45) + (word_width * 0.1))

        idx += 1
    return canvas
```

4) Error Handling

The system includes robust error handling to:

- Validate input payloads.
- Catch and log exceptions during image generation or text alignment processes.
- Return meaningful error messages to the client with appropriate HTTP status codes.

Software Requirements

1. **Development Tools**

    1) **VS Code**

    2) **Android Studio**

    3) **Flutter**

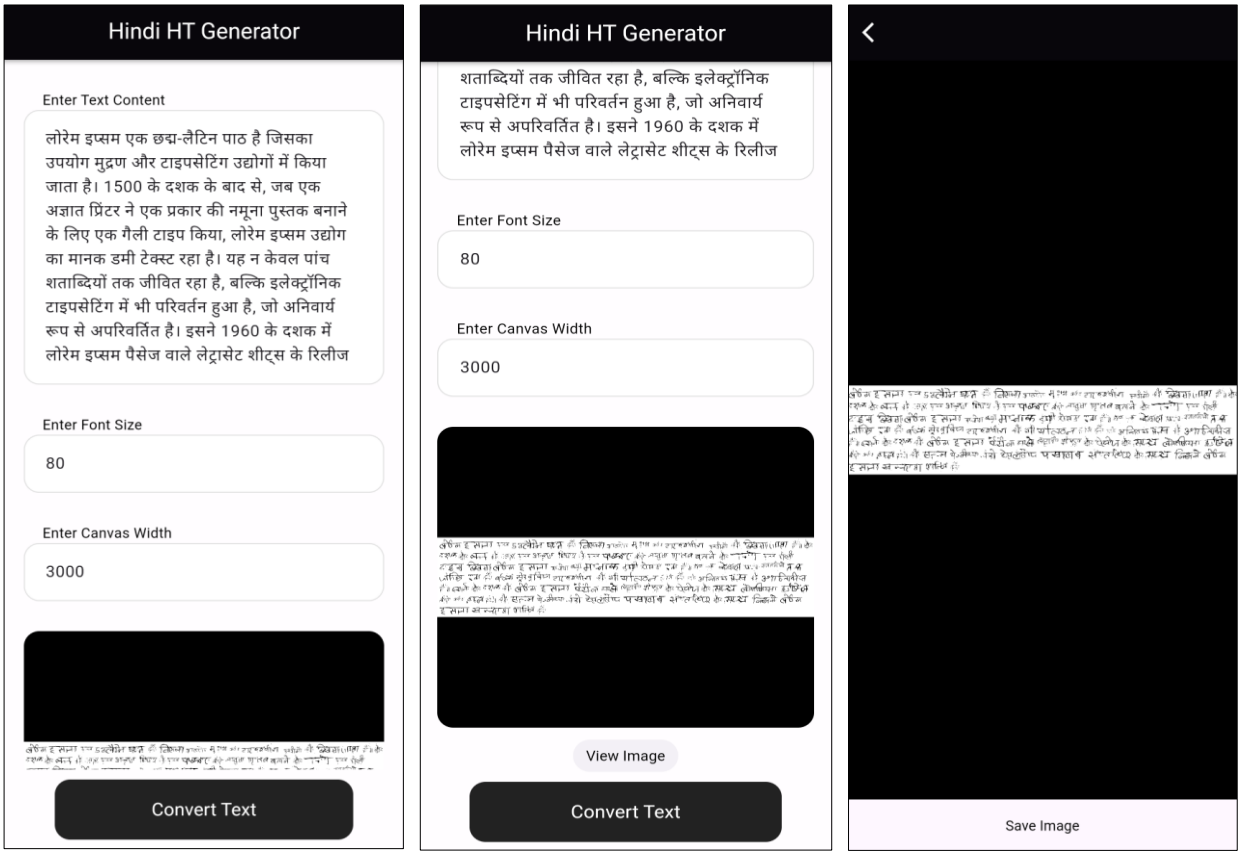
2. **Libraries and Dependencies**

- **Frontend Libraries**:
    - flutter_bloc: ^8.1.1
    - bloc: ^8.1.0
    - dio: ^4.0.6
    - photo_view: ^0.15.0
    - gal: ^2.3.0
    - path_provider: ^2.1.5

- **Backend Libraries**:

    - OS
    - Regex
    - OpenCV
    - Numpy
    - PyTorch
    - Pillow

Details of the application



This app is called **Hindi HT Generator**. It takes Hindi text that you type and transforms it into a beautifully styled handwritten text image. You can adjust the text size and layout, see a preview, and even save the final image to your phone.

First Screen: Input Your Text

- **What You See**: At the top of this screen, there's a large box where you can type or paste any Hindi text you want. Below that, there are two smaller boxes where you can adjust the font size (how big the text should be) and the canvas width (how wide the text should appear in the image).
- **What You Do**:
    1. Type your Hindi text in the big box.
    2. Decide how large the letters should be and type that number into the "Enter Font Size" box (e.g., 80 for bigger letters, 50 for smaller ones).
    3. Choose the width of the canvas (e.g., 3000 for a wide layout or smaller for a narrower one).
    4. Press the **Convert Text** button to generate your handwritten image.
- **What Happens Next**: Once you press the button, the app takes your text and the settings you entered and uses them to create a realistic handwritten image. The generated image appears right below the button.

Second Screen: Preview Your Handwriting

- **What You See**: This screen shows the result of your input—a handwritten version of the Hindi text you typed earlier. The app automatically arranges the text on a canvas according to the font size and width you selected. You can see the handwriting style in a black-bordered container, making it easy to read.
- **What You Do**:
    1. Look at the preview of your handwritten text.
    2. If you're happy with it, click the **View Image** button to see a full-sized version.
    3. If something doesn't look right, you can go back to the first screen and adjust the font size or canvas width, then try again.

Third Screen: Full Image and Save Option

- **What You See**: Here, the full handwritten text image is displayed in detail. It looks just like handwritten text but created digitally based on your input. This screen lets you scroll to see the entire text if it's too long to fit on one screen.
- **What You Do**:
    1. Carefully review the full-sized image.
    2. If you like it and want to keep it, press the **Save Image** button. The app will save this image directly to your phone, so you can use it anywhere—share it with friends, include it in documents, or print it out.

How It Works Behind the Scenes (Simplified for Audience)

1. When you press the **Convert Text** button, the app sends your input to a specialized program in the background that knows how to "draw" Hindi text in a handwriting style.
2. The program processes your text, applies the font size and layout settings, and creates a digital image that looks like real handwriting.
3. The app then displays this image for you to preview or save.

Conclusion

The **Hindi Handwriting Text Generator** is a robust, scalable, and user-friendly application designed to bridge the gap between digital Hindi text and handwritten-style outputs. With its modular architecture, efficient state management, and thoughtful UI design, the application provides a seamless experience for users while offering significant potential for future expansions.

The backend efficiently combines text preprocessing, image transformation, deep learning-based generation, and canvas construction to provide a seamless API for generating realistic handwritten text. By leveraging the Pix2Pix model, it ensures high-quality output that closely mimics real handwriting styles, while the modular structure supports extensibility for future enhancements.

References

- Flutter Documentation
  https://docs.flutter.dev

- Android Studio Documentation:
  https://source.android.com/docs

- flutter bloc Documentation:
  https://pub.dev/packages/flutter_bloc