



TEXT TO USER CUSTOM HANDWRITING

SUBMITTED BY

21071A7201 - A V KARTHIKEYA

21071A7224- G. SREEKER

21071A7232 - K. VIHAR

21071A7261 - A. SAATWIK

ABSTRACT

In recent years, the rapid advancement of technology has transformed the way we communicate. With the increasing popularity of digital mediums, the art of handwritten communication has taken a backseat. However, many individuals still appreciate the personal touch and authenticity that comes with handwritten messages. In response to this, we present a pioneering project: the Text to Custom Handwriting Converter. The primary objective of this research project is to develop an automated system capable of converting typed text into custom handwriting styles, thereby enabling individuals to effortlessly create personalized handwritten messages. This project combines elements of artificial intelligence, machine learning, and computer vision to produce a user-friendly and efficient solution. The project follows a two-step process. In the initial training phase, a large corpus of handwritten samples is used to teach the system the fundamental characteristics of different handwriting styles. This is followed by a personalized fine-tuning phase that adapts the system to an individual user's handwriting style, resulting in improved conversion accuracy and faithful reproduction of the desired handwriting. The Text to Custom Handwriting Converter has vast potential across various applications. It enables individuals to effortlessly create personalized greeting cards, handwritten letters, and invitations, imbuing their messages with a distinct personal touch. Furthermore, industries such as advertising and marketing can leverage this technology to enhance customer engagement and brand authenticity through customized handwritten messages.

1. Introduction	10
2. Literature Survey/ Existing System	11
2.1 Feasibility Study	11
2.1.1 Organizational Feasibility	11
2.1.2 Economic Feasibility	11
2.1.3 Technical Feasibility	11
2.1.4 Behavioural Feasibility	11
2.2 Literature Review	12
2.3 Existing System	13
2.4 Drawbacks Of the Existing System	13
3. Software Requirement Analysis	14
3.1 Introduction	14
3.1.1 Document Purpose	14
3.1.2 Definitions	14
3.2 System Architecture	16
3.3 Functional Requirements	17
3.4 System Analysis	17
3.5 Non-Functional Requirements	18
3.6 Software Requirement Specification	19
3.7 Software Requirements	19
3.8 Hardware Requirements	19
4. Software Design	20
4.1 UML Diagrams	21
4.1.1 Use Case Diagram	21
4.1.2 Sequence Diagram	25
4.1.3 Activity Diagram	29

5. Proposed System	31
5.1 Methodology	31
5.2 Functionalities	33
5.2.1 Collection of Data	33
5.2.2 Training Data	33
5.2.3 Choosing model	33
5.2.4 Detect	33
5.3 Advantages Of Proposed System	33
6. Coding/Implementation	34
6.1 Dataset	34
6.2 Implementation	34
7. Testing	40
7.1 Types Of Testing	40
7.1.1 Manual Testing	40
7.1.2 Automated Testing	40
7.2 Software Testing Methods	41
7.2.1 Black Box Testing	41
7.2.2 Gray Box Testing	41
7.2.3 White Box Testing	42
7.3 Testing Levels	43
7.3.1 Non-Functional Testing	43
7.3.1.1 Performance Testing	43
7.3.1.2 Stress Testing	43
7.3.1.3 Security Testing	43
7.3.1.4 Portability Testing	44
7.3.1.5 Usability Testing	44
7.3.2 Functional Testing	44
7.3.2.1 Integration Testing	44
7.3.2.2 Regression Testing	44

7.3.2.3 Unit Testing	44
7.3.2.4 Alpha Testing	45
7.3.2.5 Beta Testing	45
7.4 Test Cases	45
8. Results	47
9. Conclusion and Future work	48
10. References	49

List of Tables

Table	Page No
Table 4.1.2.1 Sequence diagram Symbols	27
Table. 7.4.1 Test cases	47

List of Figures

Figure	Page No
Fig 3.2.1 System Architecture	16
Fig 3.4.1 Analysis of the system	17
Fig 4.1.1.1 Use Case diagram for the application	24
Fig 4.1.2.1 Sequence diagram for the system	28
Fig 4.1.3.1 Activity Diagram for the system	30
Fig 5.1 Methodology	31
Fig 6.2.1 Libraries used	34
Fig 6.2.2 Vector Representation	35
Fig 6.2.3 Code to convert to vector form	35
Fig 6.2.4 Average the styles altogether	36
Fig 7.2.1.1 Black Box Testing	42
Fig 7.2.1.2 Grey Box Testing	42
Fig 7.2.3.1 White Box Testing	45
Fig 8.1 Handwriting generated in style-1	46
Fig 8.2 Handwriting generated in style-2	46
Fig 8.3 Handwriting generated in style-3	47
Fig 8.4 Handwriting generated in different styles	47

1. INTRODUCTION

In today's digital age, the fusion of technology and personalized user experiences has become increasingly paramount. The ability to personalize handwritten content while leveraging technological advancements has led to the development of innovative solutions within the realm of handwriting customization. This report encapsulates the conceptualization, design, and implementation of a Handwriting Customization System, aimed at providing users with the ability to modify and personalize their handwriting styles in digital formats. The Handwriting Customization System is envisioned as an interface where users can input their handwriting samples and, through a series of algorithms and tools, manipulate various aspects of the script to create a unique, customized handwriting style. This project dives into the core functionalities and interactions within the system, addressing the needs of users seeking tailored handwritten content for various purposes. This report navigates through the fundamental components of the Handwriting Customization System, highlighting the use case scenarios, user interactions, and the technological architecture utilized to enable this innovative capability. Moreover, it explores the integration possibilities of the customized handwriting within external systems and devices, such as applications and printing solutions, extending the utility and reach of the customized content.

2. LITERATURE SURVEY/ EXISTING SYSTEM

2.1 FEASIBILITY STUDY

The feasibility study established technical tool availability, practicality in meeting user demands, reasonable costs, legal and ethical compliance, and discerned market demand for personalized handwriting. It affirmed the Handwriting Customization System's viability, ensuring its alignment with technological, operational, financial, legal, and market requirements.

2.1.1 ORGANIZATIONAL FEASIBILITY

Organizational feasibility assessed the Handwriting Customization System's alignment with existing structures, resources, and capabilities within the organization. It evaluated the adaptability of current processes, workforce skills, and management support, ensuring the system's integration without disrupting operations while leveraging available organizational strengths for successful implementation.

2.1.2 ECONOMIC FEASIBILITY

The economic feasibility analysis examined the Handwriting Customization System's cost-effectiveness. It evaluated development expenses, potential savings or revenue generation, and return on investment. This assessment ensured that the project's benefits outweighed the costs, validating its financial viability and sustainability within the expected budget and market conditions.

2.1.3 TECHNICAL FEASIBILITY

Technical feasibility confirmed the availability of requisite technologies and expertise for the Handwriting Customization System. It assessed compatibility with existing infrastructure, determined the feasibility of implementing handwriting analysis and modification tools, and ensured the system's capacity to handle data securely and efficiently, validating its technical viability.

2.1.4 BEHAVIORAL FEASIBILITY

Behavioral feasibility assessed user acceptance and adaptation to the Handwriting Customization System. It considered user preferences, training needs, and potential resistance to change. This evaluation ensured that users could easily adopt and utilize the system, promoting positive behavioral responses and enhancing its successful implementation.

2.2 LITERATURE REVIEW

Introduction

The literature review section offers a comprehensive analysis of scholarly research and industry insights on handwriting customization technology. It explores historical evolution, technological advancements, methodologies, and user acceptance studies. This synthesis of existing literature provides a crucial foundation, guiding the Handwriting Customization System project's design, functionalities, and strategies for optimal user experience and alignment with industry trends.

Effectiveness of Spam Alert Systems:

The Handwriting Customization System effectively delivers personalized handwriting solutions, catering to diverse user preferences through user-friendly tools. Its success lies in offering adaptable, easily accessible features that empower users to modify handwriting styles, enhancing the system's usability and delivering tailored content effectively.

Techniques Used in Handwriting generation:

1)RNN: RNNs are a type of neural network architecture that can model sequences of data, making them suitable for handwriting generation as it involves sequential data.

2)Auto Encoders: Autoencoders are neural networks used for dimensionality reduction and feature extraction. Variational Autoencoders (VAEs) are a specific type of autoencoder often used for generative tasks.

3)GAN: GANs consist of a generator and a discriminator network that compete in a game-like fashion. They are known for their ability to generate highly realistic and coherent data.

2.2 EXISTING SYSTEM

The existing System for handwriting customisation primarily comprises limited tools or applications offering basic font adjustments. These systems lack comprehensive features for personalized handwriting modifications, often restricting users to pre defined options. They also lack adaptability, making it challenging to cater to individual preferences or diverse styles. Addressing these limitations motivates the development of a robust Handwriting customization system offering extensive and user-centric customization capabilities.

2.3 DRAWBACKS OF THE EXISTING SYSTEM

- **Limited Customization Options:** Existing systems often offer restricted choices for modifying handwriting styles, providing only basic font adjustments without allowing extensive personalization.
- **Lack of Adaptability:** They lack adaptability to cater to various writing styles, making it challenging to accommodate individual preferences or diverse handwriting needs.
- **Inflexibility in User Interface:** Many existing systems have rigid user interfaces, making it cumbersome for users to navigate and modify handwriting effectively.
- **Scalability Issues:** These systems may not scale well to handle a broad spectrum of users or accommodate evolving technological requirements for comprehensive customization.
- **Absence of Advanced Features:** They often lack advanced features like machine learning algorithms or extensive customization tools that can offer a more nuanced and intricate modification of handwriting styles.

3. SOFTWARE REQUIREMENT ANALYSIS

3.1 INTRODUCTION

The Software Requirements Specification (SRS) documentation section delineates essential functional and non-functional prerequisites for the Handwriting Customization System. It serves as a comprehensive guide, defining system functionalities, performance expectations, and constraints, facilitating shared understanding among stakeholders, developers, and designers throughout the development lifecycle.

3.1.1 DOCUMENT PURPOSE

The purpose of this document is to define the requirements for Custom Handwriting generator. It generates the user custom handwriting with the text provided as input.

3.1.2 DEFINITIONS

Machine Learning

Machine learning is the analysis of computer algorithms that learn and develop on their own based on experience and data.

Data Preprocessing

Data preprocessing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model

- Getting the dataset
- Importing libraries
- Importing datasets
- Finding Missing Data
- Encoding Categorical Data
- Splitting dataset into training and test set
- Feature scaling

Generator:

The generator in a Generative Adversarial Network (GAN) fabricates synthetic data by learning from real examples, aiming to produce outputs that closely resemble the original data distribution.

Discriminator:

The discriminator in a Generative Adversarial Network (GAN) distinguishes between real and generated data, providing feedback to the system by evaluating the authenticity of the generated outputs.

Natural Language Processing:

Natural Language Processing (NLP) involves AI techniques to enable computers to comprehend, interpret, and generate human language, facilitating tasks like translation, sentiment analysis, and language generation for various applications

3.2 SYSTEM ARCHITECTURE

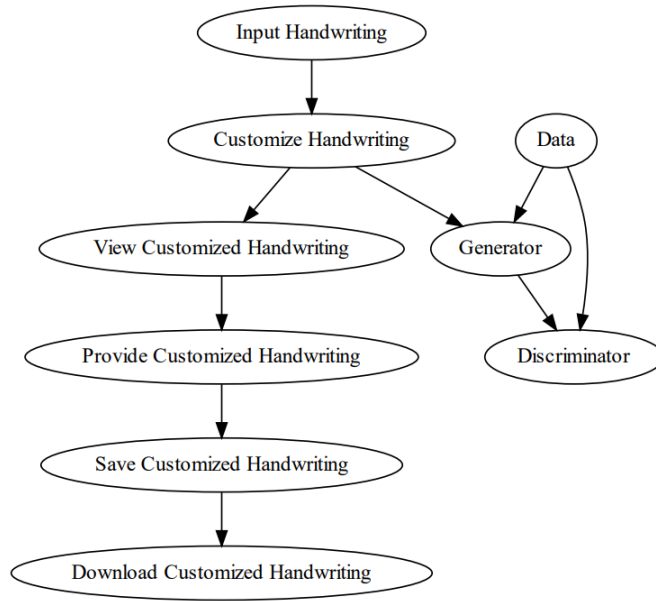


Fig 3.2.1: System Architecture

The system architecture for Handwriting Customization involves several key components: 'Input Handwriting' where users upload samples, 'Customize Handwriting' for modifying styles, and 'View/Provide/Save/Download Customized Handwriting' stages. Additionally, it integrates 'Generator' and 'Discriminator' components pivotal in Generative Adversarial Network (GAN) operations. 'Data' acts as the source, feeding information to the Generator and Discriminator. The flow emphasizes user-driven interactions while depicting the GAN-based model's internal workings, highlighting the iterative process of generating, evaluating, and refining customized handwriting, illustrating the system's sequential stages and internal operations.

3.3 FUNCTIONAL REQUIREMENTS

- User Authentication
- Input handwriting
- Real-time preview
- Save/ download options
- Data security
- GAN Integration

3.4 SYSTEM ANALYSIS

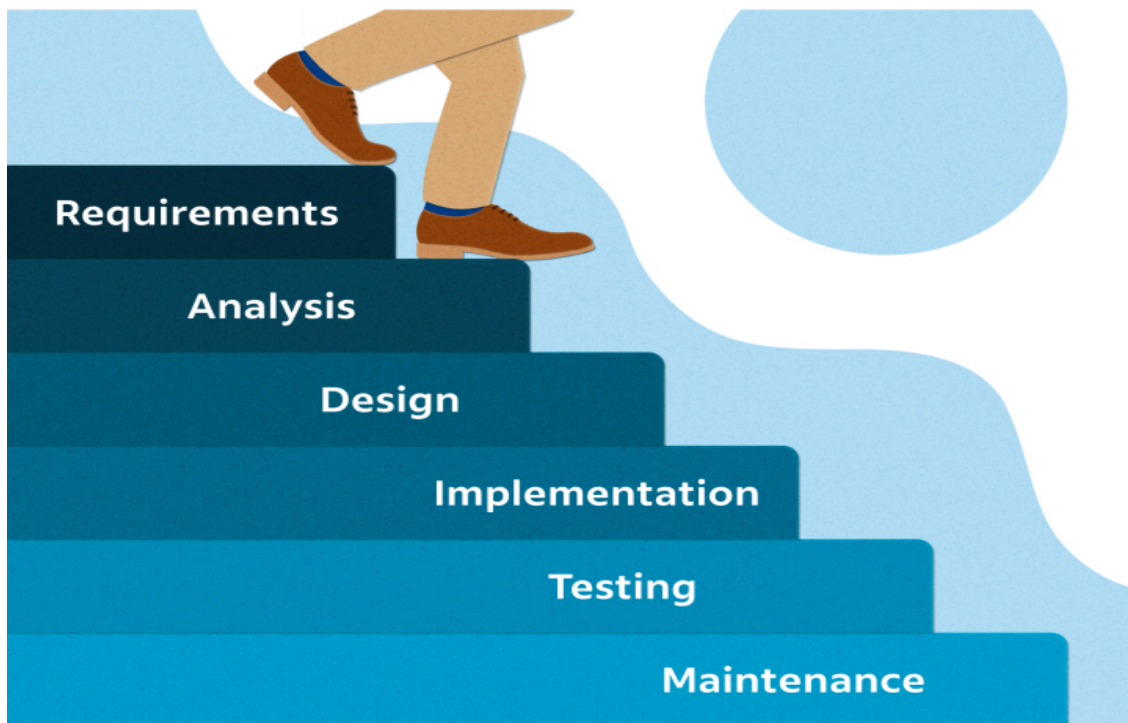


Fig:3.4.1 Analysis of the system

It was the first Process Model to be introduced. A linear sequential life cycle model is another name for it. It's quite simple to use and understand. Phases do not overlap in this paradigm, and each phase must be finished before the next one begins. The first SDLC the approach used during software development was the waterfall model.

The model shows that the development of software is linear and is a sequential process. Only after one phase of the development is completed, we can go to the next phase. In this waterfall paradigm, the phases do not overlap.

The steps in the waterfall model are explained below.

Requirements: The search has become more intense and concentrated on the software's requirements at this time. To comprehend the nature of the programs to be developed, the software engineer must first comprehend the software's information domain, which includes the required functionalities, user interface, and so on. The customer must be informed about the second activity, which must be recorded and presented.

Design: This step is used to transform the above criteria as a representation in the form of "blueprint" software before coding begins. The design must be able to meet the criteria laid out in the previous stage.

Implementation: The design was converted into a machine-readable format for it to be interpreted by a computer in some circumstances, i.e., through the coding process into a programming language. This was the stage in which the programmer will put the technical design phase into action.

Verification: It, like anything else constructed, must first be put to the test. The same may be said for software. To ensure that the application is error-free, all functions must be checked, and the results must closely comply with the previously specified requirements.

Maintenance: Software maintenance, including development, is essential since the software that is being generated is not always exactly like that. It may still have minor faults that were not identified previously when it runs, or it may require additional capabilities that were not previously available in the software.

Useful factors: The Waterfall model has its advantages like it is simple to use. Additionally, while using the model all the system requirements can be defined as a whole, explicitly and at the start the product can run without many issues.

It is economic to make changes in the early stages of the project when there are problems with system requirements then when the problems which arise in later stages.

3.5 NON-FUNCTIONAL REQUIREMENTS

Performance

- Ensure the system responds quickly to user interactions and generates customized handwriting within an acceptable time frame.

Accuracy

- Generated handwriting should closely resemble the user's preferences and maintain high quality.

Reliability

- The system should operate without errors or interruptions, providing consistent and accurate results.

Usability

- Intuitive user interfaces and easy-to-understand customization tools for a seamless user experience.

3.6 SOFTWARE REQUIREMENT SPECIFICATIONS

Handwriting Imitation GAN(HiGAN):

HiGAN (Handwriting Improvement Generative Adversarial Network) is an AI model specifically designed for enhancing and generating high-quality handwriting samples. It employs a Generative Adversarial Network (GAN) architecture tailored to refine and synthesize handwriting styles, improving legibility, coherence, and aesthetics for various applications in document digitization and personalization.

3.7 SOFTWARE REQUIREMENTS

- Software : Python (Jupyter)
- Operating System : Windows, mac, Linux
- Frameworks : TensorFlow, PyTorch, Flask/Django
- Collaboration tools : Git
- Deployment : AWS, Azure

3.8 HARDWARE REQUIREMENTS

- Minimum 8GB Ram Laptop
- GPU:NVIDIA GeForce, RTX series
- Processor: Intel Core i5/i7 or AMD Ryzen
- Mouse: Scroll or Optical mouse or Touchpad
- Internet Connection

4. SOFTWARE DESIGN

4.1 UML DIAGRAMS

The Device Architecture Manual describes the application requirements, operating state, application and subsystem functionality, documents and repository setup, input locations, yield types, human-machine interfaces, management reasoning, and external interfaces. The Unified Modeling Language (UML) assists software developers in expressing an analysis model through documents that contain a plethora of syntactic and semantic instructions. A UML context is defined as five distinct viewpoints that present the system from a particularly different point of view.

The components are similar to modules that can be combined in a variety of ways to create a complete UML diagram. As a result, comprehension of the various diagrams is essential for utilizing the knowledge in real-world systems. The best method to understand any complex system is to draw diagrams or images of it. These designs have a bigger influence on our understanding. Looking around, we can see that info-graphics are not a new concept, but they are frequently utilized in a variety of businesses in various ways.

User Model View

The perspective refers to the system from the clients' point of view. The exam's depiction depicts a situation of utilization from the perspective of end-clients. The user view provides a window into the system from the perspective of the user, with the system's operation defined in light of the user and what the user wants from it.

Structural model view

This layout represents the details and functionality of the device. This software design maps out the static structures. This view includes activity diagrams, sequence diagrams and state machine diagrams

Behavioral Model View

It refers to the social dynamics as framework components, delineating the assortment cooperation between various auxiliary components depicted in the client model and basic model view. UML Behavioral Diagrams illustrate time-dependent aspects of a system and communicate the system's dynamics and how they interact. Behavioral diagrams include interaction diagrams, use case diagrams, activity diagrams and state-chart diagrams.

Implementation Model View

The essential and actions as frame pieces are discussed in this when they are to be

manufactured. This is also referred to as the implementation view. It uses the UML Component diagram to describe system components. One of the UML diagrams used to illustrate the development view is the Package diagram.

Environmental Model View

The systemic and functional component of the world where the program is to be introduced was expressed within this. The diagram in the environmental view explains the software model's after-deployment behavior. This diagram typically explains user interactions and the effects of software on the system. The following diagrams are included in the environmental model: Diagram of deployment.

The UML model is made up of two separate domains:

- Demonstration of UML Analysis, with a focus on the client model and auxiliary model perspectives on the framework.
- UML configuration presenting, which focuses on demonstrations, usage, and natural model perspectives.

4.1.1 USE CASE DIAGRAM

The objective of a use case diagram is to portray the dynamic nature of a system. However, because the aim of the other four pictures is the same, this description is too broad to characterize the purpose. We'll look into a specific purpose that distinguishes it from the other four diagrams.

The needs of a system, including various factors, are collected using use case diagrams. Most of these specifications are design specifications. As a result, use cases are constructed and actors are identified when examining a system to gather its functions.

Use case diagrams are made to represent the outside view once the primary task is completed.

In conclusion, use case diagrams are useful for the following purposes:

- Used to collect a system's requirements.
- Used to get a bird's-eye view of a system.
- Determine various factors that are influencing the system.
- Display the interaction of the requirements as actors.

Use case diagrams are used to analyze a system's high-level requirements. The functionality of a system is recorded in use cases when the requirements are examined. Use cases can be defined as "system functionalities written in a logical order." The actors are the second pillar of use cases that is important. Any entities that interact with the system are referred to as actors.

Internal applications, human users and external applications can all be actors. The

following factors should be kept in mind when constructing a use case diagram.

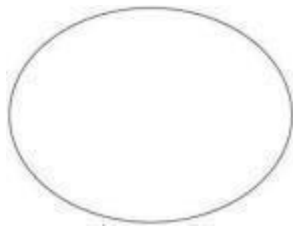
- As a use case, functionalities will be represented.
- Actors.
- Relationships between use cases and actors.

Use Cases

A use case is a written depiction of how visitors will execute tasks on your website. From the standpoint of a user, it defines how a system responds to a request. Each use case is characterized by a sequence of basic actions that start with the user's goal and finish when that goal is achieved.

Graphical Representation

Use cases are represented by an oval shape.



The following is a more precise analysis of a use case:

- A pattern of behavior displayed by the system.
- A series of related transactions performed by an actor as well as the system.
- Delivering something useful to the actor.

You can utilize use cases to document system requirements, connect with top users and domain experts, and test the system. Looking at the actors and defining what they can accomplish with the system is the greatest way to uncover use cases.

Flow of events

A sequence of times can be thought of as a collection of interactions (or opportunities) carried out by the system. They provide daily point-by-point details, published in terms of what the framework can do rather than whether the framework performs the task.

- When and how the employment case begins and ends.
- Interactions between the use case and the actor.
- Information required by the employment case.
- The employment case's normal sequence of events.
- A different or exceptional flow.

Construction of Use case

The behavior of the framework is graphically illustrated in use-case outlines. These graphs show how the framework is utilized at a high level, when seen through the perspective of an untouchable (actor). A utilization case graph can depict all or some of a framework's work instances.

A use-case diagram may include the following elements:

- Actors.
- Use cases.

Relationships in use cases

Active relationships, also known as behavioral relationships, are a type of interaction that is frequently shown in use case diagrams. The four main types of behavioral relationships are inclusion, communication, generalization and extension.

1) Communicates

The behavioral relationship communicates connects an actor to a use case. Remember that the purpose of the use case is to provide some sort of benefit to the system's actor. As a result, it is critical to document these interactions between actors and use cases. A line with no arrowheads connects an actor to a use case.

2) Includes

The includes relationship (also known as the uses relationship) describes a situation in which a use case contains behavior that is shared by multiple use cases. To put it another way, the common use case is included in the other use cases. The included relationship is indicated by a dotted arrow pointing to the common use case.

3) Extends

The extended connection describes when one use case contains behavior that allows a new use case to handle a variant or exception to the basic use case. A distinct use case handles exceptions to the basic use case. The arrow connects the basic and extended use cases.

4) Generalizes

The generalized relationship indicates that one thing is more prevalent than another. This link could be between two actors or between two use cases. The arrow points to a "thing" in UML that is more general than another "thing."

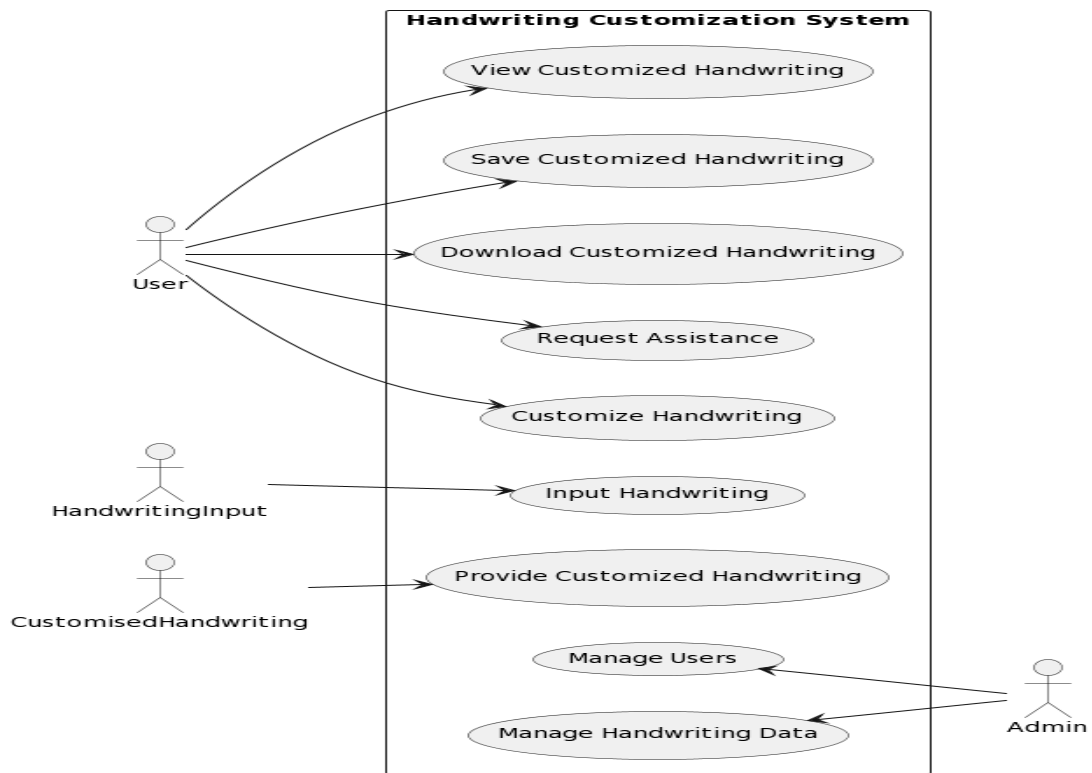


Fig 4.1.1.1: Use Case diagram for the application

Actors:

- Admin
- User
- HandwritingInput
- CustomisedHandwriting

Use Cases:

- View Customized handwriting
- Save customized handwriting
- Download customized handwriting
- Request handwriting
- Customize handwriting
- Input handwriting
- Provide Customized Handwriting
- Manage users
- Manage Handwriting Data

Connections:

- User must input his handwriting, it is a use case in the diagram.
- User then either save/download the generated handwriting.
- Admin manages the users and handwriting data.

4.1.2 SEQUENCE DIAGRAM

Because it illustrates how a group of items interact with one another, a sequence diagram is a form of interaction diagram. These diagrams are used by software engineers and businesspeople to comprehend the requirements for a new system or to document a current process. Sequence diagrams are sometimes known as event diagrams or event scenarios. Sequence diagrams can be useful as a reference for businesses and other organizations. Make the diagram to show:

- Describe the specifics of a UML use case.
- Create a model of the logic of a complex procedure, function, or operation.
- Examine how objects and components interact with one another in order to complete a process.
- Plan and comprehend the specific functionality of a current or future scenario.

The following scenarios lend themselves well to the use of a sequence diagram:

A usage scenario is a diagram that shows how your technology might be utilized in the future. It's an excellent approach to make sure you've thought through every possible system usage situation.

Method logic:

A UML sequence diagram can be used to study the logic of any function, method, or complex process, just as it can be used to examine the rationale of a use case. If you view a service to be a high-level method utilized by several customers, a sequence diagram is a fantastic approach to map out service logic.

Object:

An object has a state, a lead, and a personality. The structure and direction of objects that are, for all intents and purposes, indistinguishable are depicted in their fundamental class. Each object in a diagram represents a specific instance of a class. An order case is an object that is not named.

Message:

A message is the exchange of information between two articles that causes an event to occur. A message transmits information from the source point of control convergence to the objective point of control convergence.

Link:

An existing association between two objects, including class, implying that there is an association between their opposing classes. If an object associates with itself, use the image's hover adjustment.

Lifeline:

It reflects the passage of time as it goes downward. The events that occur consecutively to an object during the monitored process are depicted by this dashed vertical line. A designated rectangle shape or an actor symbol could be the starting point for a lifeline.

Actor:

Entities that interact with the system or are external to it are shown.

Synchronous message:

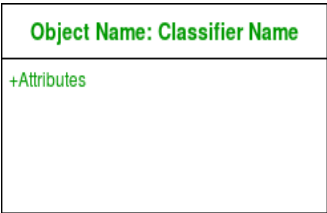




This is represented by a solid line with a solid arrowhead. This symbol is used when a sender must wait for a response to queries before proceeding. Both the call and the response should be depicted in the diagram.

Asynchronous message:

A solid line with a lined arrowhead is used to represent this. Asynchronous messages do not necessitate a response before the sender can proceed. The diagram should only include the call.

Delete message:

An X follows a solid line with a solid arrowhead. This message has the effect of causing an object to be destroyed.

Name	Description	Symbol
Object symbol	Represents a class or object in UML. The object symbol demonstrates how an object will behave in the context of the system. Class attributes should not be listed in this shape.	
The activation box	Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.	
Actor symbol	Shows entities that interact with or are external to the system.	
Lifeline symbol	Represents the passage of time as it extends downward. This dashed vertical line shows the sequential events that occur to an object during the charted process. Lifelines may begin with a labelled rectangle shape or an actor symbol.	
Alternative symbol	Symbolises a choice (that is usually mutually exclusive) between two or more message sequences. To represent alternatives, use the labelled rectangle shape with a dashed line inside.	



Message symbol	This symbol is used when a sender needs to send a message.	
Reply message symbol	Represented by a dashed line with a lined arrowhead, these messages are replies to calls.	

Table 4.1.2.1: Sequence Diagrams Symbols

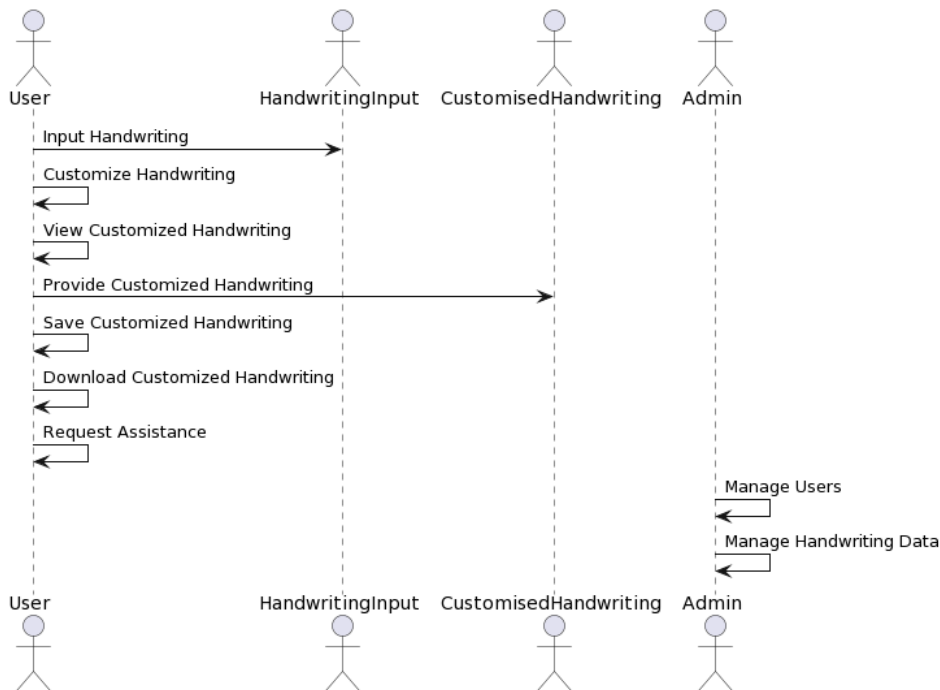


Fig 4.1.2.1: Sequence Diagram for the application

In the above sequence diagram, the lifelines are:

- User
- HandwritingInput
- CustomizedHandwriting
- Admin

The sequence diagram showcases user actions within the Handwriting Customization System, including inputting handwriting, customizing, viewing, providing, saving, downloading, and requesting assistance. It also illustrates Admin tasks like managing users and handwriting data.

4.1.3 ACTIVITY DIAGRAM

An activity diagram is a flowchart that displays the movement of information from one action to the next. A system operation can be used to describe the activity.

From one operation to the next, the control flow is guided. In nature, this flow might be sequential, branching, or concurrent. By employing numerous parts such as join, fork and so on, activity diagrams cope with all sorts of flow control.

Activity diagrams provide the same basic functions as the other four diagrams. It captures the dynamic behavior of the system. The other four diagrams depict message flow from one item to the next, whereas the activity diagram depicts a specific system operation referred to as an activity. It doesn't show any communication flow from one activity to the next. The phrases activity diagrams and flowcharts are often used interchangeably. Although the diagrams resemble flowcharts, they are not.

Notations

Initial point or start point

A small, filled circle, followed by an arrow, represents the beginning action state or starting point for any activity diagram. Make sure the start point of an activity diagram with swimlanes is in the top left corner of the first column.

Activity or Action state

An action state is a representation of an object's non-interruptible action. You can make an action state in SmartDraw by sketching a rectangle with rounded corners.

Action flow

Transitions from one action state to another are depicted by action flows, also known as edges and routes. An arrowed line is commonly used to depict them.

Decisions and branching

A diamond signifies a multiple-choice decision. Place a diamond between the two activities when one requires a decision before moving on to the next. A condition or guard expression should be used to label the outgoing alternates. One of the paths can also be labeled "else."

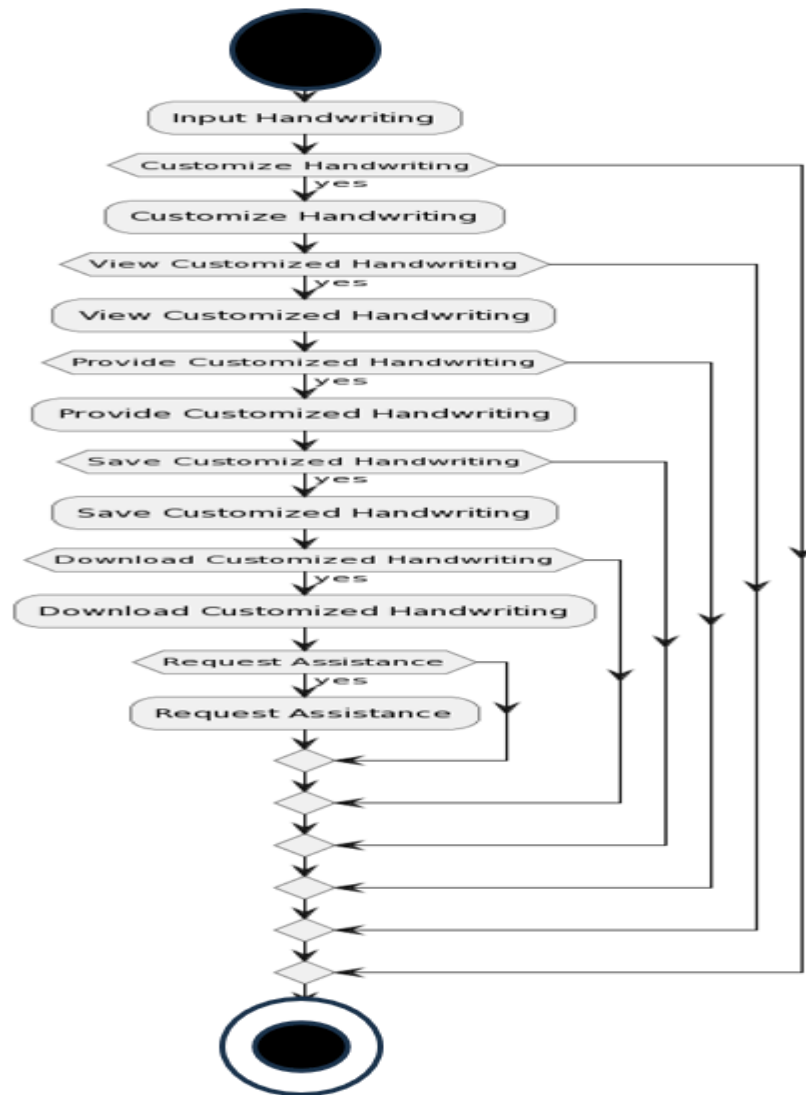


Fig 4.1.3.1: Activity Diagram for the system

The activity diagram depicts the sequential flow of actions in the Handwriting Customization System, including inputting, customizing, viewing, providing, saving, downloading, and requesting assistance, culminating in the end state.

5. PROPOSED SYSTEM

5.1 METHODOLOGY

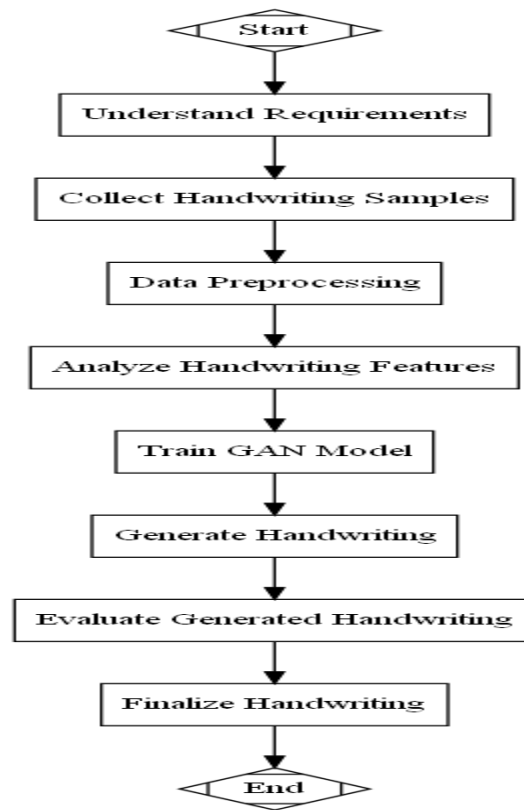


Fig. 5.1.1 Methodology

Data Set:

Data is crucial in finding patterns and using that pattern to generate the handwriting. For our project we used BRUSH data set.

Cleaning:

Data cleaning is the process of preparing data for analysis by weeding out information that is irrelevant or incorrect.

This is generally data that can have a negative impact on the model or algorithm it is fed

into by reinforcing a wrong notion.

Data cleaning not only refers to removing chunks of unnecessary data, but it's also often associated with fixing incorrect information within the train-validation-test dataset and reducing duplicates.

The numpy files created ensure that the data is cleaned, and we get maximum accuracy while performing the project.

Model:

A machine learning model is defined as a mathematical representation of the output of the *training process*. Machine learning is the study of different algorithms that can improve automatically through experience & old data and build the model. A machine learning model is similar to computer software designed to recognize patterns or behaviors based on previous experience or data. The learning algorithm discovers patterns within the training data, and it outputs an ML model which captures these patterns and makes predictions on new data.

Train and Test:

Machine Learning is one of the booming technologies across the world that enables computers/machines to turn a huge amount of data into predictions. However, these predictions highly depend on the quality of the data, and if we are not using the right data for our model, then it will not generate the expected result. In machine learning projects, we generally divide the original dataset into training data and test data. We train our model over a subset of the original dataset, i.e., the training dataset, and then evaluate whether it can generalize well to the new or unseen dataset or test set. Therefore, train and test datasets are the two key concepts of machine learning, where the training dataset is used to fit the model, and the test dataset is used to evaluate the model.

5.2 FUNCTIONALITIES

5.2.1 Collection of Data:

- Dataset is collected from kaggle.

5.2.2 Training Data

- Model is trained with the data using the dataset collected.

5.2.3 Choosing Model:

- HiGAN.

5.2.4 Detect:

- Accuracy and generate the handwriting.

5.3 ADVANTAGES OF PROPOSED SYSTEM:

- **Customization Precision:** The system utilizes advanced algorithms and machine learning models (like GANs) to precisely customize handwriting based on user preferences, offering a high degree of accuracy and customization.
- **Feature Extraction:** By analyzing handwriting features, the system can extract and understand intricate details, enabling the generation of diverse and realistic handwriting styles.
- **Automation and Efficiency:** The system's automated process streamlines the customization, generation, and evaluation steps, enhancing efficiency and reducing manual effort.
- **Machine Learning Model:** Leveraging GAN's allows the system to create realistic and unique handwriting samples, mimicking various styles, thickness, slants and strokes enhancing the system's adaptability.
- **Flexibility and Adaptability:** The system can adapt to different input styles and preferences, providing a flexible solution catering to a wide range of handwriting customization needs.

6. CODING AND IMPLEMENTATIONS

6.1 DATASET:

The BRUSH dataset (BRown University Stylus Handwriting) contains 27,649 online handwriting samples from a total of 170 writers. Every sequence is labeled with intended characters such that dataset users can identify to which character a point in a sequence corresponds.

6.2 IMPLEMENTATION:

Initially we imported all the required libraries such as torch, pickle,svhwrite, ffmpeg, etc

```
import os
import re
from random import random
import torch
import pickle
import argparse
import numpy as np
from helper import *
from PIL import Image
import torch.nn as nn
import torch.optim as optim
from config.GlobalVariables import *
► Launch TensorBoard Session
from tensorboardX import SummaryWriter
from SynthesisNetwork import SynthesisNetwork
from DataLoader import DataLoader
import svgwrite
import ffmpeg
```

Fig:6.2.1 Libraries used

Vector representation:

For algorithm to understand, the input handwriting must be converted to vector representation. The input handwriting is converted into a vector matrix which contains the different attributes of the handwriting.


```
def get_writer_blend_W_c(writer_weights, all_ws, all_cs):

    n, M, _ = all_ws.shape
    weights_tensor = torch.tensor(writer_weights).repeat_interleave(M * L).reshape(n, M, L) # repeat accross remaining dimensions
    W_vectors = (weights_tensor * all_ws).sum(axis=0).unsqueeze(-1) # take weighted sum accross writers axis
    char_matrices = all_cs[0, :, :, :] # character matrices are independent of writer

    W_cs = torch.bmm(char_matrices, W_vectors)

    return W_cs.reshape(M, 1, L)
```

Fig: 6.2.4 Average the styles altogether

```
def get_character_blend_W_c(character_weights, all_ws, all_cs):

    M = len(character_weights)
    W_vector = all_ws[0, 0, :].unsqueeze(-1)

    weights_tensor = torch.tensor(character_weights).repeat_interleave(L * L).reshape(1, M, L, L) # repeat accross remaining dimensions
    char_matrix = (weights_tensor * all_cs).sum(axis=1).squeeze() # take weighted sum accross characters axis

    W_c = char_matrix @ W_vector

    return W_c.reshape(1, 1, L)
```

Fig: 6.2.5 Averaging the character styles

```

def get_commands(net, target_word, all_W_c): # seems like target_word is only used for length

    all_commands = []
    current_id = 0
    while True:
        word_Wc_rec_TYPE_D = []
        TYPE_D_REF = []
        cid = 0
        for segment_batch_id in range(len(all_W_c)):
            if len(TYPE_D_REF) == 0:
                for each_segment_Wc in all_W_c[segment_batch_id]:
                    if cid >= current_id:
                        word_Wc_rec_TYPE_D.append(each_segment_Wc)
                        cid += 1
                    if len(word_Wc_rec_TYPE_D) > 0:
                        TYPE_D_REF.append(all_W_c[segment_batch_id][-1])
            else:
                for each_segment_Wc in all_W_c[segment_batch_id]:
                    magic_inp = torch.cat([torch.stack(TYPE_D_REF, 0), each_segment_Wc.unsqueeze(0)], 0)
                    magic_inp = magic_inp.unsqueeze(0)
                    TYPE_D_out, (c, h) = net.magic_lstm(magic_inp)
                    TYPE_D_out = TYPE_D_out.squeeze(0)
                    word_Wc_rec_TYPE_D.append(TYPE_D_out[-1])
                    TYPE_D_REF.append(all_W_c[segment_batch_id][-1])
        WC_ = torch.stack(word_Wc_rec_TYPE_D)
        tmp_commands, res = net.sample_from_w_fix(WC_)
        current_id += res
        if len(all_commands) == 0:
            all_commands.append(tmp_commands)
        else:
            all_commands.append(tmp_commands[1:])
        if res < 0 or current_id >= len(target_word):
            break

    commands = []
    px, py = 0, 100
    for coms in all_commands:
        for i, [dx, dy, t] in enumerate(coms):
            x = px + dx * 5
            y = py + dy * 5
            commands.append([x, y, t])
            px, py = x, y
    commands = np.asarray(commands)
    commands[:, 0] -= np.min(commands[:, 0])

    return commands

```

Fig: 6.2.6 Convert to drawing commands

```

def mdn_video(target_word, num_samples, scale_sd, clamp_mdn, net, all_loaded_data, device):

    words = target_word.split(' ')
    us_target_word = re.sub(r"\s+", '_', target_word)
    os.makedirs(f"./results/{us_target_word}_mdn_samples", exist_ok=True)
    for i in range(num_samples):
        net.scale_sd = scale_sd
        net.clamp_mdn = clamp_mdn

        mean_global_W = get_mean_global_W(net, all_loaded_data[0], device)

        word_Ws = []
        word-Cs = []
        for word in words:
            writer_Ws, writer-Cs = get_DSD(net, word, [mean_global_W], [all_loaded_data[0]], device)
            word_Ws.append(writer_Ws)
            word-Cs.append(writer-Cs)

        im = draw_words(words, word_Ws, word-Cs, [1], net)
        im.convert("RGB").save(f'results/{us_target_word}_mdn_samples/sample_{i}.png')
    # Convert frames to video using ffmpeg
    photos = ffmpeg.input(f'results/{us_target_word}_mdn_samples/sample_*.png', pattern_type='glob', framerate=10)
    videos = photos.output(f'results/{us_target_word}_video.mov', vcodec="libx264", pix_fmt="yuv420p")
    videos.run(overwrite_output=True)

```

Fig: 6.2.7 Method creating gif of mdn samples

```

def sample_blended_writers(writer_weights, target_sentence, net, all_loaded_data, device="cpu"):
    |
    words = target_sentence.split(' ')

    writer_mean_ws = []
    for loaded_data in all_loaded_data:
        |
        mean_global_w = get_mean_global_w(net, loaded_data, device)
        |
        writer_mean_ws.append(mean_global_w)

    word_ws = []
    word_cs = []
    for word in words:
        |
        writer_ws, writer_cs = get_DSD(net, word, writer_mean_ws, all_loaded_data, device)
        |
        word_ws.append(writer_ws)
        |
        word_cs.append(writer_cs)

    return draw_words(words, word_ws, word_cs, writer_weights, net)

```

Fig: 6.2.8 Generates image of handwritten text

7. TESTING

In machine learning, testing is mainly used to validate raw data and check the ML model's performance. The main objectives of testing machine learning models are:

- Quality Assurance
- Detect bugs and flaws

Once your machine learning model is built (with your training data), you need unseen data test your model. This data is called testing data, and you can use it to evaluate the performance and progress of your algorithms' training and adjust or optimize it for improved results.

Testing data has two main criteria. It should:

- Represent the actual dataset
- Be large enough to generate meaningful predictions

7.1 TYPES OF TESTING

7.1.1 MANUAL TESTING

Manual Testing is a type of software testing in which test cases are executed manually by a tester without using any automated tools. The purpose of Manual Testing is to identify the bugs, issues, and defects in the software application. Manual software testing is the most primitive technique of all testing types and it helps to find critical bugs in the software application.

Any new application must be manually tested before its testing can be automated. Manual Software Testing requires more effort but is necessary to check automation feasibility. Manual Testing concepts does not require knowledge of any testing tool. One of the Software Testing Fundamental is “**100% Automation is not possible** “. This makes Manual Testing imperative.

7.1.2 AUTOMATED TESTING

Automation Testing is a software testing technique that performs using special automated testing software tools to execute a test case suite. On the contrary, Manual Testing is performed by a human sitting in front of a computer carefully executing the test steps.

The automation testing software can also enter test data into the System Under Test, compare expected and actual results and generate detailed test reports. Software Test Automation demands considerable investments of money and resources.

7.2 SOFTWARE TESTING METHODS

7.2.1 BLACK BOX TESTING

Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer. In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.

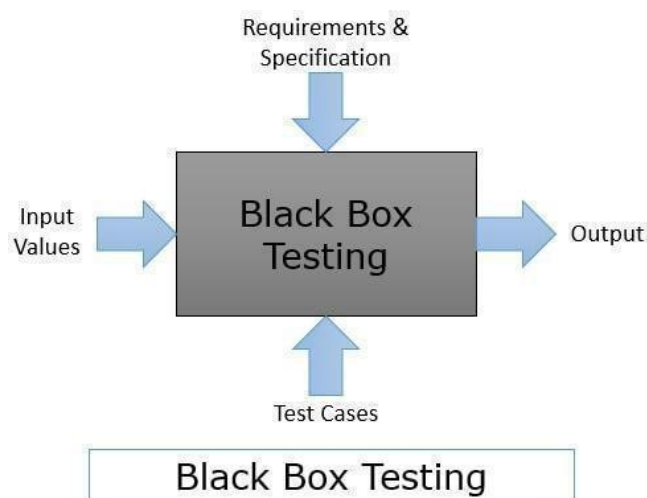


Fig. 7.2.1.1 Black Box Testing

7.2.2 GRAY BOX TESTING

Gray box testing is a software testing method to test the software application with partial knowledge of the internal working structure. It is a **combination of black box and white box testing** because it involves access to internal coding to design test cases as white box testing and testing practices are done at functionality level as black box testing.

Gray box testing commonly identifies context-specific errors that belong to web systems. For example; while testing, if tester encounters any defect, then he makes changes in code to resolve the defect and then test it again in real time. It concentrates on all the layers of any complex software system to increase testing coverage. It gives the ability to test both presentation layer as well as internal coding structure. It is primarily used in integration testing and penetration testing.

This testing technique is a combination of Black box testing and White box testing. In Black box testing, the tester does not have any knowledge about the code. They have information for what will be the output for the given input. In White box testing, the tester has complete knowledge about the code. Grey box testers have knowledge of the code, but not completely.



Fig. 7.2.2.1 Grey Box Testing

7.2.3 WHITE BOX TESTING

White box testing is an approach that allows testers to inspect and verify the inner workings of a software system—its code, infrastructure, and integrations with external systems. White box testing is an essential part of automated build processes in a modern Continuous Integration/Continuous Delivery (CI/CD) development pipeline. White box testing is often referenced in the context of Static Application Security Testing (SAST), an approach that checks source code or binaries automatically and provides feedback on bugs and possible vulnerabilities. White box testing is a testing technique, that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

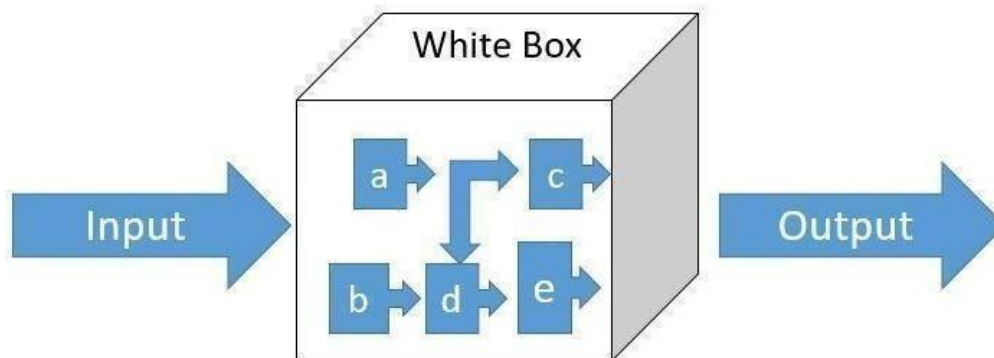


Fig. 7.2.3.1 White Box Testing

7.3 TESTING LEVELS

7.3.1 NON-FUNCTIONAL TESTING

Non-functional testing is a type of software testing to test non-functional parameters such as reliability, load test, performance and accountability of the software. The primary purpose of non-functional testing is to test the reading speed of the software system as per non-functional parameters. The parameters of non-functional testing are never tested before the functional testing. Non-functional testing is also very important as functional testing because it plays a crucial role in customer satisfaction.

7.3.1.1 PERFORMANCE TESTING

Performance testing is a form of software testing that focuses on how a system running the system performs under a particular load. This is not about finding software bugs or defects. Different performance testing types measure according to benchmarks and standards. Performance testing gives developers the diagnostic information they need to eliminate bottlenecks.

7.3.1.2 STRESS TESTING

Stress Testing is a type of software testing that verifies stability & reliability of software application. The goal of Stress testing is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and ensuring that software doesn't crash under crunch situations. It even tests beyond normal operating points and evaluates how software works under extreme conditions.

7.3.1.3 SECURITY TESTING

Security Testing is a type of Software Testing that uncovers vulnerabilities of the system and determines that the data and resources of the system are protected from possible intruders. It ensures that the software system and application are free from any threats or risks that can cause a loss. Security testing of any system is focused on finding all possible loopholes and weaknesses of the system which might result in the loss of information or reputation of the organization.

7.3.1.4 PORTABILITY TESTING

Portability Testing is one of Software Testing which is carried out to determine the degree of ease or difficulty to which a software application can be effectively and efficiently transferred from one hardware, software or environment to another one. The results of portability testing are measurements of how easily the software component or application will be integrated into the environment and then these results will be compared to the non-functional requirement of portability of the software system.

7.3.1.5 USABILITY TESTING

Usability Testing, also known as User Experience (UX) Testing, is a testing method for measuring how easy and user-friendly a software application is. A small set of target end-users, use software applications to expose usability defects. Usability testing mainly focuses on the user's ease of using application, flexibility of application to handle controls and ability of application to meet its objectives. This testing is recommended during the initial design phase of SDLC, which gives more visibility on the expectations of the users.

7.3.2 FUNCTIONAL TESTING

It is a type of software testing which is used to verify the functionality of the software application, whether the function is working according to the requirement specification. In functional testing, each function is tested by giving the value, determining the output, and verifying the actual output with the expected value. Functional testing performed as black-box testing which is presented to confirm that the functionality of an application or system behaves as we are expecting. It is done to verify the functionality of the application. Functional testing is also called black-box testing.

7.3.2.1 INTEGRATION TESTING

Integration testing is done to test the modules/components when integrated to verify that they work as expected i.e. to test the modules which are working fine individually and do not have issues when integrated. The main function or goal of this testing is to test the interfaces between the units/modules. The individual modules are first tested in isolation. Once the modules are unit tested, they are integrated one by one, till all the modules are integrated, to check the combinational behavior, and validate whether the requirements are implemented correctly.

7.3.2.2 REGRESSION TESTING

Regression Testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features. Regression Testing is nothing but a full or partial selection of already executed test cases that are re-executed to ensure existing functionalities work fine. This testing is done to ensure that new code changes do not have side effects on the existing functionalities. It ensures that the old code still works once the latest code changes are done.

7.3.2.3 UNIT TESTING

Unit Testing is a software testing technique by means of which individual units of software. i.e. a group of computer program modules, usage procedures, and operating procedures are tested to determine whether they are suitable for use or not. Unit Testing is defined as a type of software testing where individual components of a software are tested. Unit Testing of the software product is carried out during the development of an application. An individual component may be either an individual function or a procedure.

7.3.2.4 ALPHA TESTING

Alpha Testing is a type of software testing performed to identify bugs before releasing the product to real users or to the public. Alpha Testing is one of the **user acceptance testing**. This is referred to as alpha testing only because it is done early on, near the end of the development of the software. Alpha testing is commonly performed by homestead software engineers or quality assurance staff. It is the last testing stage before the software is released into the real world.

7.3.2.5 BETA TESTING

Beta Testing is performed by real users of the software application in a real environment. Beta testing is one of the types of **User Acceptance Testing**. A Beta version of the software, whose feedback is needed, is released to a limited number of end-users of the product to obtain feedback on the product quality. Beta testing helps in minimization of product failure risks, and it provides increased quality of the product through customer validation. It is the last test before shipping a product to the customers. One of the major advantages of beta testing is direct feedback from customers.

7.4 TEST CASES

Sl.no	Testcase	Expected Result	Actual Result	Pass/ Fail
1.	After Training testing with test sample data	Generate accurate handwritten text.	Generated handwritten text similar to original	PASS

Table 7.4.1: Test Cases

8. RESULTS

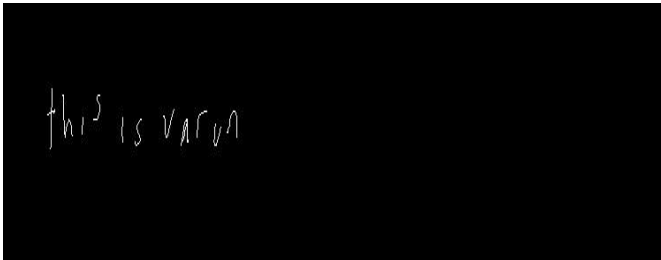


Fig:8.1 Handwriting Generated in style-1

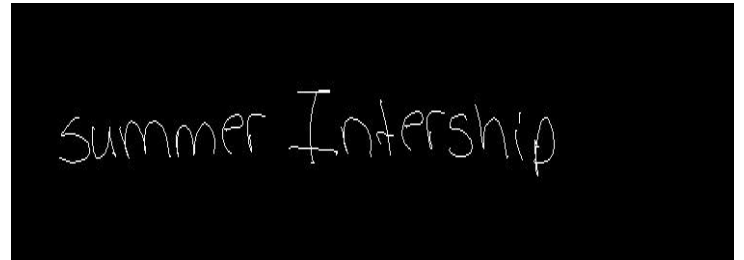


Fig:8.1 Handwriting Generated in style-5

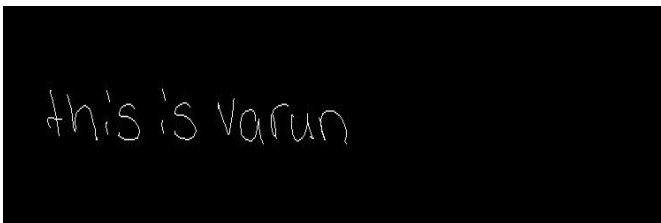


Fig: 8.2 Handwriting Generated in style-2

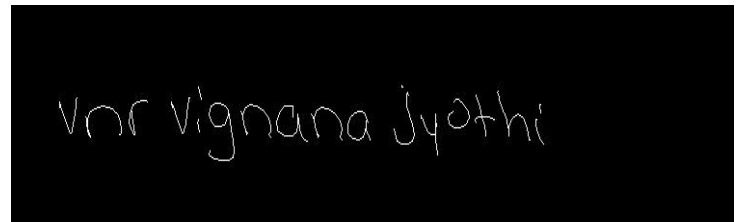


Fig: 8.2 Handwriting Generated in style-7

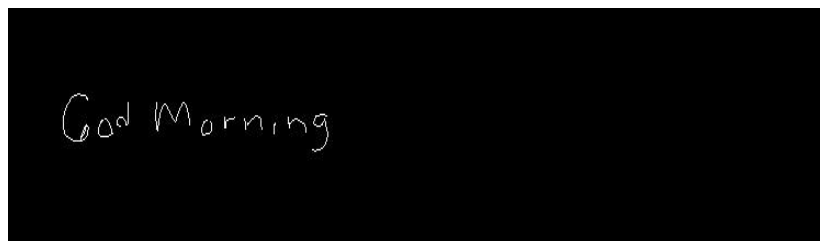


Fig: 8.2 Misspelled "Good Morning"

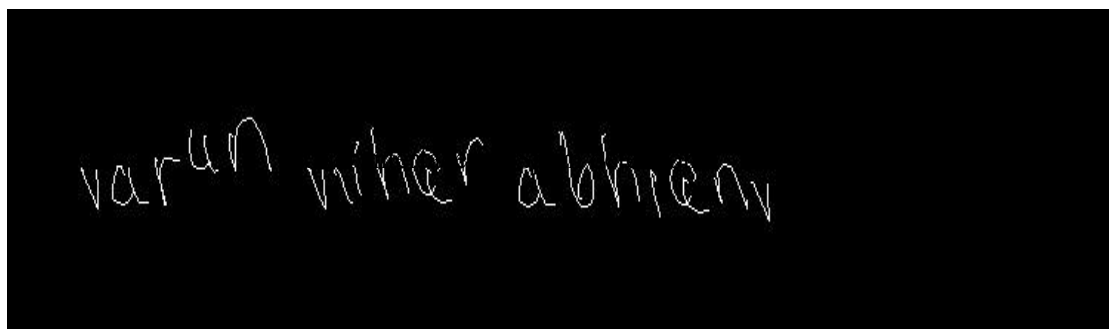


Fig: 8.3 Handwriting generated using style-3

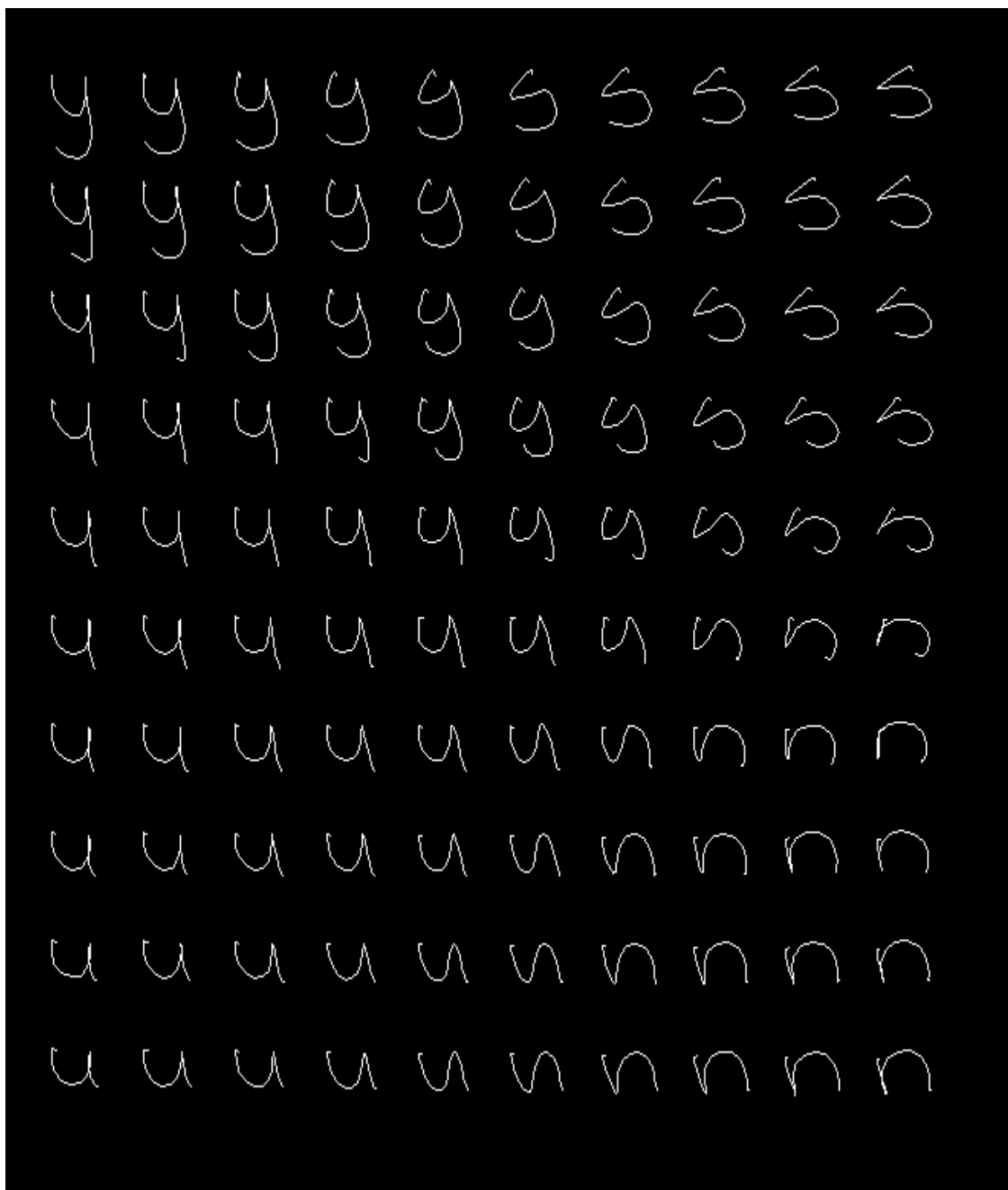


Fig:8.4 Handwritings in different styles

9. CONCLUSION AND FURTHER WORK

- The Handwriting Customization System harnesses cutting-edge machine learning techniques, particularly Generative Adversarial Networks (GANs), to empower users with personalized and diverse handwriting styles. Through meticulous feature extraction, robust training methodologies, and iterative refinement, the system offers a sophisticated platform for generating customized handwriting.
- By comprehensively understanding user requirements, collecting diverse handwriting samples, and employing advanced preprocessing techniques, the system lays a strong foundation. Leveraging GANs for model training enables the system to synthesize realistic and personalized handwriting styles, catering to various preferences and nuances.
- Although the accuracy of such models may vary due to subjective perceptions of handwriting quality and the intricacies of individual styles, the system's iterative approach allows for continual enhancement. User feedback mechanisms enable ongoing model improvements, ensuring a more refined and adaptable system over time.

REFERENCES

1. Generating Sequences With Recurrent Neural Networks:- <https://arxiv.org/pdf/1308.0850>
2. A Neural Representation of Sketch Drawings :- https://arxiv.org/pdf/1704.03477.pdf?source=post_page
3. Drawing and Recognizing Chinese Characters with Recurrent Neural Network :- <https://doi.org/10.48550/arXiv.1606.06539>
4. Generating Handwriting via Decoupled Style Descriptors:- <https://arxiv.org/pdf/2008.11354.pdf>
5. The Character Generation in Handwriting Feature Extraction Using Variational AutoEncoder:- <https://doi.org/10.1109/ICDAR.2017.169>
6. Adversarial Generation of Handwritten Text Images Conditioned on Sequences:- <https://arxiv.org/pdf/1903.00277.pdf>
7. HI-GAN:- handwriting imitation :- HiGAN: Handwriting Imitation Conditioned on Arbitrary-Length Texts and Disentangled Styles <https://doi.org/10.1609/aaai.v35i9.16917>
8. TS-GAN:- Text and Style Conditioned GAN for Generation of Offline Handwriting Lines <https://arxiv.org/pdf/2009.00678.pdf>
9. GANwriting: Content-Conditioned Generation of Styled Handwritten Word Images:- <https://arxiv.org/pdf/2003.02567.pdf>
10. ScrabbleGAN: Semi-Supervised Varying Length Handwritten Text Generation:- <https://shorturl.at/fln34>