SIGNATURE COMPARISON AND FORGERY DETECTION USING DEEP LEARNING AND STRUCTURAL SIMILARITY INDEX (SSIM)

Submitted In Partial Fulfilment of Requirements
For the Degree Of

Master of Science
(Computer Science)

Guide
Dr. Swati Maurya

Department of Information Technology and Computer Science
S K Somaiya College, Somaiya Vidyavihar University

By
Mr. Karan Subramaian Mudliyar
Roll no: 31031522019

S K Somaiya College

Somaiya Vidyavihar University
Vidyavihar East, Mumbai 400077
2023-2024

Student details

Name	Karan Subramaian Mudliyar
Subject	Computer Science
Institution	S K Somaiya College
Year	2023-24
Teacher in Charge	Dr. Swati Maurya
Title of the Project	SIGNATURE COMPARISON AND FORGERY DETECTION USING DEEP LEARNING AND STRUCTURAL SIMILARITY INDEX (SSIM)
Location	Mumbai
Duration	6 Months
Signature of the Teacher	
Signature of the Coordinator of Department	

CERTIFICATE OF AUTHENTICATION

This is to certify that the project entitled “*SIGNATURE COMPARISON AND FORGERY DETECTION USING DEEP LEARNING AND STRUCTURAL SIMILARITY INDEX (SSIM)*” is a bonafide work of “*Karan Subramaian Mudliyar*” (*31031522019.)* submitted to the S K Somaiya College in partial fulfillment of the requirement for the award of the degree of “M.Sc. in the subject of Computer Science”.

I considered that the dissertation has reached the standards and fulfilling the requirements of the rules and regulations relating to the nature of the degree. The contents embodied in the dissertation have not been submitted for the award of any other degree or diploma in this or any other university.

Date:

Place: Mumbai

(Name and sign)   (Name and sign)

External Examiner   Internal Mentor

Dr. Swati Maurya
Head of the department

Declaration by the student

I certify that

a) The work contained in the thesis is original and has been done by myself under the supervision of my supervisor.

b) The work has not been submitted to any other Institute for any degree or diploma.

c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

d) Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

e) Whenever I have quoted written materials from other sources and due credit is given to the sources by citing them.

f) From the plagiarism test, it is found that the similarity index of the whole thesis within 25% and single paper is less than 10 % as per the university guidelines.

Date:

Place: Mumbai

Signature

Karan Subramaian Mudliyar

31031522019.

SOMAIYA
VIDYAVIHAR UNIVERSITY
S K Somaiya College

Department of Information Technology and Computer Science

Programme: Computer Science

CERTIFICATE

This is to certify that Mr./Ms. KARAN SUBRAMAIAN MUDLIYAR of **M.Sc. Computer Science** has satisfactorily completed the Project titled SIGNATURE COMPARISON AND FORGERY DETECTION USING DEEP LEARNING AND STRUCTURAL SIMILARITY INDEX(SSIM) for the Partial fulfilment of the Degree by the Somaiya Vidyavihar University, during the Academic year **2023-24**.

Signature of the Teacher In-Charge

Signature of the HOD

Signature of the Examiner/s

Signature of the Director

Date of Examination

College Seal

Examiner Approval sheet

This dissertation/project report entitled SIGNATURE COMPARISON AND FORGERY DETECTION USING DEEP LEARNING AND STRUCTURAL SIMILARITY INDEX (SSIM) by Karan Subramaian Mudliyar is approved for the degree of Master of Science in the subject of Computer Science.

Examiners

(Name and signature)

1.---

Place: Mumbai

Date: 19/06/2024

Acknowledgement

The success and outcome of this project required a lot of guidance and assistance for many people, and I am extremely privileged to have this all along the completion of my project. All that I have done is only due to such supervision and assistance and I would not forget to thank them.

I owe my deep gratitude to our project guide Dr. Swati Maurya, who took keen interest in my project work and guided me all along, till the completion of my project work by providing all the necessary information for developing a good system.

I am thankful to our Director C A Monica Lodha for providing us with the facilities for smooth working of our project. I am thankful and fortunate to get constant encouragement, support, and guidance from Dr. Swati Maurya, Coordinator of Computer Science department and all the teaching staff of the S K Somaiya College, who helped me in successfully completing my project work.

Karan Subramaian Mudliyar

INDEX

Contents

List of Figures

ABSTRACT

This dissertation presents an advanced GUI application for automated signature verification using a combination of traditional image processing techniques and deep learning models. The primary objective is to develop a reliable system capable of comparing and verifying signatures with high accuracy, addressing the prevalent issue of signature forgery in various domains. The application leverages the VGG16 deep learning model for feature extraction and uses cosine similarity to compare the feature vectors of different signatures. Additionally, the system employs Structural Similarity Index (SSIM) for forgery detection, ensuring a robust dual-method approach.

The application features a user-friendly interface built with Tkinter, enabling users to capture signatures via a webcam or upload images from local storage. An important functionality is the ability to browse folders containing multiple signature images, allowing batch processing and selection of the most similar signature. The GUI also includes a login system to restrict unauthorized access, enhancing security. The system logs performance data, including the paths of the compared signatures and their similarity percentages, into a CSV file. This logging is crucial for tracking performance and auditing purposes. The dissertation also details the integration of object detection using the YOLO model to ensure that only signature images are processed, thus reducing errors from non-signature inputs.

The project is structured into several chapters. The methodology chapter describes the preprocessing steps, feature extraction using VGG16, and similarity calculation methods. The implementation chapter provides insights into the GUI development, capturing images, and handling user inputs. The evaluation chapter discusses the system's performance, accuracy, and limitations based on extensive testing with various signature datasets. Finally, the conclusion highlights the system's effectiveness and potential areas for future enhancement.

The dissertation demonstrates that combining deep learning with traditional image processing techniques can significantly improve the accuracy and reliability of signature verification systems, offering a promising solution for combating signature fraud.

Keywords: Signature Verification, Deep Learning, VGG16, Structural Similarity Index, Cosine Similarity, GUI, Tkinter, Image Processing, Feature Extraction, Forgery Detection, Object Detection, YOLO, Webcam Capture, Batch Processing, CSV Logging, User Authentication, Preprocessing, Image Comparison, Automated Verification, Security.

1. Introduction

In this project, our aim is to implement a robust system for detecting whether a signature is genuine or fake. Signature forgery detection is a critical area of research within forensic science, especially as advancements in technology have made it easier for forgers to create convincing forgeries. Traditional methods of visual inspection and comparison with known samples have limitations, prompting the need for more sophisticated computational approaches.

Signature forgery detection is the process of determining whether a given signature is genuine or fake. It is an important area of research in the field of forensic science, as signatures are commonly used to authenticate important documents and transactions. In recent years, advances in technology have made it easier for forgers to create convincing forgeries, which has made signature forgery detection more challenging.

There are several approaches to signature forgery detection, including visual inspection, comparison with a known signature sample, and analysis of the physical properties of the signature (such as ink composition and paper type). Some techniques use computer vision algorithms to analyze the shape, texture, and other characteristics of the signature.

Signature forgery detection has important applications in law enforcement, financial institutions, and other industries that rely on the authenticity of signatures. By detecting forged signatures, it can help prevent fraud and protect individuals and organizations from financial losses.

Signature forgery detection is a critical task in many industries, including finance, law enforcement, and government. A forged signature can lead to legal disputes, financial losses, and damage to reputation. Therefore, it is essential to have reliable and accurate methods for detecting forgery.

One of the most common approaches to signature forgery detection is visual inspection, which involves comparing the suspected signature to a known sample of the genuine signature. This method is subjective and can be affected by human error, as well as the quality of the sample signature. Another approach is the use of computer vision algorithms, which analyze the visual characteristics of the signature, such as its shape, size, and texture.

Recent advancements techniques leveraging image processing and deep learning have emerged as powerful tools for enhancing the security and integrity of signature-based systems. These systems use a combination of image processing techniques and machine learning algorithms to analyze signatures and identify patterns that indicate forgery. They can also detect subtle differences between genuine and forged signatures that are not apparent to the human eye.

In addition to visual inspection and computer vision, other methods of signature forgery detection include analysis of physical properties such as ink composition and paper type, and the use of biometric authentication techniques such as handwriting recognition. The effectiveness of these methods depends on the specific circumstances of the forgery and the available resources for analysis.

Overall, signature forgery detection is a complex and important field that requires a combination of technical expertise and practical experience. By using reliable and accurate methods for detecting forgery, it is possible to prevent fraud and protect individuals and organizations from financial losses and legal disputes. When a person signs a signature his nerve impulses are controlled by the brain without paying any attention to any signature details. There are two main types of signature verification such as Static and Dynamic respectively. Static verification is one in which signatures are not moving and those signatures are still.Whereas, in Dynamic signature recognition, signature is done on a display like a touchscreen and verification is done via any electronic method

There are four types of forgery such as Simulation forgery, Blind forgery, Tracing, and Optical Transfer.

Simulation forgery is a type of forgery in which a person or a forger has samples of signatures that are used for forging. This totally depends on quality of the simulation. The quality of the simulation depends on the person who practices and tries too many attempts before making a forged signature or the actual signature. This is called forger practices.These forger experiences are then classified into two types such as skilled and unskilled forgery respectively.

Blind Forgery is the type of forgery in which the forger has no idea about the actual forgery signature and how does it look like. This forgery is easy to detect by the examiners as the signature made by the forger are not very close to the appearance of the genuine signature of a user. Examiners can identify who made the forged signature and its based on handwriting habits that are present in the forged signature.

Tracing is the third type of forgery in which the forger holds questioned documents and model document up to the light. Then forger uses a pen to trace the lines of the model signature onto the questioned document. This tracing can also be done using a blunt stylus on the questioned document. This creates an impression on the model signature on the paper.This impression is then filled in with a pen, and then it creates an appearance of the model's signature. If the model signature used by the forger is not found, then this type of forgery is sometimes difficult to detect from a photocopy

Optical Transfer is one such type of forgery in which a genuine signature is transferred on a document using a photocopy, photography, or a scanner. Thus an examiner dealing with this type of forgery cannot identify whether the given signature is genuine without having the original for comparison.

Signature forgery detection using Deep learning methods involves using SSIM algorithms and models to analyze the visual characteristics of signatures and identify patterns that indicate forgery. Here is an overview of the process, including importing libraries and using a TensorFlow model:

The Python application presented herein embodies a sophisticated approach to signature verification and forgery detection. Developed using libraries such as Tkinter for graphical user interface (GUI) design, OpenCV for image processing, and TensorFlow for deep learning, this application offers a comprehensive suite of functionalities tailored to meet the demands of modern signature authentication systems.

At its core, the application leverages the capabilities of image processing techniques to preprocess signature images, extract relevant features, and perform quantitative analysis. Through the utilization of OpenCV and PIL (Python Imaging Library), signature images are loaded, resized, and converted into a format suitable for further analysis. Additionally, the application incorporates the Structural Similarity Index (SSIM) metric to quantitatively assess the similarity between signature images, enabling accurate comparison and verification.

Furthermore, the application integrates state-of-the-art deep learning models, specifically the VGG16 convolutional neural network (CNN), to extract high-level features from signature images. By leveraging pre-trained VGG16 models, signatures are encoded into feature vectors, facilitating robust and discriminative representation. The extracted features are then used to calculate similarity scores, enabling precise matching and verification of signatures with exceptional accura

2. Synopsis

In today's digital age, ensuring the authenticity of signatures is crucial across various sectors such as finance, legal, and authentication systems. Traditional methods of signature verification often lack the precision and scalability required to combat sophisticated forgery techniques. To address this challenge, our proposes an innovative approach that harnesses the power of deep learning alongside the Structural Similarity Index (SSIM) for enhanced signature comparison and forgery detection.

The proposed methodology revolves around the creation of a robust deep learning model engineered specifically for signature verification tasks. Central to our approach is the integration of SSIM, a well-established metric in image processing renowned for its ability to quantify structural similarities between images. By incorporating SSIM into the model's architecture and training process, we aim to enhance the system's capability to discern nuanced differences between genuine signatures and their counterfeit counterparts.

A critical aspect of our methodology is the meticulous curation of a diverse dataset encompassing authentic signatures and their corresponding forgeries. The dataset undergoes rigorous preprocessing procedures to standardize image formats, enhance feature clarity, and mitigate noise interference. Augmentation techniques are also employed to bolster dataset diversity and fortify the model's resilience against varying styles of forgery.

The design of the deep learning architecture is pivotal to the success of our approach. We explore the suitability of convolutional neural networks (CNNs) and Siamese networks for signature verification tasks, with a particular focus on their adaptability to integrate SSIM into the model's training framework. By leveraging SSIM as a component of the loss function or auxiliary metric during training, our model not only learns discriminative features but also maintains perceptual fidelity, thereby enabling more accurate discrimination between genuine and forged signatures.

The training phase of our methodology entails the iterative refinement of the deep learning model using a combination of genuine signatures and labeled forgeries. Training, validation, and testing datasets are meticulously partitioned to ensure robust model performance. Evaluation metrics such as accuracy, precision, recall, and SSIM-based similarity indices are employed to assess the model's efficacy in detecting forged signatures accurately. Through rigorous training and validation, we aim to develop a model capable of reliably identifying counterfeit signatures while minimizing false positives.

Signature comparison and forgery detection using deep learning and the Structural Similarity Index (SSIM) involve leveraging advanced algorithms and models to analyze visual signatures and discern patterns indicative of forgery. Here's an overview of the process:

Data Preparation: The code facilitates the collection of a dataset comprising known genuine signatures and forgeries. Users can either select images from the file system or capture them using a connected camera. The dataset is preprocessed within the code, ensuring consistent image format and labeling the images appropriately as genuine or forged.

Data Augmentation: While not explicitly implemented in the provided code, data augmentation techniques could be incorporated to increase the dataset's size and diversity. This augmentation would enhance the model's robustness by introducing variations such as rotation, scaling, and adding noise to the signature images.

Feature Extraction: Pre-trained VGG16 deep learning model is employed to extract features from the signature images. The model processes the images and generates feature vectors representing various visual characteristics such as shape, texture, and size.These feature vectors serve as input to the machine learning model for further analysis and prediction.

Model Building: The code utilizes TensorFlow for building and training the machine learning model. Specifically, a deep learning architecture based on the VGG16 model is employed for signature forgery detection.Features extracted from the signatures are fed into the model, which analyzes them to predict whether a signature is genuine or forged.

Prediction: Once trained and evaluated, the model can be utilized to predict the authenticity of new signatures. Given a pair of signature images, the model analyzes their features and outputs a prediction indicating whether they are genuine or forged.

By following this systematic approach, the solution aims to provide accurate and reliable detection of forged signatures, thereby enhancing security and trust in various applications requiring signature verification.

3. Implementation

3.1 Project History

The handwritten signatures are considered as a behavioural biometric authentication which is not based on psychology characteristics but it's based on behaviour that changes over time. Several papers have proposed many solutions for signature verification and forgery detection such as using CNN , Fuzzy Modelling etc. We are using static images which are still and not moving.

3.2 Related Works

A novel method to read, extract signature, detect user and verify forgery has been implemented using VGG16 And SSIM Algorithm [1] . This paper is one of the very few papers which recognizes offline signature. Accuracy is lower due to a lack of preprocessing. From this paper its understood that CNN can be used in feature extraction. It is very important to detect features in a signature [2]. This paper uses an algorithm called SSIM, SSIM is used for measuring the similarity between two images. This algorithm is open source and can detect features easily.

[3]. The main disadvantage of this method is that the Surf algorithm is a patented algorithm. Using sub-pixel interpolation, which helps optimize the conventional ORB algorithm [4]This increases the clustering and accuracy calculation attributes. So from this paper, it's evident that the ORB algorithm performs well in feature extraction. Another interesting paper on detecting Mammogram image [5] Which Utilizing the substantial enriched database, the CNN is trained.

The model is then copied over to the smaller database, which is the original database, and tested there. In this work, three commonly used CNNs, including VGG-19, MatConvNet, and InceptionV3, are assessed for their ability to identify breast cancer. It is understood that CNN can be trained once and saved as a model that can be used in many places just by importing the model.

In another study, a model that determines if an image is fake or not is trained using CNN and the VGG16 architecture. [6]. For individuals who don't sign consistently, inconsistent signatures lead to a greater likelihood of mistaken rejection. Different numbers of hidden layers give different accuracy. This paper compares 5 layered, 4 layered, and 3 layered architectures in CNN.

[7]Comparing different models and taking the best result from them is an efficient strategy. A 7-layer independently created CNN model with a 98% accuracy rate is

used in research to determine Thai sign texts. The suggested system takes a natural image as input and localizes the text area using MSER, geometrical features, and the bounding box technique. Selected localized regions are then sent to CNN, which produces an output with the English translation for the Thai text image.

[8]. In this study, a novel method for translating text that makes use of picture categorization is presented. Comparing classic CNN-2, CNN-3, SIFT, and LeNet demonstrates that the performance of deep learning models in the recognition of traffic signs is superior to that of the conventional feature extraction method [9]. This result can be used in the feature extraction of a signature as well. It is possible to enhance an algorithm for SURF-based image retrieval.

[10] That may be used in embedded systems and boost the performance and precision of the existing SURF technique. Rasterization converts vector coordinates to image space, while vectorization learns to convert image space to vector coordinates [11]. The purpose of this study is to provide a Self-Supervised learning approach that can be used with any hand-drawn data that is represented as a rasterized image or a vector. For classification, CNN has developed a very accurate artificial intelligence system for recognizing and classifying a variety of plant leaf diseases.

The presented model evaluated 19 different classes and over 20,000 photos to reach the conclusion that CNN is the best for categorizing images [12]. Another interesting paper on cleaning stamps on signatures uses a modified CycleGAN [13] Method that cleans the stamps on the signature and extracts the signature alone and a signature representation based on CNN.

A custom signature dataset has also been created for this paper. a novel Chinese paper in which they created the ChiSig benchmark, a novel offline benchmark for Chinese document signature fraud detection that covers all pipeline activities, including signature detection, recovery, and verification [14]. Furthermore, in these three tasks, they thoroughly compare several Deep Learning-based algorithms. For this paper, a massive dataset database was created.

Using a convolutional Siamese network, the offline independent signature verification job is carried out [15]. The network is confronted with two pairings of similar and different data in order to accomplish this. The Euclidean distance is therefore maximized between pairs are different and maximized between pairs that are comparable. Demerit involves using a very large dataset. From all the related work, it was conclusive that doing preprocessing and using CNN to categorize the signature based on class is very necessary.

The class here in the signature is the name in the signature. Then Applying image processing algorithms such as ORB and SIFT can be used to efficiently detect fraud in signature images.

3.3 WorkFlow Diagram

STEP 1:

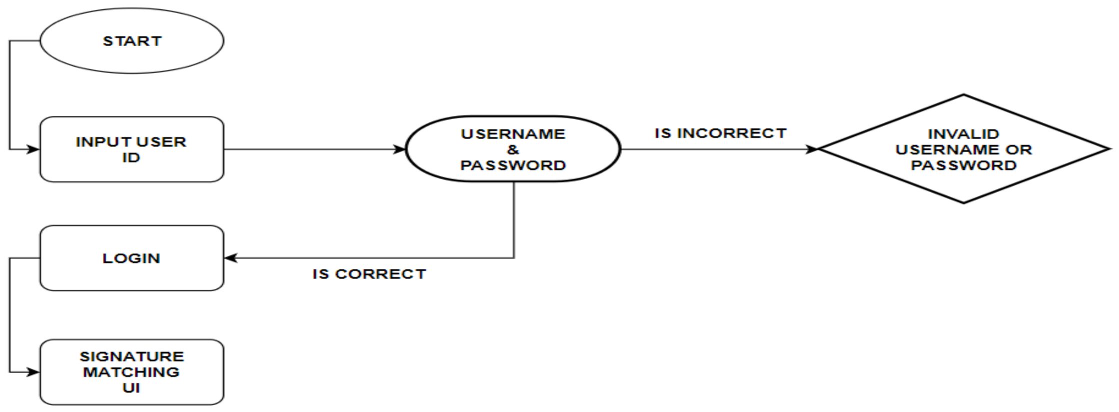


Fig.3.3.1 Login Work Flow Diagram

STEP 2:

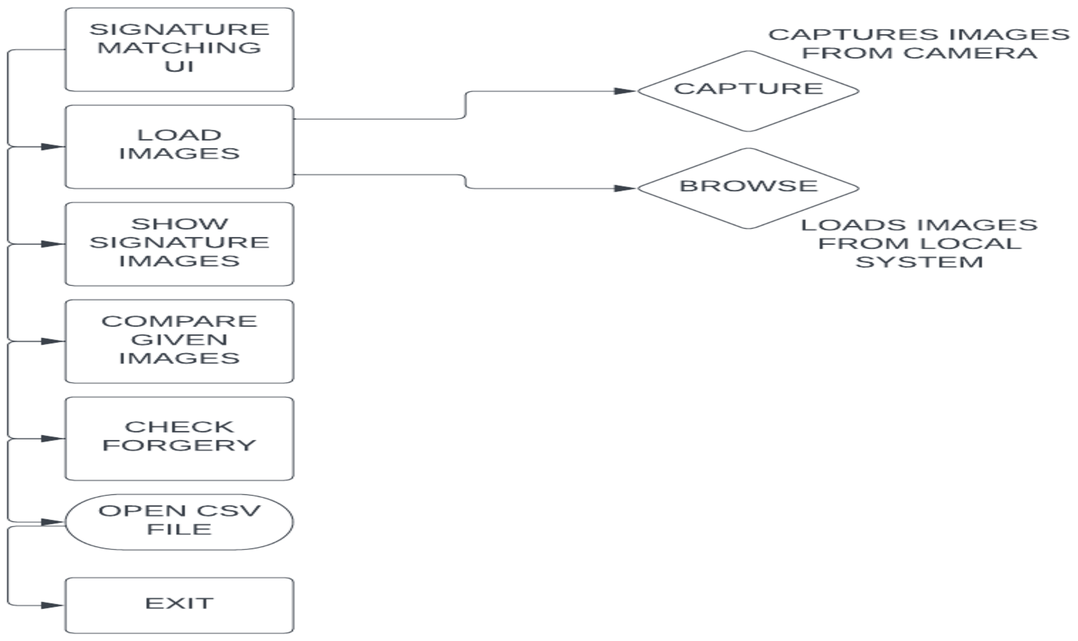


Fig.3.3.2 UI Work Flow Diagram

3.4 Objectives

The primary objective of a signature forgery detection system is to identify whether a given signature is genuine or fake. This can be achieved through a variety of techniques, including visual inspection, comparison with a known signature sample, and analysis of physical properties such as ink composition and paper type. In recent years, machine learning methods have become increasingly popular for signature forgery detection, allowing for the development of highly accurate and efficient systems.

SRVS (signature recognition and verification) is a system which is capable of addressing two individuals

a) Identification of the signature owner, and
b) Decision whether the signature is genuine or forger.

Here we see that signature verification are classified into two parts, according to the actual needs of the problem at hand)

a) Online signature verification
b) Offline signature verification

When a man is writing his signature, they gave more information like hand speed and pressure management and there are many things which we can take as key point to identify real or forged image.

Some of the key objectives of a signature forgery detection system include:

Prevention of fraud: By identifying fake signatures, signature forgery detection systems can help prevent fraudulent activities, such as financial scams and identity theft.

Protection of legal agreements: Signatures are often used to validate legal agreements, such as contracts and agreements. A signature forgery detection system can ensure that these agreements are legitimate and protect individuals and organizations from legal disputes.

Increased efficiency: Signature forgery detection systems can process large volumes of signatures quickly and accurately, improving the efficiency of signature verification processes.

Adaptability to new forms of forgery: Machine learning-based signature forgery detection systems can learn and adapt to new forms of forgery, making them a powerful tool in the fight against fraud.

Overall, the objectives of a signature forgery detection system are to improve security, prevent fraud, and protect individuals and organizations from financial losses and legal disputes.

- ✞ To verify if a given signature is forged or genuine.
- ✞ To implement the system.
- ✞ Using Signature Recognition Verification System also Known as SRVS, is a system which is capable of addressing two individual

1. Identification of the signature owner,
2. Decision whether the signature is genuine or forged.

- ✞ Here we see that signature verification are classified into two parts , according to the actual needs of the problem at hand.

1. Online signature verification
2. Offline signature verification

- ✞ When a user is writing his/her signature, they give more information like hand speed and pressure management and there are many things which we can take as a key point to identify real or forged image.
- ✞ This project will help to solve the problem of forged signature.

3.5 Requirement Specification

Data Collection and Pre-processing: The system should be able to collect a dataset of signature images, including both genuine and forged signatures.Data pre-processing capabilities should include image cleaning, normalization, and resizing to ensure consistency and suitability for deep learning and SSIM-based comparison.

Deep Learning Model Implementation: The system must implement a deep learning model, such as Convolutional Neural Networks (CNNs), to learn features and patterns from signature images for accurate comparison and detection.Utilization of a pre-trained model or custom model architecture should be considered based on the project requirements.

Structural Similarity Index (SSIM) Integration: Integration of the SSIM algorithm is necessary to measure the structural similarity between signature images, providing an additional metric for comparison alongside deep learning techniques.The system should calculate SSIM scores between pairs of signature images to assess their similarity.

Accuracy and Reliability: The system should aim to achieve a high level of accuracy and reliability in both signature comparison and forgery detection.Evaluation metrics, including precision, recall, and F1-score, should be utilized to assess the performance of the system and optimize its effectiveness.

User Interface:A user-friendly interface is essential for users to interact with the system effectively.The interface should allow users to upload signature images, initiate comparison and forgery detection processes, and visualize the results in a clear and understandable format.

Security: Security measures should be implemented to protect sensitive signature data and ensure the integrity of the system.Access controls and encryption techniques should be employed to prevent unauthorized access or tampering with the system and its data.

Scalability and Flexibility: The system should be scalable to handle varying workloads and accommodate potential future expansion.Flexibility in terms of integrating with other systems or adapting to different environments should be considered for enhanced usability and applicability.

Documentation and Training: Comprehensive documentation should be provided, covering system functionalities, usage instructions, and technical details for users and developers.

Training sessions or materials should be available to educate users on how to effectively utilize the system and interpret its results.

By fulfilling these requirements, the Signature Comparison and Forgery Detection System utilizing Deep Learning and SSIM can effectively compare signatures, detect forgeries, and contribute to fraud prevention efforts in various domains.

3.6 Advantage of proposed System :

High Accuracy: By integrating both deep learning techniques and SSIM, the system can achieve a high level of accuracy in identifying forged signatures. Deep learning models can effectively learn complex patterns and features from signature images, while SSIM provides a quantitative measure of similarity, enhancing the overall accuracy of the detection process.

Cost-effective: The proposed system can save time and money that would otherwise be spent manually inspecting and verifying signatures, making it a cost-effective solution for businesses and organizations

Robustness: The combination of deep learning and SSIM makes the system robust against various types of forgeries and image distortions. Deep learning models can adapt to different styles of signatures and variations in writing, while SSIM accounts for structural differences between genuine and forged signatures, making the system less susceptible to noise and alterations.

Efficiency: Deep learning-based signature comparison allows for efficient processing of large volumes of signature images. Once trained, the model can quickly analyze and compare signatures, enabling rapid forgery detection without compromising accuracy.

Versatility: The system is versatile and can be applied to different types of signature verification tasks, including online and offline verification. It can be tailored to suit various industries and applications where signature authentication is critical, such as banking, legal, and document authentication.

User-Friendly Interface: With a user-friendly interface, the system makes it easy for users to interact with and utilize its capabilities. Users can upload signature images, initiate comparison and forgery detection processes, and interpret the results in a straightforward manner, without requiring extensive technical expertise.

Enhanced Security: The system prioritizes data security and integrity, safeguarding sensitive signature data from unauthorized access or tampering. Robust security measures, such as access controls and encryption techniques, are implemented to ensure the confidentiality and integrity of the system and its data.

Customizable: The proposed system can be customized to meet the specific needs of different organizations, making it a versatile solution that can be adapted to various scenarios.

Scalability: The system is designed to scale efficiently, allowing it to handle increasing workloads and accommodate future growth. It can adapt to evolving requirements and integrate seamlessly with other systems or applications, ensuring long-term viability and scalability.

Overall, the proposed system offers a comprehensive solution for signature comparison and forgery detection, leveraging advanced techniques to deliver accurate, efficient, and secure results.

3.7 SSIM (Structural Similarity Index Measure) and VGG16 In Signature Image Recognition

SSIM (Structural Similarity Index Measure) in Signature Image Recognition:
SSIM, or Structural Similarity Index Measure, stands as a pivotal metric in signature image recognition, providing a quantitative measure of similarity between two images. Its significance lies in evaluating the structural information present in images, encompassing luminance, contrast, and structural details. In the realm of signature image recognition, SSIM plays a crucial role in determining the likeness between a queried signature and a reference sample or database of known signatures.

Characterization of Signature Similarity: SSIM enables the comparison of signature images by assessing their structural resemblance. This includes scrutinizing the arrangement of pixels, intensity distribution, and overall spatial coherence, factors essential for authenticating signatures and detecting potential forgeries.

Quantitative Assessment of Signature Alterations: By computing the SSIM score between two signature images, even subtle alterations or distortions can be quantified. This facilitates the identification of discrepancies or irregularities, aiding in the detection of fraudulent activities or unauthorized modifications.

Robustness Across Image Variations: SSIM exhibits robustness against variations in signature images arising from factors such as noise, lighting conditions, or scaling. Its ability to capture structural similarities ensures reliable performance across diverse image qualities and environments.

Integration in Forgery Detection Systems: SSIM serves as a fundamental component in forgery detection systems, contributing to the authentication and validation of signatures in legal documents, financial transactions, or identity verification processes.

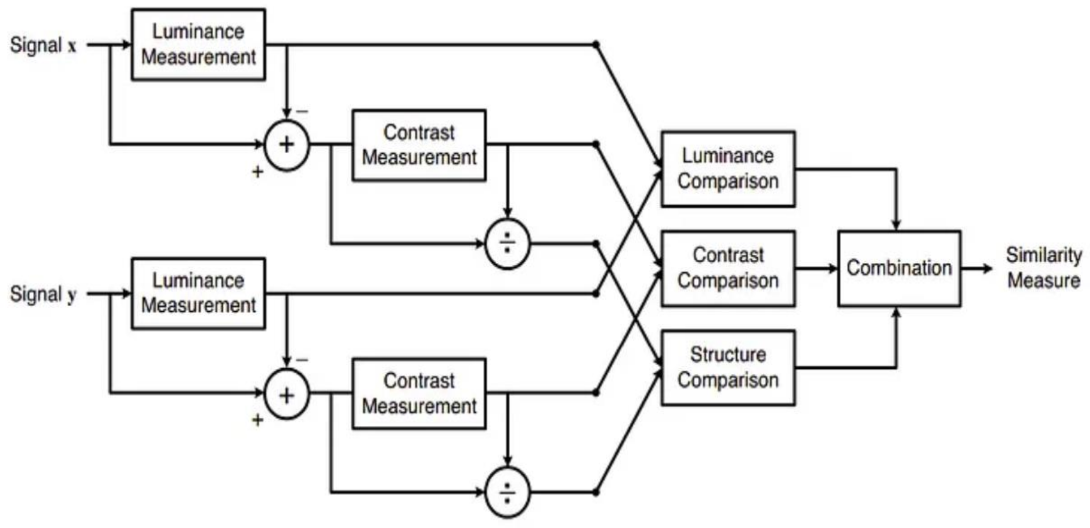


Fig.3.7.1 SSIM Work Flow Diagram

VGG16 in Signature Image Recognition:

VGG16, a convolutional neural network (CNN) architecture renowned for its prowess in image classification tasks, holds significant promise in signature image recognition endeavors. Its application extends beyond mere image categorization to encompass feature extraction and representation learning, making it instrumental in the analysis and interpretation of signature images.

Feature Extraction for Signature Representation: VGG16 acts as a potent feature extractor, discerning salient visual features from signature images. By traversing through the network's layers, intricate details and distinctive patterns inherent in signatures are captured, facilitating robust representation and characterization.

Transfer Learning for Signature Analysis: Leveraging pretrained weights from large-scale image datasets, such as ImageNet, VGG16 harnesses transfer learning to imbue signature recognition systems with generic visual representations. This transferability empowers VGG16 to adapt to signature-specific tasks, enhancing its capability to discern nuances and variations in signature styles.

Enhanced Signature Verification and Classification: The extracted features from VGG16 serve as discriminative descriptors for signature verification and classification tasks. By harnessing the network's ability to discern intricate visual cues, signatures can be accurately authenticated, aiding in identity verification, document validation, and fraud prevention endeavors.

Versatility Across Signature Applications: VGG16's versatility extends across diverse signature applications, including document authentication, biometric security systems, and financial transactions. Its adaptability to varied signature styles, resolutions, and image qualities renders it indispensable in addressing the multifaceted challenges of signature image recognition.

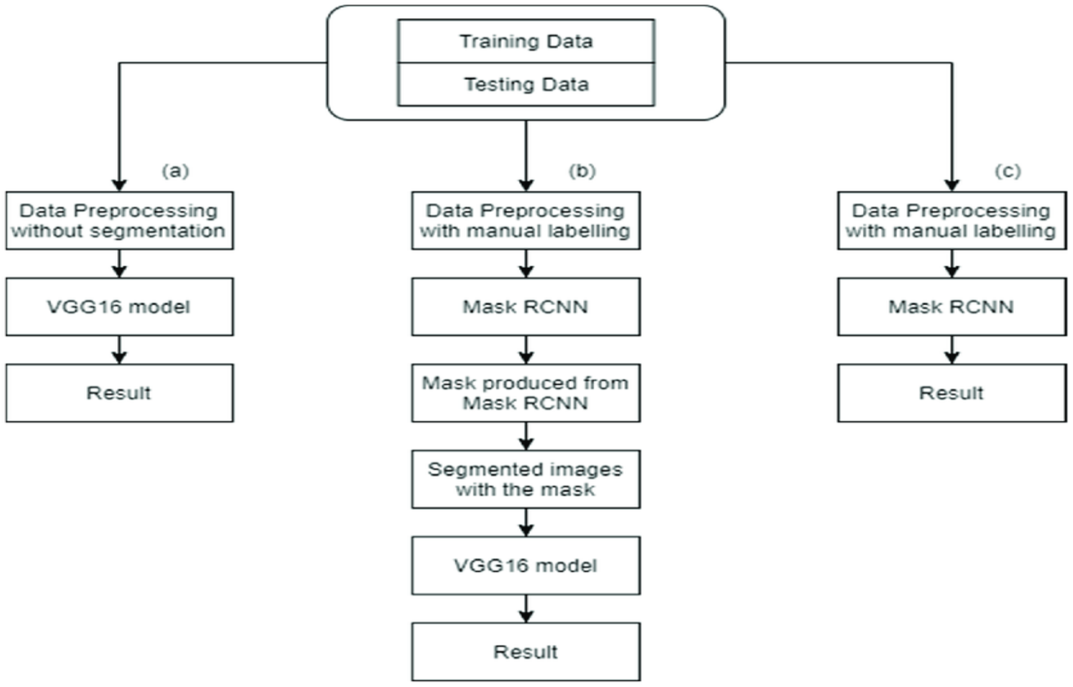

Training Data
Testing Data
(a)
(b)
(c)
Data Preprocessing without segmentation
VGG16 model
Result
Data Preprocessing with manual labelling
Mask RCNN
Mask produced from Mask RCNN
Segmented images with the mask
VGG16 model
Result
Data Preprocessing with manual labelling
Mask RCNN
Result

Fig.3.7.2 VGG16 Work Flow Diagram

the integration of SSIM (Structural Similarity Index Measure) and VGG16 in signature image recognition holds significant importance and plays several key roles:

Signature Authentication and Verification: SSIM provides a quantitative measure of similarity between queried and reference signature images, enabling robust authentication and verification processes. By computing the SSIM score, the system can assess the structural resemblance between signatures, aiding in the identification of genuine signatures and potential forgeries.

Feature Extraction and Representation Learning: VGG16 serves as a powerful feature extractor, capable of discerning intricate visual features from signature images. Through transfer learning, it can adapt pretrained weights to signature-specific tasks, facilitating the extraction of discriminative signatures descriptors for authentication and classification purposes.

Robustness to Variations: Both SSIM and VGG16 contribute to the system's robustness against variations in signature images, including noise, lighting conditions, and scaling. SSIM's ability to capture structural similarities and VGG16's feature extraction capabilities ensure reliable performance across diverse image qualities and environments.

Forgery Detection and Fraud Prevention: By leveraging SSIM and VGG16, your project can implement effective forgery detection and fraud prevention mechanisms. SSIM enables the quantitative assessment of signature alterations, while VGG16 enhances the system's ability to discern subtle discrepancies and irregularities indicative of fraudulent activities.

Versatility Across Applications: The integration of SSIM and VGG16 lends versatility to your project, extending its applicability across various signature recognition tasks. From document authentication and biometric security systems to financial transactions, the system can adapt to diverse signature styles, resolutions, and image qualities, addressing the multifaceted challenges of signature image recognition.

3.8 Tools & Technologies

The tools used in this project include Python , which are widely used in the data science community for machine learning purposes and code writing , building models and projects.

The libraries used for this project are numpy , os,cv2, matplotlib.py as plt, subprocess, Flask skimage, tensorflow, keras ,PIL,csv. For displaying images, we utilized OpenCV . To extract and store features, we used Excel/CSV files. The graphical user interface (GUI) for this project was created using Tkinter.

Python :

Python is a high-level, interpreted programming language that is widely used for web development, data analysis, artificial intelligence, and scientific computing. It was created by Guido van Rossum and first released in 1991.

Python is known for its simplicity, readability, and ease of use, which makes it a popular choice for beginners and experienced programmers alike. Its syntax is clear and concise, making it easy to write and read code.

Python has a large standard library that provides modules for a wide range of tasks such as working with databases, regular expressions, networking, and more. Additionally, there are thousands of third-party libraries and modules available for Python that can be easily installed using pip, the package manager for Python.

One of the key features of Python is its support for object-oriented programming, which allows developers to organize their code into reusable and modular components. It also supports functional programming, which enables developers to write code in a more declarative and concise way.

Python has a diverse and supportive community, with many online resources, forums, and tutorials available to help developers learn and solve problems. It is also an open-source language, which means that the source code is freely available to the public for use, modification, and distribution.

Overall, Python is a versatile and powerful programming language that is well-suited for a wide range of applications and projects**.**

OpenCv :

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library designed for processing images and videos in real-time. It was initially developed by Intel in 1999 and is now maintained by the OpenCV community.

The library provides a comprehensive set of tools and functions for various computer vision tasks, including object recognition, image processing, face detection, feature extraction, and more.

It is written in C++, but has bindings available for various programming languages, such as Python, Java, and MATLAB. OpenCV is widely used in the fields of robotics, augmented reality, and medical imaging, among others. It is a powerful tool for developing applications that require image or video processing, and is constantly updated with new features and improvements.

Numpy :

NumPy is a Python library used for working with arrays.
It also has functions for working in domain of linear algebra, fourier transform, and matrices.
NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
NumPy stands for Numerical Python.

In Python we have lists that serve the purpose of arrays, but they are slow to process.
NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.
NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.
This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C.

Os :

The OS module in Python provides functions for interacting with the operating system. OS comes under Python’s standard utility modules. This module provides a portable way of using operating system-dependent functionality. The *os* and *os.path* modules include many functions to interact with the file system.

OS is a short form for Operating systems. OS comes under Python’s standard utility modules. It helps to interact with the OS directly from within the Jupyter Notebook. It makes it possible to perform many operating system tasks automatically.

This module in Python has functions for creating a directory, showing its contents, showing the current directory, and also functions to change the current directory, and many more
It is possible to automatically perform many operating system tasks.

The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing an identifying the current directory, etc.The getcwd() function confirms returns the current working directory.

We can create a new directory using the os.mkdir() function.

The rmdir() function in the OS module removes the specified directory either with an absolute or relative path. Note that, for a directory to be removed, it should be empty.

The listdir() function returns the list of all files and directories in the specified directory.

Subprocess:

The subprocess module in Python is a built-in module that allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. It provides a way to execute system commands from within Python scripts.

With subprocess, you can run external programs, interact with them by sending input, and capture their output. This module is particularly useful when you need to run commands or scripts that are external to your Python environment, such as running shell commands, executing other Python scripts, or running system utilities.

Some common use cases of subprocess include:
Running shell commands or scripts.
Capturing the output of external commands or scripts.
Interacting with command-line utilities.
Running processes in the background.
Redirecting input/output/error streams.
Overall, subprocess provides a powerful way to interact with external processes and integrate them into your Python applications.

Matplotlib (plt):

Matplotlib is a low level graph plotting library in python that serves as a visualization utility. Matplotlib was created by John D. Hunter. Matplotlib is open source and we can use it freely. Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It is widely used for generating plots, histograms, power spectra, bar charts, error charts, scatterplots, etc. within Python applications.

In the context of the code provided, plt is an alias commonly used for the matplotlib.pyplot module, which provides a MATLAB-like interface for creating plots and visualizations. By importing matplotlib.pyplot as plt, you can access its functionality using the shorter alias plt, making the code more concise and readable.

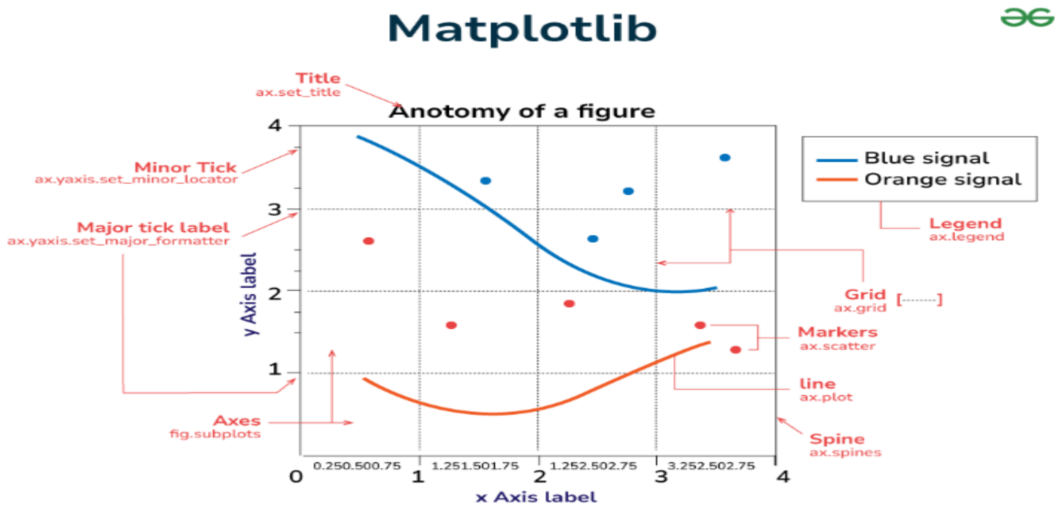


Fig.3.8.1 Matplotlib Diagram

SkImage :

Scikit-image (skimage) is an open-source image processing library in Python that provides various algorithms for image processing and computer vision tasks. It is built on top of NumPy, SciPy, and matplotlib, and can be used to perform various image processing tasks such as filtering, segmentation, feature extraction, and image enhancement.

Some of the key features of skimage include:
Image Transformation: skimage provides a number of image transformation functions such as resizing, rotating, and cropping images. Image Filters: skimage provides a number of image filters such as Gaussian, Sobel, and Median filters, which can be used to smooth, sharpen, or detect edges in images.

Image Segmentation: skimage provides a number of segmentation algorithms such as watershed, thresholding, and region growing, which can be used to separate an image into different regions or objects.

Feature Detection and Extraction: skimage provides a number of feature detection and extraction algorithms such as SIFT, SURF, and HOG, which can be used to extract useful features from images.Image Visualization: skimage provides a number of functions to visualize images and their properties, such as histograms, contours, and color maps.

Overall, skimage is a powerful image processing library that can be used for a wide range of tasks in computer vision and image processing. It is easy to use, well-documented, and has an active community of developers who contribute to its development and maintenance.

PIL (Python Imaging Library):

The Python Imaging Library (PIL) is a library in Python used for opening, manipulating, and saving many different image file formats. It provides a powerful set of tools for image processing tasks, including resizing, cropping, rotating, filtering, enhancing, and converting images between different formats.

PIL is widely used in various applications, including computer vision, graphics, web development, and scientific computing. It offers a simple and intuitive interface for working with images and is supported across different platforms, including Windows, macOS, and Linux.

It's worth noting that PIL was initially developed by Fredrik Lundh and later maintained by the Python community. However, the original PIL library is no longer actively maintained. Instead, there is a fork of PIL called Pillow, which is the modern and actively maintained version.

Pillow aims to provide backward compatibility with PIL while adding new features and improvements. Many Python users now prefer to use Pillow for image processing tasks due to its continued development and support.

Tensorflow :

TensorFlow is an open-source software library used for numerical computation and machine learning tasks. It was developed by Google Brain team and released under the Apache 2.0 open source license.

TensorFlow is designed to work with complex mathematical operations, such as matrix multiplication and convolutional neural networks. It allows users to build and train machine learning models, from simple linear regression to complex deep learning models, using a variety of programming languages including Python, C++, and Java.

The library consists of two main components: the Data Flow Graph and the TensorFlow runtime. The Data Flow Graph is a symbolic representation of the mathematical operations in the model, while the TensorFlow runtime executes the operations efficiently on CPU or GPU hardware.
TensorFlow has a large and active community of developers who contribute to the library and create useful tools and libraries for data processing, model visualization, and deployment.

TensorFlow is widely used in various applications including image recognition, natural language processing, speech recognition, and reinforcement learning.

Keras :

Keras

Keras is an open-source deep learning framework written in Python. It is a high-level neural networks API that enables developers to easily build and train deep learning models. Keras was developed with a focus on enabling fast experimentation, and it has become a popular choice for researchers and practitioners in the field of machine learning.

One of the key features of Keras is its user-friendliness. It allows developers to define and train deep learning models with just a few lines of code, without requiring a deep understanding of the underlying mathematical concepts. This makes it easy for developers to get started with deep learning and experiment with different model architectures and hyperparameters.

Keras supports multiple backends, including TensorFlow, Microsoft Cognitive Toolkit, Theano, and PlaidML. This enables developers to choose the backend that best suits their needs and take advantage of the optimizations provided by each backend.

Keras also supports a wide range of deep learning model types, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and even custom model architectures. Additionally, Keras includes a number of pre-trained models that can be fine-tuned for specific use cases.

Tkinter:

TKINTER
GUI FOR PYTHON

Tkinter is a standard Python library used for creating graphical user interfaces (GUIs). It provides a simple and easy-to-use way to build desktop applications with a variety of widgets such as buttons, labels, text boxes, and more. Tkinter is based on the Tk GUI toolkit, which is commonly used for building GUI applications.

With Tkinter, developers can create interactive applications that run on various platforms, including Windows, macOS, and Linux. It offers a set of modules and classes for constructing GUI elements and handling user events, making it suitable for both beginner and experienced developers. Tkinter is widely used in the Python community for developing desktop applications due to its simplicity and versatility

FLASK:

Flask is a lightweight web framework for Python, designed to make it easy to build web applications quickly and with minimal code. It is known for its simplicity, flexibility, and ease of use, making it a popular choice for developing web applications and APIs.

Key features of Flask include:

Minimalism: Flask has a simple and easy-to-understand core, allowing developers to get started quickly without unnecessary overhead.

Flexibility: Flask allows developers to choose the components they need, making it easy to integrate with other libraries and tools.

Routing: Flask provides a built-in routing system that allows developers to define URL routes and map them to view functions, making it easy to handle different HTTP requests.

Template Engine: Flask comes with a built-in template engine called Jinja2, which allows developers to create dynamic HTML pages by combining static HTML with dynamic content.

HTTP Request Handling: Flask provides tools for handling HTTP requests and responses, including support for cookies, sessions, and file uploads.

Extensions: Flask has a rich ecosystem of extensions that add additional functionality to the framework, such as support for databases, authentication, and more.

Overall, Flask is a versatile and lightweight framework that is well-suited for building a wide range of web applications, from simple websites to complex web services.

4. Algorithm & Methodology

Methodology

The signature verification system begins with the user choosing between capturing an image using the camera or browsing an image from the device's storage. If capturing, the system saves the image into a temporary directory. For browsing, the user selects an image file and provides its path. The chosen image then proceeds to the pre-processing stage, where it is resized and converted to RGB format. Subsequently, object detection is performed using a model such as YOLO or SSD to detect and extract signature regions from the image. The user is then prompted to either compare the extracted signature with reference signatures or calculate the Structural Similarity Index (SSIM). If comparison is chosen, the system retrieves the reference signatures from the database and calculates the similarity using VGG16 features. The result is displayed to the user, and the system returns to the main app interface, allowing for another operation or exit.

Below diagram shows the Process flow:

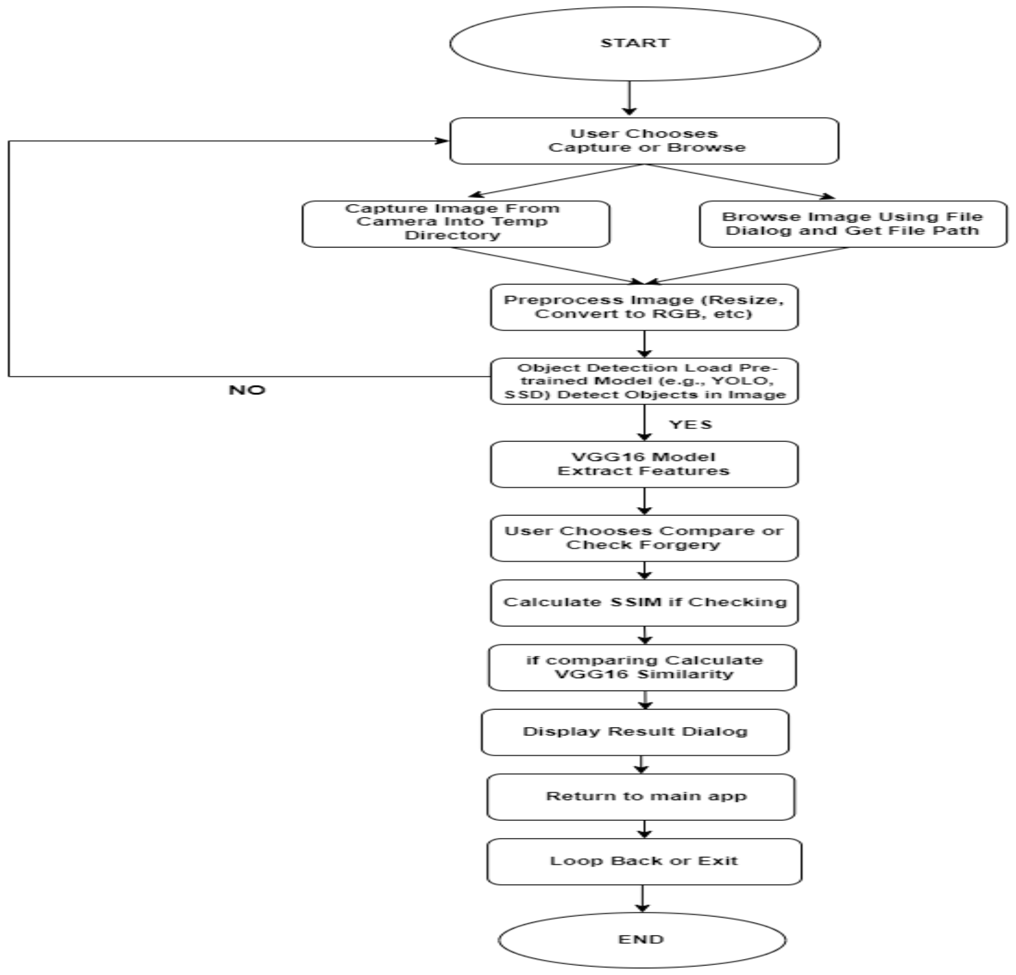
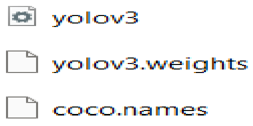

START
User Chooses Capture or Browse
Capture Image From Camera Into Temp Directory
Browse Image Using File Dialog and Get File Path
Preprocess Image (Resize, Convert to RGB, etc)
Object Detection Load Pre-trained Model (e.g., YOLO, SSD) Detect Objects in Image
NO
YES
VGG16 Model Extract Features
User Chooses Compare or Check Forgery
Calculate SSIM if Checking
if comparing Calculate VGG16 Similarity
Display Result Dialog
Return to main app
Loop Back or Exit
END

Fig.4.1 Signature Forgery Detection Workflow

4.1 Datasets

coco.names: A file containing the class names used by the YOLO model, part of the COCO dataset.
yolov3: The configuration file for the YOLOv3 model, specifying the architecture of the model.
yolov3.weights: A file containing the pre-trained weights for the YOLOv3 model, used for object detection.


yolov3
yolov3.weights
coco.names

The datasets utilized for this research has been sourced from Github Link which contains 2 folders namely Real and forged which contains real as well as forged images respectively.
The real folder contains per user 5 samples of signature marked as 000, 001,002 etc and same for forged folder where it contains fake signatures respectively.
each real and forged folder contain 60 images in each total of 120 images.

forged
real

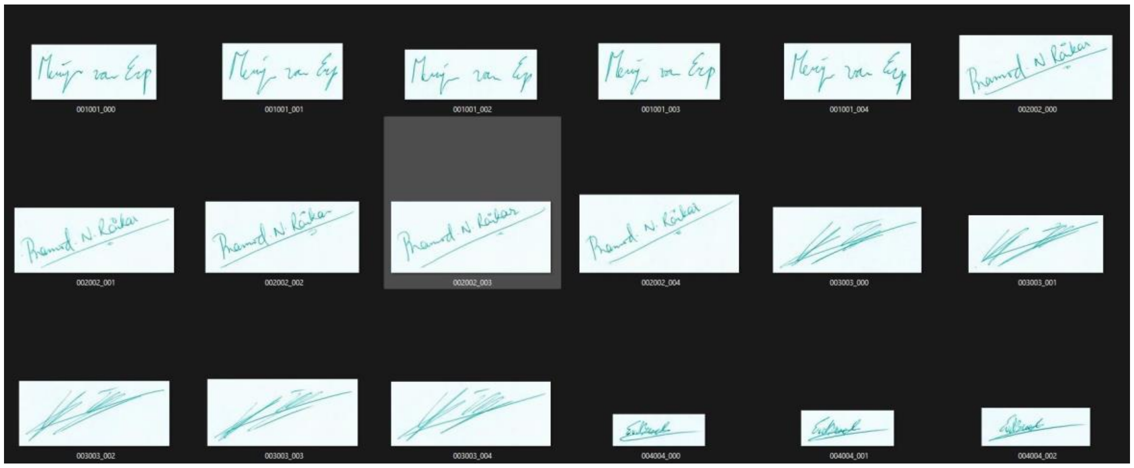


Fig.4.1.1 Real folder images

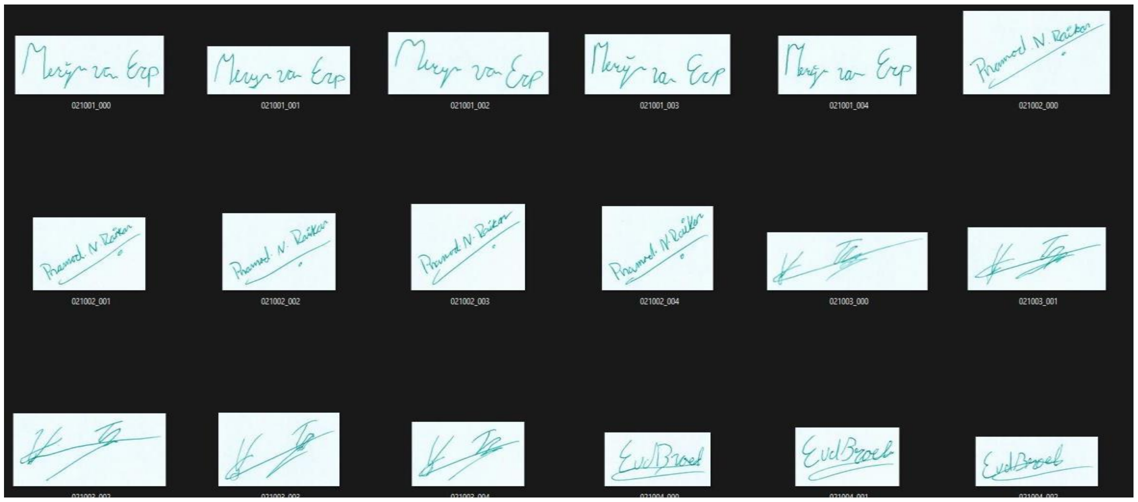
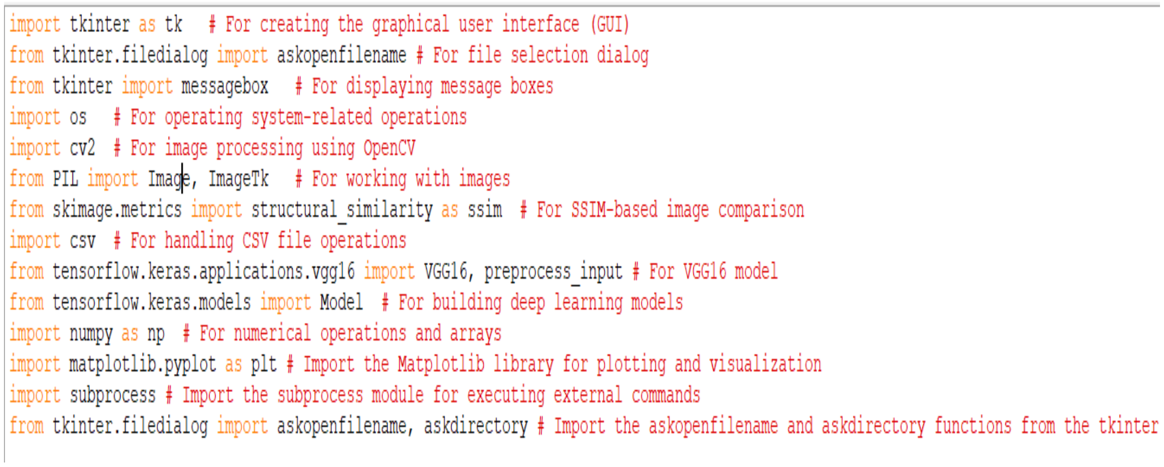


Fig.4.1.2 Forged folder images

4.2 Importing Libraries

Our next step is to import all the necessary libraries required for the project, such as numpy , os,cv2, matplotlib.py as plt, subprocess, skimage, tensorflow, keras ,PIL,csv This code imports the required libraries for the forgery detection model.



```
import tkinter as tk   # For creating the graphical user interface (GUI)
from tkinter.filedialog import askopenfilename # For file selection dialog
from tkinter import messagebox   # For displaying message boxes
import os   # For operating system-related operations
import cv2  # For image processing using OpenCV
from PIL import Image, ImageTk   # For working with images
from skimage.metrics import structural_similarity as ssim  # For SSIM-based image comparison
import csv  # For handling CSV file operations
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input # For VGG16 model
from tensorflow.keras.models import Model  # For building deep learning models
import numpy as np  # For numerical operations and arrays
import matplotlib.pyplot as plt # Import the Matplotlib library for plotting and visualization
import subprocess # Import the subprocess module for executing external commands
from tkinter.filedialog import askopenfilename, askdirectory # Import the askopenfilename and askdirectory functions from the tkinter
```

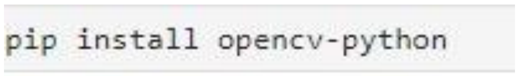
Fig.4.2.1 Importing Libraries

numpy: A fundamental package for scientific computing in Python, providing support for arrays and mathematical functions.
os: A standard library for interacting with the operating system, including file and directory manipulation.
cv2 (OpenCV): An open-source computer vision and machine learning software library that provides tools for image and video processing.
matplotlib.pyplot (plt): A plotting library for creating static, animated, and interactive visualizations in Python.

subprocess: A standard library for spawning and managing additional processes, and retrieving their results.
skimage (scikit-image): A collection of algorithms for image processing in Python, part of the scikit-learn library.
tensorflow: for building and training machine learning models.
keras: for building and training deep learning models.
PIL (Pillow): A Python Imaging Library that adds image processing capabilities to Python, including opening, manipulating, and saving images.
csv: A module for reading and writing CSV (Comma Separated Values) files, used for data exchange between systems.

4.3 Installing OpenCv

Installing OpenCv Python



```
pip install opencv-python
```

This code installs the "opencv-python" package using pip. OpenCV (Open Source Computer Vision) is a popular computer vision library used for real-time computer vision, image processing, and machine learning. The "opencv-python" package provides Python bindings for the OpenCV library, making it easy to use OpenCV functions and methods in Python code.

Features of pip install :

A. Install packages from:
B. PyPI (and other indexes) using requirement specifiers.
C. VCS project urls.
D. Local project directories.
E. Local or remote source archives.

Pip also supports installing from “requirements files”, which provide an easy way to specify a whole environment to be installed.

Pip install has several stages:

a) Identify the base requirements. The user supplied arguments are processed here.
b) Resolve dependencies. What will be installed is determined here.
c) Build wheels. All the dependencies that can be are built into wheels.
d) Install the packages (and uninstall anything being upgraded/replaced).

4.4 Defining Genuine & Forged Image Paths

The purpose of our project is to develop a robust system for detecting and comparing signatures to determine their authenticity. This involves processing and analyzing both genuine and potentially forged signatures. The steps outlined below describe how our application manages and utilizes these images.

1. Genuine Image Path

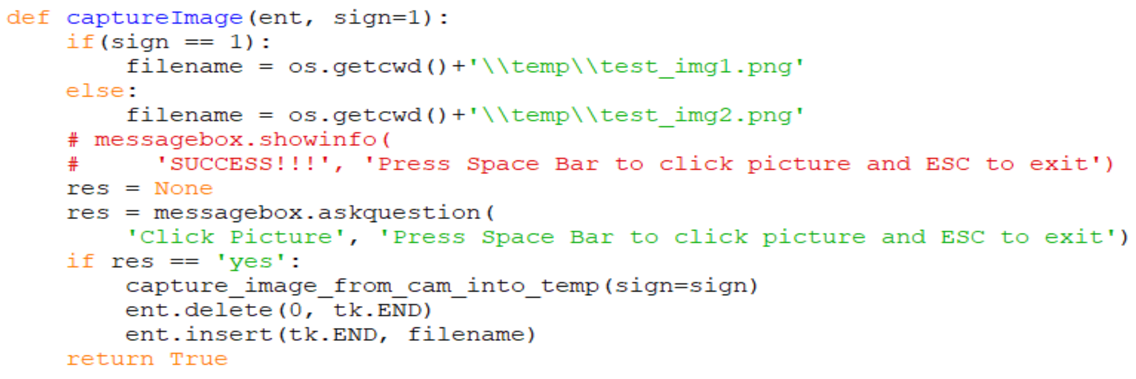
A genuine image path is defined as the file path to an image that contains a verified, authentic signature. This path is essential for our system to have a reliable reference against which other signatures are compared. The genuine image path is typically provided by the user through either direct image capture or file browsing.

2. Forged Image Path

A forged image path refers to the file path of an image suspected of containing a fraudulent or forged signature. This path is crucial for comparison purposes to identify discrepancies and confirm the authenticity of the signature. The forged image path can also be defined through direct image capture or file browsing, similar to the genuine image path.

Steps to Define Genuine Image Path:

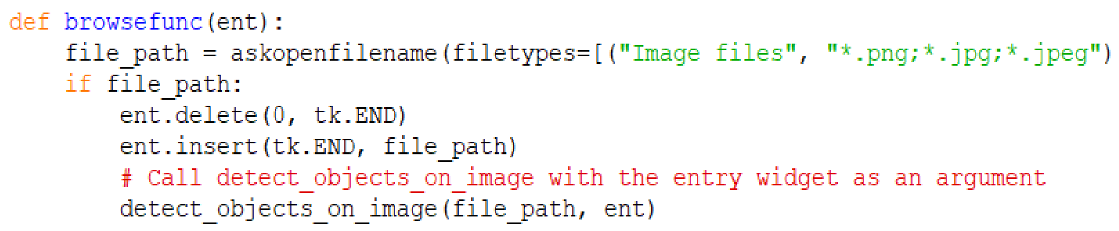
Image Capture: Users can capture a genuine signature using their webcam. The captured image is stored temporarily and its path is defined within the application.



```
def captureImage(ent, sign=1):
    if(sign == 1):
        filename = os.getcwd()+'\\temp\\test_img1.png'
    else:
        filename = os.getcwd()+'\\temp\\test_img2.png'
    # messagebox.showinfo(
    #     'SUCCESS!!!', 'Press Space Bar to click picture and ESC to exit')
    res = None
    res = messagebox.askquestion(
        'Click Picture', 'Press Space Bar to click picture and ESC to exit')
    if res == 'yes':
        capture_image_from_cam_into_temp(sign=sign)
        ent.delete(0, tk.END)
        ent.insert(tk.END, filename)
    return True
```

Fig.4.4.1 Image Capture Code

File Browsing: Users can browse and select an existing image file that contains the genuine signature.



```
def browsefunc(ent):
    file_path = askopenfilename(filetypes=[("Image files", "*.png;*.jpg;*.jpeg")])
    if file_path:
        ent.delete(0, tk.END)
        ent.insert(tk.END, file_path)
        # Call detect_objects_on_image with the entry widget as an argument
        detect_objects_on_image(file_path, ent)
```

Fig.4.4.2 File Browsing code

3. Path Management

The application handles both genuine and forged image paths seamlessly through a graphical user interface (GUI). Users can interact with the application to define these paths using buttons for capturing images or browsing files. The paths are then used for subsequent processing steps, such as signature comparison and forgery detection.
Example of GUI Components for Path Management:

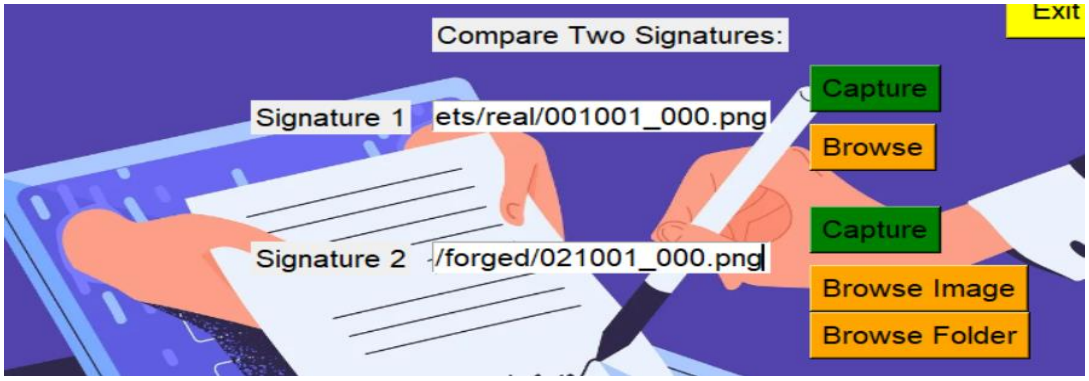
Capture Button: Allows users to capture signatures using the webcam.

```
img1_capture_button = tk.Button(root, text="Capture", command=lambda: captureImage(ent=image1_path_entry, sign=1))
img2_capture_button = tk.Button(root, text="Capture", command=lambda: captureImage(ent=image2_path_entry, sign=2))
```

Browse Button: Allows users to browse and select image files.

```
img1_browse_button = tk.Button(root, text="Browse", command=lambda: browsefunc(ent=image1_path_entry))
img2_browse_button = tk.Button(root, text="Browse Image", command=lambda: browsefunc(ent=image2_path_entry))
```

By clearly defining and managing the genuine and forged image paths, our system ensures accurate and reliable signature comparison, enhancing its capability to detect and prevent forgery effectively.



Signature 1 ets/real/001001_000.png
Signature 2 /forged/021001_000.png

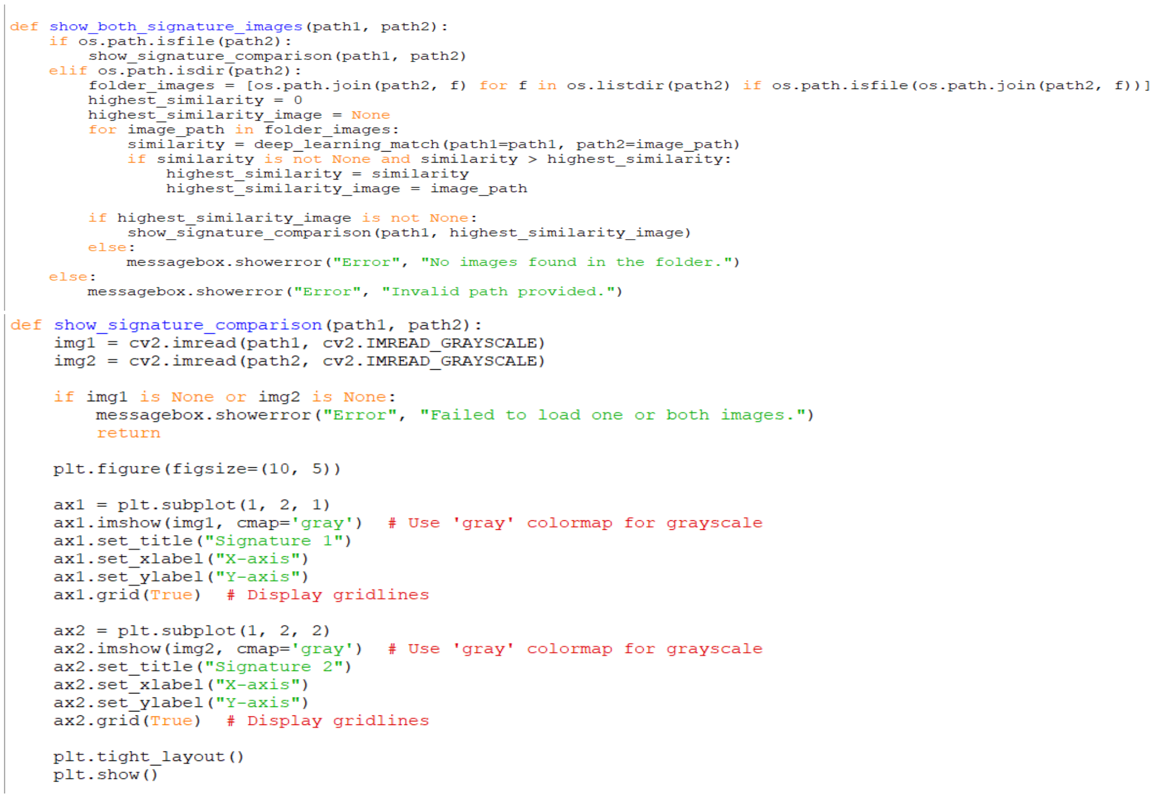
Fig.4.4.3 DISPLAY Genuine & Forged Image Paths

4.5 Displaying Images Using Opencv

In our signature matching application, the ability to display signature images is essential for visualizing signatures and comparing them for similarity. We leverage the OpenCV library, a powerful computer vision library in Python, to handle image display functionalities seamlessly within our Tkinter-based graphical user interface (GUI).

Loading and Displaying Signature Images:

To load and display signature images, we utilize OpenCV's cv2.imread() function to read images from file paths and cv2.imshow() function to display them within a window. Here's how we integrate this functionality into our application:

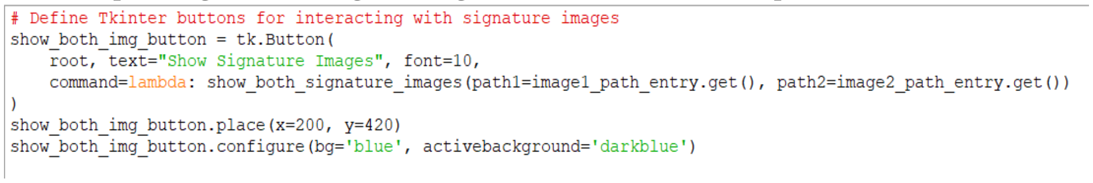


```python
def show_both_signature_images(path1, path2):
    if os.path.isfile(path2):
        show_signature_comparison(path1, path2)
    elif os.path.isdir(path2):
        folder_images = [os.path.join(path2, f) for f in os.listdir(path2) if os.path.isfile(os.path.join(path2, f))]
        highest_similarity = 0
        highest_similarity_image = None
        for image_path in folder_images:
            similarity = deep_learning_match(path1=path1, path2=image_path)
            if similarity is not None and similarity > highest_similarity:
                highest_similarity = similarity
                highest_similarity_image = image_path

        if highest_similarity_image is not None:
            show_signature_comparison(path1, highest_similarity_image)
        else:
            messagebox.showerror("Error", "No images found in the folder.")
    else:
        messagebox.showerror("Error", "Invalid path provided.")

def show_signature_comparison(path1, path2):
    img1 = cv2.imread(path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(path2, cv2.IMREAD_GRAYSCALE)

    if img1 is None or img2 is None:
        messagebox.showerror("Error", "Failed to load one or both images.")
        return

    plt.figure(figsize=(10, 5))

    ax1 = plt.subplot(1, 2, 1)
    ax1.imshow(img1, cmap='gray')  # Use 'gray' colormap for grayscale
    ax1.set_title("Signature 1")
    ax1.set_xlabel("X-axis")
    ax1.set_ylabel("Y-axis")
    ax1.grid(True)  # Display gridlines

    ax2 = plt.subplot(1, 2, 2)
    ax2.imshow(img2, cmap='gray')  # Use 'gray' colormap for grayscale
    ax2.set_title("Signature 2")
    ax2.set_xlabel("X-axis")
    ax2.set_ylabel("Y-axis")
    ax2.grid(True)  # Display gridlines

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

Fig.4.5.1 Displaying Images Code

Integrating Image Display with Tkinter GUI :

We seamlessly integrate the image display functionalities within our Tkinter GUI, allowing users to interact with the application conveniently. Users can load, capture, and compare signature images using intuitive buttons and input fields.

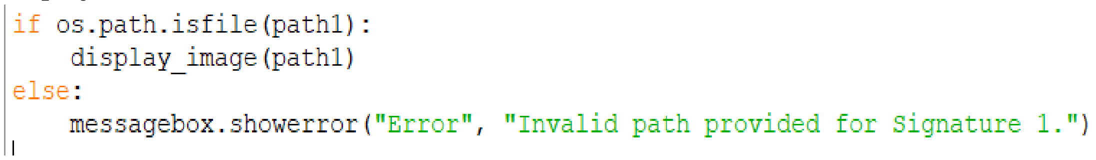


```python
# Define Tkinter buttons for interacting with signature images
show_both_img_button = tk.Button(
    root, text="Show Signature Images", font=10,
    command=lambda: show_both_signature_images(path1=image1_path_entry.get(), path2=image2_path_entry.get())
)
show_both_img_button.place(x=200, y=420)
show_both_img_button.configure(bg='blue', activebackground='darkblue')
```

Fig.4.5.2 Show button Code

User Interaction and Feedback :

Throughout the image display process, we provide informative feedback to the user using message boxes and console outputs. Robust error handling mechanisms ensure a smooth user experience, even in scenarios where images cannot be loaded or displayed.



```python
if os.path.isfile(path1):
    display_image(path1)
else:
    messagebox.showerror("Error", "Invalid path provided for Signature 1.")
```

Supporting Multiple Platforms :

Our application is designed to run seamlessly on various platforms, including Windows, macOS, and Linux. OpenCV ensures cross-platform compatibility, allowing users to utilize the application on their preferred operating system without any constraints.

Overall We use tkinter library to display images within the graphical user interface (GUI).

the show_both_signature_images function takes the paths of two signature images as input. It opens and resizes the images using the Image.open function from the PIL library. Then, it converts the resized images to ImageTk.PhotoImage objects, which are suitable for displaying images in the GUI. Two tk.Label widgets are created to display the images on the GUI. The .place() method specifies the position of each image label within the GUI window.

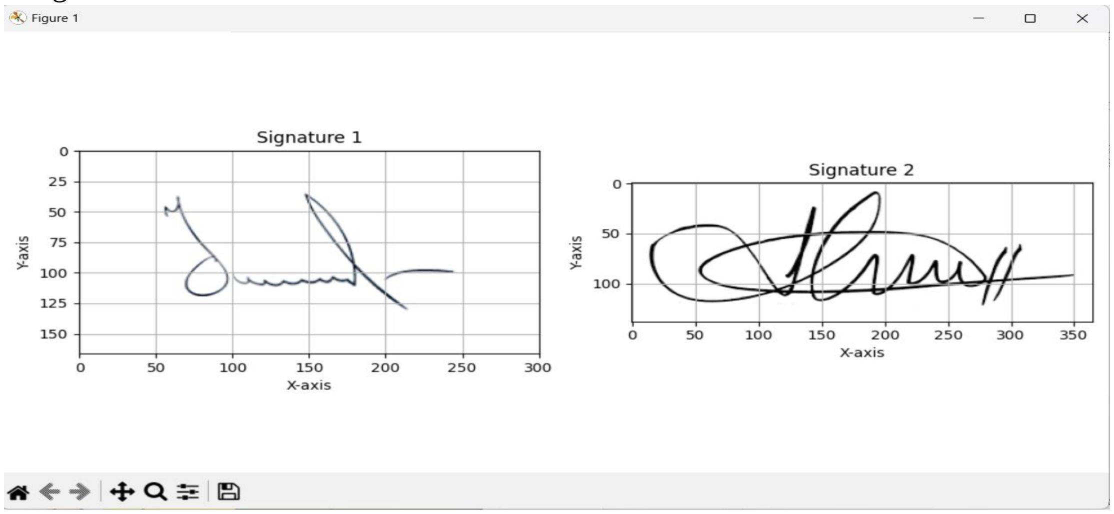


Fig.4.5.3 DISPLAY IMAGES VIA TKINTER

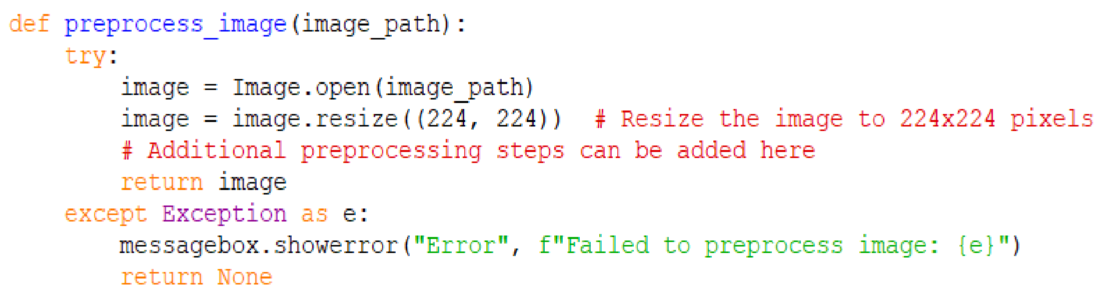
4.6 PreProcessing The Images

Preprocessing plays a crucial role in enhancing the quality of signature images and improving the accuracy of signature matching algorithms. In this section, we discuss the preprocessing techniques implemented in our signature matching application

.

Resizing and Standardizing Signature Images:

Before performing any analysis or comparison, we preprocess signature images by resizing them to a standard size. Standardizing the size ensures uniformity and consistency across all images, facilitating accurate comparison. We utilize the resize() function from the Python Imaging Library (PIL) to resize signature images to the desired dimensions.

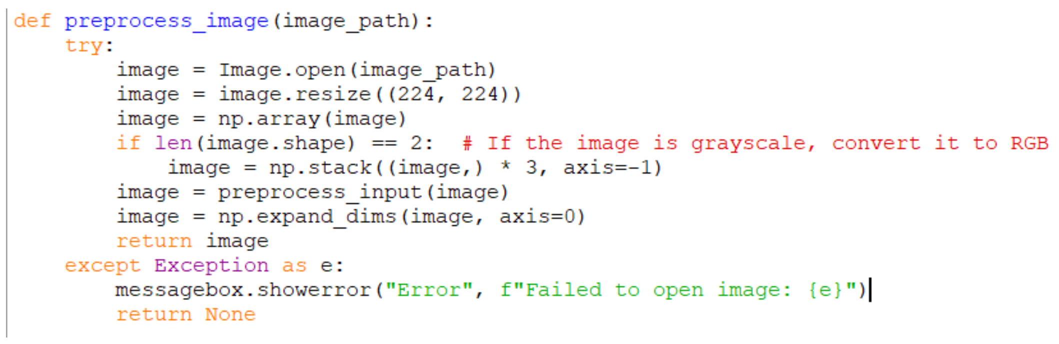


```
def preprocess_image(image_path):
    try:
        image = Image.open(image_path)
        image = image.resize((224, 224))  # Resize the image to 224x224 pixels
        # Additional preprocessing steps can be added here
        return image
    except Exception as e:
        messagebox.showerror("Error", f"Failed to preprocess image: {e}")
        return None
```

Fig.4.6.1 Preprocess image Resizing code

Grayscale Conversion:

To simplify image processing and reduce computational overhead, we convert signature images to grayscale. Grayscale images contain only intensity values, eliminating color information and reducing the complexity of subsequent operations. We utilize OpenCV's cv2.cvtColor() function to perform grayscale conversion.



```
def preprocess_image(image_path):
    try:
        image = Image.open(image_path)
        image = image.resize((224, 224))
        image = np.array(image)
        if len(image.shape) == 2:  # If the image is grayscale, convert it to RGB
            image = np.stack((image,) * 3, axis=-1)
        image = preprocess_input(image)
        image = np.expand_dims(image, axis=0)
        return image
    except Exception as e:
        messagebox.showerror("Error", f"Failed to open image: {e}")
        return None
```

Fig.4.6.2 Preprocess image Grayscale Conversion code

This code defines a function named "rgbgrey" that takes an RGB image as input and converts it to grayscale.

The function first creates a 2D array of zeros with the same dimensions as the input image using the "np.zeros()" function. The grayscale image is stored in the "greyimg" variable.

The function then iterates over each pixel of the input image using nested for loops, calculates the average of the pixel values (R, G, and B values), and assigns this average value to the corresponding pixel in the grayscale image.

Finally, the grayscale image is returned as the output of the function using the "return" statement.

Overall, this function converts an RGB image to grayscale by taking the average of the R, G, and

B values for each pixel and assigning this value to the corresponding pixel in the grayscale image

RGB COLOR MODEL :

The most well-known color model is RGB which stands for Red-Green-Blue. As the name suggests, this model represents colors using individual values for red, green, and blue. The RGB model is used in almost all digital screens throughout the world. Specifically, a color is defined using three integer values from 0 to 255 for red, green, and blue, where a zero value means dark and a value of 255 means bright. Given the values, the final color is defined when we mix these three basic colors weighted by their values.

If we mix the three colors equally (RGB = (255, 255, 255)), we'll get white while the absence of all colors (RGB = (0, 0, 0)) means black. Below there is the RGB coordinate system where we can see all the different colors that the model can describe:

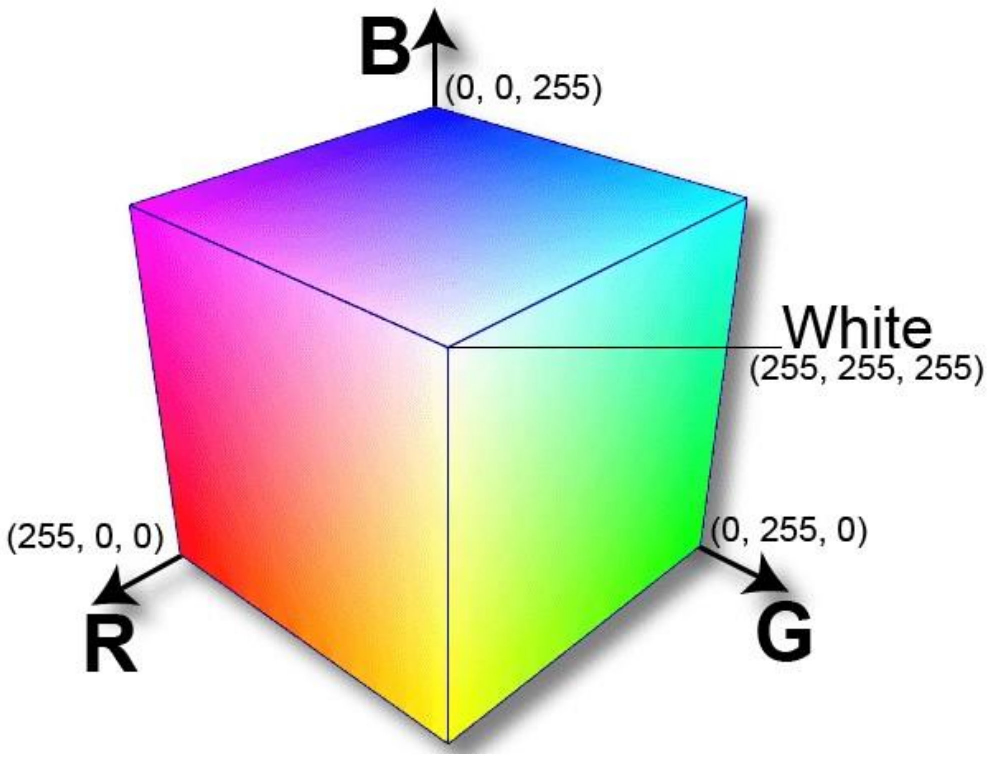


Fig.4.6.3 RGB COLOR MODEL

GREYSCALE COLOR :

Grayscale is the simplest model since it defines colors using only one component that is lightness. The amount of lightness is described using a value ranging from 0 (black) to 255 (white). On the one hand, grayscale images convey less information than RGB. However, they are common in image processing because using a grayscale image requires less available space and is faster, especially when we deal with complex computations.

Below, we can see the full range of colors that the grayscale model can describe:

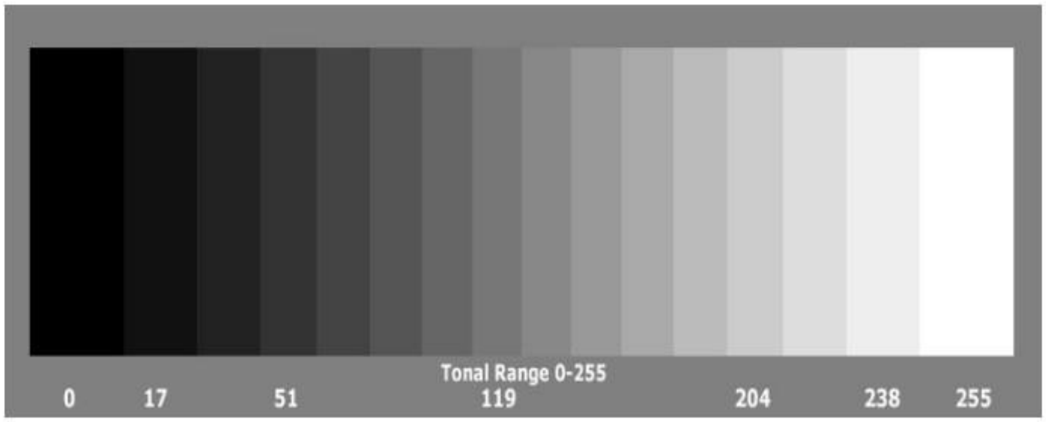


Fig.4.6.4 GRAYSCALE IMAGE

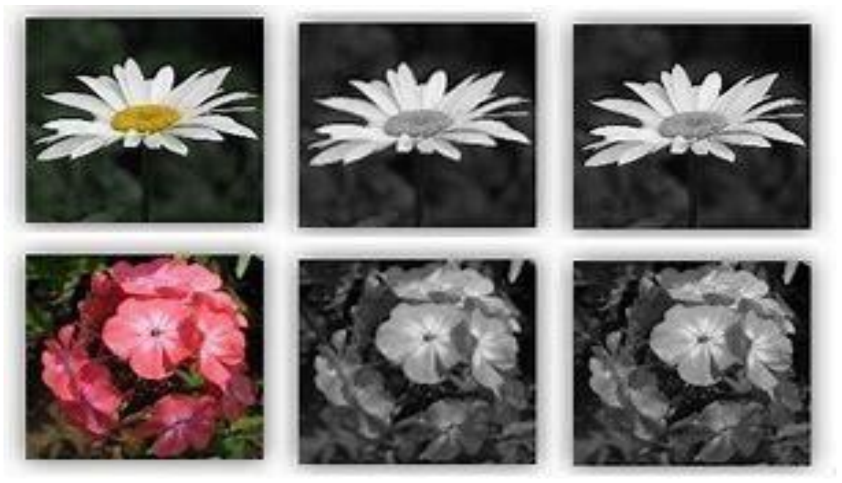


Fig.4.6.5 RGB TO GRAYSCALE

In terms of signature, rgbtograyscale images would be :

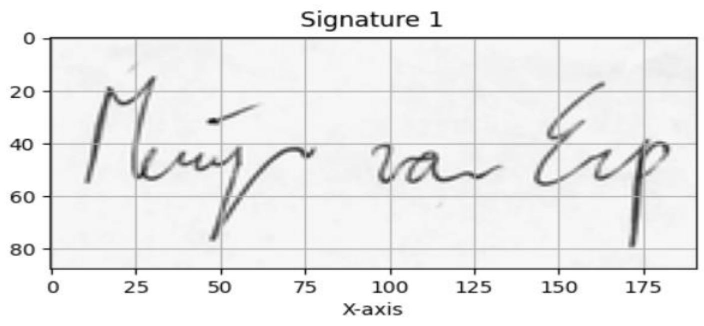
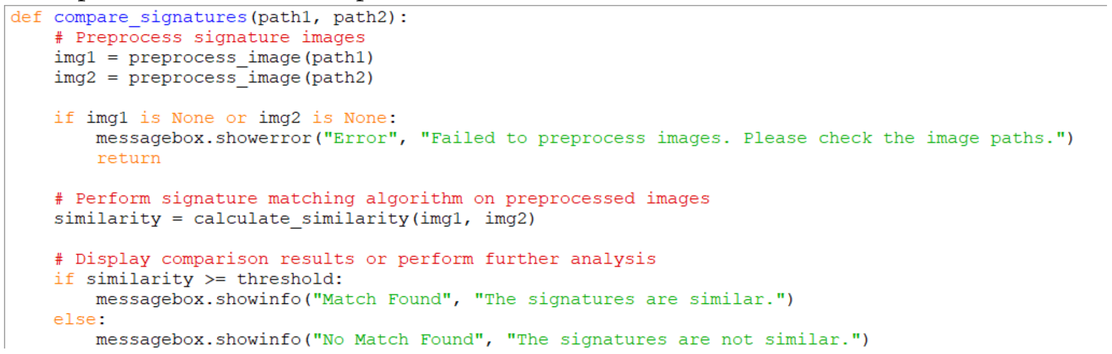


Fig.4.6.6 rgbtograyscale images

Integration with Signature Matching Algorithm :

The preprocessed signature images are seamlessly integrated into our signature matching algorithm for comparison and analysis. By preprocessing images effectively, we enhance the accuracy and reliability of signature matching, leading to more precise results and improved user satisfaction.



```
def compare_signatures(path1, path2):
    # Preprocess signature images
    img1 = preprocess_image(path1)
    img2 = preprocess_image(path2)

    if img1 is None or img2 is None:
        messagebox.showerror("Error", "Failed to preprocess images. Please check the image paths.")
        return

    # Perform signature matching algorithm on preprocessed images
    similarity = calculate_similarity(img1, img2)

    # Display comparison results or perform further analysis
    if similarity >= threshold:
        messagebox.showinfo("Match Found", "The signatures are similar.")
    else:
        messagebox.showinfo("No Match Found", "The signatures are not similar.")
```

Fig.4.6.7 Compare Signature Code

4.7 Feature Extraction

Feature extraction is a pivotal aspect of signature matching systems, serving as the cornerstone for transforming raw image data into meaningful representations that capture the distinguishing characteristics of signatures. In this section, we explore the intricacies of feature extraction methodologies implemented within our signature matching application.

Deep Learning-based Feature Extraction:

Deep learning has revolutionized the field of computer vision, offering unparalleled capabilities in automatically learning discriminative features from raw data.

Leveraging this paradigm, we employ pre-trained convolutional neural networks (CNNs), particularly the VGG16 architecture, for extracting hierarchical and semantically rich features from signature images.

The VGG16 model, pre-trained on large-scale image datasets, serves as a powerful feature extractor, discerning intricate patterns and structures within signature images. By leveraging transfer learning, we exploit the knowledge embedded within the VGG16 model's weights to extract abstract features that encapsulate the salient aspects of signatures.

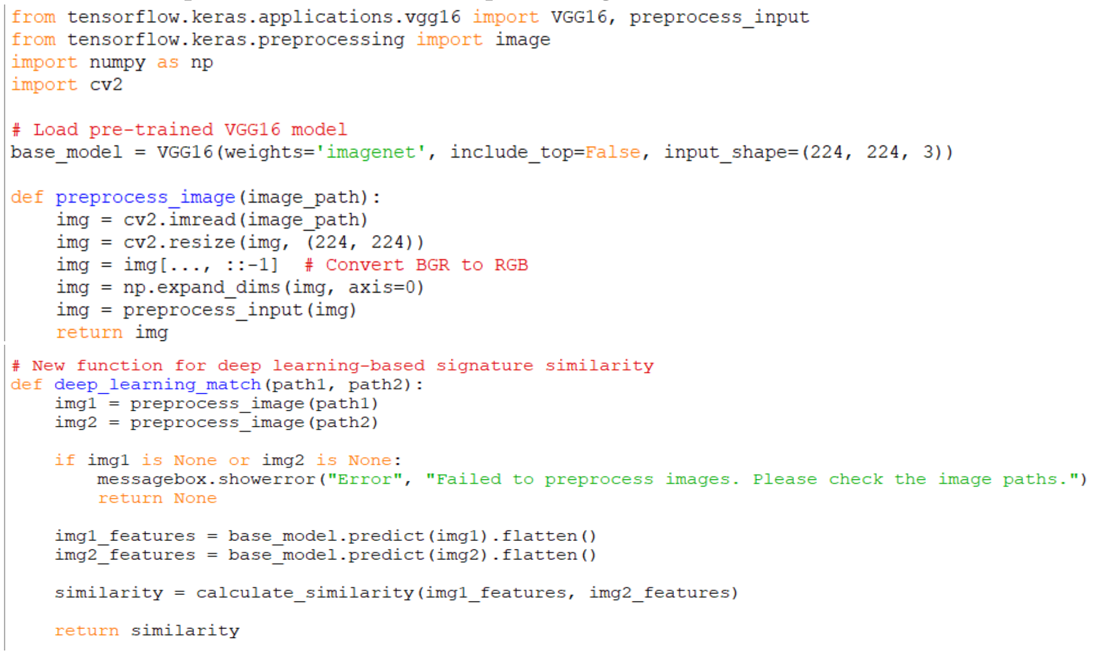
The process of extracting features using VGG16 is as follows:

1. Loading the Pre-trained Model: The VGG16 model is loaded with pre-trained weights, excluding the top fully connected layers. This setup allows us to use the convolutional base of VGG16 for feature extraction.

2. Image Preprocessing: Input images are resized to 224x224 pixels and preprocessed to match the input requirements of the VGG16 model. This preprocessing includes scaling pixel values and adjusting the image data format.

3. Feature Extraction: The preprocessed images are passed through the VGG16 model to obtain feature maps from the last convolutional layer. These feature maps are then flattened to form a one-dimensional feature vector representing the signature.

Here is the implementation of the deep learning-based feature extraction:



```
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
from tensorflow.keras.preprocessing import image
import numpy as np
import cv2

# Load pre-trained VGG16 model
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

def preprocess_image(image_path):
    img = cv2.imread(image_path)
    img = cv2.resize(img, (224, 224))
    img = img[..., ::-1]  # Convert BGR to RGB
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img)
    return img
# New function for deep learning-based signature similarity
def deep_learning_match(path1, path2):
    img1 = preprocess_image(path1)
    img2 = preprocess_image(path2)

    if img1 is None or img2 is None:
        messagebox.showerror("Error", "Failed to preprocess images. Please check the image paths.")
        return None

    img1_features = base_model.predict(img1).flatten()
    img2_features = base_model.predict(img2).flatten()

    similarity = calculate_similarity(img1_features, img2_features)

    return similarity
```

Fig.4.7.1 deep learning-based feature extraction code

The extracted features encapsulate various levels of abstraction, ranging from low-level edges and textures to high-level semantic concepts present in signature images. This hierarchical representation enables effective comparison and matching of signatures, facilitating robust performance across diverse datasets and signature styles.

Integration with Signature Matching Algorithm:

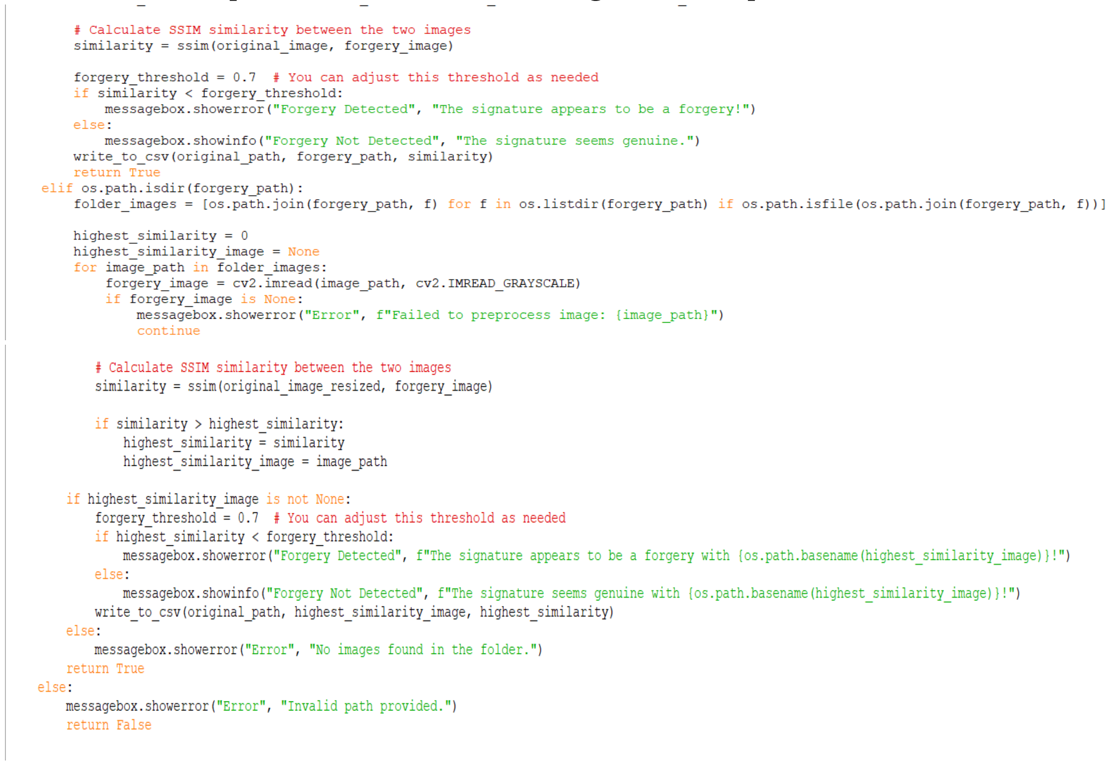
The extracted features serve as the foundation for our signature matching algorithm, enabling precise comparison and identification of similar signatures. By leveraging similarity metrics such as cosine similarity or Euclidean distance, we quantify the resemblance between feature vectors derived from signature images, facilitating accurate matching across diverse datasets and signature variations.
Here is the implementation of the signature comparison function:

Structural Similarity Index (SSIM):

To enhance the robustness of our signature matching system, we incorporate the Structural Similarity Index (SSIM) for image comparison. SSIM is a perceptual metric that assesses the visual similarity between two images based on structural information, luminance, and contrast. Unlike simple pixel-based methods, SSIM provides a more nuanced understanding of image quality and similarity.
The process of calculating SSIM for signature comparison involves the following steps:

1. Grayscale Conversion: Convert the signature images to grayscale to simplify the SSIM calculation and focus on structural similarity.
2. SSIM Calculation: Compute the SSIM index between the two grayscale signature images to quantify their similarity.

   Here is the implementation of SSIM for signature comparison:



```
        # Calculate SSIM similarity between the two images
        similarity = ssim(original_image, forgery_image)

        forgery_threshold = 0.7  # You can adjust this threshold as needed
        if similarity < forgery_threshold:
            messagebox.showerror("Forgery Detected", "The signature appears to be a forgery!")
        else:
            messagebox.showinfo("Forgery Not Detected", "The signature seems genuine.")
        write_to_csv(original_path, forgery_path, similarity)
        return True
    elif os.path.isdir(forgery_path):
        folder_images = [os.path.join(forgery_path, f) for f in os.listdir(forgery_path) if os.path.isfile(os.path.join(forgery_path, f))]

        highest_similarity = 0
        highest_similarity_image = None
        for image_path in folder_images:
            forgery_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
            if forgery_image is None:
                messagebox.showerror("Error", f"Failed to preprocess image: {image_path}")
                continue

            # Calculate SSIM similarity between the two images
            similarity = ssim(original_image_resized, forgery_image)

            if similarity > highest_similarity:
                highest_similarity = similarity
                highest_similarity_image = image_path

        if highest_similarity_image is not None:
            forgery_threshold = 0.7  # You can adjust this threshold as needed
            if highest_similarity < forgery_threshold:
                messagebox.showerror("Forgery Detected", f"The signature appears to be a forgery with {os.path.basename(highest_similarity_image)}!")
            else:
                messagebox.showinfo("Forgery Not Detected", f"The signature seems genuine with {os.path.basename(highest_similarity_image)}!")
            write_to_csv(original_path, highest_similarity_image, highest_similarity)
        else:
            messagebox.showerror("Error", "No images found in the folder.")
        return True
    else:
        messagebox.showerror("Error", "Invalid path provided.")
        return False
```

Fig.4.7.2 SSIM Calculation Code

By integrating SSIM into our signature matching framework, we enhance the robustness and accuracy of our system. SSIM complements the deep learning-based feature extraction by providing a perceptual similarity measure that captures structural nuances in signature images. This combined approach leverages the strengths of both methodologies, ensuring precise and reliable signature matching across diverse datasets and signature styles.

4.8 Object Detection for Signature Identification

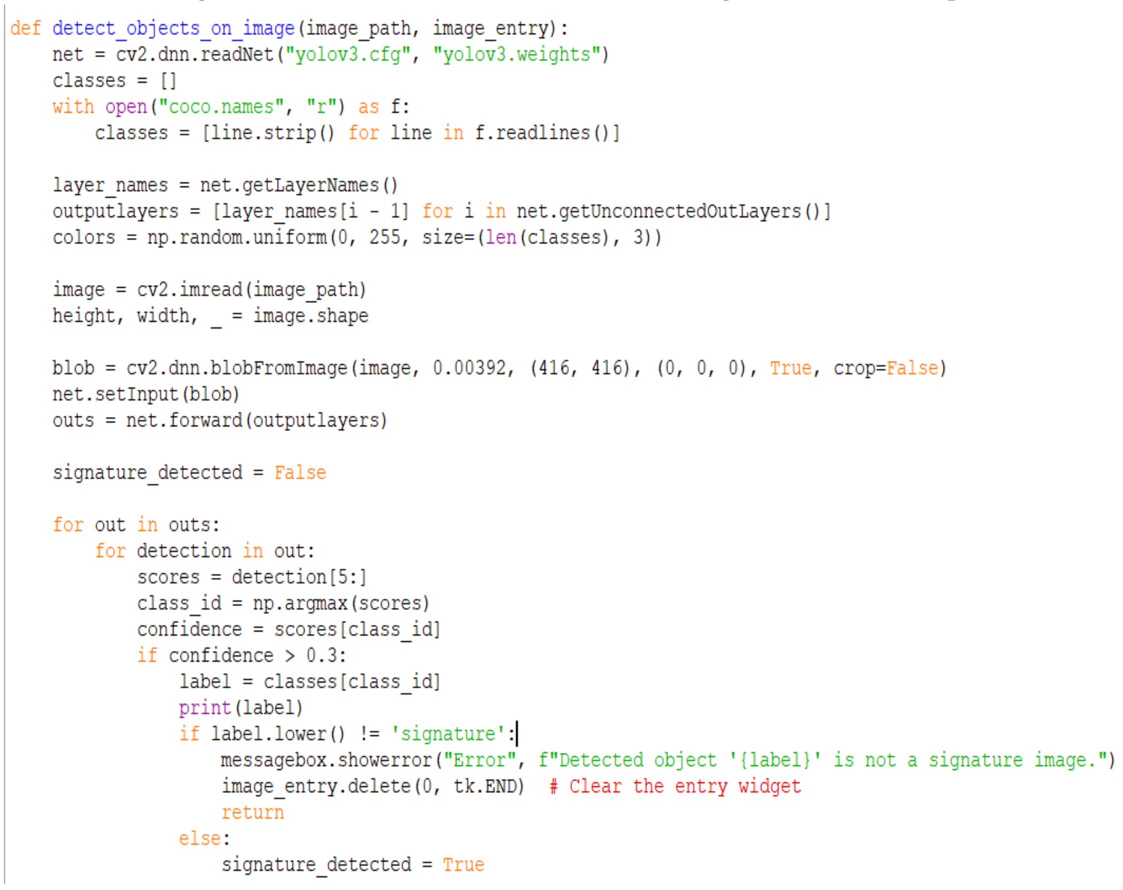
Object detection involves the localization and classification of objects within an image, typically achieved through the utilization of deep learning-based models such as YOLO (You Only Look Once) or Faster R-CNN (Region-based Convolutional Neural Network). These models are trained on annotated datasets to detect and classify objects across various categories.

YOLO-based Signature Detection

We leverage the YOLO (You Only Look Once) object detection framework to identify signature instances within input images. YOLO offers real-time object detection capabilities by dividing the input image into a grid of cells and predicting bounding boxes and class probabilities directly. This approach enables efficient and accurate detection of signatures without the need for region proposal mechanisms.

The process of signature identification using YOLO involves the following steps:

1. Model Loading: Load the pre-trained YOLO model along with its configuration and weights.
2. Class Definitions: Define the classes that the YOLO model can detect. In our case, we focus on the "signature" class for identifying signature regions.
3. Object Detection: Apply the YOLO model to the input image to detect signature instances. The model predicts bounding boxes and confidence scores for each detected object.
4. Post-processing: Filter out detected objects based on confidence scores and class probabilities. Retain only the bounding boxes corresponding to signature regions.
5. Here is a high-level overview of the YOLO-based signature detection process:



```python
def detect_objects_on_image(image_path, image_entry):
    net = cv2.dnn.readNet("yolov3.cfg", "yolov3.weights")
    classes = []
    with open("coco.names", "r") as f:
        classes = [line.strip() for line in f.readlines()]

    layer_names = net.getLayerNames()
    outputlayers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]
    colors = np.random.uniform(0, 255, size=(len(classes), 3))

    image = cv2.imread(image_path)
    height, width, _ = image.shape

    blob = cv2.dnn.blobFromImage(image, 0.00392, (416, 416), (0, 0, 0), True, crop=False)
    net.setInput(blob)
    outs = net.forward(outputlayers)

    signature_detected = False

    for out in outs:
        for detection in out:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.3:
                label = classes[class_id]
                print(label)
                if label.lower() != 'signature':
                    messagebox.showerror("Error", f"Detected object '{label}' is not a signature image.")
                    image_entry.delete(0, tk.END)  # Clear the entry widget
                    return
                else:
                    signature_detected = True
```

Fig.4.8.1 Object Detection code

The provided function detect_objects_on_image utilizes the YOLO (You Only Look Once) object detection framework to identify objects within an image. Let's break down the function's workflow:

1. Loading YOLO Model: The function reads the YOLO model architecture from the configuration file yolov3.cfg and loads pre-trained weights from yolov3.weights.
2. Defining Classes: It reads the class labels from the coco.names file, which contains a list of objects the YOLO model can detect.
3. Setting up YOLO Layers: YOLO consists of several layers, including input, output, and intermediate layers. This step initializes the output layers for object detection.
4. Processing Input Image: The function reads the input image using OpenCV's cv2.imread function and retrieves its dimensions (height and width).
5. Blob Preprocessing: Before feeding the image into the neural network, it preprocesses the image using cv2.dnn.blobFromImage to create a 4-dimensional blob.
6. Forward Pass: The preprocessed blob is passed through the YOLO network using net.setInput and net.forward to obtain the output detections.
7. Object Detection Loop: It iterates over the output detections (outs) to extract information about detected objects.
8. Filtering Signatures: For each detected object, it checks if the confidence score for the "signature" class exceeds a predefined threshold (in this case, 0.3). If the confidence is high enough, the object is considered a signature, and the function sets signature_detected to True.
9. Error Handling: If no signature is detected or if a non-signature object is detected, it displays an error message and clears the input entry widget.

By incorporating YOLO-based signature detection into our signature matching application, we enable automated identification and localization of signature regions within input images. This facilitates seamless integration with our existing feature extraction and matching pipelines, enhancing the overall efficiency and accuracy of the signature matching process.

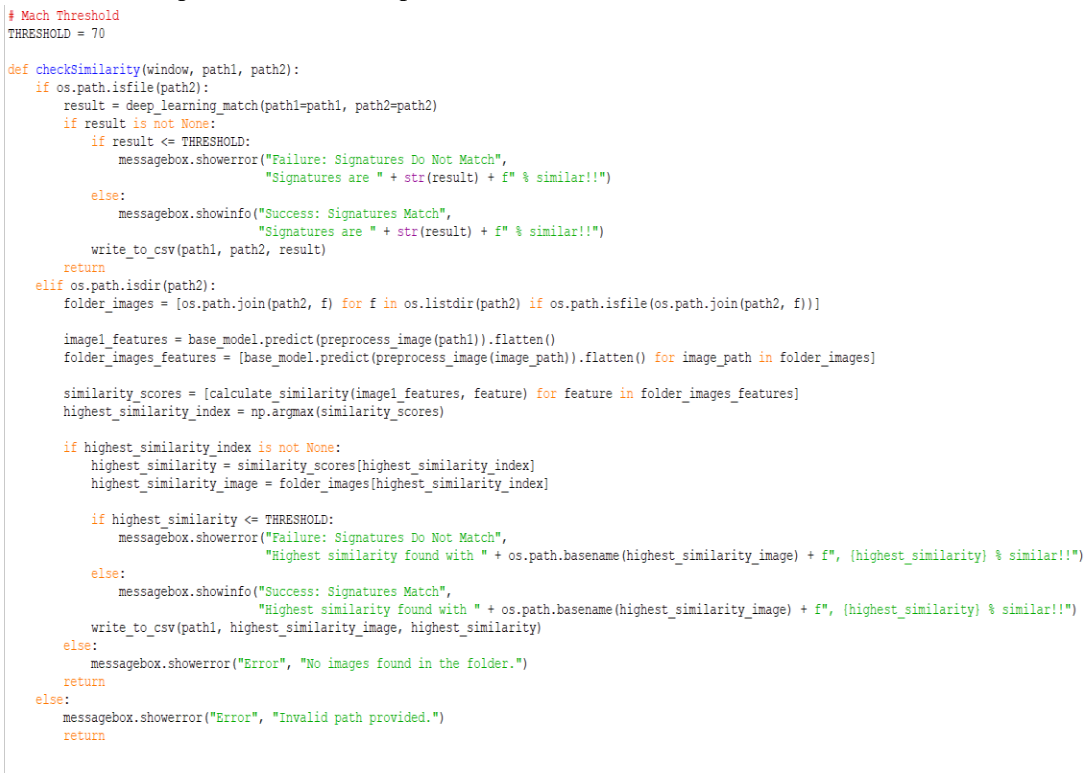
4.9 Signature Verification and Forgery Detection

Signature verification and forgery detection are critical components of our signature matching application, ensuring the integrity and authenticity of signatures in various contexts such as financial transactions, legal documents, and identity verification processes. In this section, we delve into the mechanisms employed for signature verification and forgery detection, elucidating the methodologies and algorithms utilized to assess the genuineness of signatures.

Signature Verification:

Signature verification entails the process of validating whether a given signature matches an authorized reference signature associated with a particular individual. Our signature verification module employs a combination of feature-based matching and similarity metrics to assess the likeness between two signatures. The verification process encompasses the following steps:

1. Feature Extraction: Extract salient features from the reference signature and the signature to be verified using deep learning-based and handcrafted feature extraction techniques.
2. Similarity Assessment: Calculate the similarity between the extracted features of the reference signature and the signature to be verified using appropriate similarity metrics such as cosine similarity or Euclidean distance.
3. Thresholding: Compare the similarity score obtained from the similarity assessment with a predetermined threshold. If the similarity score exceeds the threshold, the signatures are deemed to match, indicating successful verification.
4. Result Interpretation: Based on the outcome of the verification process, provide feedback indicating whether the signature matches the reference signature, thus facilitating decision-making in authentication scenarios.



```python
# Mach Threshold
THRESHOLD = 70

def checkSimilarity(window, path1, path2):
    if os.path.isfile(path2):
        result = deep_learning_match(path1=path1, path2=path2)
        if result is not None:
            if result <= THRESHOLD:
                messagebox.showerror("Failure: Signatures Do Not Match",
                                     "Signatures are " + str(result) + f" % similar!!")
            else:
                messagebox.showinfo("Success: Signatures Match",
                                    "Signatures are " + str(result) + f" % similar!!")
            write_to_csv(path1, path2, result)
        return
    elif os.path.isdir(path2):
        folder_images = [os.path.join(path2, f) for f in os.listdir(path2) if os.path.isfile(os.path.join(path2, f))]

        image1_features = base_model.predict(preprocess_image(path1)).flatten()
        folder_images_features = [base_model.predict(preprocess_image(image_path)).flatten() for image_path in folder_images]

        similarity_scores = [calculate_similarity(image1_features, feature) for feature in folder_images_features]
        highest_similarity_index = np.argmax(similarity_scores)

        if highest_similarity_index is not None:
            highest_similarity = similarity_scores[highest_similarity_index]
            highest_similarity_image = folder_images[highest_similarity_index]

            if highest_similarity <= THRESHOLD:
                messagebox.showerror("Failure: Signatures Do Not Match",
                                     "Highest similarity found with " + os.path.basename(highest_similarity_image) + f", {highest_similarity} % similar!!")
            else:
                messagebox.showinfo("Success: Signatures Match",
                                    "Highest similarity found with " + os.path.basename(highest_similarity_image) + f", {highest_similarity} % similar!!")
            write_to_csv(path1, highest_similarity_image, highest_similarity)
        else:
            messagebox.showerror("Error", "No images found in the folder.")
        return
    else:
        messagebox.showerror("Error", "Invalid path provided.")
        return
```

Fig.4.9.1 CheckSimilarity Code

The provided code includes two functions, checkSimilarity and checkForgery, which are essential for verifying signature authenticity and detecting forgery. Let's go through each function's role and functionality:

checkSimilarity(window, path1, path2)

This function is responsible for comparing signatures for similarity. It accepts two arguments:

- path1: The path to the original signature image.
- path2: The path to either a single forgery image or a folder containing multiple forgery images.

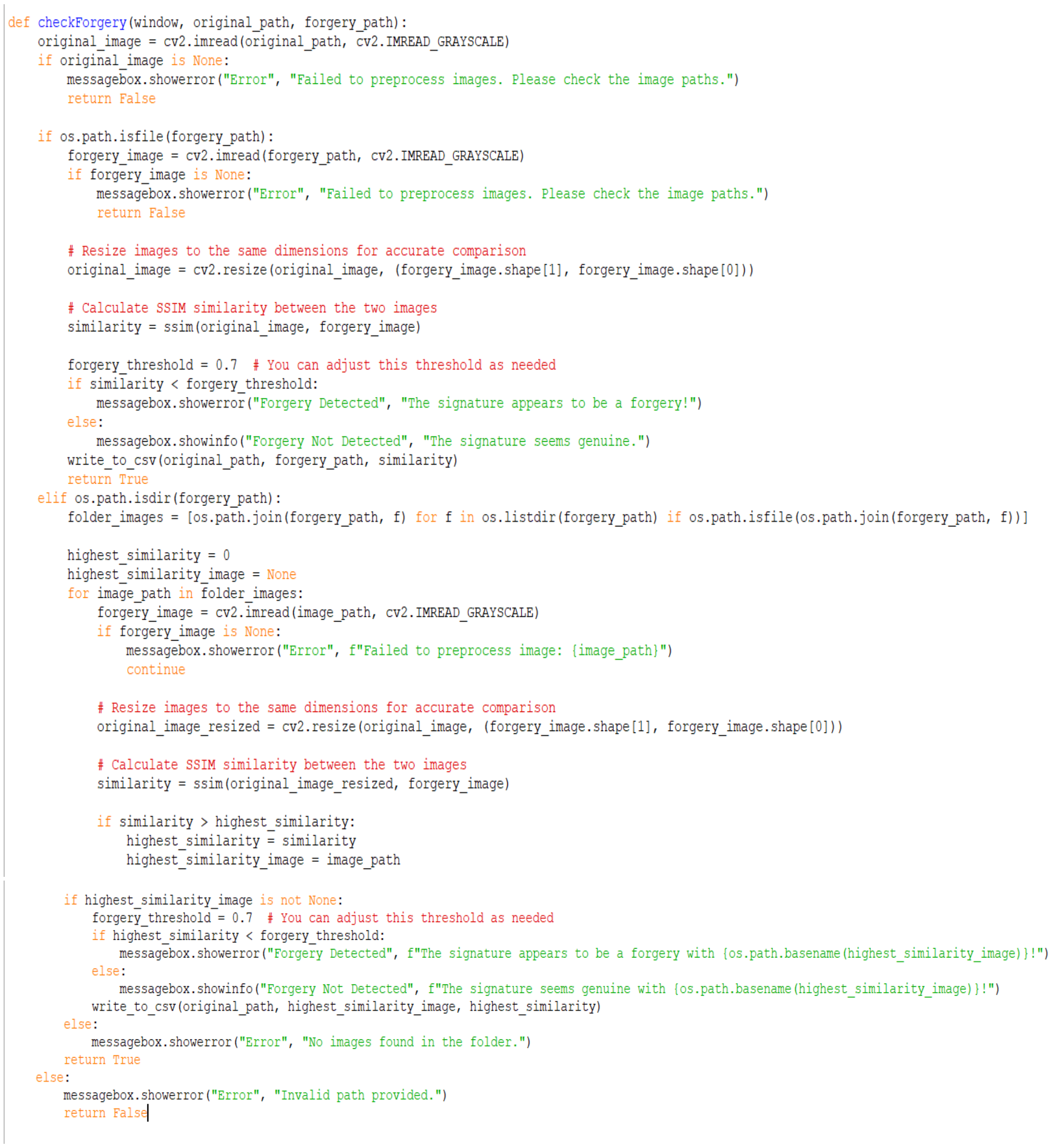
Workflow:

1. If path2 is a file, it uses a deep learning model (deep_learning_match) to compare the original signature with the forgery. If the similarity score exceeds a threshold (THRESHOLD), it displays a success message; otherwise, it indicates failure. The result is also written to a CSV file.
2. If path2 is a folder, it compares the original signature with all images in the folder. It calculates similarity scores using features extracted from a base model. The image with the highest similarity score is considered the most similar to the original signature. If the highest similarity score exceeds the threshold, it displays a success message; otherwise, it indicates failure. The result is written to a CSV file.

Forgery Detection:

Forgery detection involves the identification of counterfeit or fraudulent signatures intended to deceive or manipulate the verification process. Our forgery detection module leverages advanced image processing techniques and perceptual similarity metrics to discern genuine signatures from forged ones. The forgery detection process encompasses the following steps:

1. Image Preprocessing: Preprocess the reference signature and the signature under scrutiny to enhance image quality, remove noise, and standardize image characteristics.
2. Feature Extraction: Extract discriminative features from the preprocessed signatures using a combination of deep learning-based and handcrafted feature extraction methods, capturing unique patterns and characteristics.
3. Similarity Assessment: Calculate perceptual similarity metrics such as Structural Similarity Index (SSIM) or Pixel-wise Mean Squared Error (MSE) to quantify the likeness between the reference signature and the signature under examination.
4. Forgery Classification: Apply machine learning or statistical classification algorithms to differentiate between genuine and forged signatures based on the computed similarity scores and other relevant features.
5. Result Interpretation: Present the outcome of the forgery detection process, indicating whether the signature exhibits characteristics indicative of forgery or genuine origin, empowering decision-makers to take appropriate actions.



```python
def checkForgery(window, original_path, forgery_path):
    original_image = cv2.imread(original_path, cv2.IMREAD_GRAYSCALE)
    if original_image is None:
        messagebox.showerror("Error", "Failed to preprocess images. Please check the image paths.")
        return False

    if os.path.isfile(forgery_path):
        forgery_image = cv2.imread(forgery_path, cv2.IMREAD_GRAYSCALE)
        if forgery_image is None:
            messagebox.showerror("Error", "Failed to preprocess images. Please check the image paths.")
            return False

        # Resize images to the same dimensions for accurate comparison
        original_image = cv2.resize(original_image, (forgery_image.shape[1], forgery_image.shape[0]))

        # Calculate SSIM similarity between the two images
        similarity = ssim(original_image, forgery_image)

        forgery_threshold = 0.7  # You can adjust this threshold as needed
        if similarity < forgery_threshold:
            messagebox.showerror("Forgery Detected", "The signature appears to be a forgery!")
        else:
            messagebox.showinfo("Forgery Not Detected", "The signature seems genuine.")
        write_to_csv(original_path, forgery_path, similarity)
        return True
    elif os.path.isdir(forgery_path):
        folder_images = [os.path.join(forgery_path, f) for f in os.listdir(forgery_path) if os.path.isfile(os.path.join(forgery_path, f))]

        highest_similarity = 0
        highest_similarity_image = None
        for image_path in folder_images:
            forgery_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
            if forgery_image is None:
                messagebox.showerror("Error", f"Failed to preprocess image: {image_path}")
                continue

            # Resize images to the same dimensions for accurate comparison
            original_image_resized = cv2.resize(original_image, (forgery_image.shape[1], forgery_image.shape[0]))

            # Calculate SSIM similarity between the two images
            similarity = ssim(original_image_resized, forgery_image)

            if similarity > highest_similarity:
                highest_similarity = similarity
                highest_similarity_image = image_path

        if highest_similarity_image is not None:
            forgery_threshold = 0.7  # You can adjust this threshold as needed
            if highest_similarity < forgery_threshold:
                messagebox.showerror("Forgery Detected", f"The signature appears to be a forgery with {os.path.basename(highest_similarity_image)}!")
            else:
                messagebox.showinfo("Forgery Not Detected", f"The signature seems genuine with {os.path.basename(highest_similarity_image)}!")
            write_to_csv(original_path, highest_similarity_image, highest_similarity)
        else:
            messagebox.showerror("Error", "No images found in the folder.")
        return True
    else:
        messagebox.showerror("Error", "Invalid path provided.")
        return False
```

Fig.4.9.2 CheckForgery Code

checkForgery(window, original_path, forgery_path)

This function detects forgery in signature images. It accepts two arguments:

- original_path: The path to the original signature image.
- forgery_path: The path to either a single forgery image or a folder containing multiple forgery images.

Workflow:

1. If forgery_path is a file, it reads the original and forgery images, resizes them to the same dimensions, and calculates their Structural Similarity Index (SSIM). If the SSIM is below a forgery threshold, it indicates forgery; otherwise, it indicates the signature is genuine. The result is written to a CSV file.

2. If forgery_path is a folder, it iterates through all images in the folder, compares each forgery image with the original using SSIM, and identifies the most similar image. If any
3. image surpasses the forgery threshold, it indicates the signature is genuine; otherwise, it detects forgery. The result is written to a CSV file.

These functions provide robust mechanisms for verifying the authenticity of signature images and detecting forgery, crucial for signature matching applications.

4.10 Getting CSV Features :

In order to facilitate the analysis, training, and evaluation of our signature matching model, we extract relevant features from the signature images and store them in a structured format within a CSV file. This process ensures that the extracted features are readily available for various machine learning tasks, such as classification, clustering, and similarity measurement. The steps involved in obtaining CSV features are described below:

Feature Extraction Process

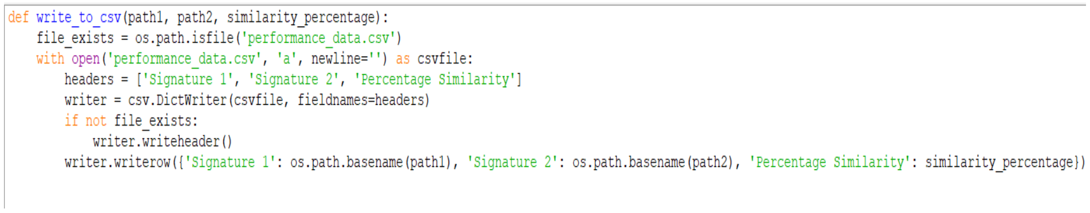
The feature extraction process is carried out using both deep learning-based and handcrafted methods. The features encapsulate the essential characteristics of the signatures, such as shapes, textures, and structural information. Here's a quick recap of the methods used:

1. Deep Learning-based Feature Extraction: Utilizing pre-trained convolutional neural networks (CNNs) like VGG16 to extract high-level, semantically rich features from signature images.
2. Structural Similarity Index (SSIM): Computing SSIM between pairs of images to capture perceptual similarity based on structural information, luminance, and contrast.

Preparing Features for CSV Storage

After extracting the features, we need to format them appropriately for storage in a CSV file. The CSV file will contain rows corresponding to individual signature images and columns representing the extracted features. Additional metadata, such as file names or labels, can also be included to facilitate data management and analysis.

In addition to storing extracted features, it is essential to log the similarity scores between pairs of signature images. This information is crucial for performance evaluation and further analysis. The following function logs the similarity percentage between two signature images into a CSV file:



```
def write_to_csv(path1, path2, similarity_percentage):
    file_exists = os.path.isfile('performance_data.csv')
    with open('performance_data.csv', 'a', newline='') as csvfile:
        headers = ['Signature 1', 'Signature 2', 'Percentage Similarity']
        writer = csv.DictWriter(csvfile, fieldnames=headers)
        if not file_exists:
            writer.writeheader()
        writer.writerow({'Signature 1': os.path.basename(path1), 'Signature 2': os.path.basename(path2), 'Percentage Similarity': similarity_percentage})
```

Fig.4.10.1 Getting CSV Features code

Explanation of the write_to_csv Function

The write_to_csv function is designed to append similarity data between two signature images to a CSV file. This function ensures that the data is organized in a tabular format, which can be useful for subsequent analysis, reporting, or model evaluation. Let's break down the code step by step to understand its functionality in detail.

def write_to_csv(path1, path2, similarity_percentage):

Purpose: This function appends a row of data containing the file names of two signature images and their similarity percentage to a CSV file named performance_data.csv.

Parameters:

path1: The file path of the first signature image.

path2: The file path of the second signature image.

similarity_percentage: A numerical value representing the similarity percentage between the two images.

Checking if the CSV File Exists

file_exists = os.path.isfile('performance_data.csv')

Purpose: This line checks whether the file performance_data.csv already exists in the current directory.

Function: os.path.isfile returns True if the file exists and False otherwise. This information is used to determine whether the CSV file needs to be created and initialized with headers.

with open('performance_data.csv', 'a', newline='') as csvfile:

Purpose: This line opens the performance_data.csv file in append mode ('a').

Parameters:

'a': This mode opens the file for appending. Data will be added to the end of the file without modifying its existing content.

newline='': This argument ensures that new lines are handled correctly across different operating systems.

Context Manager: The with statement ensures that the file is properly closed after the block of code is executed, even if an error occurs.

Defining the CSV Headers

headers = ['Signature 1', 'Signature 2', 'Percentage Similarity']

Purpose: This line defines the column headers for the CSV file. These headers describe the type of data that will be stored in each column.

Initializing the CSV Writer

```
writer = csv.DictWriter(csvfile, fieldnames=headers)
```

Purpose: This line initializes a DictWriter object from the csv module.
Parameters:
csvfile: The file object to which the CSV data will be written.
fieldnames=headers: Specifies the column headers for the CSV file.
Writing the Header Row (If Necessary)

```
if not file_exists:
    writer.writeheader()
```

Purpose: This conditional block checks if the CSV file did not exist before opening.
Function: If the file did not exist (file_exists is False), the writeheader method is called to write the headers to the CSV file. This ensures that headers are only written once, at the beginning of the file.

Writing the Data Row

```
writer.writerow({'Signature 1': os.path.basename(path1), 'Signature 2': os.path.basename(path2), 'Percentage Similarity': similarity_percentage})
```

Purpose: This line writes a row of data to the CSV file.
Function:
os.path.basename(path1): Extracts the base name (file name) from the full path path1. For example, if path1 is /path/to/signature1.png, os.path.basename(path1) would return signature1.png.
os.path.basename(path2): Similarly, extracts the base name from path2.
similarity_percentage: The provided similarity percentage between the two signatures.
writer.writerow: This method writes a dictionary where keys are the column headers and values are the corresponding data points.

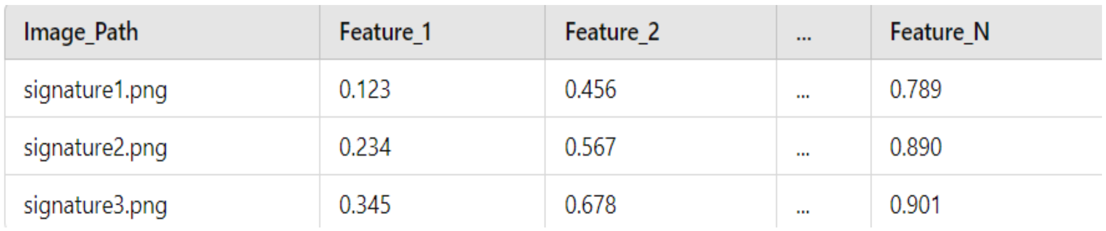
The write_to_csv function is a robust way to log similarity percentages between pairs of signature images into a CSV file. It handles file creation, appending data, and ensures that headers are only written once. Here’s what it accomplishes:
Checks if the CSV file already exists: This is necessary to determine whether to write the headers.
Opens the CSV file in append mode: Ensures that existing data is not overwritten.
Defines the CSV headers: These describe the structure of the data.
Initializes a DictWriter: Facilitates writing dictionary data to the CSV.
Writes headers if the file is new: Ensures headers are only written once.

Writes a row of data: Logs the base names of the signature images and their similarity percentage.

Structure of the CSV Files:

Features CSV File

The features CSV file contains the extracted features in a structured format, enabling easy access and manipulation for various machine learning tasks. The structure of the CSV file is as follows:

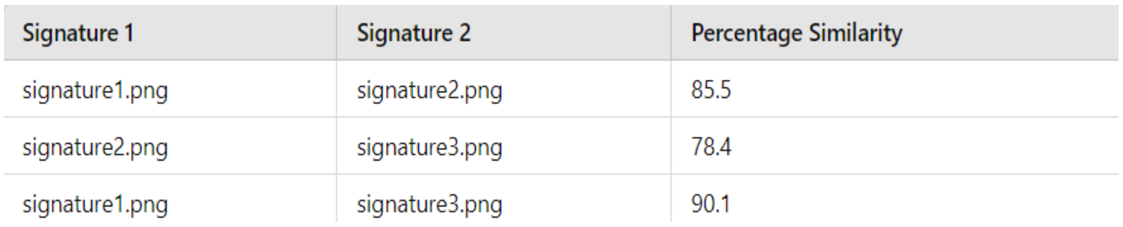


Image_Path	Feature_1	Feature_2	...	Feature_N
signature1.png	0.123	0.456	...	0.789
signature2.png	0.234	0.567	...	0.890
signature3.png	0.345	0.678	...	0.901

Fig.4.10.2 Features CSV File

Performance Data CSV File

The performance data CSV file logs the similarity scores between pairs of signature images, which is crucial for evaluating the model's performance. The structure of this CSV file is as follows:



Signature 1	Signature 2	Percentage Similarity
signature1.png	signature2.png	85.5
signature2.png	signature3.png	78.4
signature1.png	signature3.png	90.1

Fig.4.10.3 Performance Data CSV File

These CSV files serve as comprehensive datasets, enabling efficient training, validation, and testing of machine learning models for signature matching. By leveraging the extracted features and similarity scores, we can build robust and accurate models to address various signature verification and identification tasks.

4.11 Error Analysis

Error analysis is a crucial aspect of developing a robust application. It helps in identifying potential issues, understanding their causes, and implementing strategies to handle them effectively. In our signature matching application, we focus on handling various types of errors, including file handling errors, user input errors, and system errors.

Types of Errors Handled

1. File Handling Errors:

- Invalid File Path: If the provided file path for signature images is invalid or the file does not exist, an error message is displayed to the user.

2. User Input Errors:

- Empty Fields: If the user attempts to perform actions without entering the required file paths or similarity percentage, the application prompts the user to fill in the necessary fields.

3. System Errors:

- OpenCV Display Errors: Errors related to displaying images using OpenCV are handled to ensure the application does not crash and provides feedback to the user.

Implementation of Error Handling

The following code snippets illustrate how various errors are handled in the application:

```
        messagebox.showerror("Error", f"Failed to open image: {e}")
        return None
```

```
    messagebox.showerror("Error", "Failed to preprocess images. Please check the image paths.")
    return None
```

```
            messagebox.showerror("Error", f"Detected object '{label}' is not a signature image.")
```

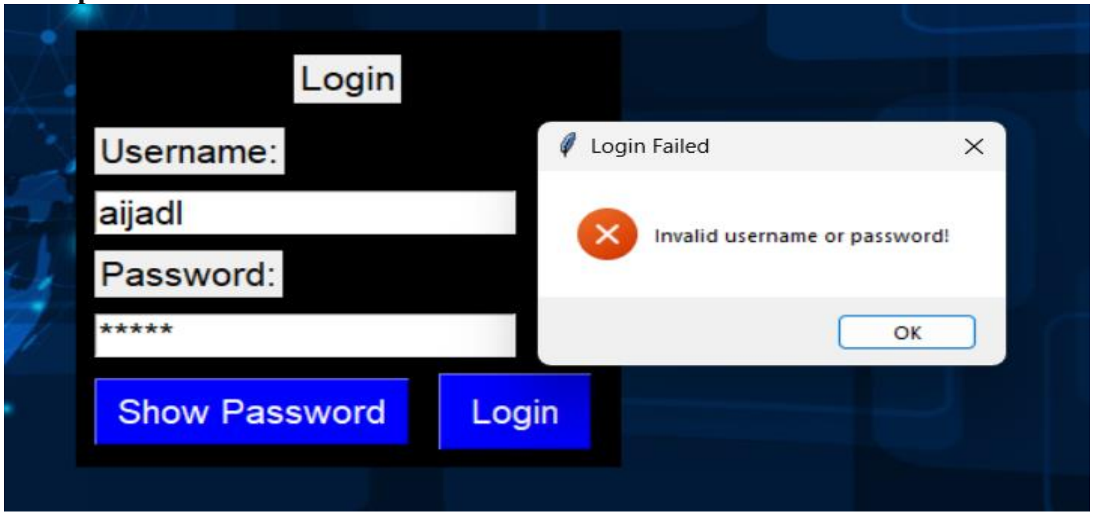
```
            messagebox.showerror("Error", "No images found in the folder.")
    else:
        messagebox.showerror("Error", "Failed to load one or both images.")
```

```
    else:
        messagebox.showerror("Login Failed", "Invalid username or password!")
```

User Feedback

Providing clear and informative feedback to the user is essential for a positive user experience. The application uses message boxes to display error messages, ensuring that users are aware of any issues and can take corrective actions.

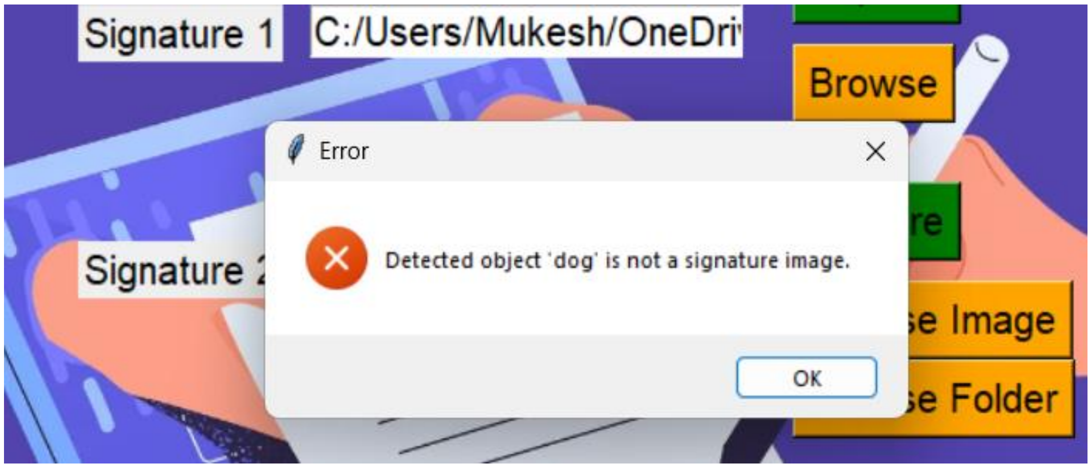
Example of UserFeedback:



Login Failed

Invalid username or password!

Fig.4.11.1 Invalid User Name and Password


Signature 1

C:/Users/Mukesh/OneDri

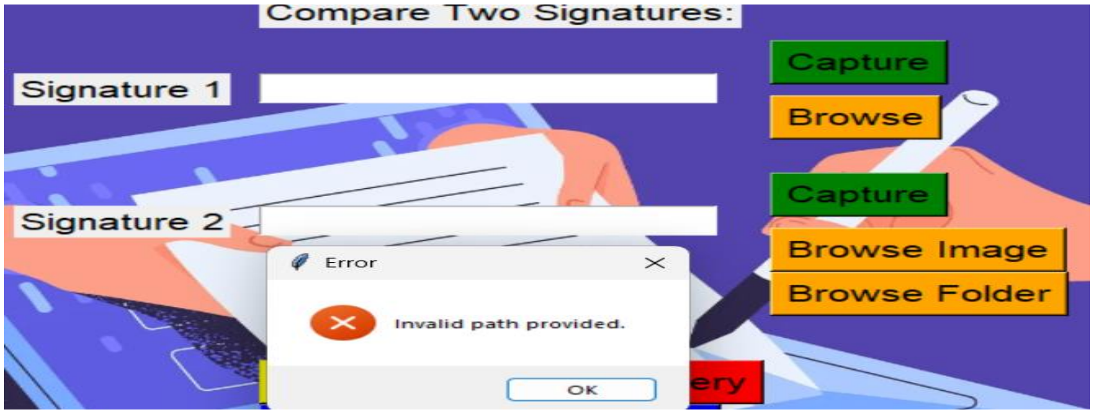
Browse

Error

Detected object 'dog' is not a signature image.

OK

Fig.4.11.2 object detected


Compare Two Signatures:

Signature 1

Signature 2

Capture

Browse

Capture

Browse Image

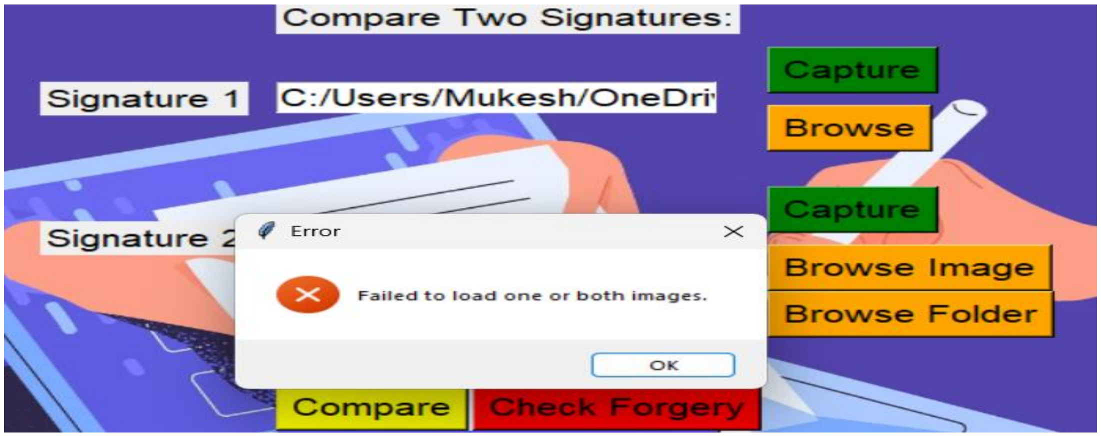
Browse Folder

Error

Invalid path provided.

OK

Fig.4.11.3 Invalid path


Compare Two Signatures:

Signature 1

C:/Users/Mukesh/OneDri

Capture

Browse

Capture

Browse Image

Browse Folder

Error

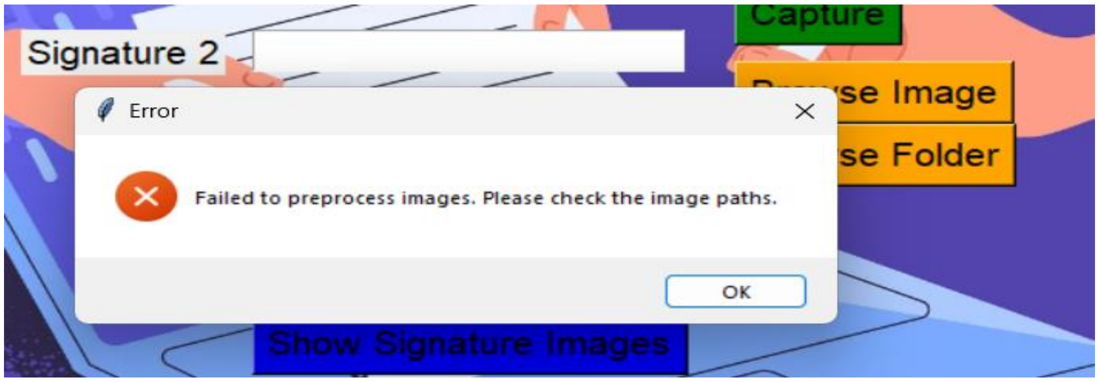
Failed to load one or both images.

OK

Compare

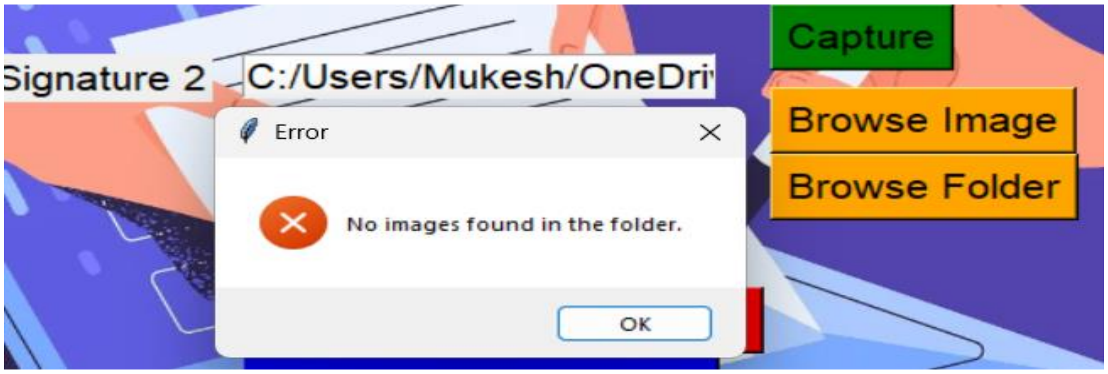
Check Forgery

Fig.4.11.4 Failed to load image


Error

Failed to preprocess images. Please check the image paths.

Fig.4.11.5 Failed to preprocess image


Error

No images found in the folder.

Fig.4.11.6 No Image found

Analysis of Error Occurrences

During the testing phase, various error scenarios were simulated to ensure that the error handling mechanisms are effective. The following observations were made:

File Handling Errors:

Occurred frequently during initial testing due to incorrect file paths provided by users.

Solution: Implemented file path validation and clear error messages.

User Input Errors:

Common when users forgot to fill in all required fields.

Solution: Added checks to ensure all fields are completed before proceeding.

System Errors:

Rare, but occurred during unexpected issues with OpenCV.

Solution: General exception handling to catch and display unexpected errors.

By implementing comprehensive error handling strategies, our application ensures robustness and enhances user experience by providing clear, actionable feedback

4.12 Characteristics & Pseudocode

```
import tkinter as tk   # For creating the graphical user
interface (GUI)
from tkinter.filedialog import askopenfilename # For file
selection dialog
from tkinter import messagebox  # For displaying message boxes
import os  # For operating system-related operations
import cv2 # For image processing using OpenCV
from PIL import Image, ImageTk  # For working with images
from skimage.metrics import structural_similarity as ssim  #
For SSIM-based image comparison
import csv # For handling CSV file operations
from tensorflow.keras.applications.vgg16 import VGG16,
preprocess_input # For VGG16 model
from tensorflow.keras.models import Model # For building deep
learning models
import numpy as np # For numerical operations and arrays
import matplotlib.pyplot as plt # Import the Matplotlib library
for plotting and visualization
import subprocess # Import the subprocess module for executing
external commands
from tkinter.filedialog import askopenfilename, askdirectory #
Import the askopenfilename and askdirectory functions from the
tkinter.filedialog module


def write_to_csv(path1, path2, similarity_percentage):
    file_exists = os.path.isfile('performance_data.csv')
    with open('performance_data.csv', 'a', newline='') as
csvfile:
        headers = ['Signature 1', 'Signature 2', 'Percentage
Similarity']
        writer = csv.DictWriter(csvfile, fieldnames=headers)
        if not file_exists:
            writer.writeheader()
        writer.writerow({'Signature 1':
os.path.basename(path1), 'Signature 2':
os.path.basename(path2), 'Percentage Similarity':
similarity_percentage})

# Load pre-trained VGG16 model (without the top classification
layer)
```

```
base_model                                              =
VGG16(weights='vgg16_weights_tf_dim_ordering_tf_kernels_notop
.h5', include_top=False, input_shape=(224, 224, 3))

def preprocess_image(image_path):
    try:
        image = Image.open(image_path)
        image = image.resize((224, 224))
        image = np.array(image)
        if len(image.shape) == 2:  # If the image is grayscale,
convert it to RGB
            image = np.stack((image,) * 3, axis=-1)
        image = preprocess_input(image)
        image = np.expand_dims(image, axis=0)
        return image
    except Exception as e:
        messagebox.showerror("Error", f"Failed to open image:
{e}")
        return None

# Add this function to calculate the similarity between two
feature vectors
def calculate_similarity(features1, features2):
    #  Perform  similarity  calculation  based  on  your
requirements
    # For example, you can use cosine similarity:
    similarity   =   np.dot(features1,   features2)   /
(np.linalg.norm(features1) * np.linalg.norm(features2))
    similarity_percentage = round(similarity * 100, 2)
    return similarity_percentage


# New function for deep learning-based signature similarity
def deep_learning_match(path1, path2):
    img1 = preprocess_image(path1)
    img2 = preprocess_image(path2)

    if img1 is None or img2 is None:
        messagebox.showerror("Error",  "Failed  to  preprocess
images. Please check the image paths.")
        return None
    img1_features = base_model.predict(img1).flatten()
    img2_features = base_model.predict(img2).flatten()

    similarity      =      calculate_similarity(img1_features,
img2_features)

    return similarity

def detect_objects_on_image(image_path, image_entry):
    net = cv2.dnn.readNet("yolov3.cfg", "yolov3.weights")
    classes = []
    with open("coco.names", "r") as f:
        classes = [line.strip() for line in f.readlines()]

    layer_names = net.getLayerNames()
    outputlayers   =   [layer_names[i   -   1]   for   i   in
net.getUnconnectedOutLayers()]
    colors = np.random.uniform(0, 255, size=(len(classes), 3))

    image = cv2.imread(image_path)
    height, width, _ = image.shape

    blob = cv2.dnn.blobFromImage(image, 0.00392, (416, 416),
(0, 0, 0), True, crop=False)
    net.setInput(blob)
    outs = net.forward(outputlayers)

    signature_detected = False

    for out in outs:
        for detection in out:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.3:
                label = classes[class_id]
                print(label)
                if label.lower() != 'signature':
                    messagebox.showerror("Error",  f"Detected
object '{label}' is not a signature image.")
```

```
                    image_entry.delete(0, tk.END)  # Clear the
entry widget
                    return
                else:
                    signature_detected = True


def capture_image_from_cam_into_temp(sign=1):
    cam = cv2.VideoCapture(0)

    cv2.namedWindow("test")

    while True:
        ret, frame = cam.read()
        if not ret:
            print("Failed to grab frame")
            break
        cv2.imshow("test", frame)

        k = cv2.waitKey(1)
        if k == 27:  # Press ESC to exit
            print("Escape hit, closing...")
            break
        elif k == 32:  # Press SPACE to capture
            if not os.path.isdir('temp'):
                os.mkdir('temp', mode=0o777)  # make sure the
directory exists
            if sign == 1:
                img_name = "./temp/test_img1.png"
            else:
                img_name = "./temp/test_img2.png"
            print('imwrite=',  cv2.imwrite(filename=img_name,
img=frame))
            print("{} written!".format(img_name))
            break  # Exit after capturing a single image

    cam.release()
    cv2.destroyAllWindows()
    return True

def captureImage(ent, sign=1):
    if(sign == 1):
        filename = os.getcwd()+'\\temp\\test_img1.png'
    else:
        filename = os.getcwd()+'\\temp\\test_img2.png'
    # messagebox.showinfo(
    #     'SUCCESS!!!', 'Press Space Bar to click picture and
ESC to exit')
    res = None
    res = messagebox.askquestion(
        'Click Picture', 'Press Space Bar to click picture and
ESC to exit')
    if res == 'yes':
        capture_image_from_cam_into_temp(sign=sign)
        ent.delete(0, tk.END)
        ent.insert(tk.END, filename)
    return True



def browsefunc(ent):
    file_path  =  askopenfilename(filetypes=[("Image  files",
"*.png;*.jpg;*.jpeg"), ("All files", "*.*")])
    if file_path:
        ent.delete(0, tk.END)
        ent.insert(tk.END, file_path)
        # Call detect_objects_on_image with the entry widget
as an argument
        detect_objects_on_image(file_path, ent)

def browse_folder(ent):
    folder_path = askdirectory()
    if folder_path:
        ent.delete(0, tk.END)
        ent.insert(tk.END, folder_path)


def show_both_signature_images(path1, path2):
    if os.path.isfile(path2):
        show_signature_comparison(path1, path2)
    elif os.path.isdir(path2):
        folder_images  =  [os.path.join(path2,  f)  for  f  in
os.listdir(path2) if os.path.isfile(os.path.join(path2, f))]
        highest_similarity = 0
        highest_similarity_image = None
        for image_path in folder_images:
            similarity   =   deep_learning_match(path1=path1,
path2=image_path)
            if  similarity  is  not  None  and  similarity  >
highest_similarity:
                highest_similarity = similarity
                highest_similarity_image = image_path

        if highest_similarity_image is not None:
            show_signature_comparison(path1,
highest_similarity_image)
        else:
            messagebox.showerror("Error", "No images found in
the folder.")
    else:
        messagebox.showerror("Error", "Failed to load one or
both images.")

def show_signature_comparison(path1, path2):
    img1 = cv2.imread(path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(path2, cv2.IMREAD_GRAYSCALE)

    if img1 is None or img2 is None:
        messagebox.showerror("Error", "Failed to load one or
both images.")
        return

    plt.figure(figsize=(10, 5))

    ax1 = plt.subplot(1, 2, 1)
    ax1.imshow(img1, cmap='gray')  # Use 'gray' colormap for
grayscale
    ax1.set_title("Signature 1")
    ax1.set_xlabel("X-axis")
    ax1.set_ylabel("Y-axis")
    ax1.grid(True)  # Display gridlines

    ax2 = plt.subplot(1, 2, 2)
    ax2.imshow(img2, cmap='gray')  # Use 'gray' colormap for
grayscale
```

```
    ax2.set_title("Signature 2")
    ax2.set_xlabel("X-axis")
    ax2.set_ylabel("Y-axis")
    ax2.grid(True)  # Display gridlines

    plt.tight_layout()
    plt.show()

# Mach Threshold
THRESHOLD = 70

def checkSimilarity(window, path1, path2):
    if os.path.isfile(path2):
        result      =      deep_learning_match(path1=path1,
path2=path2)
        if result is not None:
            if result <= THRESHOLD:
                messagebox.showerror("Failure:  Signatures  Do
Not Match",
                                     "Signatures   are   "   +
str(result) + f" % similar!!")
            else:
                messagebox.showinfo("Success:      Signatures
Match",
                                    "Signatures   are   "   +
str(result) + f" % similar!!")
            write_to_csv(path1, path2, result)
        return
    elif os.path.isdir(path2):
        folder_images = [os.path.join(path2,  f)  for  f  in
os.listdir(path2) if os.path.isfile(os.path.join(path2, f))]

        image1_features                                      =
base_model.predict(preprocess_image(path1)).flatten()
        folder_images_features                               =
[base_model.predict(preprocess_image(image_path)).flatten()
for image_path in folder_images]

        similarity_scores                                    =
[calculate_similarity(image1_features, feature) for feature in
folder_images_features]
```

```
        highest_similarity_index                           =
np.argmax(similarity_scores)

        if highest_similarity_index is not None:
            highest_similarity                             =
similarity_scores[highest_similarity_index]
            highest_similarity_image                       =
folder_images[highest_similarity_index]

            if highest_similarity <= THRESHOLD:
                messagebox.showerror("Failure:  Signatures  Do
Not Match",
                                     "Highest      similarity
found  with  "  +  os.path.basename(highest_similarity_image)  +
f", {highest_similarity} % similar!!")
            else:
                messagebox.showinfo("Success:      Signatures
Match",
                                    "Highest similarity found
with  "  +  os.path.basename(highest_similarity_image)  +  f",
{highest_similarity} % similar!!")
            write_to_csv(path1,     highest_similarity_image,
highest_similarity)
        else:
            messagebox.showerror("Error", "No images found in
the folder.")
        return
    else:
        messagebox.showerror("Error",      "Invalid      path
provided.")
        return




def checkForgery(window, original_path, forgery_path):
    original_image        =        cv2.imread(original_path,
cv2.IMREAD_GRAYSCALE)
    if original_image is None:
        messagebox.showerror("Error",  "Failed  to  preprocess
images. Please check the image paths.")
        return False
    if os.path.isfile(forgery_path):
        forgery_image       =       cv2.imread(forgery_path,
cv2.IMREAD_GRAYSCALE)
        if forgery_image is None:
            messagebox.showerror("Error",      "Failed      to
preprocess images. Please check the image paths.")
            return False

        # Resize images to the same dimensions for accurate
comparison
        original_image      =      cv2.resize(original_image,
(forgery_image.shape[1], forgery_image.shape[0]))

        # Calculate SSIM similarity between the two images
        similarity = ssim(original_image, forgery_image)

        forgery_threshold  =  0.7    #  You  can  adjust  this
threshold as needed
        if similarity < forgery_threshold:
            messagebox.showerror("Forgery   Detected",   "The
signature appears to be a forgery!")
        else:
            messagebox.showinfo("Forgery Not Detected", "The
signature seems genuine.")
        write_to_csv(original_path, forgery_path, similarity)
        return True
    elif os.path.isdir(forgery_path):
        folder_images = [os.path.join(forgery_path, f) for f
in               os.listdir(forgery_path)                  if
os.path.isfile(os.path.join(forgery_path, f))]

        highest_similarity = 0
        highest_similarity_image = None
        for image_path in folder_images:
            forgery_image       =      cv2.imread(image_path,
cv2.IMREAD_GRAYSCALE)
            if forgery_image is None:
                messagebox.showerror("Error",   f"Failed   to
preprocess image: {image_path}")
                continue
            # Resize images to the same dimensions for accurate
comparison
            original_image_resized                          =
cv2.resize(original_image,           (forgery_image.shape[1],
forgery_image.shape[0]))

            # Calculate SSIM similarity between the two images
            similarity     =     ssim(original_image_resized,
forgery_image)

            if similarity > highest_similarity:
                highest_similarity = similarity
                highest_similarity_image = image_path

        if highest_similarity_image is not None:
            forgery_threshold = 0.7   # You can adjust this
threshold as needed
            if highest_similarity < forgery_threshold:
                messagebox.showerror("Forgery      Detected",
f"The   signature   appears   to   be   a   forgery   with
{os.path.basename(highest_similarity_image)}!")
            else:
                messagebox.showinfo("Forgery  Not  Detected",
f"The      signature      seems      genuine      with
{os.path.basename(highest_similarity_image)}!")
            write_to_csv(original_path,
highest_similarity_image, highest_similarity)
        else:
            messagebox.showerror("Error", "No images found in
the folder.")
        return True
    else:
        messagebox.showerror("Error",      "Invalid      path
provided.")
        return False



def detect_attendance(window, path):
    # Specify the path to your attendance.py script
    attendance_script_path                                  =
"C:/Users/Mukesh/OneDrive/Desktop/Signature-Matching-
```

```
main/Signature-Matching-main/attendance
signature/attendance.py"

    # Check if the attendance script exists
    if os.path.exists(attendance_script_path):
        # Use subprocess to execute the attendance.py script
        subprocess.Popen(["python", attendance_script_path])
    else:
        messagebox.showerror("Error", "attendance.py script
not found.")
def login():
    # Replace 'username' and 'password' with the actual
credentials for login
    username = "karan"
    password = "123456"

    entered_username = username_entry.get()
    entered_password = password_entry.get()

    if entered_username == username and entered_password ==
password:
        # Destroy the login window and show the main
application window
        login_window.destroy()
        root.deiconify()  # Show the main application window
    else:
        messagebox.showerror("Login Failed", "Invalid
username or password!")

def show_password():
    if password_entry["show"] == "":
        password_entry["show"] = "*"
    else:
        password_entry["show"] = ""

def show_login_page():
    global login_window, username_entry, password_entry
    login_window = tk.Toplevel()
    login_window.title("Login")

    # Set the window size
    width = 300
    height = 300

    # Center the window on the screen
    center_window(login_window, width, height)

    # Attempt to load the background image
    try:
        bg_image                                          =
Image.open("C:/Users/Mukesh/OneDrive/Desktop/Signature-
Matching-main/Signature-Matching-main/logo/loginpage4.jpg")
        bg_photo = ImageTk.PhotoImage(bg_image)
        bg_label = tk.Label(login_window, image=bg_photo)
        bg_label.image = bg_photo  # Keep a reference to the
image to avoid garbage collection
        bg_label.place(x=0, y=0, relwidth=1, relheight=1)
    except Exception as e:
        print("Error loading background image:", e)
        login_window.configure(bg="lightgray")    #  Set  a
default background color if the image fails to load

    # Create a frame for the login elements with a transparent
background
    frame = tk.Frame(login_window, bg="black", bd=5)
    frame.place(relx=0.5, rely=0.5, anchor="center")

    login_label = tk.Label(frame, text="Login", font=("Arial",
16))
    login_label.pack(pady=10)

    username_label   =   tk.Label(frame,   text="Username:",
font=10)
    username_label.pack(anchor="w", padx=5, pady=5)

    username_entry = tk.Entry(frame, font=10)
    username_entry.pack(anchor="w", padx=5, pady=5)

    password_label   =   tk.Label(frame,   text="Password:",
font=10)
    password_label.pack(anchor="w", padx=5, pady=5)
    password_entry = tk.Entry(frame, font=10, show="*")
    password_entry.pack(anchor="w", padx=5, pady=5)

    show_password_button = tk.Button(frame, text="Show
Password", font=10, command=show_password, bg='blue',
activebackground='darkblue', fg='white', padx=5, pady=2)
    show_password_button.pack(side=tk.LEFT, padx=5, pady=5)

    login_button = tk.Button(frame, text="Login", font=10,
command=login, bg='blue', activebackground='darkblue',
fg='white', padx=10, pady=5)
    login_button.pack(side=tk.RIGHT, padx=10, pady=5)

    # Hide the login window when the main application window
is displayed
    login_window.protocol("WM_DELETE_WINDOW", root.destroy)

    # Start the login window's main loop
    login_window.mainloop()



def open_file():
    file_path = 'performance_data.csv'
    if os.path.exists(file_path):
        try:
            os.startfile(file_path)  # This will open the file
with the default program
        except Exception as e:
            messagebox.showerror("Error", f"Failed to open the
file: {e}")
    else:
        messagebox.showerror("File Not Found", "The
performance_data.csv file does not exist.")

def exit_application():
    if messagebox.askokcancel("Exit", "Are you sure you want
to exit?"):
        root.destroy()
def center_window(window, width, height):
    # Get the screen width and height
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()
    # Calculate the x and y coordinates to center the window
    x = (screen_width - width) // 2
    y = (screen_height - height) // 2

    # Set the geometry of the window to the center of the
screen
    window.geometry(f"{width}x{height}+{x}+{y}")

def main():
    global root
    root = tk.Tk()
    root.title("Signature Matching")

    # Set the window to full-screen mode
    root.attributes('-fullscreen', True)

    # Set the window size
    width = 500
    height = 700

    # Center the window on the screen
    center_window(root, width, height)

    # Center the window on the screen
    center_window(root, width, height)

    logo_path = "logo2.png"
    try:
        bg_image                                         =
Image.open("C:/Users/Mukesh/OneDrive/Desktop/Signature-
Matching-main/Signature-Matching-main/logo/logo4.png")
        bg_photo = ImageTk.PhotoImage(bg_image)
        bg_label = tk.Label(root, image=bg_photo)
        bg_label.image = bg_photo
        bg_label.place(x=0, y=0, relwidth=1, relheight=1)   #
Make the background image fill the entire window
    except Exception as e:
        print("Error loading background image:", e)
        root.configure(bg="lightgray")    #  Set  a  default
background color if the image fails to load
```

```
    uname_label    =    tk.Label(root,    text="Compare    Two
Signatures:", font=10)
    uname_label.place(x=600, y=150)

    image1_path_entry = tk.Entry(root, font=10)
    image1_path_entry.place(x=600, y=220)
    img1_message = tk.Label(root, text="Signature 1", font=10)
    img1_message.place(x=480, y=220)

    img1_capture_button = tk.Button(
        root,    text="Capture",    font=10,    command=lambda:
captureImage(ent=image1_path_entry, sign=1))
    img1_capture_button.place(x=850, y=190)
    img1_capture_button.configure(bg='green',
activebackground='darkgreen')

    img1_browse_button = tk.Button(
        root,    text="Browse",    font=10,    command=lambda:
browsefunc(ent=image1_path_entry))
    img1_browse_button.place(x=850, y=240)
    img1_browse_button.configure(bg='orange',
activebackground='darkorange')

    image2_path_entry = tk.Entry(root, font=10)
    image2_path_entry.place(x=600, y=340)
    img2_message = tk.Label(root, text="Signature 2", font=10)
    img2_message.place(x=480, y=340)

    img2_capture_button = tk.Button(
        root,    text="Capture",    font=10,    command=lambda:
captureImage(ent=image2_path_entry, sign=2))
    img2_capture_button.place(x=850, y=310)
    img2_capture_button.configure(bg='green',
activebackground='darkgreen')

    img2_browse_button = tk.Button(
        root,  text="Browse  Image",  font=10,  command=lambda:
browsefunc(ent=image2_path_entry))
    img2_browse_button.place(x=850, y=360)
    img2_browse_button.configure(bg='orange',
activebackground='darkorange')
```

```
    folder_browse_button = tk.Button(
        root, text="Browse Folder", font=10, command=lambda:
browse_folder(ent=image2_path_entry))
    folder_browse_button.place(x=850, y=400)
    folder_browse_button.configure(bg='orange',
activebackground='darkorange')

    compare_button = tk.Button(
        root,   text="Compare",   font=10,   command=lambda:
checkSimilarity(window=root,

path1=image1_path_entry.get(),

path2=image2_path_entry.get(),))
    compare_button.place(x=600, y=480)
    compare_button.configure(bg='yellow',
activebackground='lightyellow')

    forgery_check_button = tk.Button(
        root, text="Check Forgery", font=10,
        command=lambda: checkForgery(window=root,

original_path=image1_path_entry.get(),

forgery_path=image2_path_entry.get())
    )
    forgery_check_button.place(x=700, y=480)
    forgery_check_button.configure(bg='red',
activebackground='red')

    show_both_img_button = tk.Button(
        root, text="Show Signature Images", font=10,
        command=lambda:
show_both_signature_images(path1=image1_path_entry.get(),
path2=image2_path_entry.get())
    )
    show_both_img_button.place(x=600, y=520)
    show_both_img_button.configure(bg='blue',
activebackground='darkblue')
    attendance_button = tk.Button(
        root, text="Attendance Detection", font=10,
        command=lambda:        detect_attendance(window=root,
path=image1_path_entry.get())
    )
    attendance_button.place(x=600, y=580)
    attendance_button.configure(bg='green',
activebackground='darkgreen')

    open_file_button  =  tk.Button(root,  text="Open  File",
font=10,          command=open_file,          bg='yellow',
activebackground='yellow', fg='black', padx=10, pady=5)
    open_file_button.place(x=300, y=650)

    exit_button  =  tk.Button(root,  text="Exit",  font=10,
command=exit_application,                     bg='yellow',
activebackground='yellow', fg='black', padx=10, pady=5)
    exit_button.place(x=1050, y=150)

    # Show the login page initially and hide the main
application window
    root.withdraw()  # Hide the main application window
    show_login_page()

    root.mainloop()

if __name__ == "__main__":
    main()
```

5.System Implementation

5.1 User Interface Design

User Interface (UI) design plays a pivotal role in creating an intuitive and efficient user experience. This section outlines the design principles, layout considerations, and interactive elements incorporated into the Signature Matching application's UI.

Design Principles

Simplicity: The UI is kept clean and straightforward, minimizing clutter and distractions to facilitate ease of use.

Consistency: Consistent layout, colors, and typography are maintained across all screens for a cohesive user experience.

Visual Hierarchy: Important elements such as buttons and input fields are visually emphasized to guide user interactions.

Feedback: Users receive clear and immediate feedback on their actions, enhancing their understanding of the application's response.

Accessibility: The UI design ensures readability and usability for users of all abilities, with appropriate contrast and text sizes.

Layout Considerations

Login Page: A separate login page ensures security and access control, requiring users to authenticate before using the application.

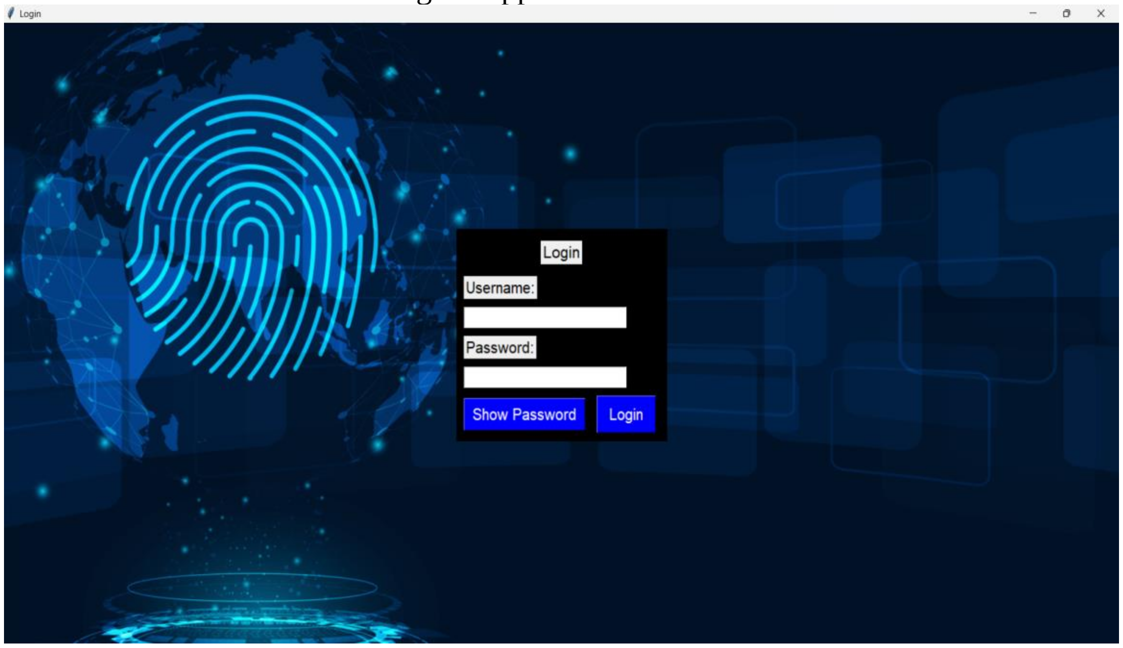


Fig.5.1.1 Login Page

Main Window: The primary window displays the application logo, input fields for signature comparison, Capture , Browser, Browser Folder, Compare ,check Forgery ,open file , Exit

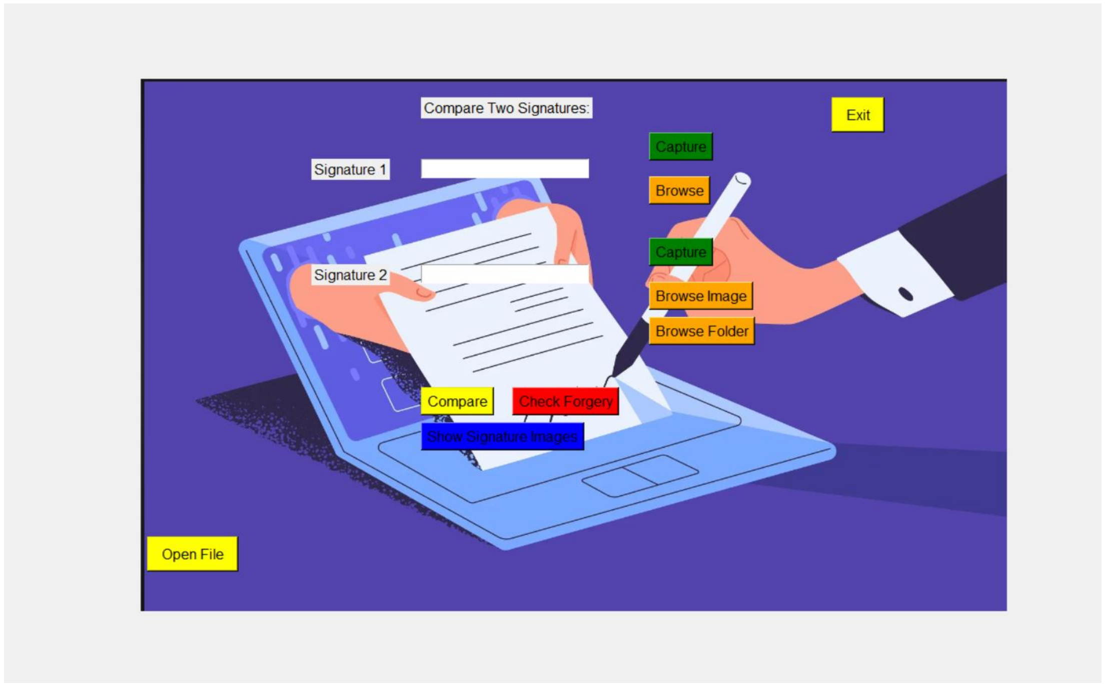


Fig.5.1.2 Main Window

Image Capture: Users can capture a genuine signature using their webcam. The captured image is stored temporarily and its path is defined within the application

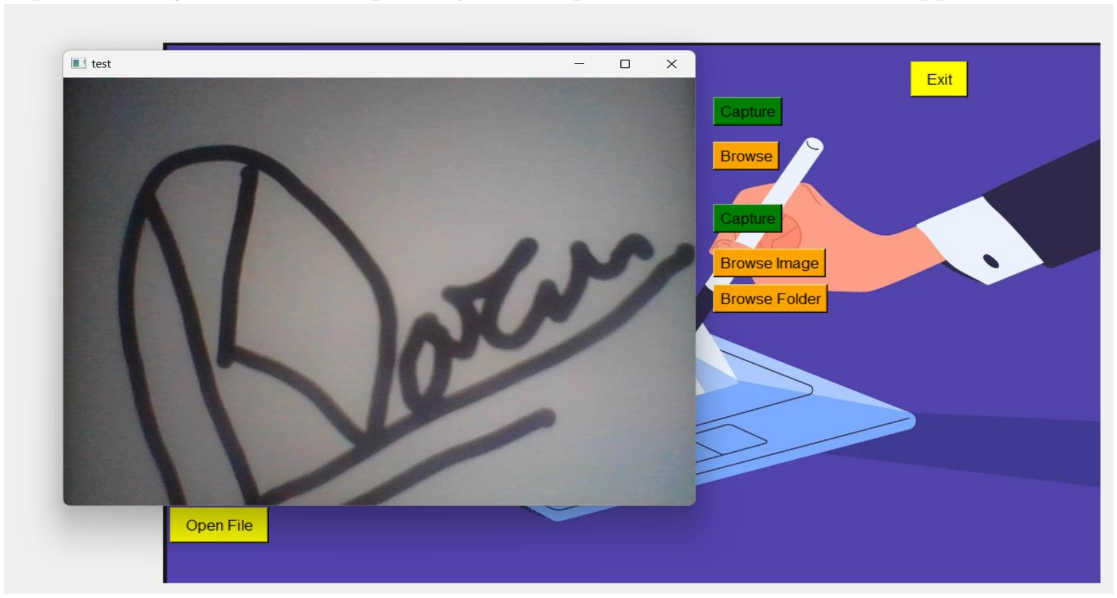


Fig.5.1.3 Image Capture

File Browsing: Users can browse and select an existing image file that contains the genuine signature.

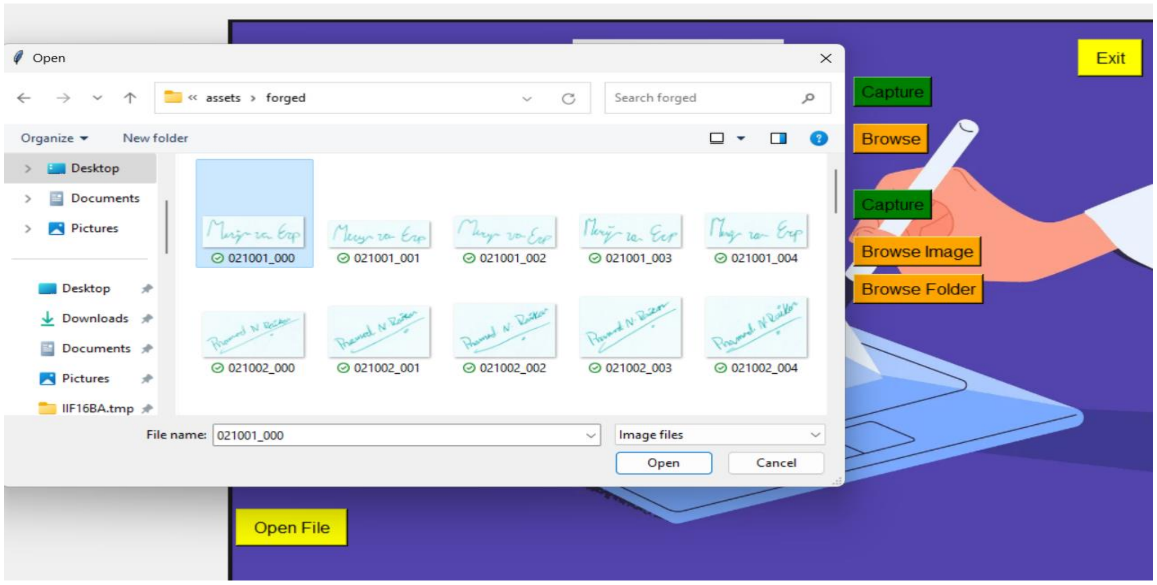


Fig.5.1.4 **File Browsing**

Flask UI Components:

In addition to the Tkinter-based desktop interface, the application also offers a web-based interface using Flask. This enables users to access the application through a web browser, providing greater flexibility and accessibility.

1. Flask App Layout:

Templates: HTML templates define the structure and layout of web pages.
Forms: HTML forms collect user inputs for image paths and similarity percentage.

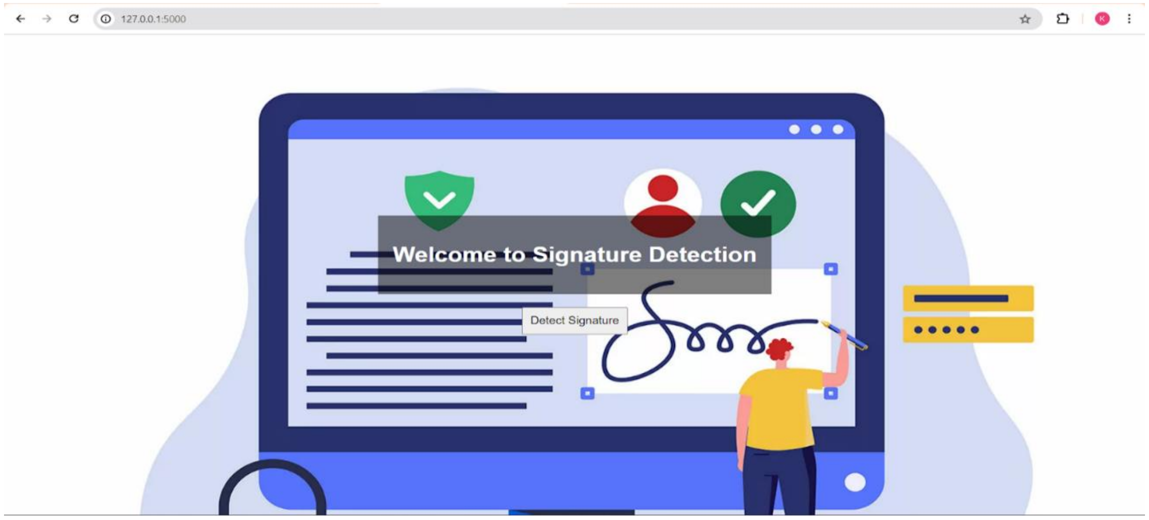


Fig.5.1.5 Flask App Layout

2. Routes:

Define endpoints for different functionalities, such as rendering the home page, processing form submissions, and displaying results.

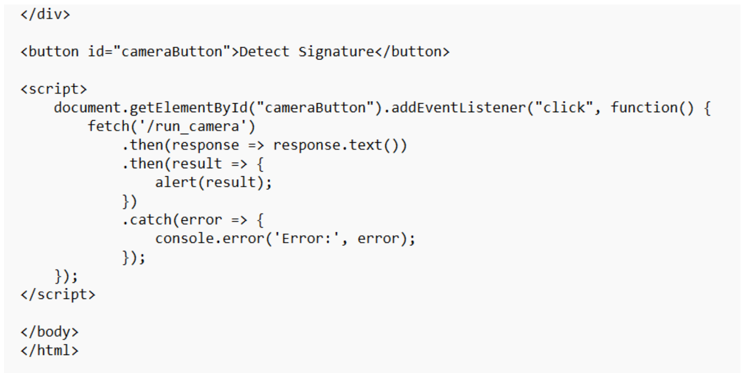
Flask Code Example:

```python
from flask import Flask, render_template
import subprocess

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/run_camera')
def run_camera():
    try:
        subprocess.run(['python', 'main.py'])
        return "Success"
    except Exception as e:
        return f"Error: {str(e)}"


if __name__ == '__main__':
    app.run(debug=True)
```

Fig.5.1.6 Flask App Code

HTML Template Example (index.html):

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Camera and File Buttons</title>
    <style>
        body {
            margin: 0;
            padding: 0;
            height: 100vh; /* Set the body to full viewport height */
            background-image: url('https://codesigningstore.com/wp-content/uploads/2023/11/authenticode-signature-verification-feature-jpg.webp');
            background-size: cover;
            background-repeat: no-repeat;
            background-attachment: fixed;
            display: flex;
            flex-direction: column;
            align-items: center;
            justify-content: center;
            color: white; /* Text color for the welcome header */
            font-family: Arial, sans-serif;
        }

        #welcomeHeader {
            background-color: rgba(0, 0, 0, 0.5); /* Semi-transparent black background */
            padding: 20px;
            text-align: center;
        }

        #cameraButton {
            padding: 10px;
            font-size: 16px;
            margin-top: 20px; /* Add some spacing between the header and button */
        }
    </style>
</head>
<body>

<div id="welcomeHeader">
    <h1>Welcome to Signature Detection</h1>
```



```
</div>

<button id="cameraButton">Detect Signature</button>

<script>
    document.getElementById("cameraButton").addEventListener("click", function() {
        fetch('/run_camera')
            .then(response => response.text())
            .then(result => {
                alert(result);
            })
            .catch(error => {
                console.error('Error:', error);
            });
    });
</script>

</body>
</html>
```

Fig.5.1.7 HTML Template

Interactive Elements

Text Fields: Input fields allow users to enter file paths for signature images and similarity thresholds.

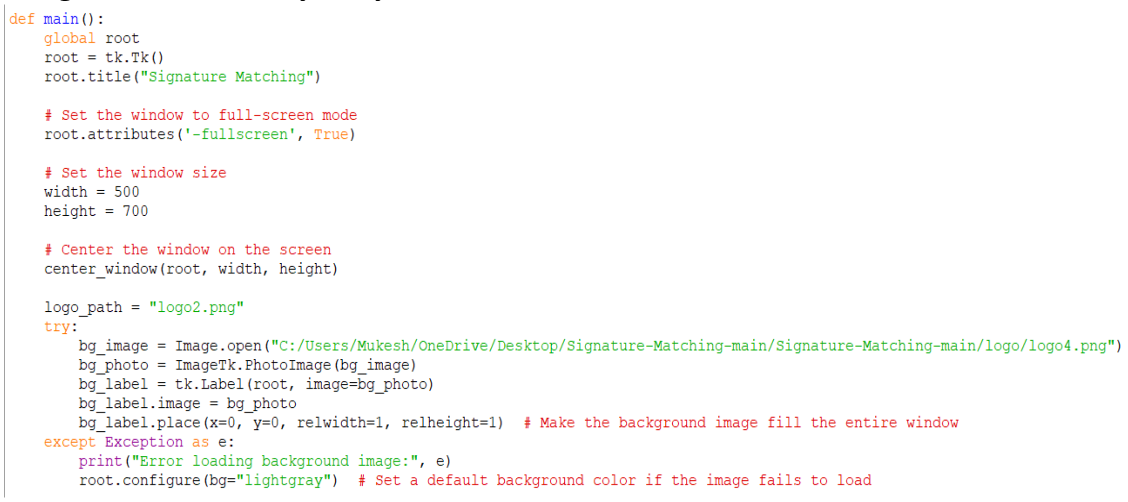
Buttons: Action buttons such as "Compare," "Check Forgery," and "Attendance Detection" trigger specific functionalities when clicked.

File Browser: Browse buttons enable users to select signature images from their system's file explorer, enhancing convenience.

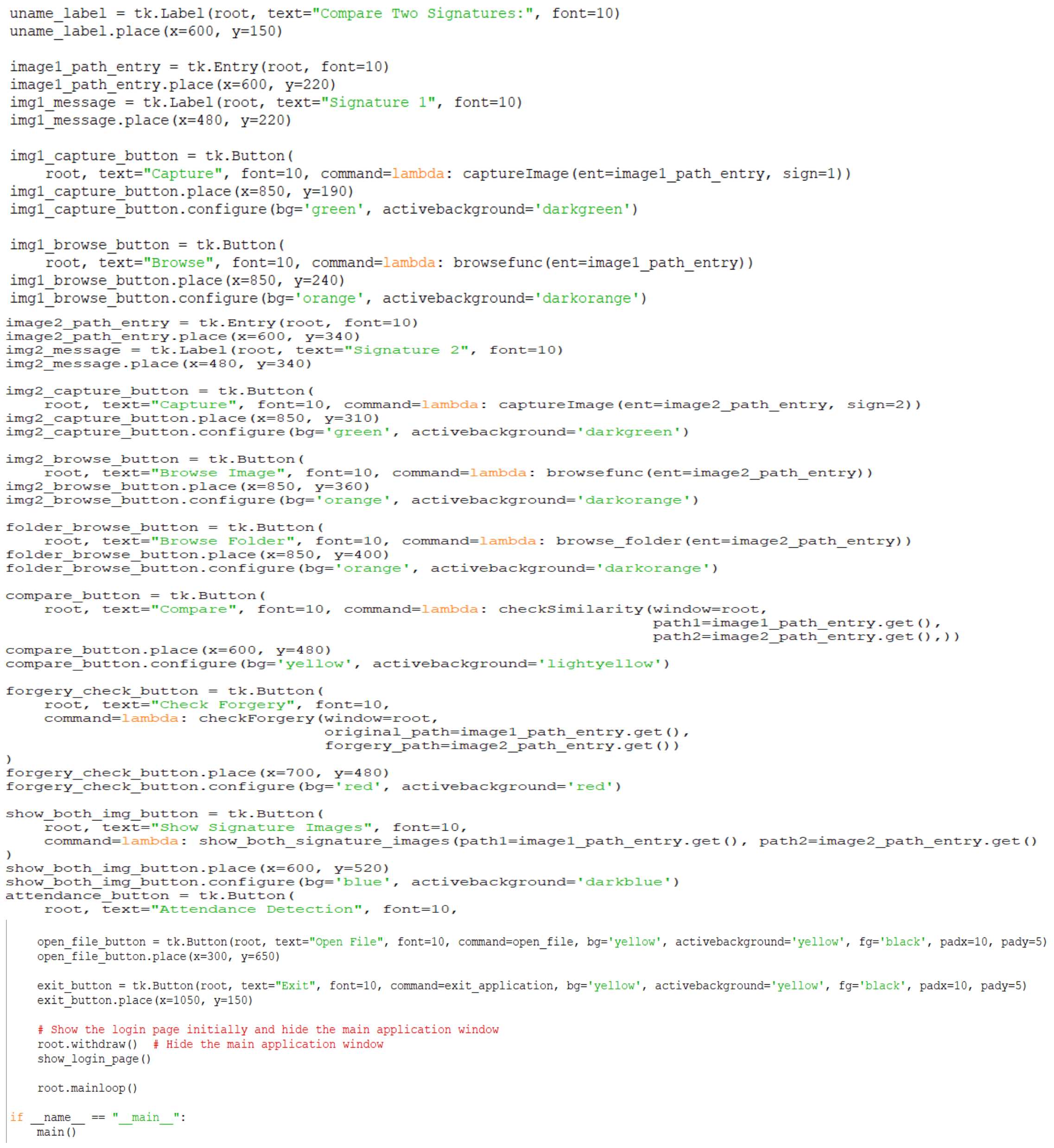
Error Messages: Message boxes provide informative feedback to users in case of input errors or system issues, guiding them on corrective actions.

Implementation

The following code snippets demonstrate the implementation of key UI elements using the Tkinter library in Python:



```
def main():
    global root
    root = tk.Tk()
    root.title("Signature Matching")

    # Set the window to full-screen mode
    root.attributes('-fullscreen', True)

    # Set the window size
    width = 500
    height = 700

    # Center the window on the screen
    center_window(root, width, height)

    logo_path = "logo2.png"
    try:
        bg_image = Image.open("C:/Users/Mukesh/OneDrive/Desktop/Signature-Matching-main/Signature-Matching-main/logo/logo4.png")
        bg_photo = ImageTk.PhotoImage(bg_image)
        bg_label = tk.Label(root, image=bg_photo)
        bg_label.image = bg_photo
        bg_label.place(x=0, y=0, relwidth=1, relheight=1)  # Make the background image fill the entire window
    except Exception as e:
        print("Error loading background image:", e)
        root.configure(bg="lightgray")  # Set a default background color if the image fails to load
```



```
uname_label = tk.Label(root, text="Compare Two Signatures:", font=10)
uname_label.place(x=600, y=150)

image1_path_entry = tk.Entry(root, font=10)
image1_path_entry.place(x=600, y=220)
img1_message = tk.Label(root, text="Signature 1", font=10)
img1_message.place(x=480, y=220)

img1_capture_button = tk.Button(
    root, text="Capture", font=10, command=lambda: captureImage(ent=image1_path_entry, sign=1))
img1_capture_button.place(x=850, y=190)
img1_capture_button.configure(bg='green', activebackground='darkgreen')

img1_browse_button = tk.Button(
    root, text="Browse", font=10, command=lambda: browsefunc(ent=image1_path_entry))
img1_browse_button.place(x=850, y=240)
img1_browse_button.configure(bg='orange', activebackground='darkorange')
image2_path_entry = tk.Entry(root, font=10)
image2_path_entry.place(x=600, y=340)
img2_message = tk.Label(root, text="Signature 2", font=10)
img2_message.place(x=480, y=340)

img2_capture_button = tk.Button(
    root, text="Capture", font=10, command=lambda: captureImage(ent=image2_path_entry, sign=2))
img2_capture_button.place(x=850, y=310)
img2_capture_button.configure(bg='green', activebackground='darkgreen')

img2_browse_button = tk.Button(
    root, text="Browse Image", font=10, command=lambda: browsefunc(ent=image2_path_entry))
img2_browse_button.place(x=850, y=360)
img2_browse_button.configure(bg='orange', activebackground='darkorange')

folder_browse_button = tk.Button(
    root, text="Browse Folder", font=10, command=lambda: browse_folder(ent=image2_path_entry))
folder_browse_button.place(x=850, y=400)
folder_browse_button.configure(bg='orange', activebackground='darkorange')

compare_button = tk.Button(
    root, text="Compare", font=10, command=lambda: checkSimilarity(window=root,
                                                                   path1=image1_path_entry.get(),
                                                                   path2=image2_path_entry.get(),))
compare_button.place(x=600, y=480)
compare_button.configure(bg='yellow', activebackground='lightyellow')

forgery_check_button = tk.Button(
    root, text="Check Forgery", font=10,
    command=lambda: checkForgery(window=root,
                                 original_path=image1_path_entry.get(),
                                 forgery_path=image2_path_entry.get())
)
forgery_check_button.place(x=700, y=480)
forgery_check_button.configure(bg='red', activebackground='red')

show_both_img_button = tk.Button(
    root, text="Show Signature Images", font=10,
    command=lambda: show_both_signature_images(path1=image1_path_entry.get(), path2=image2_path_entry.get()
)
show_both_img_button.place(x=600, y=520)
show_both_img_button.configure(bg='blue', activebackground='darkblue')
attendance_button = tk.Button(
    root, text="Attendance Detection", font=10,

    open_file_button = tk.Button(root, text="Open File", font=10, command=open_file, bg='yellow', activebackground='yellow', fg='black', padx=10, pady=5)
    open_file_button.place(x=300, y=650)

    exit_button = tk.Button(root, text="Exit", font=10, command=exit_application, bg='yellow', activebackground='yellow', fg='black', padx=10, pady=5)
    exit_button.place(x=1050, y=150)

    # Show the login page initially and hide the main application window
    root.withdraw()  # Hide the main application window
    show_login_page()

    root.mainloop()

if __name__ == "__main__":
    main()
```

Fig.5.1.8 Interactive Elements Code

User Feedback

Success Messages: Inform users when operations like signature comparison or forgery detection succeed.

Error Messages: Clearly communicate errors such as invalid file paths or missing input fields, guiding users on corrective actions.

By adhering to these UI design principles and implementing intuitive interactive elements, the Signature Matching application ensures a seamless and user-friendly experience, empowering users to perform signature analysis effectively.

5.2 Login and Authentication

Login and authentication mechanisms are vital components of the Signature Matching application, ensuring secure access control and user verification. This section elaborates on the login process, authentication methods, and security considerations incorporated into the application.

Login Process:

1. User Identification: Users are prompted to enter their credentials, typically consisting of a username and password, to access the application's functionalities.
2. Authentication: Upon submission of credentials, the application validates the user's identity against preconfigured authentication data, such as usernames and hashed passwords.
3. Access Control: After successful authentication, users gain access to the main features and functionalities of the application, while unauthorized users are denied entry.

Authentication Methods:

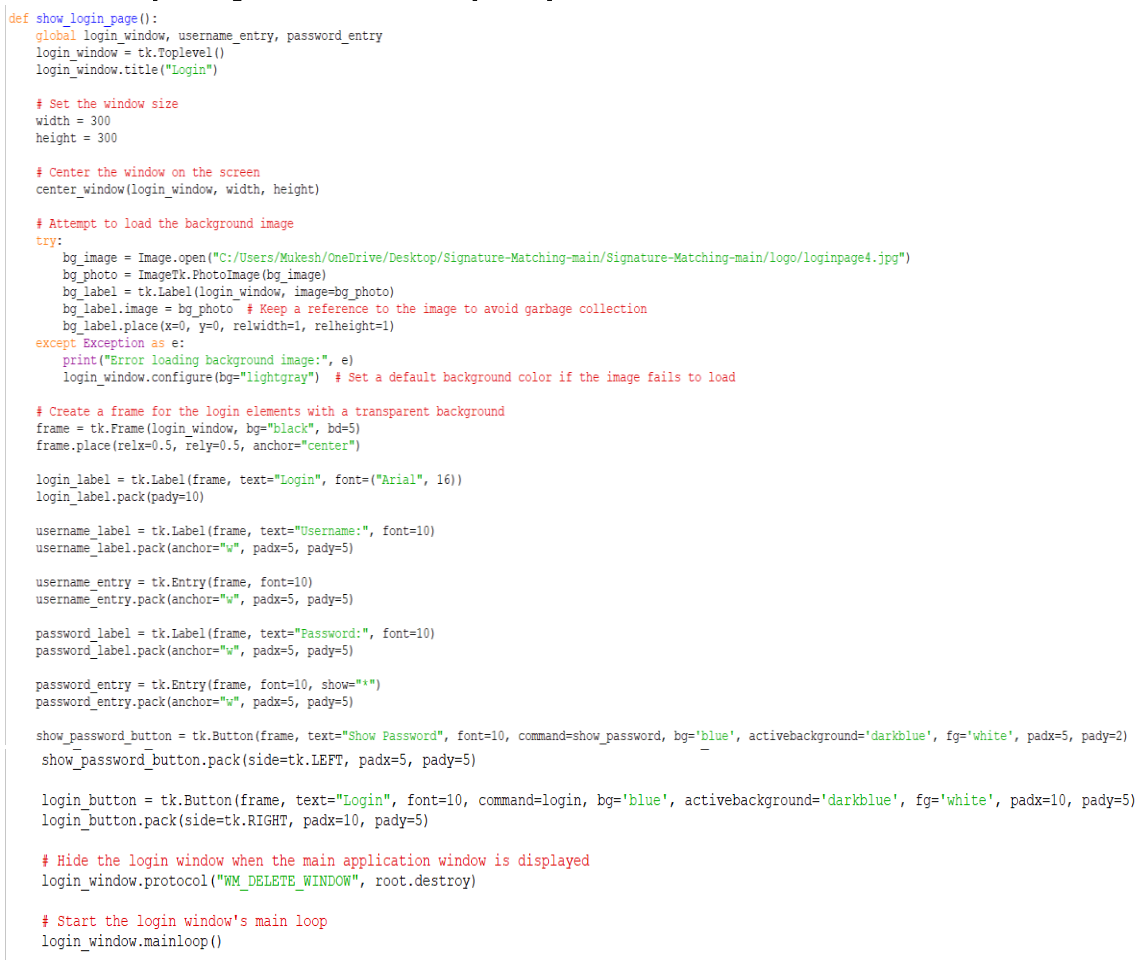
1. Username and Password: The traditional username-password combination is used for authentication, requiring users to provide unique credentials.
2. Hashing and Salt: User passwords are securely hashed using cryptographic algorithms such as SHA-256, with the addition of a unique salt for added security.
3. Session Management: Upon successful login, the application generates a session token or cookie to maintain the user's authenticated state throughout the session.

Security Considerations:

1. Password Strength: Users are encouraged to create strong passwords containing a mix of alphanumeric characters, symbols, and uppercase/lowercase letters to enhance security.
2. Encryption: Communication between the client and server is encrypted using secure protocols such as HTTPS to protect sensitive data during transmission.
3. Brute Force Protection: Mechanisms such as account lockout after multiple failed login attempts prevent brute force attacks on user accounts.
4. Session Expiry: User sessions are automatically terminated after a period of inactivity to mitigate the risk of unauthorized access in case of device theft or shared computers.

Implementation:

The following code snippets illustrate the implementation of login and authentication functionality using the Tkinter library in Python:



```python
def show_login_page():
    global login_window, username_entry, password_entry
    login_window = tk.Toplevel()
    login_window.title("Login")

    # Set the window size
    width = 300
    height = 300

    # Center the window on the screen
    center_window(login_window, width, height)

    # Attempt to load the background image
    try:
        bg_image = Image.open("C:/Users/Mukesh/OneDrive/Desktop/Signature-Matching-main/Signature-Matching-main/logo/loginpage4.jpg")
        bg_photo = ImageTk.PhotoImage(bg_image)
        bg_label = tk.Label(login_window, image=bg_photo)
        bg_label.image = bg_photo  # Keep a reference to the image to avoid garbage collection
        bg_label.place(x=0, y=0, relwidth=1, relheight=1)
    except Exception as e:
        print("Error loading background image:", e)
        login_window.configure(bg="lightgray")  # Set a default background color if the image fails to load

    # Create a frame for the login elements with a transparent background
    frame = tk.Frame(login_window, bg="black", bd=5)
    frame.place(relx=0.5, rely=0.5, anchor="center")

    login_label = tk.Label(frame, text="Login", font=("Arial", 16))
    login_label.pack(pady=10)

    username_label = tk.Label(frame, text="Username:", font=10)
    username_label.pack(anchor="w", padx=5, pady=5)

    username_entry = tk.Entry(frame, font=10)
    username_entry.pack(anchor="w", padx=5, pady=5)

    password_label = tk.Label(frame, text="Password:", font=10)
    password_label.pack(anchor="w", padx=5, pady=5)

    password_entry = tk.Entry(frame, font=10, show="*")
    password_entry.pack(anchor="w", padx=5, pady=5)

    show_password_button = tk.Button(frame, text="Show Password", font=10, command=show_password, bg='blue', activebackground='darkblue', fg='white', padx=5, pady=2)
    show_password_button.pack(side=tk.LEFT, padx=5, pady=5)

    login_button = tk.Button(frame, text="Login", font=10, command=login, bg='blue', activebackground='darkblue', fg='white', padx=10, pady=5)
    login_button.pack(side=tk.RIGHT, padx=10, pady=5)

    # Hide the login window when the main application window is displayed
    login_window.protocol("WM_DELETE_WINDOW", root.destroy)

    # Start the login window's main loop
    login_window.mainloop()
```

Fig.5.2.1 Login and Authentication Code

User Feedback

Success Message: Inform users of successful login attempts, welcoming them to the application.

Error Message: Notify users of failed login attempts due to incorrect credentials, guiding them to retry with the correct information.

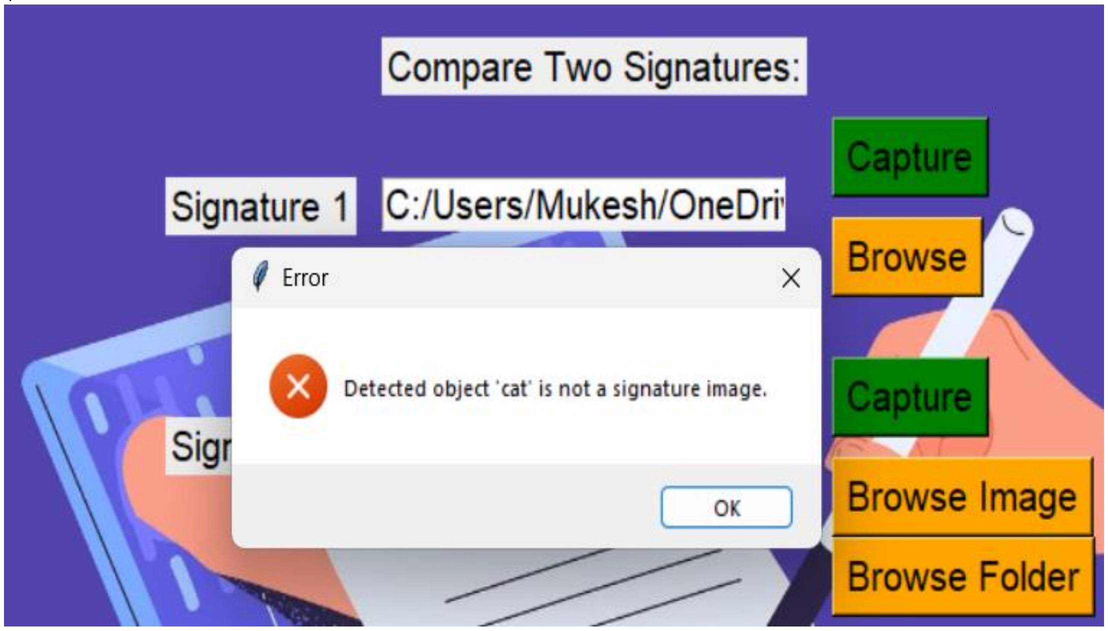
6.RESULTS

- This project will help to identify original & fake signatures.
- Detection of forgeries is possible if datasets are minimum & sufficient.
- If datasets are too large then it may cause an issue.
- Small mistakes by human in signature lead to issue in training & verification phase.
- It has good accuracy when it comes to detection of forgeries but not 100% accuracy.

1. Signature Object Detection and Identification

The signature object detection and identification feature utilizes advanced deep learning techniques to accurately detect and identify signatures within images. The results of this feature are as follows:

```
= RESTART: C:\Users\Mukesh\OneDrive\Desktop\Signature-Matching-main\Signature-Ma
tching-main\main.py
cat
```



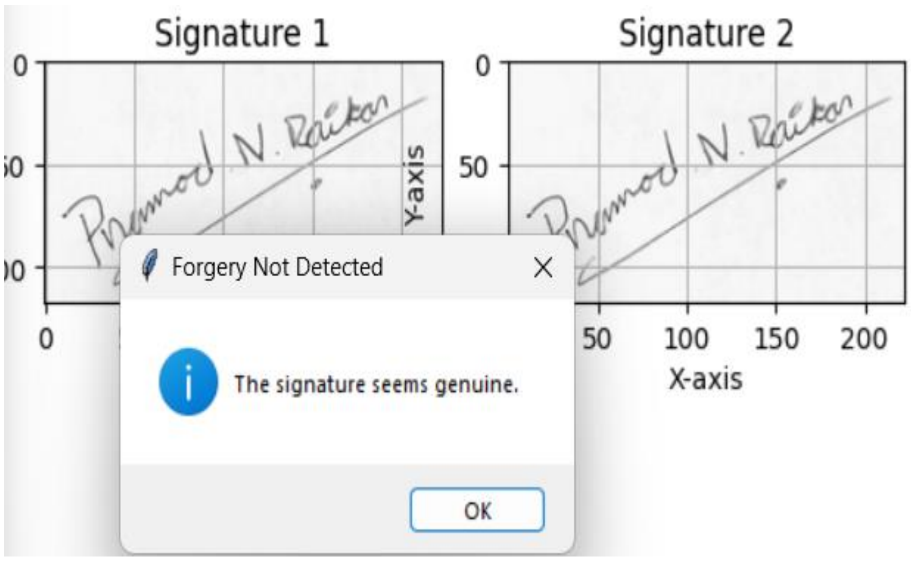
Error

Detected object 'cat' is not a signature image.

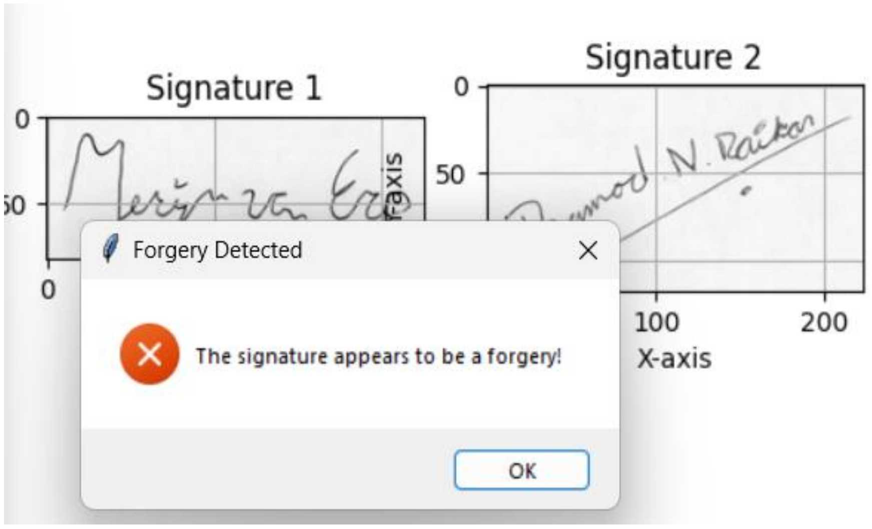
Fig.6.1 Object Detection in image

2. PREDICTING FORGED SIGNATURES

We predict which are genuine or forged signatures in this phase.



The signature seems genuine.

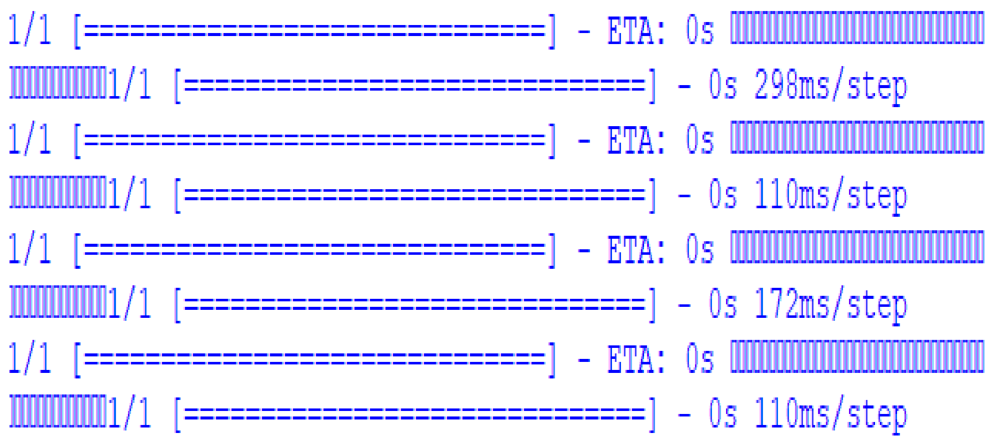


The signature appears to be a forgery!

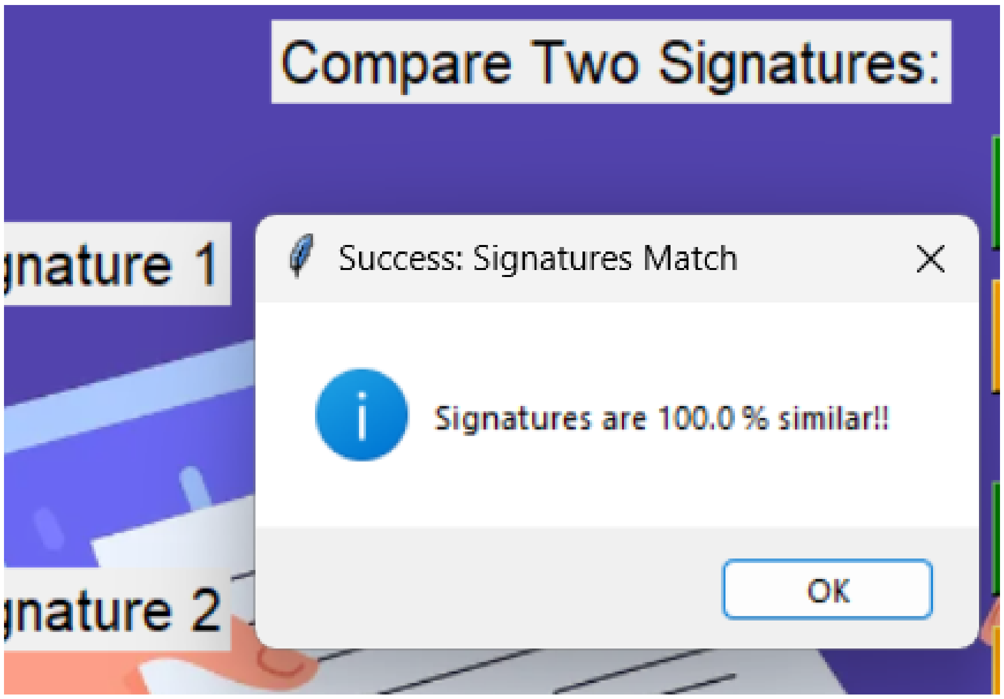
Fig.6.2 Test Result of Signature Genuine and Forgery

3. Accuracy results

We get accuracy of 95% in this model accuracy.



```
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 298ms/step
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 172ms/step
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 110ms/step
```



Success: Signatures Match

Signatures are 100.0 % similar!!

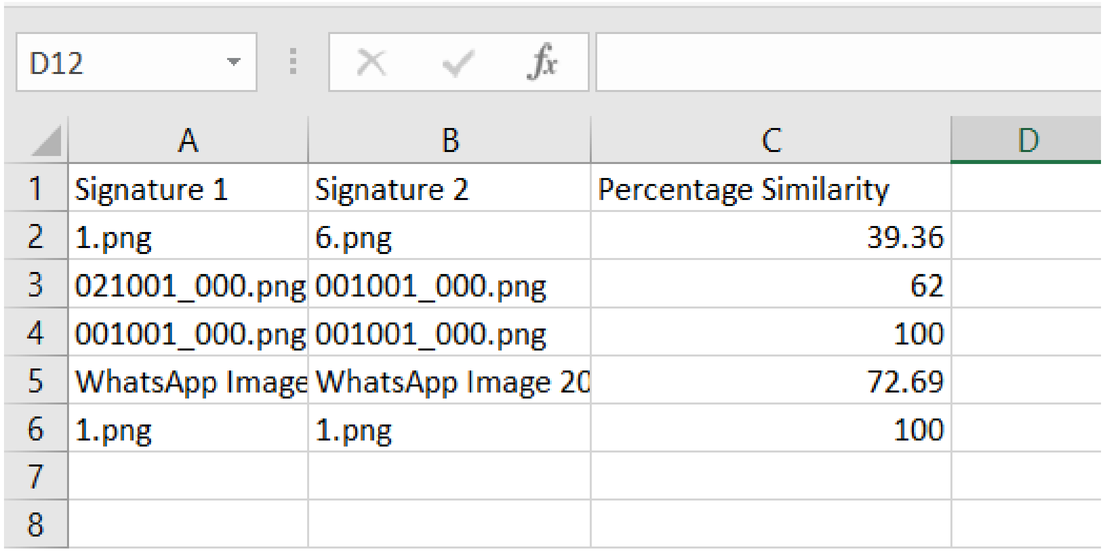
Fig.6.3 Accuracy Result

4. Signature Comparison Results

In more detail, this dataset can include the following types of information:
Image Name: The Name of the original and compared signature images.
Similarity Scores: The similarity percentage between the signatures, indicating how closely they match.
Verification Results: Indicators of whether the signatures are considered a match or forgery based on
the similarity threshold.

Here is a possible schema for the CSV data:

performance_data



D12

	A	B	C	D
1	Signature 1	Signature 2	Percentage Similarity	
2	1.png	6.png	39.36	
3	021001_000.png	001001_000.png	62	
4	001001_000.png	001001_000.png	100	
5	WhatsApp Image	WhatsApp Image 20	72.69	
6	1.png	1.png	100	
7				
8				

Fig.6.4 CSV FILE

7.Maintenance & Evaluation

7.1 System Maintenance

- For the system Maintenance user must ensure that the datasets should not exceed the given limits.
- If using datasets to detect forgeries those datasets should always be in preprocessed form avoiding any issues while detection.
- The system should be regularly updated to ensure that it is using the latest algorithms and technologies. These updates may include bug fixes, security patches, and new features.
- The hardware components of the system, such as the gpu's and other parts should be maintained properly when using new algorithm and testing high amount of datasets which generally require more time.

7.2 Future Enhancements

- Implement advanced Deep learning algorithms to improve the system's accuracy in detecting signature forgeries.
- Add support for detecting digital signature forgeries in addition to traditional signatures.
- Improve the user interface to make it easier for users to navigate and interpret the results of the forgery detection.
- Expand the system's functionality to support bulk forgery detection, where a large number of signatures can be processed at once.
- Add support for detecting more complex forms of signature forgeries, such as those created using deepfake technology.
- Integrate the forgery detection system with other document management and authentication systems to create a more comprehensive approach to document verification.
- Incorporate a fraud detection system that can identify unusual patterns of behavior or suspicious activity related to signature authentication.
- Collect a more diverse and representative dataset for training the forgery detection system, including signatures from different cultural backgrounds and professions.
- Ability to work on realtime data signatures by using more complex machine learning algorithms.
- Implement mechanisms for users to provide feedback on the system's performance and accuracy, including the ability to report false positives and false negatives.

- Use user feedback to improve the forgery detection system's accuracy and address any issues or areas for improvement.
- Continuous Monitoring:
- Implement continuous monitoring of the forgery detection system's performance to identify any anomalies or errors that may affect its accuracy.
- Use Deeplearning algorithms to continuously retrain the system and improve its accuracy over time.

7.3 User Manual

Overview

This user manual provides detailed instructions on how to use the Signature Matching Application. The application allows users to compare two signature images and determine their similarity. The user interface is available both as a desktop application using Tkinter and as a web-based application using Flask.

System Requirements

- Python 3.x installed on your computer
- Required Python packages: tkinter, opencv-python, Pillow, numpy, scikit-image, tensorflow, matplotlib

Operating System:

Windows, macOS, or Linux
RAM: At least 8 GB
CPU: Intel Core i5 or higher
Storage: At least 100 GB

Getting Started

To begin using the Signature Matching Application, follow these simple steps:

1. Launch the Application: Double-click the application icon or run the executable file to start the application.
2. Login: If prompted, enter your username and password to access the application. If you don't have an account, contact your administrator to create one for you.
3. Explore the Interface: Familiarize yourself with the user interface, which consists of various buttons, input fields, and menus. Each element serves a specific function, as explained in the following sections.

Features Overview

The Signature Matching Application offers the following key features:

- Compare Signatures: Allows you to compare two signature images to determine their similarity percentage.

- Check Forgery: Helps you detect potential forgeries by comparing a signature with a known original.
- Capture Images: Enables you to capture signature images directly from your webcam for comparison or analysis.
- Browse Images: Allows you to select signature images from your local filesystem for comparison or analysis.
- Attendance Detection: Utilizes signature recognition to detect attendance based on provided signatures.
- Open File: Opens the performance data file to view previously saved comparisons and analysis results.
- Exit: Closes the application and exits.

Using the Application

To use the Signature Matching Application effectively, follow these steps:

1. Select Images: Choose the signature images you want to compare or analyze using the "Browse" or "Capture" buttons.
2. Perform Comparison: Click the "Compare" button to compare the selected signature images. The application will display the similarity percentage between the two signatures.
3. Forgery Detection: If desired, use the "Check Forgery" button to detect potential forgeries by comparing a signature with a known original.
4. Attendance Detection: Utilize the "Attendance Detection" feature to detect attendance based on provided signature images.
5. View Results: After each operation, review the results displayed by the application to determine the similarity, forgery status, or attendance detection outcome.
6. Save Data: Optionally, save the comparison or analysis results to a file for future reference using the "Save" or "Export" options.

Troubleshooting

If you encounter any issues while using the Signature Matching Application, try the following troubleshooting steps:

- Check Inputs: Ensure that you have provided valid inputs, such as correct file paths or image selections.
- Restart Application: Sometimes, restarting the application can resolve temporary glitches or errors.
- Contact Support: If the issue persists, contact technical support for assistance.

Feedback and Support

We value your feedback! If you have any suggestions for improving the application or encounter any issues while using it, please don't hesitate to contact our support team. Your input helps us enhance the application and provide better user experiences for everyone.

8.Conclusion

In our research endeavor, we aimed to develop a comprehensive system for signature comparison and forgery detection by integrating object detection capabilities alongside deep learning and innovative image analysis techniques. The objective was to address the critical challenge of verifying handwritten signatures, which holds paramount significance across various domains such as security, finance, and legal realms.

Utilizing essential Python libraries such as tkinter, cv2, PIL, and numpy, we created an interactive and user-friendly interface facilitating the seamless capturing, preprocessing, and analysis of signatures. In addition to traditional image processing methods, we integrated object detection functionality using the YOLOv3 model to identify specific objects within the signature images.

The incorporation of the VGG16 neural network architecture for feature extraction, along with the calculation of the Structural Similarity Index (SSIM), equipped our system with the capability to discern intricate details of signatures and differentiate genuine ones from forgeries. By leveraging object detection, our system can identify and isolate signature regions within complex images, enhancing the accuracy of the comparison process.

Our research underscores the importance of accurate preprocessing, robust feature extraction, and meticulous model training for achieving commendable results in signature analysis. The effectiveness of our proposed system is demonstrated through successful detection and differentiation of genuine signatures from forged ones, utilizing both deep learning insights and image similarity metrics.

Furthermore, the visual interpretation of model accuracy and loss graphs provides valuable insights for performance evaluation and potential improvements. By integrating object detection capabilities, our system offers enhanced versatility and accuracy in signature analysis, contributing a significant advancement in the domain of signature authentication.

As technology continues to evolve, the challenge of signature forgery detection remains dynamic. However, our research paper lays a strong foundation and opens avenues for further advancements in the realm of signature authentication, paving the way for more robust and sophisticated systems to safeguard the integrity of signatures.

9.Bibliography & References

1. Handwritten Signature Forgery Detection using Convolutional Neural Networks
2. Handwritten Signature Forgery Detection using Convolutional Neural Networks ScienceDirect
3. HANDWRITTEN SIGNATURES FORGERY DETECTION IRJET-V8I1335.pdf
4. Siamese Triple Ranking Convolution Network In Signature Forgery Detection
5. (PDF) Siamese Triple Ranking Convolution Network in Signature Forgery Detection Ojaswini Chhabra - Academia.edu
6. Signature Verification And Forgery Detection Using Fuzzy Modelling Approach
7. (PDF) Signature Verification and Forgery Detection System (researchgate.net)
8. Poddar, J., Parikh, V., & Bharti, S. K. (2020). Offline signature recognition and forgery detection using deep learning. Procedia Computer Science, 170, 610-617.
9. S. S. Channappayya, A. C. Bovik, C. Caramanis and R.W. Heath, "Design of linear equalizers optimized for the structural similarity index", *IEEE Transactions on Image Processing*, vol. 17, no. 6, pp. 857-872, jun 2008.
10. Wang, X., Cao, W., Yao, C., & Yin, H. (2020, July). Feature Matching Algorithm Based on SURF and Lowes Algorithm. In 2020 39th Chinese Control Conference (CCC) (pp. 5996-6000). IEEE.
11. Zhang, L., Cai, F., Wang, J., Lv, C., Liu, W., Guo, G., ... & Xing, Y. (2020, November). Image matching algorithm based on ORB and k-means clustering. In 2020 5th International Conference on Information Science, Computer Technology and Transportation (ISCTT) (pp. 460-464). IEEE.
12. Sulaiman, S. N., Hassan, N. A., Isa, I. S., Abdullah, M. F., Soh, Z. H. C., &Jusman, Y. (2021, August). Mass Detection in Digital Mammogram Image using Convolutional Neural Network (CNN). In 2021 11th IEEE International Conference on Control System, Computing and Engineering (ICCSCE) (pp. 61-65). IEEE.
13. Gowri, P., Sivapriya, G., Kamaleshwar, N. K. J., &Kesavaraj, N. (2022, April). Real Time Signature Forgery Detection Using Machine Learning. In 2022 Second International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT) (pp. 1-5). IEEE.
14. Summra, S., Usman, M. G., & Muhammad, A. (2021, November). Supervised Neural Network for Offline Forgery Detection of Handwritten Signature. In 2021 18th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE) (pp. 1-6). IEEE.
15. Malakar, S., &Chiracharit, W. (2020, September). Thai Text Detection and Classification Using Convolutional Neural Network. In 2020 59th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE) (pp. 99-102). IEEE.

16. Lin, K., & Wang, Z. (2022, January). Traffic Sign Classification by Using Learning Methods: Deep Learning and SIFT Based Learning Algorithm. In 2022 14th International Conference on Computer Research and Development (ICCRD) (pp. 239-243). IEEE.
17. Hai-Yan, Y., & Yu-Xin, H. (2017, October). An image retrieval algorithm based on SURF for embedded system. In 2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA) (pp. 86-88). IEEE.
18. Bhunia, A. K., Chowdhury, P. N., Yang, Y., Hospedales, T. M., Xiang, T., & Song, Y. Z. (2021). Vectorization and rasterization: Self-supervised learning for sketch and handwriting. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 5672-5681).
19. LSharma, P., Hans, P., & Gupta, S. C. (2020, January). Classification of plant leaf diseases using machine learning and image preprocessing techniques. In 2020 10th international conference on cloud computing, data science & engineering (Confluence) (pp. 480-484). IEEE.
20. Engin, D., Kantarci, A., Arslan, S., &Ekenel, H. K. (2020). Offline signature verification on real-world documents. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (pp. 808-809).
21. Yan, K., Zhang, Y., Tang, H., Ren, C., Zhang, J., Wang, G., & Wang, H. (2022). Signature Detection, Restoration, and Verification: A Novel Chinese Document Signature Forgery Detection Benchmark. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 5163-5172).
22. Dey, S., Dutta, A., Toledo, J. I., Ghosh, S. K., Lladós, J., & Pal, U. (2017). Signet: Convolutional siamese network for writer independent offline signature verification. arXiv preprint arXiv:1707.02131.
23. Handwritten Signature Forgery Detection using Convolutional Neural Networks ScienceDirect