SMART SIGNATURE VERIFICATION SYSTEM

A project submitted to the Bharathidasan University
in partial fulfillment of the requirements
for the award of the Degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Submitted by

KAYATHRI V
235114248

Under the guidance of
Dr. M. JAYAKKUMAR M.Sc., M.Phil., MCA., Ph.D.,
Associate Professor

PG DEPARTMENT OF COMPUTER SCIENCE (S.F)
BISHOP HEBER COLLEGE (AUTONOMOUS)
(Nationally Reaccredited by NAAC at 'A++' Grade with a CGPA of 3.69 out of 4)
(Recognized by UGC as "College of Excellence")
(Affiliated to Bharathidasan University)

TIRUCHIRAPPALLI-620 017

OCTOBER – 2025

DECLARATION

I hereby declare that the project work presented is originally done by me under the guidance of **Dr. M. JAYAKKUMAR M.Sc., M.Phil., MCA., Ph.D., Associate Professor, PG Department of Computer Science (S.F), Bishop Heber College (Autonomous), Tiruchirappalli-620 017** and has not been included in any other thesis/project submitted for any other degree.

Name of the Candidate : KAYATHRI V

Register Number : 235114248

Batch : 2023 -2026

Signature of the Candidate

Dr. M. JAYAKKUMAR M.Sc., M.Phil., MCA., Ph.D.,
Associate Professor ,
PG Department of Computer Science (S.F),
Bishop Heber College (Autonomous),
Tiruchirappalli – 620017

Date:

CERTIFICATE

This is to certify that the project work entitled **"SMART SIGNATURE VERIFICATION SYSTEM"** is a bonafide record work done by **KAYATHRI V, 235114248** in partial fulfillment of the requirements for the award of the degree of **BACHELOR OF SCIENCE IN COMPUTER SCIENCE** during the period **2023 – 2026.**

Place: Trichy **Signature of the Guide**

PG DEPARTMENT OF COMPUTER SCIENCE (S.F)
BISHOP HEBER COLLEGE (AUTONOMOUS),
(Nationally Reaccredited by NAAC at 'A++' Grade with a CGPA of 3.69 out of 4)
("Recognized by UGC as "College of Excellence")
(Affiliated to Bharathidasan University)
TIRUCHIRAPPALLI - 620017

Date:

Course Title: Project **Course Code: U21CS6PJ**

CERTIFICATE

The Viva-Voce examination for the candidate **KAYATHRI V, 235114248** was held on _____.

Signature of the Guide **Signature of the HOD**

Examiners:

1.

2.

ACKNOWLEDGEMENT

I want to thank the Principal **Dr. J. PRINCY MERLIN, M.Sc., SET, B.Ed., M.Phil., Ph.D., PGDLC, Principal,** Bishop Heber College (Autonomous), sincerely. She offered real encouragement along the way. Gave us steady support too. And that infrastructure was just excellent. All of it helped get the project done right. The one called Smart Signature Verification System. Her motivation never let up. Leadership like that builds a place for learning. For innovation even. Right through the whole development.

I would like to thank our Head of the Department**, Dr. G. SOBERS SMILES DAVID**, **M.C.A., M.Phil., NET., Assistant Professor, Head,** PG Department of Computer Science, Bishop Heber College (Autonomous), Tiruchirappalli from the heart. Guidance came from him. Kept us on track. Monitoring was constant. Technical advice helped big time. Suggestions came when needed. Boosted the quality of everything, Accuracy too. In the work we did.

A special thanks to the Guide**, Dr. M. JAYAKKUMAR M.Sc., M.Phil., M.C.A., Ph.D., Associate Professor**, PG Department of Computer Science, Bishop Heber College (Autonomous) Tiruchirappalli. Dedication in guiding us. Expert supervision in every step. Encouragement that stuck around. Inputs were valuable. Expertise professional. Feedback constructive always. Built the base for hitting objectives. We succeeded because of that.

KAYATHRI V

0% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography

Match Groups

- 1 Not Cited or Quoted 0%
  Matches with neither in-text citation nor quotation marks
- 0 Missing Quotations 0%
  Matches that are still very similar to source material
- 0 Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

0% Internet sources
0% Publications
0% Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

SYNOPSIS:

People still talk about how signature checks are key in this digital world. They help confirm who someone is in places like banks, legal papers, government files, and schools. Doing it by hand takes forever. It leads to mistakes from people. And it is not reliable with tons of documents. So to fix that, they came up with the Smart Signature Verification System. It is an offline app that uses machine learning and image processing. The goal is to check handwritten signatures fast and right. This system automates the whole verification thing. It looks at structural features, geometric ones, and statistical patterns in the signature image. First, you get the user's signature image. Then preprocess it to make it better. That means turning it grayscale, removing noise, thresholding, and resizing. All that standardizes the image for analysis. Next, extract features like shape, slant, curves, and pixel intensities. Those get compared to stored references. A trained model does the work, say SVM or CNN. It decides if the signature is real or fake. The setup runs offline. No internet needed. That keeps it secure. It works great in spots with bad connections, like rural banks or offices. Offline means better privacy for data. No one sneaks in on sensitive stuff. The architecture has modules for data collection, preprocessing, feature extraction, training the model, and output. They all fit together for good validation. It cuts down on people doing the work. So fewer errors. And things run smoother. The interface is easy. Upload a signature. See the results. Fits right into admin or bank flows. Organizations get faster, more reliable ID checks. Data stays safe. Fraud drops. This project shows how image processing and AI tackle real authentication issues. It highlights smart automation for documents. And it sets up for more, like multi-user stuff, cloud links, deeper learning. The Smart Signature Verification System is a solid way to handle secure signatures offline.

CONTENTS

S.N0	Contents	Page No
1	**Project Description** 1.1 Introduction 1.2 Existing System 1.3 Proposed System	1
2	**Logical Development** 2.1 Data Flow Design 2.2 DFD Levels 2.3 Architectural Design	15
3	**Data Base Design** 3.1 Data Dictionary 3.2 Table Design 3.3 Relationships	25
4	**Program Design** 4.1 Module & Description	30
5	**Testing** 5.1 Types of Testing	36
6	**Conclusion**	44
7	**References**	45
8	**Appendix** 8.1 Source Code 8.2 O/P Screen	47

CHAPTER- 1
PROJECT DESCRIPTION

1.1 INTRODUCTION

People still rely on handwritten signatures a lot in this digital age. You know, for all sorts of things like banking or legal stuff. They are pretty trusted. Still, checking them by hand takes forever. Its subjective too. And not always spot on Especially with big piles of documents or good fakes. So, they came up with this Smart Signature Verification System. Its meant to fix those issues. The thing works offline. Uses image processing and machine learning to check signatures automatically. It looks at shapes, textures, how its oriented, stroke patterns. All that from the image. Algorithms make it accurate. Reliable too. The process has a few main steps. First, preprocessing the image. Turn it grayscale. Filter out noise. Resize it so everything matches. Then, extract features. Geometric ones, structural stuff. Feed that into a model like SVM or neural network. Train it on real signatures. After that, it classifies new ones. Compares to the database ones. Says genuine or forged. Big plus is the offline part. No need for internet. Keeps data private. Great for banks, government spots, law offices. Even works in remote places with no connection. Protects from hacks. Makes it secure. It cuts down on human work. Less mistakes from manual checks. Processes tons fast. Gets results quick. User interface is simple. Upload, run, done in seconds. This setup mixes computer vision, AI, pattern recognition. Shows how tech solves real identity problems. Could lead to more, like cloud links or deeper learning models. Tie into digital docs. The Smart Signature Verification System offline gives a solid way to check signatures without people doing it. Swaps old manual methods for automated ones based on data. Boosts accuracy. Improves security. Helps push digital changes in authentication.

1.2 EXISTING SYSTEM

The current setup for checking handwritten signatures mostly involves people doing it by hand. It depends a lot on what experts know from experience. When someone brings in a signature to verify, a trained person or handwriting specialist looks at it closely. They compare it to the ones on file or the reference examples. Things like the shape of the lines, the way it slants, how much pressure shows up, and the overall flow all come into play. This method has stuck around for years now. Still, it feels pretty subjective. Results can vary a ton, and errors happen because humans are not perfect. The final call rests on the examiner's background, how focused they are that day, and their personal take on it. One person might see it differently from the next.

Lots of places rely on this manual check, you know. Banks do it for transactions. Offices use it for approvals on papers. Legal spots need it for making sure things are authentic. The old way starts to drag when you have piles of documents to go through. It takes up way too much time and energy. Plus, if someone fakes a signature really well, it might slip right past the human eye. That opens the door to fraud. Confidential stuff could get misused in the process.

These days, a few setups try basic digital compares. They match images by looking at each pixel. But those tools are not smart about real differences. Writing style can shift a bit. Pressure changes. Angles vary. Even things like lighting or how the scan turns out mess it up. Natural tweaks in how someone writes over time throw them off too. So, they end up unreliable for everyday stuff. On top of that, many need the internet or some online storage to work. That does not fit in spots with spotty connections or rules about keeping data private.

Without real smarts or automation built in, the whole thing suffers. Accuracy stays low. Processing drags on forever. Security feels shaky. No machine learning or fancy image tricks help sort genuine from fake. Forged ones that look close enough fool it every time. Organizations keep running into these issues. They struggle to get verification that is secure, spot on, and quick.

The current approach really shows the hole here. We need something better. Automated and solid, that works without the web. Keeps privacy intact. Delivers steady, right answers every go. The proposed Smart Signature Verification System fills that spot just fine.

DISADVANTAGES:

The old signature checking setup relies mostly on people doing it by hand or with some basic tools. It runs into big issues with getting things right, wasting time, and protecting info. All that pushes for something smarter and automatic, like this offline Smart Signature Verification System. Thing is, here are the key problems.

Lack of Accuracy:

First off, accuracy just is not there. When folks eyeball signatures, its all based on what they see, which feels off sometimes. Experts might still mess up if the real one and the fake look too close. That drops the whole thing's dependability, you know.

Time-Consuming Process:

Then there the time suck. Going over every signature one by one takes forever and a lot of focus. In places like banks or schools with tons of papers, it drags everything down, makes it way inefficient.

Human Error and Bias:

Human mistakes creep in too, along with biases. Results depend on how tired or stressed the checker is, or even their own views. One person says It’s good, another says no, for the same mark. Unreliable as heck.

No Automated Feature Extraction:

No real way to pull out features automatically either. Old methods skip image tech or learning algorithms to break down the shapes or stats in a signature. So they miss those tiny tells between real and fake.

Difficulty in Handling Variations:

Signatures change a bit each time someone writes them, naturally. But the system does not handle that well. Leads to kicking out good ones by mistake, or letting fakes through.

Dependence on Internet or Online Systems:

Some half-auto setups need the internet to hit databases or tools. That is a pain in spots with bad connections. Plus it worries people about privacy leaks.

High Risk of Forgery and Fraud:

Fakes slip by easy since nothing smart checks them. Big risks for fraud, especially where money or laws are involved and real signatures matter a ton.

Poor Scalability and Data Security:

Scaling up is rough, and security too. More users mean more data to handle manually or online, which overwhelms it. Storing stuff digitally just invites hacks or spills.

1.3 PROPOSED SYSTEM

The Smart Signature Verification System works offline. It tackles problems in old manual checks and half-automated ones. This setup brings in smart automation. It verifies handwritten signatures with real accuracy. The system mixes image handling and machine learning stuff. That way it spots real signatures from fakes pretty well. All of it runs without any online link. So data stays safe and private, you know.

Verification here is all automatic. No need for much human input. That cuts out errors from people judging by eye. First off, it grabs signature samples. Then preprocessing kicks in. Things like turning to grayscale, cutting noise, setting thresholds, resizing images. All that standardizes the pictures. Makes feature pulling more reliable.

Next comes pulling out features. Algorithms look for signature quirks. Geometric ones like shapes, angles, widths, heights. Structural bits too, loops, strokes, curves. And stats on pixels, textures, spreads. Those get fed into a learning model. Could be SVM or CNN type. Trained on real and fake signature sets already.

The model checks the new signature against stored ones. Matches close to the reference patterns. Then it calls it genuine. If not, forged. This keeps results steady and spot on.

Big plus is running offline completely. Preprocessing to final call, all local. No internet required. Speeds things up. Keeps info confidential too. Fits secure spots like banks, offices, legal places. No outside servers mean no breach worries.

Users get a simple interface. Upload a signature easy. Run the check. See results quick, in seconds. Can grow it for more users. Add fresh data sets. Retrain the model for different styles.

In the end, this offline smart system delivers solid efficiency and security. Blends auto tech with smarts. Cuts manual work. Boosts precision. Handles tough spots reliably. Uses AI and image tech to link old methods with new demands.

ADVANTAGES:

The smart signature verification system, the offline one they proposed, really tackles the old problems with just doing checks by hand. It mixes in image processing and machine learning stuff, you know, to give better accuracy, quicker speeds, stronger security, and easier use overall. Thing is, it brings a bunch of solid benefits.

High Accuracy and Reliability:

It nails high accuracy and reliability. The system pulls in advanced algorithms that look at geometric features, structural ones, and statistical bits of the signature. That way, it spots even the tiniest differences between real signatures and fakes. Manual ways just cannot match that consistency.

Automation of the Verification Process:

Automation takes over the whole verification thing. From preprocessing the images right through to classifying them, no human steps in much at all. That cuts out those subjective errors people make. Results come faster, more dependable too.

Offline Functionality and Enhanced Data Security:

Since it runs fully offline, no internet needed for any of the processing or checks. Sensitive data stays right there on the local setup. Keeps things secure, stops any unauthorized peeks or leaks from happening.

Time Efficiency:

Time savings are pretty big here. Automating it all means less time spent analyzing and sorting signatures. It handles a bunch at once in no time. Great for big outfits dealing with tons of documents all the time.

User-Friendly Interface:

The interface is user-friendly, simple to work with. Users upload images easy, run the verification, see the results without hassle. Even folks not tech-savvy can get by with it.

Reduction of Human Error:

It cuts down human errors a lot. Manual checks rely on what people see and their experience, which varies. But this uses set algorithms for objective looks. No fatigue or bias sneaking in to mess things up.

Adaptability and Scalability:

Adaptability stands out, scalability too. Train it on fresh signature data, and it gets better as time goes on. Scale it for more users, or stretch it across different groups and industries.

Improved Fraud Detection:

Fraud detection improves noticeably. Smart feature pulling and classification methods catch forgeries better than old techniques. Lowers risks of scams or identity mix-ups.

Cost-Effective and Resource Saving:

Costs drop once it’s in place. No need for as many staff doing manual verifications. Saves on paper records and all those repeat checks too.

Consistency in Verification Results:

Results stay consistent every time. Being data-driven and algorithm-based, it does not waver no matter how many signatures or under what conditions they get processed.

CHAPTER-2
LOGICAL DEVELOPMENT

2.1 Data Flow Design:

Data Flow Design, or DFD, basically lays out how data moves around in a system. It is this graphical thing that tracks data from where it comes in, through processing, and out to wherever it goes. Plus, it covers storage and how the system actually uses that data.

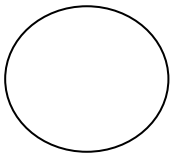
Key Points:

- It represents processes, you know. That means it shows the changes or transformations that happen to the data along the way.
- It also shows data flow pretty clearly. Data moves between processes, data stores, and even external entities.
- Then there is the storage part. It highlights where data gets kept inside the system.
- And it helps with system analysis a lot. You get a solid understanding of what the system does, all without diving into any program code.

Purpose:

- It analyzing and modeling the system in a straightforward way.
- It identifies how data gets handled, stored, and processed.
- It aids in designing the system, documenting it, and developing it too.

Data Flow Symbols:

SYMBOLS	DESCRIPTION
	• It sends data through the system or takes data back from it is known as an **Entity.**
	• A **process** or task that is performed by the system.
	• A **data store,** it is place where data is been stored.
	• It indicates the movements of data between entities, processes and data stores.

2.2 DFD Levels

Level 0

The context level one. It basically shows the whole system as just this one big process. And how it deals with stuff outside itself.

The purpose here is simple. It presents the system like a single unit. And it highlights interactions with external entities.

Those entities include the user. That's the person getting their signature checked or added.

Then there's the admin. Who handles updates and management for the database. The main system process is the Smart Signature Verification System.

It takes in signatures. And it verifies them. Data flows in a couple ways. First, the user sends a signature to the system.

Then the system gives back a result. Like accepted or rejected. Second, the admin updates the signature database.

So the system can use that fresh data for verifications. This level keeps things high-level. It gives an overview. Doesn't dive into internal stuff. Just inputs, outputs, and those outside interactions.

Level 0 DFD (Context Diagram)

Level 1

It breaks things down more. Into major functional modules. You see the flow between them.

The modules start with signature acquisition. That collects the signature digitally. Maybe from a tablet or scanner. Or touch screen.

Then preprocessing. Cleans it up. Removes noise, resizes, normalizes. Gets it ready for analysis.

Feature extraction comes next. Pulls out key characteristics. Like stroke patterns, speed, pressure, shapes.

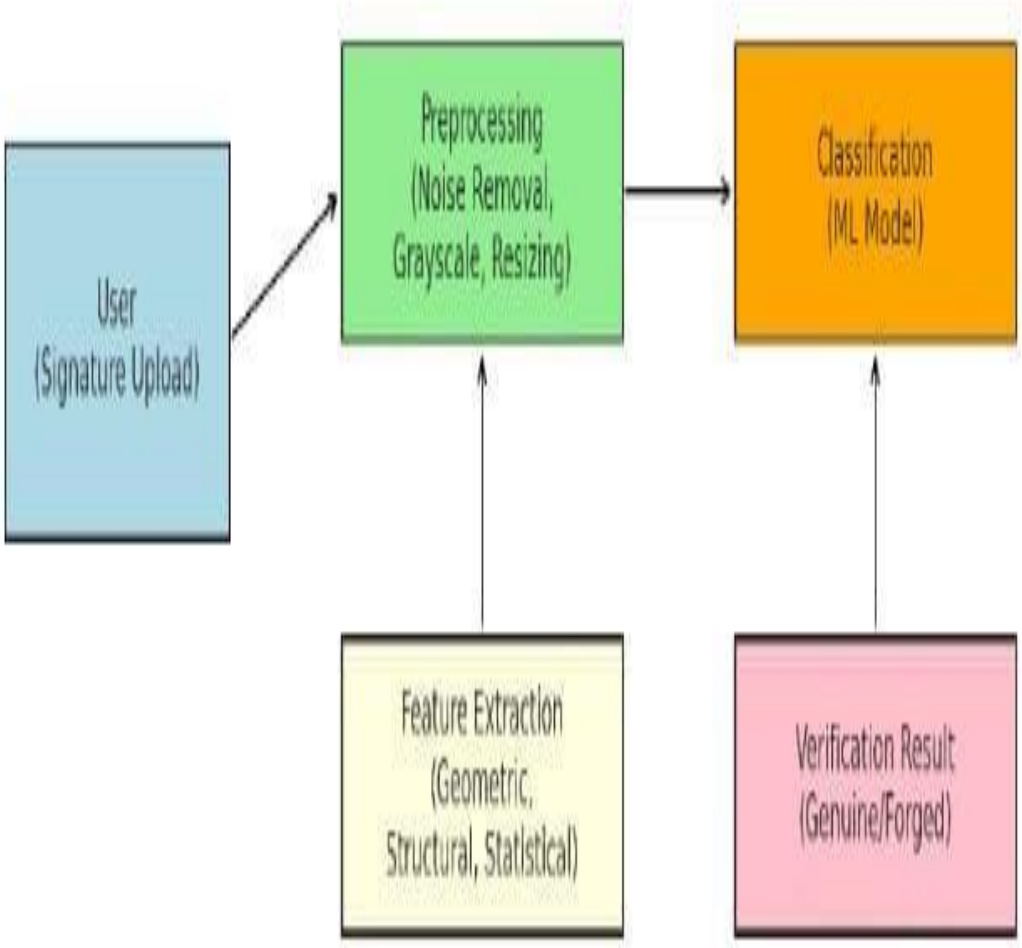
Verification module compares those features. Against stored templates. Calculates similarity.

Finally, the result module. Sends the verification outcome to the user.There are data stores too.

The signature database holds genuine signatures. User database keeps info and logs.

This level shows data moving around. Between modules. Gives a functional view. Step by step from input to output.

Level 1 Data Flow Diagram



Level 2

It exact internal processing,For signature acquisition. It captures the input digitally. Converts handwritten stuff to image or vector data.

Preprocessing has steps. Noise removal clears unwanted pixels or artifacts. Normalization adjusts size and orientation.

Smoothing and binarization make it a clean black-and-white image.Feature extraction pulls geometric features.

Like length, height, stroke curvature, angles. And dynamic ones. Pen pressure, writing speed, pen lifts.

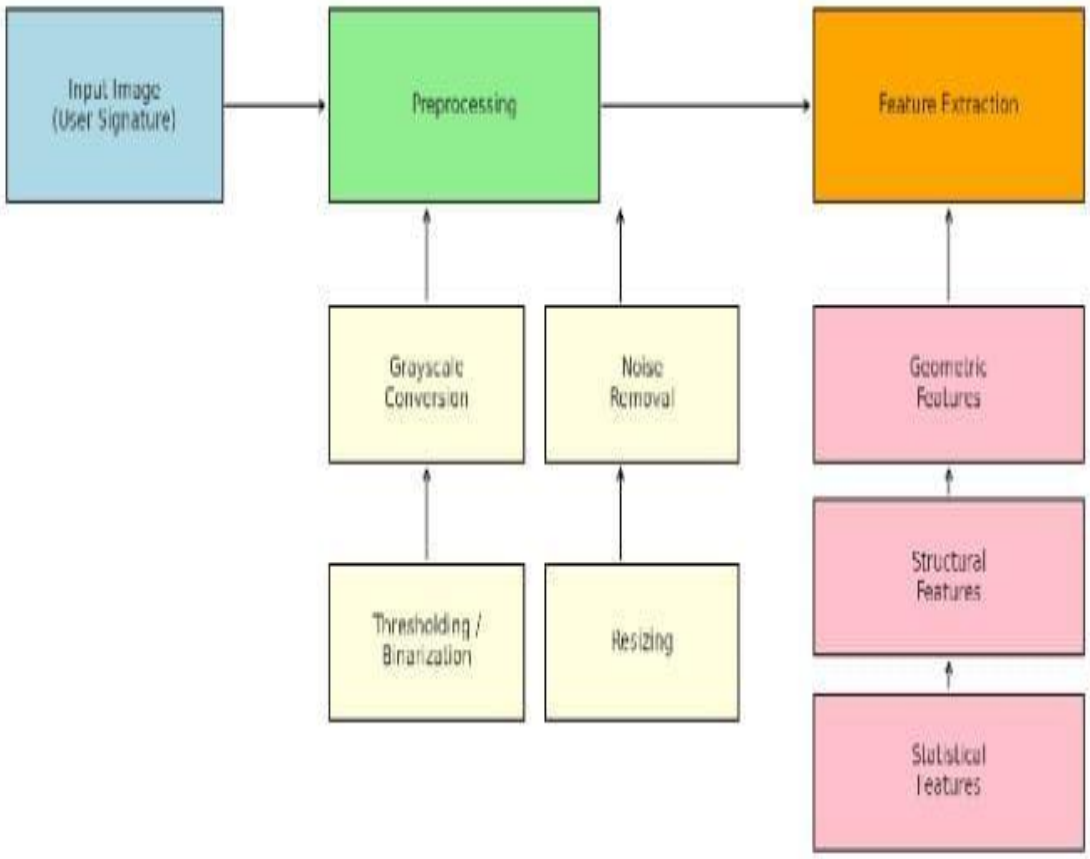
Verification compares features to stored templates. Calculates a similarity score. If the score meets or beats the threshold, its genuine.

Otherwise, forged.Result module sends feedback. Accepted or rejected to the user. And updates logs in the database. For later.

This level spells out operations in each module. Most detailed DFD. Good for implementation. And actual coding.Key points on the data flow design. Data flow shows info moving. From user to modules and databases.

Data stores are where info gets saved. For processing or later use.Processes represent actions. That transform data. From input to output.External entities are sources and recipients. Outside the system boundary.

Level 2 Data Flow Diagram (Detailed View)



2.3 Architectural Design

The architectural design lays out the whole structure for the system. It shows how the various parts work together to verify signatures. This setup makes sure everything runs efficiently. It also handles security well. Plus, it keeps things modular.

Architecture Overview:

The Smart Signature Verification System uses a modular client-server setup. Here are the main components.

User Interface Module.

- Users can submit signatures for checking through this.
- It shows results like genuine or forged.
- Login and authentication happen here too for users.

Preprocessing Module.

- This turns the signature into a standard format.
- It removes noise. It resizes the image. Converts to grayscale. Normalizes everything.

Feature Extraction Module.

- It pulls out key features from the signatures. Things like strokes. Geometric patterns. Texture. Dynamic properties if they exist.
- The signature image gets turned into a feature vector. That is for matching later.

Verification Module.

- This compares the feature vector to ones stored in the database.
- It uses ML or DL algorithms to compute similarity. Like CNNs. Or Euclidean distance. Cosine similarity too.

Decision Module.

- It decides if the signature is genuine or forged. Based on a similarity threshold.
- Results get logged. With timestamps. Into user records.

Database Module.

- Signature Database holds genuine images. And their feature vectors.
- User Records Database keeps profiles. Verification logs. History of everything.

Admin Module.

- Admins manage user accounts here.
- They upload genuine signatures to the database.
- It generates reports. Analytics on verification activities.

System Flow:

- User submits the signature via UI.
- It goes to the Preprocessing Module.
- Preprocessed version heads to Feature Extraction.
- Verification Module checks features against database ones.
- Decision Module figures out the similarity score. Sees if it hits the threshold.
- Result comes back to the user. Gets stored in the database.
- Admin Module pulls logs. Reports. Handles database updates.

Architectural Layers:

Picture the system in layers.

Presentation Layer.

- That is the UI part.
- Web interface or mobile app.
- It takes user input., Shows verification results.

Application Layer.

- The logic side.
- Handles preprocessing, Feature extraction, Verification, Decision making.
- Core algorithms for signature checking live here.

Data Layer.

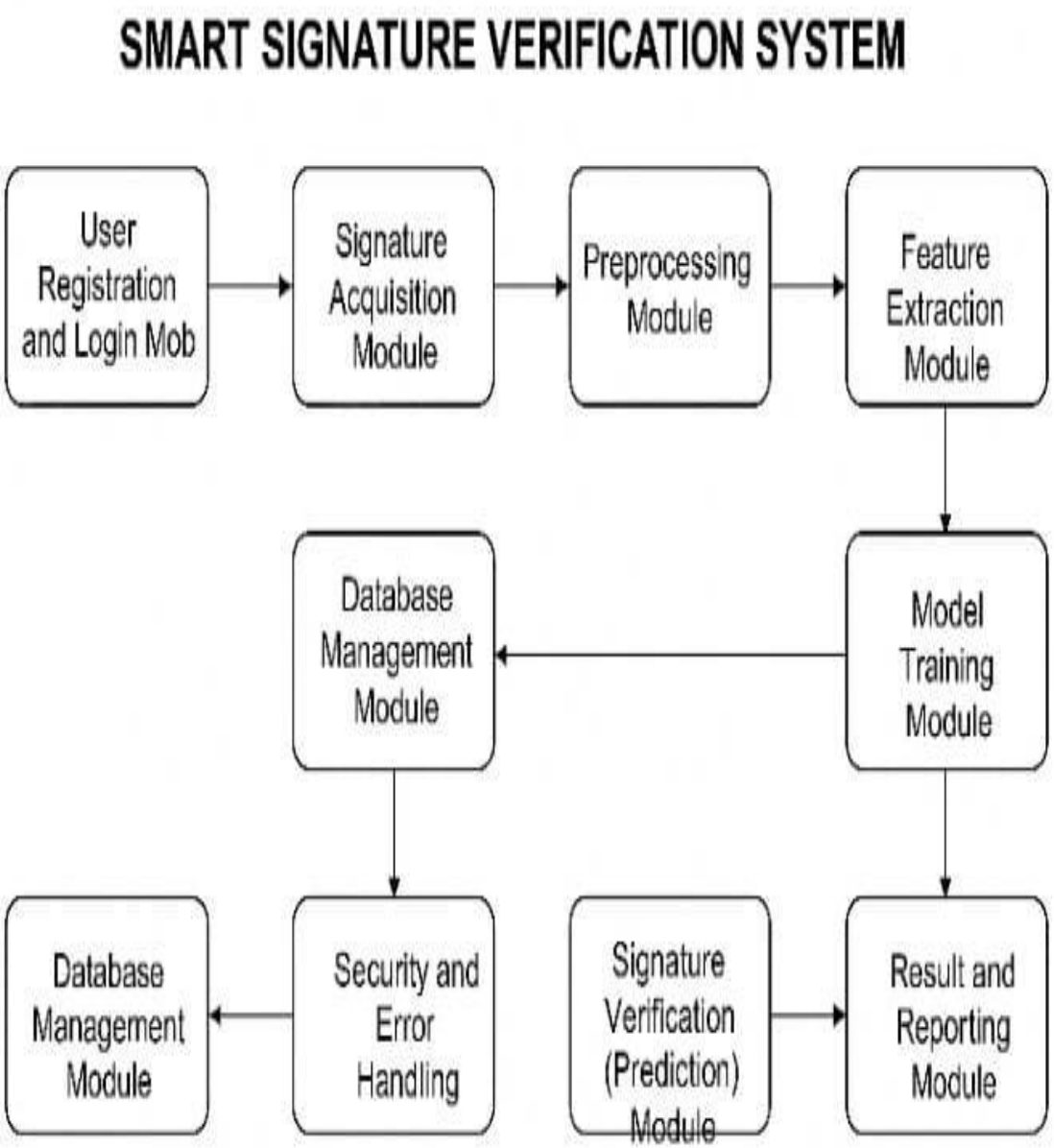
- Database stuff.
- Stores images, Feature vectors, User records.
- Secure access and retrieval for the verification process.

Technology Stack.

- Frontend UI uses HTML. CSS, JavaScript, ReactJS for web Or Flutter for mobile.
- Backend logic in Python, With Flask or Django. TensorFlow or Keras for the ML DL parts.
- Database is MySQL or PostgreSQL for user data, NoSQL like MongoDB for feature vectors.
- Security involves encryption for signature images, Secure authentication for users.
- This design supports reliable signature verification. It scales well for real world use.

Key Features:

- One thing is how modular it is. Each module deals with just one specific task. That setup makes maintenance and upgrades a lot simpler.
- Security stands out here. The databases keep sensitive data, like signature images, locked down tight.
- It scales pretty well too. You can bring on multiple users without issues. Plus, storage ramps up easy for adding new signatures.
- Verification gets accurate through ML and DL algorithms. They handle the precise matching of signatures. Admins get solid control overall. They manage users and databases. They even generate those verification reports when needed.


SMART SIGNATURE VERIFICATION SYSTEM
User Registration and Login Mob
Signature Acquisition Module
Preprocessing Module
Feature Extraction Module
Database Management Module
Model Training Module
Database Management Module
Security and Error Handling
Signature Verification (Prediction) Module
Result and Reporting Module

CHAPTER- 3

DATABASE DESIGN

3.1 Data Dictionary

Field Name	Description	Type	Notes
UserID	Unique ID for each user	Integer	Primary Key
UserName	Name of the user	Varchar(100)	Not Null
Email	User email	Varchar(150)	Unique
SignatureID	Unique ID for each signature	Integer	Primary Key
User(Fk)	Links signature to user	Integer	Foreign Key
SignatureImage	Image of signature	Blob/Varchar	Not Null
SignatureType	Genuine or Forged	Varchar(20)	Check
FeatureID	ID of extracted features	Integer	Primary Key
FeatureVector	Extracted signature features	Text/JSON	Not Null
ModelID	ID of trained ML model	Integer	Primary Key
AlgorithmType	Algorithm used(SVM/CNN)	Varchar(50)	Not Null
Accuracy	Accuracy of model(%)	Decimal(5,2)	Range 0-100
AttemptID	Verification attempt ID	Integer	Primary Key
Result	Verification Result	Varchar(20)	Not Null

3.2 Table Design

People still talk about how the database in the Smart Signature Verification System handles user info and all that signature stuff. It keeps things organized for verification results too. The setup makes sure data stays solid and you can pull up info without much hassle. Let me break down the table designs real quick.

The Users table holds details on everyone getting their signatures checked. user_id is the main key here. It just auto-increments for each new user. full_name goes in next. That one can’t be empty at all. Then email. It has to be unique so no duplicates mess things up. phone_number is there if you want it. But it’s optional, you know. created_at tracks when the user signed up. It defaults to right now basically.

Signatures table deals with the actual images and extra bits. signature_id leads the way as primary key. Auto-increment like the others. user_id links back to Users so you know who owns it. signature_image stores the pic itself. Could be a blob or just a path to the file. Signature_type tells if it’s real or fake. uploaded_at notes the upload time. Defaults to current again.

Verification Results table keeps track of every check that happens. verification_id is the unique id for each try. Primary and auto. user_id points to the Users table for the person involved. input_signature_id connects to Signatures for what’s being tested. result says genuine or forged straight up. confidence_score is extra. It shows the certainty from 0 to 100 if you have it. verified_at logs the time of the check. Defaults to now like the test.

Users Table

Field Name	Data Type	Description	Constraints
User_id	INT	Unique id for each user	Primary key, Auto Increment
Full_name	VARCHAR(100)	Full name of the user	Not Null
Email	VARCHAR(100)	Email address of the user	Unique, Not Null
Phone_number	VARCHAR(15)	Contact Number	Nullable
Created_at	DATETIME	Date of registration	CURRENT TIMESTAMP

Signatures Table

Field Name	Data Type	Description	Constraints
Signature_id	INT	Unique ID for each signature	Primary Key, Auto Increment
User_id	INT	User who owns the signature	Foreign Key->Users (user_id)
Signature_image	BLOB/VARCHAR	Signature image(or file path)	Not Null
Signature_type	ENUM	‘genuine’ or ‘forged’	Not Null
Uploaded_at	DATETIME	Date of signature uploaded	Default CURRENT_TIMES

Verification Results Table

Field Name	Data Type	Description	Constraints
Verification_id	INT	Unique ID for each verification attempt	Primary key, Auto increment
User_id	INT	User whose signature is being verified	Foreign Key-> Users(user_id)
Input_signature_id	INT	ID of the signature being verified	Foreign Key-> Signatures(sign_id)
Result	ENUM	‘genuine’ or ‘forged’	Not Null
Confidence_score	FLOAT	Confidence of verification (0-100)	Nullable
Verified_at	DATETIME	Date & time of verification	Default CURRENT_TIMESTAMP

3.3 Relationships:

Entities and relationships in this setup. First off, the User entity. Its primary key is UserID. Attributes include UserName, UserEmail, Password, UserRole, and DateCreated. Pretty straightforward for handling user accounts.

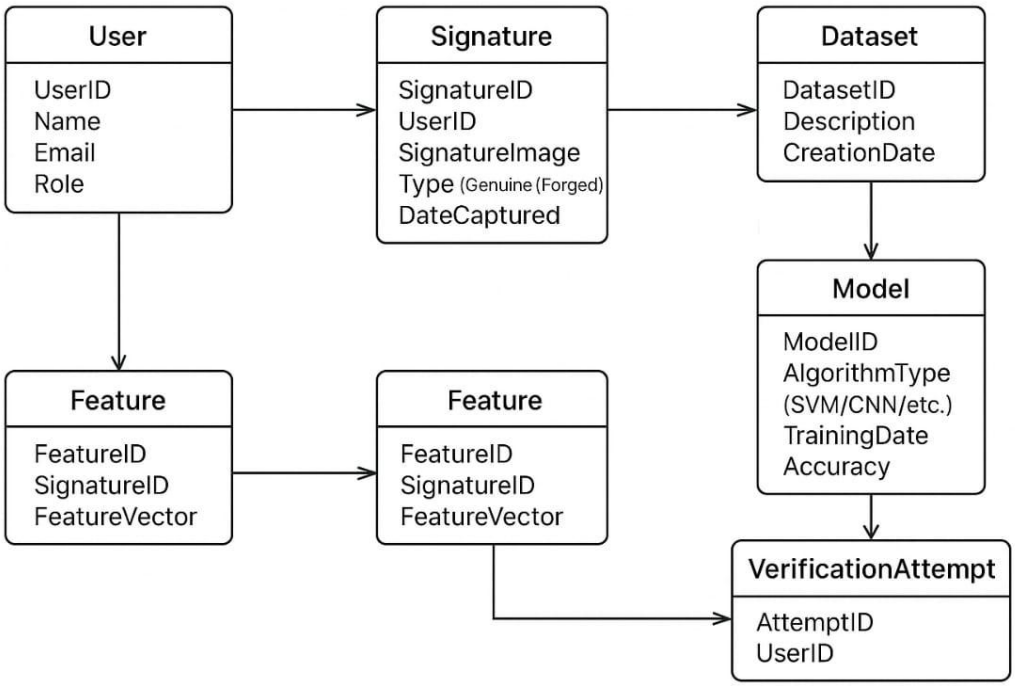
Then there's the Signature entity. Primary key is SignatureID. It has UserID as a foreign key, plus SignatureImage, PreprocessedImage, FeatureVector, and DateUploaded. The relationship here, each user can have multiple signatures. So its one-to-many from User to Signature. You know, makes sense for storing different samples.

Next, Verification. Primary key VerificationID. Attributes are SignatureID as foreign key, UserID as another foreign key, SimilarityScore, Threshold, VerificationResult, and Timestamp. Relationship wise, each signature can go through multiple verification attempts. Again, one-to-many from Signature to Verification. Basically tracks all the checks over time.

Admin entity now. Primary key AdminID. Attributes like AdminName, AdminEmail, Password, DateCreated. Simple enough for admin users.

Finally, Reports. Primary key ReportID. Has Admin ID as foreign key, ReportName, ReportContent, DateGenerated. Each admin can generate multiple reports, so one-to-many from Admin to Reports. Keeps things organized for oversight.

Relationships Diagram



CHAPTER 4

PROGRAM DESIGN

People still talk about the program design for this smart signature verification system. It lays out how everything fits together with modules and functions and the way data moves around. Thing is the whole setup stays modular so you can scale it up pretty easily and keep maintenance straightforward.

Modular design breaks the system down into a few key parts. First off there's the user interface module. That one takes care of all the user stuff you know interacting with the system. Users can upload their signature images right there. They get to see the verification results too. Plus there's registration and login options for getting in.

For admins it's a separate interface. They handle user management from there. They can check verification logs whenever. And generate reports on demand.

The functions in this module include user_register for signing up. Then user_login to get access. Upload_signature lets you send in the image. Display_result shows what came back. Admin_login for the bosses. And generate_report pulls together the data.

Next up the preprocessing module works on those raw signature images. It standardizes everything so analysis goes smooth. Functions like resize_image take the picture and adjust its size. Convert_to_grayscale turns it black and white. Remove_noise cleans up any junk. Normalize_image makes it consistent overall.

4.1 Module & Description

The Smart Signature Verification System got built with modules in mind. That keeps things clear, flexible, and not too hard to fix up later. Each module handles one part of checking signatures. It starts with getting users signed up and collecting data, then goes into pulling out features and making predictions. The whole setup splits into these connected pieces, and they all add up to how the project works.

User Registration and Login Module

This part is where everything begins for users. They register and log in safely. The idea here is to make sure only the right people can upload signatures or check them. When signing up, folks give basic info like their name, email, and some ID stuff. Plus, they hand over real signature samples that get saved in the database.

Logging in let’s them get back in with those credentials. It keeps data safe and controls who sees what. Good checks stop anyone unwanted from poking around sensitive signature info. This module really handles who the users are and keeps the whole verification spot secure.

Signature Acquisition Module

This one deals with grabbing or uploading the signature pictures. Users can scan and upload them, or draw them right there with some device. It makes sure the image comes in right, with good format and quality. That matters a lot for doing the analysis without mess ups.

It also looks over the file to see if it works, checks for okay types like JPEG or PNG, and sees if the resolution holds up. If something is bad or broken, it flags it and tells the user. By handling how data comes in, this module sets up the signatures so they work fine in the next steps.

Preprocessing Module

The preprocessing module gets the signature image ready for looking at closer. Raw pictures usually have noise, weird lighting, or backgrounds that vary. So this step cleans it up and makes it consistent.

It does a few things. First, grayscale conversion turns the color into black and white shades to make analysis simpler. Then noise removal uses smoothing or thresholds to ditch extra pixels. Binarization makes it straight black and white for pulling features clearly. Resizing and normalization tweak the size and brightness so everything matches up.

After all that, the image is clean and standard. It heads to feature extraction better. This really boosts how accurate and quick the verification turns out.

Feature Extraction Module

This module sits at the heart of it all. It pulls out the key traits from the cleaned signature image. Those traits show what makes each person's writing unique.

Things like geometric stuff, shape, size, aspect ratio come out. Texture from how pixels spread intensity. Stats too, mean, variance, skewness. They turn into number vectors. Those feed into the machine learning part.

It only sends the useful bits to the classifier. That cuts down on time but keeps accuracy high.

Model Training Module

Here the system trains the machine learning or deep learning model. It uses sets of real and fake signatures. The model picks up on patterns and features that tell real from fake.

Training goes through loops. The model tweaks itself to cut down errors. CNNs get used a lot since they handle image spotting well. After training, it can call new signatures right in tests or regular use.

It also checks validation and how it performs. That makes sure the model is solid for real time work.

Signature Verification Module

This is the part that does the real checking. User uploads a signature. It gets compared to the saved real ones in the database.

The upload goes through preprocessing and feature pull. Then those features hit the trained model. It says genuine or forged based on that.

Results show up for the user, with scores or match percentages. It runs fast and accurate. Good for on the spot verifications.

Database Management Module

This module takes care of storing and pulling data. It holds user info, real signatures, features pulled, model stuff, all safe.

It uses indexing and normalization for quick grabs and smart space use. Logs every verification for checks and records.

Encryption and limits keep it secure. No tampering. It keeps data straight and feeds all other modules smoothly.

Result and Reporting Module

After checking, this handles showing the outcome and reports. It says genuine or forged to the user. Can make reports with user details, results, accuracy numbers, time stamps.

Those reports help for looking back, keeping records, or linking to other systems like banks. It makes the whole thing more useful and open.

Security and Error Handling Module

Security matters big in biometric setups like this. The module keeps data private, whole, and safe all through. Stops unauthorized looks, changes, or bad use.

It catches errors too, like bad inputs, lost files, model fails. Shows messages to help the user. Keeps things running smooth.

Overall Module Interaction

These modules tie together for the full system. User hits the interface to sign up or check a signature. It runs through preprocessing, features, verification. Grabs data from database. Ends with report.

The modular way makes it easy to grow, change, maintain. Room for adds like phone apps, checking multiples, cloud storage.

CHAPTER 5

TESTING

Testing comes up a lot in software work. You do it to make sure the whole setup runs like it should. It checks against the specs and spits out what you expect. Thing is, the main goal stays finding bugs early and patching them. You verify functions too. Plus you confirm reliability and speed all around.

This project focuses on the smart signature verifier. Testing means seeing if it tags signatures as real or fake properly. We run all kinds of inputs at it. Valid ones work fine. Forged stuff gets caught. Blank images do not crash things. Invalid files handle without issues.

The process covers every piece separately. Each part needs to nail its job spot on. Image processing happens right. Features get extracted clean. Predictions come out accurate. Oh and the full system ties together without hitches. Results stay consistent every time.

Through all this, performance gets a real look. Accuracy too. Reliability holds up. So the smart system delivers secure checks on signatures. Fast ones. Dependable overall. It hits every project aim dead on.

Testing with screen

Testing with screens is a big part of checking the Smart Signature Verification System. It makes sure everything runs smooth, quick, and solid even when your offline. They feed it real signatures and fakes too, then watch how the outputs show up across various displays. Take the signature input screen. Users grab their signature by uploading it or capturing it fresh with a scanner, maybe a digital pen, or even a camera. The setup there lets them pick the image file easy enough and hit submit to start the verification. Thing is, it handles standard formats without a hitch, like jpg files, png ones, or bmp. You see a simple layout, something along the lines of a box saying Upload Signature, with buttons for Choose File and Submit right below.

Next comes the preprocessing screen. Once that image is in, it gets cleaned up through steps like turning it grayscale, stripping out noise, and resizing to fit. Users get to peek at the original next to the processed version, just to confirm the system standardized things right. Again, the display splits it out, one part for the Original Signature with the image shown, and below that the Preprocessed Signature with its display. This way, you can visually check if preprocessing did its job properly.

Then there's the feature extraction screen. After that prep work, the system pulls out geometric features, structural ones, and statistical bits from the signature. It even shows the feature vector pulled together, like numbers such as 0.45, 0.78, 0.32, 0.89, and so on, mainly for debugging or double-checking. What this does is make sure each signature gets a unique breakdown, setting it up for solid classification later. Finally, the verification and result screen wraps it up. Those extracted features go straight into the trained machine learning model. The system crunches it and spits out if the signature is Genuine or Forged, displaying the outcome clear as day.

The layout might say Signature Verification at the top, then Result: Genuine or whatever it is. Users get that instant feedback right there, which keeps things real-time and straightforward. From testing, genuine signatures come through with high accuracy most times. Forged ones get caught reliably, keeping the false acceptance rate low. And since it all runs offline, it’s secure and practical in spots with no internet or just spotty access. Users follow the whole flow across these screens, starting from input, through preprocessing and feature pulls, right to the final output.

5.1 Types of Testing:

- Unit Testing
- Integration Testing
- System Testing
- Performance Testing
- Accuracy Testing
- Security Testing
- User Acceptance Testing

1. Unit Testing

The idea here is to check out each part of your system on its own. Make sure everything in their works right by itself.

In your project, you got a few modules to look at.

Take the signature preprocessing module. It needs to resize those signature images properly. Also denoise them and normalize too.

Then there's the feature extraction module. You want to make sure it pulls out the right stuff. Like stroke length or pressure levels. Or even the pixel patterns.

Model training module comes next. Just ensure it runs through training without any glitches. And that the loss function ends up at a decent number.

Prediction module is key. Test it to see if it calls signatures authentic or not correctly. Use inputs you already know.

Example:

Feed in a blank image. The system should not crash on that.

Try a messed up image that's corrupted. It ought to throw a good error message.

Pull features from a signature you know well. Those features should line up with what you expect.

2. Integration Testing

This part tests how the modules talk to each other. Basically make sure they play nice together.

Scenarios to cover.

Run a signature through preprocessing first. Then to feature extraction. Finally to model prediction. It all has to flow smooth.

Database integration too. Check that it grabs the stored real signatures right for checking against.

Example:
You submit a signature. System preprocesses it. Extracts features. Compares to what's stored. Then spits out if It's genuine or forged. Correctly of course.

3. System Testing

Here you test the whole thing end to end. Ensure it hits all the requirements laid out.

For the smart signature verification system, scenarios include.

Verifying real signatures from folk's whole signed up. Spotting forged ones accurately.

Dealing with a bunch of users in the database.

Keeping response times solid for verification asks.

Example:

Run a batch of 50 signatures. Mix genuine and forged. Check the accuracy percentage at the end.

See if it handles bad inputs nice. Like files that aren't images.

4. Performance Testing

Purpose: It is to see if the system stays quick and doesn't bog down under pressure.

Key tests:

Response time: Time how long it takes to verify one signature.

Scalability: See what happens when lots of signatures hit at once.

Memory usage: Watch how much ram it chews during preprocessing and pulling features.

5. Accuracy Testing

This evaluates how reliable the system is at telling real from fake signatures.

Metrics to track. True positive rate. That's genuine ones caught right.

True negative rate. Forged ones turned away correctly.

False positive and false negative rates too.

Overall accuracy. Figure it as (TP plus TN) divided by total signatures tested.

Example.

You got a test set with 100 genuine and 50 forged.

System nails 95 genuine and 45 forged. So accuracy is (95 plus 45) over 150. Comes to 93.33 percent.

6. Security Testing

Aim is to keep the system solid. Stop any tampering or breaks.

Tests to do.

See if outsiders can snag stored signatures.

Make sure model parameters and training data stay locked down.

Test input checks to block bad or sneaky inputs.

7. User Acceptance Testing, or UAT

This confirms the system does what users actually need.

Scenarios for end users.

Admins or staff upload signatures. Get back instant verification.

It should be straightforward to use. Clear output like genuine or forged.

Reports and logs need to be spot on. Easy to get to.

8. Testing Tools :

Pythons unit test or py test work great. For unit and integration stuff.

OpenCV or PIL handle image processing tests.

TensorFlow or PyTorch for checking model predictions and training.

JMeter or Locust do performance testing.

Then manual testing covers UAT and security check.

CHAPTER 6
COCLUSION

People still talk about this Smart Signature Verification System project. It turned out pretty well. They built it to check handwritten signatures in a secure way. No need for internet at all. It uses image processing stuff. Machine learning too. And feature extraction to spot real ones from fakes. Thing is, this shows offline verification works just fine. It fits real life uses like banking or legal papers. Even school stuff where you need to confirm who someone is. The whole thing got built in modules. Each one handles its own job. Data collection pulls together a bunch of real and fake signatures. Preprocessing cleans up the images. Makes them ready for checking. Feature extraction pulls out shapes and patterns. Stats on the signatures too. That builds a solid set for the classifier. Classification runs on things like SVM or Random Forest. Neural networks even. It tells genuine from forged pretty accurately. Then verification outputs the results. User interface is simple. You can see the steps along the way. Testing went smooth. High accuracy on real signatures. Not many false positives for fakes. They checked units, integration, the whole system. Performance and accuracy too. Everything held up. Offline means no network risks. Secure for places like banks. Legal spots that need it tight. Key part here is how it mixes accuracy with easy use. Users upload an image. Get results quick. No tech know-how required. The interface shows verification right away. Let’s you peek at preprocessing or features. Builds trust that way. System scales up easy. You can add better ML later. More data for different languages or scripts. Even dynamic stuff like pressure and speed in strokes. For sharper checks. This offline signature system feels practical. Reliable too. Speeds up checks over manual ones. Cuts dependency there. High security and accuracy come with it. Project spotlights machine learning and image processing in biometrics. Sets up ground for more offline work. Valuable for spots needing secure automated verification. Blends tech, efficiency, usability just right.

CHAPTER 7

REFERENCE

Books

1. Gonzalez, R.C., & Wood, R.E. (2018) / Digital Image Processing (4^{th} Edition) / Pearson Education.

2. Jain, A.K., Flynn, P., & Ross, A.A. (2007) / Handbook of Biometrics / Springer.

3. Bishop, C.M. (2006) / Pattern Recognition and Machine Learning / Springer.

4. Duda, R.O., Hart, P.E., & Stork, D.G. (2001) / Pattern Classification (2^{nd} Edition) / Wiley-Interscience.

5. Kisku, D,R., Rattani, A., & Bicego, M. (2016) / Biometric Authentication: A Machine Learning Approach / CRC Press.

6. Bishop, Y. & Nabendu, P. (2019) / Machine Learning for Beginners : A Practical Guide to Implementing Machine Learning Algorithms / Apress .

Website

1. OpenCV Documentation – Image Processing and Computer Vision Library / https://docs.opencv.org

2. Scikit-learn Documentation – Machine Learning in Python / https://scikit-learn.org

3. ResearchGate – Signature Verification Related Research Papers / https://www.researchgate.net

4. IEEE Xplore Digital Library – Biometric Authentication Systems / https://ieeexplore.ieee.org

5. Kaggle – Signature Verification Datasets and Competitions / https://www.kaggle.com

6. Towards Data Science – Articles on Machine Learning and Image Processing https://towardsdatascience.com

CHAPTER 8

APPENDIX

8.1 SOURCE CODE:

preprocessing.py

```
import cv2

import numpy as np

import os


def preprocess_image(image_path, output_size=(200, 100)):

    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    if img is None:

        print(f"Error loading image: {image_path}")

        return None

    img = cv2.resize(img, output_size)

    img = cv2.GaussianBlur(img, (5, 5), 0)

    _, img = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV +
cv2.THRESH_OTSU)
```

```
    return img
if _name_ == "_main_":
    input_folders = ["dataset/genuine", "dataset/forged"]
    output_folders = ["preprocessed/genuine", "preprocessed/forged"]

    for in_folder, out_folder in zip(input_folders, output_folders):
        os.makedirs(out_folder, exist_ok=True)
        for filename in os.listdir(in_folder):
            if filename.endswith(".png") or filename.endswith(".jpg"):
                img_path = os.path.join(in_folder, filename)
                processed = preprocess_image(img_path)

  if processed is not None:
                    cv2.imwrite(os.path.join(out_folder, filename), processed)

    print("✅ Preprocessing completed successfully!")
```

augument_genuine.py:

```
# augment_genuine.py

import os

import cv2

import numpy as np


GENUINE_PATH = "preprocessed/genuine"

AUG_PATH = "preprocessed/genuine_aug"

os.makedirs(AUG_PATH, exist_ok=True)


def augment_image(img):

    augmented = []

    augmented.append(img)  # original
```

```
    # Small rotations
    for angle in [-5, 5]:
        M = cv2.getRotationMatrix2D((img.shape[1]//2, img.shape[0]//2), angle, 1)
        rotated = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]), borderMode=cv2.BORDER_REPLICATE)
        augmented.append(rotated)

    # Small shifts
    for dx, dy in [(-2,0), (2,0), (0,-2), (0,2)]:
        M = np.float32([[1,0,dx],[0,1,dy]])
        shifted = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]), borderMode=cv2.BORDER_REPLICATE)
        augmented.append(shifted)

    return augmented
```

```
# Process all genuine images
for file in os.listdir(GENUINE_PATH):
    if file.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(GENUINE_PATH, file)
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        aug_images = augment_image(img)
        base_name = os.path.splitext(file)[0]
        for idx, aug in enumerate(aug_images):
            save_path = os.path.join(AUG_PATH, f"{base_name}_aug{idx}.jpg")
            cv2.imwrite(save_path, aug)

print(f"✅ Genuine augmentation completed! Saved in {AUG_PATH}")
```

augument_forged.py:

```
# augment_forged.py

import os

import cv2

import numpy as np


# ---------- PATHS ----------

FORGED_PATH = "preprocessed/forged"       # folder with existing preprocessed forged images

AUG_PATH = "preprocessed/forged_aug"     # folder to save augmented images

os.makedirs(AUG_PATH, exist_ok=True)


# ---------- AUGMENTATION FUNCTION ----------

def augment_image(img):

    augmented = []
```

```
    # Original
    augmented.append(img)

    # Rotation ±5° and ±10°
    for angle in [-10, -5, 5, 10]:
        M = cv2.getRotationMatrix2D((img.shape[1]//2, img.shape[0]//2), angle,
1)
        rotated = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]),
borderMode=cv2.BORDER_REPLICATE)
        augmented.append(rotated)

    # Slight shift
    for dx, dy in [(-5,0), (5,0), (0,-5), (0,5)]:
        M = np.float32([[1,0,dx],[0,1,dy]])
        shifted = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]),
borderMode=cv2.BORDER_REPLICATE)
        augmented.append(shifted)

    # Add Gaussian noise
```

```
    noise = np.random.normal(0, 10, img.shape).astype(np.uint8)
    noisy = cv2.add(img, noise)
    augmented.append(noisy)

    return augmented

# ---------- PROCESS ----------
for file in os.listdir(FORGED_PATH):
    if file.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(FORGED_PATH, file)
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        aug_images = augment_image(img)

        base_name = os.path.splitext(file)[0]
        for idx, aug in enumerate(aug_images):
            save_path = os.path.join(AUG_PATH, f"{base_name}_aug{idx}.jpg")
            cv2.imwrite(save_path, aug)

print(f"✅ Augmentation completed! Augmented images saved in {AUG_PATH}")
```

train.py:

```
# train.py

import os

import cv2

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.utils.class_weight import compute_class_weight

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Dropout

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.optimizers import Adam


# ---------- PATHS ----------

PREPROCESSED_PATH = "preprocessed"

MODEL_PATH = "model/signature_model_v2.h5"

IMAGE_SIZE = (100, 200)
```

```
CHANNELS = 1


# ---------- LOAD DATA ----------
def load_data():
    images = []
    labels = []


    for label, folder in enumerate(["forged", "genuine"]):  # 0=forged, 1=genuine
        folder_path = os.path.join(PREPROCESSED_PATH, folder)
        for file in os.listdir(folder_path):
            if file.lower().endswith(('.png', '.jpg', '.jpeg')):
                img = cv2.imread(os.path.join(folder_path, file),
cv2.IMREAD_GRAYSCALE)
                img = cv2.resize(img, IMAGE_SIZE)
                img = img / 255.0
                images.append(img)
                labels.append(label)


    X = np.array(images).reshape(-1, IMAGE_SIZE[0], IMAGE_SIZE[1],
CHANNELS)
```

```
    y = np.array(labels)
    return X, y

# ---------- LOAD AND SPLIT ----------
X, y = load_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)

# ---------- CLASS WEIGHTS ----------
class_weights = compute_class_weight(class_weight='balanced',
classes=np.unique(y_train), y=y_train)
class_weights = dict(enumerate(class_weights))
print("✅ Class Weights:", class_weights)
# ---------- MODEL ----------
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(IMAGE_SIZE[0],
IMAGE_SIZE[1], CHANNELS)),
    MaxPooling2D(2,2),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(2,2),
```

```
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(2, activation='softmax')
])
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# ---------- TRAIN ----------
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=20,
                    batch_size=16,
                    class_weight=class_weights)

# ---------- SAVE ----------
model.save(MODEL_PATH)
print(f"✅ Model saved successfully at: {MODEL_PATH}")
```

test.py:

```
# test.py

import os

import cv2

import numpy as np

from tensorflow.keras.models import load_model


# ---------- PATH SETTINGS ----------

MODEL_PATH = "model/signature_model_v2.h5"

TEST_FOLDER = "test_signatures"   # Folder where test images are kept

IMAGE_SIZE = (100, 200)          # same as training

CHANNELS = 1                     # grayscale


# ---------- LOAD TRAINED MODEL ----------

model = load_model(MODEL_PATH)

print("✅ Model loaded successfully!")
```

```
# ---------- PREPROCESS FUNCTION ----------

def preprocess_image(image_path):

    """Preprocess test signature image same as training"""

    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    if img is None:

        raise FileNotFoundError(f"Image not found: {image_path}")

    img = cv2.resize(img, IMAGE_SIZE)

    _, img = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV +
cv2.THRESH_OTSU)

    img_norm = img / 255.0

    return img_norm.reshape(1, IMAGE_SIZE[0], IMAGE_SIZE[1],
CHANNELS)


# ---------- TESTING FUNCTION ----------

def test_signatures():

    threshold = 0.7   # decision boundary (you can adjust 0.6–0.8)

    print("\n🔍 Starting Signature Verification...\n")


    # Loop through all test images
```

```
for filename in os.listdir(TEST_FOLDER):
    if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(TEST_FOLDER, filename)
        processed = preprocess_image(img_path)

        # Predict probabilities
        prediction = model.predict(processed)[0]
        prob_forged, prob_genuine = prediction[0], prediction[1]

        # Apply threshold
        if prob_genuine >= threshold:
            result = f"Genuine ✅ (p={prob_genuine:.3f})"
        else:
            result = f"Forged ❌ (p={prob_genuine:.3f})"

        print(f"{filename} → {result}")

print("\n✅ Testing completed!")
```

```
# ---------- MAIN ----------

if _name_ ==”_main_”

test_signatures()
```

8.2 O/P Screen

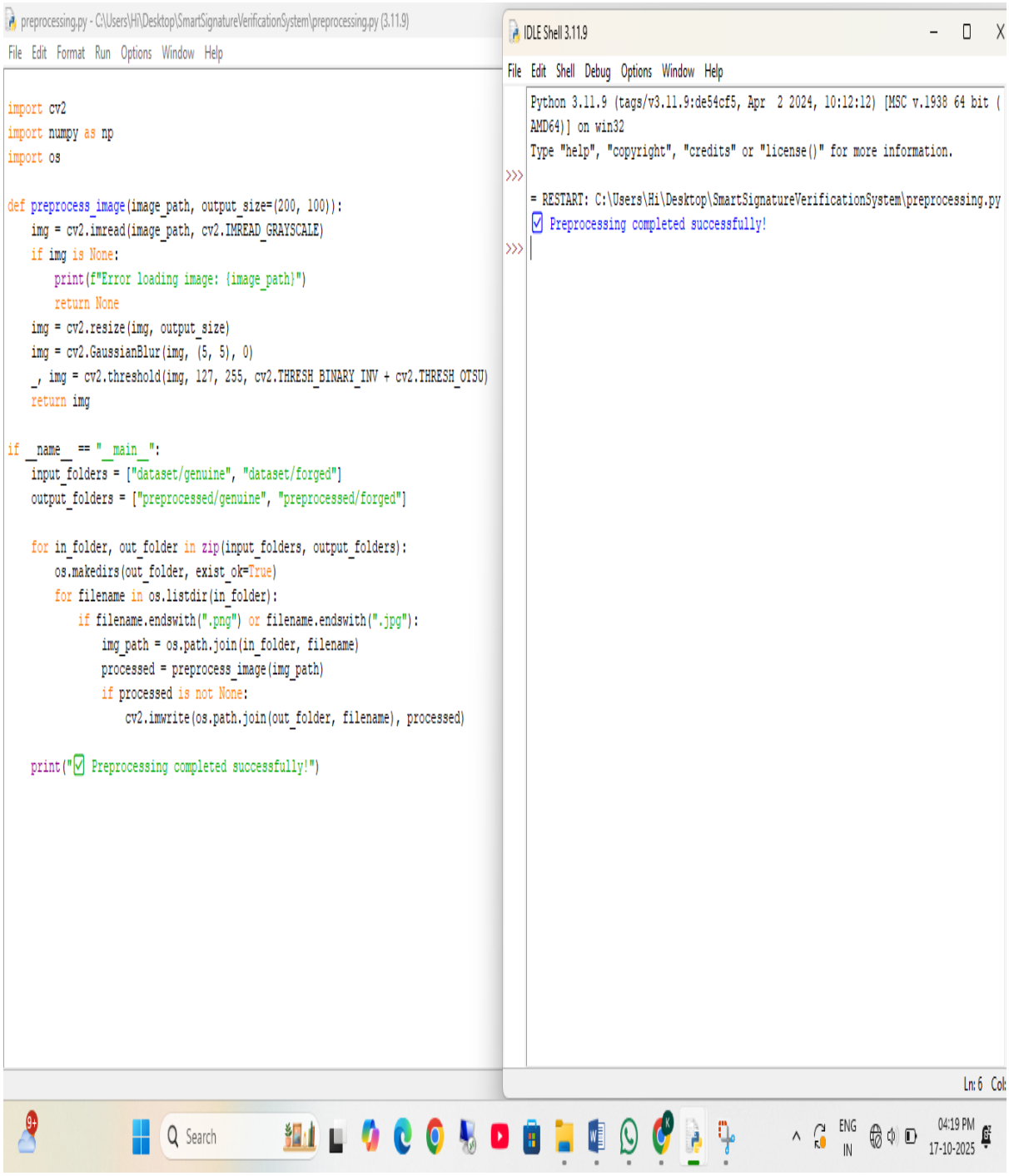


Figure 8.2.1 preprocessing.py

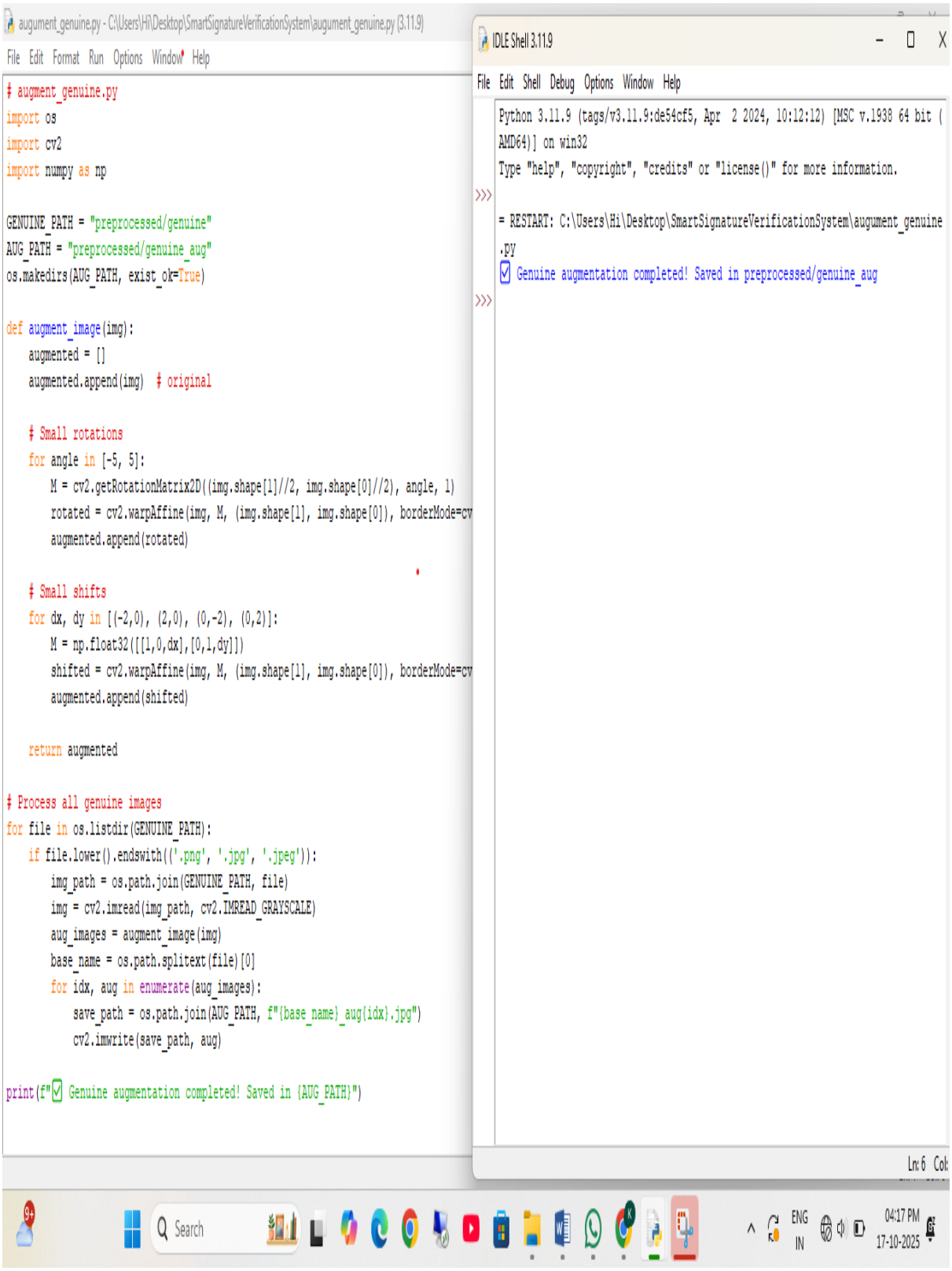


Figure 8.2.2 augument_genuine.py

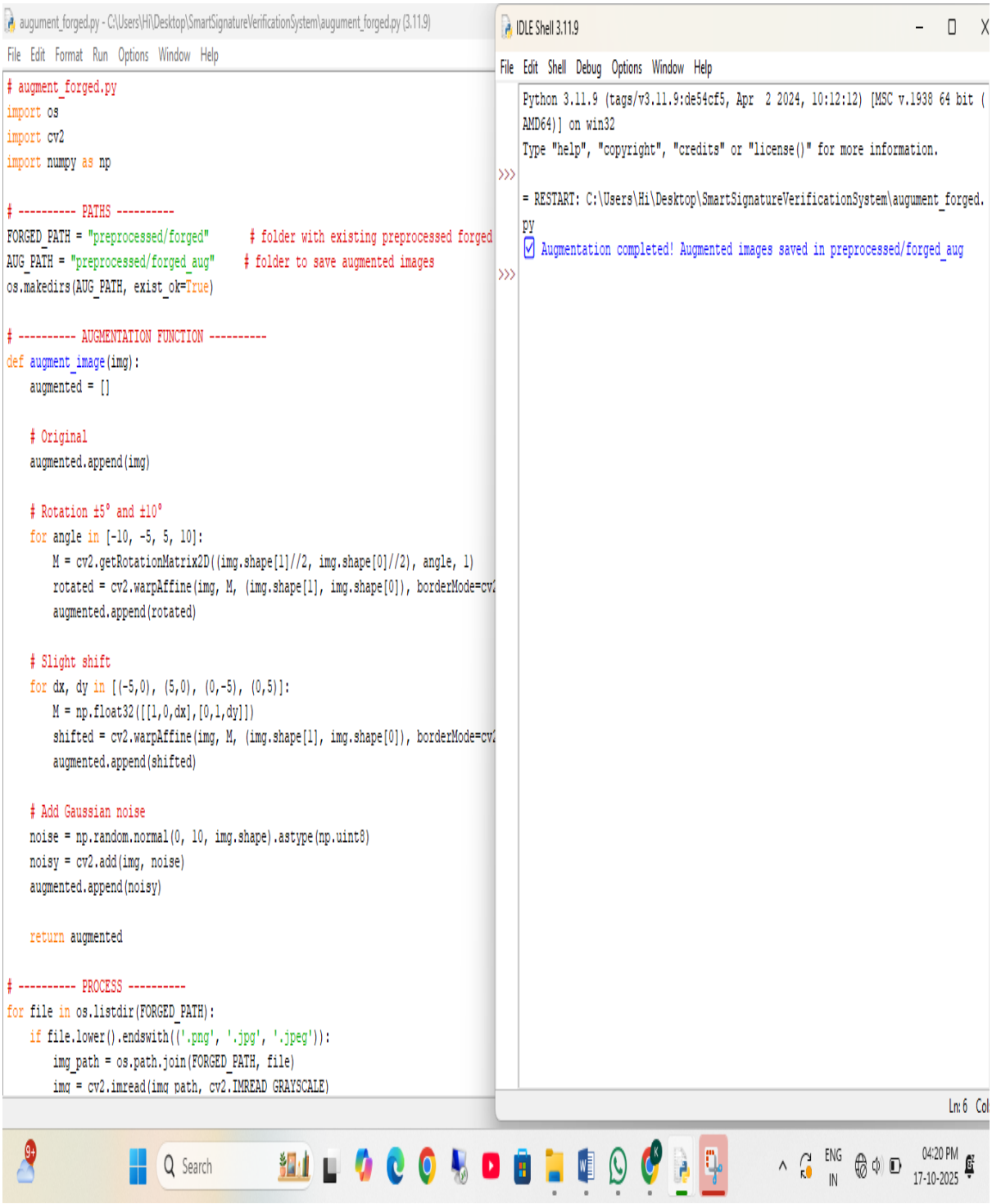


Figure 8.2.3 augument_forged.py

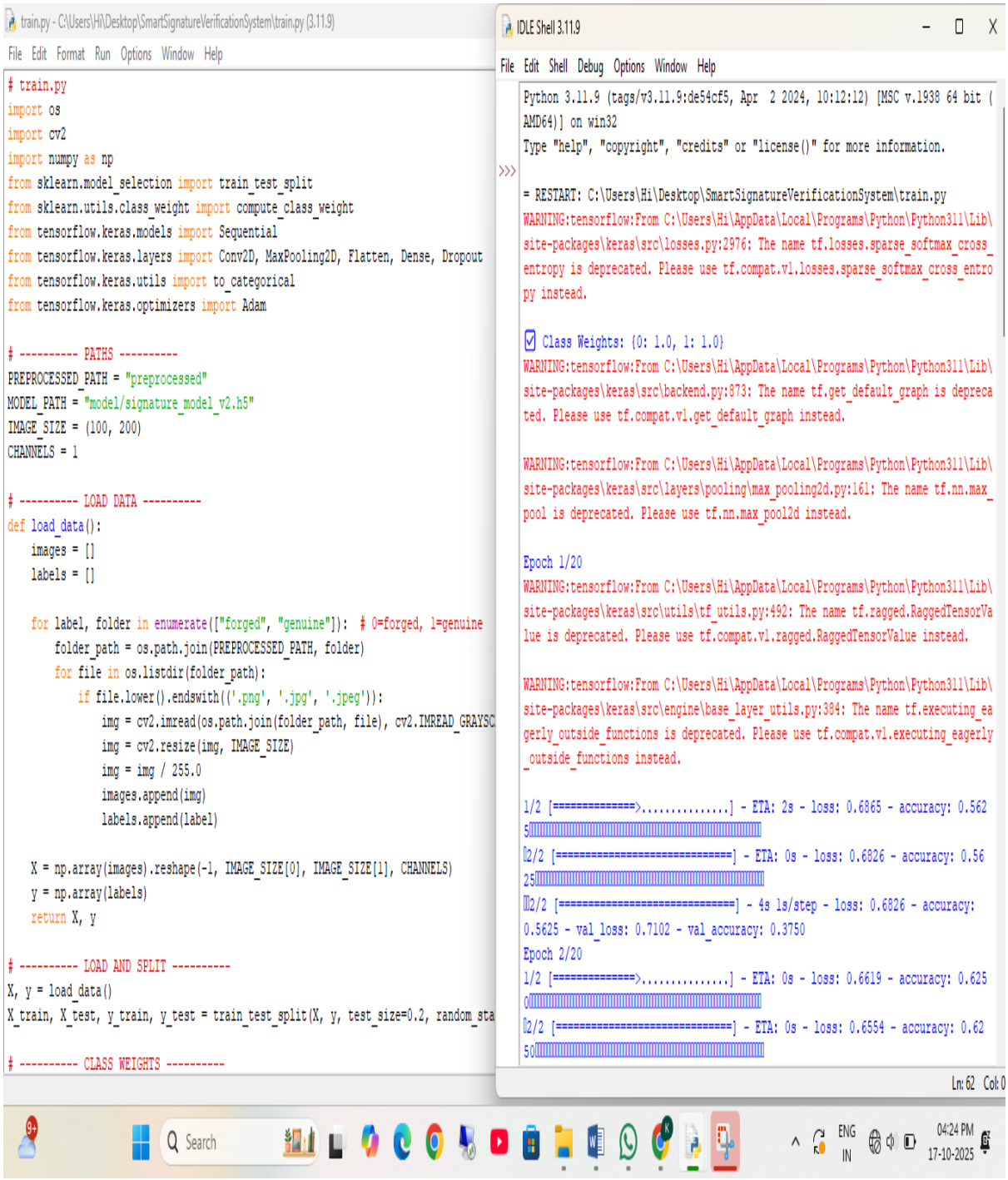


Figure 8.2.4 train.py

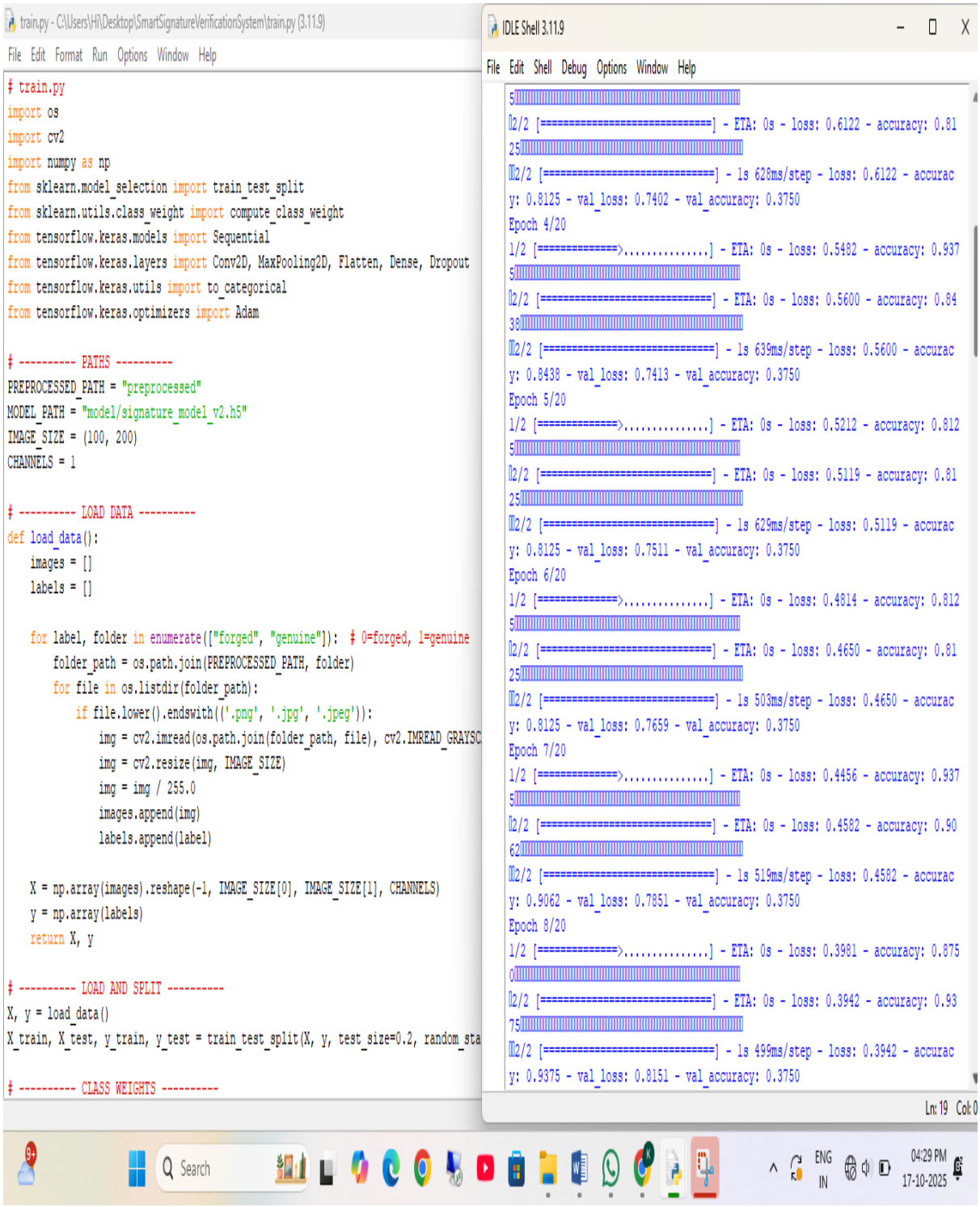


Figure 8.2.5 train.py

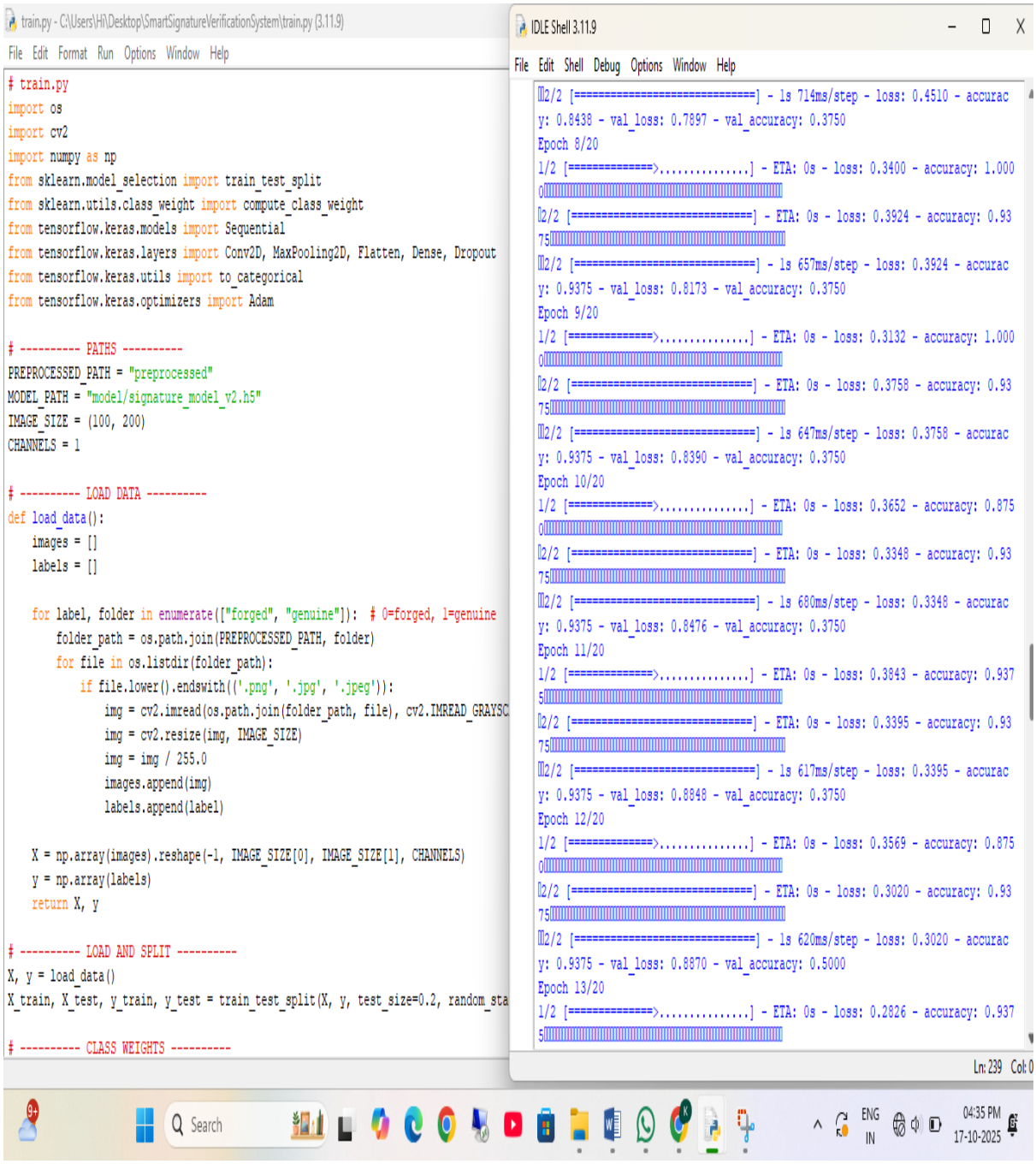


Figure 8.2.6 train.py

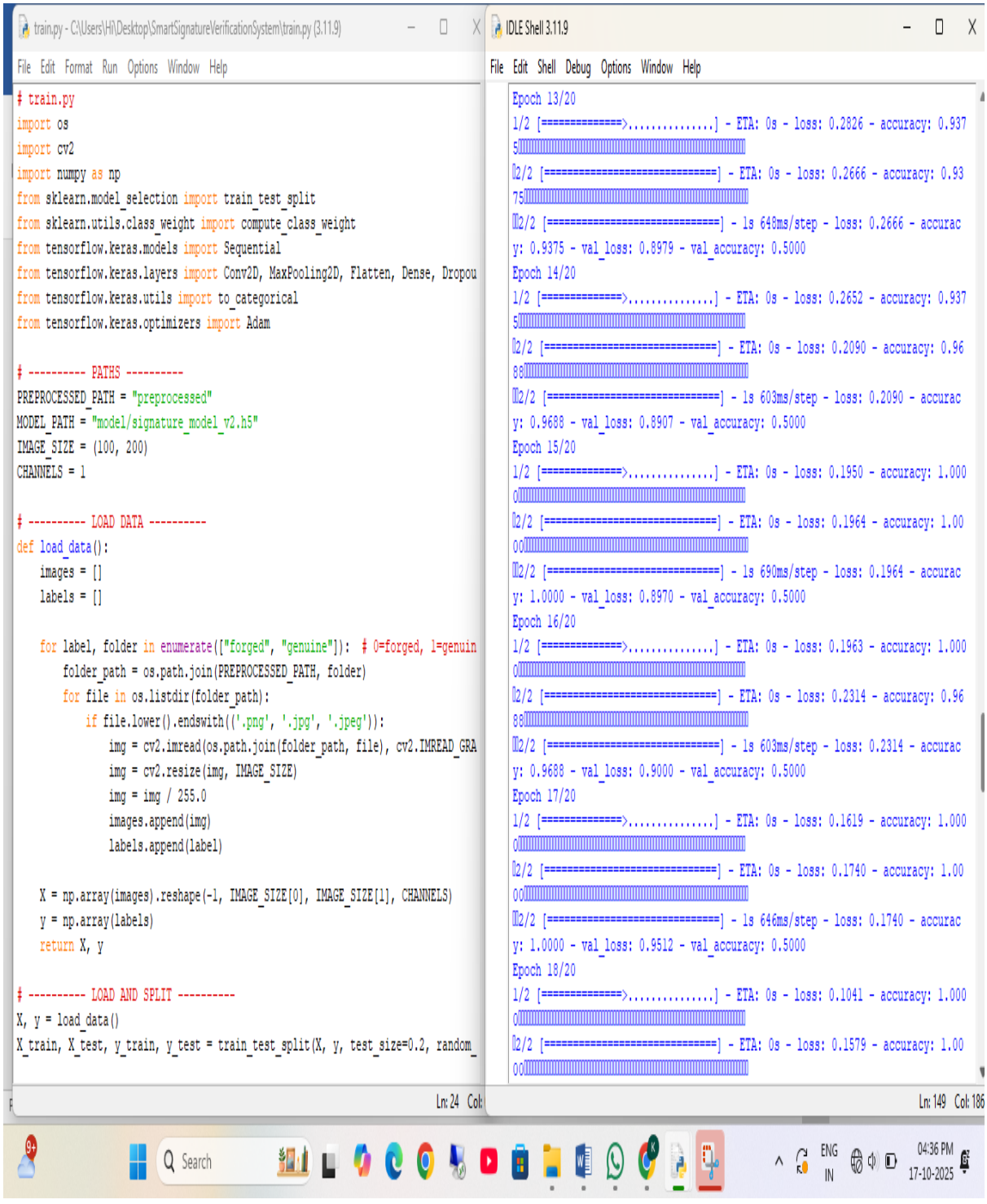


Figure 8.2.7 train.py

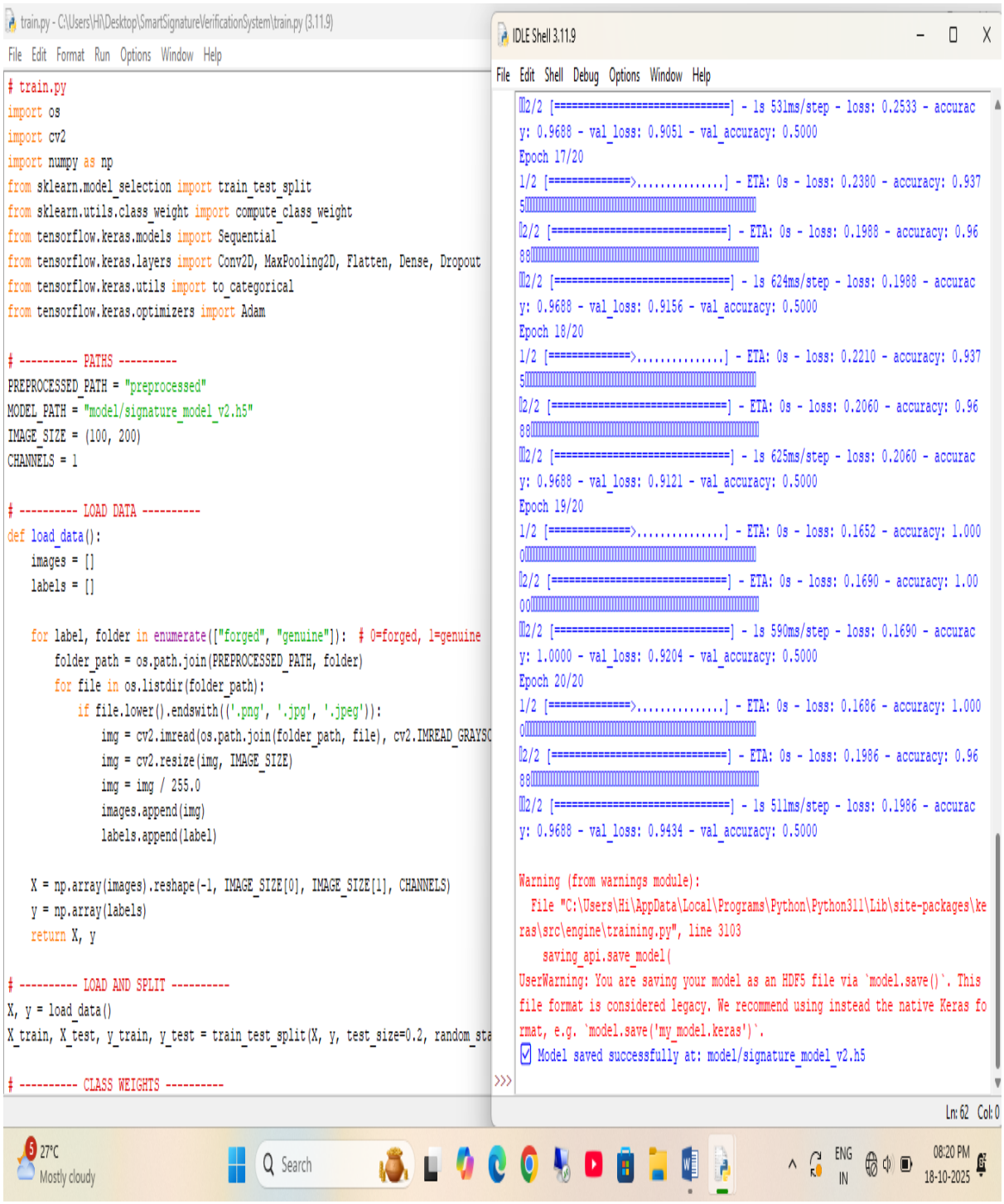
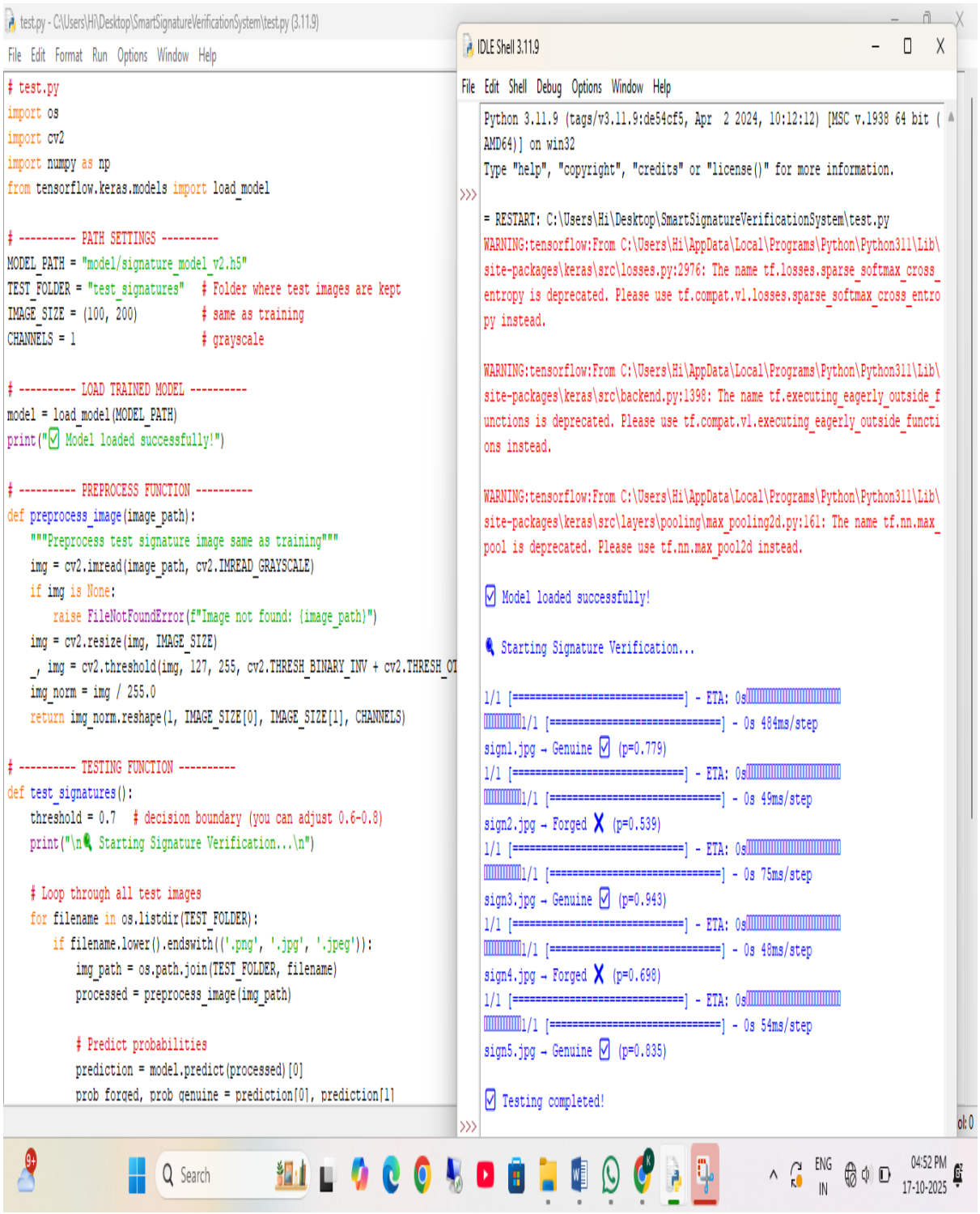


Figure 8.2.8 train.py


test.py - C:\Users\Hi\Desktop\SmartSignatureVerificationSystem\test.py (3.11.9)
File Edit Format Run Options Window Help
# test.py
import os
import cv2
import numpy as np
from tensorflow.keras.models import load_model
# ---------- PATH SETTINGS ----------
MODEL_PATH = "model/signature_model_v2.h5"
TEST_FOLDER = "test_signatures" # Folder where test images are kept
IMAGE_SIZE = (100, 200)   # same as training
CHANNELS = 1   # grayscale
# ---------- LOAD TRAINED MODEL ----------
model = load_model(MODEL_PATH)
print("✅ Model loaded successfully!")
# ---------- PREPROCESS FUNCTION ----------
def preprocess_image(image_path):
"""Preprocess test signature image same as training"""
img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
if img is None:
raise FileNotFoundError(f"Image not found: {image_path}")
img = cv2.resize(img, IMAGE_SIZE)
_, img = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OT
img_norm = img / 255.0
return img_norm.reshape(1, IMAGE_SIZE[0], IMAGE_SIZE[1], CHANNELS)
# ---------- TESTING FUNCTION ----------
def test_signatures():
threshold = 0.7 # decision boundary (you can adjust 0.6-0.8)
print("\n🔍 Starting Signature Verification...\n")
# Loop through all test images
for filename in os.listdir(TEST_FOLDER):
if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
img_path = os.path.join(TEST_FOLDER, filename)
processed = preprocess_image(img_path)
# Predict probabilities
prediction = model.predict(processed)[0]
prob_forged, prob_genuine = prediction[0], prediction[1]
IDLE Shell 3.11.9
File Edit Shell Debug Options Window Help
Python 3.11.9 (tags/v3.11.9:de54cf5, Apr 2 2024, 10:12:12) [MSC v.1938 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\Hi\Desktop\SmartSignatureVerificationSystem\test.py
WARNING:tensorflow:From C:\Users\Hi\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
WARNING:tensorflow:From C:\Users\Hi\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.
WARNING:tensorflow:From C:\Users\Hi\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\pooling\max_pooling2d.py:161: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.
✅ Model loaded successfully!
🔍 Starting Signature Verification...
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 484ms/step
sign1.jpg → Genuine ✅ (p=0.779)
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 49ms/step
sign2.jpg → Forged ❌ (p=0.539)
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 75ms/step
sign3.jpg → Genuine ✅ (p=0.943)
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 48ms/step
sign4.jpg → Forged ❌ (p=0.698)
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 54ms/step
sign5.jpg → Genuine ✅ (p=0.835)
✅ Testing completed!
Search
ENG IN
04:52 PM 17-10-2025

8.2.9 test.py