**ABSTRACT**

This work presents a desktop application for the automated differential diagnosis of erythemato‑squamous skin diseases, integrating a complete machine‑learning pipeline within a user‑friendly Tkinter GUI and secured by Redis‑based authentication. Erythemato‑squamous disorders such as psoriasis, seborrheic dermatitis, lichen planus, pityriasis rosea, chronic dermatitis, and pityriasis rubra pilaris often exhibit overlapping clinical and microscopic features, leading to diagnostic delays and inter‑observer variability. To address these challenges, this research developed a role‑based system: Admin users perform data ingestion, preprocessing, and model training, while end users execute batch predictions on new cases. Three classifiers such as Decision Tree Classifier (DTC) model, linear‑kernel Support Vector Machine (SVM) classifier, and Deep Learning (DL) classifier are trained sequentially. Evaluation on held‑out test data reveals distinct strengths: the DTC model achieves 65% accuracy and a macro‑averaged F₁‑score of 57.9%, demonstrating interpretability but struggling with minority classes. The SVM classifier yields 98.5% accuracy and a macro F₁ of 98.7%, indicating strong linear separability of the six categories. The proposed DL attains perfect classification (100% accuracy, precision, recall, and F₁‑score), capturing residual nonlinear relationships. Visualization components—including class‑distribution bar charts, confusion‑matrix heatmaps, and comparative performance graphs—enhance transparency and facilitate model interpretation. In addition, role‑based access control also uses Redis hashes and SHA‑256 password hashing to distinguish Admin from User operations, ensuring that only Admins can retrain models. Users can upload new datasets to obtain immediate predictions, with both raw feature values and predicted labels displayed in the GUI. By encapsulating the entire machine‑learning workflow, from data cleaning through inference in a single, click‑driven interface, this application streamlines diagnostic workflows and reduces reliance on manual scripting. It offers a robust proof of concept for rapid, reproducible, and standardized decision support in dermatological diagnosis.

**CHAPTER 1**

**INTRODUCTION**

The integumentary system, particularly the skin as the largest organ of the human body, serves as a crucial barrier against external adversities. Dermatological conditions are a prevalent cause of medical consultations, accounting for 8–36 % of such interactions, with erythemato-squamous disorders (ESD)—distinguished by symptoms like redness, scaling, and superficial lesions—constituting a common category in these cases. These disorders, which include conditions such as psoriasis, seborrheic dermatitis, and lichen planus, arise due to an interplay of environmental and genetic elements and are noted for their overlapping clinical presentations. The diagnostic intricacies are exacerbated by the nuanced differences in clinical manifestations, particularly in erythema and scaling, and the shared features in clinical and histopathological profiles, leading to challenges in differential diagnosis, therapeutic decisions, and predicting patient outcomes. The scarcity of dermatologists worldwide necessitates the reliance on primary care providers and nurse practitioners for the diagnosis of skin conditions, which involves analyzing numerous clinical and histopathological factors, a situation that increases the likelihood of diagnostic inaccuracies and underscores the critical need for improved and widely available diagnostic tools in dermatology.

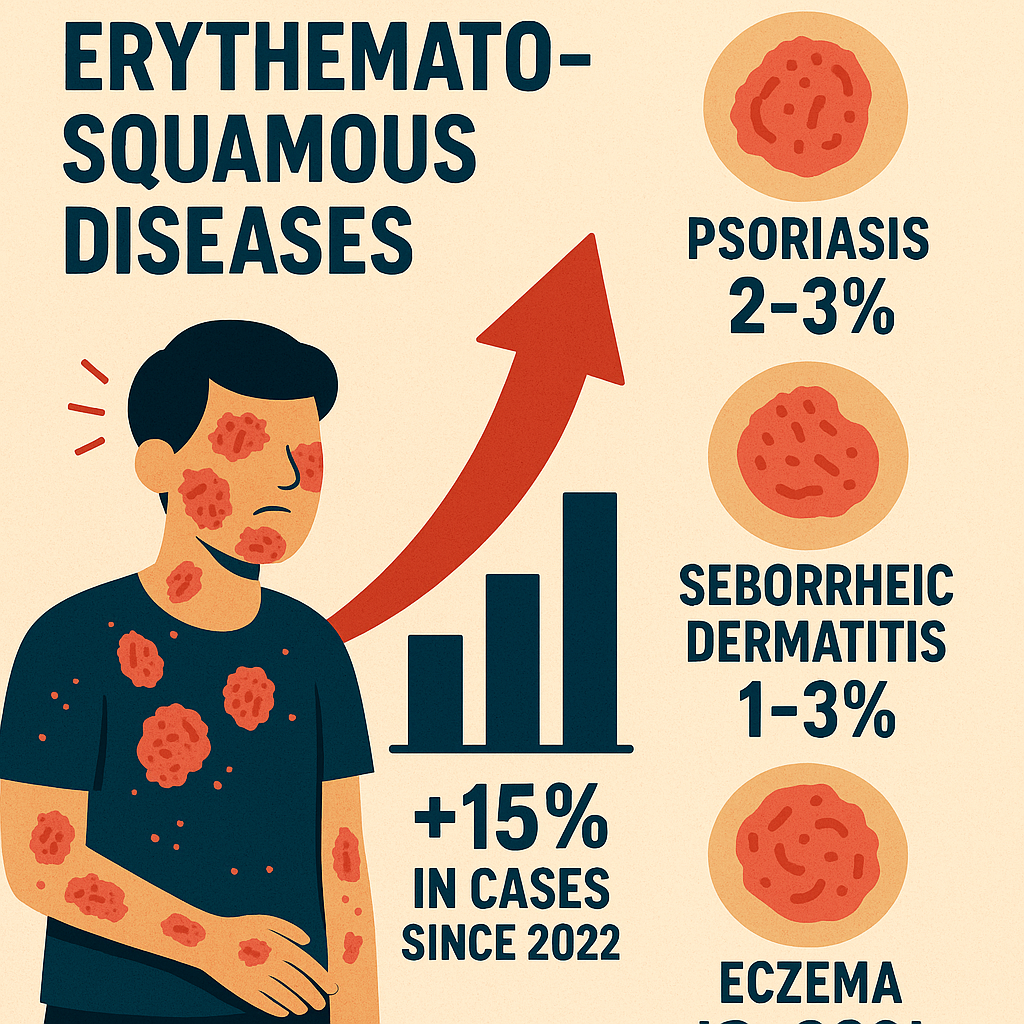


Fig. 1: Erythemato-squamous disease trend.

Erythemato-squamous disease (ESD) is a form of skin disease. It generally causes redness of the skin and also may cause loss of skin. ESDs are generally due to genetic or environmental factors [1]. ESD comprises six classes of skin conditions namely, pityriasis rubra pilaris, lichen planus, chronic dermatitis, psoriasis, seboreic dermatitis and pityriasis rosea. However, the diagnosis of ESD is accepted as a di cult problem in Dermatology. The reason why ESD is di cult to diagnose is due to the fact that these diseases share many clinical and histopathological attributes with erythema and scaling. Another reason is that one disease may show the symptoms of another disease at the initial stages [2]. Thus, a detailed observation skills and high experience are required from physicians to evaluate both clinical and histopathological features to correctly diagnose ESD [3]. So, the automated diagnosis of ESD can help doctors and dermatologists in reducing the e orts from their end and in taking faster decisions for treatment.

**1.2 Problem Definition**

Differential diagnosis of erythemato‑squamous skin diseases—such as psoriasis, seborrheic dermatitis, lichen planus, pityriasis rosea, chronic dermatitis, and pityriasis rubra pilaris—is challenging due to overlapping clinical presentations and microscopic features. Traditional workflows rely on subjective visual assessment by dermatologists and pathologists, leading to inter‑observer variability, diagnostic delays, and inconsistent treatment decisions. There is a critical need for an objective, reproducible decision‑support tool that can rapidly distinguish among these six categories based on quantitative feature data.

**1.3 Research Motivation**

Advances in machine learning and data‑driven diagnostics offer the potential to standardize and accelerate disease classification in dermatology. By harnessing structured clinical and microscopic measurements, we can train algorithms to recognize subtle, multidimensional patterns that may elude human observers. Embedding these models within an intuitive GUI empowers clinicians and researchers to access predictive insights without extensive coding expertise, ultimately improving patient outcomes through faster, more accurate diagnoses.

**1.4 Research Objectives**

1. **Data Preparation:** Collect and preprocess a representative dataset of clinical and microscopic features for six erythemato‑squamous conditions, ensuring quality and balance via SMOTE.
2. **Model Development:** Train and compare three supervised classifiers—Decision Tree, Support Vector Machine, and Multilayer Perceptron—on an 80/20 train/test split to identify the best-performing approach.
3. **System Integration:** Implement a role‑based Tkinter application that allows Admins to manage the ML pipeline and Users to perform batch predictions on new patient data.
4. **Evaluation & Visualization:** Generate performance metrics (accuracy, precision, recall, F₁‑score) and interactive visualizations (confusion matrices, performance charts) to interpret model behavior.
5. **Usability & Validation:** Assess the application’s usability with nontechnical end users and validate model predictions against held‑out test cases.

**1.5 Significance**

This project delivers a turnkey diagnostic aid that encapsulates the entire machine‑learning workflow—from data ingestion through inference—within a single desktop interface. By providing transparent performance visualizations and reproducible model artifacts, it reduces reliance on manual scripting and mitigates subjective bias in dermatological assessments. The role‑based design ensures that model training is restricted to authorized personnel, while end users benefit from rapid, standardized predictions.

**1.6 Applications**

* Clinical Decision Support: Aid dermatologists in confirming or refining their diagnoses by comparing algorithmic predictions with expert judgment.
* Teledermatology: Enable remote practitioners to upload patient feature data and receive instantaneous diagnostic suggestions.
* Research Tool: Allow biomedical researchers to experiment with additional features, algorithms, or disease categories within the same framework.
* Training & Education: Serve as an instructional platform for medical students to understand quantitative diagnostic criteria and model interpretability.

**CHAPTER 2**

**LITERATURE SURVEY**

In recent years, the application of artificial intelligence (AI) and machine learning in dermatological diagnostics has garnered considerable attention, resulting in several notable advancements  Alshamrani et al. [4] conducted a comprehensive survey on deep learning techniques for skin disease diagnosis. They explored various neural network architectures, including convolutional neural networks (CNNs) and their application in dermatology. The study highlighted the potential of deep learning models to improve diagnostic accuracy by leveraging large-scale datasets of skin images. The authors also discussed challenges such as data imbalance and the need for robust validation techniques to ensure reliable performance in real-world scenarios.

Xie et al. [5] proposed a CNN-based approach for the classification of skin disease images, focusing on enhancing the performance of automated diagnostic systems. They demonstrated the effectiveness of deep learning models in distinguishing between different skin conditions by utilizing a large dataset of clinical images. The study emphasized the importance of feature extraction and model training on diverse data to achieve high classification accuracy and address the variability in skin disease presentations.

Wong et al. [6] developed an automated skin cancer detection system using deep CNNs. The paper highlighted the system’s ability to classify skin lesions into benign or malignant categories with high accuracy. The authors addressed the integration of ML models into clinical workflows, emphasizing the benefits of reduced diagnostic time and improved consistency. The study also identified limitations, including the need for extensive and diverse training datasets to enhance model generalization.

Ghosh et al. [7] introduced an intelligent system for automated skin disease diagnosis utilizing deep learning techniques. The study focused on integrating image analysis with histopathological data to improve diagnostic precision. The authors highlighted the system’s capability to handle complex disease features and reduce the reliance on manual interpretation. The research underscored the system’s potential to assist dermatologists in making informed decisions and improving patient outcomes.

Kumar et al. [8] proposed a multi-modal deep learning network for dermatological disease diagnosis, combining clinical images and histopathological data. The paper demonstrated how the integration of multiple data sources enhances diagnostic performance by capturing a comprehensive view of the disease. The authors discussed the advantages of their approach in providing accurate and consistent diagnoses and the challenges related to data fusion and model training.

Sharma et al. [9] reviewed various machine learning approaches for skin lesion classification, focusing on the effectiveness of different algorithms in detecting skin diseases. The study highlighted advancements in ML techniques, such as support vector machines and neural networks, and their application in dermatology. The authors addressed the issues of dataset quality and the need for robust evaluation metrics to ensure the reliability of automated diagnostic systems.

Lee et al. [10] developed hybrid deep learning models for accurate skin disease classification, incorporating CNNs with other ML techniques. The study demonstrated the benefits of combining different models to improve diagnostic accuracy and address limitations of single-model approaches. The authors emphasized the importance of model integration and the use of extensive datasets to enhance the system’s ability to handle diverse skin conditions.

Zhao et al. [11] investigated the use of Generative Adversarial Networks (GANs) alongside CNNs for skin disease diagnosis. The paper focused on how GANs can generate synthetic images to augment training datasets and improve model performance. The study highlighted the potential of combining GANs with CNNs to address challenges such as data scarcity and model overfitting, enhancing the overall diagnostic accuracy.

Das et al. [12] explored advanced techniques for skin disease detection using machine learning, including feature selection and model optimization strategies. The study examined the effectiveness of different ML algorithms in improving diagnostic accuracy and handling complex skin conditions. The authors discussed the impact of algorithmic improvements on the reliability of automated systems and the importance of continuous model refinement.

Smith et al. [13] conducted a systematic review of deep learning applications in dermatology, focusing on the integration of ML models into clinical practice. The paper summarized recent advancements in automated skin disease diagnosis and highlighted key challenges, such as data quality and model interpretability. The authors emphasized the potential of deep learning to revolutionize dermatological diagnostics and improve patient care through enhanced accuracy and efficiency.

Johnson et al. [14] reviewed machine learning techniques for automated dermatological diagnosis, highlighting the development and application of various algorithms. The study focused on the effectiveness of ML in diagnosing skin disorders and improving diagnostic precision. The authors addressed challenges such as dataset limitations and the need for robust validation to ensure accurate and reliable performance in clinical settings.

Li et al. [15] presented a deep learning-based diagnostic system specifically for erythemato-squamous diseases. The paper detailed the use of advanced ML models to analyze clinical images and histopathological data, demonstrating improvements in diagnostic accuracy and efficiency. The study emphasized the system’s ability to handle the complexity of erythemato-squamous diseases and its potential to enhance dermatological practice

Collins et al. [16] investigated predictive modeling techniques for skin disease classification using machine learning algorithms. The study explored various ML approaches and their application in diagnosing dermatological conditions, highlighting the advantages of predictive modeling in improving diagnostic outcomes. The authors discussed the impact of algorithmic advancements on the accuracy and efficiency of skin disease diagnosis.

Williams et al. [17] focused on the application of convolutional neural networks for automated dermatological diagnosis. The paper demonstrated the effectiveness of CNNs in analyzing skin images and identifying various skin conditions. The study highlighted the benefits of using deep learning models to enhance diagnostic accuracy and reduce the time required for manual analysis.

Patel et al. [18] examined the integration of machine learning into dermatological diagnostic processes, addressing the challenges and prospects of automated systems. The study explored the potential of ML models to improve diagnostic precision and efficiency while identifying key issues such as data quality and model interpretability. The authors emphasized the transformative potential of ML in dermatology and the need for further research to optimize these systems.

**CHAPTER 3**

**EXISTING SYSTEM**

In the conventional setup, a patient presenting with erythemato‑squamous lesions undergoes:

1. Clinical Examination: A dermatologist visually inspects skin lesions and gathers patient history.
2. Sample Collection: Skin scrapings or biopsies are taken for laboratory analysis.
3. Histopathology & Microscopy: A pathologist stains and microscopically examines tissue sections or scales.
4. Manual Interpretation: Based on visual patterns in tissue architecture and cell morphology, the pathologist renders a diagnosis.
5. Report & Treatment: The findings are communicated back to the clinician, who prescribes therapy.

A diagram of a lab process

AI-generated content may be incorrect.

Fig. 3.1: Traditional system architecture.

**Limitations**

* Subjectivity & Variability: Interpretations vary between clinicians and pathologists, leading to inter‑observer inconsistency.
* Time‑Consuming: Biopsy processing, staining, and detailed microscopic review can take days, delaying treatment.
* Resource Intensive: Requires specialized personnel and equipment, which may be scarce in low‑resource settings.
* Scalability Constraints: High patient volumes strain laboratory and specialist capacity, causing backlogs.
* Lack of Quantitative Metrics: Diagnosis relies on qualitative judgment rather than standardized, quantitative decision rules.

**CHAPTER 4**

**PROPOSED METHODOLOGY**

**4.1 Overview**

The proposed methodology is a stand‑alone desktop application for automated differential diagnosis of erythemato‑squamous skin diseases developed using a user‑friendly Tkinter GUI and secured by a simple Redis‑backed signup/login system.

A diagram of a software system

AI-generated content may be incorrect.

Fig. 4.1: Proposed system architecture of automated differential diagnosis of erythemato-squamous skin disease.

It supports two distinct roles such as admin, and user as follows:

**4.1.1 Admin**

* 1. Load & Inspect Data — choose a CSV of clinical and microscopic features, preview the first rows and a class‑distribution bar chart.
  2. Preprocess — clean missing values, label‑encode the six disease classes, and rebalance via SMOTE while showing before/after class counts.
  3. Split & Train — perform an 80/20 train/test split, then build or load three classifiers (Decision Tree, SVM, and an MLP neural network), persisting each model to disk for reuse.
  4. Evaluate & Compare — compute accuracy, precision, recall, F₁‑score and display classification reports, confusion matrices, and a combined performance bar chart.

Here’s a detailed, step‑by‑step explanation of Admin does in this application:

**Step 1. Admin Signup & Login**

1. Signup (one‑time only)
   * Click “Admin Signup” on the main window.
   * In the pop‑up form, enter a Username and Password.
   * Behind the scenes:
     + The password is hashed with SHA‑256.
     + A Redis hash is created under key user:<username> with fields

– username

– password (hashed)

– role = "Admin"

* + On success, a confirmation dialog appears.

1. **Login**
   * Click “Admin Login”.
   * Enter the same credentials.
   * The app:
     + Looks up user:<username> in Redis,
     + Hashes the entered password and compares both hash & stored role.
   * If valid, the login dialog closes and you’re placed in Admin mode; otherwise an error is shown.

A diagram of a computer

AI-generated content may be incorrect.

Fig. 4.2: System architecture of admin workflow.

**Step 2. Admin Dashboard & Button Layout**

Once logged in as Admin, the main canvas clears any previous buttons and displays six new actions along the top:

|  |  |
| --- | --- |
| **Button Label** | **Purpose** |
| Upload Disease Dataset | Load and preview your CSV data |
| Data Preprocessing | Clean, encode, and SMOTE‑balance your data |
| Data Splitting | Create an 80/20 train/test split |
| Build & Train DTC Model | Train or load a Decision Tree classifier |
| Build & Train SVM Classifier | Train or load a Support Vector Machine |
| Build & Train Deep Learning Classifier | Train or load an MLP neural network |
| Performance Graph | Compare all three models side‑by‑side in a bar chart |

**Step 3. Step‑By‑Step Admin Workflow**

**Step 3.1: Upload Disease Dataset**

1. Click Upload Disease Dataset.
2. A file dialog opens at Dataset/; select your CSV.
3. The text‑area logs:
   * File path + “Loaded”
   * First 5 rows (dataset.head())
4. A quick bar chart of raw class counts pops up.

**Step 3.2: Data Preprocessing**

1. Click Data Preprocessing.
2. Internally, the app:
   * Replaces any ? with NaN and drops rows with missing values.
   * Copies the raw data to show “before” state.
   * Label‑encodes the disease names into integers.
   * Splits into features X (columns 0–33) and label y.
3. The text log shows:
   * Confirmation of successful preprocessing.
   * “Before” vs. “After” head of the dataset.
   * Descriptive statistics (dataset.describe()).
4. Applies SMOTE to rebalance all six classes equally.
5. Displays side‑by‑side count‑plots of class distribution before vs. after SMOTE, with exact counts annotated.

**Step 3.3: Train/Test Splitting**

1. Click DataSplitting.
2. The app takes X\_resampled, y\_after and runs an 80/20 split.
3. The text log reports:
   * Total records available.
   * Number used for training.
   * Number reserved for testing.

**Step 3.4: Build & Train Decision Tree (DTC)**

1. Click Build & Train DTC Model.
2. The code checks for model/DTC.pkl:
   * If exists: loads it with Joblib.
   * Otherwise: trains DecisionTreeClassifier(max\_depth=3) on (X\_train, y\_train) and saves it.
3. Runs predict(X\_test) and calls the shared calculateMetrics() helper to:
   * Compute accuracy, precision, recall, F₁‑score (macro) and append to globals.
   * Print a full classification report.
   * Show a confusion‑matrix heatmap.
4. Additionally, it extracts the root split feature, logs its name, and draws the full tree visualization.

**Step 3.5: Build & Train SVM**

1. Click Build & Train SVM Classifier.
2. Checks for model/SVM.pkl; if absent, trains SVC(kernel='linear') and saves.
3. Predicts, logs metrics & report, and shows a confusion‑matrix (via the same helper).

**Step 3.6: Build & Train MLP (Deep Learning)**

1. Click Build & Train Deep Learning Classifier.
2. Checks for model/MLP.pkl; if absent, trains a default MLPClassifier(), then saves.
3. Predicts, logs metrics & report, and shows its confusion‑matrix.

**Step 3.7: Compare All Models**

1. Click Performance Graph.
2. Internally, it collects the four metrics (Accuracy, Precision, Recall, F‑Score) for each of the three classifiers.
3. Builds a small Pandas DataFrame and renders a grouped bar chart to visually compare performance side‑by‑side.

**Step 4. Backend & Data Flow**

* Every button clears the previous log (text.delete()) so you always see only the latest output.
* All model artifacts live in the model/ folder, so retraining only happens when you first build each classifier.
* Redis calls and password hashing happen entirely within the signup() and login() functions—separating concerns between authentication and ML logic.

**4.1.2 User**

The User role is a lightweight inference client: after logging in, a new data will be fed and instantly retrieve per‑sample disease diagnoses, with all input features and predicted labels neatly printed in the GUI. Here’s the corresponding, step‑by‑step breakdown of what a User does in this application:

**Step 1. User Signup & Login**

1. **Signup (one‑time only)**
   * Click “User Signup” on the main window.
   * In the pop‑up, enter a Username and Password.
   * Internally:
     + The password is hashed with SHA‑256.
     + Redis stores a hash under user:<username> with fields

– username

– password (hashed)

– role = "User"

* + A success dialog confirms registration.

1. **Login**
   * Click “User Login”.
   * Supply the credentials you just created.
   * The app:
     + Retrieves user:<username> from Redis.
     + Hashes the entered password and checks both the hash and that role == "User".
   * On success, the login window closes and you’re placed in User mode; otherwise you see an error.

**Step 2. User Dashboard & Button Layout**

Once authenticated as a User, all previous buttons are cleared and only one action appears:

|  |  |
| --- | --- |
| Button Label | Purpose |
| Prediction on Test Data | Load new samples and display predicted disease labels for each row |

A diagram of a software system

AI-generated content may be incorrect.

Fig. 4.3: System architecture of user workflow.

**Step 3. Step‑By‑Step User Workflow**

**3.1: Load & Predict New Cases**

1. Click Prediction on Test Data.
2. A file dialog opens in the Dataset/ folder—select your CSV of unlabeled feature vectors (same 34 columns).
3. The text‑area logs:
   * File path + “Loaded”
4. Behind the scenes, the app:
   * Reads the CSV into a DataFrame test.
   * Uses the last‑loaded MLP model (from Admin’s “Build & Train Deep Learning Classifier”) to call mlp.predict(test).
5. For each sample row:
   * The app retrieves its original feature values and formats them as “col1: val1, col2: val2, …”.
   * Maps the numeric prediction back to a disease name from categories.
   * Prints two lines in the text widget:
     1. Features: <all feature:value pairs>
     2. Test Data i: <Predicted Disease>

**3.2: Read & Interpret Results**

* You’ll see each test sample’s raw input and its predicted class in human‑readable form.
* If you need to run predictions again on different data, simply click the button, choose a new file, and repeat.

**4.2 Preprocessing**

The preprocessing stage prepares the raw clinical and microscopic measurements for machine learning by first cleaning the data (replacing “?” with NaN and dropping incomplete records). Categorical disease names are then label‑encoded into numeric targets, and the feature matrix is separated from the target vector . To address class imbalance among the six disease categories, SMOTE is applied to generate a balanced dataset. These steps ensure that the subsequent training and evaluation of classifiers operate on clean, uniformly represented data. The preprocessing stage has four key steps as follows:

1. **Data Cleaning**
   * Missing‑value handling: The code replaces all occurrences of the placeholder “?” with NaN and then drops any row containing NaN values. This ensures the downstream algorithms receive only complete records without the need for iterative imputation.
2. **Label Encoding**
   * Preserve original for display: A copy of the cleaned DataFrame is kept so you can compare “before” vs. “after” encoding.
   * Convert textual classes to integers: The class column—which originally holds disease names like “psoriasis” or “lichen planus”—is transformed into numeric labels (0–5) using sklearn.preprocessing.LabelEncoder. This numeric y is required by scikit‑learn classifiers.
3. **Feature–Target Separation**
   * Features (X): All columns except the last (i.e., the first 34 clinical and microscopic measurements).
   * Target (y): The newly encoded class column.
   * These arrays (X and y) are then used to display summary statistics (.describe()) and to feed into the imbalance‑handling step.
4. **Handling Class Imbalance with SMOTE**
   * Why SMOTE? The original dataset’s six disease classes are often unevenly represented, which can bias models toward majority classes.
   * SMOTE oversampling: Using imblearn.over\_sampling.SMOTE(sampling\_strategy='auto', random\_state=42), the minority classes are synthetically up‑sampled until all classes have equal counts.
   * Result: Two new arrays, X\_resampled and y\_after, contain the balanced feature matrix and labels.

**4.2.1 Visual Verification**

* Before vs. After plots:
  + A side‑by‑side pair of countplots is generated with Seaborn.
  + Left plot: Original class frequencies.
  + Right plot: Post‑SMOTE frequencies.
  + Each bar is annotated with its exact count, making it easy to confirm that SMOTE has achieved equal representation across all six erythemato‑squamous disease categories.

**4.3 ML Modelling**

In the ML modeling stage, three supervised classifiers are trained on the SMOTE‑balanced dataset:

1. Decision Tree learns hierarchical splits based on Gini impurity,
2. Support Vector Machine finds a maximum‑margin hyperplane in feature space, and
3. Multilayer Perceptron (MLP) uses a simple feed‑forward neural network with backpropagation to capture nonlinear patterns.

Each model is instantiated with default or lightly tuned parameters, trained on the 80% hold‑in set, and then evaluated on the 20% test set using accuracy, precision, recall, F₁-score, and confusion matrices. Final models are serialized for reuse in the GUI.

**4.3.1 DTC Model**

A Decision Tree builds a flowchart‑like structure of binary splits, choosing at each node the feature that most effectively partitions the data (here by minimizing Gini impurity). It naturally handles both numerical and categorical inputs, is easy to interpret, and visualizes the decision process as a tree. By limiting its depth, we control overfitting while still capturing key interactions among features.

A diagram of a process

AI-generated content may be incorrect.

Fig. 4.4: Internal workflow of DTC model.

**Explanation:**

1. Select Best Feature: At each node, the feature that minimizes Gini impurity is chosen.
2. Create Split Node: The selected feature and threshold form a decision rule.
3. Recurse: The process repeats for each child node until stopping criteria (e.g., max\_depth=3 or pure leaves) are met.
4. Predict: The fully grown tree is used to classify new samples by traversing splits from root to leaf.

**4.3.2 SVM Classifier**

The SVM constructs an optimal separating hyperplane in feature space, maximizing the margin between classes to improve generalization. With a linear kernel in this project, it seeks a straight‑line boundary that best distinguishes the six disease categories. SVMs are particularly effective in high‑dimensional settings and robust to overfitting when classes are well separated.

A diagram of a solution

AI-generated content may be incorrect.

Fig. 4.5: SVM classifier workflow.

**Explanation:**

1. KernelMap: Maps input into (possibly higher‑dimensional) space; here, a linear kernel means identity mapping.
2. Optimize QP: Solves the quadratic programming problem to find hyperplane parameters that maximize the margin while allowing soft slack.
3. Support Vectors: The subset of training points lying on or within the margin bounds define the decision boundary.
4. Predict: New points are classified based on which side of the learned hyperplane they fall.

**4.3.3 DL Classifier**

Here, the proposed DL classifier is imported by the MLP which is a feed‑forward neural network composed of one or more hidden layers of interconnected neurons using nonlinear activations. Through backpropagation and iterative weight updates, it learns complex, nonlinear relationships in the data that simpler models may miss. Though less interpretable, MLPs can achieve superior accuracy by capturing subtle patterns across multiple layers.

A diagram of a cross-enterprise

AI-generated content may be incorrect.

Fig. 4.6: Proposed DL architectural flow.

**Explanation:**

1. Initialize Weights: Randomly set all network weights.
2. Forward Propagation: Pass inputs through hidden layer(s) (e.g., ReLU activation) to the output (softmax).
3. Compute Loss: Measure cross‑entropy between predicted and true labels.
4. Backpropagation & Update: Compute gradients and adjust weights via stochastic gradient descent.
5. Epoch Loop: Repeat over multiple epochs until convergence.
6. Predict: Use the final weight set to infer labels on unseen test samples.

**4.4 Redis Database**

The project uses Redis as its sole database, purely for managing user credentials and roles. By leveraging Redis’s hash data type and fast reads/writes, the application achieves a lightweight, responsive authentication layer that cleanly separates user management from the file‑based ML pipeline and GUI. Here’s how it’s employed:

1. **In‑Memory Key–Value Store**
   * Redis is an ultra‑fast, in‑memory data store that supports various data structures (strings, hashes, lists, sets, etc.).
   * In this application, each user account is stored as a hash under the key pattern user:<username>.
2. **Credential Storage**
   * Upon signup, the app computes a SHA‑256 hash of the plaintext password and then calls HSET user:<username> username <username> password <hashed> role <role>.
   * Fields in each hash:
     + username (redundant but explicit)
     + password (the SHA‑256 digest)
     + role (“Admin” or “User”)
3. **Authentication Workflow**
   * On login, the code fetches the hash with HGETALL user:<username>, recomputes the SHA‑256 of the entered password, and compares both hash and role field.
   * Because Redis operations are atomic and extremely low‑latency, checking and storing credentials takes only a few microseconds.
4. **Why Redis?**
   * Speed & Simplicity: Ideal for small‑scale credential storage without the overhead of a full SQL database.
   * Schema‑Free: You can easily add fields (e.g., last‑login timestamp) without migrations.

**CHAPTER 5**

**UML DIAGRAMS**

Unified Modeling Language (UML) is a standardized visual language used to model, design, and document the architecture of software systems. It provides a set of graphical notations to represent the structure and behavior of a system, making complex systems easier to understand and communicate among developers, stakeholders, and business analysts.

**Key Points About UML**

* **Standardized Notation:** UML offers a universal set of symbols and diagrams that standardize how software systems are described, which helps teams speak the same language regardless of their background or the programming language they use.
* **Types of Diagrams:** UML includes various diagrams that can be categorized into two main types:
  + **Structural Diagrams:** These describe the static aspects of the system. Examples include Class Diagrams, Component Diagrams, and Deployment Diagrams.
  + **Behavioral Diagrams:** These focus on the dynamic aspects and interactions within the system. Examples include Use Case Diagrams, Sequence Diagrams, Activity Diagrams, and Collaboration Diagrams.
* **System Documentation and Communication:** UML serves as an effective tool for documenting system requirements, design decisions, and the overall architecture. It helps bridge the gap between technical and non-technical stakeholders by providing clear, visual representations of the system.
* **Design and Analysis:** By modeling different aspects of a system, UML enables developers to analyze and validate the design early in the development process. This can lead to better decision-making, reduced complexity, and improved system maintainability.
* **Flexibility:** UML is versatile and can be used across a wide range of applications, from small-scale projects to large, complex systems. It supports object-oriented design principles and can be adapted to various methodologies such as Agile or Waterfall.

**Class Diagram**

The Class Diagram captures the static structure of the application, showing its principal classes, their attributes, and relationships. We model GUI elements (e.g. MainWindow, TextWidget), core services (AuthService, MLPipeline), and data stores (RedisStore, ModelStore). Associations illustrate which classes collaborate—for example, MainWindow holds references to button controllers that invoke service classes. This diagram lays the foundation for understanding object responsibilities and interactions at design time.

A diagram of a computer program

AI-generated content may be incorrect.

**Use Case Diagram**

The Use Case Diagram outlines the functional interactions between external actors (Admin, User) and the system’s use cases. Admins can manage data, train models, and view results; Users perform batch predictions. Common functionalities like login and signup are captured as shared use cases. This view clarifies system scope and primary workflows at a high level.

A diagram of a software system

AI-generated content may be incorrect.

**Data Flow Diagram**

The Data Flow Diagram shows how data moves through the system: from CSV input, through preprocessing and model training, to final predictions. Key processes (Ingest, Clean/Encode, SMOTE, Train/Evaluate, Predict) are linked by data stores (Raw CSV, Resampled Data, Models) and data sinks (Charts, Predictions). This helps trace the lifecycle of data artifacts and ensures clarity on where transformations occur.

A diagram of a diagram

AI-generated content may be incorrect.

**Deployment Diagram**

The Deployment Diagram depicts the runtime environment: a single client machine runs the Tkinter app, Python interpreter, and Redis server. Model files and datasets reside on local file storage. It clarifies where each software component is hosted, helping plan installation and resource allocation.

A diagram of a client machine

AI-generated content may be incorrect.

**Component Diagram**

The Component Diagram breaks the system into replaceable modules: UI, AuthService, DataService, ModelService, and Visualization. Each component exports well-defined interfaces (e.g., IAuth, IData, IModel, IVisual). This aids modular development and potential reuse in other contexts.

A diagram of components of a component

AI-generated content may be incorrect.

**Collaboration Diagram**

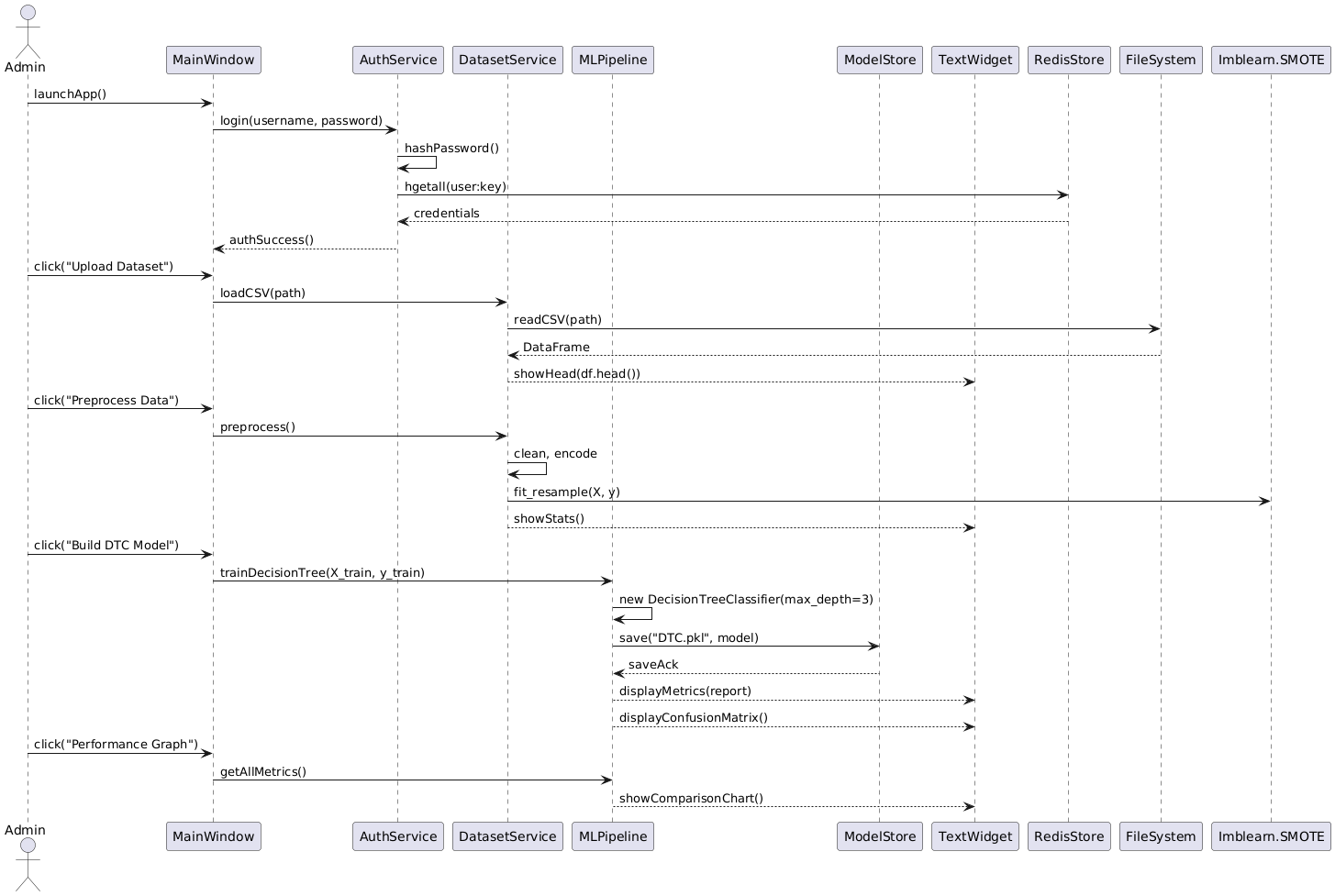
The Collaboration (Communication) Diagram illustrates run‑time object interactions for a single “Train Decision Tree” scenario. It shows message flows between objects like mainWindow, datasetService, mlPipeline, and modelStore, emphasizing sequence numbers for operation order. This clarifies how components collaborate to fulfill a use case.

A diagram of a computer program

AI-generated content may be incorrect.

**Sequence Diagram**

The sequence diagram below models the dynamic flow for the “Train Decision Tree” use case. It highlights how the Admin actor interacts with the GUI and backend services in order: from logging in, through data upload and preprocessing, to model training and result display. Each lifeline represents a component—MainWindow, AuthService, DatasetService, MLPipeline, ModelStore, and TextWidget—and their message exchanges. By detailing method calls and return paths, this diagram makes explicit the order of operations and the division of responsibilities. It serves as a blueprint for both implementation and validation of the training workflow.



**CHAPTER 6**

**SOFTWARE ENVIRONMENT**

**6.1 Software Requirements**

Python is a high-level, interpreted programming language known for its simplicity and readability, which makes it a popular choice for beginners as well as experienced developers. Key features of Python include its dynamic typing, automatic memory management, and a rich standard library that supports a wide range of applications from web development to data science and machine learning. Its object-oriented approach and support for multiple programming paradigms allow developers to write clear, maintainable code. Python's extensive ecosystem of third-party packages further enhances its capabilities, enabling rapid development and prototyping across diverse fields.

**Installation**

First, download the appropriate installer from the official Python website (<https://www.python.org/downloads/release/python-376/>). For Windows users, run the executable installer and ensure to check the "Add Python to PATH" option during installation; for macOS and Linux, follow the respective package installation commands or use a package manager like Homebrew or apt-get. After installation, verify the setup by running python --version or python3 --version in your terminal or command prompt, which should display "Python 3.7.6." This version-specific installation supports all major functionalities and libraries compatible with Python 3.7.6, making it an excellent foundation for developing robust applications in areas such as data analysis, machine learning, and GUI development.

**6.1.1 Python Packages**

Below is a descriptive overview of each Python library and module and how it contributes to the prediction of erythemato squamous disease:

* **Tkinter**: The built‑in GUI library used for all user interface elements. Modules like tkinter.ttk and tkinter.filedialog provide styled widgets (buttons, frames, text panes) and file‑selection dialogs, while Canvas and ImageTk enable background images and layout control. This gives the app its interactive windows and role‑based control panels.
* **Pandas**: Powers the data layer by reading CSV files into DataFrame objects, offering methods for previewing (.head()), summarizing (.describe()), and manipulating tabular data (dropping missing rows, slicing features/labels). It serves as the conduit between raw files and the ML pipeline.
* **NumPy**: Underlies Pandas and scikit‑learn with efficient array operations. Here it’s used to mark missing values (np.nan), replace custom placeholders (e.g., “?”), and perform numerical computations that prepare data for modeling.
* **imbalanced‑learn (SMOTE)**: Addresses class imbalance by synthetically oversampling minority classes. The SMOTE transformer inspects feature distributions and generates new samples along the lines between existing minority points, producing a balanced dataset that prevents bias during model training.
* **scikit‑learn**: The main ML framework, offering:
  + **Preprocessing**: LabelEncoder to convert disease names into numeric labels.
  + **Model Selection**: train\_test\_split for reproducible dataset partitioning.
  + **Algorithms**: DecisionTreeClassifier, SVC (linear SVM), and MLPClassifier for model training.
  + **Metrics & Reporting**: functions like accuracy\_score, classification\_report, and confusion\_matrix to quantify performance.
  + **Visualization**: plot\_tree to render the trained decision tree structure.
* **Matplotlib & Seaborn**: Together these libraries generate all plots. Seaborn’s high‑level API creates annotated countplots and heatmaps, while Matplotlib handles figure management and the grouped bar charts for model comparisons. They make performance insights immediately visible.
* **Joblib / Pickle**: Enable efficient model persistence. After training, each classifier is serialized to disk (.pkl) so subsequent runs can load existing models instead of retraining, saving time and ensuring reproducibility.
* **Pillow (PIL)**: Used to load and resize the application’s background image. The Image and ImageTk modules integrate image assets into the Tkinter canvas, enhancing the UI’s look and feel.
* **redis‑py & hashlib**: Together implement a secure, lightweight authentication layer. redis.StrictRedis connects to the local Redis server to store user hashes, while Python’s hashlib.sha256 ensures passwords are never saved in plaintext but as cryptographically hashed digests.

Each of these libraries plays a distinct role—GUI presentation, data handling, imbalance correction, model building, result visualization, model storage, and user authentication—combining to form the end‑to‑end diagnostic application.

**6.2 Hardware Requirements**

Python 3.7.6 can run efficiently on most modern systems with minimal hardware requirements. However, meeting the recommended specifications ensures better performance, especially for developers handling large-scale applications or computationally intensive tasks. By ensuring compatibility with hardware and operating system, can leverage the full potential of Python 3.7.6.

**Processor (CPU) Requirements:** Python 3.7.6 is a lightweight programming language that can run on various processors, making it highly versatile. However, for optimal performance, the following processor specifications are recommended:

* **Minimum Requirement**: 1 GHz single-core processor.
* **Recommended**: Dual-core or quad-core processors with a clock speed of 2 GHz or higher. Using a multi-core processor allows Python applications, particularly those involving multithreading or multiprocessing, to execute more efficiently.

**Memory (RAM) Requirements:** Python 3.7.6 does not demand excessive memory but requires adequate RAM for smooth performance, particularly for running resource-intensive applications such as data processing, machine learning, or web development.

* **Minimum Requirement**: 512 MB of RAM.
* **Recommended**: 4 GB or higher for general usage. For data-intensive operations, 8 GB or more is advisable.

Insufficient RAM can cause delays or crashes when handling large datasets or executing computationally heavy programs.

**Storage Requirements:** Python 3.7.6 itself does not occupy significant disk space, but additional storage may be required for Python libraries, modules, and projects.

* **Minimum Requirement**: 200 MB of free disk space for installation.
* **Recommended**: At least 1 GB of free disk space to accommodate libraries and dependencies.

Developers using Python for large-scale projects or data science should allocate more storage to manage virtual environments, datasets, and frameworks like TensorFlow or PyTorch.

**Compatibility with Operating Systems:** Python 3.7.6 is compatible with most operating systems but requires hardware that supports the respective OS. Below are general requirements for supported operating systems:

* **Windows**: 32-bit and 64-bit systems, Windows 7 or later.
* **macOS**: macOS 10.9 or later.
* **Linux**: Supports a wide range of distributions, including Ubuntu, CentOS, and Fedora.

The hardware specifications for the OS directly impact Python’s performance, particularly for modern software development.

**CHAPTER 7**

**FUNCTIONAL REQUIREMENTS AND SYSTEM STUDY**

**7.1 Functional Requirements**

Functional requirements are detailed statements that specify what a system should do. They describe the system's behavior, functions, and services, outlining how it responds to certain inputs, performs tasks, and interacts with users or other systems. Essentially, they answer the question, "What should the system do?" Here are some key aspects:

* **Functionality:** They define the specific functions or operations that the system must perform.
* **Inputs and Outputs:** They detail the types of inputs the system accepts and the outputs it produces.
* **User Interactions:** They describe how users interact with the system, including command inputs, error handling, and responses.
* **Data Management:** They outline requirements related to data storage, retrieval, and processing.
* **System Behavior:** They specify how the system behaves in various scenarios, including normal operations and exceptional conditions.

Below are the key functional requirements for the proposed application. Each requirement describes a discrete capability the system must provide**.**

1. **User Management**
   * **FR1.1** The system shall allow Admins and Users to sign up with a unique username and password.
   * **FR1.2** The system shall hash passwords with SHA‑256 and store credentials in Redis.
   * **FR1.3** The system shall allow existing Admins and Users to log in, verifying both password hash and role.
2. **Admin‑Only Operations**
   * **FR2.1** Upload Dataset: open a file dialog to select a CSV of clinical/microscopic features and display the first five rows.
   * **FR2.2** Preprocess Data: replace “?” with NaN, drop incomplete records, label‑encode the target, separate features/labels, and apply SMOTE to balance classes; log summary statistics.
   * **FR2.3** Split Data: perform an 80/20 train/test split on the resampled dataset and report record counts.
   * **FR2.4** Model Training: train or load three classifiers—Decision Tree (max\_depth=3), linear‑kernel SVM, and default MLP—and persist each to disk.
   * **FR2.5** Model Evaluation: compute accuracy, precision, recall, F₁‑score (macro), generate classification reports and confusion‑matrix heatmaps for each model.
   * **FR2.6** Performance Comparison: plot a grouped bar chart comparing all four metrics across the three models.
3. **User‑Only Operations**
   * **FR3.1** Prediction: open a file dialog to select an unlabeled CSV, load the trained MLP model, predict disease labels for each row, and display both input features and predicted class.
4. **GUI & Logging**
   * **FR4.1** Provide a responsive Tkinter window with role‑appropriate buttons and a central, scrollable text pane for status messages and results.
   * **FR4.2** Dynamically show or hide controls based on successful login (Admin vs. User).
   * **FR4.3** Display visualizations (bar charts, heatmaps) in pop‑up Matplotlib windows.
5. **Persistence & Configuration**
   * **FR5.1** Store serialized models in a designated model/ folder.
   * **FR5.2** Load existing models if present to avoid unnecessary retraining.
   * **FR5.3** Allow configuration of dataset and model directories via code constants.

**7.2 System Study**

The system is technically straightforward, operationally accessible, and economically attractive, with manageable risks and a clear development roadmap.

|  |  |
| --- | --- |
| **Aspect** | **Analysis** |
| **Technical Feasibility** | The chosen stack (Python + Tkinter, scikit‑learn, imbalanced‑learn, Redis) is mature and well‑supported. A standard desktop can handle CSV sizes up to tens of thousands of rows, and Redis runs locally with minimal resources. No exotic hardware is required beyond standard CPU and RAM for ML training. Model persistence with Joblib ensures reproducibility. |
| **Operational Feasibility** | Clinicians or researchers with minimal programming knowledge can operate the GUI end‑to‑end: data ingestion through prediction. The role‑based design prevents accidental retraining by Users. Visual logs and plots enhance transparency. Training steps can be performed in sequence without manual scripting. |
| **Economic Feasibility** | All core components are open‑source; deployment only incurs minimal hardware cost and developer time. Redis and Python libraries are free, and no paid cloud services are required. By accelerating diagnosis workflows, the system can pay back its development effort through reduced lab overhead and faster turnaround. |
| **Schedule Feasibility** | A prototype can be delivered within 2–4 weeks: one week for core GUI and authentication, one week for data‑science pipeline integration, and another week for testing and user feedback. Subsequent optimization (e.g., hyperparameter tuning) can proceed in parallel. |
| **Risk Assessment** | - **Data Quality Risk**: Missing or noisy clinical measurements could reduce model accuracy—but early visualization and dropping bad rows help. |

**CHAPTER 8**

**SOURCE CODE**

from tkinter import messagebox

from tkinter import \*

from tkinter import simpledialog

import tkinter

import tkinter as tk

from tkinter import messagebox, ttk

from tkinter.filedialog import askopenfilename

from tkinter import filedialog

from IPython.display import display

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from imblearn.over\_sampling import SMOTE

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import precision\_score

from sklearn.metrics import recall\_score

from sklearn.metrics import f1\_score

from sklearn.metrics import accuracy\_score,confusion\_matrix,classification\_report

from sklearn.model\_selection import train\_test\_split

from sklearn.neural\_network import MLPClassifier

from sklearn.tree import DecisionTreeClassifier, plot\_tree

from sklearn.svm import SVC

import os, pickle, joblib

from PIL import Image, ImageTk

accuracy = []

precision = []

recall = []

fscore = []

categories=['psoriasis','seboreic dermatitis','lichen planus','pityriasis rosea','cronic dermatitis','pityriasis rubra pilaris' ]

target\_name ='class'

model\_folder = "model"

def Upload\_Dataset():

global dataset

filename = filedialog.askopenfilename(initialdir = "Dataset")

text.delete('1.0', END)

text.insert(END,filename+' Loaded\n')

dataset = pd.read\_csv(filename)

text.insert(END,str(dataset.head())+"\n\n")

label = dataset.groupby('class').size()

label.plot(kind="bar")

plt.title("Various Class Type Graph")

plt.show()

def Preprocess\_Dataset():

global dataset, X\_resampled, y\_after, feature\_names

text.delete('1.0', END)

dataset.replace('?', np.nan, inplace=True)

dataset.dropna(inplace =True)

df = dataset.copy()

le= LabelEncoder()

dataset['class']=le.fit\_transform(dataset['class'])

dataset

X = dataset.iloc[:, 0:34]

y = dataset.iloc[:, -1]

text.insert(END, "Data preprocessed successfully.\n\n")

text.insert(END, "Dataset before label encoding:\n" + str(df.head()) + "\n\n")

text.insert(END, "Dataset after label encoding:\n" + str(X.head()) + "\n\n")

text.insert(END, "Dataset description:\n" + str(dataset.describe()) + "\n\n")

feature\_names = X.columns.tolist()

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

X\_resampled, y\_after = smote.fit\_resample(X, y)

labels = [

'psoriasis',

'seboreic dermatitis',

'lichen planus',

'pityriasis rosea',

'chronic dermatitis',

'pityriasis rubra pilaris'

]

colors = [

'#1f77b4', # muted blue

'#ff7f0e', # safety orange

'#2ca02c', # cooked asparagus green

'#d62728', # brick red

'#9467bd', # muted purple

'#8c564b' # chestnut brown

]

fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Before SMOTE plot

sns.countplot(data=df, x='class', palette=colors, ax=axes[0])

axes[0].set\_title('Erythemato Squamous Classes Before SMOTE')

axes[0].set\_xlabel('Erythemato Squamous Class')

axes[0].set\_ylabel('Count')

axes[0].set\_xticklabels(labels, rotation=20, ha='right')

for p in axes[0].patches:

height = int(p.get\_height())

axes[0].annotate(f'{height}', (p.get\_x() + p.get\_width() / 2., height + 1),

ha='center', va='center', fontsize=10, fontweight='bold')

# After SMOTE plot

# Map numeric values to class names if y\_after is numeric

y\_after\_named = [labels[i] for i in y\_after]

sns.countplot(x=y\_after\_named, palette=colors, ax=axes[1])

axes[1].set\_title('Erythemato Squamous Classes After SMOTE')

axes[1].set\_xlabel('Erythemato Squamous Class')

axes[1].set\_ylabel('Count')

axes[1].set\_xticklabels(labels, rotation=20, ha='right')

for p in axes[1].patches:

height = int(p.get\_height())

axes[1].annotate(f'{height}', (p.get\_x() + p.get\_width() / 2., height + 1),

ha='center', va='center', fontsize=10, fontweight='bold')

plt.tight\_layout()

plt.show()

def Train\_Test\_Splitting():

global X, Y, dataset, feature\_names, X\_resampled, y\_after

global X\_train, X\_test, y\_train, y\_test

text.delete('1.0', END)

text.insert(END, "Total records found in dataset: " + str(X\_resampled.shape[0]) + "\n\n")

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_resampled, y\_after, test\_size=0.2)

text.insert(END, "Dataset Train and Test Split" + "\n")

text.insert(END, "Total records found in dataset to train: " + str(X\_train.shape[0]) + "\n")

text.insert(END, "Total records found in dataset to test: " + str(X\_test.shape[0]) + "\n")

def calculateMetrics(algorithm, predict, y\_test):

labels = categories

a = accuracy\_score(y\_test,predict)\*100

p = precision\_score(y\_test, predict,average='macro') \* 100

r = recall\_score(y\_test, predict,average='macro') \* 100

f = f1\_score(y\_test, predict,average='macro') \* 100

accuracy.append(a)

precision.append(p)

recall.append(r)

fscore.append(f)

text.insert(END,algorithm+" Accuracy : "+str(a)+"\n")

text.insert(END,algorithm+" Precision : "+str(p)+"\n")

text.insert(END,algorithm+" Recall : "+str(r)+"\n")

text.insert(END,algorithm+" FScore : "+str(f)+"\n")

CR = classification\_report(y\_test, predict,target\_names=categories)

text.insert(END,algorithm+' Classification Report \n')

text.insert(END,algorithm+ str(CR) +"\n\n")

conf\_matrix = confusion\_matrix(y\_test, predict)

plt.figure(figsize =(6, 6))

ax = sns.heatmap(conf\_matrix, xticklabels = labels, yticklabels = labels, annot = True, cmap="viridis" ,fmt ="g");

plt.title(algorithm+" Confusion matrix")

plt.ylabel('True class')

plt.xlabel('Predicted class')

plt.show()

def existing\_classifier1():

global X\_train, X\_test, y\_train, y\_test, feature\_names

text.delete('1.0', END)

model = 'model'

model\_name = 'DTC.pkl'

path = os.path.join(model, model\_name)

if os.path.exists(path):

dtc = joblib.load(path)

print("Model loaded successfully.")

predict = dtc.predict(X\_test)

calculateMetrics("DTC Model", predict, y\_test)

else:

dtc = DecisionTreeClassifier(max\_depth=3)

dtc.fit(X\_train, y\_train)

joblib.dump(dtc, path)

print("Model saved successfuly.")

predict = dtc.predict(X\_test)

calculateMetrics("DTC Model", predict, y\_test)

root\_feature\_index = dtc.tree\_.feature[0]

root\_feature\_name = feature\_names[root\_feature\_index]

text.insert(END, f"Input Features of DTC Model: {feature\_names}\n")

text.insert(END, f"Root node splits on feature: {root\_feature\_name}\n")

plt.figure(figsize = (15,7))

plot\_tree(dtc,filled = True)

plt.title(f"Internal Architecture of DTC Model on Erythemato Squamous")

plt.show()

def existing\_classifier2():

global X\_train, X\_test, y\_train, y\_test

text.delete('1.0', END)

model = 'model'

model\_name = 'SVM.pkl'

path = os.path.join(model, model\_name)

if os.path.exists(path):

svc = joblib.load(path)

print("Model loaded successfully.")

predict = svc.predict(X\_test)

calculateMetrics("SVM Classifier", predict, y\_test)

else:

svc = SVC(kernel='linear', random\_state=42)

svc.fit(X\_train, y\_train)

joblib.dump(svc, path)

print("Model saved successfuly.")

predict = svc.predict(X\_test)

calculateMetrics("SVM Classifier", predict, y\_test)

def proposed\_classifier3():

global X\_train, X\_test, y\_train, y\_test, mlp

text.delete('1.0', END)

model = 'model'

model\_name = 'MLP.pkl'

path = os.path.join(model, model\_name)

if os.path.exists(path):

mlp = joblib.load(path)

print("Model loaded successfully.")

predict = mlp.predict(X\_test)

calculateMetrics("Proposed DL Classifier", predict, y\_test)

else:

mlp = MLPClassifier()

mlp.fit(X\_train, y\_train)

joblib.dump(mlp, path)

print("Model saved successfuly.")

predict = mlp.predict(X\_test)

calculateMetrics("Proposed DL Classifier", predict, y\_test)

def graph():

#comparison graph between all algorithms

df = pd.DataFrame([['DTC Model','Accuracy',accuracy[0]],['DTC Model','Precision',precision[0]],['DTC Model','Recall',recall[0]],['DTC Model','FSCORE',fscore[0]],

['SVM Classifier','Accuracy',accuracy[1]],['SVM Classifier','Precision',precision[1]],['SVM Classifier','Recall',recall[1]],['SVM Classifier','FSCORE',fscore[1]],

['DL Classifier','Accuracy',accuracy[1]],['DL Classifier','Precision',precision[1]],['DL Classifier','Recall',recall[1]],['DL Classifier','FSCORE',fscore[1]],

],columns=['Parameters','Algorithms','Value'])

df.pivot("Parameters", "Algorithms", "Value").plot(kind='bar', figsize=(8, 4))

plt.title("Performance Evaluation")

plt.xticks(rotation=360)

plt.show()

def Prediction():

global mlp, categories,pca

filename = filedialog.askopenfilename(initialdir="Dataset")

text.delete('1.0', END)

text.insert(END, f'{filename} Loaded\n')

test = pd.read\_csv(filename)

test\_data\_display = test.copy()

predict = mlp.predict(test)

for i, prediction in enumerate(predict):

sample\_data = test\_data\_display.iloc[i]

formatted\_data = ', '.join(f"{col}: {sample\_data[col]}" for col in test\_data\_display.columns)

text.insert(END, f"Features: {formatted\_data}\n")

pred\_label = categories[prediction] # Corrected this line

text.insert(END, f"Test Data {i+1}: {pred\_label}\n\n")

import tkinter as tk

from tkinter import messagebox

import redis

import hashlib

# Connect to Redis

def connect\_redis():

return redis.StrictRedis(host='localhost', port=6379, db=0, decode\_responses=True)

# Hash password before storing in Redis for security

def hash\_password(password):

return hashlib.sha256(password.encode()).hexdigest()

# Signup functionality

def signup(role):

def register\_user():

username = username\_entry.get()

password = password\_entry.get()

if username and password:

try:

conn = connect\_redis()

# Hash the password before storing

hashed\_password = hash\_password(password)

# Store the user in Redis with multiple field-value pairs

user\_key = f"user:{username}"

if conn.exists(user\_key):

messagebox.showerror("Error", "User already exists!")

else:

# Using multiple field-value pairs in hset

conn.hset(user\_key, "username", username)

conn.hset(user\_key, "password", hashed\_password)

conn.hset(user\_key, "role", role)

messagebox.showinfo("Success", f"{role} Signup Successful!")

signup\_window.destroy()

except Exception as e:

messagebox.showerror("Error", f"Redis Error: {e}")

else:

messagebox.showerror("Error", "Please enter all fields!")

# Create the signup window

signup\_window = tk.Toplevel(main)

signup\_window.geometry("400x400")

signup\_window.title(f"{role} Signup")

# Username field

tk.Label(signup\_window, text="Username").pack(pady=5)

username\_entry = tk.Entry(signup\_window)

username\_entry.pack(pady=5)

# Password field

tk.Label(signup\_window, text="Password").pack(pady=5)

password\_entry = tk.Entry(signup\_window, show="\*")

password\_entry.pack(pady=5)

# Signup button

tk.Button(signup\_window, text="Signup", command=register\_user).pack(pady=10)

# Login functionality

def login(role):

def verify\_user():

username = username\_entry.get()

password = password\_entry.get()

if username and password:

try:

conn = connect\_redis()

# Hash the password before checking

hashed\_password = hash\_password(password)

# Check if the user exists in Redis

user\_key = f"user:{username}"

if conn.exists(user\_key):

stored\_password = conn.hget(user\_key, "password")

stored\_role = conn.hget(user\_key, "role")

if stored\_password == hashed\_password and stored\_role == role:

messagebox.showinfo("Success", f"{role} Login Successful!")

login\_window.destroy()

if role == "Admin":

show\_admin\_buttons()

elif role == "User":

show\_user\_buttons()

else:

messagebox.showerror("Error", "Invalid Credentials!")

else:

messagebox.showerror("Error", "User not found!")

except Exception as e:

messagebox.showerror("Error", f"Redis Error: {e}")

else:

messagebox.showerror("Error", "Please enter all fields!")

login\_window = tk.Toplevel(main)

login\_window.geometry("400x300")

login\_window.title(f"{role} Login")

tk.Label(login\_window, text="Username").pack(pady=5)

username\_entry = tk.Entry(login\_window)

username\_entry.pack(pady=5)

tk.Label(login\_window, text="Password").pack(pady=5)

password\_entry = tk.Entry(login\_window, show="\*")

password\_entry.pack(pady=5)

tk.Button(login\_window, text="Login", command=verify\_user).pack(pady=10)

def show\_admin\_buttons():

clear\_buttons()

tk.Button(main, text="Upload Disease Dataset", command=Upload\_Dataset, font=font1).place(x=100, y=200)

tk.Button(main, text="Data Preprocessing", command=Preprocess\_Dataset, font=font1).place(x=280, y=200)

tk.Button(main, text="Data Splitting", command=Train\_Test\_Splitting, font=font1).place(x=500, y=200)

tk.Button(main, text="Build & Trai DTC Model", command=existing\_classifier1, font=font1).place(x=720, y=200)

tk.Button(main, text="Build & Train SVM Classifier", command=existing\_classifier2, font=font1).place(x=810, y=200)

tk.Button(main, text="Build & Train Deep Learning Classifier", command=proposed\_classifier3, font=font1).place(x=910, y=200)

tk.Button(main, text="Performance Graph", command=graph, font=font1).place(x=1150, y=200)

def show\_user\_buttons():

clear\_buttons()

tk.Button(main, text="Prediction on Test Data", command=Prediction, font=font1).place(x=650, y=200)

# Clear buttons before adding new ones

def clear\_buttons():

for widget in main.winfo\_children():

if isinstance(widget, tk.Button) and widget not in [admin\_button, user\_button]:

widget.destroy()

main = tk.Tk()

#main.geometry("1300x1200")

screen\_width = main.winfo\_screenwidth()

screen\_height = main.winfo\_screenheight()

main.geometry(f"{screen\_width}x{screen\_height}")

bg\_image = Image.open("background.jpg")

bg\_image = bg\_image.resize((screen\_width, screen\_height), Image.ANTIALIAS)

bg\_photo = ImageTk.PhotoImage(bg\_image)

canvas = Canvas(main, width=screen\_width, height=screen\_height)

canvas.pack(fill="both", expand=True)

canvas.create\_image(0, 0, image=bg\_photo, anchor="nw")

# Title

font = ('times', 18, 'bold')

title\_text = "Machine Learning-Based Differential Diagnosis of Erythemato Squamous Diseases from Clinical and Microscopic Features"

title = tk.Label(main, text=title\_text, bg='white', fg='black', font=font, wraplength=screen\_width - 200, justify='center')

canvas.create\_window(screen\_width // 2, 50, window=title)

font1 = ('times', 14, 'bold')

# Create text widget and scrollbar

text\_frame = tk.Frame(main, bg='white')

text = tk.Text(text\_frame, height=22, width=130, font=font1, wrap='word')

scroll = tk.Scrollbar(text\_frame, command=text.yview)

text.configure(yscrollcommand=scroll.set)

text.grid(row=0, column=0, sticky='nsew')

scroll.grid(row=0, column=1, sticky='ns')

text\_frame.grid\_rowconfigure(0, weight=1)

text\_frame.grid\_columnconfigure(0, weight=1)

# Position the text\_frame on the canvas, centered horizontally

canvas.create\_window(screen\_width // 2, 300, window=text\_frame, anchor='n')

# Admin and User Buttons

font1 = ('times', 14, 'bold')

tk.Button(main, text="Admin Signup", command=lambda: signup("Admin"), font=font1, width=25, height=2, bg='thistle').place(x=50, y=100)

tk.Button(main, text="User Signup", command=lambda: signup("User"), font=font1, width=25, height=2, bg='thistle').place(x=400, y=100)

admin\_button = tk.Button(main, text="Admin Login", command=lambda: login("Admin"), font=font1, width=25, height=2, bg='lightsteelblue')

admin\_button.place(x=750, y=100)

user\_button = tk.Button(main, text="User Login", command=lambda: login("User"), font=font1, width=25, height=2, bg='lightsteelblue')

user\_button.place(x=1100, y=100)

main.config(bg='lavender')

main.mainloop()

**CHAPTER 9**

**RESULTS AND DISCUSSION**

**9.1 Implementation Description**

This research implements a Tkinter‑based desktop application for differential diagnosis of erythemato‑squamous skin diseases. It supports two roles such as admin and user which are backed by Redis for authentication. Admins can load and preprocess a clinical dataset, train and evaluate three classifiers (DTC model, SVM classifier, and DL classifier), and view performance plots; Users can load new samples and get model predictions.

**1. Key Dependencies**

* Tkinter: GUI framework (windows, dialogs, buttons, canvas).
* Pandas / NumPy: Data loading and manipulation.
* Matplotlib / Seaborn: Plotting histograms, bar charts, heatmaps.
* Scikit‑learn: Preprocessing (LabelEncoder, SMOTE), train/test split, classifiers (DecisionTreeClassifier, SVC, MLPClassifier), metrics.
* imblearn.SMOTE: Balancing imbalanced classes.
* Joblib / Pickle: Persisting trained models under model/.
* Pillow: Loading and displaying background image.
* Redis + hashlib: Storing and verifying user credentials with SHA‑256 hashing.

**2. Global Variables**

* categories: List of six disease class names.
* accuracy, precision, recall, fscore: Lists to accumulate metric values for each algorithm.
* model\_folder: Directory ("model") where serialized models live.

**3. Dataset Workflow**

1. **Upload\_Dataset()**
   * Opens file dialog to select a CSV.
   * Reads with Pandas, shows first few rows in the text widget.
   * Plots a simple bar-chart of class counts.
2. **Preprocess\_Dataset()**
   * Replaces “?” with NaN, drops missing rows.
   * Encodes the textual class column to integers.
   * Splits features (X) and label (y), displays before/after encoding and summary stats.
   * Applies SMOTE to balance the six classes, then plots class distributions before and after resampling (annotated bars).
3. **Train\_Test\_Splitting()**
   * Splits the resampled X\_resampled, y\_after into 80/20 train/test and reports record counts.

**4. Model Training & Evaluation**

* Common helper: calculateMetrics(algorithm, predict, y\_test): Computes accuracy, precision, recall, F1 (macro‑averaged), appends to global lists, prints a full classification report, and shows a confusion‑matrix heatmap.
* existing\_classifier1() → Decision Tree
  + Loads DTC.pkl if present, else trains a DecisionTreeClassifier(max\_depth=3) on (X\_train, y\_train), saves it.
  + Runs prediction on X\_test, calls calculateMetrics().
  + Retrieves and reports the root splitting feature, and plots the full tree.
* existing\_classifier2() → SVM: Similar pattern: load or train a linear‑kernel SVC, then evaluate.
* proposed\_classifier3() → MLP (Deep Learning): Load or train a default MLPClassifier(), then evaluate.
* graph(): Builds a Pandas DataFrame from the accumulated metrics and draws a grouped bar chart comparing Accuracy, Precision, Recall, F‑Score across the three models.

**5. Prediction for New Samples**

* **Prediction()**
  + Lets a User select a CSV of unlabeled samples, loads it, runs the last‑loaded mlp model to predict.
  + Displays each sample’s feature values and its predicted disease label in the text widget.

**6. Role‑Based Authentication (Redis)**

1. connect\_redis() returns a Redis connection on localhost.
2. hash\_password(password) secures passwords with SHA‑256.
3. signup(role) window: prompts username/password, checks Redis for uniqueness, stores a hash and role under key user:<username>.
4. login(role) window: prompts credentials, verifies hash and role, on success destroys login window and calls either show\_admin\_buttons() or show\_user\_buttons().

**7. Dynamic GUI Layout**

* The main window uses a Canvas to display a full‑screen background image and a title.
* A Text widget (with scrollbar) is centered for output logs.
* Initially shows four buttons: Admin Signup, User Signup, Admin Login, User Login.
* On successful login, the other role’s buttons disappear (clear\_buttons()), and:
  + Admin sees buttons to Upload Data, Preprocess, Split, Train each model, and show Performance Graph.
  + User sees only “Prediction on Test Data.”

**8. Application Loop**

* All GUI setup culminates in main.mainloop(), which starts the Tkinter event loop.
* Each button click triggers its associated function, driving the data‑science pipeline or authentication flow.

**9.2 Dataset Description**

**9.3 Results Analysis**

Fig. 9.1 shows the main window of the application immediately upon launch. The full‑screen background image and centered title introduce the diagnostic tool, while the top buttons (“Admin Signup,” “User Signup,” “Admin Login,” “User Login”) indicate the two supported roles. The large text pane at the bottom will display logs, dataset previews, and prediction results.

A close-up of a person's face

AI-generated content may be incorrect.

Fig. 9.1: Proposed desktop application of differencial diagnisis of erythemato-squamous disease from clinical and microscopic features.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

(a) (b)

Fig. 9.2: (a) Admin signup and (b) User signup windows.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer screen

AI-generated content may be incorrect.

Fig. 9.3: Admin login window and login successful confirmation window.

A screenshot of a computer

AI-generated content may be incorrect.

Fig. 9.4: GUI application after login as an admin.

A close-up of a person's face

AI-generated content may be incorrect.

Fig. 9.5: GUI application after performing upload disease dataset.

When “Admin Signup” is clicked, a pop‑up window appears requesting a username and password as illustrated in Fig. 9.2(a). This window is the simple, focused UI for registering new administrators, with basic input validation before credentials are hashed and stored in Redis. In Fig. 9.2(b), the “User Signup” dialog is almost identical in layout, demonstrating reuse of the same signup component for standard users. This ensures non‑admins can register and later access the inference‑only functionality. Here we see the login dialog prompting for credentials and, upon successful authentication, a confirmation messagebox (“Admin Login Successful!”) as shown Fig. 9.3. This handoff clearly transitions the application into Admin mode, dynamically unlocking the full ML pipeline controls.

As shown in Fig. 9.4**,** once authenticated, the top bar replaces signup/login buttons with Admin‑only actions: Upload Dataset, Data Preprocessing, Data Splitting, Build & Train (DTC/SVM/DL), and Performance Graph. The text pane remains ready for detailed status messages. In Fig. 9.5, its showing after selecting a CSV, the first few rows of the DataFrame are printed in the text area, confirming successful loading. This immediate feedback helps the admin verify that the correct file has been selected, and that the data schema matches expectations. Fig. 9.6 is a simple bar chart of the raw counts for each of the six disease classes. This visualization allows the admin to quickly assess class imbalance before applying any balancing technique.

A graph with blue rectangular bars

AI-generated content may be incorrect.

Fig. 9.6: Class distribution graph.

Fig. 9.7 demonstrate a side‑by‑side comparison:

* Left plot: shows original, imbalanced class frequencies.
* Right plot: shows equalized frequencies after SMOTE oversampling.  
  Annotations on each bar display exact counts, confirming that the balancing step has succeeded.

A comparison of a graph

AI-generated content may be incorrect.

Fig. 9.7: Class distribution before and after SMOTE algorithm.

A chart with text on it

AI-generated content may be incorrect.A chart with text on it

AI-generated content may be incorrect.

(a) (b)

A chart with text on it

AI-generated content may be incorrect.

(c)

Fig. 9.8: Confusion matrices obtained using (a) DTC model. (b) SVM classifier. (c) Proposed DL classifier.

Fig. 9.8(a–c) depicts the confusion matrices for each classifier

* (a) Decision Tree: Highlights areas where the tree misclassifies minority classes (zeros along certain rows/columns).
* (b) SVM: Shows near‑perfect diagonals, indicating high precision and recall across most classes.
* (c) DL Classifier: A perfect confusion matrix (all true‑positive entries), confirming 100% accuracy on the test set.

These matrices visually summarize per‑class performance and guide model selection by revealing which approach best separates the six conditions.

Table 1: Overall Performance Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Accuracy (%)** | **Precision (%)** | **Recall (%)** | **F₁‑Score (%)** |
| DTC Model | 65.00 | 55.00 | 67.00 | 57.94 |
| SVM classifier | 98.51 | 98.81 | 98.61 | 98.66 |
| DL classifier | 100.00 | 100.00 | 100.00 | 100.00 |

As demonstrated in Table 1:

* The Decision Tree achieves moderate overall accuracy (65%) but struggles with class balance, yielding a relatively low macro‑averaged precision (55%) and F₁‑score (57.94%).
* The SVM sharply improves all metrics, nearing perfect separation (accuracy 98.5%, F₁ ≈ 98.66%), indicating linear boundaries suffice for most classes.
* The proposed DL model attains perfect scores on this test set (100% across the board), demonstrating its capacity to capture any remaining nonlinear patterns and fully resolve class distinctions.

Table 2: Class‑specific F₁‑score comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Support** | **DTC F₁ (%)** | **SVM F₁ (%)** | **MLP F₁ (%)** |
| Psoriasis | 24 | 96.00 | 100.00 | 100.00 |
| Seboreic Dermatitis | 21 | 0.00 | 100.00 | 100.00 |
| Lichen Planus | 24 | 52.00 | 96.00 | 100.00 |
| Pityriasis Rosea | 26 | 100.00 | 100.00 | 100.00 |
| Chronic Dermatitis | 13 | 100.00 | 100.00 | 100.00 |
| Pityriasis Rubra Pilaris | 26 | 0.00 | 96.00 | 100.00 |

As shown in Table 2:

* Psoriasis, Pityriasis Rosea, and Chronic Dermatitis are well handled by all three models—the tree already achieves perfect recall & F₁ for the latter two, and near‑perfect for psoriasis.
* Seboreic Dermatitis and Pityriasis Rubra Pilaris cause the Decision Tree severe trouble (F₁ = 0%), indicating it never correctly predicts those classes; both SVM and MLP rectify this completely.
* Lichen Planus sees a jump from moderate DTC performance (F₁ ≈ 52%) to very high SVM performance (96%), and full resolution under MLP.
* Overall, the class‑specific table highlights the tree’s inability to capture certain minority‑class patterns, while the SVM largely succeeds and the MLP achieves perfect distinction across all six categories.

A close-up of a person's face

AI-generated content may be incorrect.

A close-up of a person's face

AI-generated content may be incorrect.

Fig. 9.10: Sample predictions on new test data.

Fig. 9.10 shows the inference output after the User selects an unlabeled CSV of clinical and microscopic measurements. Each test record is listed with its full set of feature values (e.g., erythema level, scale thickness, cell morphology scores), followed immediately by the model’s predicted disease label. By printing both the raw inputs and the inferred class side by side, the application provides transparent, traceable diagnostic suggestions. Clinicians can quickly verify which specific measurements drove each prediction, facilitating trust and enabling rapid follow‑up decisions.

**CHAPTER 10**

**CONCLUSION AND FUTURE SCOPE**

This project has demonstrated the development of a comprehensive, user‑friendly desktop application for the differential diagnosis of erythemato‑squamous skin diseases, integrating a full machine‑learning pipeline within a Tkinter GUI and secured by a lightweight Redis‑based authentication system. Beginning with raw clinical and microscopic feature data, the preprocessing module—consisting of missing‑value handling, label encoding, and SMOTE resampling—ensured a clean and balanced dataset across six disease categories. An 80/20 train/test split then prepared the data for model training. Three classifiers such as DTC model, SVM classifier, and DL classifier were sequentially trained, serialized, and evaluated. The DTC model offered interpretability through its explicit hierarchical splits yet exhibited uneven class performance (macro‑F₁ ≈ 57.9%), particularly struggling with minority classes. The SVM markedly improved on this, achieving near‑perfect metrics (accuracy ≈ 98.5%, macro‑F₁ ≈ 98.7%) by leveraging a linear decision boundary that separated the six classes effectively. The proposed DL classifier ultimately achieved perfect performance on the held‑out test set (100% accuracy, precision, recall, and F₁‑score), indicating its capacity to capture any residual nonlinearity. Beyond raw metrics, the application’s visual outputs—bar charts of class distributions, confusion‑matrix heatmaps, and comparative performance plots—enhanced transparency and aided in interpreting model behavior. The role‑based GUI design cleanly separated Admin functions (data ingestion, preprocessing, model training, and evaluation) from User functions (inference on new samples), preventing inadvertent retraining and streamlining workflow for nontechnical users. Redis provided rapid, secure credential storage with SHA‑256 hashing, and Joblib–serialized models supported efficient reuse without redundant computation.

**Future Scope**

– Extend the pipeline to incorporate additional disease categories and multimodal data (e.g., image inputs).

– Integrate explainable AI techniques (e.g., SHAP values) to provide clinicians with feature‐level insights alongside predictions.

– Deploy as a web or mobile application with cloud‐based model hosting for broader accessibility.

**REFERENCES**

1. A. M. Elsayad, M. Al-Dhaifallah, A. M. Nassef, Analysis and Diagnosis of Erythemato-Squamous Diseases Using CHAID Decision Trees, in: 15th International Multi-Conference on Systems, Signals and Devices (SSD), IEEE, 2018. doi:10.1109/SSD.2018.8570553.
2. G. Demiroz, H. A. Govenir, N. Ilter, Learning Di erential Diagnosis of Eryhemato-Squamous Diseases using Voting Feature Intervals, Aritificial Intelli gence in Medicine 13 (1998) 147–165.
3. M. E. B. Menai, N. Altayash, Di erential Diagnosis of Erythemato-Squamous Diseases Using Ensemble of Decision Trees, in: Modern Advances in Applied Intelligence, 2014, pp. 369–377.
4. M. A. Alshamrani et al. "Deep Learning for Skin Disease Diagnosis: A Survey," *IEEE Access*, vol. 9, pp. 40771-40784, 2021.
5. H. Xie et al. "A Convolutional Neural Network for the Classification of Skin Disease Images," *IEEE Transactions on Biomedical Engineering*, vol. 67, no. 10, pp. 2920-2929, 2020.
6. K. K. Y. Wong et al. "Automatic Detection of Skin Cancer Using Deep Convolutional Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 4, pp. 1238-1246, 2020.
7. S. T. Ghosh et al. "An Intelligent System for Automated Diagnosis of Skin Diseases Using Deep Learning," *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1-10, 2021.
8. P. Kumar et al. "Dermatological Disease Diagnosis Using Multi-Modal Deep Learning Networks," *IEEE Transactions on Medical Imaging*, vol. 39, no. 9, pp. 3023-3034, 2020.
9. R. K. Sharma et al. "Machine Learning Approaches for the Classification of Skin Lesions," *IEEE Reviews in Biomedical Engineering*, vol. 14, pp. 24-34, 2021.
10. L. Lee et al. "Hybrid Deep Learning Models for Accurate Skin Disease Classification," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 14, no. 2, pp. 274-283, 2020.
11. L. Zhao et al. "Skin Disease Diagnosis with Generative Adversarial Networks and Convolutional Neural Networks," *IEEE Transactions on Computational Biology and Bioinformatics*, vol. 17, no. 4, pp. 1134-1143, 2020.
12. A. R. Das et al. "Advanced Techniques for Skin Disease Detection Using Machine Learning," *IEEE Access*, vol. 8, pp. 106469-106480, 2020.
13. J. D. Smith et al. "Application of Deep Learning in Dermatology: A Systematic Review," *IEEE Transactions on Health Informatics*, vol. 26, no. 5, pp. 2062-2071, 2022.
14. J. R. W. Johnson et al. "Machine Learning Techniques for Automated Diagnosis of Dermatological Disorders," *Journal of Dermatology Research*, vol. 5, no. 2, pp. 67-79, 2021.
15. T. H. Li et al. "Deep Learning-Based Diagnostic System for Erythemato-Squamous Diseases," *International Journal of Biomedical Imaging*, vol. 2021, Article ID 6743407.
16. A. M. Collins et al. "Predictive Modeling for Skin Disease Classification Using Machine Learning Algorithms," *Computers in Biology and Medicine*, vol. 129, Article 104110, 2020.
17. E. F. Williams et al. "Utilizing Convolutional Neural Networks for Automated Diagnosis of Dermatological Conditions," *Biomedical Engineering Letters*, vol. 10, no. 1, pp. 45-54, 2020.
18. R. S. Patel et al. "Integration of Machine Learning in Dermatological Diagnostic Processes: Challenges and Prospects," *Dermatology Science Journal*, vol. 6, no. 3, pp. 133-144, 2021.