Parasite Image Classification using Machine Learning

Assessment Task 2,
42177 Image Processing and Pattern Recognition (Spring 2025)

Team Name **Group 11**

Team Members **Muhib Al Muquit**
Sheikh Wasif
Shuntian Shi
Xiaoling Tong
Xinyi Dai
Zhengjie Chen

Date **31/10/2025**

Disclaimer

This project is for academic purposes only and is submitted as part of the 42177 Image Processing and Pattern Recognition course. The work presented in this report is entirely the product of the team's collaborative efforts. While every attempt has been made to ensure the accuracy and validity of the methods and results, any errors or inconsistencies are the responsibility of the team. All data and models used in this project are for educational purposes only and are not intended for real-world application without further validation

Github repo of this project: https://github.com/XLingTong/42177-parasite-classification

Contents

Table of Figures

1. Executive Summary

Accurate parasite identification supports timely treatment and public-health surveillance. Field images from smartphone microscopes are common and often degraded by blur, noise, low resolution, or compression, so models must be both accurate and robust.

This project had two objectives: (1) implement and compare classical and deep learning classifiers for multi-class parasite microscopy; and (2) evaluate robustness under smartphone-like degradations applied to validation/test only. We used a single primary dataset (Mendeley "Microscopic Images of Parasites Species", v3; six parasite classes), a manifest-driven stratified split (60/20/20), uniform 256×256 resizing and normalisation, and a common evaluation protocol (Accuracy, macro-Precision/Recall/F1, confusion matrices).

The value of our approach is twofold. First, it delivers a fair, reproducible benchmark across seven models. The CNN achieved the highest clean-set performance (98.59% accuracy; 98.08% F1). Among classical baselines, XGBoost and Random Forest were strongest. Second, it tests practical resilience: most models were stable under blur, resolution loss, and JPEG; however, the CNN was highly sensitive to additive Gaussian noise without noise-aware training, while classical models remained steady, offering a low-cost, low-variance option on noisy inputs.

This report is organised as follows: Section 2 introduces the problem, motivation, and scope. Section 3 details the methodology, including environment, dataset, preprocessing, model training, and the universal evaluation protocol. Section 4 presents results on clean and degraded tests, plus efficiency and cost. Section 5 discusses goal alignment, deviations from plan, challenges, lessons learnt, and future work. Section 6 concludes with key findings and practical recommendations.

2. Introduction to the Problem

Accurate and fast parasite identification helped clinicians treat patients on time and supported surveillance in the community. The World Health Organization advised “test before treat” for suspected malaria using microscopy or rapid tests because correct diagnosis improved care and enabled monitoring (WHO, 2015). This underscored the value of dependable parasite recognition tools, especially where specialist expertise was limited. With the rising feasibility of smartphone microscopes, robustness became essential: these devices are portable and low cost, and recent research highlighted their suitability for biomedical applications, including pathogen detection (Raju et al., 2024). Field evaluations of smartphone-based malaria screening systems also reported promising patient-level performance in resource-limited settings (Yu et al., 2023). Because many images could originate from phones, our models needed to remain accurate when image quality dropped. Prior studies showed that common degradations such as blur, noise, and compression reduce image-classification performance, reinforcing the need to evaluate models beyond clean laboratory conditions (Hendrycks & Dietterich, 2019).

Problem definition and scope

This project has two objectives:

1. implement and compare multiple classification methods for parasite microscope images; and
2. simulate lower-quality, smartphone-style images by reducing resolution, adding noise, and introducing blur to mimic real-world captures and test model robustness.

The project is multi-class image classification on still micrographs. Detection, localisation, and segmentation are out of scope. Video is out of scope. The work includes data preparation, class definition, model training, validation, testing, and robustness assessment under controlled degradations.

Simulation of real-world conditions

A key aspect of the project was the planned simulation of real-world imaging to evaluate robustness. While the dataset provided high-quality microscope images, such conditions are not guaranteed in field settings where smartphone-assisted microscopes are increasingly used. To replicate realistic conditions, we generated degraded versions of the validation and test sets while keeping training data clean. Four artefact families were introduced: resolution loss (downsampling and upsampling), blur (Gaussian and short motion blur), noise (Gaussian and Poisson to reflect low-light acquisition and sensor limits), and compression (JPEG re-encoding at reduced quality). Each artefact was applied at preset severities to produce a controlled gradient of image quality. Model performance was then compared across clean and degraded conditions, with macro-F1 differences serving as the primary measure of robustness. This ensured that the evaluation reflected practical usage scenarios and addressed requirements for accessible, low-cost diagnostics in peripheral clinics and field programs.

3. Overview of the Project and Methodology

The goal of this project is to conduct an experimental analysis and comparative case study to evaluate the performance of various machine learning models in the task of classifying parasite species from a set of labeled images. By experimenting with different techniques, ranging from traditional classifiers like Logistic Regression (LogReg), k-Nearest Neighbors (kNN), and Support Vector Machines (SVM) to more advanced models like Convolutional Neural Networks (CNN) and XGBoost (XGB), we aim to compare their effectiveness in handling this image classification challenge.

The project also seeks to assess how these models perform on both clean and degraded datasets, providing a comprehensive analysis of their robustness in real-world scenarios.

This section will describe the methodology used to approach the problem, starting with an overview of the dataset and the preprocessing steps applied to standardize the data. The different machine learning models utilized for the analysis will be detailed, highlighting the techniques and strategies for each. Furthermore, we will outline the evaluation methodology, including the performance metrics used to compare the models, and discuss the tools and resources that supported the implementation and collaboration throughout the project.

3.1 Implementation Environment

All experiments were conducted using a combination of **MATLAB**, **Python**, and **Jupyter Notebook**, selected for their complementary strengths in image processing, machine learning, and reproducible research. MATLAB was primarily used for dataset preprocessing and degradation generation, while Python was used for model training, evaluation, and visualisation, with Jupyter Notebook providing an interactive environment for experiment documentation and iteration.

Software and Tools

- **MATLAB R2023b** – used for image preprocessing, dataset management, and degradation simulation, with the following toolboxes:
  - *Image Processing Toolbox* – for operations such as resizing, normalisation, filtering, and noise addition.
  - *Computer Vision Toolbox* – for implementing degradation techniques including motion blur, resolution loss, and Gaussian blur.
  - *Deep Learning Toolbox* – used for building, training, and testing the Convolutional Neural Network (CNN) model, including layer definitions, optimization, and evaluation.
- **Python 3.12.4** – used for training and evaluating models, leveraging the following major libraries:
  - *scikit-learn* – for implementing traditional machine learning models such as Logistic Regression, kNN, Decision Tree, Random Forest, and SVM.
  - *XGBoost* – for training the XGBoost classifier and optimising its performance.
  - *NumPy, Pandas, Matplotlib, and Seaborn* – for data analysis, metrics computation, and visualisation of model results.

Hardware Specifications

All preprocessing and training tasks were executed on a workstation with the following specifications (main workstation):

- Processor: 13th Gen Intel® Core™ i5-13400 (2.50 GHz)
- Installed RAM: 16.0 GB (15.8 GB usable)
- Graphics Card: Intel® UHD Graphics 730
- Storage: 500GB SSD
- Operating System: Windows 11 Pro (64-bit)

3.2 Project Management and Teamwork

Team Structure

Team Member	Contribution
Muhib Al Muquit	Logistic Regression
Sheikh Wasif	k-Nearest Neighbours (kNN)
Shuntian Shi	Decision Trees and Random Forests
Xiaoling Tong	Convolutional Neural Network (CNN), data management, data preprocessing, evaluation
Xinyi Dai	XGBoost
Zhengjie Chen	Support Vector Machine (SVM)

Data and Resource Management

A centralized preprocessing pipeline was developed using MATLAB and Python, ensuring that all models used the same standardized dataset.

The cleaned and degraded datasets were stored on a shared Google Drive for backup and accessibility. Each model script was configured to read from the same manifest file (manifest.csv), maintaining data consistency across experiments.

We used a shared GitHub repository [1] to facilitate collaboration and version control. This repository became the central hub for all project-related resources, including:

- **Data**: Cleaned and degraded datasets, which were made accessible via Google Drive and referenced consistently within the repository.
- **Pipelines**: Data preprocessing scripts, model training scripts, and evaluation code.
- **Manifests**: The manifest.csv file was shared and used across all models to maintain consistency in data splitting and labelling.

The GitHub repository was organized with clear subfolders for data, scripts, and results.

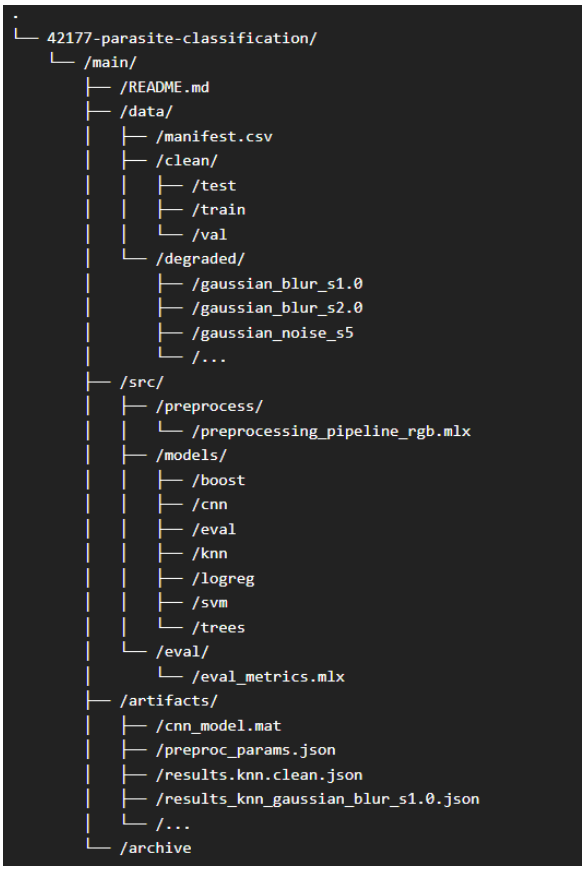


Figure 1 directory tree of github repository

Consistency and Integration

To ensure comparability, all team members followed a unified evaluation framework. A common function (evaluate_model) was defined to calculate accuracy, precision, recall, F1-score, and confusion matrices. Each member reported results in the same format (.json/.csv), which were later consolidated using a shared visualization notebook for performance comparison. We had one team member dedicated to ensure consistency across models.

Communication and Collaboration

Microsoft Teams were used for daily communication, ensuring rapid response to any issues that arose. These tools facilitated real-time feedback on model performance, troubleshooting, and clarifications on tasks. For shared file backup and collaborative editing, Google Drive was used, enabling team members to access and archive files in real-time.

In case of technical issues or questions regarding code or results, issues were communicated on Github, ensuring that all problems were addressed promptly and that solutions were tracked for future reference.

[1] The Github repo can be access here: https://github.com/XLingTong/42177-parasite-classification

Teamwork Challenges and Solutions

One challenge we faced was ensuring consistent data preprocessing across the Python and MATLAB environments. Each team member worked with different tools: some used Python (with libraries like scikit-learn and OpenCV), while others used MATLAB for convenient deep learning toolboxes. This created a risk of discrepancies in how data was handled, particularly in the preprocessing stages (e.g., image resizing, normalization, and augmentation).

To address this, we defined a unified preprocessing pipeline that was applied consistently across both environments. For Python, we standardized the preprocessing steps using scripts that followed the same logic as the MATLAB functions. In MATLAB, we ensured that preprocessing steps like resizing and normalization followed the same parameters.

Additionally, we used a shared manifest file to synchronize dataset splits and labels across all models. This approach minimized any inconsistencies and allowed for a smooth comparison of results across the different tools.

3.3 Dataset

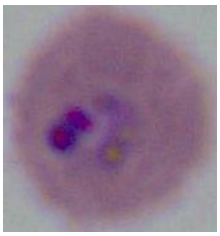
For this project, we used the **Microscopic Images of Parasites Species (Mendeley Data, v3)** dataset (Li & Zhang, 2020), which consists of 34,298 microscopic images across six parasite groups and two host-cell classes. The images were captured at 400× and 1000× magnification, with heterogeneous staining, focus, illumination, and compression. The dataset is licensed under CC BY 4.0, making it publicly available for research.

In our initial specification, the plan was to use two datasets: Mendeley Data (v3) as Dataset-A, and the Image Bank of Parasitology (Institute, S. I. D., 2018) as Dataset-B. The Image Bank of Parasitology dataset was originally considered as a secondary dataset, consisting of 377 images. However, this dataset was ultimately abandoned due to its small size, which was insufficient for training deep learning models. The dataset also had a long-tail class distribution and inconsistent image quality, with varying focus, illumination, and contrast typical of archival material. While it could be useful for stress-testing and cross-dataset generalisation, it was deemed less suitable for our needs.

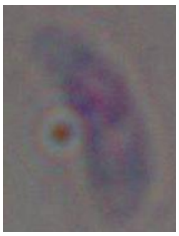
The dataset used for this project contains the following parasite classes and their respective image counts:

- **Plasmodium**: 843 images
- **Toxoplasma gondii**: 6,691 images (2,933 at 1000× and 3,758 at 400× magnification)
- **Babesia**: 1,173 images
- **Leishmania**: 2,701 images
- **Trypanosome**: 2,385 images
- **Trichomonad**: 10,134 images

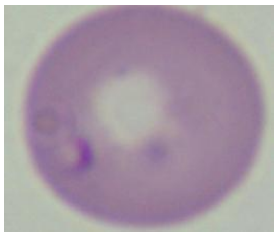
Sample images from each class:



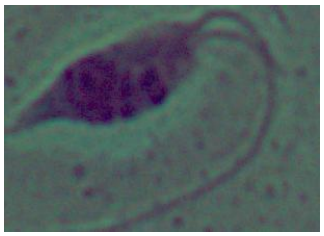
Plasmodium



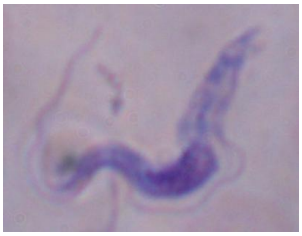
Toxoplasma gondii



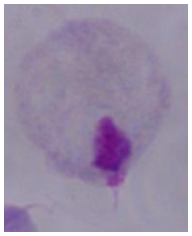
Babesia



Leishmania



Trypanosome



Trichomonad

Additionally, the dataset includes two host-cell classes:

- **Red blood cell**: 8,995 images
- **Leukocyte**: 1,376 images (461 at 1000× and 915 at 400× magnification)

Class Design

The primary task of this project is parasite classification, where we focus on the six parasite classes mentioned above. During preprocessing, the **host-cell classes** were excluded from the dataset to reduce computational cost and complexity.

3.4 Data Preprocessing

A centralized preprocessing pipeline was developed using MATLAB to ensure that all models used the same standardized dataset. Key toolboxes used include the Image Processing Toolbox for image manipulation (e.g., resizing, normalizing, and filtering) and the Computer Vision Toolbox for advanced image processing functions like motion blur and resolution loss. This approach facilitated uniform data handling, ensuring consistency across different machine learning models and enabling a fair comparison of their performances.

The preprocessing pipeline involved several key stages:

1. **Dataset Preparation**:
   The raw images were scanned from the specified directory and organized into a table containing image paths and their corresponding labels. The dataset was then stratified

into **train**, **validation**, and **test** subsets using a **60/20/20** split. This stratified approach ensured that each class was well-represented in all subsets, reducing any potential bias in training.

Below is the distribution of the parasite classes across the subsets:

Subset	Label	Count
test	babesia	236
test	leishmania	541
test	plasmodium	170
test	toxoplasma1000x	588
test	toxoplasma400x	753
test	trichomonad	2028
test	trypanosome	477
train	babesia	703
train	leishmania	1620
train	plasmodium	505
train	toxoplasma1000x	1759
train	toxoplasma400x	2254
train	trichomonad	6080
train	trypanosome	1431
validation	babesia	234
validation	leishmania	540
validation	plasmodium	168
validation	toxoplasma1000x	586
validation	toxoplasma400x	751
validation	trichomonad	2026
validation	trypanosome	477

2. **Resizing and Normalizing Images**:
   Each image was **resized** to a uniform size of **256 x 256 pixels** to ensure consistency in input dimensions across all models.

   The pixel values were **normalized** to the range [0, 1] using the im2double function. This step was crucial for standardizing the image data, which aids in faster convergence and better model performance during training.

3. **Data Degradation**:
   To test the robustness of the models, degraded versions of the images were generated for the **validation** and **test** subsets. The following degradation techniques were applied to simulate real-world variations and assess how well the models handle imperfect data:

   Resolution Loss (Down-sampling and Up-sampling):

Down-sampling reduces the resolution of the image by a factor of f (e.g., 2x, 4x), which simulates lower-resolution images. These images are then up-sampled back to the original size using bilinear interpolation, which helps the model adapt to images that might have a lower quality due to compression or poor resolution.
For example:

- Down-sampling by a factor of 2 and up-sampling back to the original size.
- Down-sampling by a factor of 4 and up-sampling back to the original size.

Gaussian Blur:
Gaussian blur is applied to simulate the effect of out-of-focus images or motion blur. The level of blur is controlled by the sigma (s) value, which determines the extent of the blur. Two levels of blur were applied:

- s = 1.0: A moderate blur that simulates slight focus issues.
- s = 2.0: A stronger blur effect that represents more significant focus problems or slight motion blur.

Example: A Gaussian kernel is used to blur the image, where the kernel size is determined by the sigma value. The larger the sigma, the more blurry the image becomes.

Motion Blur:
Motion blur is used to simulate the effect of moving objects or camera shake, creating a linear blur in a specific direction. The kernel length controls the intensity of the blur, with a longer kernel resulting in a more pronounced blur.
For example:

- k = 5: A moderate motion blur, simulating a fast-moving object.
- k = 10: A longer blur kernel, simulating a slower or more sustained motion.

Gaussian Noise:
Gaussian noise is added to the image to simulate environmental noise, such as sensor noise, which can occur in low-light conditions or due to compression. The noise is controlled by its standard deviation (s), with a higher value introducing more visible noise. Two levels of noise were applied:

- s = 5: Low-intensity noise, simulating mild sensor noise.
- s = 15: Higher-intensity noise, representing significant environmental noise or distortion.

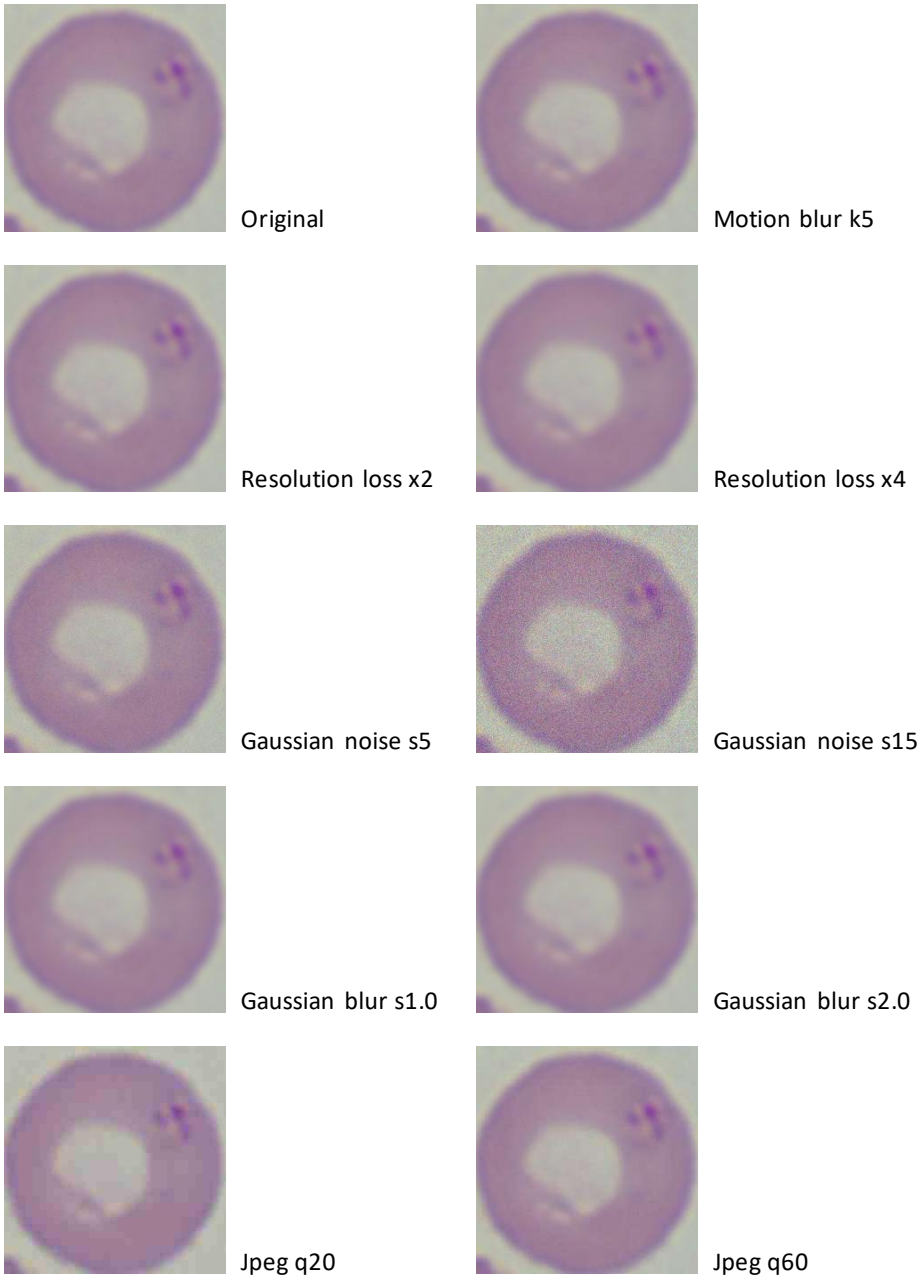
JPEG Compression:
JPEG compression introduces artefacts and quality degradation due to compression, commonly seen in web images or low-bitrate images.
Three different quality levels were applied:

- Quality = 60: A lower-quality image, with significant compression artefacts.
- Quality = 40: A much more compressed image, with visible artefacts.
- Quality = 20: Extremely low-quality image with severe compression artefacts, often resulting in visible pixelation.

Below are some samples of 'babesia\img_0000003' after degradation (Figure 2). The difference between the degraded images and the original ones is quite subtle to the

human eye, with the distortions introduced by the degradation techniques being minimal but still present at a computational level.

Figure 2 Samples of 'babesia\img_0000003' after degradation



4. **Manifest and Preprocessing Parameters**:
   A manifest CSV file was generated, containing the paths, labels, and subset assignments (train, validation, test). This file served as a reference for later stages of the project.

The preprocessing parameters (such as target image size, colour mode, and normalization method) were saved in a JSON file to ensure reproducibility and transparency of the preprocessing pipeline.

5. **File Saving and Organization**:
   The processed images were saved in an organized directory structure, separated by subset and label. This made the dataset easy to access for training and evaluation purposes.
   The augmented (degraded) images were saved in separate directories according to the type of degradation applied, such as resolution loss, Gaussian blur, or noise.

Justification of Methods

The preprocessing steps were carefully selected to ensure consistency and efficiency in handling the data. Resizing images ensured a uniform input size, while normalization standardized pixel values, promoting faster and more stable model training. Data augmentation enhanced model generalization by introducing variability in the data, helping the model adapt to real-world scenarios.

Data Sharing and Saving Outcomes

The processed images and associated metadata were saved in clearly structured directories. The manifest file and preprocessing parameters were stored in accessible locations, with the entire dataset backed up on Google Drive for safe keeping and easy access when needed. Additionally, all processed images were stored on GitHub, where they were organized by type of processing, ensuring that team members could easily access and share results across experiments.

3.5 Model Training and Evaluation

Classic Machine Learning Models

The following traditional machine learning models were implemented:

- **Logistic Regression (LogReg)**
- **k-Nearest Neighbors (kNN)**
- **Decision Trees (Tree)**
- **Random Forests (RF)**
- **Support Vector Machines (SVM)**
- **XGBoost (XGB)**

We adjusted several pipelines but most of them hit the RAM limit. The pipeline that turned out to be working was implemented to operate under low-RAM constraints by streaming images in batches, fitting a global StandardScaler and an IncrementalPCA (IPCA) on the training set, transforming all splits into compact features, training models on those features, and evaluating on clean and multiple degraded test conditions.

PCA was applied to reduce the dimensionality of the data and decrease computational complexity. By retaining the most important features that captured the maximum variance, PCA ensured faster training times while preserving critical information for model learning.

Diagram (Figure 3) below illustrates the workflow of our training pipeline design:

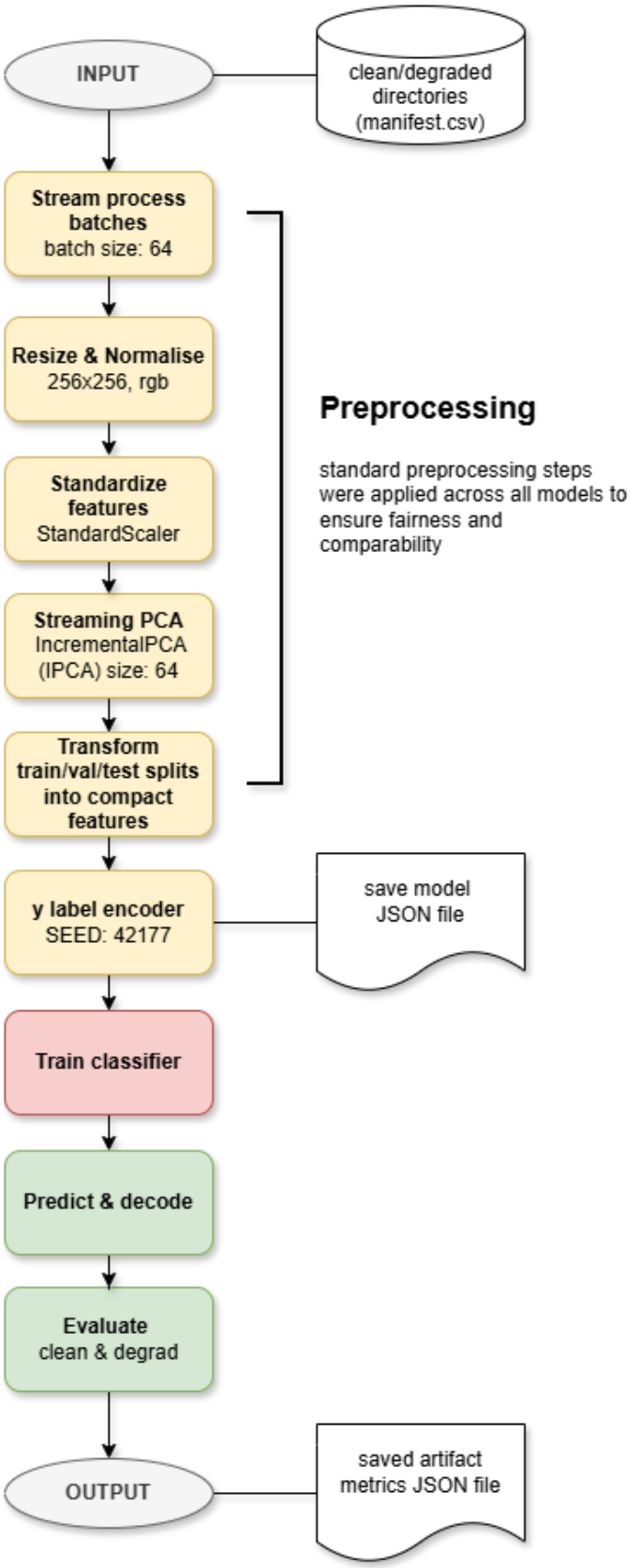


Figure 3 classic models pipeline workflow

The table below summarises the key parameters and example of output produced by each classic classifier evaluated in this experiment:

Classifier	Key hyperparameters	Artifacts produced
Logistic Regression (LogReg)	n_estimators=400; max_depth=8; lr=0.05; subsample=0.9; colsample_bytree=0.9; seed=42177	results_xgb_clean.json; results_xgb_.json; xgb_model.json; scaler.pkl; ipca.pkl
KNeighborsClassifier (kNN)	n_neighbors=5; weights="distance"; metric="euclidean"; n_jobs=-1	results_knn_clean.json; results_knn_.json; scaler.pkl; ipca.pkl
LinearSVC (SVM)	C=1.0; loss="squared_hinge"; dual=False; max_iter=2000	results_svm_clean.json; results_svm_.json; svm.pkl; scaler.pkl; ipca.pkl
DecisionTreeClassifier; RandomForestClassifier	DT: criterion="gini"; RF: n_estimators=200; n_jobs=-1; seed=42177	results_tree_clean.json; results_tree_.json; results_rf_clean.json; results_rf_.json; models + scaler/ipca
XGBoost (multi:softmax)	n_estimators=400; max_depth=8; lr=0.05; subsample=0.9; colsample_bytree=0.9; seed=42177	results_xgb_clean.json; results_xgb_.json; xgb_model.json; scaler.pkl; ipca.pkl

Deep Learning

The deep learning approach used a lightweight convolutional neural network (CNN) implemented in MATLAB to classify preprocessed images. The workflow comprised path initialisation, reproducible data loading, training-only augmentation, a compact three-block architecture with global average pooling, Adam optimisation, and a standardised evaluation pipeline.

Parameters and Defaults Summary

Parameter	Default/value
Image Input Size	[256, 256, 3]
RNG Seed	42177 as preset
Data Augmentation	Horizontal reflection and random rotation in the range [-10, 10] degrees (only for training)
Optimizer	Adam with MiniBatchSize = 64, MaxEpochs = 4, InitialLearnRate = 1e-3

Artifacts Saved	cnn_model.mat (network, classes, training info) and results_cnn_clean.mat (evaluation results)
Datastores	imdsTrain, imdsVal, imdsTest; augmented datastores: adsTrain, adsVal, adsTest

Data access and reproducibility

- Paths to the clean dataset, artefact directory, and evaluation script were initialised; the evaluation folder was added to the MATLAB path to expose the evaluation function.
- A fixed random seed ensured reproducibility of weight initialisation, data shuffling, and augmentation.
- Train, validation, and test sets were loaded via imageDatastore(..., 'IncludeSubfolders', true, 'LabelSource', 'foldernames'), with labels extracted from imdsTrain. The input size was set to [256 256 3].

Screenshot 1: datastore construction and RNG setup (MATLAB).

```
% Seed for reproducibility
rng(42177);

% Datastores
imdsTrain = imageDatastore(fullfile(rootClean,'train'),"IncludeSubfolders",true,"LabelSource","foldernames");
imdsVal   = imageDatastore(fullfile(rootClean,'val'),  "IncludeSubfolders",true,"LabelSource","foldernames");
imdsTest  = imageDatastore(fullfile(rootClean,'test'), "IncludeSubfolders",true,"LabelSource","foldernames");

classes   = categories(imdsTrain.Labels);
inputSize = [256 256 3];
```

Data augmentation

- Augmentation was applied to the training stream only using imageDataAugmenter with horizontal reflection and random rotation in the range [−10, 10] degrees.
- augmentedImageDatastore produced adsTrain with augmentation and adsVal and adsTest without augmentation.

Screenshot 2: augmentation configuration and augmentedImageDatastore creation.

```
% Data augmentation
aug = imageDataAugmenter('RandXReflection',true,'RandRotation',[-10 10]);
adsTrain = augmentedImageDatastore(inputSize, imdsTrain, 'DataAugmentation', aug);
adsVal   = augmentedImageDatastore(inputSize, imdsVal);
adsTest  = augmentedImageDatastore(inputSize, imdsTest);
```

Network architecture.

- The network consisted of an input layer, three convolutional blocks with batch normalisation and ReLU activations, two max-pooling stages, a final convolutional block, global average pooling, a fully connected layer to the number of classes, softmax, and a classification layer.
- Input normalisation inside the network was disabled because images were already scaled during preprocessing.

Screenshot 3: layer definition.

Define the convolutional neural network architecture

```
% MODEL
layers = [
    imageInputLayer(inputSize,"Normalization","none")
    convolution2dLayer(3,32,"Padding","same")
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,64,"Padding","same")
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,128,"Padding","same")
    batchNormalizationLayer
    reluLayer

    globalAveragePooling2dLayer
    fullyConnectedLayer(numel(classes))
    softmaxLayer
    classificationLayer
];
```

Training configuration.

- Optimiser: Adam.
- Mini-batch size: 64.
- Max epochs: 4 (set due to compute constraints).
- Initial learning rate: 1e−3.
- Validation data: adsVal.
- Shuffling: every epoch.
- Verbose output and training-progress plots enabled.

Model training and persistence.

- The model was trained with trainNetwork(adsTrain, layers, options). Wall-clock training time was recorded.
- The trained network, class list, training log, and elapsed time were saved as cnn_model.mat in the artefact directory.

Screenshot 5: training invocation and artefact saving.

```
%% Train
t0 = tic;
[net, trainInfo] = trainNetwork(adsTrain, layers, options);
trainSecs = toc(t0);
fprintf('Training time: %.1f min\n', trainSecs/60);

% Save model
save(fullfile(artifactsDir,'cnn_model.mat'),'net','classes','trainInfo','trainSecs');
```

Figure 4 in next page shows the progress of training.

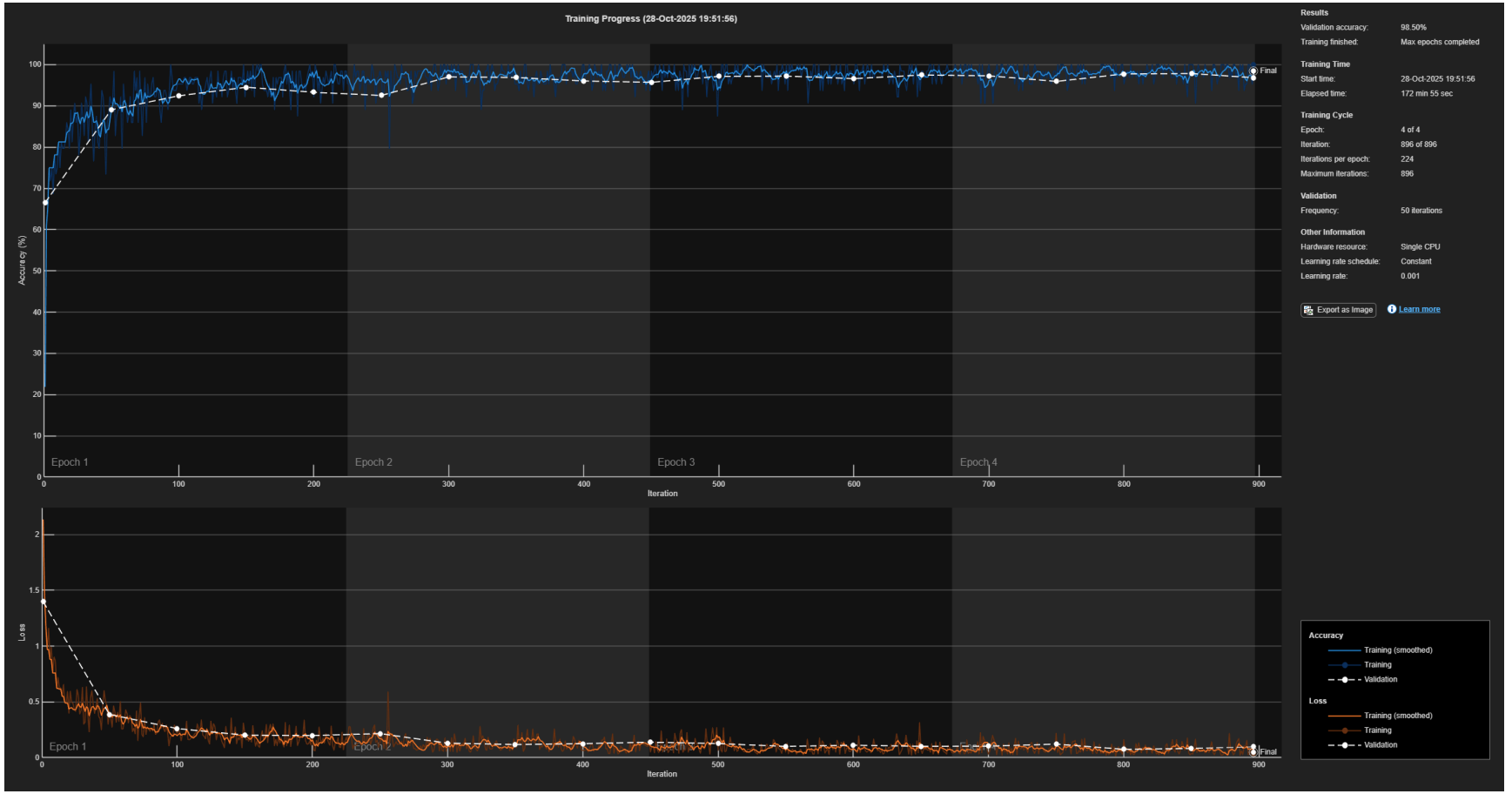
Inference on the held-out test set.

- A test datastore was recreated without augmentation.
- Predictions were obtained via classify(net, adsTest), and test-time inference duration was recorded.

Screenshot 6: test datastore recreation and classification.

```
%% Load trained model
load('D:\LocalUser\42177 Project\artifacts\cnn_model.mat', 'net', 'classes');

%% Recreate datastores for test set
rootClean = 'D:\LocalUser\42177 Project\data\clean';
imdsTest  = imageDatastore(fullfile(rootClean,'test'), ...
    "IncludeSubfolders", true, "LabelSource", "foldernames");
adsTest   = augmentedImageDatastore([256 256 3], imdsTest);
```

Figure 4 screenshot of CNN model training progress



Evaluation

All models followed the **same evaluation protocol** (confusion matrix with fixed label order; Accuracy; macro-Precision; macro-Recall; macro-F1), but the implementations differed by environment:

- CNN (MATLAB): evaluated with the shared MATLAB function.
- Classical models (Python): evaluated with an equivalent Python implementation that mirrors the MATLAB logic (fixed labels order; identical macro averaging and formulas).

The clean test and all degraded tests were evaluated using exactly the same methods and protocol.

The evaluation approaches to test:

1. *Predictive performance.* To evaluate core classification quality under class imbalance, we reported **Accuracy, macro-Precision, macro-Recall, and macro-F1**, with a **confusion matrix** to expose class-wise errors that averages can hide.
2. *Comparability across models.* To ensure like-for-like results, we enforced **a fixed label order** from the manifest for all confusion matrices and metrics, preventing label-mapping drift between MATLAB and Python pipelines.
3. *Efficiency (compute cost).* To assess training and deployment cost, we recorded **training time and clean-test inference time (wall-clock seconds),** providing a basis for comparing iteration speed and expected latency.
4. *Robustness to real-world artefacts*. To measure sensitivity to image degradations, we evaluated each model on degraded conditions (same evaluator, same label order) and compared against clean using the same metrics; ΔF1 relative to clean was used as the primary robustness indicator.
5. *Scalability and practicality.* To reflect memory constraints typical of field pipelines, degraded-set inference was executed in streamed batches and aggregated per condition, demonstrating throughput under realistic resource limits.
6. *Reproducibility and audit.* To enable re-runs and external audit, results were saved per model and per condition (JSON for Python scripts; .mat for the MATLAB CNN), together with elapsed seconds for timing analyses.

A single protocol with fixed label order ensures like-for-like comparability across environments and models, prevents label-mapping drift, and standardises macro-averaging, appropriate under class imbalance where accuracy alone can be misleading. Recording both training time and inference time captures the practical cost profile (iteration speed and deployment latency), while streaming degraded inference avoids memory pressure and enables robustness comparisons that reflect real-world throughput constraints.

4. Results

The results of the models were evaluated using these performance metrics: Accuracy, Precision, Recall, F1-score, and confusion matrix. Efficiency was measured with elapsed time. These metrics were compared across all models to support performance analysis.

4.1 Results on clean test

The results are shown in the table and bar chart below.

	kNN	XGB	Tree	RF	SVM	LogReg	CNN
accuracy	94.16%	96.07%	90.36%	94.31%	86.89%	89.63%	98.59%
precision	95.36%	97.23%	88.22%	96.24%	87.58%	89.80%	97.64%
recall	92.08%	93.90%	87.11%	91.11%	81.89%	88.62%	98.54%
F1	93.61%	95.45%	87.62%	93.28%	83.83%	89.11%	98.08%

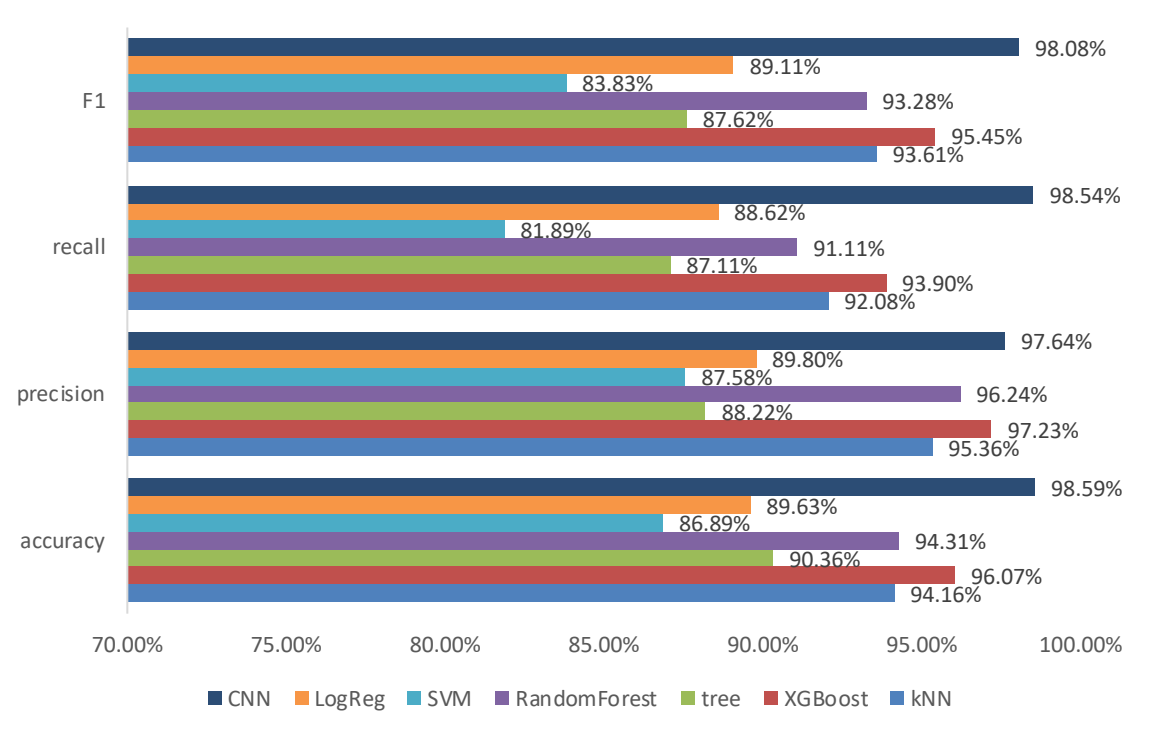


Figure 5 results metrics bar chart

CNN vs classic classifiers

CNN substantially outperforms all classic methods (e.g. Accuracy 98.59%, F1 98.08%).

Reasons can include:

- CNN operates on raw 2D spatial structure and learns hierarchical features (edges → textures → shapes) rather than relying on flattened pixels → PCA.
- Data augmentation during CNN training increases robustness to small transformations (flip/rotation), improving generalization on test and degraded sets.
- End-to-end learning tailors feature extraction for the classification objective; classical pipelines separate handcrafted preprocessing (PCA) from the classifier.

Relative behaviour:

- The gap (≈2 to 8 percentage points) is meaningful for multi-class tasks; CNN reduces both false positives and false negatives, reflected in high precision and recall.
- Classic ensembles (XGBoost, RandomForest) close some of the gap but cannot match the spatial inductive bias of CNNs.

Comparison among classic classifiers

Overall ranking by Accuracy / F1:

Ranking	Classifier	Accuracy	F1
1	XGBoost	96.07%	95.45%
2	RandomForest	94.31%	93.28%
3	kNN	94.16%	93.61%
4	DecisionTree	90.36%	87.62%
5	LogisticRegression	89.63%	89.11%
6	SVM	86.89%	83.83%

- XGBoost leads on all metrics, showing best trade-off between precision and recall.
- Random Forest and kNN are strong and close to each other; RandomForest slightly better on precision, kNN slightly better on recall in this set.
- Decision Tree underperforms ensemble/tree-based siblings, indicating single-tree variance and overfitting or limited generalization.
- Logistic Regression performs reasonably but lags ensembles; its macro-F1 near 89% suggests it handles class balance moderately well with PCA features.
- Linear SVM scored lowest across metrics, suggesting a linear decision boundary on PCA features is insufficient for some class separations here.

Precision vs Recall trade-offs:

- High precision with lower recall (e.g., XGBoost precision 97.23% vs recall 93.90%) indicates conservative predictions with few false positives but some misses.
- kNN shows balanced precision/recall (95.36% / 92.08%), implying nearest-neighbour structure preserves class neighborhoods in PCA space.
- SVM's relatively lower recall (81.89%) shows many false negatives, it misses true instances more often than other models.

Confusion Matrices

Additionally, confusion matrices were plotted for each model to visualize classification accuracy. Below graphs (figure 5 - 11) are the visualised confusion matrix of each classifier.

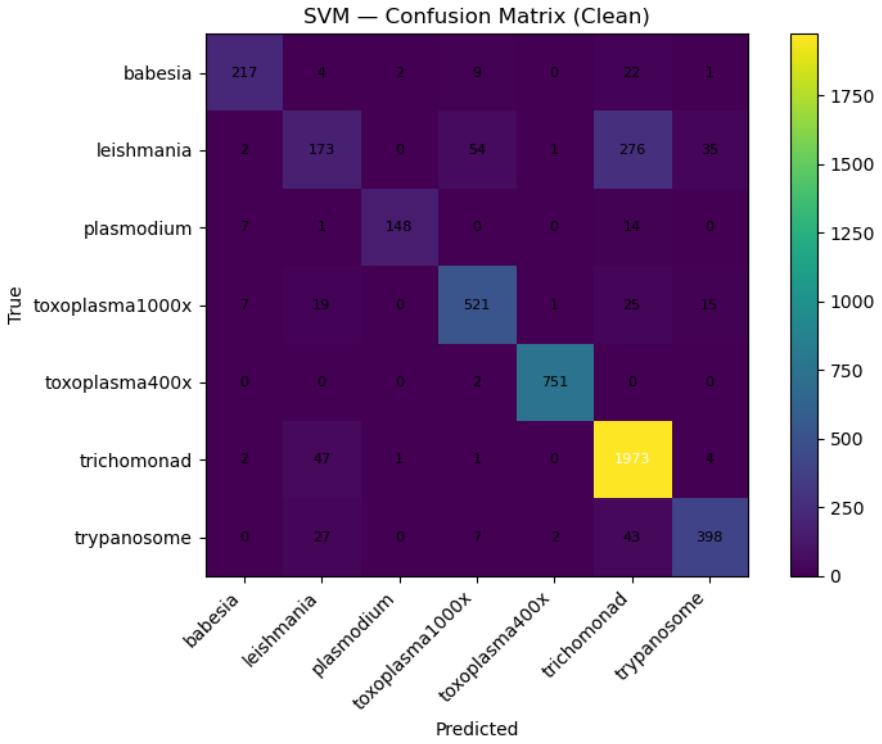


Figure 6 confusion matrix - SVM

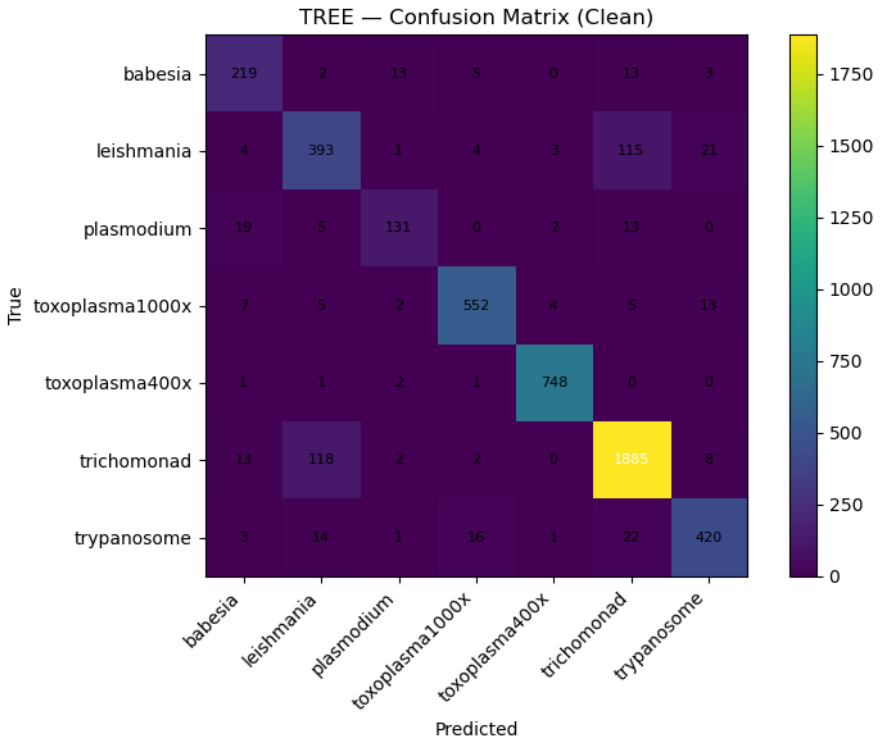


Figure 7 confusion matrix - tree

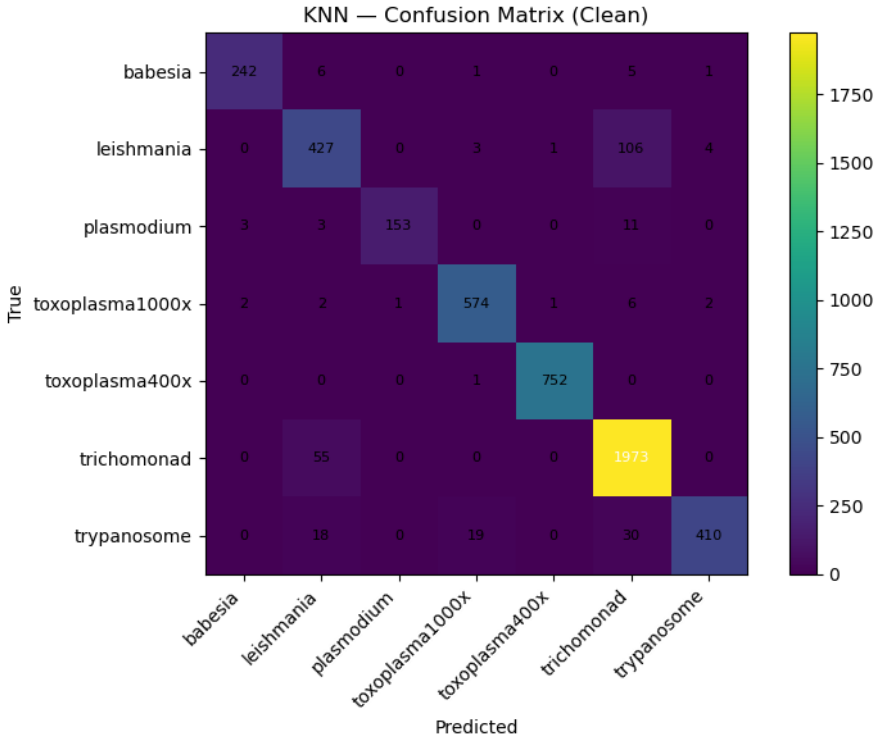


Figure 8 confusion matrix - knn

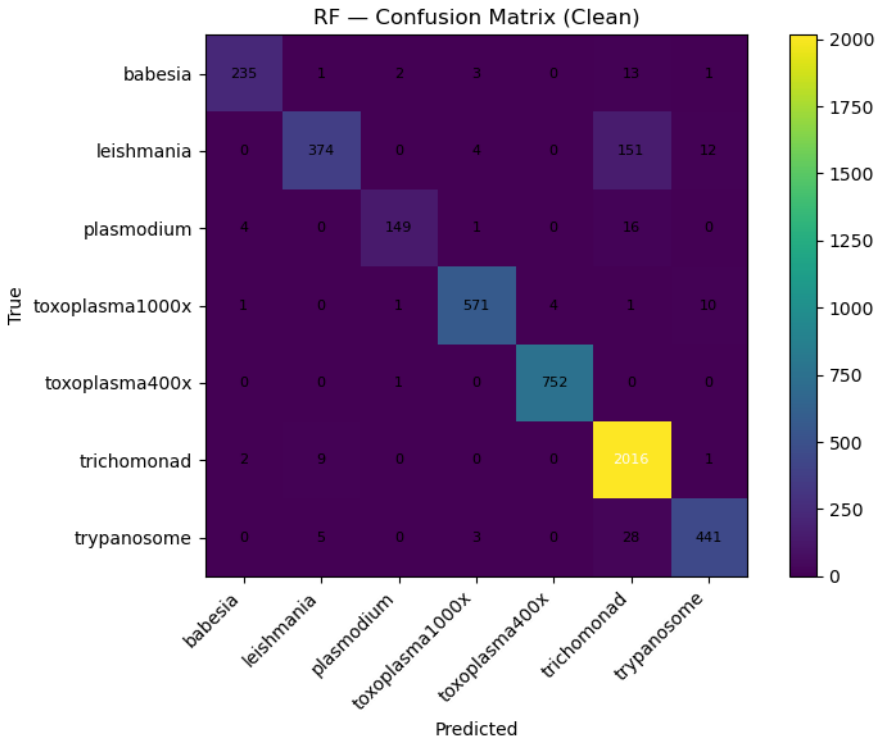


Figure 9 confusion matrix - random tree

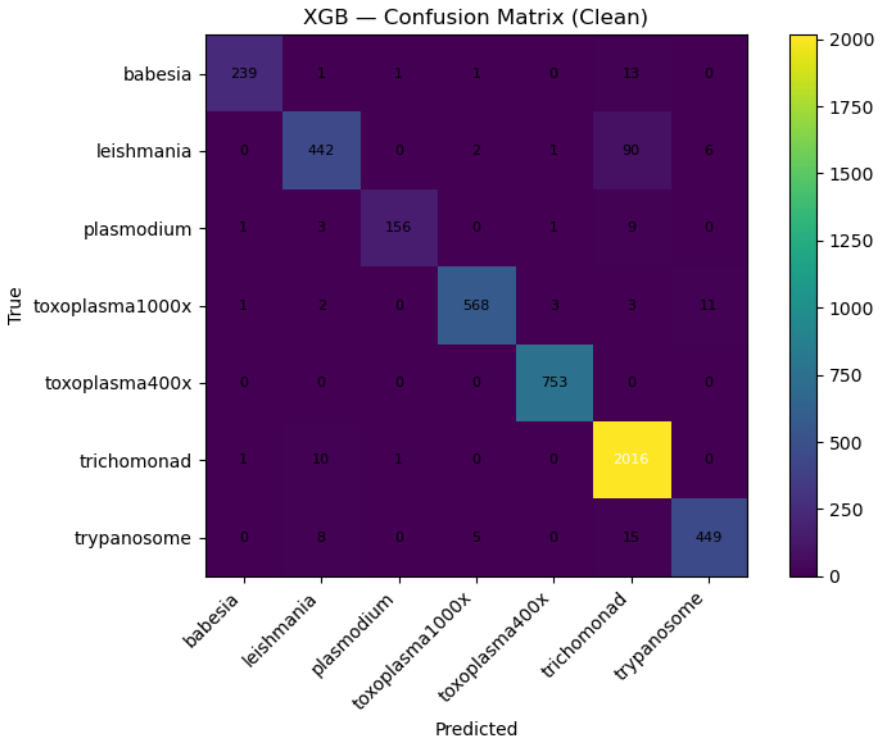


Figure 10 confusion matrix - XGBoost

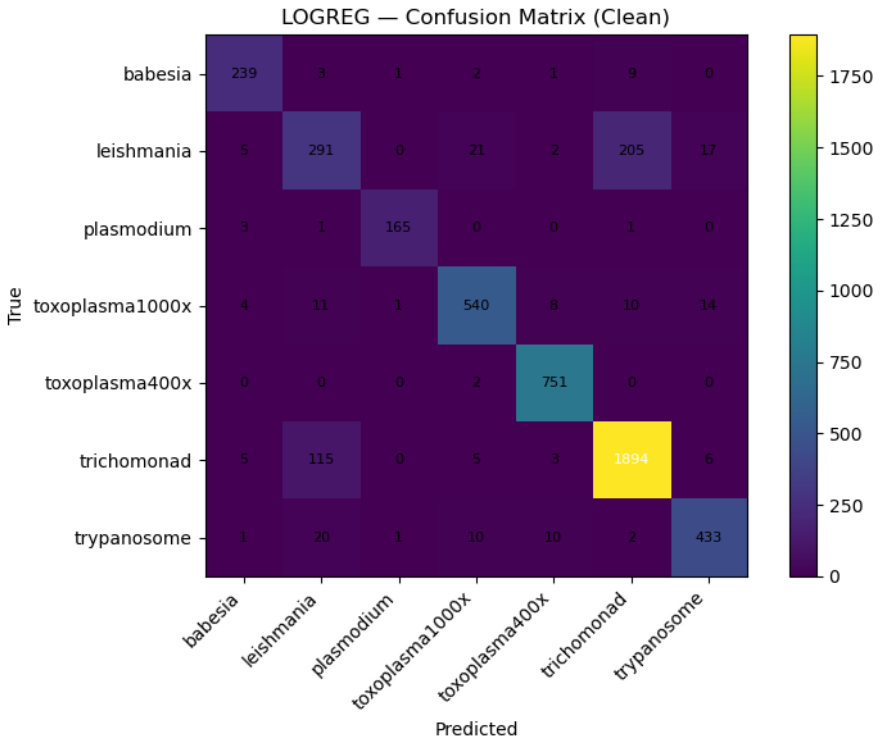


Figure 11 confusion matrix - logistic regression

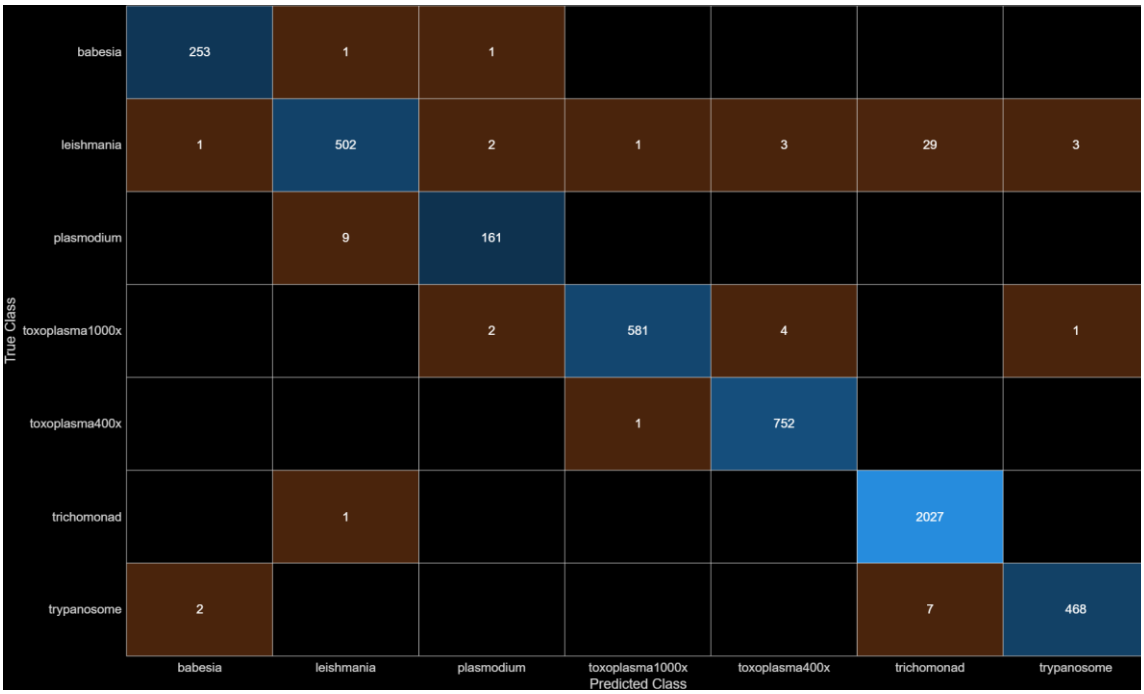


Figure 12 confusion matrix - CNN

CNN (best overall):

Near-perfect diagonal: babesia 253, leishmania 502, plasmodium 161, toxoplasma1000× 581, toxoplasma400× 752, trichomonad 2027, trypanosome 468.

Residuals are small and concentrated (e.g., leishmania→trichomonad = 29), reflecting robust learned features and invariances.

XGBoost (best classic):

Strong diagonals; main weakness is leishmania→trichomonad (90). Excellent on toxoplasma400× (753/753) and trichomonad (2016 with minimal errors).

Occasional babesia→trichomonad (13) and trypanosome→trichomonad (15).

Random Forest (ensemble):

Similar profile to XGBoost but slightly more leishmania→trichomonad (151) and trypanosome→trichomonad (28).

Toxoplasma classes remain very strong; plasmodium stable with small bleed to trichomonad.

Decision Tree (single tree; the "random forest" matrix with wide off-diagonals):

Noticeably broader error spread: leishmania→trichomonad (115), trichomonad inflow from leishmania/babesia; more scatter across toxoplasma1000×/trypanosome.

Matches its lower aggregate accuracy/F1 in your summary.

kNN:

Competitive diagonals; characteristic confusions: leishmania→trichomonad (106) and trypanosome→(toxoplasma1000×=19, trichomonad=30).

Performs well on both toxoplasma classes and plasmodium.

Logistic Regression:

Major issue: leishmania→trichomonad (205). Otherwise broadly reasonable diagonals (e.g., plasmodium 165, toxoplasma400× 751).

Trichomonad is strong (1894) but attracts inflow from leishmania/babesia.

SVM (weakest here):

Large leishmania→trichomonad (276) plus extra spill from babesia and trypanosome. Indicates difficulty with high-dimensional pixel vectors despite PCA/standardisation.

Class-by-class patterns (key confusions)

Leishmania → Trichomonad is the dominant error mode for most classic models.

CNN: 29 mislabels (leishmania→trichomonad) versus 502 true; markedly lower than others.

XGBoost: 90 mislabels; Random Forest: 151; kNN: 106; Logistic: 205; SVM: 276 (weakest).

Interpretation: class-imbalance + morphological/texture similarity with trichomonad.

Trichomonad is generally handled well but still draws some spill-over from several classes in weaker models.

CNN: 2027 true with near-zero off-diagonals.

Classic models: small but systematic inflow from leishmania and occasional from babesia/trypanosome.

Toxoplasma (400× vs 1000×)

Toxoplasma400× is near-perfect for all models (CNN: 752/752; XGBoost/RF/kNN also almost perfect).

Toxoplasma1000×: tiny cross-confusions to 400× or trypanosome in several classics; CNN reduces these to single-digit counts.

Plasmodium

High true counts across all models; very few mislabels (CNN: 161 true, near-zero elsewhere; classic models show small bleed to trichomonad).

Babesia

Mostly correct across models; small leaks to trichomonad (e.g., XGBoost/RandomForest/LogReg) and occasional to leishmania/others in weaker models.

Trypanosome

CNN is very strong (468 true, trivial off-diagonals).

Classic models show some confusion into trichomonad and a few into toxoplasma1000× (e.g., kNN: 19 to toxoplasma1000×, 30 to trichomonad; XGBoost: 15 to trichomonad).

These patterns can be explained by class imbalance, with the large trichomonad class biasing classic models' decision boundaries and minority classes such as leishmania being pulled toward the dominant class due to shared texture cues. The distinction between models also comes down to feature learning. The CNN learns discriminative morphological features directly from pixel data, sharply reducing leishmania-trichomonad confusion, while classic models depend on vectorised or PC-reduced features that capture global variance but not the fine, class-specific cues necessary for optimal separation. Magnification variants (toxoplasma 400× vs 1000×) are generally well separated by most models, particularly the CNN and tree ensembles, with only minor cross-talk.

Efficiency and Cost

Summary of training time (from lowest to highest)

	Decision Tree	SVM	Random Forest	LogReg	XGBoost	kNN	CNN
Train time (min)	0.03	0.04	0.06	0.12	0.20	0.29	190.3
Clean test time (s)	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	70.7

To ensure fairness, the timings were measured on the same workstation with

- a 13th Gen Intel® Core™ i5-13400 2.50 GHz processor and,
- 16.0 GB installed RAM (15.8 GB usable).

All runs used the CPU only; no GPU acceleration was involved. Reported times are mean wall-clock measurements (use time.perf_counter or equivalent) and left-censored values such as <0.01 s indicates measurements below the chosen reporting threshold or the timer's noise floor.

The models show a wide trade-off between training cost and inference efficiency: the CNN dominates training time by far (190.3 minutes) while classical models are orders of magnitude cheaper to train (LogReg 0.12 min, SVM 0.04 min, kNN 0.29 min, RandomForest 0.06 min, Tree 0.03 min, XGBoost 0.20 min). In practice this means that the CNN demands the highest compute budget and produces the greatest deployment cost per inference unless inference is highly batched or served on powerful hardware; classical models are far cheaper to train, faster to iterate, and better suited to low-latency or resource-constrained deployment.

Pre-training overhead for classical models

However, before training starts, the classical pipeline incurs an additional 20–30 minutes to fit the scaler and PCA on the training set (IncrementalPCA with batching). The CNN does not require this preprocessing fit stage.

When training a single classical model, this 20–30 min overhead can exceed the model's own fit time (e.g., Logistic Regression at 0.12 min or Random Forest at 0.06 min). When training multiple classical models on the same PCA features, the PCA/scaler fit is amortised: subsequent models reuse the cached features, so marginal training time remains in the seconds range.

The CNN's training remains the dominant cost (~190.3 min) even after accounting for the classical preprocessing overhead. Yet the CNN avoids a PCA dependency, which simplifies the pipeline, reduces the risk of feature drift, and can improve portability (no requirement to persist and apply a separate PCA transform at inference).

4.2 Results on robustness test (Accuracy / F1-score (ΔF1))

Degrad / Level	KNN (%)	XGB (%)	TREE (%)	RF (%)	SVM (%)	LOGREG (%)	CNN (%)
Clean (baseline)	94.16% / 93.61% (+0.00%)	96.07% / 95.45% (+0.00%)	90.36% / 87.62% (+0.00%)	94.31% / 93.28% (+0.00%)	86.89% / 83.83% (+0.00%)	89.63% / 89.11% (+0.00%)	98.59% / 98.08% (+0.00%)
Resolution loss — down 2×	94.14% / 93.59% (-0.02%)	96.07% / 95.45% (+0.00%	90.48% / 87.78% (+0.12%)	94.31% / 93.29% (+0.00%)	86.89% / 83.81% (+0.00%)	89.59% / 89.07% (-0.04%)	98.57% / 98.05% (−0.03%)
Resolution loss — down 4×	94.12% / 93.57% (-0.04%)	96.09% / 95.48% (+0.03%)	90.46% / 87.75% (+0.10%)	94.35% / 93.33% (+0.04%)	86.82% / 83.70% (-0.06%)	89.65% / 89.15% (+0.02%)	98.57% / 98.06% (−0.02%)
Gaussian blur s = 1.0	94.16% / 93.61% (+0.00%)	96.07% / 95.45% (+0.00%)	90.42% / 87.73% (+0.06%)	94.29% / 93.27% (-0.02%)	86.89% / 83.81% (+0.00%)	89.61% / 89.10% (-0.02%)	98.57% / 98.05% (−0.03%)
Gaussian blur s = 2.0	94.14% / 93.59% (-0.02%)	96.05% / 95.43% (−0.02%)	90.48% / 87.80% (+0.12%)	94.37% / 93.35% (+0.06%)	86.85% / 83.74% (-0.04%)	89.61% / 89.11% (-0.02%)	98.55% / 98.04% (−0.04%)
Motion blur k = 5	94.14% / 93.59% (-0.02%)	96.07% / 95.45% (+0.00%)	90.42% / 87.68% (+0.06%)	94.29% / 93.27% (-0.02%)	86.89% / 83.81% (+0.00%)	89.63% / 89.11% (+0.00%)	98.57% / 98.05% (−0.03%)
Gaussian noise s = 5	94.16% / 93.62% (+0.00%)	96.07% / 95.45% (+0.00%)	90.38% / 87.59% (+0.02%)	94.39% / 93.40% (+0.08%)	86.91% / 83.85% (+0.02%)	89.59% / 89.03% (-0.04%)	92.75% / 91.49% (−6.59%)
Gaussian noise s = 15	94.16% / 93.63% (+0.00%)	96.07% / 95.46% (+0.01%)	90.30% / 87.52% (-0.06%)	94.26% / 93.25% (-0.04%)	86.78% / 83.67% (-0.10%)	89.69% / 89.15% (+0.06%)	31.32% / 23.07% (−75.01%)
JPEG quality = 60	94.18% / 93.70% (+0.02%)	95.95% / 95.33% (−0.12%)	90.30% / 87.44% (-0.06%)	94.35% / 93.35% (+0.04%)	86.80% / 83.74% (-0.08%)	89.53% / 88.92% (-0.10%)	98.67% / 98.17% (+0.09%)
JPEG quality = 40	94.26% / 93.78% (+0.10%)	96.01% / 95.40% (−0.05%)	90.27% / 87.47% (-0.08%)	94.39% / 93.41% (+0.08%)	86.85% / 83.73% (-0.04%)	89.21% / 88.45% (-0.42%)	98.57% / 98.01% (−0.07%)
JPEG quality = 20	93.89% / 93.30% (-0.27%)	95.84% / 95.16% (−0.29%)	89.53% / 86.73% (-0.83%)	94.37% / 93.34% (+0.06%)	86.08% / 82.71% (-0.81%)	88.15% / 87.53% (-1.48%)	97.61% / 97.13% (−0.95%)

The robustness table reports each model's Accuracy / F1-score on the clean baseline and under each degradation condition (resolution loss, Gaussian blur, motion blur, Gaussian noise, JPEG compression). For degraded tests, we use the same evaluation pipeline and metrics as the clean set; to simplify comparison we present Accuracy / F1-score with ΔF1 relative to clean. This focuses attention on (i) overall correctness (Accuracy) and (ii) class-imbalance–aware performance (macro-F1), while ΔF1 compactly expresses sensitivity to each degradation without reprinting full confusion matrices.

Most classical models (kNN, Random Forest, XGBoost, Decision Tree, Logistic Regression, SVM) exhibit minimal performance drift under mild degradations. The CNN remains strong under blur, resolution loss, and JPEG compression, but shows marked sensitivity to additive Gaussian noise, especially at higher variance.

Resolution loss (2×, 4×).
All models change by ≤0.10 pp F1; CNN stays at 98.05–98.06% F1 (−0.04/−0.02). Downsample–upsample at these levels does not materially harm classification.

Blur (Gaussian σ=1.0/2.0; Motion k=5).
Classicals remain effectively flat (e.g., RandomForest within ±0.06 pp F1). CNN varies within −0.04 to −0.03 pp F1. These blur severities are largely benign.

Gaussian noise (σ=5, 15).

- σ=5: Classical models are stable (ΔF1 within ±0.08 pp). CNN drops to 91.49% F1 (−6.59 pp).
- σ=15: Classicals still modestly affected (e.g., RF −0.03 pp F1; LogReg +0.04 pp). CNN degrades sharply to 23.07% F1 (−75.01 pp).
  This indicates strong CNN sensitivity to pixel-level noise without noise-aware training/denoising; classical pipelines with PCA/standardisation appear more noise-tolerant at tested severities.

JPEG compression (Q=60/40/20).

- Q=60: negligible changes or slight improvements (CNN +0.09 pp; XGBoost +0.03 pp).
- Q=40: small, mixed deltas (within ±0.42 pp for classics; CNN −0.07 pp).
- Q=20: gradual performance loss across models (e.g., Tree −0.83 pp F1; CNN −0.95 pp), still far milder than Gaussian noise effects for CNN.

These results indicate that while the CNN remains resilient to blur, resolution loss, and JPEG compression, it is highly sensitive to additive Gaussian noise unless explicitly mitigated. By contrast, the classical models are consistently stable across all tested degradations, exhibiting only sub-percentage shifts in F1 and reflecting resilient decision rules in feature space. For deployments where sensor noise is likely, CNN performance should be hardened through noise-aware augmentation (Gaussian/Poisson), denoising pre-filters, or transfer learning from noise-robust backbones; otherwise, classical models offer a reliable, low-variance fallback on noisy inputs.

5. Discussion and Challenges

Throughout the project, we were able to meet the majority of our goals. We implemented and compared multiple classifiers on parasite microscopy images and assessed robustness under controlled degradations, using a reproducible pipeline and clearly reported metrics.

5.1 Objectives and Intended Outcomes

Our plan specified two primary objectives supported by reproducible practice.

First, **implement and compare multiple classification methods for parasite images.**
We trained six classic models,Logistic Regression, kNN, SVM, Decision Tree, Random Forest, and XGBoost, alongside a CNN. A stratified 60/20/20 split with a fixed seed (42177) ensured fair comparison. Results were presented with accuracy, precision, recall, F1, and confusion matrices. For example, the CNN achieved 98.59% accuracy and 98.08% F1, outperforming the best classic model (XGBoost, 96.07% / 95.45%). We adopted sound evaluation and reproducibility practices. We standardised inputs to 256×256, fixed the RNG seed (42177), logged preprocessing parameters, and saved artefacts. Confusion matrices were produced for each model, for instance, the characteristic leishmania-trichomonad confusion is visible across classic models, while the CNN sharply reduces it.

Second, **simulate lower-quality, smartphone-style captures to test robustness.**
We generated degraded copies of the validation and test sets only (never used for fitting), covering resolution loss (down 2×/4×), Gaussian blur (σ = 1.0, 2.0), motion blur (k = 5), Gaussian noise (σ = 5, 15), and JPEG compression (Q = 60/40/20). We then reported deltas from the clean baseline.

5.2 Deviations from Plan

Some planned elements were deliberately not executed due to limitation of resources and the need to maintain scope control and reproducibility.

1. **Dataset-B (Image Bank of Parasitology) was not used for cross-dataset generalisation.**
   Why: The profile lists only 377 images with heterogeneous modalities and many "All rights reserved" items. In practice, the available, consistently licenced content and usable class counts were insufficient for a reliable, reproducible external test.
   Impact: We did not run a formal cross-dataset generalisation study.
   Example: Instead of external domain shift via Dataset-B, we emphasised synthetic degradations on Dataset-A's validation/test sets (e.g., blur/noise/JPEG) to probe robustness without confounding licence and class-count issues.

2. **Host-cell superclass robustness check was omitted.**
   Why: To control scope and compute cost, we focused the primary task on the six parasite classes and excluded host cells during preprocessing.
   Impact: We did not explicitly measure confusion between parasite classes and non-parasite fields.
   Example: Our manifest and quick counts show only the six parasite labels propagating

through the pipeline; no "host" superclass appears in the saved splits or artefacts.

3. **Optional preprocessing steps (grayscale, background correction, contrast adjustment) and de-duplication of near-identicals were not implemented.**
   Why: The curated nature of Dataset-A and the need to keep the pipeline simple and reproducible across MATLAB/Python led us to a minimal, robust set: resize + normalise. Cross-tool parity (MATLAB for CNN; Python for classics) was prioritised.
   Impact: We potentially left some marginal gains in normalisation/contrast on the table and did not systematically remove near-duplicates across splits.
   Example: The preprocessing script applies resize to 256×256 and intensity scaling, saves parameters (preproc_params.json), and proceeds directly to stratified splitting; there is no grayscale/background-correction block or near-duplicate removal pass.

4. **Degradation suite excluded Poisson noise and a longer motion-blur kernel.**
   Why: We fixed parameters a priori to match time/compute constraints, keeping the matrix of conditions tractable.
   Impact: The robustness map does not include sensor-like shot noise or stronger motion streaking; results therefore emphasise Gaussian families and JPEG artefacts.
   Example: Implemented levels: Gaussian blur σ = 1.0, 2.0; motion blur k = 5; Gaussian noise σ = 5, 15; JPEG Q = 60/40/20. Conditions such as Poisson noise and motion blur k = 10 were defined in planning but not executed.

The evidence aligns with our aims on both fronts.

The CNN's diagonal dominant confusion matrix (e.g., leishmania 502 correct with only 29 misclassified as trichomonad) shows that learned morphological features separate classes effectively, whereas classical models exhibit the characteristic leishmania - trichomonad spill (e.g., Logistic Regression with 205 such errors), consistent with limits of vectorised/PCA features.

On robustness: CNN is stable under blur, resolution loss, and JPEG, but exhibits pronounced sensitivity to Gaussian noise (e.g., −6.59 pp at σ=5; −75.01 pp at σ=15), whereas classical models show sub-percentage F1 shifts across all degradations.

5.3 Challenges

Data preprocessing and cross-environment consistency

Challenges: The initial Python pipeline on Google Colab ran for hours due to limited RAM/CPU, and the team split tooling across MATLAB (CNN) and Python (SVM, Logistic Regression, XGBoost, etc.). This produced inconsistencies in directory paths, image handling, and evaluation functions (MATLAB evaluate_model.mlx versus Python evaluate_model()), and required careful alignment of resizing, normalisation, and PCA.

Solution: We re-implemented a minimal, mirrored preprocessing in both environments (resize to 256×256 and intensity scaling), and fixed a single source of truth via the manifest.csv and SEED = 42177. We standardised outputs to a shared JSON/CSV format and used a common metric set (accuracy, precision, recall, F1, confusion matrix), ensuring that results were directly comparable across languages and libraries.

Memory pressure and computational limits

Challenges: Several classic-model scripts attempted to load flattened image tensors into memory, triggering MemoryError (≈21 GB RAM projected). The CNN also demanded long wall-clock time (≈3 hours) even after capping to 4 epochs only due to GPU/compute constraints.

Solution: We switched to IncrementalPCA, reduced NCOMP, and processed data in mini-batches to keep memory bounded. For deep learning, we constrained epochs (MaxEpochs = 4) and adopted moderate batch size (MiniBatchSize = 64) to fit hardware limits.

What could have been done
More extensive hyperparameter tuning and longer CNN schedules were not feasible. With additional resources we would: (i) use a dedicated GPU instance (cloud or lab), (ii) enable mixed-precision training, (iii) adopt transfer learning from lightweight backbones, or (iv) employ feature caching to reduce repeated I/O and preprocessing.

Data integrity across clean and degraded sets

Challenges: Aligning degraded files to the manifest was error-prone; some paths were missing or mismatched, causing Error using imread: Unable to find file.

Solution: We validated all file references against manifest.csv prior to evaluation, regenerated any missing artefacts, and enforced the rule that degradations are created only for val/test to avoid data leakage. This eliminated path drift between clean and degraded copies.

Reproducibility, versioning, and collaboration flow

Challenges: Early file sharing via Google Drive led to version conflicts; different members occasionally used different splits/seeds, producing non-comparable outputs. Team members using Jupyter/Python and MATLAB also faced reproducibility friction.

Solution: We migrated to a unified GitHub repository with a standardised folder layout (data/, scripts/, results/, artifacts/), branched workflows, and commit history. We enforced SEED = 42177, a shared manifest, and a common evaluation schema (accuracy, precision, recall, F1, confusion matrix with consistent label order). This ensured subsequent results were like-for-like.

Time management and execution sequencing

Challenge: Downstream modellers had to wait for the preprocessing pipeline to complete; the CNN's long training time further limited opportunities for parameter sweeps.

Solution: We staged work so classics could begin once the manifest and clean set were frozen, while degradations (val/test only) were generated in parallel.

What could have been done
A task queue with small, incremental data drops (e.g., class-balanced shards) and automated CI jobs would reduce idle time. For CNN, early-stopping with checkpointing, and a short learning-rate range test could have improved schedule efficiency.

Hardware and software constraints

Problem: Team machines varied (some CPU-only, no GPU). External USB 3.0 storage throttled I/O. MATLAB Parallel Computing Toolbox was unavailable on some systems, preventing parpool-based acceleration.

Solution: We moved active datasets to the local drive to improve throughput, kept batch sizes conservative, and avoided memory-hungry transforms.

What could have been done
With access to the Parallel Computing Toolbox or a shared GPU node, we would enable parallel datastores, prefetching, and parallelised augmentations; alternatively, containerised environments (Docker) plus pinned package versions would further standardise performance across hosts.

5.4 Future Work

Given additional time and resources, we would prioritise the following extensions to strengthen validity, robustness, and practical utility.

Model improvements, transfer learning, and tuning depth
Transfer learning: Initialise from lightweight, pretrained backbones (e.g., MobileNetV3, EfficientNet-B0/B1, ResNet-18/34) and adopt a staged schedule: freeze the backbone and train the classifier head; then unfreeze upper blocks for discriminative fine-tuning with a lower learning rate. Combine with mixed-precision training, early stopping, and checkpointing to enable longer effective training under fixed compute.

For classical models, expand hyperparameter grids; introduce feature caching to accelerate repeats and enable more exhaustive searches.

Cross-dataset generalisation and domain shift
Reintroduce an external corpus to test generalisation under domain shift. If Dataset-B remains unsuitable due to size and licensing, source an alternative public archive with permissive licensing and sufficient per-class counts. Run a like-for-like evaluation protocol and report deltas against the in-domain baseline.

Host-cell superclass robustness
Implement the planned “host” superclass to quantify parasite–non-parasite confusion. Train parasite-only models and parasite+host models, report changes in macro-F1 and per-class recall, and inspect confusion matrices for spill-over into the host category.

Expanded degradation families and severities
Add Poisson (shot) noise and a longer motion-blur kernel; extend resolution loss and include camera artefacts such as colour cast, exposure variation, and mild geometric perturbations. Calibrate severities to realistic smartphone capture conditions and report systematic Δ from clean.

Data quality and leakage control

Implement near-duplicate detection across splits to reduce optimistic bias, and add automated integrity checks for the degraded sets (file existence, label order, class balances). Maintain checksums and a manifest verifier to prevent path drift.

Imbalance-aware training
Evaluate class weighting, focal loss, or cost-sensitive learning for classical models and the CNN to mitigate minority-class under-performance, particularly for leishmania versus trichomonad.
Model improvements and tuning depth

For the CNN, explore transfer learning with lightweight backbones, mixed-precision training, and early-stopping with checkpointing to enable longer schedules under fixed compute. For classical models, broaden hyperparameter sweeps and assess feature caching to accelerate repeated runs.

Uncertainty, calibration, and decision support
Add probability calibration and threshold analysis to align operating points with application needs. Report expected calibration error alongside accuracy and F1, and provide per-class precision–recall curves.

Explainability and targeted error analysis
Use saliency/Grad-CAM for the CNN to verify that decisions attend to biologically plausible structures. Couple explanations with focused audits of recurrent confusions to inform data augmentation or annotation refinement.

Ensembling and hybrid strategies
Assess model ensembling within classical methods and CNN–classical hybrids to improve stability under degradations, reporting gains versus added compute.

Reproducibility and automation
Containerise environments, pin package versions, and add continuous integration to validate manifests and evaluation I/O. Automate the full pipeline with scripted entry points so that experiments are re-runnable end-to-end.

Efficiency and deployment considerations
Profile I/O and compute to remove bottlenecks, store intermediate features on SSD, and evaluate ONNX or similar export paths for downstream deployment. Record inference latency and memory footprints on representative hardware.

5.5 Lessons Learnt

Model capability and trade-offs
We observed a clear performance hierarchy: the CNN outperformed all classical baselines on accuracy and F1, while among the classical models XGBoost and Random Forest were consistently strongest, kNN was competitive, and SVM and Logistic Regression lagged. However, CNNs were highly vulnerable to additive gaussian noise in the absence of noise-aware training, whereas classical models remained stable, likely due to standardisation/PCA feature pipelines. The lesson is that feature learning yields the largest gains on clean and most degraded conditions, but robustness to sensor noise requires explicit mitigation (e.g., noise-aware augmentation,

denoising, or transfer learning). Ensembles provide strong, efficient baselines with stable performance under a range of degradations, and simpler linear/kernel methods continue to struggle on high-dimensional image inputs unless supported by careful feature engineering or dimensionality reduction.

Data preparation and reproducibility

We learnt that a single manifest, fixed seed (42177), and standardised outputs made cross-environment (MATLAB/Python) results comparable and more feasible. A minimal, consistent preprocessing contract was more valuable than a more complex but uneven pipeline. Fixing a single manifest, a stratified 60/20/20 split, and SEED = 42177, together with saved artefacts and shared metric schemas, made multi-model comparison straightforward and made faults easier to detect. We also learnt that optional steps such as near-duplicate removal and additional normalisation should be planned early, because retrofitting them late is operationally expensive.

Team process and tool interoperability

Mixing MATLAB and Python is feasible if the contract is explicit. A shared repository, standardised folder structure, and common output formats reduced divergence across environments. The lesson is that clear interfaces and lightweight conventions are more effective than ad hoc coordination when teams and tools differ.

Compute constraints shape outcomes

The project emphasised that engineering choices are constrained by hardware. IncrementalPCA, batch processing and conservative epoch budgets allowed us to complete a broad comparison, but limited hyperparameter sweeps and CNN schedule depth. We learnt that better resource planning, feature caching, and staged training would enable deeper exploration without compromising timelines.

6. Conclusion

In conclusion, we successfully implemented multiple machine learning models to classify parasite images, achieving strong performance across most models. We delivered a fair, reproducible comparison of seven models for parasite image classification. The CNN achieved the strongest clean-set performance (98.59% accuracy; 98.08% F1). Among classical baselines, XGBoost and Random Forest were strongest; kNN was competitive. Robustness patterns were clear: the CNN remained strong under blur, resolution loss, and JPEG compression but was highly sensitive to additive Gaussian noise at higher variance. Classical models were stable across all degradations, with only small F1 changes.

Efficiency results were consistent. The CNN incurred the highest compute cost and the longest training and inference times. Classical models trained in seconds and were fast at inference, especially when PCA features were reused. The 20–30 minutes required to fit the scaler and PCA was a one-off cost that could be amortised across models. Evaluation was aligned across environments using a single manifest, a fixed seed, consistent label ordering, and the same metrics.

During the implementation, we overcame slow preprocessing, cross-environment differences, and memory limits by standardising the pipeline, enforcing a shared manifest and seed, and using Incremental PCA with batched processing. We learnt that (i) feature learning brings the largest accuracy gains but needs noise-aware training for robustness, (ii) ensembles offer strong, efficient classical baselines, (iii) strict data contracts (manifest, fixed seed, fixed label order) are key to reproducibility, and (iv) hardware constraints shape feasible schedules and tuning depth. These lessons explain both our successes and where results were limited, and they inform clear next steps: noise-robust training, transfer learning, and expanded robustness testing.

References

Hendrycks, D., & Dietterich, T. (2019). Benchmarking neural network robustness to common corruptions and perturbations. *arXiv preprint arXiv:1903.12261*.

Institute, S. I. D. (2018). *Image Bank of Parasitology* Flickr. https://www.flickr.com/people/163527902@N02/

Li, S., & Zhang, Y. (2020). *Microscopic Images of Parasites Species* Version V3) [images]. https://doi.org/10.17632/38jtn4nzs6.3

Raju, G., Ranjan, A., Banik, S., Poddar, A., Managuli, V., & Mazumder, N. (2024). A commentary on the development and use of smartphone imaging devices. *Biophysical Reviews*, *16*(2), 151-163.

WHO. (2015). *Guidelines for the treatment of malaria*. World Health Organization.

Yu, H., Mohammed, F. O., Abdel Hamid, M., Yang, F., Kassim, Y. M., Mohamed, A. O., Maude, R. J., Ding, X. C., Owusu, E. D., & Yerlikaya, S. (2023). Patient-level performance evaluation of a smartphone-based malaria diagnostic application. *Malaria journal*, *22*(1), 33.

Appendix 1 Script of Image data clean and degradation preprocessing pipeline

```
clear; clc;
```

config

```
RAW_DIR = 'D:\LocalUser\42177 Project\raw';
OUT_CLEAN = 'D:\LocalUser\42177 Project\data\clean';
MANIFEST = 'D:\LocalUser\42177 Project\data\manifest.csv';
ART_DIR = 'D:\LocalUser\42177 Project\artifacts';

TARGET = [256 256]; % [H W]
COLOR_MODE = "rgb"; % keep color
SEED = 42177; rng(SEED); % locks the randomness so dataset split and results are repeatable

if ~exist(OUT_CLEAN,'dir'), mkdir(OUT_CLEAN); end
if ~exist(ART_DIR,'dir'), mkdir(ART_DIR); end
```

```
%% SCAN RAW → TABLE
imds = imageDatastore(RAW_DIR, "IncludeSubfolders",true, ...
    "FileExtensions",{'.jpg','.jpeg','.png','.tif','.tiff','.bmp'}, ...
    "LabelSource","foldernames");

T = table(string(imds.Files), lower(string(imds.Labels)), ...
    'VariableNames', {'filepath','label'});
% labels: babesia, leishmania, plasmodium, toxoplasma400x, toxoplasma1000x, trichomonad, trypanosome

%% STRATIFIED SPLIT 60/20/20
labels = categories(categorical(T.label));
subset = strings(height(T),1);
idxAll = (1:height(T))';

for k = 1:numel(labels)
    idx = idxAll(T.label==labels(k));
    idx = idx(randperm(numel(idx)));
    n = numel(idx); nTr = floor(0.6*n); nVa = floor(0.2*n);
    subset(idx(1:nTr))         = "train";
    subset(idx(nTr+1:nTr+nVa)) = "val";
    subset(idx(nTr+nVa+1:end)) = "test";
end

T.subset = subset;
T.id = (1:height(T)).';
T = movevars(T,'id','Before','filepath');

%% SAVE MANIFEST
if ~exist(fileparts(MANIFEST),'dir'), mkdir(fileparts(MANIFEST)); end
writetable(T(:,["id","filepath","label","subset"]), MANIFEST);

%% SAVE PREPROC PARAMS (JSON)
P = struct('target_h',TARGET(1),'target_w',TARGET(2),'color_mode',COLOR_MODE, ...
           'normalization','im2double_0to1','seed',SEED);
fid = fopen(fullfile(ART_DIR,'preproc_params.json'),'w');
fwrite(fid,jsonencode(P,'PrettyPrint',true));
fclose(fid);

%% PREPROCESS FUNCTION — resize + normalize only
function Iout = preprocess_one(path, TARGET)
    I = imread(path);
    I = im2double(imresize(I, TARGET, 'Method','bilinear'));  % resize + normalize 0–1
    Iout = im2uint8(I);  % save as uint8 for disk
end

%% APPLY PREPROCESSING
N = height(T);
tic;
for i = 1:N
    sub = T.subset(i);      if sub=="", continue; end
    path = char(T.filepath(i));
    lab  = char(T.label(i));
    id   = T.id(i);

    outDir = char(fullfile(OUT_CLEAN, sub, lab));
    if ~exist(outDir,'dir'), mkdir(outDir); end

    Iout = preprocess_one(path, TARGET);
    imwrite(Iout, char(fullfile(outDir, sprintf("img_%07d.png", id))));

    if mod(i,1000)==0
        fprintf('Processed %d / %d (%.1f%%) in %.1f min\n', i, N, 100*i/N, toc/60);
    end
end
```

```
Processed 1000 / 23927 (4.2%) in 0.5 min
Processed 2000 / 23927 (8.4%) in 1.2 min
Processed 3000 / 23927 (12.5%) in 1.7 min
Processed 4000 / 23927 (16.7%) in 2.3 min
Processed 5000 / 23927 (20.9%) in 2.9 min
Processed 6000 / 23927 (25.1%) in 3.4 min
Processed 7000 / 23927 (29.3%) in 3.9 min
Processed 8000 / 23927 (33.4%) in 4.3 min
Processed 9000 / 23927 (37.6%) in 4.7 min
Processed 10000 / 23927 (41.8%) in 5.1 min
Processed 11000 / 23927 (46.0%) in 5.5 min
Processed 12000 / 23927 (50.2%) in 6.0 min
Processed 13000 / 23927 (54.3%) in 6.6 min
Processed 14000 / 23927 (58.5%) in 7.3 min
Processed 15000 / 23927 (62.7%) in 8.0 min
Processed 16000 / 23927 (66.9%) in 8.6 min
Processed 17000 / 23927 (71.0%) in 9.3 min
Processed 18000 / 23927 (75.2%) in 9.9 min
Processed 19000 / 23927 (79.4%) in 10.6 min
Processed 20000 / 23927 (83.6%) in 11.2 min
```

```
%% CREATE DEGRADED COPIES (val + test only)
OUT_DEG = 'D:\LocalUser\42177 Project\data\degraded';
if ~exist(OUT_DEG,'dir'), mkdir(OUT_DEG); end

levels = struct();
levels.resolution     = [2,4];        % downsample factors
levels.gaussian_blur  = [1.0,2.0];    % sigma
levels.motion_blur    = [5];          % kernel length
levels.gaussian_noise = [5,15];       % noise std dev
levels.jpeg           = [60,40,20];   % quality

degSub = T(ismember(T.subset, ["val","test"]), :);
fprintf('Generating degradations for %d images...\n', height(degSub));
```

```
Generating degradations for 9575 images...
```

```
for i = 1:height(degSub)
    path = fullfile(OUT_CLEAN, degSub.subset(i), degSub.label(i), sprintf("img_%07d.png", degSub.id(i)));
    I = im2double(imread(path));
    [h,w,~] = size(I);

    % 1. Resolution loss
    for f = levels.resolution
        small = imresize(I, [h/f, w/f], 'bilinear');
        up    = imresize(small, [h,w], 'bilinear');
        outDir = fullfile(OUT_DEG, sprintf('resolution_x%d', f), degSub.subset(i), degSub.label(i));
        if ~exist(outDir,'dir'), mkdir(outDir); end
        imwrite(up, fullfile(outDir, sprintf('img_%07d.png', degSub.id(i))));
    end

    % 2. Gaussian blur
    for s = levels.gaussian_blur
        K = fspecial('gaussian', max(1,2*ceil(3*s)+1), s);
        J = imfilter(I, K, 'replicate');
        outDir = fullfile(OUT_DEG, sprintf('gaussian_blur_s%.1f', s), degSub.subset(i), degSub.label(i));
        if ~exist(outDir,'dir'), mkdir(outDir); end
        imwrite(J, fullfile(outDir, sprintf('img_%07d.png', degSub.id(i))));
    end

    % 3. Motion blur
    for k = levels.motion_blur
        K = fspecial('motion', k, 0);
        J = imfilter(I, K, 'replicate');
        outDir = fullfile(OUT_DEG, sprintf('motion_blur_k%d', k), degSub.subset(i), degSub.label(i));
        if ~exist(outDir,'dir'), mkdir(outDir); end
        imwrite(J, fullfile(outDir, sprintf('img_%07d.png', degSub.id(i))));
    end

    % 4. Gaussian noise
    for s = levels.gaussian_noise
        J = imnoise(I, 'gaussian', 0, (s/255)^2);
        outDir = fullfile(OUT_DEG, sprintf('gaussian_noise_s%d', s), degSub.subset(i), degSub.label(i));
        if ~exist(outDir,'dir'), mkdir(outDir); end
        imwrite(J, fullfile(outDir, sprintf('img_%07d.png', degSub.id(i))));
    end

    % 5. JPEG compression
    for q = levels.jpeg
        outDir = fullfile(OUT_DEG, sprintf('jpeg_q%d', q), degSub.subset(i), degSub.label(i));
        if ~exist(outDir,'dir'), mkdir(outDir); end
        imwrite(I, fullfile(outDir, sprintf('img_%07d.jpg', degSub.id(i))), 'Quality', q);
    end

    if mod(i,500)==0
        fprintf('Degraded %d / %d (%.1f%%)\n', i, height(degSub), 100*i/height(degSub));
    end
end
```

```
Degraded 500 / 9575 (5.2%)
Degraded 1000 / 9575 (10.4%)
Degraded 1500 / 9575 (15.7%)
Degraded 2000 / 9575 (20.9%)
Degraded 2500 / 9575 (26.1%)
Degraded 3000 / 9575 (31.3%)
Degraded 3500 / 9575 (36.6%)
Degraded 4000 / 9575 (41.8%)
Degraded 4500 / 9575 (47.0%)
Degraded 5000 / 9575 (52.2%)
Degraded 5500 / 9575 (57.4%)
Degraded 6000 / 9575 (62.7%)
Degraded 6500 / 9575 (67.9%)
Degraded 7000 / 9575 (73.1%)
Degraded 7500 / 9575 (78.3%)
Degraded 8000 / 9575 (83.6%)
Degraded 8500 / 9575 (88.8%)
Degraded 9000 / 9575 (94.0%)
Degraded 9500 / 9575 (99.2%)
```

```
fprintf('All degradations complete.\n');
```

```
All degradations complete.
```

Appendix 2 Script of classic model – logistic regression

42177 — PCA → Logistic Regression (clean + degraded), manifest-driven

- Uses manifest and preprocessed data.
- Streams for scaler/PCA to avoid RAM spikes.
- Evaluates on clean test and all degraded test sets.

```
# %% [markdown]
# ## 0) Imports and global config

# %%
import os, json, time, random, numpy as np, pandas as pd
from glob import glob
from PIL import Image
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import IncrementalPCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support, accuracy_score
```

[1] Python

```
# ------------------------- CONFIG -------------------------
# Paths
MANIFEST = r"D:\LocalUser\42177 Project\data\manifest.csv"
ROOT_CLEAN = r"D:\LocalUser\42177 Project\data\clean"
ROOT_DEG = r"D:\LocalUser\42177 Project\data\degraded"
ART = r"D:\LocalUser\42177 Project\artifacts"

# hyperparameters
TARGET_HW = (256, 256)
SEED = 42177
BATCH = 64          # streaming batch for scaler/IPCA/transform
NCOMP = 64           # PCA components

os.makedirs(ART, exist_ok=True)
random.seed(SEED); np.random.seed(SEED)
```

[7] Python

1) Manifest and label set

- Manifest defines the project's class list and fixed splits.

```
# %%
df = pd.read_csv(MANIFEST)  # expects: id, filepath, label, subset, mag
LABELS = sorted(df['label'].unique().tolist())  # deterministic order
LABELS
```

[8] Python

```
['babesia',
 'leishmania',
 'plasmodium',
 'toxoplasma1000x',
 'toxoplasma400x',
 'trichomonad',
 'trypanosome']
```

2) I/O helpers

- Read RGB 256×256 images.
- Stream batches for train to reduce RAM.

```
# ------------------------- DATA HELPERS -------------------------
EXTS = ('.png','.jpg','.jpeg','.tif','.tiff','.bmp')

def _list_images(folder):
    if not os.path.isdir(folder): return []
    return [os.path.join(folder,f) for f in os.listdir(folder) if f.lower().endswith(EXTS)]

def manifest_labels(manifest_path):
    df = pd.read_csv(manifest_path)
    # use sorted labels for deterministic order across environments
    return sorted(df['label'].unique().tolist())

LABELS = manifest_labels(MANIFEST)

def stream_clean_subset(subset, target=TARGET_HW):
    """Yield (Xb, yb) batches from data/clean/<subset>/<label> with Xb in [0,1], flattened."""
    for lab in LABELS:
        folder = os.path.join(ROOT_CLEAN, subset, lab)
        paths = _list_images(folder)
        for i in range(0, len(paths), BATCH):
            chunk = paths[i:i+BATCH]
```

[9] ⊗ 3 ⚠ 3 Cell 4 of 16

Appendix 3 Script of classic model – knn

42177 Image Processing and Pattern Recognition

k-Nearest Neighbors (kNN) Model – PCA + Scaler Pipeline

Overview

This notebook implements a **k-Nearest Neighbors (kNN)** classifier for the parasite image classification project.
It follows the **same preprocessing and evaluation framework** as the Logistic Regression and SVM models to ensure fair, consistent comparison across all team models.

Key Features

- Uses the shared **manifest.csv** and **data/clean** folder structure (train/val/test splits fixed).
- Applies **global streaming** for memory-safe processing.
- Employs **StandardScaler** and **IncrementalPCA (NCOMP=64)** for feature standardisation and dimensionality reduction.
- Trains and evaluates a **kNN classifier (k=5, distance-weighted, Euclidean)** on compact PCA features.
- Produces evaluation metrics identical to `evaluate_model.mlx` in MATLAB (Accuracy, Precision, Recall, F1).
- Saves JSON results to the `artifacts/` folder for clean and degraded test sets.

Workflow

1. Load manifest and define label order.
2. Stream training data → fit StandardScaler and IncrementalPCA incrementally.
3. Transform train/test sets in batches to avoid memory overflow.
4. Train kNN on PCA-reduced features.
5. Evaluate on clean and degraded test sets.
6. Save results (`results_knn_clean.json`, `results_knn_<degradation>.json`).

```python
# %% 0) Imports and config
import os, json, time, random, gc
import numpy as np
import pandas as pd
from PIL import Image

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import IncrementalPCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support, accuracy_score

# Paths
MANIFEST   = r"D:\LocalUser\42177 Project\data\manifest.csv"
ROOT_CLEAN = r"D:\LocalUser\42177 Project\data\clean"
ROOT_DEG   = r"D:\LocalUser\42177 Project\data\degraded"
ART        = r"D:\LocalUser\42177 Project\artifacts"
os.makedirs(ART, exist_ok=True)

# Hyperparameters (kept small for 16 GB RAM)
TARGET_HW  = (256, 256)
SEED       = 42177
BATCH      = 64
NCOMP      = 64
K          = 5              # kNN neighbors
WEIGHTS    = "distance"     # "uniform" or "distance"
METRIC     = "euclidean"    # distance metric

random.seed(SEED); np.random.seed(SEED)
EXTS = ('.png','.jpg','.jpeg','.tif','.tiff','.bmp')
```

[1] Python

```python
# %% 1) Labels from manifest (fixed order)
df = pd.read_csv(MANIFEST)   # expects: id, filepath, label, subset, mag
LABELS = sorted(df['label'].unique().tolist())
LABELS
```

[2] Python

```
['babesia',
 'leishmania',
 'plasmodium',
 'toxoplasma1000x',
 'toxoplasma400x',
 'trichomonad',
 'trypanosome']
```

```python
# %% 2) Streaming IO helpers
def _list_images(folder):
    if not os.path.isdir(folder): return []
    return [os.path.join(folder,f) for f in os.listdir(folder) if f.lower().endswith(EXTS)]

def stream_clean_global(subset, batch=BATCH, target=TARGET_HW):
    """Yield (Xb, yb) batches from data/clean/<subset>/<label>, RGB→[0,1], flattened."""
    paths = []
    for lab in LABELS:
        d = os.path.join(ROOT_CLEAN, subset, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32) / 255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)

def stream_degraded_global(cond, subset="test", batch=BATCH, target=TARGET_HW):
    base = os.path.join(ROOT_DEG, cond, subset)
    if not os.path.isdir(base):
        return
    paths = []
    for lab in LABELS:
        d = os.path.join(base, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32) / 255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)
```

```python
# %% 3) Metrics packer (aligned with evaluate_model.mlx)
def evaluate_and_pack(y_true, y_pred, labels=LABELS):
    y_true = np.asarray(y_true).astype(str)
    y_pred = np.asarray(y_pred).astype(str)
    labels = [str(x) for x in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    acc = accuracy_score(y_true, y_pred)
    prec, rec, f1, _ = precision_recall_fscore_support(
        y_true, y_pred, labels=labels, average="macro", zero_division=0
    )
    return {
        "Accuracy": float(acc),
        "Precision": float(prec),
        "Recall": float(rec),
        "F1": float(f1),
        "ConfusionMatrix": cm.tolist(),
        "Labels": labels
    }
```

```python
# %% 4) Fit StandardScaler + IncrementalPCA (global stream, low RAM)
scaler = StandardScaler(with_mean=True, with_std=True)
for Xb, _ in stream_clean_global("train"):
    scaler.partial_fit(Xb.astype(np.float32, copy=False))
    del Xb; gc.collect()
print("Scaler fitted (global stream).")

ipca = IncrementalPCA(n_components=NCOMP, batch_size=BATCH)
for Xb, _ in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))  # may be float64 inside
    if Xb_std.shape[0] >= NCOMP:
        ipca.partial_fit(Xb_std)
    del Xb, Xb_std; gc.collect()
print(f"IPCA fitted (n_components={NCOMP}).")
```

```
Scaler fitted (global stream).
IPCA fitted (n_components=64).
```

```python
# %% 5) Transform TRAIN/TEST in streams → compact features
Xtr_chunks, ytr_chunks = [], []
for Xb, yb in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    Xb_pca = ipca.transform(Xb_std)
    Xtr_chunks.append(Xb_pca.astype(np.float32, copy=False))
    ytr_chunks.append(yb)
Xtr = np.vstack(Xtr_chunks); ytr = np.concatenate(ytr_chunks)

Xte_chunks, yte_chunks = [], []
for Xb, yb in stream_clean_global("test"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    Xb_pca = ipca.transform(Xb_std)
    Xte_chunks.append(Xb_pca.astype(np.float32, copy=False))
    yte_chunks.append(yb)
Xte = np.vstack(Xte_chunks); yte = np.concatenate(yte_chunks)
```

```python
# %% 6) Train kNN on PCA features
knn = KNeighborsClassifier(n_neighbors=K, weights=WEIGHTS, metric=METRIC, n_jobs=-1)

print("Training kNN...")
t_train = time.time()
knn.fit(Xtr, ytr)
train_secs = time.time() - t_train
print(f"Training completed in {train_secs/60:.2f} minutes")
```

```
Training kNN...
Training completed in 0.00 minutes
```

```python
# %% 7) Evaluate on clean test and save JSON
t_eval = time.time()
yhat = knn.predict(Xte)
test_secs = time.time() - t_eval

res_clean = evaluate_and_pack(yte, yhat, LABELS)
res_clean["TrainElapsedSeconds"] = round(train_secs, 2)
res_clean["TestElapsedSeconds"]  = round(test_secs, 2)

out_clean = os.path.join(ART, "results_knn_clean.json")
with open(out_clean, "w") as f:
    json.dump(res_clean, f, indent=2)

print("Saved:", out_clean)
res_clean
```

```
Saved: D:\LocalUser\42177 Project\artifacts\results_knn_clean.json

{'Accuracy': 0.9416043225270158,
 'Precision': 0.953594793203548,
 'Recall': 0.9207970920035234,
 'F1': 0.936050782482773,
 'ConfusionMatrix': [[242, 6, 0, 1, 0, 5, 1],
  [0, 427, 0, 3, 1, 106, 4],
  [3, 3, 153, 0, 0, 11, 0],
  [2, 2, 1, 574, 1, 6, 2],
  [0, 0, 0, 1, 752, 0, 0],
  [0, 55, 0, 0, 0, 1973, 0],
  [0, 18, 0, 19, 0, 30, 410]],
 'Labels': ['babesia',
  'leishmania',
  'plasmodium',
  'toxoplasma1000x',
  'toxoplasma400x',
  'trichomonad',
  'trypanosome'],
 'TrainElapsedSeconds': 0.0,
 'TestElapsedSeconds': 0.29}
```

```python
# %% 8) Evaluate on degraded test sets (streaming)
conds = [d for d in os.listdir(ROOT_DEG) if os.path.isdir(os.path.join(ROOT_DEG, d))]

for c in conds:
    y_true, y_pred = [], []
    any_batch = False
    for Xb, yb in stream_degraded_global(c, "test", batch=BATCH):
        any_batch = True
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std)
        yhat_b = knn.predict(Xb_pca)
        y_true.extend(yb.tolist()); y_pred.extend(yhat_b.tolist())
    if not any_batch:
        print(f"Skip {c}: no files."); continue

    resg = evaluate_and_pack(np.array(y_true), np.array(y_pred), LABELS)
    out = os.path.join(ART, f"results_knn_{c}.json")
    with open(out, "w") as f:
        json.dump(resg, f, indent=2)
    print("Saved:", out)
```

Appendix 4 Script of classic model – svm

42177 — PCA → SVM (clean + degraded), manifest-driven

- Global streaming for StandardScaler + IncrementalPCA
- BATCH=64, NCOMP=64
- Batch transform to avoid large allocations
- Results save JSON to artifacts/

```
# %% 0) Imports and config
import os, json, time, random, numpy as np, pandas as pd
from PIL import Image
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import IncrementalPCA
from sklearn.svm import LinearSVC
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support, accuracy_score

# Paths
MANIFEST   = r"D:\LocalUser\42177 Project\data\manifest.csv"
ROOT_CLEAN = r"D:\LocalUser\42177 Project\data\clean"
ROOT_DEG   = r"D:\LocalUser\42177 Project\data\degraded"
ART        = r"D:\LocalUser\42177 Project\artifacts"
os.makedirs(ART, exist_ok=True)

# Hyperparameters
TARGET_HW  = (256, 256)
SEED       = 42177
BATCH      = 64
NCOMP      = 64

random.seed(SEED); np.random.seed(SEED)
EXTS = ('.png','.jpg','.jpeg','.tif','.tiff','.bmp')
```

```
# %% 1) Labels from manifest (fixed order)
df = pd.read_csv(MANIFEST)  # expects: id, filepath, label, subset, mag
LABELS = sorted(df['label'].unique().tolist())
LABELS
```

[2]

```
['babesia',
 'leishmania',
 'plasmodium',
 'toxoplasma1000x',
 'toxoplasma400x',
 'trichomonad',
 'trypanosome']
```

```
# %% 2) I/O streaming (global stream)
def _list_images(folder):
    if not os.path.isdir(folder): return []
    return [os.path.join(folder,f) for f in os.listdir(folder) if f.lower().endswith(EXTS)]

def stream_clean_global(subset, batch=BATCH, target=TARGET_HW):
    paths = []
    for lab in LABELS:
        d = os.path.join(ROOT_CLEAN, subset, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32) / 255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        yield np.stack(Xb), np.array(yb)

def stream_degraded_global(cond, subset="test", batch=BATCH, target=TARGET_HW):
    base = os.path.join(ROOT_DEG, cond, subset)
    if not os.path.isdir(base):
        return
    paths = []
    for lab in LABELS:
        d = os.path.join(base, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32) / 255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)
```

[3]

```python
# %% 3) Metrics packer (aligned with evaluate_model.mlx)
def evaluate_and_pack(y_true, y_pred, labels=LABELS):
    y_true = np.asarray(y_true).astype(str)
    y_pred = np.asarray(y_pred).astype(str)
    labels = [str(x) for x in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    acc = (y_true == y_pred).mean().item()
    prec, rec, f1, _ = precision_recall_fscore_support(
        y_true, y_pred, labels=labels, average="macro", zero_division=0
    )
    return {
        "Accuracy": float(acc),
        "Precision": float(prec),
        "Recall": float(rec),
        "F1": float(f1),
        "ConfusionMatrix": cm.tolist(),
        "Labels": labels
    }
```

[4]

```python
# %% 4) Fit StandardScaler + IncrementalPCA (global stream, low RAM)
from sklearn.utils import shuffle
import gc

scaler = StandardScaler(with_mean=True, with_std=True)
for Xb, _ in stream_clean_global("train"):
    scaler.partial_fit(Xb.astype(np.float32, copy=False))
    del Xb; gc.collect()
print("Scaler fitted (global).")

ipca = IncrementalPCA(n_components=NCOMP, batch_size=BATCH)
for Xb, _ in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    if Xb_std.shape[0] >= NCOMP:
        ipca.partial_fit(Xb_std)
    del Xb, Xb_std; gc.collect()
print(f"IPCA fitted (n_components={NCOMP}).")
```

[5]

```
Scaler fitted (global).
IPCA fitted (n_components=64).
```

```python
# %% 5) Transform TRAIN/TEST in streams → compact features
Xtr_chunks, ytr_chunks = [], []
for Xb, yb in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    Xb_pca = ipca.transform(Xb_std)
    Xtr_chunks.append(Xb_pca.astype(np.float32, copy=False))
    ytr_chunks.append(yb)
Xtr = np.vstack(Xtr_chunks); ytr = np.concatenate(ytr_chunks)

Xte_chunks, yte_chunks = [], []
for Xb, yb in stream_clean_global("test"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    Xb_pca = ipca.transform(Xb_std)
    Xte_chunks.append(Xb_pca.astype(np.float32, copy=False))
    yte_chunks.append(yb)
Xte = np.vstack(Xte_chunks); yte = np.concatenate(yte_chunks)
```

[6]

Train Linear SVM

```python
from sklearn.svm import LinearSVC
```

[8]

```python
# %% 6) Train Linear SVM (on PCA features)
svm = LinearSVC(C=1.0, loss='squared_hinge', dual=False, max_iter=2000)
import time
print("Training Linear SVM...")
t_train = time.time()
svm.fit(Xtr, ytr)
train_secs = time.time() - t_train
print(f"Training completed in {train_secs/60:.2f} minutes")
```

```
Training Linear SVM...
Training completed in 0.04 minutes
```

Evaluation

```python
# %% 7) Evaluate on clean test set and save
t_eval = time.time()
yhat = svm.predict(Xte)
test_secs = time.time() - t_eval
```

[10]

Evaluation

```
# %% 7) Evaluate on clean test set and save
t_eval = time.time()
yhat = svm.predict(Xte)
test_secs = time.time() - t_eval

res_clean = evaluate_and_pack(yte, yhat, LABELS)
res_clean["TrainElapsedSeconds"] = round(train_secs, 2)
res_clean["TestElapsedSeconds"]  = round(test_secs, 2)

out_clean = os.path.join(ART, "results_svm_clean.json")
with open(out_clean, "w") as f:
    json.dump(res_clean, f, indent=2)
print("Saved:", out_clean)
```

[10]

```
Saved: D:\LocalUser\42177 Project\artifacts\results_svm_clean.json
```

```
# %% 8) Evaluate on degraded test sets (streaming)
conds = [d for d in os.listdir(ROOT_DEG) if os.path.isdir(os.path.join(ROOT_DEG, d))]
for c in conds:
    y_true, y_pred = [], []
    any_batch = False
    for Xb, yb in stream_degraded_global(c, "test", batch=BATCH):
        any_batch = True
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std)
        yhat_b = svm.predict(Xb_pca)
        y_true.extend(yb.tolist()); y_pred.extend(yhat_b.tolist())
    if not any_batch:
        print(f"Skip {c}: no files."); continue
    resg = evaluate_and_pack(np.array(y_true), np.array(y_pred), LABELS)
    out = os.path.join(ART, f"results_svm_{c}.json")
    with open(out, "w") as f: json.dump(resg, f, indent=2)
    print("Saved:", out)
```

[11]

Appendix 5 Script of classic model – trees

Image Processing and Pattern Recognition - Group Assignment

Feature Extraction + Decision Tree & Random Forest Classifiers

By: Shuntian Shi
Purpose: This script reads preprocessed image data from manifest.csv, trains both Decision Tree and Random Forest models, evaluates performance, and visualizes the results.

```
# %% 0) Imports and config
import os, json, time, random, gc
import numpy as np
import pandas as pd
from PIL import Image

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import IncrementalPCA
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support, accuracy_score

# Paths
MANIFEST   = r"D:\LocalUser\42177 Project\data\manifest.csv"
ROOT_CLEAN = r"D:\LocalUser\42177 Project\data\clean"
ROOT_DEG   = r"D:\LocalUser\42177 Project\data\degraded"
ART        = r"D:\LocalUser\42177 Project\artifacts"
os.makedirs(ART, exist_ok=True)

# Hyperparameters (aligned with LR/SVM/kNN low-RAM defaults)
TARGET_HW  = (256, 256)
SEED       = 42177
BATCH      = 64
NCOMP      = 64

# Trees
DT_PARAMS = dict(criterion="gini", max_depth=None, random_state=SEED)
RF_PARAMS = dict(n_estimators=200, max_depth=None, n_jobs=-1, random_state=SEED)

random.seed(SEED); np.random.seed(SEED)
EXTS = ('.png','.jpg','.jpeg','.tif','.tiff','.bmp')
```

[1] Python

```
# %% 1) Labels from manifest (fixed order)
df = pd.read_csv(MANIFEST)  # expects: id, filepath, label, subset, mag
LABELS = sorted(df['label'].unique().tolist())
LABELS
```

[2] Python

```
['babesia',
 'leishmania',
 'plasmodium',
 'toxoplasma1000x',
 'toxoplasma400x',
 'trichomonad',
 'trypanosome']
```

```
# %% 2) Streaming IO helpers
def _list_images(folder):
    if not os.path.isdir(folder): return []
    return [os.path.join(folder,f) for f in os.listdir(folder) if f.lower().endswith(EXTS)]

def stream_clean_global(subset, batch=BATCH, target=TARGET_HW):
    paths = []
    for lab in LABELS:
        d = os.path.join(ROOT_CLEAN, subset, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32) / 255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)

def stream_degraded_global(cond, subset="test", batch=BATCH, target=TARGET_HW):
    base = os.path.join(ROOT_DEG, cond, subset)
    if not os.path.isdir(base): return
    paths = []
    for lab in LABELS:
        d = os.path.join(base, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
```

[3]

```python
def stream_degraded_global(cond, subset="test", batch=BATCH, target=TARGET_HW):
    base = os.path.join(ROOT_DEG, cond, subset)
    if not os.path.isdir(base): return
    paths = []
    for lab in LABELS:
        d = os.path.join(base, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d, f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32) / 255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)
```

[3]

```python
# %% 3) Metrics packer (aligned with evaluate_model.mlx)
def evaluate_and_pack(y_true, y_pred, labels=LABELS):
    y_true = np.asarray(y_true).astype(str)
    y_pred = np.asarray(y_pred).astype(str)
    labels = [str(x) for x in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    acc = accuracy_score(y_true, y_pred)
    prec, rec, f1, _ = precision_recall_fscore_support(
        y_true, y_pred, labels=labels, average="macro", zero_division=0
    )
    return {
        "Accuracy": float(acc),
        "Precision": float(prec),
        "Recall": float(rec),
        "F1": float(f1),
        "ConfusionMatrix": cm.tolist(),
        "Labels": labels
    }
```

[4]

```python
# %% 4) Fit StandardScaler + IncrementalPCA (global stream, low RAM)
scaler = StandardScaler(with_mean=True, with_std=True)
for Xb, _ in stream_clean_global("train"):
    scaler.partial_fit(Xb.astype(np.float32, copy=False))
    del Xb; gc.collect()
print("Scaler fitted (global stream).")

ipca = IncrementalPCA(n_components=NCOMP, batch_size=BATCH)
for Xb, _ in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    if Xb_std.shape[0] >= NCOMP:
        ipca.partial_fit(Xb_std)
    del Xb, Xb_std; gc.collect()
print(f"IPCA fitted (n_components={NCOMP}).")
```

[6]

```
Scaler fitted (global stream).
IPCA fitted (n_components=64).
```

```python
# %% 5) Transform TRAIN/TEST in streams → compact features
Xtr_chunks, ytr_chunks = [], []
for Xb, yb in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    Xb_pca = ipca.transform(Xb_std)
    Xtr_chunks.append(Xb_pca.astype(np.float32, copy=False))
    ytr_chunks.append(yb)
Xtr = np.vstack(Xtr_chunks); ytr = np.concatenate(ytr_chunks)

Xte_chunks, yte_chunks = [], []
for Xb, yb in stream_clean_global("test"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    Xb_pca = ipca.transform(Xb_std)
    Xte_chunks.append(Xb_pca.astype(np.float32, copy=False))
    yte_chunks.append(yb)
Xte = np.vstack(Xte_chunks); yte = np.concatenate(yte_chunks)
```

[7]

```python
# %% 6) Train Decision Tree and Random Forest
dt = DecisionTreeClassifier(**DT_PARAMS)
rf = RandomForestClassifier(**RF_PARAMS)

print("Training Decision Tree...")
t0 = time.time(); dt.fit(Xtr, ytr); dt_secs = time.time() - t0
print(f"DT training time: {dt_secs/60:.2f} min")

print("Training Random Forest...")
t0 = time.time(); rf.fit(Xtr, ytr); rf_secs = time.time() - t0
print(f"RF training time: {rf_secs/60:.2f} min")
```

[8]

```
Training Decision Tree...
DT training time: 0.03 min
Training Random Forest...
RF training time: 0.06 min
```

```python
# %% 7) Evaluate on clean test and save JSON
# Decision Tree
t1 = time.time(); yhat_dt = dt.predict(Xte); dt_test = time.time() - t1
res_dt = evaluate_and_pack(yte, yhat_dt, LABELS)
res_dt["TrainElapsedSeconds"] = round(dt_secs, 2)
res_dt["TestElapsedSeconds"]  = round(dt_test, 2)
out_dt = os.path.join(ART, "results_tree_clean.json")
with open(out_dt, "w") as f: json.dump(res_dt, f, indent=2)
print("Saved:", out_dt)

# Random Forest
t1 = time.time(); yhat_rf = rf.predict(Xte); rf_test = time.time() - t1
res_rf = evaluate_and_pack(yte, yhat_rf, LABELS)
res_rf["TrainElapsedSeconds"] = round(rf_secs, 2)
res_rf["TestElapsedSeconds"]  = round(rf_test, 2)
out_rf = os.path.join(ART, "results_rf_clean.json")
with open(out_rf, "w") as f: json.dump(res_rf, f, indent=2)
print("Saved:", out_rf)
```

[9]

```
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_clean.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_clean.json
```

```python
# %% 8) Evaluate on degraded test sets (streaming)
conds = [d for d in os.listdir(ROOT_DEG) if os.path.isdir(os.path.join(ROOT_DEG, d))]

for c in conds:
    # Decision Tree
    y_true, y_pred = [], []
    any_batch = False
    for Xb, yb in stream_degraded_global(c, "test", batch=BATCH):
        any_batch = True
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std)
        y_pred.extend(dt.predict(Xb_pca).tolist())
        y_true.extend(yb.tolist())
    if any_batch:
        res = evaluate_and_pack(np.array(y_true), np.array(y_pred), LABELS)
        out = os.path.join(ART, f"results_tree_{c}.json")
        with open(out, "w") as f: json.dump(res, f, indent=2)
        print("Saved:", out)

    # Random Forest
    y_true, y_pred = [], []
    any_batch = False
    for Xb, yb in stream_degraded_global(c, "test", batch=BATCH):
        any_batch = True
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std)
        y_pred.extend(rf.predict(Xb_pca).tolist())
        y_true.extend(yb.tolist())
    if any_batch:
        res = evaluate_and_pack(np.array(y_true), np.array(y_pred), LABELS)
        out = os.path.join(ART, f"results_rf_{c}.json")
        with open(out, "w") as f: json.dump(res, f, indent=2)
        print("Saved:", out)
```

[10]

```
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_gaussian_blur_s1.0.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_gaussian_blur_s1.0.json
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_gaussian_blur_s2.0.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_gaussian_blur_s2.0.json
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_gaussian_noise_s15.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_gaussian_noise_s15.json
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_gaussian_noise_s5.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_gaussian_noise_s5.json
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_jpeg_q20.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_jpeg_q20.json
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_jpeg_q40.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_jpeg_q40.json
Saved: D:\LocalUser\42177 Project\artifacts\results_tree_jpeg_q60.json
Saved: D:\LocalUser\42177 Project\artifacts\results_rf_jpeg_q60.json
```

Appendix 6 Script of classic model – xgboost

42177 — XGBoost (clean + degraded), manifest-driven

- Uses data/clean splits and manifest.csv label order
- Global streaming + StandardScaler + IncrementalPCA (NCOMP=64) for consistency
- Saves JSON results to artifacts/

```
# %% 0) Setup
# If needed: pip install xgboost
!pip install xgboost

import os, json, time, random, gc
import numpy as np
import pandas as pd
from PIL import Image

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import IncrementalPCA
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support, accuracy_score
from xgboost import XGBClassifier

# Paths
MANIFEST   = r"D:\LocalUser\42177 Project\data\manifest.csv"
ROOT_CLEAN = r"D:\LocalUser\42177 Project\data\clean"
ROOT_DEG   = r"D:\LocalUser\42177 Project\data\degraded"
ART        = r"D:\LocalUser\42177 Project\artifacts"
os.makedirs(ART, exist_ok=True)

# Hyperparameters (low-RAM defaults)
TARGET_HW  = (256, 256)
SEED       = 42177
BATCH      = 64
NCOMP      = 64

random.seed(SEED); np.random.seed(SEED)
EXTS = ('.png','.jpg','.jpeg','.tif','.tiff','.bmp')
```

[1] ✓ 5.1s Python

```
Requirement already satisfied: xgboost in c:\users\xlton\anaconda3\lib\site-packages (3.1.1)
Requirement already satisfied: numpy in c:\users\xlton\anaconda3\lib\site-packages (from xgboost) (1.26.4)
Requirement already satisfied: scipy in c:\users\xlton\anaconda3\lib\site-packages (from xgboost) (1.13.1)
```

```
# %% 1) Labels from manifest (fixed order)
df = pd.read_csv(MANIFEST)  # expects: id, filepath, label, subset, mag
LABELS = sorted(df['label'].unique().tolist())
LABELS
```

[2] ✓ 0.0s Python

```
['babesia',
 'leishmania',
 'plasmodium',
 'toxoplasma1000x',
 'toxoplasma400x',
 'trichomonad',
 'trypanosome']
```

```
# %% 2) Streaming helpers
def stream_clean_global(subset, batch=BATCH, target=TARGET_HW):
    paths = []
    for lab in LABELS:
        d = os.path.join(ROOT_CLEAN, subset, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d,f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32)/255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)

def stream_degraded_global(cond, subset="test", batch=BATCH, target=TARGET_HW):
    base = os.path.join(ROOT_DEG, cond, subset)
    if not os.path.isdir(base):
        return
    paths = []
    for lab in LABELS:
        d = os.path.join(base, lab)
        if not os.path.isdir(d):
            continue
        paths += [(os.path.join(d,f), lab) for f in os.listdir(d) if f.lower().endswith(EXTS)]
    for i in range(0, len(paths), batch):
        chunk = paths[i:i+batch]
        Xb, yb = [], []
        for p, lab in chunk:
            im = Image.open(p).convert("RGB").resize(target)
            arr = np.asarray(im, dtype=np.float32)/255.0
            Xb.append(arr.reshape(-1)); yb.append(lab)
        if Xb:
            yield np.stack(Xb), np.array(yb)
```

[3]

```python
# %% 3) Metrics packer (aligned with evaluate_model.mlx)
def evaluate_and_pack(y_true, y_pred, labels=LABELS):
    y_true = np.asarray(y_true).astype(str)
    y_pred = np.asarray(y_pred).astype(str)
    labels = [str(x) for x in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    acc = accuracy_score(y_true, y_pred)
    prec, rec, f1, _ = precision_recall_fscore_support(
        y_true, y_pred, labels=labels, average="macro", zero_division=0
    )
    return {
        "Accuracy": float(acc),
        "Precision": float(prec),
        "Recall": float(rec),
        "F1": float(f1),
        "ConfusionMatrix": cm.tolist(),
        "Labels": labels
    }
```

[4] ✓ 0.0s

```python
# %% 4) Fit StandardScaler + IncrementalPCA on TRAIN (global streaming, low RAM)
scaler = StandardScaler(with_mean=True, with_std=True)
for Xb, _ in stream_clean_global("train"):
    scaler.partial_fit(Xb.astype(np.float32, copy=False))
    del Xb; gc.collect()
print("Scaler fitted (global stream).")

ipca = IncrementalPCA(n_components=NCOMP, batch_size=BATCH)
for Xb, _ in stream_clean_global("train"):
    Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
    if Xb_std.shape[0] >= NCOMP:
        ipca.partial_fit(Xb_std)
    del Xb, Xb_std; gc.collect()
print(f"IPCA fitted (n_components={NCOMP}).")
```

[5] ✓ 13m 17.3s

```
Scaler fitted (global stream).
IPCA fitted (n_components=64).
```

```python
# %% 5) Transform TRAIN/VAL/TEST in streams → compact features
def transform_split(split):
    Xc, yc = [], []
    for Xb, yb in stream_clean_global(split):
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std).astype(np.float32, copy=False)
        Xc.append(Xb_pca); yc.append(yb)
    X = np.vstack(Xc); y = np.concatenate(yc)
    return X, y

Xtr, ytr = transform_split("train")
Xva, yva = transform_split("val")
Xte, yte = transform_split("test")

Xtr.shape, Xva.shape, Xte.shape
```

[6] ✓ 2m 31.1s

```
((14390, 64), (4810, 64), (4812, 64))
```

Train XGBoost (multiclass) — with label encoding

```python
# %% 6) Train XGBoost (multiclass) — with label encoding

from sklearn.preprocessing import LabelEncoder

# after LABELS is defined and Xtr,ytr,Xva,yva,Xte,yte are built (strings)
le = LabelEncoder()
le.classes_ = np.array(LABELS)  # keep consistent label order

ytr_i = le.transform(ytr)
yva_i = le.transform(yva)
yte_i = le.transform(yte)

# train
xgb = XGBClassifier(
    objective="multi:softmax",
    num_class=len(LABELS),
    n_estimators=400,
    max_depth=8,
    learning_rate=0.05,
    subsample=0.9,
    colsample_bytree=0.9,
    reg_lambda=1.0,
    reg_alpha=0.0,
    tree_method="hist",
    random_state=SEED,
    n_jobs=-1
)

print("Training XGBoost...")
t_train = time.time()
```

[7]

```
print("Training XGBoost...")
t_train = time.time()
xgb.fit(Xtr, ytr_i, eval_set=[(Xva, yva_i)], verbose=False)
train_secs = time.time() - t_train
print(f"Training completed in {train_secs/60:.2f} minutes")

# Predict and decode labels back to strings for evaluation
t_eval = time.time()
yhat_i = xgb.predict(Xte)
test_secs = time.time() - t_eval
yhat = le.inverse_transform(yhat_i)
print(f"Clean test time: {test_secs:.1f} s")
```

[7] ✓ 10.8s

```
Training XGBoost...
Training completed in 0.18 minutes
Clean test time: 0.0 s
```

```
# %% 7) EVALUATE — clean test set
# Predict (ints) then decode to strings
t_eval = time.time()
yhat_i = xgb.predict(Xte)
test_secs = time.time() - t_eval
yhat = le.inverse_transform(yhat_i)          # strings

# --- sanity checks
print("Xte shape:", Xte.shape)
print("len(yte):", len(yte), "len(yhat):", len(yhat))
print("labels:", LABELS)

yte  = np.asarray(yte).astype(str)
yhat = np.asarray(yhat).astype(str)
LABELS = [str(x) for x in LABELS]

# show mismatches if any
uniq_yte  = sorted(set(yte))
uniq_yhat = sorted(set(yhat))
missing_from_LABELS_yte  = sorted(set(uniq_yte)  - set(LABELS))
missing_from_LABELS_yhat = sorted(set(uniq_yhat) - set(LABELS))
print("unique yte :", uniq_yte)
print("unique yhat:", uniq_yhat)
print("missing_from_LABELS_yte :", missing_from_LABELS_yte)
print("missing_from_LABELS_yhat:", missing_from_LABELS_yhat)

assert len(yte) == len(yhat) and len(yte) > 0
assert set(uniq_yte).issubset(set(LABELS))
assert set(uniq_yhat).issubset(set(LABELS))

# Evaluate and save
res_clean = evaluate_and_pack(yte, yhat, LABELS)
res_clean["TrainElapsedSeconds"] = round(train_secs, 2)
res_clean["TestElapsedSeconds"]  = round(test_secs, 2)

out_clean = os.path.join(ART, "results_xgb_clean.json")
with open(out_clean, "w") as f:
    json.dump(res_clean, f, indent=2)

print("Saved:", out_clean)
print("Accuracy: %.2f%%  F1: %.2f%%" % (100*res_clean["Accuracy"], 100*res_clean["F1"]))
```

[8] ✓ 0.0s

```
Xte shape: (4812, 64)
len(yte): 4812 len(yhat): 4812
labels: ['babesia', 'leishmania', 'plasmodium', 'toxoplasma1000x', 'toxoplasma400x', 'trichomonad', 'trypanosome']
unique yte : ['babesia', 'leishmania', 'plasmodium', 'toxoplasma1000x', 'toxoplasma400x', 'trichomonad', 'trypanosome']
unique yhat: ['babesia', 'leishmania', 'plasmodium', 'toxoplasma1000x', 'toxoplasma400x', 'trichomonad', 'trypanosome']
missing_from_LABELS_yte : []
missing_from_LABELS_yhat: []
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_clean.json
Accuracy: 96.07%  F1: 95.45%
```

```
print(f"Precision: {100*res_clean['Precision']:.2f}%  Recall: {100*res_clean['Recall']:.2f}%")
```

[9] ✓ 0.0s

```
Precision: 97.23%  Recall: 93.90%
```

```
import csv

# %% 8) Evaluate on DEGRADED test sets (CSV per condition)
def evaluate_degraded_condition(cond, subset="test"):
    y_true_all, y_pred_all = [], []
    t0 = time.time()

    for Xb, yb in stream_degraded_global(cond, subset=subset, batch=BATCH, target=TARGET_HW):
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std).astype(np.float32, copy=False)

        yhat_i = xgb.predict(Xb_pca)
        yhat = le.inverse_transform(yhat_i)

        y_true_all.append(yb.astype(str))
        y_pred_all.append(yhat.astype(str))

        del Xb, yb, Xb_std, Xb_pca, yhat_i, yhat
        gc.collect()
```

[10]

```python
import csv

# %% 8) Evaluate on DEGRADED test sets (CSV per condition)
def evaluate_degraded_condition(cond, subset="test"):
    y_true_all, y_pred_all = [], []
    t0 = time.time()

    for Xb, yb in stream_degraded_global(cond, subset=subset, batch=BATCH, target=TARGET_HW):
        Xb_std = scaler.transform(Xb.astype(np.float32, copy=False))
        Xb_pca = ipca.transform(Xb_std).astype(np.float32, copy=False)

        yhat_i = xgb.predict(Xb_pca)
        yhat = le.inverse_transform(yhat_i)

        y_true_all.append(yb.astype(str))
        y_pred_all.append(yhat.astype(str))

        del Xb, yb, Xb_std, Xb_pca, yhat_i, yhat
        gc.collect()

    if not y_true_all:
        print(f"[WARN] No samples found for degraded condition '{cond}'")
        return None

    y_true = np.concatenate(y_true_all)
    y_pred = np.concatenate(y_pred_all)

    metrics = evaluate_and_pack(y_true, y_pred, LABELS)
    metrics["TestElapsedSeconds"] = round(time.time() - t0, 2)
    return metrics

degraded_conditions = [
    c for c in os.listdir(ROOT_DEG)
    if os.path.isdir(os.path.join(ROOT_DEG, c))
]

all_deg_results = {}
for c in degraded_conditions:
    print(f"Evaluating degraded condition: {c}")
    res_deg = evaluate_degraded_condition(c, subset="test")
    if res_deg is None:
        print(f"Skipped: {c}")
        continue

    all_deg_results[c] = res_deg

    # flatten for CSV
    out = os.path.join(ART, f"results_xgb_{c}.csv")
    with open(out, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["Metric", "Value"])
        writer.writerow(["Accuracy", res_deg["Accuracy"]])
        writer.writerow(["Precision", res_deg["Precision"]])
        writer.writerow(["Recall", res_deg["Recall"]])
        writer.writerow(["F1", res_deg["F1"]])
        writer.writerow(["TestElapsedSeconds", res_deg["TestElapsedSeconds"]])
        # confusion matrix as JSON string to keep 1-row format
        writer.writerow(["Labels", json.dumps(res_deg["Labels"])])
        writer.writerow(["ConfusionMatrix", json.dumps(res_deg["ConfusionMatrix"])])
    print(f"Saved: {out}")

# optional index (still JSON, per run)
idx_path = os.path.join(ART, "results_xgb_degraded_all.json")
with open(idx_path, "w") as f:
    json.dump(all_deg_results, f, indent=2)
print(f"Saved index of degraded results: {idx_path}")
```

[10] ✓ 5m 59.5s Pytho

```
Evaluating degraded condition: gaussian_blur_s1.0
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_gaussian_blur_s1.0.csv
Evaluating degraded condition: gaussian_blur_s2.0
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_gaussian_blur_s2.0.csv
Evaluating degraded condition: gaussian_noise_s15
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_gaussian_noise_s15.csv
Evaluating degraded condition: gaussian_noise_s5
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_gaussian_noise_s5.csv
Evaluating degraded condition: jpeg_q20
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_jpeg_q20.csv
Evaluating degraded condition: jpeg_q40
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_jpeg_q40.csv
Evaluating degraded condition: jpeg_q60
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_jpeg_q60.csv
Evaluating degraded condition: motion_blur_k5
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_motion_blur_k5.csv
Evaluating degraded condition: resolution_x2
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_resolution_x2.csv
Evaluating degraded condition: resolution_x4
Saved: D:\LocalUser\42177 Project\artifacts\results_xgb_resolution_x4.csv
Saved index of degraded results: D:\LocalUser\42177 Project\artifacts\results_xgb_degraded_all.json
```

Appendix 7 Script of deep learning model – CNN

CNN

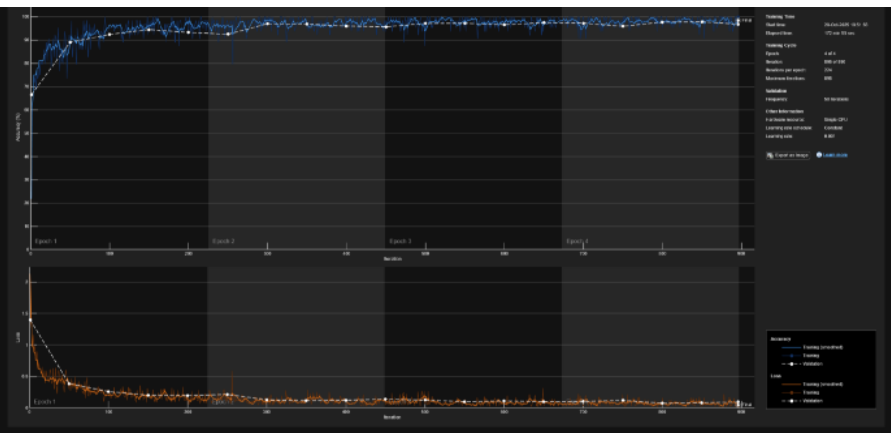
```
%% TRAIN CNN ON CLEAN DATA (RGB, 256x256) — train_cnn_clean.mlx
clear; clc;
```

```
% DATA
% Paths
rootClean     = 'D:\LocalUser\42177 Project\data\clean';
artifactsDir  = 'D:\LocalUser\42177 Project\artifacts';
evalPath      = 'D:\LocalUser\42177 Project\evaluate_model.mlx';  % updated path
if ~exist(artifactsDir,'dir'), mkdir(artifactsDir); end

% Add the folder (not the .mlx file) to MATLAB path
addpath(fileparts(evalPath));

% Seed for reproducibility
rng(42177);

% Datastores
imdsTrain = imageDatastore(fullfile(rootClean,'train'),"IncludeSubfolders",true,"LabelSource","foldernames");
imdsVal   = imageDatastore(fullfile(rootClean,'val'),  "IncludeSubfolders",true,"LabelSource","foldernames");
imdsTest  = imageDatastore(fullfile(rootClean,'test'), "IncludeSubfolders",true,"LabelSource","foldernames");

classes   = categories(imdsTrain.Labels);
inputSize = [256 256 3];

% Data augmentation
aug = imageDataAugmenter('RandXReflection',true,'RandRotation',[-10 10]);
adsTrain = augmentedImageDatastore(inputSize, imdsTrain, 'DataAugmentation', aug);
adsVal   = augmentedImageDatastore(inputSize, imdsVal);
adsTest  = augmentedImageDatastore(inputSize, imdsTest);
```

Define the convolutional neural network architecture

```
% MODEL
layers = [
    imageInputLayer(inputSize,"Normalization","none")
    convolution2dLayer(3,32,"Padding","same")
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,64,"Padding","same")
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,128,"Padding","same")
    batchNormalizationLayer
    reluLayer

    globalAveragePooling2dLayer
    fullyConnectedLayer(numel(classes))
    softmaxLayer
    classificationLayer
];

% Training options
options = trainingOptions('adam', ...
    'MiniBatchSize',64, ...
    'MaxEpochs',4, ...   % operation compute capacity can only afford 4
    'InitialLearnRate',1e-3, ...
    'ValidationData',adsVal, ...
    'Shuffle','every-epoch', ...
    'Verbose',true, ...
    'Plots','training-progress');

%% Train
t0 = tic;
[net, trainInfo] = trainNetwork(adsTrain, layers, options);
```

```
Training on single CPU.
|=========================================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validation  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss     |      Rate       |
|=========================================================================================================================|
|       1 |           1 |       00:02:47 |       21.88% |       66.53% |       2.1333 |       1.3968 |          0.0010 |
|       1 |          50 |       00:10:41 |       90.62% |       89.04% |       0.3949 |       0.3873 |          0.0010 |
|       1 |         100 |       00:18:52 |       95.31% |       92.43% |       0.2157 |       0.2606 |          0.0010 |
|       1 |         150 |       00:26:59 |       95.31% |       94.51% |       0.2242 |       0.2021 |          0.0010 |
|       1 |         200 |       00:34:49 |       92.19% |       93.28% |       0.2108 |       0.1983 |          0.0010 |
|       2 |         250 |       00:42:54 |       96.88% |       92.52% |       0.1383 |       0.2157 |          0.0010 |
|       2 |         300 |       00:50:57 |       98.44% |       97.03% |       0.1103 |       0.1300 |          0.0010 |
|       2 |         350 |       00:58:49 |       95.31% |       96.86% |       0.1274 |       0.1177 |          0.0010 |
|       2 |         400 |       01:06:50 |      100.00% |       96.03% |       0.0657 |       0.1253 |          0.0010 |
|       3 |         450 |       01:14:38 |       98.44% |       95.70% |       0.0515 |       0.1392 |          0.0010 |
|       3 |         500 |       01:22:36 |       98.44% |       97.15% |       0.0652 |       0.1262 |          0.0010 |
|       3 |         550 |       01:32:26 |      100.00% |       97.21% |       0.0522 |       0.1010 |          0.0010 |
|       3 |         600 |       01:45:18 |       95.31% |       96.59% |       0.1145 |       0.1114 |          0.0010 |
|       3 |         650 |       01:58:11 |       98.44% |       97.46% |       0.0698 |       0.1017 |          0.0010 |
|       4 |         700 |       02:08:04 |      100.00% |       97.28% |       0.0309 |       0.1028 |          0.0010 |
|       4 |         750 |       02:16:05 |       96.88% |       95.93% |       0.2178 |       0.1218 |          0.0010 |
|       4 |         800 |       02:24:12 |       95.31% |       97.73% |       0.1631 |       0.0748 |          0.0010 |
|       4 |         850 |       02:37:04 |      100.00% |       97.84% |       0.0385 |       0.0823 |          0.0010 |
|       4 |         896 |       02:49:14 |      100.00% |       96.82% |       0.0374 |       0.0995 |          0.0010 |
|=========================================================================================================================|
Training finished: Max epochs completed.
```



```
trainSecs = toc(t0);
fprintf('Training time: %.1f min\n', trainSecs/60);
```

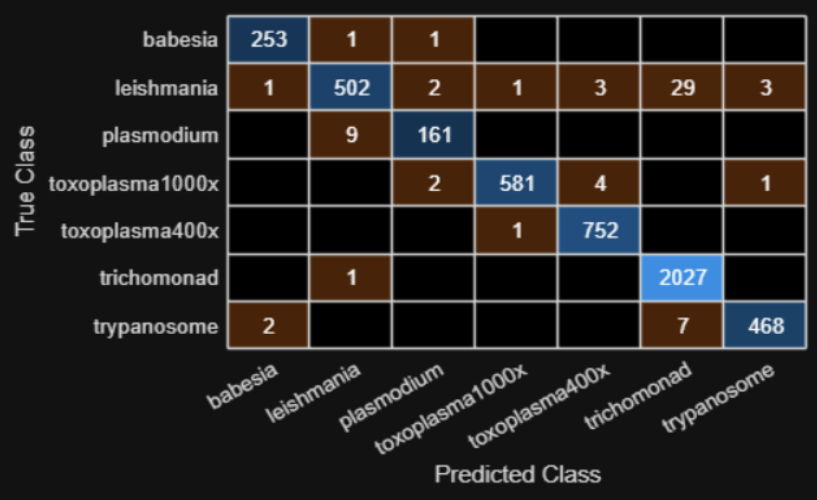
```
Training time: 190.3 min
```

```
% Save model
save(fullfile(artifactsDir,'cnn_model.mat'),'net','classes','trainInfo','trainSecs');
```

Evaluation

```
%% Load trained model
```

```
Clean test time: 70.7 s
Accuracy: 98.59%
Precision: 97.64%  Recall: 98.54%  F1: 98.08%
```




```
load('D:\LocalUser\42177 Project\artifacts\cnn_model.mat', 'net', 'classes');

%% Recreate datastores for test set
rootClean = 'D:\LocalUser\42177 Project\data\clean';
imdsTest  = imageDatastore(fullfile(rootClean,'test'), ...
    "IncludeSubfolders", true, "LabelSource", "foldernames");
adsTest   = augmentedImageDatastore([256 256 3], imdsTest);

%% Evaluate on clean test set
t1 = tic;
[YPred, ~] = classify(net, adsTest);
testSecs = toc(t1);
fprintf('Clean test time: %.1f s\n', testSecs);
```

```
Clean test time: 72.0 s
```

```
y_true = string(imdsTest.Labels);
y_pred = string(YPred);

%% Evaluate model
results = evaluate_model(y_true, y_pred, string(classes));
```

```
Accuracy: 98.59%
Precision: 97.64%  Recall: 98.54%  F1: 98.08%
```

```
%% Save results as .csv
artifactsDir = 'D:\LocalUser\42177 Project\artifacts';
if ~exist(artifactsDir, 'dir')
    mkdir(artifactsDir);
end

% Convert confusion matrix to table and save
confMatTable = array2table(results.ConfusionMatrix, ...
    'VariableNames', results.Labels, ...
    'RowNames', results.Labels);
writetable(confMatTable, fullfile(artifactsDir, 'results_cnn_confmx.csv'), 'WriteRowNames', true);

% Save summary metrics (accuracy, precision, recall, etc.)
summaryTable = struct2table(rmfield(results, {'ConfusionMatrix','Labels'}));
summaryTable.TestTime = testSecs;
writetable(summaryTable, fullfile(artifactsDir, 'results_cnn_clean.csv'));

fprintf('Results saved to CSV in %s\n', artifactsDir);
```

```
Results saved to CSV in D:\LocalUser\42177 Project\artifacts
```

Appendix 8 Script of CNN model evaluation function

Evaluation

"This file preserves the original evaluation logic used for reference. All models now implement equivalent Python-based evaluation internally."

Usage

results = evaluate_model(y_true, y_pred, labels)

y_true and y_pred are categorical arrays or string vectors of class labels.

labels is the ordered list of all class names.

Purpose

This script provides a **common evaluation function** for all models in the 42177 Image Processing and Pattern Recognition project.

Every team member should use this same function to calculate:

- Accuracy
- Precision
- Recall
- F1-score (macro average)
- Confusion matrix

Example:

labels = ["babesia","leishmania","plasmodium", ...

"toxoplasma1000x","toxoplasma400x","trichomonad","trypanosome"];

Replace with your model's true and predicted labels

results = evaluate_model(y_true, y_pred, labels);

Save results (optional)

save('artifacts/results_<your_model>.mat','results');

```

**Expected Inputs**

- `y_true` : vector of true class labels (categorical or string)
- `y_pred` : vector of predicted labels (same size as y_true)
- `labels` : ordered list of all class names

**Output**

- Returns a structure containing:
- Accuracy
- Precision
- Recall
- F1
- ConfusionMatrix
- Labels

```
function results = evaluate_model(y_true, y_pred, labels)
% evaluate_model
% Inputs:
%   y_true, y_pred - string arrays of true and predicted labels
%   labels - string array of all class names (ordered)
% Output:
%   results struct with Accuracy, Precision, Recall, F1, ConfusionMatrix, Labels

    % --- Force all to string type ---
    y_true = string(y_true);
    y_pred = string(y_pred);
    labels = string(labels);

    % --- Confusion matrix using strings ---
    cm = confusionmat(y_true, y_pred, 'Order', labels);

    % --- Compute metrics ---
    acc = sum(diag(cm)) / sum(cm(:));
    prec = diag(cm) ./ max(sum(cm,2),1);
    rec = diag(cm) ./ max(sum(cm,1)',1);
    f1   = 2 * (prec .* rec) ./ max(prec + rec, eps);

    % --- Results struct ---
    results = struct(...
        'Accuracy', acc, ...
        'Precision', mean(prec,'omitnan'), ...
        'Recall', mean(rec,'omitnan'), ...
        'F1', mean(f1,'omitnan'), ...
        'ConfusionMatrix', cm, ...
        'Labels', labels);

    % --- Display summary ---
    fprintf('Accuracy: %.2f%%\n', 100*acc);
    fprintf('Precision: %.2f%% Recall: %.2f%% F1: %.2f%%\n', ...
        100*results.Precision, 100*results.Recall, 100*results.F1);
end
```
```

Appendix 9 Script of results visualisation

visualise_results.py

Visualise metrics for classic models (exclude CNN). Reads JSONs from artifacts/.

Generate + Code + Markdown

```python
import os, json
import numpy as np
import matplotlib.pyplot as plt

# ---- config ----
ART = r"D:\LocalUser\42177 Project\artifacts"
MODELS = ["logreg", "svm", "knn", "tree", "rf", "xgb"]  # no 'cnn'
METRICS = ["Accuracy", "Precision", "Recall", "F1"]
```

[1] Python

```python
# ---- i/o helpers ----
def load_json_result(model: str, suffix: str = "clean"):
    path = os.path.join(ART, f"results_{model}_{suffix}.json")
    if not os.path.exists(path):
        return None
    with open(path) as f:
        return json.load(f)


def load_many(models, suffix="clean"):
    out = {}
    for m in models:
        r = load_json_result(m, suffix=suffix)
        if r is not None:
            out[m] = r
    return out
```

[2] Python

```python
# ---- plotting ----
def bar_plot(results: dict, metric: str, title_suffix: str = "Clean"):
    names = [m.upper() for m in results.keys()]
    vals = [results[m][metric] for m in results.keys()]
    plt.figure(figsize=(7, 4))
    plt.bar(names, vals)
    plt.ylim(0, 1)
    plt.title(f"{metric} — {title_suffix}")
    plt.ylabel(metric)
    plt.xlabel("Model")
    plt.tight_layout()
    plt.show()


def plot_confusion_matrix(cm: np.ndarray, labels: list, title: str):
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation="nearest")
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(labels))
    plt.xticks(tick_marks, labels, rotation=45, ha="right")
    plt.yticks(tick_marks, labels)
    # annotate
    thresh = cm.max() if cm.size else 0
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            val = int(cm[i, j])
            color = "white" if thresh and val > thresh * 0.6 else "black"
            plt.text(j, i, str(val), ha="center", va="center", color=color, fontsize=8)
    plt.ylabel("True")
    plt.xlabel("Predicted")
    plt.tight_layout()
    plt.show()
```

[3] Python

```python
# ---- 1) load clean results (exclude CNN) ----
results_clean = load_many(MODELS, suffix="clean")
if not results_clean:
    raise SystemExit("No clean results JSONs found in artifacts/.")
```

[4] Python

```python
# ---- 2) bar charts for clean metrics ----
for metric in METRICS:
    bar_plot(results_clean, metric, title_suffix="Clean")
```