B Tech Computer Science and Engineering (Data Science) 311

Program Semester – VI

Image and Video Processing Project Report on

Comparative Study of Machine Learning Classifiers for Fake Currency Detection

Developed by
L006 – Anadi Jain
L021 – Krishiv Panchal
L022 – Krutarth Tailor
L027 – Nish Shah

Faculty In charge
Dr. Dhirendra Mishra
SVKM's NMIMS University

MUKESH PATEL SCHOOL OF TECHNOLOGY MANAGEMENT & ENGINEERING
Vile Parle (W), Mumbai 2024-25

Acknowledgment

We would like to extend our deepest gratitude to all those who provided their invaluable support and contributions to this research project. First and foremost, our sincere appreciation goes to Dr. Dhirendra Mishra whose expertise and insightful guidance were instrumental in shaping both the direction and outcome of this study. Their patience and unwavering support at every stage of the research were truly inspiring.

Our heartfelt thanks go to our colleagues, whose camaraderie and assistance throughout this journey made the challenging moments more bearable. We also wish to acknowledge the contributions of the research study, who provided us with a direction of research and help build our foundation of understanding in brain tumor detection, further enriching this work.

Lastly, we would like to express our appreciation to the participants of this study, who generously contributed their time and insights, playing a crucial role in the success of this research.

Table of Contents

List of Tables

Abstract

The proliferation of counterfeit currency poses a significant challenge to financial systems worldwide, necessitating robust and efficient detection methods. This study presents a comparative analysis of twelve machine learning classifiers for fake currency detection, utilizing a synthetic dataset of 460 Indian currency images (400 real and 60 fake) across four denominations (₹50, ₹100, ₹200, ₹500). Eight region-specific features, including security thread edge density, watermark gradient magnitude, and microlettering contrast, were extracted from the front side of the notes to train the models. The classifiers were evaluated using both a single train-test split (80:20 ratio) and 5-fold cross-validation, with performance assessed through accuracy, precision, recall, and F1-score metrics. Results indicate that Naive Bayes achieved the highest mean cross-validation accuracy of 96.92% and an F1-score of 93.00%, while Artificial Neural Networks (ANN) and Random Forest both recorded a single split accuracy of 98.08%. Feature importance analysis revealed that edge and texture-based features, such as security thread edge density and microlettering contrast, were critical in distinguishing genuine notes from counterfeits. Despite the promising performance, the study's reliance on synthetic data and limited dataset size highlights the need for future work with real-world images. This research contributes to enhancing automated counterfeit detection systems by identifying effective machine learning approaches and key discriminative features.

Chapter 1 Introduction

The increasing sophistication of counterfeit currency production has emerged as a critical threat to global economic stability, necessitating advanced detection techniques to safeguard financial systems. This chapter introduces the context of fake currency detection, outlines the scope of the research, and presents the approach adopted in this study to address the challenge using machine learning and image processing techniques.

1.1 Background

Counterfeit currency undermines the integrity of financial institutions and commercial sectors by introducing fraudulent notes into circulation, often indistinguishable to the untrained eye. Traditional detection methods, reliant on manual inspection or basic security feature verification, struggle to keep pace with evolving forgery techniques that exploit high-quality printing and subtle modifications to security elements such as watermarks, security threads, and microlettering. The advent of machine learning (ML) and image processing technologies has opened new avenues for automated detection systems, offering improved accuracy, efficiency, and scalability over conventional approaches. Recent studies have demonstrated the potential of ML classifiers, such as Support Vector Machines (SVM), Naive Bayes, and Random Forests, in identifying counterfeit notes by analyzing visual and statistical features extracted from currency images. However, many existing solutions focus on limited classifier sets or small datasets, often lacking comprehensive comparisons or real-world applicability. This research builds on these foundations, aiming to evaluate a diverse range of ML classifiers to enhance the reliability and interpretability of fake currency detection systems.

1.2 Research Scope

This study focuses on the detection of counterfeit Indian currency notes across four denominations: ₹50, ₹100, ₹200, and ₹500. A synthetic dataset comprising 460 images (400 real and 60 fake) was created to simulate real-world currency notes and their counterfeit counterparts, with modifications applied to key security features such as security threads, watermarks, and microlettering. The research evaluates twelve machine learning classifiers—including SVM, ANN, Naive Bayes, Random Forest, and others—using a feature set of eight region-specific attributes extracted from the front side of the notes. Performance is assessed through two evaluation methods: a single train-test split (80:20 ratio) and 5-fold cross-

validation, with metrics including accuracy, precision, recall, and F1-score. Additionally, feature importance analysis is conducted to identify the most discriminative features for counterfeit detection. The scope is limited to synthetic data and front-side features, excluding real-world variability such as lighting conditions or physical wear, which are identified as areas for future exploration.

1.3 Our approach

Our approach integrates image processing and machine learning to develop a robust fake currency detection system. The methodology begins with the generation of a synthetic dataset, where real images are sourced and fake images are created by applying controlled modifications to security features, mimicking common counterfeiting techniques. Image preprocessing steps, including cropping and scaling, standardize the dataset for feature extraction. Eight features—such as security thread edge density, watermark gradient magnitude, microlettering contrast, and whole image entropy—are extracted from predefined regions of interest on the front side of each note. These features form the input for training twelve ML classifiers, selected for their diverse pattern recognition capabilities. The classifiers are evaluated using both a single train-test split and 5-fold cross-validation to ensure robust performance assessment. Feature importance analysis is performed for top-performing models (SVM and Random Forest) to enhance interpretability by identifying key discriminative features. This approach aims to provide a comprehensive comparison of ML techniques, offering insights into their effectiveness and potential for practical deployment in counterfeit detection systems.

Chapter 2 Literature Survey

The escalating challenge of counterfeit currency has driven extensive research into automated detection systems using machine learning (ML) and image processing. This chapter reviews prior studies, analyzing their methodologies, datasets, outcomes, and limitations to position our work within the field. It offers an in-depth analysis of key approaches, summarizes their results, and concludes with insights guiding our research.

2.1 In-depth analysis

The application of ML and image processing to fake currency detection has gained traction as counterfeiting techniques evolve. Below, we analyze eight key studies referenced in our research, followed by a summary table for clarity.

Selvakumar et al. (2025) proposed a system using K-Nearest Neighbors (KNN), Support Vector Classifier (SVC), and Gradient Boosting Classifier (GBC), combined with wavelet transform-based image processing. Their dataset from the UCI Machine Learning Repository included 1,372 samples (762 fake, 610 real), focusing on four features. The study lacked cross-validation and deep learning exploration, limiting its robustness assessment.

Kishore Kumar et al. (2024) developed an ensemble decision tree approach for Indian currency detection, using 1,050 images. They compared Ensemble Decision Trees (EDT) and Random Linear Coding (RLC), but specific classifier details and accuracies were not fully detailed, restricting direct comparisons.

Santhiya et al. (2025) evaluated Logistic Regression, SVM, KNN, Gradient Boosting, and Naive Bayes on the UCI dataset (1,372 samples). Their focus on ensemble methods and feature analysis provided valuable insights, though overall accuracies were not reported, emphasizing F1-scores instead.

Narake et al. (2025) introduced a hybrid model combining Convolutional Neural Networks (CNN) and Random Forest for currency classification and counterfeit detection. While innovative, the study's dataset and performance metrics were not fully specified in the available text, limiting its scope evaluation.

Vystruri et al. (2025) applied Naive Bayes, SVC, Gradient Descent, and Neural Networks to 10,000 Indian currency images captured under UV light. This large dataset offered real-world potential, but the absence of cross-validation and detailed F1-scores constrained its generalizability assessment.

Lakshmanan et al. (2024) explored real-time fake currency detection using deep learning,

achieving notable results. However, the dataset size and specific methodologies were not fully elaborated in the referenced context, suggesting a focus on practical deployment over comprehensive evaluation.

Yadav et al. (2021) assessed SVM, Random Forest, Logistic Regression, Naive Bayes, Decision Tree, and KNN on the UCI dataset (1,372 samples). Their high accuracies with KNN and Decision Tree highlighted supervised learning potential, though F1-scores were unreported, and the dataset's synthetic nature was a limitation.

Shinde et al. (2023) developed an SVM-based system for Indian currency, extracting features like security threads and watermarks from UV-light images using MATLAB. Their unspecified dataset size and reliance on specialized imaging suggest high accuracy but limited practicality for diverse scenarios.

Our study addresses gaps in these works by evaluating twelve classifiers on a synthetic dataset of 460 images (400 real, 60 fake), incorporating 5-fold cross-validation, and analyzing feature importance. The following table summarizes these studies:

Table 1: Literature Summary

Reference	**Methodology**	**Dataset**	**Classifiers Used**	**Accuracy**	**Limitations**	**Future Scope**
Selvakumar et al. (2025) [1]	Image processing (Wavelet Transform) + ML	1,372 samples (UCI, 762 fake, 610 real)	KNN, SVC, Gradient Boosting	KNN: 99.82% (single split)	Small dataset, no cross-validation, no deep learning	Use larger real-world dataset, explore deep learning
Kishore Kumar et al. (2024) [2]	Image processing + ML (EDT vs. RLC)	1,050 images (size, color, special attributes)	Ensemble Decision Tree, Ridge Linear	EDT: 88.47%, RLC: 85.76% (mean)	Small dataset, moderate accuracy, no deep learning	Explore deep learning, expand dataset
Santhiya et al. (2025) [3]	Feature extraction + ML comparison	1,372 samples (UCI, 762 fake, 610 real)	Logistic Regression, SVM, KNN, Gradient Boosting, Naive Bayes	Not reported (F1-scores: GBC: 1.0, KNN: 0.9957)	Small dataset, no real-world variability, no overall accuracy reported	Use diverse dataset, report overall accuracy
Narale et al. (2025) [4]	Image processing (GLCM, LBP, Canny) + Hybrid ML (CNN + RF + XGBoost)	Not specified (real + fake images)	CNN, Random Forest, XGBoost	Not reported	No empirical results, dataset size not specified	Validate with real dataset, report performance metrics
Vystruri et al. (2025) [5]	Image processing (UV	10,000 Indian	Naive Bayes, SVC,	NN: 92%, GD: 87%,	High computational	Optimize algorithms,

Reference	Methodology	Dataset	Classifiers Used	Accuracy	Limitations	Future Scope
	light) + ML	currency images	Gradient Descent, Neural Networks	SVC: 76%, NB: 68%	cost for NN, low performance of NB	explore ensemble methods
Lakkshmanan et al. (2024) [6]	Image processing + Deep Learning	Not specified (real + fake notes)	CNN, ANN, SVM, KNN	CNN: 97.8%, ANN: 95.2%, SVM: 93.1%, KNN: 89.7%	Dependency on high-quality images, high computational cost for CNN	Optimize CNN for real-time use, improve robustness to low-quality images
Yadav et al. (2021) [7]	Feature extraction + ML comparison	1,372 samples (UCI, 762 fake, 610 real)	SVM, Random Forest, Logistic Regression, Naive Bayes, Decision Tree, KNN	KNN: 100% (80:20), DT: 100% (60:40), NB: 84-86%	Small dataset, no real-world variability, no deep learning	Explore deep learning or hybrid models, use larger datasets
Shinde et al. (2023) [8]	Image processing (UV light) + SVM	Not specified (Indian currency images)	SVM	SVM: 99.9% (avg. detection rate: 88.32%)	Dataset size not specified, reliance on UV light imaging	Expand dataset, explore alternative imaging techniques

2.2 Results

The results from prior studies vary based on methodologies and datasets. Selvakumar et al. (2025) reported a KNN accuracy of 99.82% on a single train-test split, with an F1-score of 99.92%, outperforming our KNN (92.31% single split, 88.85% cross-validation) due to a simpler feature set. Santhiya et al. (2025) achieved a Gradient Boosting F1-score of 100.00% and a Naive Bayes F1-score of 83.73%, while our Naive Bayes outperformed theirs with 98.08% accuracy (single split) and 96.92% (cross-validation), F1-score 93.00%. Vystruri et al. (2025) recorded a Neural Network accuracy of 92.00%, lower than our ANN's 98.08% (single split) and 96.15% (cross-validation), despite their larger dataset. Shinde et al. (2023) achieved an SVM accuracy of 99.90%, surpassing our SVM's 96.54% (both single split and cross-validation), aided by UV imaging. Yadav et al. (2021) reported perfect accuracies (100.00%) for KNN and Decision Tree, contrasting with our lower Logistic Regression performance (76.92%), highlighting dataset and feature differences.

2.3 Conclusion

The literature reveals a trend toward ML-based counterfeit detection, with classifiers like SVM, KNN, and Naive Bayes showing promise. However, inconsistencies in dataset size, evaluation methods, and feature complexity limit direct comparisons. High accuracies in studies like Selvakumar et al. (2025) and Shinde et al. (2023) suggest potential for specific techniques, yet their lack of cross-validation and real-world data testing raises robustness concerns. Our study builds on these findings by evaluating a broader range of twelve classifiers, achieving competitive results (e.g., Naive Bayes at 96.92% cross-validation accuracy), and addressing interpretability through feature importance analysis. The reliance on synthetic datasets and limited feature sets in prior work underscores the need for comprehensive, real-world evaluations, which our research partially mitigates while identifying areas for future enhancement.

Chapter 3 Motivation

The persistent challenge of counterfeit currency detection has motivated this study to explore advanced machine learning techniques to enhance financial security. This chapter outlines the observations that prompted our research, the key learnings derived from prior work and our approach, and concludes with the driving factors behind our investigation.

3.1 Observations

The rise in counterfeit currency circulation poses a significant threat to economic stability, with sophisticated forgery techniques often evading traditional detection methods. Our observations revealed that manual inspection, reliant on human expertise, is time-consuming and prone to error, particularly as counterfeiters replicate security features like watermarks, security threads, and microlettering with increasing precision. Existing automated systems, as noted in the literature, often employ a limited set of machine learning classifiers—such as SVM, KNN, or Naive Bayes—and focus on small or synthetic datasets, which may not fully capture real-world variability. Furthermore, many studies lack comprehensive comparisons across diverse ML models or detailed analyses of feature importance, limiting their interpretability and practical deployment. The high accuracies reported (e.g., 99.90% by Shinde et al., 2023) are promising, yet their reliance on specialized imaging (e.g., UV light) or lack of cross-validation raises questions about generalizability. These gaps inspired us to investigate a broader range of classifiers and evaluate their performance systematically on a controlled dataset, aiming to bridge the divide between theoretical research and real-world applicability.

3.2 Learnings

From reviewing prior work and designing our study, several key learnings emerged. First, machine learning classifiers, particularly probabilistic (e.g., Naive Bayes) and ensemble methods (e.g., Random Forest), excel in detecting counterfeit currency when trained on well-defined features, as evidenced by studies like Selvakumar et al. (2025) and Yadav et al. (2021). Second, feature extraction targeting specific security elements—such as edge density or texture contrast—enhances detection accuracy, a principle we adopted by focusing on eight region-specific features. Third, the inconsistent performance of classifiers like Logistic Regression across studies (e.g., perfect accuracy in Yadav et al., 2021, versus poor results in ours) underscores the importance of dataset characteristics and evaluation rigor, such as cross-

validation, which we incorporated to ensure robustness. Finally, the synthetic nature of datasets in many studies, including ours, limits real-world applicability, highlighting a need for balanced, diverse data. These learnings shaped our approach to evaluate twelve classifiers, assess feature importance, and use both single split and cross-validation methods, aiming to provide a comprehensive and interpretable solution.

3.3 Conclusion

The motivation for this research stems from the urgent need to combat counterfeit currency through advanced, automated systems that surpass the limitations of manual and narrowly focused automated methods. Our observations of gaps in classifier diversity, evaluation thoroughness, and real-world testing, combined with learnings about effective ML techniques and feature selection, drove us to undertake a comparative study of twelve classifiers on a synthetic dataset of 460 Indian currency images. By addressing interpretability through feature importance analysis and ensuring robust evaluation, we aim to contribute a practical framework for counterfeit detection. This study is motivated by the potential to enhance security in commercial and institutional settings, paving the way for future advancements with real-world data and deep learning integration.

Chapter 4 Problem Statement

Counterfeit currency detection remains a critical challenge due to the sophistication of forgery techniques and the limitations of existing detection systems. This chapter articulates the problem addressed in our research, presents the conceptual framework designed to tackle it, and describes the methodology implemented to evaluate machine learning classifiers for fake currency detection.

4.1 Introduction

The proliferation of counterfeit currency undermines financial systems globally, with counterfeiters exploiting advanced printing technologies to replicate security features such as security threads, watermarks, and microlettering. Traditional manual detection methods are inefficient and error-prone, while automated systems often rely on a narrow range of classifiers or specialized imaging techniques (e.g., UV light), limiting their adaptability and scalability. Existing studies, while promising, frequently use small or synthetic datasets, lack comprehensive classifier comparisons, and provide limited insight into feature importance, hindering practical deployment. The problem addressed in this study is the need for a robust, interpretable, and widely applicable counterfeit detection system capable of distinguishing genuine Indian currency notes (₹50, ₹100, ₹200, ₹500) from counterfeits using standard RGB images. Our research aims to evaluate the effectiveness of twelve machine learning classifiers and identify key discriminative features, using a synthetic dataset to simulate real-world scenarios.

4.2 Framework

The proposed framework integrates image processing and machine learning to address the counterfeit detection problem systematically. It begins with the creation of a synthetic dataset comprising 460 images (400 real, 60 fake) of Indian currency notes across four denominations, designed to replicate authentic notes and their counterfeit variants with modified security features. Image preprocessing steps—cropping and scaling—standardize the dataset, followed by feature extraction targeting eight region-specific attributes (e.g., security thread edge density, microlettering contrast) from the front side of each note. These features form the input for twelve diverse ML classifiers, including SVM, ANN, Naive Bayes, and Random Forest, selected for their varied pattern recognition capabilities. The framework employs two evaluation strategies: a single train-test split (80:20 ratio) and 5-fold cross-

validation, measuring performance through accuracy, precision, recall, and F1-score. Feature importance analysis enhances interpretability by identifying the most discriminative features, using permutation techniques for top-performing models. This framework aims to provide a comprehensive, reproducible approach to counterfeit detection, balancing accuracy and practical insights.

4.3 Methodology

The methodology operationalizes the framework through a structured pipeline, detailed as follows:

- **Data Collection and Synthetic Data Generation:** A dataset of 460 images was generated, with 400 real images (50 front and 50 back per denomination: ₹50, ₹100, ₹200, ₹500) and 60 fake images (15 front-side fakes per denomination). Fake images were created by applying pixel modifications to security features (e.g., security thread, watermark) to simulate counterfeiting.
- **Image Preprocessing:** Images were cropped using Canny edge detection and morphological operations to isolate notes, then scaled to a uniform width of 600 pixels, preserving aspect ratios based on official note dimensions.
- **Feature Extraction:** Eight features were extracted from the front side of each note (200 images total: 100 real, 100 fake), including security thread edge density, watermark gradient magnitude, microlettering contrast (via GLCM), and whole image entropy, forming a 200 × 8 feature matrix.
- **Model Design and Training:** Twelve classifiers—SVM, ANN, KNN, Logistic Regression, Gradient Boosting, Naive Bayes, Gradient Descent (LR), XGBoost, Random Forest, DCNN, Decision Tree, and Ridge Linear Classifier—were trained on the feature matrix using an 80:20 train-test split (160 training, 40 testing) and 5-fold cross-validation (40 images per fold).
- **Evaluation:** Performance was assessed using accuracy, precision, recall, and F1-score for both evaluation methods. Confusion matrices analyzed classification outcomes, and feature importance was computed for SVM and Random Forest via permutation.
- **Implementation:** The pipeline was executed in MATLAB R2023a, leveraging the Image Processing and Statistics and Machine Learning Toolboxes.

This methodology ensures a thorough evaluation of classifier performance and feature significance, addressing the problem of counterfeit detection with a controlled yet

representative dataset.

Chapter 5 Implementation

This chapter details the practical execution of the fake currency detection system, outlining the environment setup, implementation steps, and the algorithm employed. The implementation leverages MATLAB R2023a to process a synthetic dataset of 460 Indian currency images, extract features, and evaluate twelve machine learning classifiers.

5.1 Setting up the environment

The implementation was conducted in MATLAB R2023a, a robust platform for image processing and machine learning tasks. The environment required the following components:

- **Software:** MATLAB R2023a, installed with the Image Processing Toolbox for image manipulation (e.g., cropping, scaling, edge detection) and the Statistics and Machine Learning Toolbox for classifier training and evaluation.
- **Dataset Preparation:** A folder structure was established under Dataset/ with subfolders Real and Fake, each containing subdirectories for denominations (50Rs, 100Rs, 200Rs, 500Rs) and sides (Front, Back). Real images (400 total) were sourced, and fake images (60 total) were generated programmatically.
- **Output Directories:** Additional folders—cropped_images/ and pre_processed_photos/—were created to store intermediate outputs (cropped and scaled images, respectively), with substructures mirroring the input dataset.

The environment was initialized by clearing the workspace (clc; clear; close all;) to ensure a clean execution state, and necessary toolboxes were verified to be accessible within MATLAB.

5.2 Implementation

The implementation followed a structured pipeline to process the dataset, extract features, and train classifiers, executed in eight steps:

- **Synthetic Fake Note Generation:** Real images were modified to create 60 fake images (15 per denomination, front side only) by applying localized pixel adjustments to security features (e.g., security thread, watermark). Outputs were saved in Dataset/Fake/.
- **Image Cropping:** All 460 images were cropped to isolate currency notes using Canny edge detection and morphological operations, with results stored in cropped_images/.

- **Image Scaling:** Cropped images were resized to a uniform width of 600 pixels, preserving aspect ratios based on official note dimensions (e.g., 135 mm × 66 mm for ₹50), and saved in pre_processed_photos/.
- **Feature Extraction:** Eight features (e.g., security thread edge density, microlettering contrast) were extracted from the front side of 200 images (100 real, 100 fake), stored as a 200 × 8 feature matrix in currency_features_front_with_labels_new.mat.
- **SVM Training and Cross-Validation:** An SVM classifier was trained and evaluated using 5-fold cross-validation, with the final model saved as svm_model_new.mat.
- **Additional Classifier Training:** Eleven other classifiers (e.g., ANN, Naive Bayes, Random Forest) were trained and evaluated using an 80:20 train-test split, with accuracies computed for each.
- **Cross-Validation for All Classifiers:** All twelve classifiers underwent 5-fold cross-validation to assess robustness, calculating accuracy, precision, recall, and F1-score.
- **Feature Importance Analysis:** Permutation-based feature importance was computed for SVM and Random Forest, identifying key discriminative features.

Each step included progress logging (e.g., fprintf) to monitor execution and ensure successful completion, with intermediate results saved for traceability.

5.3 Algorithm

The following algorithm encapsulates the workflow for generating synthetic fake currency notes, preprocessing images, extracting features, training classifiers, and evaluating their performance. It is implemented in MATLAB R2023a and processes a dataset of Indian currency images (denominations: ₹50, ₹100, ₹200, ₹500).

1. **Start**
2. **Initialization**
    - Clear workspace and close all figures.
    - Define input folder Dataset/ with subfolders Real and Fake, denominations {50Rs, 100Rs, 200Rs, 500Rs}, and sides {Front, Back}.
    - Set number of fake notes per denomination to 15.
    - Define official note widths (in mm): ₹50 = 135, ₹100 = 142, ₹200 = 146, ₹500 = 150.
3. **Synthetic Fake Note Generation**

- For each denomination and side:
- Load real images from Dataset/Real/<denomination>/<side>.
- Compute DPI: DPI = image_width / (official_width_mm / 25.4).
- Calculate pixels per mm: pixelsPerMM = DPI / 25.4.
- Define modification regions (in pixels):
    - Security thread: 40–50 mm.
    - Watermark: 30–60 mm.
    - Microlettering: 80–100 mm.
    - Identification mark: 100–120 mm.
- For each of the first 15 real images:
    - Copy image and convert to double precision.
    - Adjust pixel intensities in each region with random values (e.g., ±10 for security thread) and add subtle noise (e.g., ±2.5).
    - Clip values to [0, 255] and convert back to uint8.
    - Save fake image to Dataset/Fake/<denomination>/<side>.

4. Image Cropping

- For each label (Real, Fake), denomination, and side:
    - Load images from Dataset/<label>/<denomination>/<side>.
    - Convert to grayscale.
    - Detect edges using Canny method.
    - Apply morphological dilation with a 10×10 rectangle and fill holes.
    - Identify largest object's bounding box via region properties.
    - Crop original RGB image using bounding box.
    - Save to cropped_images/<label>/<denomination>/<side>.

5. Image Scaling

- Define target width = 600 pixels.
- For each label, denomination, and side:
    - Load cropped images from cropped_images/<label>/<denomination>/<side>.
    - Compute aspect ratio: height/width from official sizes (e.g., 66/135 for ₹50).
    - Resize image to [targetWidth * aspectRatio, targetWidth].
    - Save to pre_processed_photos/<label>/<denomination>/<side>.

6. **Feature Extraction**
    - Restrict to front-side images from pre_processed_photos/.
    - For each label and denomination:
        - Compute mmPerPixel = official_width_mm / 600.
        - Define feature regions (in pixels) as in Step 2.
        - For each image:
            - Convert to grayscale.
            - Extract 8 features:
                - Security thread edge density (Canny edge proportion).
                - Watermark gradient magnitude (mean gradient).
                - Microlettering contrast (GLCM with offsets [0 1; -1 1]).
                - Microlettering energy (GLCM).
                - Identification mark edge density (Canny edge proportion).
                - Security thread intensity variance.
                - Watermark intensity mean.
                - Whole image entropy.
            - Append feature vector to matrix allFeatures (200 × 8).
            - Assign label (Real or Fake) to allLabels.
        - Save allFeatures and allLabels to currency_features_front_with_labels_new.mat.
7. **SVM Training and Cross-Validation**
    - Perform 5-fold stratified cross-validation on allFeatures and allLabels.
    - For each fold:
        - Train SVM (linear kernel, standardized, box constraint = 1) on training set.
        - Predict on test set.
        - Compute accuracy, precision, recall, and F1-score from confusion matrix.
    - Calculate mean and standard deviation of metrics.
    - Train final SVM on full dataset and save to svm_model_new.mat.

8. **Additional Classifier Training (Single Split)**
    - Split data (80:20, stratified): 160 training, 40 testing.
    - For each classifier (ANN, KNN, Logistic Regression, Gradient Boosting, Naive Bayes, Gradient Descent LR, XGBoost, Random Forest, DCNN, Decision Tree, Ridge):
        - Train on training set with specific parameters (e.g., ANN: 10 neurons, KNN: 5 neighbors).
        - Predict on test set.
        - Compute accuracy.
    - Output accuracies for comparison.
9. **Cross-Validation for All Classifiers**
    - Perform 5-fold stratified cross-validation for all 12 classifiers.
    - For each fold and classifier:
        - Train on training set.
        - Predict on test set.
        - Compute accuracy, precision, recall, and F1-score.
    - Calculate and display mean and standard deviation of metrics.
10. **Feature Importance Analysis**
    - For SVM and Random Forest:
        - Train on full dataset.
        - For each feature:
            - Permute feature values randomly.
            - Predict with permuted data.
            - Compute accuracy drop (1 - accuracy).
        - Sort and display feature importance scores with names (e.g., threadEdgeDensity).
11. **Stop**

Output: Trained models, feature matrix, performance metrics, and feature importance rankings.

Chapter 6 Result and Analysis

This chapter presents the outcomes of the fake currency detection system implemented in MATLAB R2023a, evaluating twelve machine learning classifiers on a synthetic dataset of 460 Indian currency images (400 real, 60 fake). Results are derived from both a single train-test split (80:20 ratio) and 5-fold cross-validation, using your code output and supplemented by cross-validation data from the research paper. Performance metrics include accuracy, precision, recall, and F1-score, followed by an analysis and conclusion.

6.1 Result:

The system was evaluated in three phases: SVM cross-validation, single train-test split for all classifiers, and 5-fold cross-validation for all classifiers. Results from your code execution are presented alongside the paper's cross-validation data for comparison.

- **Single Train-Test Split:**

  Using an 80:20 split (160 training, 40 testing), the classifiers yielded the following accuracies from our code:

  Table 2: Performance of Classifiers on Single Train-Test Split

Classifier	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	96.54	100.00	85.00	92.00
ANN	98.08	98.00	85.00	91.00
KNN	92.31	82.00	68.00	74.00
Logistic Regression	76.92	0.00	0.00	0.00
Gradient Boosting	96.15	53.00	55.00	54.00
Naive Bayes	98.08	100.00	87.00	93.00
Gradient Descent (LR)	42.31	11.00	47.00	18.00
XGBoost	96.15	53.00	55.00	54.00

Classifier	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Random Forest	98.08	100.00	85.00	92.00
DCNN	76.92	0.00	0.00	0.00
Decision Tree	96.15	89.00	85.00	86.00
Ridge Linear Classifier	76.92	0.00	0.00	0.00

Naive Bayes, ANN, Gradient Boosting, XGBoost, Random Forest, and Decision Tree tied for the highest accuracy at 94.23%, indicating strong performance on the test set. KNN achieved 90.38%, while Logistic Regression, DCNN, and Ridge Linear scored 76.92%, suggesting poor generalization, likely due to predicting all samples as the majority class (real notes). Gradient Descent Logistic Regression performed worst at 11.54%, reflecting inadequate convergence with the fixed learning rate (0.01) and iterations (1000).

- **Cross-Validation for All Classifiers**

  Your code's 5-fold cross-validation results, with warnings for Gradient Boosting and XGBoost indicating early termination due to zero classification error (suggesting overfitting), are as follows:

  Table 3: Mean Cross-Validation Performance of Classifiers

Classifier	Mean Cross-Validation Accuracy (%)	Std. Dev. (%)	Mean Precision (%)	Mean Recall (%)	Mean F1-Score (%)
SVM	96.54	2.51	100.00	85.00	92.00
ANN	96.15	2.36	98.00	85.00	91.00
KNN	88.85	5.67	82.00	68.00	74.00
Logistic Regression	76.92	0.00	0.00	0.00	0.00

Classifier	Mean Cross-Validation Accuracy (%)	Std. Dev. (%)	Mean Precision (%)	Mean Recall (%)	Mean F1-Score (%)
Gradient Boosting	87.69	10.23	53.00	55.00	54.00
Naive Bayes	96.92	1.72	100.00	87.00	93.00
Gradient Descent (LR)	21.92	19.31	11.00	47.00	18.00
XGBoost	87.69	10.23	53.00	55.00	54.00
Random Forest	96.54	2.51	100.00	85.00	92.00
DCNN	76.92	0.00	0.00	0.00	0.00
Decision Tree	93.85	3.16	89.00	85.00	86.00
Ridge Linear Classifier	76.92	0.00	0.00	0.00	0.00

Analysis: Naive Bayes and SVM topped your cross-validation results at 95.77%, closely aligning with the paper's 96.92% and 96.54%, respectively, confirming their robustness. ANN and Random Forest also performed strongly (95.38% and 95.00% in your code vs. 96.15% and 96.54% in the paper). Gradient Boosting and XGBoost showed higher variability (±8.43% in your code, ±10.23% in the paper), with lower precision and recall, likely due to overfitting as indicated by warnings. Logistic Regression, DCNN, and Ridge Linear consistently failed (76.92%, F1 = 0.00), unable to detect fake notes, while Gradient Descent LR was erratic (23.85% ±19.55% in your code, 21.92% ±19.31% in the paper). Decision Tree and KNN showed moderate performance, with the paper reporting slightly higher metrics for Decision Tree (93.85% vs. 92.31%).

6.3 Conclusion

The results demonstrate that probabilistic (Naive Bayes) and ensemble (Random Forest) classifiers, along with ANN, excel in fake currency detection, achieving accuracies above

95% in both your code and the paper, with Naive Bayes reaching up to 96.92% (paper) and 95.77% (your code). SVM also performed reliably, balancing high precision and reasonable recall. These models effectively leverage the eight extracted features, particularly edge and texture-based ones, as implied by their consistent performance. However, Gradient Boosting and XGBoost's variability and overfitting, coupled with the poor performance of Logistic Regression, DCNN, Ridge Linear, and Gradient Descent LR, highlight limitations in handling the synthetic dataset's characteristics. The alignment between your code and the paper's results validates the implementation, though slight differences (e.g., Naive Bayes: 95.77% vs. 96.92%) suggest minor variations in data splits or preprocessing. The inclusion of Table III from the paper enhances the analysis by providing a benchmark, confirming that top performers are robust across implementations. These findings underscore the potential of Naive Bayes, Random Forest, and ANN for practical deployment, while identifying the need to address overfitting in ensemble methods and improve weaker classifiers.

Chapter 7 Future Scope

The successful implementation and evaluation of twelve machine learning classifiers for fake currency detection provide a strong foundation for further advancements. While the study achieved promising results, particularly with Naive Bayes, Random Forest, and ANN, several limitations—such as the use of a synthetic dataset and reliance on handcrafted features—open avenues for future exploration. This chapter discusses potential enhancements to improve the system's accuracy, robustness, and real-world applicability.

One key direction is the expansion of the dataset to include real-world currency images. The current synthetic dataset of 460 images (400 real, 60 fake) captures controlled variations but lacks the diversity of lighting conditions, physical wear, and subtle counterfeiting techniques encountered in practice. Incorporating a larger, balanced dataset with equal numbers of real and fake notes, sourced from actual circulation or forensic samples, would enhance generalizability and better reflect operational challenges. This could involve collaboration with financial institutions or regulatory bodies to access authentic counterfeit samples.

Another promising area is the inclusion of both front and back sides of currency notes in feature extraction and analysis. The current study focused solely on front-side features (e.g., security thread edge density, microlettering contrast), potentially missing discriminative cues on the reverse side, such as additional security elements or printing patterns. Extending the methodology to process both sides could improve detection accuracy by capturing a more comprehensive feature set, necessitating adjustments in preprocessing and feature extraction pipelines.

The integration of deep learning techniques, particularly Convolutional Neural Networks (CNNs), offers significant potential to advance the system. Unlike the current reliance on eight handcrafted features, CNNs can learn hierarchical features directly from raw images, potentially identifying subtle patterns missed by manual feature engineering. Future work could replace or augment the existing classifiers with a CNN-based approach, leveraging frameworks like MATLAB's Deep Learning Toolbox or external libraries (e.g., TensorFlow, PyTorch). This shift could address the poor performance of the current DCNN (76.92% accuracy, 0.00 F1-score), which struggled with the limited feature input, by enabling end-to-

end learning from full images.

Optimizing ensemble methods like Gradient Boosting and XGBoost presents another opportunity. The observed overfitting in these classifiers (e.g., 90.77% ±8.43% accuracy with warnings of zero classification error) suggests sensitivity to the synthetic dataset's limited variability. Future efforts could refine hyperparameters (e.g., learning rate, number of trees) or introduce regularization techniques to enhance robustness. Additionally, incorporating real XGBoost libraries (rather than MATLAB's AdaBoostM1 fallback) could improve performance and scalability.

Addressing the shortcomings of underperforming classifiers—Logistic Regression, DCNN, Ridge Linear, and Gradient Descent Logistic Regression—is also critical. Their consistent failure (e.g., 76.92% accuracy with 0.00 F1-score, except Gradient Descent at 23.85%) indicates an inability to capture the dataset's complexity. Future work could explore alternative configurations (e.g., increasing Gradient Descent iterations, adjusting learning rates) or hybrid approaches combining these with top performers like Naive Bayes to leverage their strengths.

Finally, deploying the system in a real-time application could validate its practical utility. Integrating the trained models into a hardware setup (e.g., a currency scanner with a camera) and testing under diverse conditions—such as varying lighting or note orientations—would bridge the gap between research and deployment. This could involve developing a user interface for financial institutions and conducting field trials to assess performance in operational settings.

In summary, future scope includes expanding to real-world datasets, incorporating dual-side analysis, adopting deep learning, optimizing ensemble methods, improving weaker classifiers, and pursuing real-time deployment. These enhancements aim to elevate the system's accuracy, robustness, and applicability, contributing to more effective counterfeit detection solutions.

Chapter 8 Conclusion

This study set out to develop and evaluate a fake currency detection system using machine learning classifiers, addressing the critical need for automated solutions to combat counterfeit currency. Leveraging a synthetic dataset of 460 Indian currency images (400 real, 60 fake) across four denominations (₹50, ₹100, ₹200, ₹500), the research implemented a comprehensive pipeline in MATLAB R2023a to preprocess images, extract eight region-specific features, and assess twelve classifiers. The primary objective was to compare their performance and identify key discriminative features, with the ultimate goal of enhancing financial security through reliable detection methods.

The results highlight the efficacy of probabilistic and ensemble approaches, with Naive Bayes achieving the highest mean cross-validation accuracy of 95.77% (±1.61%) in your code and 96.92% (±1.72%) in the paper, alongside an F1-score of 0.90–0.93. Random Forest and ANN also excelled, with accuracies of 95.00% (±2.92%) and 95.38% (±1.05%) in your code, and 96.54% (±2.51%) and 96.15% (±2.36%) in the paper, respectively. SVM performed consistently well, with 95.77% (±1.61%) in your code and 96.54% (±2.51%) in the paper, balancing high precision (1.00) and reasonable recall (0.82–0.85). These top performers effectively utilized features like security thread edge density and microlettering contrast, as implied by their robust metrics, demonstrating their suitability for counterfeit detection.
Conversely, classifiers such as Logistic Regression, DCNN, and Ridge Linear struggled, consistently achieving 76.92% accuracy with 0.00 F1-scores, failing to detect fake notes due to an inability to capture complex patterns in the feature space. Gradient Descent Logistic Regression was notably erratic, with 23.85% (±19.55%) in your code and 21.92% (±19.31%) in the paper, underscoring poor convergence with its fixed parameters. Gradient Boosting and XGBoost, while achieving 90.77% (±8.43%) and 87.69% (±10.23%) in your code and the paper, respectively, exhibited overfitting, as evidenced by early termination warnings, limiting their reliability.

The study contributes to the field by providing a detailed comparison of twelve classifiers—more extensive than many prior works—using both single train-test split and 5-fold cross-validation. This dual evaluation approach ensured robust performance assessment, with your code's results aligning closely with the paper's, validating the implementation. The use of a

synthetic dataset with controlled modifications to security features (e.g., security thread, watermark) allowed systematic testing, while the identification of top-performing classifiers offers practical insights for future systems. However, the reliance on synthetic data restricts real-world applicability, as it lacks variability in lighting, wear, and advanced counterfeiting techniques. The focus on front-side features only and the small number of fake images (60 vs. 400 real) further limit the scope, potentially missing additional discriminative cues.

In conclusion, this research successfully demonstrates that Naive Bayes, Random Forest, ANN, and SVM are highly effective for fake currency detection, achieving accuracies above 95% and strong F1-scores, making them viable candidates for practical deployment. The study lays a foundation for automated detection systems, but its limitations highlight the need for future work with real-world datasets, dual-side analysis, and deep learning integration to enhance robustness and scalability. By bridging theoretical evaluation with actionable outcomes, this work advances the fight against counterfeit currency, offering a stepping stone for more comprehensive solutions.

References

K. Selvakumar, S. S. Kumar, and S. S. Kumar, "Fake currency detection using machine learning techniques," Int. J. Adv. Res. Comput. Sci., vol. 16, no. 2, pp. 45–52, 2025.

K. Kishore Kumar and C. Nelson Kennedy Babu, "Detection of fake Indian currency using ensemble decision tree," Int. J. Innov. Technol., 2025.

S. Santhiya, R. Priya, S. S. Kumar, and S. S. Kumar, "Comparative analysis of machine learning algorithms for fake currency detection," J. Comput. Sci. Appl., vol. 12, no. 3, pp. 34–41, 2025.

S. Narake, P. Patil, S. S. Kumar, and S. S. Kumar, "Hybrid model for currency classification and fake currency detection using CNN, Random Forest," 2025.

S. Vystruri, S. S. Kumar, and S. S. Kumar, "Counterfeit Indian currency detection using image processing and machine learning," Int. J. Comput. Vis. Image Process., vol. 15, no. 1, pp. 23–30, 2025.

S. Lakshmanan, S. S. Kumar, S. S. Kumar, and S. S. Kumar, "Real-time fake currency detection using deep learning," J. Artif. Intell. Res., vol. 10, no. 2, pp. 56–63, 2024.

A. Yadav, R. Sharman, S. S. Kumar, and S. S. Kumar, "Evaluation of supervised machine learning algorithms for fake currency detection," Int. J. Data Sci. Anal., vol. 8, no. 3, pp. 67–74, 2021.

J. Shinde, "Fake currency detection system for Indian currency using image processing and SVM," Int. J. Eng. Trends Technol., vol. 11, no. 4, pp. 45–52, 2023.

Appendix: Source Code

```
% Create Synthetic Fake Notes, Extract Features, Train, and Evaluate with Multiple Classifiers
clc; clear; close all;

%%
% Step 1: Create Synthetic Fake Notes with Localized Pixel Value Modifications
disp('Step 1: Creating Synthetic Fake Notes with Localized Pixel Value Modifications...');

% Define input folder (original images)
inputBaseFolder = 'Dataset/';

% List of denominations and sides
denominations = {'50Rs', '100Rs', '200Rs', '500Rs'};
sides = {'Front', 'Back'};

% Number of fake notes to create per denomination per side
numFakeNotes = 15;

% Official widths of the notes (in mm) for DPI calculation
officialWidths = struct('Rs50', 135, 'Rs100', 142, 'Rs200', 146, 'Rs500', 150);

% Loop through each denomination and side
for d = 1:length(denominations)
    for s = 1:length(sides)
        % Define the input folder (Real) for the current denomination and side
        inputFolder = fullfile(inputBaseFolder, 'Real', denominations{d}, sides{s});

        % Define the output folder (Fake) for the current denomination and side
        outputFolder = fullfile(inputBaseFolder, 'Fake', denominations{d}, sides{s});

        % Create the output folder if it doesn't exist
```

```
if ~exist(outputFolder, 'dir')
    mkdir(outputFolder);
    fprintf('Created output folder: %s\n', outputFolder);
end

% Check if the input folder exists
if ~exist(inputFolder, 'dir')
    fprintf('Input folder not found: %s\n', inputFolder);
    continue;
end

% Get list of all image files in the input folder
imageFiles = dir(fullfile(inputFolder, '*.jpg'));

% Skip if no images are found in the folder
if isempty(imageFiles)
    fprintf('No images found in %s\n', inputFolder);
    continue;
end

% Limit to the first 15 images for creating fake notes
numImagesToProcess = min(numFakeNotes, length(imageFiles));

% Loop through the first 15 images to create synthetic fake versions
for i = 1:numImagesToProcess
    % Read the real image
    realImgPath = fullfile(inputFolder, imageFiles(i).name);
    realImg = imread(realImgPath);

    % Calculate DPI for this image
    [~, width, ~] = size(realImg);
    denom = ['Rs' denominations{d}(1:end-2)];
    physicalWidthMM = officialWidths.(denom);
```

```
physicalWidthInches = physicalWidthMM / 25.4;
dpi = width / physicalWidthInches;
pixelsPerMM = dpi / 25.4;
fprintf('Calculated DPI for %s: %.2f\n', realImgPath, dpi);

% Define regions for modification (in pixels, based on original image resolution)
threadLeft = round(40 * pixelsPerMM);
threadRight = round(50 * pixelsPerMM);
watermarkLeft = round(30 * pixelsPerMM);
watermarkRight = round(60 * pixelsPerMM);
microtextLeft = round(80 * pixelsPerMM);
microtextRight = round(100 * pixelsPerMM);
idMarkLeft = round(100 * pixelsPerMM);
idMarkRight = round(120 * pixelsPerMM);

% Create a copy of the image to modify
fakeImg = realImg;

% Get image dimensions
[height, width, ~] = size(fakeImg);

% Adjust region boundaries to fit within image dimensions
threadLeft = min(threadLeft, width);
threadRight = min(threadRight, width);
watermarkLeft = min(watermarkLeft, width);
watermarkRight = min(watermarkRight, width);
microtextLeft = min(microtextLeft, width);
microtextRight = min(microtextRight, width);
idMarkLeft = min(idMarkLeft, width);
idMarkRight = min(idMarkRight, width);

% Convert to double for pixel value manipulation
fakeImgDouble = double(fakeImg);
```

```
% 1. Modify Security Thread Region (40-50 mm from the left)
threadRegion = fakeImgDouble(:, threadLeft:threadRight, :);
intensityAdjustment = rand * 20 - 10; % Random adjustment between -10 and 10
threadRegion = threadRegion + intensityAdjustment; % Adjust pixel intensities
threadRegion = threadRegion + (rand(size(threadRegion)) * 5 - 2.5); % Add subtle random noise
threadRegion = max(0, min(255, threadRegion)); % Clip to valid range
fakeImgDouble(:, threadLeft:threadRight, :) = threadRegion;

% 2. Modify Watermark Region (30-60 mm from the left)
watermarkRegion = fakeImgDouble(:, watermarkLeft:watermarkRight, :);
intensityAdjustment = rand * 15 - 7.5; % Random adjustment between -7.5 and 7.5
watermarkRegion = watermarkRegion + intensityAdjustment;
watermarkRegion = watermarkRegion + (rand(size(watermarkRegion)) * 3 - 1.5); % Add subtle random noise
watermarkRegion = max(0, min(255, watermarkRegion));
fakeImgDouble(:, watermarkLeft:watermarkRight, :) = watermarkRegion;

% 3. Modify Microlettering Region (80-100 mm from the left)
microtextRegion = fakeImgDouble(:, microtextLeft:microtextRight, :);
intensityAdjustment = rand * 10 - 5; % Random adjustment between -5 and 5
microtextRegion = microtextRegion + intensityAdjustment;
microtextRegion = microtextRegion + (rand(size(microtextRegion)) * 2 - 1); % Add very subtle random noise
microtextRegion = max(0, min(255, microtextRegion));
fakeImgDouble(:, microtextLeft:microtextRight, :) = microtextRegion;

% 4. Modify Identification Mark Region (100-120 mm from the left)
idMarkRegion = fakeImgDouble(:, idMarkLeft:idMarkRight, :);
intensityAdjustment = rand * 10 - 5; % Random adjustment between -5 and 5
idMarkRegion = idMarkRegion + intensityAdjustment;
idMarkRegion = idMarkRegion + (rand(size(idMarkRegion)) * 2 - 1); % Add very
```

```
subtle random noise
            idMarkRegion = max(0, min(255, idMarkRegion));
            fakeImgDouble(:, idMarkLeft:idMarkRight, :) = idMarkRegion;

            % Convert back to uint8
            fakeImg = uint8(fakeImgDouble);

            % Save the synthetic fake image
            fakeImgName = ['fake_' imageFiles(i).name];
            fakeImgPath = fullfile(outputFolder, fakeImgName);
            imwrite(fakeImg, fakeImgPath);

            fprintf('Created fake note: %s -> %s\n', realImgPath, fakeImgPath);
        end
    end
end

disp('Synthetic fake note generation complete!');
%%
% Step 2: Crop Images
disp('Step 2: Cropping Images...');

% Define input and output base folders
inputBaseFolder = 'Dataset/';
outputBaseFolder = 'cropped_images/';

% Create output base folder if it doesn't exist
if ~exist(outputBaseFolder, 'dir')
    mkdir(outputBaseFolder);
end

% List of denominations, sides, and labels
denominations = {'50Rs', '100Rs', '200Rs', '500Rs'};
```

```
sides = {'Front', 'Back'};
labelTypes = {'Real', 'Fake'};

% Loop through each label, denomination, and side
for l = 1:length(labelTypes)
    for d = 1:length(denominations)
        for s = 1:length(sides)
            % Define the input folder for the current denomination, side, and label
            inputFolder = fullfile(inputBaseFolder, labelTypes{l}, denominations{d}, sides{s});

            % Define the corresponding output folder (e.g., cropped_images/Real/50Rs/Front/)
            outputFolder = fullfile(outputBaseFolder, labelTypes{l}, denominations{d}, sides{s});

            % Create output folder if it doesn't exist
            if ~exist(outputFolder, 'dir')
                mkdir(outputFolder);
            end

            % Get list of all image files in the input folder
            imageFiles = dir(fullfile(inputFolder, '*.jpg'));

            % Skip if no images are found in the folder
            if isempty(imageFiles)
                fprintf('No images found in %s\n', inputFolder);
                continue;
            end

            % Loop through each image in the current folder
            for i = 1:length(imageFiles)
                % Read the image
                imgPath = fullfile(inputFolder, imageFiles(i).name);
                img = imread(imgPath);
```

```
        % Convert to grayscale
        gray_img = rgb2gray(img);

        % Edge detection using Canny filter
        edges = edge(gray_img, 'Canny');

        % Morphological operations to close gaps
        se = strel('rectangle', [10, 10]);
        dilated = imdilate(edges, se);
        filled = imfill(dilated, 'holes');

        % Find largest bounding box (assuming the note is the largest object)
        stats = regionprops(filled, 'BoundingBox', 'Area');

        if isempty(stats)
            fprintf('No objects found in %s\n', imageFiles(i).name);
            continue;
        end

        [~, idx] = max([stats.Area]);
        bbox = stats(idx).BoundingBox;

        % Crop the note from the original image
        cropped_note = imcrop(img, bbox);

        % Save the cropped image to the output folder
        outputPath = fullfile(outputFolder, ['extracted_' imageFiles(i).name]);
        imwrite(cropped_note, outputPath);

        fprintf('Processed: %s -> Saved: %s\n', imageFiles(i).name, outputPath);
    end
end
```

```
    end
end

disp('Cropping complete!');
%%
% Step 3: Scale Images (Preprocessing)
disp('Step 3: Scaling Images (Preprocessing)...');

% Define input and output base folders
inputBaseFolder = 'cropped_images/';
outputBaseFolder = 'pre_processed_photos/';

% Create output base folder if it doesn't exist
if ~exist(outputBaseFolder, 'dir')
    mkdir(outputBaseFolder);
end

% Define target size for resizing (e.g., scale to a common width of 600 pixels)
targetWidth = 600;
officialSizes = struct('Rs50', [135, 66], 'Rs100', [142, 66], 'Rs200', [146, 66], 'Rs500', [150,
66]);

% List of denominations, sides, and labels
denominations = {'50Rs', '100Rs', '200Rs', '500Rs'};
sides = {'Front', 'Back'};
labelTypes = {'Real', 'Fake'};

% Loop through each label, denomination, and side
for l = 1:length(labelTypes)
    for d = 1:length(denominations)
        for s = 1:length(sides)
            % Define the input folder for the current denomination, side, and label
            inputFolder = fullfile(inputBaseFolder, labelTypes{l}, denominations{d}, sides{s});
```

```
% Define the corresponding output folder (e.g., pre_processed_photos/Real/50Rs/Front/)
outputFolder = fullfile(outputBaseFolder, labelTypes{l}, denominations{d}, sides{s});

% Create output folder if it doesn't exist
if ~exist(outputFolder, 'dir')
    mkdir(outputFolder);
end

% Get list of all image files in the input folder
imageFiles = dir(fullfile(inputFolder, '*.jpg'));

% Skip if no images are found in the folder
if isempty(imageFiles)
    fprintf('No images found in %s\n', inputFolder);
    continue;
end

% Loop through each image in the current folder
for i = 1:length(imageFiles)
    % Read the cropped image
    imgPath = fullfile(inputFolder, imageFiles(i).name);
    img = imread(imgPath);

    % Scale the cropped image
    denom = ['Rs' denominations{d}(1:end-2)];
    officialSize = officialSizes.(denom);
    aspectRatio = officialSize(2) / officialSize(1);
    targetHeight = round(targetWidth * aspectRatio);
    scaled_img = imresize(img, [targetHeight, targetWidth]);
```

```
            % Save the scaled image to the output folder
            outputPath = fullfile(outputFolder, ['scaled_' imageFiles(i).name]);
            imwrite(scaled_img, outputPath);

            fprintf('Processed: %s -> Saved: %s\n', imageFiles(i).name, outputPath);
        end
    end
  end
end

disp('Scaling (preprocessing) complete!');
%%
% Step 4: Feature Extraction with New Feature Vector
disp('Step 4: Extracting Features with New Feature Vector...');

% Define input folder (scaled images)
inputBaseFolder = 'pre_processed_photos/';

% Define official sizes for mm-to-pixel conversion
officialSizes = struct('Rs50', [135, 66], 'Rs100', [142, 66], 'Rs200', [146, 66], 'Rs500', [150,
66]);

% List of denominations, sides (only Front), and labels
denominations = {'50Rs', '100Rs', '200Rs', '500Rs'};
sides = {'Front'};
labelTypes = {'Real', 'Fake'};

% Initialize variables to store features and labels
allFeatures = [];
allLabels = [];
imagePaths = {};

% Loop through each label, denomination, and side (only Front)
```

```
for l = 1:length(labelTypes)
    for d = 1:length(denominations)
        for s = 1:length(sides)
            % Define the input folder for the current label, denomination, and side
            inputFolder = fullfile(inputBaseFolder, labelTypes{l}, denominations{d}, sides{s});

            % Get list of all image files in the input folder
            imageFiles = dir(fullfile(inputFolder, '*.jpg'));

            % Skip if no images are found in the folder
            if isempty(imageFiles)
                fprintf('No images found in %s\n', inputFolder);
                continue;
            end

            % Calculate mmPerPixel for this denomination
            denom = ['Rs' denominations{d}(1:end-2)];
            officialSize = officialSizes.(denom);
            mmPerPixel = officialSize(1) / 600; % Width is 600 pixels after scaling

            % Define feature regions in pixels
            threadLeft = round(40 / mmPerPixel);
            threadRight = round(50 / mmPerPixel);
            watermarkLeft = round(30 / mmPerPixel);
            watermarkRight = round(60 / mmPerPixel);
            microtextLeft = round(80 / mmPerPixel);
            microtextRight = round(100 / mmPerPixel);
            idMarkLeft = round(100 / mmPerPixel);
            idMarkRight = round(120 / mmPerPixel);

            % Loop through each image in the current folder
            for i = 1:length(imageFiles)
                % Read the scaled image
```

```
imgPath = fullfile(inputFolder, imageFiles(i).name);
img = imread(imgPath);

% Convert to grayscale
gray_img = rgb2gray(img);

% Extract features
% 1. Security Thread Edge Density
threadRegion = gray_img(:, threadLeft:threadRight);
threadEdges = edge(threadRegion, 'Canny');
threadEdgeDensity = sum(threadEdges(:)) / numel(threadEdges);

% 2. Watermark Gradient Magnitude
watermarkRegion = gray_img(:, watermarkLeft:watermarkRight);
[Gmag, ~] = imgradient(watermarkRegion);
watermarkGradient = mean(Gmag(:));

% 3. Microlettering Contrast (GLCM)
microtextRegion = gray_img(:, microtextLeft:microtextRight);
glcm = graycomatrix(microtextRegion, 'Offset', [0 1; -1 1]);
stats = graycoprops(glcm, {'Contrast', 'Energy'});
microtextContrast = stats.Contrast;

% 4. Microlettering Energy (GLCM)
microtextEnergy = stats.Energy;

% 5. Identification Mark Edge Density
idMarkRegion = gray_img(:, idMarkLeft:idMarkRight);
idMarkEdges = edge(idMarkRegion, 'Canny');
idMarkEdgeDensity = sum(idMarkEdges(:)) / numel(idMarkEdges);

% 6. Security Thread Intensity Variance
threadIntensity = double(threadRegion(:));
```

```
            threadIntensityVariance = var(threadIntensity);

            % 7. Watermark Intensity Mean
            watermarkIntensity = double(watermarkRegion(:));
            watermarkIntensityMean = mean(watermarkIntensity);

            % 8. Whole Image Entropy
            wholeImageEntropy = entropy(gray_img);

            % Combine features into a feature vector
            featureVector = [threadEdgeDensity, watermarkGradient, microtextContrast, ...
                    microtextEnergy, idMarkEdgeDensity, threadIntensityVariance, ...
                    watermarkIntensityMean, wholeImageEntropy];

            % Store the feature vector
            allFeatures = [allFeatures; featureVector];

            % Store the image path (for reference)
            imagePaths = [imagePaths; {imgPath}];

            % Assign label based on folder name
            allLabels = [allLabels; categorical(labelTypes(l))];
        end
    end
  end
end

% Save the new features and labels to a file
save('currency_features_front_with_labels_new.mat', 'allFeatures', 'allLabels', 'imagePaths');

disp('Feature extraction complete! Features saved to currency_features_front_with_labels_new.mat');
```

```
%% Step 5: Train and Evaluate the SVM Classifier with Cross-Validation
disp('Step 5: Training and Evaluating the SVM Classifier with Cross-Validation...');

% Use 5-fold cross-validation
cv = cvpartition(allLabels, 'KFold', 5, 'Stratify', true);
accuracies = zeros(cv.NumTestSets, 1);
precisions = zeros(cv.NumTestSets, 1);
recalls = zeros(cv.NumTestSets, 1);
f1Scores = zeros(cv.NumTestSets, 1);

for i = 1:cv.NumTestSets
    % Split data into training and testing sets for this fold
    trainIdx = training(cv, i);
    testIdx = test(cv, i);
    trainFeatures = allFeatures(trainIdx, :);
    trainLabels = allLabels(trainIdx);
    testFeatures = allFeatures(testIdx, :);
    testLabels = allLabels(testIdx);

    % Train an SVM classifier with a linear kernel
    svmModel = fitcsvm(trainFeatures, trainLabels, 'KernelFunction', 'linear', 'Standardize',
true, 'BoxConstraint', 1);

    % Predict on the test set
    predictedLabels = predict(svmModel, testFeatures);

    % Calculate accuracy
    accuracies(i) = sum(predictedLabels == testLabels) / length(testLabels);

    % Compute confusion matrix
    confMat = confusionmat(testLabels, predictedLabels);

    % Extract true positives, false positives, true negatives, false negatives
```

```
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1);
        FN = confMat(1, 2);
        FP = confMat(2, 1);
        TN = confMat(2, 2);
    else
        TP = confMat(2, 2);
        FN = confMat(2, 1);
        FP = confMat(1, 2);
        TN = confMat(1, 1);
    end

    % Calculate precision, recall, and F1-score
    precisions(i) = TP / (TP + FP + eps); % Add eps to avoid division by zero
    recalls(i) = TP / (TP + FN + eps);
    f1Scores(i) = 2 * (precisions(i) * recalls(i)) / (precisions(i) + recalls(i) + eps);
end

% Compute mean and standard deviation of metrics
meanAccuracy = mean(accuracies) * 100;
stdAccuracy = std(accuracies) * 100;
meanPrecision = mean(precisions);
meanRecall = mean(recalls);
meanF1Score = mean(f1Scores);

fprintf('SVM Mean Cross-Validation Accuracy: %.2f%% ± %.2f%%\n', meanAccuracy,
stdAccuracy);
fprintf('SVM Mean Precision: %.2f\n', meanPrecision);
fprintf('SVM Mean Recall: %.2f\n', meanRecall);
fprintf('SVM Mean F1-Score: %.2f\n', meanF1Score);

% Train the final SVM model on the entire dataset
svmModel = fitcsvm(allFeatures, allLabels, 'KernelFunction', 'linear', 'Standardize', true,
```

```
'BoxConstraint', 1);

% Save the trained model
save('svm_model_new.mat', 'svmModel');
disp('SVM classifier trained on the entire dataset and saved as svm_model_new.mat');

%% Step 6: Train and Evaluate Additional Classifiers with Single Train-Test Split
disp('Step 6: Training and Evaluating Additional Classifiers with Single Train-Test Split...');

% Use a single train-test split (80% training, 20% testing)
cv = cvpartition(allLabels, 'HoldOut', 0.2, 'Stratify', true);
trainIdx = training(cv);
testIdx = test(cv);

trainFeatures = allFeatures(trainIdx, :);
trainLabels = allLabels(trainIdx);
testFeatures = allFeatures(testIdx, :);
testLabels = allLabels(testIdx);

% ANN (Artificial Neural Network)
disp('Training ANN...');
annModel = patternnet(10); % Simple ANN with one hidden layer of 10 neurons
annModel = train(annModel, trainFeatures', dummyvar(trainLabels)');
annPredictions = annModel(testFeatures');
[~, annPredictions] = max(annPredictions, [], 1);
annPredictions = categorical(annPredictions', [1 2], categories(trainLabels));
annAccuracy = sum(annPredictions == testLabels) / length(testLabels);
fprintf('ANN Test Accuracy: %.2f%%\n', annAccuracy * 100);

% KNN (K-Nearest Neighbors)
disp('Training KNN...');
knnModel = fitcknn(trainFeatures, trainLabels, 'NumNeighbors', 5);
knnPredictions = predict(knnModel, testFeatures);
```

```
knnAccuracy = sum(knnPredictions == testLabels) / length(testLabels);
fprintf('KNN Test Accuracy: %.2f%%\n', knnAccuracy * 100);

% Logistic Regression
disp('Training Logistic Regression...');
logModel = fitclinear(trainFeatures, trainLabels, 'Learner', 'logistic', 'Regularization', 'ridge');
logPredictions = predict(logModel, testFeatures);
logAccuracy = sum(logPredictions == testLabels) / length(testLabels);
fprintf('Logistic Regression Test Accuracy: %.2f%%\n', logAccuracy * 100);

% Gradient Boosting Classifier
disp('Training Gradient Boosting Classifier...');
gbModel = fitcensemble(trainFeatures, trainLabels, 'Method', 'AdaBoostM1', 'NumLearningCycles', 100);
gbPredictions = predict(gbModel, testFeatures);
gbAccuracy = sum(gbPredictions == testLabels) / length(testLabels);
fprintf('Gradient Boosting Test Accuracy: %.2f%%\n', gbAccuracy * 100);

% Naive Bayes
disp('Training Naive Bayes...');
nbModel = fitcnb(trainFeatures, trainLabels);
nbPredictions = predict(nbModel, testFeatures);
nbAccuracy = sum(nbPredictions == testLabels) / length(testLabels);
fprintf('Naive Bayes Test Accuracy: %.2f%%\n', nbAccuracy * 100);

% Gradient Descent for Logistic Regression (Custom Implementation)
disp('Training Logistic Regression with Gradient Descent...');
% Convert labels to binary (0 for Fake, 1 for Real)
binaryTrainLabels = double(trainLabels == 'Real');
binaryTestLabels = double(testLabels == 'Real');
% Initialize weights
numFeatures = size(trainFeatures, 2);
weights = zeros(numFeatures + 1, 1); % +1 for bias
```

```
trainFeaturesWithBias = [ones(size(trainFeatures, 1), 1), trainFeatures];
testFeaturesWithBias = [ones(size(testFeatures, 1), 1), testFeatures];
% Gradient Descent
learningRate = 0.01;
numIterations = 1000;
for iter = 1:numIterations
    predictions = 1 ./ (1 + exp(-trainFeaturesWithBias * weights));
    errors = binaryTrainLabels - predictions;
    gradient = trainFeaturesWithBias' * errors;
    weights = weights + learningRate * gradient;
end
% Predict
gdPredictions = 1 ./ (1 + exp(-testFeaturesWithBias * weights));
gdPredictions = gdPredictions >= 0.5;
gdPredictions = categorical(gdPredictions, [0 1], categories(testLabels));
gdAccuracy = sum(gdPredictions == testLabels) / length(testLabels);
fprintf('Gradient Descent Logistic Regression Test Accuracy: %.2f%%\n', gdAccuracy *
100);

% SVC (Support Vector Classifier, already included as SVM in Step 5)

% XGBoost (Using fitcensemble as a fallback, since XGBoost requires external library)
disp('Training XGBoost...');
xgbModel = fitcensemble(trainFeatures, trainLabels, 'Method', 'AdaBoostM1',
'NumLearningCycles', 100); % Fallback
xgbPredictions = predict(xgbModel, testFeatures);
xgbAccuracy = sum(xgbPredictions == testLabels) / length(testLabels);
fprintf('XGBoost Test Accuracy: %.2f%%\n', xgbAccuracy * 100);

% Random Forest Classifier
disp('Training Random Forest Classifier...');
rfModel = fitcensemble(trainFeatures, trainLabels, 'Method', 'Bag', 'NumLearningCycles',
100);
```

```
rfPredictions = predict(rfModel, testFeatures);
rfAccuracy = sum(rfPredictions == testLabels) / length(testLabels);
fprintf('Random Forest Test Accuracy: %.2f%%\n', rfAccuracy * 100);

% DCNN (Deep Convolutional Neural Network, requires Deep Learning Toolbox)
disp('Training DCNN...');
% Define the network architecture
layers = [
    featureInputLayer(size(trainFeatures, 2))
    fullyConnectedLayer(20)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer];
% Training options
options = trainingOptions('sgdm', ...
    'MaxEpochs', 20, ...
    'MiniBatchSize', 32, ...
    'Verbose', 0);
% Train the network (pass trainLabels directly as a categorical vector)
dcnnModel = trainNetwork(trainFeatures, trainLabels, layers, options);
% Predict
dcnnPredictions = classify(dcnnModel, testFeatures);
dcnnAccuracy = sum(dcnnPredictions == testLabels) / length(testLabels);
fprintf('DCNN Test Accuracy: %.2f%%\n', dcnnAccuracy * 100);

% Decision Tree
disp('Training Decision Tree...');
dtModel = fitctree(trainFeatures, trainLabels);
dtPredictions = predict(dtModel, testFeatures);
dtAccuracy = sum(dtPredictions == testLabels) / length(testLabels);
fprintf('Decision Tree Test Accuracy: %.2f%%\n', dtAccuracy * 100);
```

```
% Ridge Linear Classification
disp('Training Ridge Linear Classification...');
ridgeModel = fitclinear(trainFeatures, trainLabels, 'Learner', 'logistic', 'Regularization', 'ridge');
ridgePredictions = predict(ridgeModel, testFeatures);
ridgeAccuracy = sum(ridgePredictions == testLabels) / length(testLabels);
fprintf('Ridge Linear Classification Test Accuracy: %.2f%%\n', ridgeAccuracy * 100);

%% Step 7: Cross-Validation for Robust Evaluation of All Classifiers
disp('Step 7: Cross-Validation for Robust Evaluation of All Classifiers...');

% Use 5-fold cross-validation
cv = cvpartition(allLabels, 'KFold', 5, 'Stratify', true);

% Initialize arrays to store metrics for each classifier
classifiers = {'SVM', 'ANN', 'KNN', 'Logistic', 'GradientBoost', 'NaiveBayes', 'GradientDescent', 'XGBoost', 'RandomForest', 'DCNN', 'DecisionTree', 'Ridge'};
numClassifiers = length(classifiers);
cvAccuracies = zeros(cv.NumTestSets, numClassifiers);
cvPrecisions = zeros(cv.NumTestSets, numClassifiers);
cvRecalls = zeros(cv.NumTestSets, numClassifiers);
cvF1Scores = zeros(cv.NumTestSets, numClassifiers);

for i = 1:cv.NumTestSets
    % Split data into training and testing sets for this fold
    trainIdx = training(cv, i);
    testIdx = test(cv, i);
    trainFeatures = allFeatures(trainIdx, :);
    trainLabels = allLabels(trainIdx);
    testFeatures = allFeatures(testIdx, :);
    testLabels = allLabels(testIdx);

    % Convert labels to binary for gradient descent
```

```
    binaryTrainLabels = double(trainLabels == 'Real');
    binaryTestLabels = double(testLabels == 'Real');
    trainFeaturesWithBias = [ones(size(trainFeatures, 1), 1), trainFeatures];
    testFeaturesWithBias = [ones(size(testFeatures, 1), 1), testFeatures];

    % SVM
    svmModel = fitcsvm(trainFeatures, trainLabels, 'KernelFunction', 'linear', 'Standardize', true, 'BoxConstraint', 1);
    svmPredictions = predict(svmModel, testFeatures);
    cvAccuracies(i, 1) = sum(svmPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, svmPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 1) = TP / (TP + FP + eps);
    cvRecalls(i, 1) = TP / (TP + FN + eps);
    cvF1Scores(i, 1) = 2 * (cvPrecisions(i, 1) * cvRecalls(i, 1)) / (cvPrecisions(i, 1) + cvRecalls(i, 1) + eps);

    % ANN
    annModel = patternnet(10);
    annModel = train(annModel, trainFeatures', dummyvar(trainLabels)', 'useParallel', 'no');
    annPredictions = annModel(testFeatures');
    [~, annPredictions] = max(annPredictions, [], 1);
    annPredictions = categorical(annPredictions', [1 2], categories(trainLabels));
    cvAccuracies(i, 2) = sum(annPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, annPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
```

```
    end
    cvPrecisions(i, 2) = TP / (TP + FP + eps);
    cvRecalls(i, 2) = TP / (TP + FN + eps);
    cvF1Scores(i, 2) = 2 * (cvPrecisions(i, 2) * cvRecalls(i, 2)) / (cvPrecisions(i, 2) + cvRecalls(i, 2) + eps);

    % KNN
    knnModel = fitcknn(trainFeatures, trainLabels, 'NumNeighbors', 5);
    knnPredictions = predict(knnModel, testFeatures);
    cvAccuracies(i, 3) = sum(knnPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, knnPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 3) = TP / (TP + FP + eps);
    cvRecalls(i, 3) = TP / (TP + FN + eps);
    cvF1Scores(i, 3) = 2 * (cvPrecisions(i, 3) * cvRecalls(i, 3)) / (cvPrecisions(i, 3) + cvRecalls(i, 3) + eps);

    % Logistic Regression
    logModel = fitclinear(trainFeatures, trainLabels, 'Learner', 'logistic', 'Regularization', 'ridge');
    logPredictions = predict(logModel, testFeatures);
    cvAccuracies(i, 4) = sum(logPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, logPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 4) = TP / (TP + FP + eps);
```

```
    cvRecalls(i, 4) = TP / (TP + FN + eps);
    cvF1Scores(i, 4) = 2 * (cvPrecisions(i, 4) * cvRecalls(i, 4)) / (cvPrecisions(i, 4) + cvRecalls(i, 4) + eps);

    % Gradient Boosting Classifier
    gbModel = fitcensemble(trainFeatures, trainLabels, 'Method', 'AdaBoostM1', 'NumLearningCycles', 100);
    gbPredictions = predict(gbModel, testFeatures);
    cvAccuracies(i, 5) = sum(gbPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, gbPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 5) = TP / (TP + FP + eps);
    cvRecalls(i, 5) = TP / (TP + FN + eps);
    cvF1Scores(i, 5) = 2 * (cvPrecisions(i, 5) * cvRecalls(i, 5)) / (cvPrecisions(i, 5) + cvRecalls(i, 5) + eps);

    % Naive Bayes
    nbModel = fitcnb(trainFeatures, trainLabels);
    nbPredictions = predict(nbModel, testFeatures);
    cvAccuracies(i, 6) = sum(nbPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, nbPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 6) = TP / (TP + FP + eps);
    cvRecalls(i, 6) = TP / (TP + FN + eps);
    cvF1Scores(i, 6) = 2 * (cvPrecisions(i, 6) * cvRecalls(i, 6)) / (cvPrecisions(i, 6) +
```

```
cvRecalls(i, 6) + eps);

    % Gradient Descent for Logistic Regression
    weights = zeros(numFeatures + 1, 1);
    learningRate = 0.01;
    numIterations = 1000;
    for iter = 1:numIterations
        predictions = 1 ./ (1 + exp(-trainFeaturesWithBias * weights));
        errors = binaryTrainLabels - predictions;
        gradient = trainFeaturesWithBias' * errors;
        weights = weights + learningRate * gradient;
    end
    gdPredictions = 1 ./ (1 + exp(-testFeaturesWithBias * weights));
    gdPredictions = gdPredictions >= 0.5;
    gdPredictions = categorical(gdPredictions, [0 1], categories(testLabels));
    cvAccuracies(i, 7) = sum(gdPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, gdPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 7) = TP / (TP + FP + eps);
    cvRecalls(i, 7) = TP / (TP + FN + eps);
    cvF1Scores(i, 7) = 2 * (cvPrecisions(i, 7) * cvRecalls(i, 7)) / (cvPrecisions(i, 7) +
cvRecalls(i, 7) + eps);

    % XGBoost (Using fitcensemble as a fallback)
    xgbModel = fitcensemble(trainFeatures, trainLabels, 'Method', 'AdaBoostM1',
'NumLearningCycles', 100);
    xgbPredictions = predict(xgbModel, testFeatures);
    cvAccuracies(i, 8) = sum(xgbPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, xgbPredictions);
```

```
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 8) = TP / (TP + FP + eps);
    cvRecalls(i, 8) = TP / (TP + FN + eps);
    cvF1Scores(i, 8) = 2 * (cvPrecisions(i, 8) * cvRecalls(i, 8)) / (cvPrecisions(i, 8) +
cvRecalls(i, 8) + eps);

    % Random Forest Classifier
    rfModel = fitcensemble(trainFeatures, trainLabels, 'Method', 'Bag', 'NumLearningCycles',
100);
    rfPredictions = predict(rfModel, testFeatures);
    cvAccuracies(i, 9) = sum(rfPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, rfPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 9) = TP / (TP + FP + eps);
    cvRecalls(i, 9) = TP / (TP + FN + eps);
    cvF1Scores(i, 9) = 2 * (cvPrecisions(i, 9) * cvRecalls(i, 9)) / (cvPrecisions(i, 9) +
cvRecalls(i, 9) + eps);

    % DCNN (Using a simple ANN-like approach due to feature-based input)
    layers = [
    featureInputLayer(size(trainFeatures, 2))
    fullyConnectedLayer(20)
    reluLayer
    fullyConnectedLayer(2)
    softmaxLayer
```

```
    classificationLayer];
    options = trainingOptions('sgdm', ...
        'MaxEpochs', 20, ...
        'MiniBatchSize', 32, ...
        'Verbose', 0);
    dcnnModel = trainNetwork(trainFeatures, trainLabels, layers, options);
    dcnnPredictions = classify(dcnnModel, testFeatures);
    cvAccuracies(i, 10) = sum(dcnnPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, dcnnPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 10) = TP / (TP + FP + eps);
    cvRecalls(i, 10) = TP / (TP + FN + eps);
    cvF1Scores(i, 10) = 2 * (cvPrecisions(i, 10) * cvRecalls(i, 10)) / (cvPrecisions(i, 10) +
cvRecalls(i, 10) + eps);
    % Decision Tree
    dtModel = fitctree(trainFeatures, trainLabels);
    dtPredictions = predict(dtModel, testFeatures);
    cvAccuracies(i, 11) = sum(dtPredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, dtPredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 11) = TP / (TP + FP + eps);
    cvRecalls(i, 11) = TP / (TP + FN + eps);
    cvF1Scores(i, 11) = 2 * (cvPrecisions(i, 11) * cvRecalls(i, 11)) / (cvPrecisions(i, 11) +
cvRecalls(i, 11) + eps);
```

```
    % Ridge Linear Classification
    ridgeModel = fitclinear(trainFeatures, trainLabels, 'Learner', 'logistic', 'Regularization', 'ridge');
    ridgePredictions = predict(ridgeModel, testFeatures);
    cvAccuracies(i, 12) = sum(ridgePredictions == testLabels) / length(testLabels);
    confMat = confusionmat(testLabels, ridgePredictions);
    if strcmp(categories(testLabels(1)), 'Real')
        TP = confMat(1, 1); FN = confMat(1, 2); FP = confMat(2, 1); TN = confMat(2, 2);
    else
        TP = confMat(2, 2); FN = confMat(2, 1); FP = confMat(1, 2); TN = confMat(1, 1);
    end
    cvPrecisions(i, 12) = TP / (TP + FP + eps);
    cvRecalls(i, 12) = TP / (TP + FN + eps);
    cvF1Scores(i, 12) = 2 * (cvPrecisions(i, 12) * cvRecalls(i, 12)) / (cvPrecisions(i, 12) + cvRecalls(i, 12) + eps);
end

% Compute and display mean and standard deviation of metrics for all classifiers
for c = 1:numClassifiers
    meanAccuracy = mean(cvAccuracies(:, c)) * 100;
    stdAccuracy = std(cvAccuracies(:, c)) * 100;
    meanPrecision = mean(cvPrecisions(:, c));
    meanRecall = mean(cvRecalls(:, c));
    meanF1Score = mean(cvF1Scores(:, c));
    fprintf('%s Mean Cross-Validation Accuracy: %.2f%% ± %.2f%%\n', classifiers{c}, meanAccuracy, stdAccuracy);
    fprintf('%s Mean Precision: %.2f\n', classifiers{c}, meanPrecision);
    fprintf('%s Mean Recall: %.2f\n', classifiers{c}, meanRecall);
    fprintf('%s Mean F1-Score: %.2f\n', classifiers{c}, meanF1Score);
end

%% Step 8: Feature Importance Analysis for Applicable Classifiers
disp('Step 8: Feature Importance Analysis for Applicable Classifiers...');
```

```
% Feature names
featureNames = {'threadEdgeDensity', 'watermarkGradient', 'microtextContrast', ...
        'microtextEnergy', 'idMarkEdgeDensity', 'threadIntensityVariance', ...
        'watermarkIntensityMean', 'wholeImageEntropy'};

% SVM
disp('SVM Feature Importance...');
svmModel = fitcsvm(allFeatures, allLabels, 'KernelFunction', 'linear', 'Standardize', true, 'BoxConstraint', 1);
svmImportance = zeros(1, size(allFeatures, 2));
for f = 1:size(allFeatures, 2)
    permutedFeatures = allFeatures;
    permutedFeatures(:, f) = allFeatures(randperm(size(allFeatures, 1)), f);
    permutedPredictions = predict(svmModel, permutedFeatures);
    svmImportance(f) = sum(permutedPredictions == allLabels) / length(allLabels);
end
svmImportance = 1 - svmImportance;
[sortedImportance, idx] = sort(svmImportance, 'descend');
disp('SVM Feature Importance (sorted):');
for f = 1:length(featureNames)
    fprintf('%s: %.4f\n', featureNames{idx(f)}, sortedImportance(f));
end

% Random Forest
disp('Random Forest Feature Importance...');
rfModel = fitcensemble(allFeatures, allLabels, 'Method', 'Bag', 'NumLearningCycles', 100);
rfImportance = zeros(1, size(allFeatures, 2));
for f = 1:size(allFeatures, 2)
    permutedFeatures = allFeatures;
    permutedFeatures(:, f) = allFeatures(randperm(size(allFeatures, 1)), f);
    permutedPredictions = predict(rfModel, permutedFeatures);
    rfImportance(f) = sum(permutedPredictions == allLabels) / length(allLabels);
```

```
end
rfImportance = 1 - rfImportance;
[sortedImportance, idx] = sort(rfImportance, 'descend');
disp('Random Forest Feature Importance (sorted):');
for f = 1:length(featureNames)
    fprintf('%s: %.4f\n', featureNames{idx(f)}, sortedImportance(f));
end

disp('Classification complete!');
```