Assessment Task 2 – Image Processing and Pattern Recognition
Project Implementation

Project Group 38
Jun Hyun Lim
Raymond Hsu
Hani Motassam
Thisuni Rupatunge
William Tran
Benjamin White

Table of Contents

Executive Summary 3
Introduction to the Problem 3
  Project Overview and Motivation 3
  Recap of Two-Stage Design (Hazard Filter ® Fire vs Smoke) 4
    Stage 1, Hazard Filter: 4
    Stage 2, Hazard Classification (Fire vs Smoke): 4
  Implementation Environment 4
Overview of the project and methodology 5
  System Architecture and Workflow 5
  Dataset Description and Licensing 5
  Data Curation and Pre-processing 6
  Feature Extraction Methods 6
    Purpose: 6
    Implementation details: 6
    Dimensionality, scaling, reuse: 7
  Classification Techniques 7
    Stage 1 – Hazard Filter (FIRE/SMOKE vs NON-FIRE) 8
    Stage 2 – Hazard Classification (FIRE vs SMOKE) 8
  Evaluation Plan and Performance Metrics 9
    Design and protocol: 9
    Why PR-based metrics for Stage 1: 9
    Stage-wise metrics reported by the notebook: 10
    End-to-end (three-class) evaluation. 10
    Qualitative diagnostics. 10
Project Management and Teamwork 10
  Contribution Table 11
Results 12
  Stage 1 12
  Stage 2 13
Deployment 16
Discussion and Challenges 16
Conclusion 17
Future Recommendations 18
Appendix 21
  Link to Google Colab: 21
  Code: 21

Executive Summary

This report documents the production of an automated image-based bushfire early detection system. Based on image classification, the target system is intended to identify bushfire hazards from onsite image capture devices to tackle one of Australia's most serious threats to public health and safety.

Having expounded on the benefits of a two-stage architecture for our target system previously, we undertook to develop a deployable model consistent with our proposal. Using a public domain image dataset sourced from Kaggle consisting of a broad 42,900 samples organised suitably for a supervised machine learning process, a sequence of sample selection, curation, pre-processing and feature extraction was engaged in the leadup to model learning.

Developed with Python in Google Colab sourcing from a dataset hosted on a shared Google Drive, we managed a purposeful software development flow resulting in a working two-stage classification model. Generated classifier performance evaluation metrics indicate solid performance on both stages and overall, with noted room for improvement in dealing with Non-Hazard scenarios.

Acknowledged challenges raised throughout centre around deficient recall on Non-Hazard cases and computational resource limits, to be addressed for future progressions upon this project. Additional recommendations advise potential re-implementation directions to be pursued.

Introduction to the Problem

Project Overview and Motivation

Bushfires are one of the biggest natural hazards impacting Australia. They pose serious threats to human life, wildlife, and infrastructure. Their increasing frequency and severity, fueled by climate change and extended drought conditions, have heightened the need for quick and dependable early-warning systems. Traditional detection methods, like satellite monitoring and watchtower observation, face issues with delays, cloud cover, and human error. To overcome these problems, this project introduces an automated two-stage image classification system that detects and categorises bushfire hazards directly from visual data.

This system aims to improve public safety by allowing faster situational awareness and reducing false negatives that can lead to disastrous outcomes. By using computer vision and pattern recognition techniques, the proposed model shows how traditional image-processing methods can provide lightweight, understandable, and repeatable alternatives to deep learning models, especially in situations with limited resources or where deployment is on the edge.

This project also supports ongoing efforts for climate resilience and environmental monitoring. It follows global trends in AI-driven disaster management, where explainability and reliability are just as important as accuracy. The prototype developed can serve as a basis for future enhancements, like thermal imaging or drone-based surveillance for early fire detection.

Recap of Two-Stage Design (Hazard Filter ® Fire vs Smoke)

The proposed architecture uses a two-stage detection pipeline aimed at improving operational efficiency and safety.

Stage 1, Hazard Filter:

- A binary classifier identifies hazardous images (fire vs smoke) from non-hazardous ones. The goal at this stage is to maximise recall. This is to ensure that no real fire or smoke event is missed, even if it results in some minor false positives.

Stage 2, Hazard Classification (Fire vs Smoke):

- After a potential hazard is identified, a second classifier decides if the image shows visible flames (fire) or diffuse plumes (smoke). This classification helps in emergency decision-making. Fire detections usually need immediate action, while smoke alerts may need verification by patrols for early-stage ignitions.

This staged approach balances cost efficiency and decision accuracy. It reflects the workflow of emergency services, where quick hazard screening comes first, followed by a focused evaluation of severity and response priority.

Additionally, separating detection from classification makes the model modular. Stage 1 can run continuously on live video feeds. In contrast, Stage 2 can be activated as needed to refine results, which cuts down on energy use and reduces false alarms in large-scale operations.

Implementation Environment

The system was implemented in Python using Google Colab to ensure a reproducible, cloud-based development environment. Core libraries and frameworks included:

- NumPy and Pandas for numerical operations and dataset management
- OpenCV, scikit-image for image preprocessing and feature extraction (LBP, HOG, colour statistics including a red-excess index)
- Scikit-learn for machine-learning models (SVM & k-Nearest Neighbours) and performance evaluation
- Matplotlib and Seaborn for visualisation of results and confusion matrices

Google Colab provides a reproducible, collaborative environment with version-controlled experimentation, and collaborative workflow integration among team members. All random seeds, preprocessing pipelines, and dataset partitions were fixed to maintain reproducibility, while strict dataset curation ensured ethical and responsible AI use by, we verified structure and filtered by extension, no manual relabelling or removals were performed.

Overview of the project and methodology

System Architecture and Workflow

The proposed system uses a structured, two-stage image classification pipeline to detect and categorise bushfire-related hazards with high operational efficiency and reliability. The workflow follows a sequential architecture made up of five main modules. Each module supports a reproducible and scalable detection framework.

1. Data acquisition and curation – importing and verifying the public Forest Fire, Smoke, and Non-Fire Image Dataset (Minha, 2023).
2. Pre-processing – normalising and augmenting imagery to ensure consistent lighting, contrast, and size.
3. Feature extraction – computing handcrafted descriptors that capture colour, texture, and gradient information relevant to smoke and fire.
4. Classification – applying lightweight machine-learning models (SVM, optional kNN) to perform Stage 1 (Hazard vs Non-Hazard) and Stage 2 (Fire vs Smoke) decisions.
5. Evaluation and visualisation – assessing accuracy, recall, precision, and F1 through confusion matrices, ROC, and PR curves.

We use a folder scanner that ignores case (e.g., Smoke vs smoke) Hyphenation/underscores must match the directory names. We also use a specific file order to ensure consistent results.

This modular setup allows for transparency, interpretability, and independent improvement of each component. The workflow reflects real-world emergency management logic, starting with a fast, continuously operating hazard-filtering stage. This is followed by a targeted classification stage that sharpens alerts for accuracy in the situation.

By combining efficiency, clarity, and reproducibility, the system offers a solid foundation for future integration into automated early-warning and environmental monitoring platforms.

Dataset Description and Licensing

The dataset employed for the model development is a *Forest fire, smoke and non-fire image dataset* sourced from Kaggle (Minha, 2023). This is a collated collection of images of forest fires, smoke and miscellaneous non-fire and non-smoke scenes, formally allocated to Fire, Smoke and Non-Fire label categories. From a total of 42,900 images, 32,400 are allocated for Training (10,800 per label category) and 10,500 for Testing (3,500 per label category), effectively providing a split of 75.52% for Training and 24.48% for Testing. Regarding licensing, the dataset is listed as CC0: Public Domain on Kaggle, allowing for functionally unrestricted use as far as we are concerned.

For reproducibility within Colab limits, we evaluate a specific set of 10,000 images, selected using a stable SHA-1 key based on file paths. The Train/Val/Test distribution is 6,016, 1,505, and 2,479 images. The Stage-2 (hazard-only) splits are 4,005, 1,002, and 1,628 respectively.

Data Curation and Pre-processing

The dataset is already organised into train and test directories, with subfolders for the three classes. A verification script scans and counts valid image files using common extensions (.jpg, .jpeg, .png, .tif) to ensure data integrity before training the model (Northcutt et al., 2021).

To standardise the visual input, each image is resized to 256 x 256 pixels and converted from BGR to RGB colour space using OpenCV (El-Madafri et al., 2024). Lighting variations are corrected using LAB histogram equalisation. This method adjusts the luminance (L) channel while keeping colour information intact. This step ensures consistent brightness and contrast across scenes captured under different daylight and smoke conditions (El-Madafri et al., 2024).

During training, light augmentation is used to improve generalisation:

- Horizontal flips (probability = 0.5)
- Small rotations within ± 5 degrees (probability = 0.3)
- Random contrast/brightness jitter Î[0.9, 1.1] and beta Î[-10, 10] (probability = 0.3)

These controlled changes simulate natural environmental variability, like wind-driven smoke, camera tilt, and changing sunlight, without altering class meanings (El-Madafri et al., 2024). Augmented samples are only used during training, while the validation and test sets remain unchanged for a fair evaluation.

The resulting dataset is divided into separate *training*, *validation*, and *testing* subsets; from the original training set, 20% of samples are randomly held out for validation using stratified sampling to preserve class balance (as specified in Assessment 1; Minha, 2023).

Feature Extraction Methods

Purpose:

This section describes how we turn images into numerical inputs for the two-stage classifiers. We create a lightweight, clear hybrid descriptor that combines texture (LBP) (Ojala et al., 2002), edge/shape (HOG) (Dalal & Triggs, 2005), and colour (HSV/YCrCb statistics and a red-excess ratio) (Çelik & Demirel, 2009) to capture the main visual features of fire (coherent, high-magnitude gradients, red-yellow colours, high brightness) compared to smoke (diffuse, low-contrast texture, low saturation) (Gragnaniello et al., 2024; Jin et al., 2023). Extracted features are saved to enable reproducibility and lowered compute times across multiple runs. We use the same feature stack in Stage 1 (hazard filter) and Stage 2 (fire vs smoke) for consistency and efficient reuse.

Implementation details:

- Inputs: images standardised to 256x256 (RGB) after LAB equalisation.

- LBP (texture): P=8, R=1, method="uniform". Build a 10-bin (P+2) L1-normalised histogram from the grayscale image.
- HOG (edge/shape): orientations=9, pixels_per_cell=(16, 16), cells_per_block=(2, 2), block_norm="L2-Hys". On 256x256, this yields 16x16 cells and 15x15 blocks, 36 values/block.
- Colour statistics & ratio:
  - HSV per channel: mean, std, median, min, max ® 15 dims.
  - YCrCb: Cr mean/std, Cb mean/std, and (Cr – Cb) mean ® 5 dims.
  - Red-excess $R - \frac{G+B}{2}$ ® 1 dim.
- Concatenation order: [LBP | HOG | Colour]. Features are cached to artifacts/cache/ for reproducible runs.

Dimensionality, scaling, reuse:

- Dimensionality: » 8,131 features per image = LBP 10 + HOG 8,100 + Colour 21.
- Scaling: standardise to zero-mean/unit-variance using StandardScaler fit on training data only, apply the same scaler to validation/test.
- Reuse across stages: the same feature stack feeds Stage 1 (hazard filter) and Stage 2 (fire vs smoke) for consistency and efficiency. Separate scalers/models are kept per stage because Stage 2 operates on hazard-only images.

Optional note on scope/ablation:
We only use LBP, HOG, and colour statistics. We considered GLCM/Haralick features in Assessment 1, but we did not implement them in A2 to keep inference lightweight and to avoid overlap with LBP and HOG.

Classification Techniques

The classification stage of this project was divided into two phases, each designed to balance performance, interpretability, and computational efficiency. In Stage 1, a calibrated Linear Support Vector Machine (SVM) was implemented to distinguish hazardous images (fire or smoke) from non-hazardous ones. The model was trained on combined texture, edge, and colour features extracted through LBP, HOG, and HSV/YCrCb descriptors, using stratified sampling to maintain class balance. Class weighting (class_weight='balanced') and standardisation (StandardScaler) were applied to manage uneven distributions and high-dimensional feature spaces. Probability calibration with a sigmoid function enabled threshold tuning based on Precision–Recall trade-offs, ensuring high recall to minimise false negatives—an essential requirement for early hazard detection. In Stage 2, the same feature set was used to classify fire versus smoke using another Linear SVM and a k-Nearest Neighbours (kNN) model as a benchmark. The SVM achieved better linear separability and faster inference, while kNN provided a non-parametric comparison. Both stages were evaluated using accuracy, precision, recall, F1-score, and ROC/PR-AUC metrics. The consistent use of lightweight, explainable classifiers allowed the system to deliver reliable and interpretable results suitable for real-time bushfire detection while maintaining computational efficiency within Google Colab’s resource constraints.

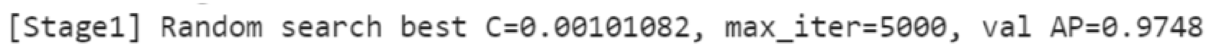
Stage 1 – Hazard Filter (FIRE/SMOKE vs NON-FIRE)

Goal: Quickly screen images for potential hazards and pass only the likely hazards to Stage 2.

Model:
Linear SVM (LinearSVC) trained on the combined feature vector, with:

- class_weight=’balanced’ to offset class proportions,
- Feature scaling via StandardScaler (fit on train only, applied to validation/test),
- Probability calibration using CalibratedClassifierCV(method='sigmoid', cv='prefit') to turn SVM scores into well-behaved probabilities.
- Hyperparameters (C, max_iter) are tuned via random search on validation AP.

Hyperparameter Tuning: Random search was used to fine tune two hyperparameters for the Linear SVM modelling in Stage 1. Due to limited compute resources in Google Colab, the number of search iterations was capped at 25, so the search space and number of tuned parameters were also deliberately constrained. The model was evaluated on the validation split using average precision as the evaluation metric. C and max iterations were tuned by the random search for SVM in Stage 1. The C value controls the strength of regularisation meaning lower values encourage a wider margin and more regularisation, while higher values allow the model to fit the training data more tightly. Whereas the max iterations value determines the maximum number of optimisation steps the solver can take, if set too low, the model may not converge and if set too high, computation runs longer than necessary. The best combination achieved an average precision of 0.9748 on the validation split as seen in Figure 1.



```
[Stage1] Random search best C=0.00101082, max_iter=5000, val AP=0.9748
```

Figure 1. Stage 1 - PR curve (val) with chosen threshold.

Thresholding: From validation probabilities, we examine the Precision-Recall curve and select the first threshold where precision is 0.80 or higher. If this is not possible, we use the best F1 point instead. This process makes the gating clear and reproducible. The chosen threshold is saved to artifacts/reports/stage1_threshold.json.

Why this choice. Linear SVMs are quick and stable in high-dimensional feature spaces, around 8,000 dimensions, and after calibration, they provide the probabilities needed for effective threshold tuning.

Stage 2 – Hazard Classification (FIRE vs SMOKE)

Goal: For images flagged as hazards by Stage 1, distinguish FIRE from SMOKE.

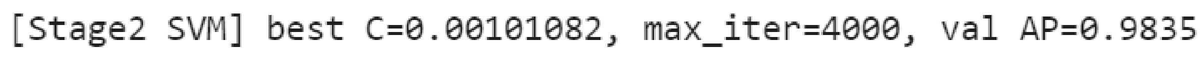
Model(s):

- Primary: Linear SVM (LinearSVC + CalibratedClassifierCV(method='sigmoid', cv='prefit')) on the same feature vector and scaling strategy as Stage 1.

- Benchmark (optional): k-Nearest Neighbours (best n_neighbors and weights selected by validation AP) as a non-linear baseline.

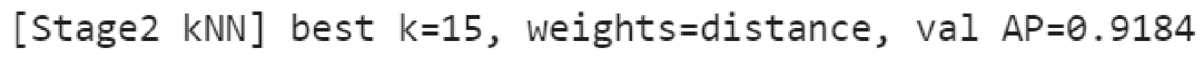
Training data: Only hazard images (FIRE/SMOKE) are used, labels are encoded as 0=FIRE, 1=SMOKE. A separate scaler is fit for Stage 2.

Hyperparameter Tuning: Random search was used to fine tune two hyperparameters for both the Linear SVM and kNN models in Stage 2. Since the number of random search iterations was limited to 25 to reduce computation load, the search space and the number of hyperparameters were also limited to prioritise finding a more optimal combination. Both models were evaluated on the validation split using average precision as the evaluation metric. For the Linear SVM, C, which controls regularisation strength, was sampled from a log uniform range, and the maximum number of iterations, which caps the number of optimiser steps and affects convergence time, was drawn from a small discrete set. For kNN, the number of neighbours and the weighting scheme were tuned from discrete options, where n_neighbors sets the size of the voting set and weights selects equal voting or distance weighted voting. Figures 2 and 3 show the best values found for SVM and kNN in Stage 2.



```
[Stage2 SVM] best C=0.00101082, max_iter=4000, val AP=0.9835
```

Figure 2. Stage 2 (SVM) - ROC & PR curves (test).



```
[Stage2 kNN] best k=15, weights=distance, val AP=0.9184
```

Figure 3. Stage 2 - Confusion matrix (test).

Why this choice: Using the same custom descriptor at different stages makes maintenance easier and keeps inference efficient. The calibrated SVM delivers good linear separation based on gradient, texture, and colour cues, while kNN provides a straightforward non-parametric reference.

Evaluation Plan and Performance Metrics

Design and protocol:

- We maintain the dataset's provided train/test split (Minha, 2023). In the training set, we set aside 20% for validation using stratified sampling. Preprocessing and feature extraction are the same across splits, but augmentation is only for training.
- Scalers are fit on the training set only at each stage and then applied to validation and test sets to prevent leakage.
- To ensure reproducibility under Colab limits, we evaluate a consistent subset of 10,000 images using a stable SHA-1 ordering. All figures and tables are exported to artifacts/reports/. Models are stored in artifacts/models/, and qualitative failure galleries go to artifacts/galleries/.

Why PR-based metrics for Stage 1:

Stage 1 acts as a safety gate (hazard vs non-hazard). Failing to detect a real hazard (false negative) is more critical than a false alarm (false positive). Therefore, we focus on recall while maintaining acceptable precision. Precision-Recall (PR) offers better insights than ROC, especially when positive instances are rare or costs differ (Davis & Goadrich, 2006; Saito & Rehmsmeier, 2015). Therefore:

- Plot the PR curve on validation, compute Average Precision (AP), and
- Choose the operating threshold as the first point with precision at 0.8 or higher if unavailable, we pick the best F1 point. The threshold is saved (stage1_threshold.json) and then frozen before testing.

Stage-wise metrics reported by the notebook:

- Stage 1 (Hazard filter):
    - Primary: PR curve, AP, chosen precision/recall at the operating point.
    - Secondary: ROC-AUC (for comparability) and a confusion matrix + classification report (precision, recall, F1, support).
- Stage 2 (Fire vs Smoke):
    - Primary: Per-class precision/recall/F1 and macro-F1 (class-balancing), plus overall accuracy.
    - Curves: PR (AP) and ROC-AUC computed from calibrated probabilities.
    - Values: Count and normalised confusion matrices, and a per-class bar chart of precision/recall/F1.
    - Rationale: with two hazard classes of roughly similar size in our subset, macro-F1 fairly aggregates class performance, while accuracy is shown but not relied upon alone (Powers, 2020).

End-to-end (three-class) evaluation.

With the Stage-1 threshold fixed, we run the full pipeline and report:

- A three-class confusion matrix ((fire, smoke, and non-fire) and a classification report.
- Interpretation focuses on:
    - Stage-1 FNs (missed hazards) as the highest-cost error,
    - Stage-1 FPs (non-fire routed to Stage 2), acceptable within ops budget,
    - Stage-2 swaps (fire <> smoke), which affect severity/response priority but are less critical than missing hazards.

Qualitative diagnostics.

Failure galleries for each true class help us identify common confusions (for example, fog, steam, and glare mistaken for non-fire negatives). This information is valuable for future data curation and feature improvements.

Project Management and Teamwork

With regards to communications, the group's most productive and regular communication was the weekly in-person workshops on Monday evenings. This regular meeting was used to ensure everyone was on the same page on the project and to clarify any points of interest or ambiguity. The team also made use of

Microsoft Teams both for comprehensive textual communication and to hold ad-hoc group calls to address points of interest or ambiguity as needed and ensure everyone was in the loop on how the code implementation was progressing.

Our team approached the project with a generally collaborative up-for-grabs approach to enable everyone to play their strengths and interests. Rather than strict allocations of roles, our team saw the benefit of being able to iteratively build upon each other's contributions in the report, so various sections became products of iterative development by multiple contributors.

With the code-writing process, appreciating the potential for too many cooks to spoil the broth, we allocated the task to 2 (later 3) of our most Python-familiar members to take the lead. The code was written on a shared Google Colab notebook with the dataset hosted on a shared Google Drive folder, allowing for all members nonetheless to see the progress throughout the code-writing process and provide their own feedback and writing contributions here and there. Those who didn't take the lead on the code would focus their share of efforts mostly on writing the report.

Contribution Table

Name	Student No.	Contribution
Raymond Hsu		Results Deployment
Jun Hyun Lim		Code (lead) Introduction to the Problem Overview of the Project and Methodology Results Deployment Future Recommendations Appendix Document control
Hani Motassam		Code (lead) Classification Techniques
Thisuni Rupatunge		Discussion and Challenges Conclusion
William Tran		Code (contributing) Executive Summary Dataset Description and Licensing Project Management and Teamwork Results Discussion and Challenges Future Recommendations
Benjamin White		Code (lead) Environment Setup Classification Techniques Results Discussion and Challenges Future Recommendations

Table 1. Project Team 38 contribution table.

Results

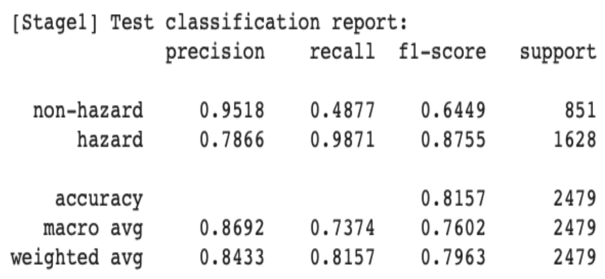
Stage 1

Stage 1 uses an SVM to classify images as either hazard or non-hazard. The hazard class combines two classes from the original dataset, fire and smoke. This means that there is a class imbalance, as there are twice as many hazards as there are non-hazards. Thus, accuracy is not an optimal evaluation metric because it is biased by the class imbalance. While Stage 1 achieved a high accuracy of 81.6%, the precision and recall results are more meaningful due to this imbalance.

This classification is intended to be applied in the early detection of bushfires, so it is better for this system to be more liberal in classifying images as hazards, as this is preferred to missing potential hazards. In Figure 4, the hazard class has a very high recall percentage of 98.7%, meaning very few hazards were missed. Whereas the precision score for hazards was lower at 78.7%, meaning that out of 100 hazards detected, 79 of those would actually be a hazard. The non-hazard class also achieved a high precision at 95.2% but severely underperformed on recall, with the system only detecting 48.8% of all non-hazards. While a system that can better distinguish between hazards and non-hazards is ideal, the performance of the system to accurately detect 98.7% of all hazards is the preferred outcome for the application in bushfire early detection. Future system improvements should prioritise increasing the non-hazard recall score, as this will reduce the amount of human intervention required to correct any non-hazard images that are commonly misclassified as hazards.

The precision-recall curve in Figure 5 shows a stable relationship between precision and recall across all thresholds, with an average precision of 96.8% for the hazard class. During threshold selection, the threshold was fixed on the validation data at a precision greater than 80%. On the test data, precision at the fixed point was 78.7% which aligned with a recall of 98.7% on the PR Curve, reflecting minor sampling shift between validation and test while preserving a very high recall score.

Finally, the ROC curve shown in Figure 6 highlights the strong performance of the SVM model in accurately classifying true hazards, achieving an Area Under the Curve (AUC) of 94.7%. The smooth curve across all true and false positive rates demonstrates consistent model behaviour and strong discriminative ability between hazard and non-hazard images. Overall, the system performs effectively for its intended application, prioritising the early detection of hazards to give fire and rescue services more time to respond and contain fires before they spread.



```
[Stage1] Test classification report:
              precision    recall  f1-score   support

  non-hazard     0.9518    0.4877    0.6449       851
      hazard     0.7866    0.9871    0.8755      1628

    accuracy                         0.8157      2479
   macro avg     0.8692    0.7374    0.7602      2479
weighted avg     0.8433    0.8157    0.7963      2479
```

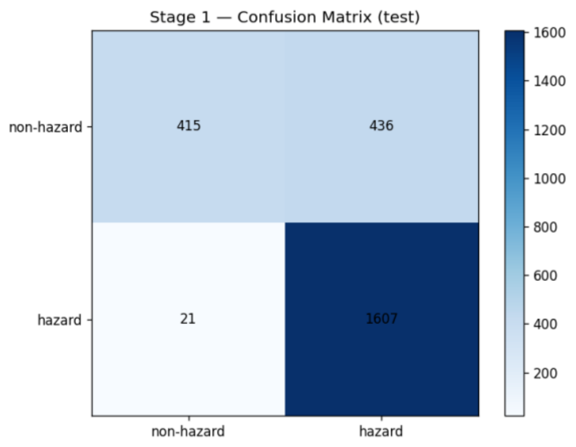
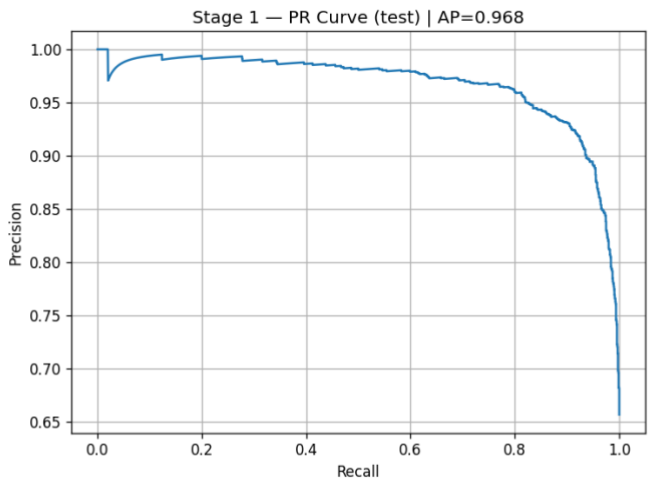


Figure 4. Stage 1 - Classification report & confusion matrix (test).



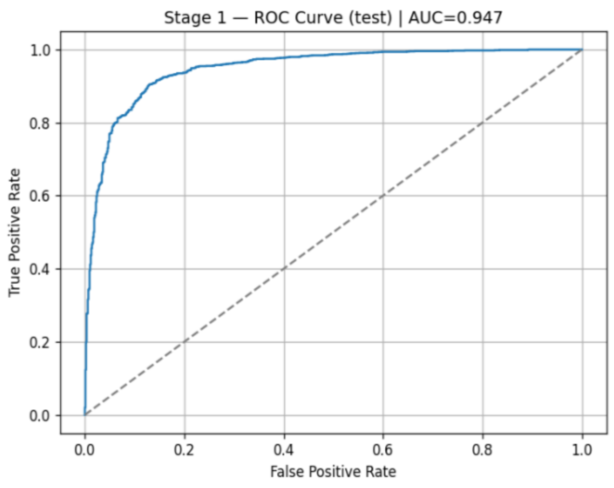


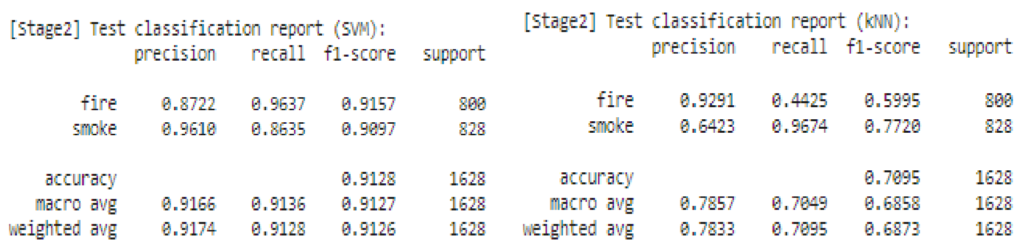
Figure 5. Stage 1 - Precision–Recall curve (test).

Figure 6. Stage 1 - ROC curve (test).

Stage 2

Stage 2 determines the priority of images with known hazards or predicted hazards by further categorising them as either smoke or fire. Two models were trained and evaluated for Stage 2, a SVM classifier and a k-Nearest Neighbours (kNN) classifier. Figure 7 compares the classification reports of both models. The SVM model achieved an accuracy of 91.3%, whereas the kNN model performed worse, with a classification accuracy of 70.1% across the two balanced classes. Moreover, the SVM demonstrated excellent performance across both precision and recall, with all scores exceeding 86% for both classes seen in Figure 9. In contrast, the kNN model again performed less effectively, with a weighted average F1-score of 68.7%. The weaker performance of kNN was especially evident in its recall for the fire class (44%) and precision for the smoke class (64%). This indicates a classifier biased to be stricter when calling Fire and thus, toward calling Smoke. Figure 8 further illustrates this performance gap, showing that kNN misclassified 446 true fire images as smoke, while the SVM only misclassified 29 true fire images. As a result, the SVM model was selected for Stage 2 due to its superior performance across all evaluation metrics from the classification report and confusion matrix.

Furthermore, Figure 10 highlights the robustness of the SVM model, achieving an Area Under the Curve (AUC) score of 98%, indicating a near perfect ROC curve. This demonstrates the model's ability to distinguish effectively between the two classes at all thresholds, maintaining high true positive rates even at low false positive rates. Additionally, Figure 11 also presents a near perfect Precision-Recall curve with an average precision score of 98%, confirming the model's consistent discriminative power across varying recall levels. These results reinforce the reliability of the SVM model to accurately distinguish fire from smoke in Stage 2. Therefore, the evaluation metrics confirm the SVM model's excellent ability to priorities hazards by classifying images as either fire or smoke in Stage 2.



```
[Stage2] Test classification report (SVM):
              precision    recall  f1-score   support

        fire     0.8722    0.9637    0.9157       800
       smoke     0.9610    0.8635    0.9097       828

    accuracy                         0.9128      1628
   macro avg     0.9166    0.9136    0.9127      1628
weighted avg     0.9174    0.9128    0.9126      1628
```

```
[Stage2] Test classification report (kNN):
              precision    recall  f1-score   support

        fire     0.9291    0.4425    0.5995       800
       smoke     0.6423    0.9674    0.7720       828

    accuracy                         0.7095      1628
   macro avg     0.7857    0.7049    0.6858      1628
weighted avg     0.7833    0.7095    0.6873      1628
```

Figure 7. Stage 2- Classification Reports for SVM and kNN modelling on test data.

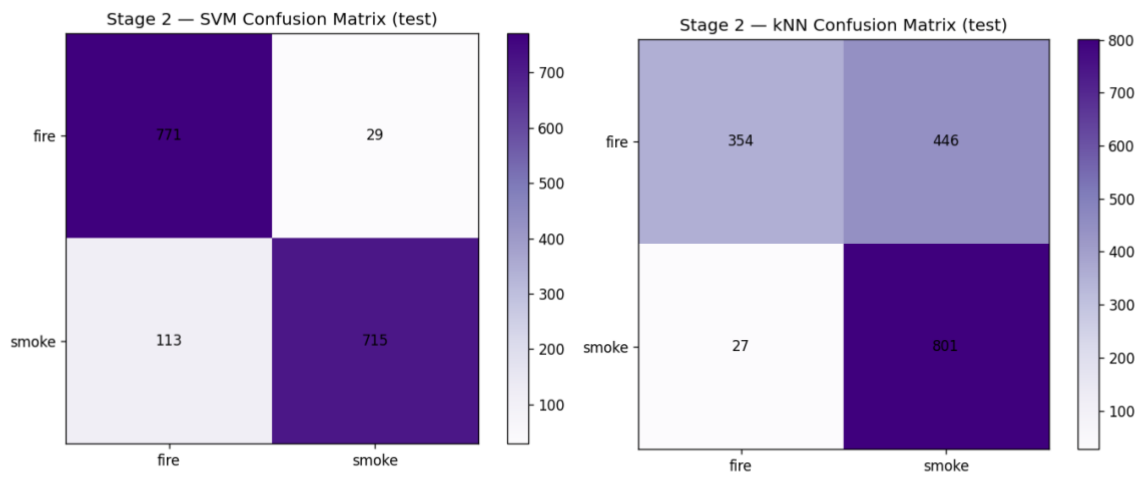


Figure 8. Stage 2- Confusion Matrices for SVM and kNN modelling on test data.

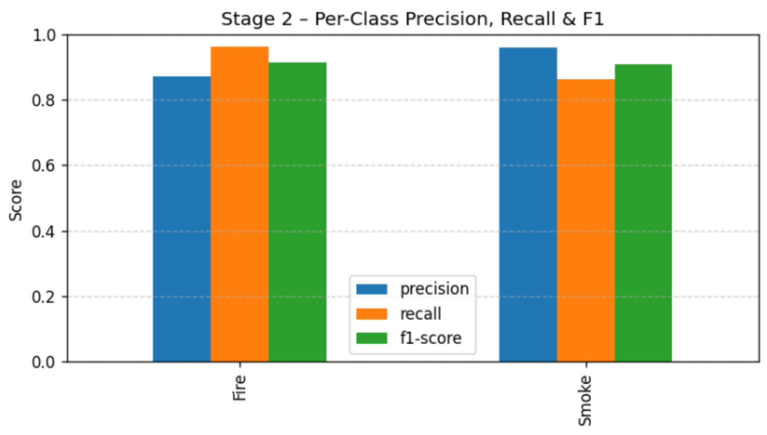
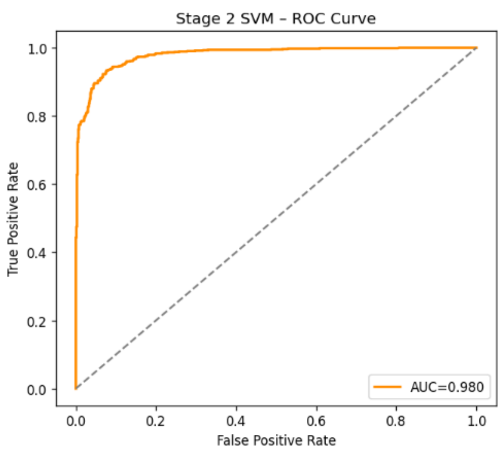


Figure 9. Stage 2- Column Chart Comparing Precision, Recall and F1 for SVM modelling.



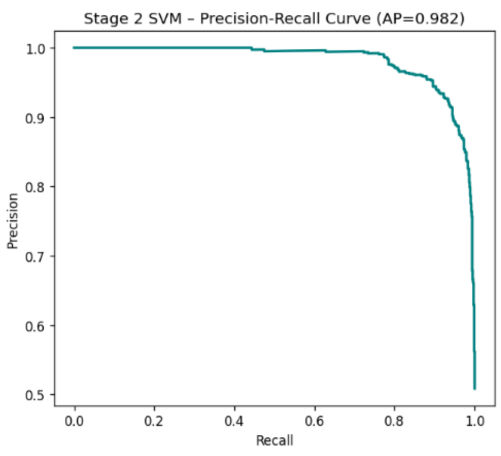


Figure 10. Stage 2- ROC Curve for SVM

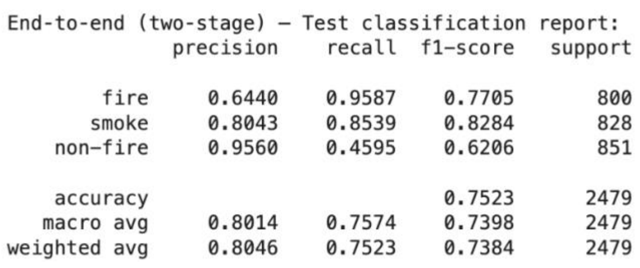
Figure 11. Stage 2- PR Curve for SVM

Two-Stage Implementation

The two-stage pipeline first uses Stage 1's SVM binary classifier to determine whether an image contains a hazard based on a calibrated threshold. If the predicted probability exceeds this threshold, the image is passed to Stage 2, where the tuned SVM then classifies it further as either fire or smoke. Images below the threshold in Stage 1 are classified as non-hazard. This structure enables early filtering of non-hazard images in Stage 1, followed by Stage 2 which classifies the hazard images as either fire or smoke for response priority.

Ultimately, the end-to-end system achieved an overall test accuracy of 75.2% across the balanced three class setup (fire, smoke, non-fire), as seen in Figure 12. When individually assessed, Stage 1 showed very low recall performance for the non-hazard class. In this complete implementation, Figure 13 shows that Stage 2 frequently classifies these true non-hazard images as containing fire. Thus, suggesting that certain visual features characteristic of fire images are also common in non-hazard images and are influencing these Stage 1 misclassifications.

Therefore, the two-stage system exhibits good performance in recognising fire, smoke, and non-fire scenarios with a reasonable accuracy score. Future development of the system should focus on identifying why a large number of non-hazard images are being misclassified as fire images and implementing improvements to increase recall for true non-hazard cases in Stage 1.


```
End-to-end (two-stage) — Test classification report:
              precision    recall  f1-score   support

        fire     0.6440    0.9587    0.7705       800
       smoke     0.8043    0.8539    0.8284       828
    non-fire     0.9560    0.4595    0.6206       851

    accuracy                         0.7523      2479
   macro avg     0.8014    0.7574    0.7398      2479
weighted avg     0.8046    0.7523    0.7384      2479
```

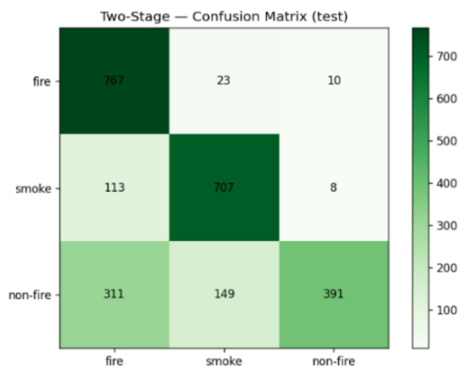


Figure 12. Two-Stage – Classification Report (test).

Figure 13. Two-Stage – Confusion Matrix (test).

Deployment

Our two-stage classifier is set up as a lightweight, CPU-first service that follows the training pipeline for clear and auditable inference. When it starts, it loads saved files: artifacts/models/stage1_scaler.joblib, artifacts/models/stage1_calibrated_svm.joblib, artifacts/reports/stage1_threshold.json (fixed from the validation PR curve), artifacts/models/stage2_scaler.joblib, and artifacts/models/stage2_calibrated_svm.joblib. Each input image is resized to 256×256 RGB, LAB-L equalised, and encoded with the same descriptor used in training (LBP + HOG + colour statistics, including a red-excess term). Stage 1 yields a calibrated hazard probability compared to the saved threshold; only gated hazards are passed to Stage 2 for FIRE/SMOKE classification. Outputs and figures (PR/ROC curves, confusion matrices, failure galleries) are written to artifacts/reports/ and artifacts/galleries/. No raw images are stored by default (failure galleries are intentionally saved), and the dataset's CC0 licence is respected. The pipeline has no GPU requirement; running on a GPU-equipped host is possible but provides no additional acceleration.

The end-to-end modelling process works with any dataset and can be easily applied to future collections. Onboarding involves scanning directories case-insensitively, filtering for extensions (.jpg/.jpeg/.png/.tif), and maintaining stable order. No additional splitting or training is performed at deployment; the service loads the saved artifacts and applies the existing pipeline to incoming images. Using the same handcrafted descriptor helps tolerate changes in camera make and resolution; if domain shift is substantial (e.g., fog or snow), the PR curve will reveal it, and the Stage-1 threshold can be reset without code changes. Failure galleries (top false positives/negatives) guide targeted curation and threshold policy.jpeg/.png/.

For still frames from live video, each frame acts like a regular input and goes through the same per-frame process. In practice, a thin temporal wrapper samples at a configurable rate (e.g., 0.5-2 fps). It debounces by requiring K of the last M frames to be hazards, for example, 3 out of 5. It stabilises FIRE/SMOKE using sliding-window aggregation, either through majority vote or mean probability. This approach reduces temporary false alarms and supports deployment at the edge. Artifacts are tracked under artifacts/{models, reports} for audits and rollbacks. The Stage-1 operating point can be adjusted by editing artifacts/reports/stage1_threshold.json and restoring the previous file if necessary.

Discussion and Challenges

For the most part, our aim to create an automated two-step bushfire detection by image classification system was satisfied. The Stage 1 Hazard Filter, with an ROC curve AUC of 0.947 and a PR curve AP of 0.968, exhibited good general classification performance, with consistently strong performance in recognising actual fire hazards.

A noted concern was the lagging recall on the non-Hazard class, indicating a substantial propensity for false alarms. We noted that false positives (false alarm) were preferable to false negatives (failure to detect) given the context of our intended deployment (catastrophic bushfires). Nonetheless, we do also have an interest in reducing the false positive rate.

Referring to the two-stage (overall) confusion matrix, we can see of those incorrectly classified of the actual non-fire samples, those predicting Fire attracted over two-thirds of the error (311 of 460 incorrectly classified).

By manual reference to the failure gallery, a few characteristics of the ‘True: Non-Fire, Pred: Fire’ samples stand out as potential reasons for misclassification, including jagged/incongruent lighter shapes contrasted with darker surrounds, reddish colours, significant light artifacts

Similarly, of the misclassified ‘True: Non-Fire, Pred: Smoke’ samples, some evident characteristics lending to the misclassification include greyish flowy/path-like shapes, significant haze/mist, presence of snow

A salient problem encountered in the implementation phase was the computational limitations afforded by working with Google Colab’s free offerings. This particularly manifested during the execution of the train/test/validation splitting and caching operation, which proved to be the most significant performance bottleneck of the code. In the interests of time, we had to reduce the dataset size to 10k images across all splits and save the extracted features to a shared Google Drive folder.

For the hyperparameter tuning, using random search across a large search space with many runs proved computationally costly, especially given Colab’s free-tier resource constraints. Thus, we had to lower the number of fits tried, meaning fewer hyperparameters could be tuned, and the best values found may not have been the optimal combination.

Conclusion

This project was able to reveal that a resource effective two-stage system of image classification is possible to detect bushfires automatically. The system offered most effectively the combination of traditional computer vision methods with machine learning classifiers, satisfying urgent demands concerning hazard detection in times when human control may not be available. This technique has reached a balance between accuracy, explainability, and computational efficiency-major factors in real control of emergency applications where time and reliability are of high importance. Two stage designs showed that conventional algorithms of manual feature extraction like LBP, HOG, and colour statistics, with appropriate settings and verification, can generate results of high quality and transparency. Though the system has not been as perfect as one would want it to be, with all computational constraints and difficulties with evidence datasets, it featured some good results in the detection and classification phase, which pinpoints its possible applicability in the domain of extensive early warning systems against bushfires. Further, this experience taught us how to make changes regarding precision–recall trade-offs, data quality, and model interpretability.

Future Recommendations

Acquisition of greater computational capability:
During the development of this project, the desirability of better computational capability was appreciated. Working with Google Colab’s free offerings, we found ourselves enacting compromises in the training process, notably in sample sizing and hyperparameter tuning, to benefit the timely running and testing of our project implementation. If we were to redo or extend upon this project, it would be clearly advisable to seek substantially enhanced computational capacity. This could mean Colab’s paid tier offerings or some other platforms.

Addressing Recall on Non-Hazard:
A recognised point of interest regarding the final model has been its difficulty in dealing with Non-Hazard/Non-Fire instances, namely its low Recall. Throughout the restricted hyperparameter tuning we undertook (given aforementioned resource limitations), the low recall on Non-Hazard recognition proved persistent. A recommended direction of future study involves re-attempting hyperparameter tuning with more comprehensive search ranges and hyperparameter sets, preferably after acquiring improved computational capability (Wang et al., 2025).

Integrate thermal imaging:
While normal optical image capture devices are presumably more cost-friendly for on-site bushfire early detection and warning, the potential of integrating thermal/infrared capture devices to work alongside them offers a tempting direction for future expansion of our project implementation. The benefit of being able to cross-reference our normal light-minded implementation with an infrared-minded one to validate an overall detection output could result in a more robust end system (Liu et al., 2023).

Explore different angles/deployment modes:
With the diversity of images provided in the dataset, ranging from satellite imagery to security footage to trail cams, we could conceivably pursue several more specific implementations for different perspectives, like a specific birds-eye implementation deployable for an aerial drone platform. We would likely have to seek additional samples to construct sufficiently large new datasets for each intended implementation, but such a direction could broaden the potential our chosen dataset offers (Bugaric et al., 2025).

Explore different classifiers:
The classifiers employed within this project implementation were selected with a focus on computationally lean operation. A potential future direction may be to explore implementations with less concern for computational lightness during normal operation, such as various neural networks. If such implementations can deliver better classification performance for acceptable computational demand, they may be worthwhile pursuing (Elhanashi et al., 2025).

Reference

Bugarić, M., Krstinić, D., Šerić, L., & Stipaničev, D. (2025). Current Trends in Wildfire Detection, Monitoring and Surveillance. Fire, 8(9), 356. https://doi.org/10.3390/fire8090356

Çelik, T., & Demirel, H. (2009). Fire detection in video sequences using a generic color model. Fire Safety Journal, 44(2), 147–158. https://doi.org/10.1016/j.firesaf.2008.05.005

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 1, 886–893 vol. 1. https://doi.org/10.1109/CVPR.2005.177

Davis, J., & Goadrich, M. (2006). The relationship between Precision-Recall and ROC curves. ICML 2006 : Proceedings, Twenty-Third International Conference on Machine Learning, 233–240. https://doi.org/10.1145/1143844.1143874

El-Madafri, I., Peña, M., & Olmedo-Torre, N. (2024). Real-Time Forest Fire Detection with Lightweight CNN Using Hierarchical Multi-Task Knowledge Distillation. Fire (Basel, Switzerland), 7(11), 392. https://doi.org/10.3390/fire7110392

Elhanashi, A., Essahraui, S., Dini, P., & Saponara, S. (2025). [Rev. of Early Fire and Smoke Detection Using Deep Learning: A Comprehensive Review of Models, Datasets, and Challenges]. Applied Sciences, 15(18), Article 10255. https://doi.org/10.3390/app151810255

Gragnaniello, D., Greco, A., Sansone, C., & Vento, B. (2024). Fire and smoke detection from videos: A literature review under a novel taxonomy. Expert Systems with Applications, 255, Article 124783. https://doi.org/10.1016/j.eswa.2024.124783

Jin, C., Wang, T., Alhusaini, N., Zhao, S., Liu, H., Xu, K., & Zhang, J. (2023). Video Fire Detection Methods Based on Deep Learning: Datasets, Methods, and Future Directions. Fire (Basel, Switzerland), 6(8), 315. https://doi.org/10.3390/fire6080315

Liu, Y., Zheng, C., Liu, X., Tian, Y., Zhang, J., & Cui, W. (2023). Forest Fire Monitoring Method Based on UAV Visual and Infrared Image Fusion. Remote Sensing (Basel, Switzerland), 15(12), Article 3173. https://doi.org/10.3390/rs15123173

Minha, A. (2023). *Forest fire, smoke, and non-fire image dataset* [Data set]. Kaggle. https://www.kaggle.com/datasets/amerzishminha/forest-fire-smoke-and-non-fire-image-dataset

Northcutt, C. G., Athalye, A., & Mueller, J. (2021). Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks. https://doi.org/10.48550/arxiv.2103.14749

Ojala, T., Pietikainen, M., & Maenpaa, T. (2002). Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24(7), 971–987. https://doi.org/10.1109/TPAMI.2002.1017623

Powers, D. M. W. (2020). Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. https://doi.org/10.48550/arxiv.2010.16061

Saito, T., & Rehmsmeier, M. (2015). The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. PloS One, 10(3), e0118432. https://doi.org/10.1371/journal.pone.0118432

Wang, C., Xu, C., Akram, A., Wang, Z., Shan, Z., & Zhang, Q. (2025). Wildfire Smoke Detection System: Model Architecture, Training Mechanism, and Dataset. International Journal of Intelligent Systems, 2025(1). https://doi.org/10.1155/int/1610145

Appendix

Link to Google Colab:

Google Colab:
https://colab.research.google.com/drive/1Ff7HIWK-z2c2NSRYcmEuTYLeDTW_czm2?usp=sharing

Code:

Import necessary libraries and set variables:

```
#if running in a fresh environment, uncomment to install minimal deps
#%pip install -q numpy pandas scikit-image scikit-learn matplotlib
opencv-python tqdm joblib

#import standard libraries
import os, sys, json, math, random, glob, time, shutil
from pathlib import Path
import numpy as np, collections
import pandas as pd
from tqdm import tqdm
import cv2
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
from sklearn.calibration import CalibratedClassifierCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import (classification_report, confusion_matrix,
roc_curve, precision_recall_curve, average_precision_score, auc)
from joblib import dump, load
from skimage.feature import local_binary_pattern, hog
from collections import Counter
import hashlib
import collections
import seaborn as sns
from sklearn.metrics import roc_curve, auc, precision_recall_curve,
average_precision_score,accuracy_score, f1_score,
classification_report, confusion_matrix

#set random seeds for reproducibility
np.random.seed(42)
random.seed(42)

plt.rcParams["figure.dpi"] = 120 #set resolution for figures
```

Mount to shared google drive:

```
from google.colab import drive
```

```
# Mount Google Drive
drive.mount('/content/drive', force_remount=True)

DATA_DIR =
'/content/drive/Shareddrives/image_processing/FOREST_FIRE_SMOKE_AND_NON
_FIRE_DATASET' #path to dataset in our team's shared google drive

#set image & pipeline settings
IMG_SIZE = (256, 256) #width x height
VAL_FRACTION = 0.2 #validation split fraction
RANDOM_SEED = 42 #random seed for reproducibility
N_JOBS = max(1, os.cpu_count() - 1) #number of parallel jobs for
feature extraction & kNN

#define feature/caching directories
ARTIFACTS_DIR = Path("./artifacts")
CACHE_DIR = ARTIFACTS_DIR / "cache"
MODEL_DIR = ARTIFACTS_DIR / "models"
REPORT_DIR = ARTIFACTS_DIR / "reports"
GALLERY_DIR = ARTIFACTS_DIR / "galleries"

#create directories if they don't exist
for d in [CACHE_DIR, MODEL_DIR, REPORT_DIR, GALLERY_DIR]:
    d.mkdir(parents=True, exist_ok=True)

#define classes and label mapping
CLASSES = ["fire", "smoke", "non-fire"]
LABEL_MAP = {c: i for i, c in enumerate(CLASSES)}
HAZARD_SET = {"fire", "smoke"}
```

Verify dataset structure and img counts:

```
#find all image extensions types from datset
file_extensions = []
for root, dirs, files in os.walk(DATA_DIR): #loop through dataset
directory
    for file in files:
        _, ext = os.path.splitext(file) #get every file extension
        if ext:
            file_extensions.append(ext.lower()) #append to list in
lowercase

extension_counts = Counter(file_extensions) #count occurrences of each
file extension

#print file extension counts
print("File extensions found in", DATA_DIR, "and their counts:")
for ext, count in extension_counts.most_common():
    print(f"- {ext}: {count}")
```

```
#scan dataset directory to verify folder structure and image counts

IMG_EXTS = {".jpg",".jpeg",".png",".tif"} #leave out .gif extensions

root = Path(DATA_DIR) #dataset root
for split in ["train","test"]:
    d = root / split #path to train and test directories
    print(f"\n[{split}] subfolders:")
    for p in d.iterdir(): #iterate through subdirectories
        if p.is_dir(): #print the name and image count of each
subdirectory
            print("  -", p.name, "(count:", sum(1 for _ in p.rglob("*")
if _.suffix.lower() in IMG_EXTS),")")
```

Store all valid images for train and test:

```
#store all valid image files for train and test splits

def scan_split(split_dir): #
    split_dir = Path(split_dir) #create Path object
    counts = {c: 0 for c in CLASSES} #initialize counts dictionary
    files = [] #list to hold file paths and labels

    #map folder names to directory paths
    name_to_dir = {
        child.name.casefold(): child
        for child in split_dir.iterdir()
        if child.is_dir()
    }

    #iterate through each class
    for cname in CLASSES:
        d = name_to_dir.get(cname.casefold()) #get directory for class
        images = [] #list to hold image paths for this class
        for p in d.rglob("*"): #recursively search for image files
            if p.is_file() and p.suffix.lower() in IMG_EXTS:
                images.append(p) #append valid image paths

        counts[cname] = len(images) #add image count
        files.extend([(str(p), cname) for p in images]) #add file paths
and class for every valid image found

    return counts, files

#find all valid images in train and test directories
train_counts, train_files = scan_split(Path(DATA_DIR) / "train")
test_counts,  test_files  = scan_split(Path(DATA_DIR) / "test")

#print summary of image counts
print("Train counts:", train_counts)
print("Test counts :", test_counts)
print("Total images:", len(train_files) + len(test_files))
```

Feature Extraction Functions:

```
#read images and set colour and size settings
def read_image(path, size=IMG_SIZE):
    img = cv2.imread(path) #read image using OpenCV
    if img is None: #if image reading fails, raise an error
        raise IOError(f"Failed to read image: {path}")
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) #convert BGR to RGB
    img = cv2.resize(img, size, interpolation=cv2.INTER_AREA) #resize
image
    return img

#normalise image using LAB colour space
def normalize_image(img):
    img_lab = cv2.cvtColor(img, cv2.COLOR_RGB2LAB) #convert RGB to LAB
    l, a, b = cv2.split(img_lab) #split LAB channels
    l = cv2.equalizeHist(l) #equalize L channel histogram
    img_lab = cv2.merge([l, a, b]) #merge channels back
    out = cv2.cvtColor(img_lab, cv2.COLOR_LAB2RGB) #convert LAB back to
RGB
    return out #return normalised image

#light augmentation for model training robustness
def light_augment(img):
    out = img.copy()
    if random.random() < 0.5: #50% chance image is flipped left to
right
        out = np.fliplr(out).copy()
    if random.random() < 0.3: #30 chance image is rotated at a random
angle
        angle = random.uniform(-5, 5)
        M = cv2.getRotationMatrix2D((out.shape[1]/2, out.shape[0]/2),
angle, 1.0)
        out = cv2.warpAffine(out, M, (out.shape[1], out.shape[0]),
flags=cv2.INTER_LINEAR,
                             borderMode=cv2.BORDER_REFLECT_101)
    if random.random() < 0.3: #30% chance brightness and contrast are
randomly adjusted
        alpha = random.uniform(0.9, 1.1) #contrast
        beta  = random.uniform(-10, 10) #brightness
        out = cv2.convertScaleAbs(out, alpha=alpha, beta=beta)
    return out

#LBP feature extraction variables
LBP_P = 8
LBP_R = 1
LBP_METHOD = "uniform"
```

```
LBP_NBINS = LBP_P + 2

#LBP feature extraction
def features_lbp(gray):
    lbp = local_binary_pattern(gray, P=LBP_P, R=LBP_R,
method=LBP_METHOD) #compute LBP
    hist, _ = np.histogram(lbp.ravel(), bins=LBP_NBINS, range=(0,
LBP_NBINS), density=True) #compute histogram
    return hist.astype(np.float32) #return histogram

#HOG feature extraction
def features_hog(gray):
    fd = hog(gray, orientations=9, pixels_per_cell=(16, 16),
cells_per_block=(2, 2), block_norm='L2-Hys', visualize=False,
feature_vector=True) #compute HOG features
    return fd.astype(np.float32) #return HOG features

#colour features extraction
def features_colour(img_rgb):
    img_hsv = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2HSV) #convert RGB to
HSV
    img_ycc = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2YCrCb) #convert RGB
to YCrCb
    feats = []
    for ch in cv2.split(img_hsv): #loop through HSV channels
        feats += [ch.mean(), ch.std(), float(np.median(ch)), ch.min(),
ch.max()] #compute mean, std, median, min, max for each channel
    Y, Cr, Cb = cv2.split(img_ycc) #split YCrCb channels
    feats += [Cr.mean(), Cr.std(), Cb.mean(), Cb.std(), (Cr.mean() -
Cb.mean())] #compute stats for Cr and Cb channels
    r, g, b = cv2.split(img_rgb) #split RGB channels
    reds = r.mean() - ((g.mean() + b.mean())/2.0) #compute redness
feature
    feats += [reds] #append redness feature
    return np.array(feats, dtype=np.float32) #return all colour
features

#combined feature extraction function
def extract_features(path):
    img = read_image(path, IMG_SIZE)
    img = normalize_image(img)
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    lbp = features_lbp(gray)
    hg  = features_hog(gray)
    col = features_colour(img)
    return np.concatenate([lbp, hg, col], axis=0) #return combined
features
```

```
#combined feature extraction with augmentation for taining images
def extract_features_with_aug(path, is_train=False):
    img = read_image(path, IMG_SIZE)
    img = normalize_image(img)
    if is_train:
        img = light_augment(img) #apply augmentation if training image
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    lbp = features_lbp(gray)
    hg  = features_hog(gray)
    col = features_colour(img)
    return np.concatenate([lbp, hg, col], axis=0) #return combined
features
```

Data Processing Pipeline:

```
"""
1. build a deterministic 10k image subset for train and test, then
create a stratified train and validation (20% of train data) split
2. label stage 1 classes, hazard (fire and smoke classes) or non hazar,
and derive stage 2 classes by filtering to for only fire and smoke
class labels
3. extract features per image with light augmentation for training then
cache X and y for each split to limit rerunning the pipeline and lower
compuatation
"""

MAX_IMAGES = 10000 #limit test and train splits to 10k images combined
(lower computation load)

#stable key function for reproducible sorting
def stable_key(path_str: str) -> str:
    return hashlib.sha1(path_str.encode('utf-8')).hexdigest()

#build initial sorted lists for train and test image files
train_files_sorted_all = sorted(train_files, key=lambda t: t[1])
test_files_sorted_all  = sorted(test_files,  key=lambda t: t[1])

#combine train and test files with tags
all_tagged = [('train', p, c) for p, c in train_files_sorted_all] + \
             [('test',  p, c) for p, c in test_files_sorted_all]

all_tagged_sorted = sorted(all_tagged, key=lambda t:
stable_key(str(t[1]))) #sort by stable key for reproducibility

#create 10k image subset
subset = all_tagged_sorted[:min(MAX_IMAGES, len(all_tagged_sorted))]

#recreate train and test image lists with data subset
train_files_sorted = [(p, c) for s, p, c in subset if s == 'train']
test_files_sorted  = [(p, c) for s, p, c in subset if s == 'test']
```

```
#print summary of split
print(f"Using a fixed subset of {len(subset)} images
"f"(train={len(train_files_sorted)}, test={len(test_files_sorted)})")

#build train/validation split with stratification
def build_split(files_labels, VAL_FRACTION, seed=42): #
    X_paths = [p for p, _ in files_labels] #extract image paths
    y_cls   = [LABEL_MAP[c] for _, c in files_labels] #extract class
labels

    #stratified train/validation split
    train_idx, val_idx = train_test_split(
        np.arange(len(X_paths)),
        test_size=VAL_FRACTION,
        random_state=seed,
        stratify=y_cls
    )
    #function to subset data based on indices
    def subset(idxs):
        return [X_paths[i] for i in idxs], [y_cls[i] for i in idxs]

    return subset(train_idx), subset(val_idx) #return train and
validation splits

#stage 1 class lables, hazard (fire/smoke) vs non-hazard
def labels_stage1(y_cls):
    return np.array([1 if CLASSES[y] in {"fire","smoke"} else 0 for y
in y_cls], dtype=np.int64) #1:hazard, 0:non-hazard

#stage 2 filtering for fire and smoke classes only
def filter_stage2(X_paths, y_cls):
    X2, y2 = [], []
    for p, y in zip(X_paths, y_cls):
        cname = CLASSES[y] #get class name
        if cname in {"fire", "smoke"}:
            X2.append(p)
            y2.append(0 if cname=="fire" else 1)  #0:fire, 1:smoke
    return X2, np.array(y2, dtype=np.int64)

#feature caching function to avoid rerunning lenghty code
def cache_features(split_name, X_paths, y_vec, is_train=False):
    cache_path = Path("./artifacts/cache") / f"{split_name}.npz" #path
to temporary cache directory
    cache_path.parent.mkdir(parents=True, exist_ok=True)
    if cache_path.exists():
        data = np.load(cache_path, allow_pickle=False) #load cached
data
```

```
        return data["X"], data["y"]
    feats = []

    #extract features for each image and store in list
    for p in tqdm(X_paths, desc=f"Extracting {split_name}"):
        f = extract_features_with_aug(p, is_train=is_train) #extract
features with augmentation if training data
        feats.append(f) #append features to list

    #convert features and labels to numpy arrays and save to cache
    X = np.vstack(feats).astype(np.float32)
    y = np.array(y_vec, dtype=np.int64)
    np.savez_compressed(cache_path, X=X, y=y)

    return X, y

#build train/validation splits (validation is 20% of training data)
(train_X_paths, train_y_cls), (val_X_paths, val_y_cls) =
build_split(train_files_sorted, VAL_FRACTION, 42)

#build test data paths and labels
test_X_paths = [p for p,_ in test_files_sorted]
test_y_cls   = [LABEL_MAP[c] for _,c in test_files_sorted]

#print summary of dataset splits
print(f"Train/Val/Test sizes (images):
{len(train_X_paths)}/{len(val_X_paths)}/{len(test_X_paths)}")

#save test_y_cls to cache for later use
test_y_cls_cache_path = Path("./artifacts/cache") / "test_y_cls.npy"
#cache path for test_y_cls
test_y_cls_cache_path.parent.mkdir(parents=True, exist_ok=True) #make
directory if it doesn't exist
np.save(test_y_cls_cache_path, test_y_cls) #save test_y_cls to cache
print(f"Saved test_y_cls to {test_y_cls_cache_path}") #print
confirmation

#save test_X_paths to cache for later use
test_X_paths_cache_path = Path("./artifacts/cache") /
"test_X_paths.npy" #cache path for test_X_paths
test_X_paths_cache_path.parent.mkdir(parents=True, exist_ok=True) #make
directory if it doesn't exist
np.save(test_X_paths_cache_path, test_X_paths) #save test_X_paths to
cache
print(f"Saved test_X_paths to {test_X_paths_cache_path}") #print
confirmation

#label data for stage 1, hazard (fire & smoke classes) vs non-hazard
```

```
y1_tr = labels_stage1(train_y_cls)
y1_va = labels_stage1(val_y_cls)
y1_te = labels_stage1(test_y_cls)

#cache features for stage 1
X1_tr, y1_tr = cache_features("stage1_train", train_X_paths, y1_tr,
is_train=True)
X1_va, y1_va = cache_features("stage1_val",   val_X_paths,   y1_va,
is_train=False)
X1_te, y1_te = cache_features("stage1_test",  test_X_paths,  y1_te,
is_train=False)

#filter data for stage 2 so only fire and smoke classes remain
X2_tr_paths, y2_tr = filter_stage2(train_X_paths, train_y_cls)
X2_va_paths, y2_va = filter_stage2(val_X_paths,   val_y_cls)
X2_te_paths, y2_te = filter_stage2(test_X_paths,  test_y_cls)

#cache features for stage 2
X2_tr, y2_tr = cache_features("stage2_train", X2_tr_paths, y2_tr,
is_train=True)
X2_va, y2_va = cache_features("stage2_val",   X2_va_paths, y2_va,
is_train=False)
X2_te, y2_te = cache_features("stage2_test",  X2_te_paths, y2_te,
is_train=False)

#print summary of feature shapes for stage 1 and stage 2
print("Stage1 shapes:", X1_tr.shape, X1_va.shape, X1_te.shape)
print("Stage2 shapes:", X2_tr.shape, X2_va.shape, X2_te.shape)
```

Save or Load Extracted Features:

```
DATA_DIR = '/content/drive/Shareddrives/image_processing' #path to
dataset in our team's shared google drive
drive_cache_dir = Path(DATA_DIR) / 'artifacts' / 'cache' #path to cache
directory on in shared drive
drive_cache_dir.mkdir(parents=True, exist_ok=True) #make directory if
it doesn't exist

temp_cache_dir = Path('/content') / 'artifacts' / 'cache' #path to
temporary cache directory in colab


#expected saved split (file) names
split_names = [
    'stage1_train', 'stage1_val', 'stage1_test',
    'stage2_train', 'stage2_val', 'stage2_test'
]

#copy saved extracted features from temp cache to shared drive if
present
```

```
for name in split_names:
    src = temp_cache_dir / f'{name}.npz' #source path in temp cache
    dst = drive_cache_dir / f'{name}.npz' #destination path in shared
drive cache
    if src.exists(): #if temp file exists, copy to destination
        shutil.copy2(src, dst)
        print(f'Copied {src} -> {dst}')
    else: #if temp file missing, try loading from drive
        print(f'No temp file for {name}, will try loading from drive')

#function to load npz files from shared drive cache
def load_npz_from_drive(name: str):
    p = drive_cache_dir / f'{name}.npz' #path to npz file in shared
drive
    if not p.exists():
        raise FileNotFoundError(f'Missing file on drive: {p}')
    data = np.load(p, allow_pickle=False) #load data from npz file
    return data['X'], data['y'] #return features and labels

print('\n')

#load cached features for stage 1 and stage 2 from shared drive
X1_tr, y1_tr = load_npz_from_drive('stage1_train')
X1_va, y1_va = load_npz_from_drive('stage1_val')
X1_te, y1_te = load_npz_from_drive('stage1_test')

X2_tr, y2_tr = load_npz_from_drive('stage2_train')
X2_va, y2_va = load_npz_from_drive('stage2_val')
X2_te, y2_te = load_npz_from_drive('stage2_test')

#confirm loaded feature shapes
print(f'Loaded features from: {drive_cache_dir}')
print("Stage1 shapes:", X1_tr.shape, X1_va.shape, X1_te.shape)
print("Stage2 shapes:", X2_tr.shape, X2_va.shape, X2_te.shape, '\n')

#copy test_y_cls.npy to shared drive if present in temp, else load from
Drive
temp_test_path  = temp_cache_dir / "test_y_cls.npy" #path to temp
test_y_cls.npy
drive_test_path = drive_cache_dir / "test_y_cls.npy" #path to drive
test_y_cls.npy

if temp_test_path.exists(): #if test_y_cls exists in temp, copy it to
shared drive
    shutil.copy2(temp_test_path, drive_test_path)
    print(f"Copied {temp_test_path} -> {drive_test_path}")
elif drive_test_path.exists(): #else if test_y_cls exists in shared
drive, load it
```

```
    test_y_cls = np.load(drive_test_path, allow_pickle=False)
    print(f"Loaded test_y_cls from {drive_test_path}") #print
confirmation
else:
    raise FileNotFoundError(f"Missing test_y_cls.npy in both temp
({temp_test_path}) and drive ({drive_test_path})")

#copy test_X_paths.npy to shared drive if present in temp, else load
from Drive
temp_test_path  = temp_cache_dir / "test_X_paths.npy" #path to temp
test_X_paths.npy
drive_test_path = drive_cache_dir / "test_X_paths.npy" #path to drive
test_X_paths.npy

#if test_X_paths exists in temp, copy it to shared drive
if temp_test_path.exists():
    shutil.copy2(temp_test_path, drive_test_path)
    print(f"Copied {temp_test_path} -> {drive_test_path}")
elif drive_test_path.exists(): #else if test_X_paths exists in shared
drive, load it
    test_X_paths = np.load(drive_test_path, allow_pickle=False)
    print(f"Loaded test_X_paths from {drive_test_path}")
else:
    raise FileNotFoundError(f"Missing test_X_paths.npy in both temp
({temp_test_path}) and drive ({drive_test_path})")
```

Stage 1 Implementation and Evaluation:

```
#Stage 1: Calibrated Linear SVM with random search tuning for C and
max_iterations

scaler1 = StandardScaler().fit(X1_tr) #fit scaler on training data

#scale stage 1 data
X1_tr_s = scaler1.transform(X1_tr)
X1_va_s = scaler1.transform(X1_va)
X1_te_s = scaler1.transform(X1_te)

#RANDOM SEARCH FOR BEST C AND MAX_ITER VALUES
rng = np.random.RandomState(42) #random number generator with fixed
seed

#generate random C value from log-uniform distribution
def C_value(low=1e-3, high=1e3):
    return float(10 ** rng.uniform(np.log10(low), np.log10(high)))

#generate random max_iter value from predefined set
def max_iter():
    return int(rng.choice([2000, 3000, 4000, 5000]))
```

```
fits = 25 #number of random search iterations

best_ap = -np.inf #best average precision score
best_C = None #best C value
best_iter = None #best max_iter value
fit_count=0 #fit counter

#random search loop
for _ in range(fits):
    fit_count+=1 #increment fit counter
    C = C_value() #generate random C value
    iters = max_iter() #generate random max_iter value

    #train Linear SVM with random hyperparameters
    svc = LinearSVC(C=C, class_weight="balanced", random_state=42, 
max_iter=iters)
    svc.fit(X1_tr_s, y1_tr) #train on train data
    ap = average_precision_score(y1_va, svc.decision_function(X1_va_s)) 
#score on validation data

    print(fit_count, '. Testing C=', C, ' Iter=', iters, ' AP=', ap) 
#print current fit results

    #update best hyperparameters if current AP is better
    if ap > best_ap:
        best_ap, best_C, best_iter = ap, C, iters

print(f"[Stage1] Random search best C={best_C:.6g}, 
max_iter={best_iter}, val AP={best_ap:.4f}") #print best 
hyperparameters


#train final Linear SVM with best hyperparameters
base_svc = LinearSVC(C=best_C, class_weight='balanced', 
random_state=42, max_iter=best_iter)
base_svc.fit(X1_tr_s, y1_tr)

#calibrate classifier using sigmoid method
clf1 = CalibratedClassifierCV(base_svc, method='sigmoid', cv='prefit')
clf1.fit(X1_tr_s, y1_tr)

#evaluate on validation set to choose threshold
probs_va = clf1.predict_proba(X1_va_s)[:,1] #get probabilities for 
positive class
prec, rec, thr = precision_recall_curve(y1_va, probs_va) #compute 
precision-recall curve
ap_va = average_precision_score(y1_va, probs_va) #compute average 
precision score
```

```
target_precision = 0.80 #desired precision level for threshold
selection
best_idx = None

#find threshold achieving target precision
for i in range(len(thr)):
    if prec[i] >= target_precision:
        best_idx = i
        break
if best_idx is None: #if no threshold meets target precision, choose
one maximizing F1-score
    f1s = 2*prec[:-1]*rec[:-1]/(prec[:-1]+rec[:-1]+1e-9)
    best_idx = int(np.argmax(f1s))

thr_star = thr[best_idx] if best_idx < len(thr) else 0.5 #chosen
threshold

print(f"[Stage1] AP@Val = {ap_va:.4f} | Chosen threshold =
{thr_star:.3f} | precision={prec[best_idx]:.3f},
recall={rec[best_idx]:.3f}") #print chosen threshold and metrics

#compute ROC curve and AUC for validation set
fpr, tpr, roc_thr = roc_curve(y1_va, probs_va)
roc_auc = auc(fpr, tpr)

#plot PR and ROC curves for validation set
plt.figure(); plt.plot(rec, prec); plt.xlabel("Recall");
plt.ylabel("Precision"); plt.title("Stage 1 — PR (val)");
plt.grid(True); plt.show()
plt.figure(); plt.plot(fpr, tpr); plt.xlabel("FPR"); plt.ylabel("TPR");
plt.title(f"Stage 1 — ROC (val) AUC={roc_auc:.3f}"); plt.grid(True);
plt.show()

#evaluate on test set using chosen threshold
probs_te = clf1.predict_proba(X1_te_s)[:,1]
y1_hat = (probs_te >= thr_star).astype(int)
print("[Stage1] Test classification report:")
print(classification_report(y1_te, y1_hat, target_names=["non-
hazard","hazard"], digits=4))

#compute and plot confusion matrix for test set
cm1 = confusion_matrix(y1_te, y1_hat)
plt.figure(); plt.imshow(cm1, cmap='Blues'); plt.title("Stage 1 —
Confusion Matrix (test)"); plt.colorbar()
plt.xticks([0,1], ["non-hazard","hazard"]); plt.yticks([0,1], ["non-
hazard","hazard"])
for (i,j),v in np.ndenumerate(cm1): plt.text(j, i, str(v), ha='center',
va='center')
```

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

#compute PR curve and AP on test set
prec_te, rec_te, _ = precision_recall_curve(y1_te, probs_te)
ap_te = average_precision_score(y1_te, probs_te)

#compute ROC curve and AUC on test set
fpr_te, tpr_te, _ = roc_curve(y1_te, probs_te)
roc_auc_te = auc(fpr_te, tpr_te)

#plot PR and ROC curves for test set
plt.figure(); plt.plot(rec_te, prec_te); plt.xlabel("Recall");
plt.ylabel("Precision"); plt.title(f"Stage 1 — PR Curve (test) |
AP={ap_te:.3f}"); plt.grid(True); plt.tight_layout(); plt.show()
plt.figure(); plt.plot(fpr_te, tpr_te); plt.plot([0, 1], [0, 1],
linestyle='--', color='grey'); plt.xlabel("False Positive Rate");
plt.ylabel("True Positive Rate"); plt.title(f"Stage 1 — ROC Curve
(test) | AUC={roc_auc_te:.3f}"); plt.grid(True); plt.tight_layout();
plt.show()

#save stage 1 models, scaler, and threshold to artifacts directory
Path("./artifacts/models").mkdir(parents=True, exist_ok=True)
Path("./artifacts/reports").mkdir(parents=True, exist_ok=True)
dump(scaler1, "./artifacts/models/stage1_scaler.joblib")
dump(clf1,    "./artifacts/models/stage1_calibrated_svm.joblib")
with open("./artifacts/reports/stage1_threshold.json", "w") as f:
    json.dump({"threshold": float(thr_star), "target_precision":
target_precision}, f, indent=2)
```

Stage 2 Implementation and Evaluation:

Random Search for both Stage 2 SVM and kNN Classifiers.

```
#Random Search for Best Hyperparameters for SVM and KNN

#scale features
scaler2 = StandardScaler().fit(X2_tr)
X2_tr_s = scaler2.transform(X2_tr)
X2_va_s = scaler2.transform(X2_va)
X2_te_s = scaler2.transform(X2_te)

rng2 = np.random.RandomState(42) #random number generator with fixed
seed

#SVM random search (C, max_iter)
def C_value(low=1e-3, high=1e3): #generate random C value from log-
uniform distribution
    return float(10 ** rng2.uniform(np.log10(low), np.log10(high)))

def max_iter(): #generate random max_iter value from predefined set
    return int(rng2.choice([1000, 2000, 3000, 4000]))
```

```
fits = 25 #no. of random search iterations
best_ap = -np.inf #best average precision score
best_C2 = None #best C value
best_iter2 = None #best max_iter value
fit_count = 0 #fit counter

#random search loop for stage 2 SVM
for _ in range(fits):
    fit_count += 1 #increment fit counter
    C = C_value() #generate random C value
    iters = max_iter() #generate random max_iter value

    #train Linear SVM with random hyperparameters
    svc_2 = LinearSVC(C=C, class_weight="balanced", random_state=42,
max_iter=iters)
    svc_2.fit(X2_tr_s, y2_tr) #train on stage 2 train data
    ap_val = average_precision_score(y2_va,
svc_2.decision_function(X2_va_s)) #score on stage 2 validation data

    print(f"[Stage2 SVM] {fit_count}. C={C:.6g} iters={iters}
AP_val={ap_val:.4f}") #print current fit results

    #update best hyperparameters if current AP is better
    if ap_val > best_ap:
        best_ap, best_C2, best_iter2 = ap_val, C, iters

print(f"[Stage2 SVM] best C={best_C2:.6g}, max_iter={best_iter2}, val
AP={best_ap:.4f}, \n") #print best hyperparameters

#kNN Random Search
def sample_n_neighbors(): #generate random n_neighbors value from
predefined set
    return int(rng2.choice([3, 5, 7, 9, 11, 15]))

def sample_weights(): #randomly choose weights type
    return str(rng2.choice(['uniform', 'distance']))

fits = 25 #number of random search iterations for kNN
best_ap = -np.inf #best average precision score
best_k = None #best n_neighbors value
best_w = None #best weights type
fit_count = 0 #fit counter

#random search loop for kNN
for _ in range(fits):
    fit_count += 1 #increment fit counter
    k = sample_n_neighbors() #generate random n_neighbors value
    w = sample_weights() #generate random weights type

    #train kNN with random hyperparameters
    knn = KNeighborsClassifier(n_neighbors=k, weights=w, n_jobs=max(1,
os.cpu_count()-1))
    knn.fit(X2_tr_s, y2_tr) #train on stage 2 train data
    ap_val = average_precision_score(y2_va,
knn.predict_proba(X2_va_s)[:, 1]) #score on stage 2 validation data

    print(f"[Stage2 kNN] {fit_count}. k={k} weights={w}
AP_val={ap_val:.4f}") #print current fit results

    #update best hyperparameters if current AP is better
    if ap_val > best_ap:
        best_ap, best_k, best_w = ap_val, k, w

print(f"[Stage2 kNN] best k={best_k}, weights={best_w}, val
AP={best_ap:.4f}, \n") #print best hyperparameters
```

Train classifiers with best hyperparameters.

```
#fit final stage 2 SVM with best hyperparameters
svc2 = LinearSVC(C=best_C2, class_weight='balanced', random_state=42,
max_iter=best_iter2)
svc2.fit(X2_tr_s, y2_tr)

#calibrate classifier
clf2 = CalibratedClassifierCV(svc2, method='sigmoid', cv='prefit')
clf2.fit(X2_tr_s, y2_tr)

#quick val score for SVM
probs2_va = clf2.predict_proba(X2_va_s)[:,1]
ap2_va = average_precision_score(y2_va, probs2_va)
print(f"[Stage2] AP@Val (SVM) = {ap2_va:.4f}")

y2_hat_te = clf2.predict(X2_te_s) #predict on test set

#print cr for test set with SVM
print("[Stage2] Test classification report (SVM):")
print(classification_report(y2_te, y2_hat_te,
target_names=["fire","smoke"], digits=4))

#compute and plot confusion matrix for test set with SVM
cm2 = confusion_matrix(y2_te, y2_hat_te)
plt.figure(); plt.imshow(cm2, cmap='Purples'); plt.title("Stage 2 — SVM
Confusion Matrix (test)"); plt.colorbar()
plt.xticks([0,1], ["fire","smoke"]); plt.yticks([0,1],
["fire","smoke"])
for (i,j),v in np.ndenumerate(cm2): plt.text(j, i, str(v), ha='center',
va='center')
```

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

Path("./artifacts/models").mkdir(parents=True, exist_ok=True) #make
models directory if it doesn't exist
plt.savefig("artifacts/reports/stage2_confusion_test.png") #save
confusion matrix figure for stage 2 svm

#fit final kNN model with best hyperparameters
knn = KNeighborsClassifier(n_neighbors=best_k, weights=best_w,
n_jobs=max(1, os.cpu_count()-1))
knn.fit(X2_tr_s, y2_tr)

y2_hat_knn = knn.predict(X2_te_s) #predict on test set with kNN

#print cr for test set with kNN
print("[Stage2] Test classification report (kNN):")
print(classification_report(y2_te, y2_hat_knn,
target_names=["fire","smoke"], digits=4))

#compute and plot confusion matrix for test set with kNN
cm_knn = confusion_matrix(y2_te, y2_hat_knn)
plt.figure(); plt.imshow(cm_knn, cmap='Purples'); plt.title("Stage 2 —
kNN Confusion Matrix (test)"); plt.colorbar()
plt.xticks([0,1], ["fire","smoke"]); plt.yticks([0,1],
["fire","smoke"])
for (i,j), v in np.ndenumerate(cm_knn): plt.text(j, i, str(v),
ha='center', va='center')
plt.tight_layout(); plt.show()

#save stage 2 models and scaler to artifacts directory
dump(scaler2, "./artifacts/models/stage2_scaler.joblib")
dump(clf2,    "./artifacts/models/stage2_calibrated_svm.joblib")
dump(knn,    "./artifacts/models/stage2_knn.joblib")
```

Stage 2 SVM Evaluation.

```
#probabilities from calibrated SVM
probs2_te = clf2.predict_proba(X2_te_s)[:, 1]

#compute ROC and AUC
fpr2, tpr2, _ = roc_curve(y2_te, probs2_te)
roc_auc2 = auc(fpr2, tpr2)

precision2, recall2, _ = precision_recall_curve(y2_te, probs2_te)
#compute precision recall curve
ap2 = average_precision_score(y2_te, probs2_te) #compute average
precision

#compute accuracy & F1 scores
acc2 = accuracy_score(y2_te, y2_hat_te)
```

```
macro_f1_2 = f1_score(y2_te, y2_hat_te, average='macro')
micro_f1_2 = f1_score(y2_te, y2_hat_te, average='micro')

print(f"[Stage2 SVM] Accuracy={acc2:.4f} | Macro-F1={macro_f1_2:.4f} |
Micro-F1={micro_f1_2:.4f} | AP={ap2:.4f} | AUC={roc_auc2:.4f}") #print
evaluation metrics

#plot ROC curve
plt.figure(figsize=(11,5))
plt.subplot(1,2,1)
plt.plot(fpr2, tpr2, color='darkorange', lw=2,
label=f"AUC={roc_auc2:.3f}")
plt.plot([0,1],[0,1],'--',color='grey')
plt.title("Stage 2 SVM - ROC Curve")
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate");
plt.legend()

#plot precision recall curve
plt.subplot(1,2,2)
plt.plot(recall2, precision2, color='teal', lw=2)
plt.title(f"Stage 2 SVM - Precision-Recall Curve (AP={ap2:.3f})")
plt.xlabel("Recall"); plt.ylabel("Precision")
plt.tight_layout()
plt.show()

#plot confusion matrix in percent
cm2 = confusion_matrix(y2_te, y2_hat_te)
cmn = cm2.astype('float') / cm2.sum(axis=1)[:, np.newaxis]
plt.figure(figsize=(5,4))
sns.heatmap(cmn, annot=True, fmt=".2%", cmap="Purples", cbar=False)
plt.title("Stage 2 SVM - Confusion Matrix (Percent)")
plt.xlabel("Predicted"); plt.ylabel("True")
plt.xticks([0.5,1.5], ["Fire","Smoke"]); plt.yticks([0.5,1.5],
["Fire","Smoke"])
plt.show()

#bar chart pe class precision, recall and f1
metrics2 = classification_report(
    y2_te, y2_hat_te,
    target_names=["Fire","Smoke"], output_dict=True
)
df2 = pd.DataFrame(metrics2).T.loc[["Fire","Smoke"],
["precision","recall","f1-score"]]
df2.plot(kind='bar', figsize=(7,4))
plt.title("Stage 2 - Per-Class Precision, Recall & F1")
plt.ylabel("Score")
plt.ylim(0,1)
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()

#save reports for stage 2 svm
Path("artifacts/reports").mkdir(parents=True, exist_ok=True)
plt.savefig("artifacts/reports/stage2_roc_pr_curves.png")
df2.to_csv("artifacts/reports/stage2_per_class_metrics.csv")
```

End to End Two Stage Implementation:

```
#run end to end two stage evaluation
def run_two_stage(X_paths, y_cls, scaler1, clf1, thr1, scaler2, clf2):
    final_preds = []
    for p, y in tqdm(list(zip(X_paths, y_cls)), desc="Two-stage
inference"): #loop through image paths and true labels
        f = extract_features(p) #extract features without augmentation
        f1 = scaler1.transform([f]) #scale feature for stage 1
        prob_hazard = clf1.predict_proba(f1)[0,1] #probability of
hazard class
        if prob_hazard >= thr1: #if hazard probability exceeds
threshold (classified as hazard)
            #perform stage 2 classification for fire vs smoke
            f2 = scaler2.transform([f]) #scale feature for stage 2
            y2 = clf2.predict(f2)[0]  # 0:fire, 1:smoke
            final_preds.append(y2) #append fire/smoke prediction
        else:
            final_preds.append(2) #classify as non-hazard
    return np.array(final_preds, dtype=np.int64) #return final
predictions array

with open("./artifacts/reports/stage1_threshold.json") as f:
    THR1 = json.load(f)["threshold"] #load saved stage 1 threshold

y_true_3 = np.array([LABEL_MAP[c] for c in [list(LABEL_MAP.keys())[i]
for i in test_y_cls]], dtype=np.int64) #true class labels for test set

#run two-stage evaluation on test set
y_pred_3 = run_two_stage(test_X_paths,
                         test_y_cls,

load("./artifacts/models/stage1_scaler.joblib"),

load("./artifacts/models/stage1_calibrated_svm.joblib"),
                         THR1,

load("./artifacts/models/stage2_scaler.joblib"),

load("./artifacts/models/stage2_calibrated_svm.joblib"))

#print end to end two stage classification report
```

```
print("End-to-end (two-stage) — Test classification report:")
print(classification_report(y_true_3, y_pred_3,
target_names=["fire","smoke","non-fire"], digits=4))

#show confusion matrix for end to end two stage classification
cm3 = confusion_matrix(y_true_3, y_pred_3, labels=[0,1,2])
plt.figure(); plt.imshow(cm3, cmap='Greens'); plt.title("Two-Stage —
Confusion Matrix (test)"); plt.colorbar()
plt.xticks([0,1,2], ["fire","smoke","non-fire"]); plt.yticks([0,1,2],
["fire","smoke","non-fire"])
for (i,j),v in np.ndenumerate(cm3): plt.text(j, i, str(v), ha='center',
va='center')
plt.tight_layout(); plt.show()

with open("./artifacts/reports/metrics_summary.json", "w") as f:
    json.dump({"stage1_threshold": float(THR1), "confusion_end_to_end":
cm3.tolist()}, f, indent=2) #save metrics summary to json
```

Gallery of Failed Predictions from Combined Stage 1 & 2:

```
#function to save a gallery of failed class predictions per class
def save_failure_gallery(paths, y_true, y_pred, class_id, out_png,
max_n=24):
    idxs = [i for i,(yt,yp) in enumerate(zip(y_true,y_pred)) if
yt==class_id and yp!=class_id] #get indices of failed predictions
    random.shuffle(idxs) #shuffle indices for randomn selection of
images
    idxs = idxs[:max_n] #limit to max_n images

    #gallery layout settings
    cols = 6 #number of columns in gallery
    rows = math.ceil(len(idxs)/cols) #number of rows in gallery
    fig, axes = plt.subplots(rows, cols, figsize=(cols*2.2, rows*2.2))
#create subplots
    axes = np.array(axes).reshape(rows, cols) #reshape axes array

    for ax in axes.ravel(): #turn off all axes initially
        ax.axis('off')
    for k, i in enumerate(idxs): #loop through failed indices
        ax = axes.ravel()[k] #get corresponding axis
        img = read_image(paths[i], (256,256)) #read image
        ax.imshow(img) #display image
        ax.set_title(f"true:{['fire','smoke','non-
fire'][y_true[i]]}\npred:{['fire','smoke','non-fire'][y_pred[i]]}",
fontsize=8) #set title with true and predicted labels
        ax.axis('off') #turn off axis

    #configure, save and show gallery plot
    plt.tight_layout()
    Path("./artifacts/galleries").mkdir(parents=True, exist_ok=True)
    plt.savefig(out_png, dpi=200)
    plt.show()

#save failure galleries for each class
save_failure_gallery(test_X_paths, y_true_3, y_pred_3, class_id=0, 
out_png="./artifacts/galleries/fail_fire.png")
save_failure_gallery(test_X_paths, y_true_3, y_pred_3, class_id=1, 
out_png="./artifacts/galleries/fail_smoke.png")
save_failure_gallery(test_X_paths, y_true_3, y_pred_3, class_id=2, 
out_png="./artifacts/galleries/fail_nonfire.png")
```

Deployment:

```
#load in saved artifacts to classify new images with
def load_artifacts():
    s1 = load("./artifacts/models/stage1_scaler.joblib")
    c1 = load("./artifacts/models/stage1_calibrated_svm.joblib")
    s2 = load("./artifacts/models/stage2_scaler.joblib")
    c2 = load("./artifacts/models/stage2_calibrated_svm.joblib")
    with open("./artifacts/reports/stage1_threshold.json") as f:
        thr1 = json.load(f)["threshold"]
    return s1, c1, thr1, s2, c2 #return loaded artifacts

#prediction function for a directory of new images
def predict_dir(img_dir, out_csv="predictions.csv"):
    s1, c1, thr1, s2, c2 = load_artifacts() #load saved artifacts
    img_paths = sorted([str(p) for p in Path(img_dir).glob("**/*") if 
p.suffix.lower() in IMG_EXTS]) #get sorted list of image paths
    records = []
    for p in tqdm(img_paths, desc="Predicting"):
        f = extract_features(p) #extract features
        f1 = s1.transform([f]); prob = c1.predict_proba(f1)[0,1] 
#predict hazard probability
        if prob >= thr1: #if hazard detected, classify as fire or smoke
            f2 = s2.transform([f]); y2 = c2.predict(f2)[0]  #0:fire, 
1:smoke
            label = ["fire","smoke"][y2] #assign label based on 
prediction
        else:
            label = "non-fire" #assign non-fire label
        records.append({"path": p, "label": label, "hazard_prob": 
float(prob)}) #store prediction record

    #save predictions to CSV
    df = pd.DataFrame(records)
    csv_path = Path(out_csv)
    df.to_csv(csv_path, index=False)
    print("Saved:", csv_path.resolve()) #print the path to a csv with 
the predictions
```

```
#example usage:
#predict_dir("/path/to/new_images",
out_csv="./artifacts/reports/two_stage_predictions.csv")
```