

Seeing Through the Chaos: Automated Disaster Assessment from Satellite Imagery

Robin Holzinger

robin.holzinger@berkeley.edu

Keaton Lee

keatonlee@berkeley.edu

Milan Liessens Dujardin

milan.ld@berkeley.edu

University of California, Berkeley

DATA 200 – Graduate Project

December 4, 2025

Presentation video (YouTube)

Abstract

We present a systematic analytical framework for classifying disaster types and assessing damage levels in satellite imagery under severe data scarcity and extreme class imbalance. Our methodology extracts signal from noisy, limited data through three interdependent components: (1) structure-preserving artifact correction (mirroring-based occlusion handling) that outperforms naive imputation and generalizes across model families; (2) exploratory data analysis guiding feature engineering to identify generalizable discriminative features based on color distributions and structural patterns; and (3) task-aware model selection comparing classical and deep learning approaches under realistic constraints. To ensure fair comparison and robust generalization, we develop fully automated training and prediction pipelines that tune hyperparameters via stratified k-fold cross-validation and grid search, evaluate the resulting models on held-out sets, and finally retrain the selected configurations on the full dataset for deployment. For disaster-type classification, color-based features prove nearly linearly separable (logistic regression: 99–100%), with augmented CNNs achieving comparable performance despite severe data sparsity. For damage assessment (200 images, extreme minority imbalance with 6 level-2 samples), classical ensemble methods (LightGBM: 49.43% macro F1) substantially outperform deep learning, suggesting tree-based models more robustly navigate sparse feature regions than neural networks under severe data constraints. We demonstrate that systematic artifact correction, principled feature analysis, and empirically-grounded model selection are essential for reliable classification from noisy, limited satellite data, with direct implications for resource-constrained disaster response.

1 Introduction

Natural disasters inflict severe humanitarian and economic consequences throughout the world, causing economic losses that exceed \$300 billion annually [1]. The frequency and intensity of these events are increasing due to climate change [2], with recent devastating hurricanes and wildfires—including Hurricane Matthew [3] and the Southern California wildfires [4]—exemplifying the growing threat to communities and critical infrastructure. An effective disaster response is critically dependent on the availability of timely and reliable information. However, acquiring such information in fast-moving disaster environments remains a significant operational challenge. Satellite imagery presents a viable means of addressing this challenge: it offers detailed visual cues that can be automatically analyzed to produce rapid, objective damage assessments. Yet, manual interpretation of satellite images remains time-consuming, costly, and prone to human error, making large-scale analysis a slow and resource-intensive process. These limitations motivate the development of automated machine learning systems capable of delivering near real-time assessments.

1.1 Motivation

This work addresses two fundamental challenges in automated disaster assessment from satellite imagery: (1) *disaster type classification*—distinguishing between different disaster types (wildfire versus flooding) from post-event imagery, and (2) *damage level assessment* — quantifying the severity of building damage in hurricane-affected areas on a four-level scale. These tasks differ significantly in their complexity: disaster type classification relies primarily on color and spectral signatures (fire-scorched landscapes versus flooded areas), while damage assessment requires fine-grained structural analysis to distinguish subtle differences between damage severity levels.

Our dataset, derived from the xView2 Challenge Dataset [5], presents several challenges typical of real-world satellite image analysis: limited training data (400 and 200 training images respectively), severe class imbalance (the rarest damage category contains only 6 samples), and image quality issues in the form of black bars likely due to sensor failures or preprocessing artifacts. These constraints make this problem an ideal testbed for evaluating the effectiveness of different machine learning approaches under data scarcity conditions.

1.2 Related Work

Automated damage assessment from satellite imagery has progressed from classical computer vision to modern deep learning. The xView2 Challenge established a benchmark with the xBD dataset of 850,736 building annotations across 19 disaster events [5]. Top-performing teams achieved F1-scores near 0.80 using ensemble deep learning methods, though at the cost of requiring the full large-scale dataset and substantial computational resources. Classical approaches rely on hand-crafted features (color histograms, Haralick descriptors, edge detection) paired with traditional classifiers like SVMs or random forests [6–8]. While interpretable and efficient, these methods struggle to capture the spatial patterns needed for fine-grained damage assessment [9].

CNNs have shown promise for satellite image analysis, though training from scratch requires large datasets to avoid overfitting [10]. Transfer learning—leveraging pre-trained neural networks by reusing their learned feature

representations and training only a new task-specific classifier head—has proven more effective on smaller datasets [11]. However, transfer learning from ImageNet models (trained on natural camera images) to satellite imagery can be limited by the domain gap between these image types [12].

Recent self-supervised vision transformers offer new possibilities. DINOv2, pre-trained on diverse imagery including satellite data, achieves strong performance with minimal fine-tuning [13]. Satellite-specific models like SatMAE (using masked autoencoding on Sentinel-2 time series) and Prithvi (trained on over 1 TB of Landsat-Sentinel-2 data) demonstrate state-of-the-art results but require substantial data and computational resources [14, 15].

For small datasets, data augmentation and class-imbalance mitigation are critical [12]. Yet most prior work relies on thousands of images or more, leaving severely data-constrained regimes underexplored.

1.3 Our Contributions

This work presents several contributions to the field of automated disaster assessment:

1. We compare classical and deep learning methods on a small, imbalanced satellite image dataset. Our evaluation includes logistic regression, gradient boosted trees (LightGBM [16]), CNNs, and transfer learning with DinoV2, offering insights into which approaches are most effective for different tasks.
2. We experiment with techniques for handling extreme class imbalance and data scarcity, including stratified k-fold cross-validation, upsampling, and augmentation.
3. We achieve near-perfect performance on binary disaster type classification, obtaining close to 100% accuracy using simple logistic regression on hand-crafted color features. This result shows that classical methods remain highly competitive when features are strongly discriminative and training data is sparse.

1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 describes our dataset, including its structure, challenges, and preprocessing steps. Section 3 details our methodology, including feature engineering, model architectures, and training procedures. Section 4 presents our results, comparing performance across different models and analyzing per-class performance. Section 5 discusses our findings, limitations, and future directions. Section 6 concludes with a summary of contributions and practical implications.

2 Data

2.1 Dataset Overview

Our analysis is based on a subset of the xView2 Challenge Dataset [5], consisting of 200 high-resolution satellite images ($1024 \times 1024 \times 3$ RGB) divided equally into three disaster-specific subsets: 200 wildfire images (`socal-fire`), 200 flooding images (`midwest-flooding`), and 200 hurricane images (`hurricane-matthew`). These collections support two supervised learning tasks. For disaster type classification, we distinguish wildfire from flooding imagery using the `socal-fire` and `midwest-flooding` subsets. For damage level assessment, we use only the `hurricane-matthew` subset, whose images are annotated with building damage severity on a 4-level scale (0–3).

The dataset exhibits substantial damage level imbalance. Within the damage assessment subset, level 2 (major damage) contains only 6 examples (3%), making it the rarest category. Figure 2, included in the appendix, summarizes the distribution of disaster types and damage levels.

2.2 Image Quality and Black-Bar Artifacts

A notable artifact across the dataset is the presence of large black pixel regions (RGB = (0,0,0)), typically appearing as nearly horizontal or vertical bars along image borders. Approximately 20% of all images contain such bars, and in extreme cases they cover more than 85% of the total image area. While `hurricane-matthew` contains no images with more than 50% black coverage, both wildfire and flooding subsets include heavily occluded samples (Figure 4).

These artifacts distort color and texture distributions and risk introducing spurious predictive cues if not addressed. We evaluated several correction strategies: mean/median imputation (filling black pixels with the mean/median non-black color of the image) and mirror-based filling (reflecting the nearest valid region across the black area). Mean/median imputation produced large uniform regions that skewed spectral features, whereas

mirroring better preserved local structure and avoided artificial color artifacts. We also attempted slicing the images to smaller images to remove the black bars, but we did not find this useful.

Since black bar treatment is generic and can be used for all model types, all images were corrected using the mirroring approach prior to any feature extraction or modeling.

2.3 Exploratory Data Analysis

Our exploratory analysis revealed clear visual differences between disaster types and more subtle distinctions across damage levels. Rendering representative samples across all disaster–damage combinations (Figure 1) showed that wildfire images typically contain warmer tones and higher textural complexity, while flooding scenes exhibit cooler blue–green hues and smoother textures. These qualitative observations guided a more systematic evaluation of color, texture, and structural characteristics.

Color-based statistics proved highly discriminative for the disaster type task. Mean RGB intensities showed that `socal-fire` images have elevated red-channel values, whereas `midwest-flooding` images show stronger blue-channel responses. HSV distributions reinforced these findings: saturation values in particular distinguished fire from flooding (Figure 7). Color-ratio features such as red-to-green and green-to-blue further amplified these separations and later became useful hand-crafted features (Figure 8).

Texture and structural features contributed complementary information. Edge-to-area ratios derived from Sobel and Canny operators showed slightly higher fragmentation in wildfire scenes (Figure 10). Local Binary Pattern metrics (mean, entropy, nonzero counts) captured finer-grained differences in surface texture across disasters (Figure 12). Haralick features, extracted from gray-scaled versions of the images, highlighted subtle variations in texture homogeneity and contrast, though individual features had limited discriminative power (Figures 15–??). Additional structural indicators such as corner counts and connected-component statistics showed modest but consistent differences between wildfire and flooding imagery (Figures 9). Although no single structural feature was strongly predictive on its own, their combination informed a broad texture–structure component of the classical feature set.

2.4 Data Preparation for Modeling

Insights from the EDA guided the construction of our modeling pipeline. All images were first corrected for black-bar artifacts using the mirroring procedure. For classical machine learning methods, we extracted a 227-dimensional feature set comprising color histograms, HSV statistics, color ratios, Haralick features, Local Binary Pattern metrics, edge densities, corner counts, and connected-component summaries. These features reflect the color and texture characteristics that showed distributional differences during EDA.

For CNNs trained from scratch, images were downscaled to 64×64 and normalized to the $[0, 1]$ range. For transfer learning with DINOv2, images were only resized by a few pixels for compatibility with the model in order to preserve fine-grained structural cues important for damage classification.

Given the severe imbalance, we used stratified 5-fold cross-validation for all models and applied balanced upsampling for the damage assessment task during training. Disaster type classification was left unupsampled as classes were balanced already.

Overall, the EDA exposed clear spectral differences between disaster types, subtle structural cues for damage assessment, and significant data-quality and balance issues—directly informing our feature engineering and preprocessing choices for the modeling that follows.

3 Methodology

We systematically compare classical and deep learning approaches for two supervised tasks, disaster type classification and damage level assessment, under realistic constraints of limited data, severe class imbalance, and image artifacts. Our methodology encompasses feature engineering, model selection and implementation, training procedures with hyperparameter optimization, and evaluation strategies. All models are wrapped in a unified training interface that standardizes fitting, prediction, and preprocessing, enabling consistent cross-validation and hyperparameter optimization across model families.

3.1 Feature Engineering

We used pixel-based and structure-based features for damage level and disaster type classification. We selected the features we found to be most promising to distinguish the different classes, as identified through our exploratory

data analysis. Our CNN models automatically embed images into feature space, while classical models use hand-crafted features.

3.1.1 Pixel-based Features

As color-based features we use aggregated statistics of single pixel values. These features do not reflect relations between pixels, but rather the distributions over all pixels within one image. The numbers in brackets in the lists below represent the total numbers of corresponding feature values.

- Color entropy values for each channel (3)
- Color ratios for each pair of channels (3)
- Color moments: mean, std, skewness for each channel (9)
- Color histogram bins for each channel (96)
- Hue, saturation and value (HSV) histogram bins for each channel (96)

3.1.2 Structure-based Features

Structured features reflect the relationships between pixels.

- Haralick features [6] (13)
- Number of nonzero elements after Haralick filtering [6] (1)
- Edge to area ratio after Canny filtering [8] (1)
- Corner count (1)
- Local Binary Pattern features: entropy, mean, std, nonzero count (4)

Note: During EDA, we explored Sobel edge detection [7] and connected component statistics (count, mean, and std), but these were excluded from the final feature set because they caused models to predict only a single class on the holdout set, likely due to distributional differences between training and test data. The final combined representation is 227-dimensional (207 color + 20 structural) for classical damage prediction models.

3.1.3 Learned Representations: DinoV2 Embeddings

For the damage task, we also evaluate learned representations using DinoV2 [13]. First, we scale all image pixel values to the range [0,1] and then normalize per channel using ImageNet’s mean and standard deviation (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]). We resize the images to 1022x1022 for compatibility with the model and feed them through a frozen DinoV2 ViT-S/14 encoder to produce 384-dimensional embeddings per image. In an attempt to increase model performance, we apply nine augmentations on every image in the training data, including rotations, flipping, and color jittering (brightness, contrast, and saturation). We included each augmentation as part of the training data (for both cross validation and the final training) whenever the model was being trained on the original image, i.e. the unaugmented image, using a `TrainableDataset` object. However, the results did not significantly improve performance, which is also suggested by [17]. This is not surprising, as DinoV2 is trained to be insensitive to such transformations.

3.2 Model Selection and Rationale

We tested different models for both tasks, including simple **logistic regression** models, **boosted trees** (i.e., **LightGBM** [16]), and **neural architectures** like **multi-layer-perceptrons** (MLPs) as well as **convolutional neural networks** (inspired by AlexNet [18]).

Logistic regression by nature requires minimal hyperparameter tuning. For the disaster type task, where color features strongly discriminate classes, logistic regression is appropriate as it provides interpretable coefficients and handles linearly separable problems efficiently. For damage assessment, logistic regression serves as a baseline to evaluate whether hand-crafted features capture sufficient information for the more nuanced damage classification task.

Convolutional neural networks (CNNs) enable automatic feature learning through sequences of convolutional filters and pooling layers. However, we note that deep networks typically require more examples to reach their full potential. CNNs are appropriate for tasks requiring spatial pattern recognition, but our small dataset limits their effectiveness.

LightGBM seems well-suited for damage prediction because it can capture nonlinear feature interactions between color and texture features, which may be necessary to distinguish subtle differences between damage levels. Its tree-based structure also handles class imbalance naturally through class weighting.

For the damage task, we also evaluate **transfer learning with DinoV2**, which is appropriate given our small dataset size. Pre-trained vision transformers can leverage knowledge learned from large-scale image datasets, making them good candidates for few-shot learning scenarios like our damage level 2 class with only 6 samples.

3.3 Model Implementation

We built an abstract `ModelTrainer` class providing a unified `sklearn`-inspired API for both Scikit-Learn [19] and PyTorch [20] models, with utilities for state management (resetting parameters, saving/loading checkpoints). This abstraction enables consistent cross-validation and hyperparameter tuning across all model types, including our logistic regression and CNN implementations.

3.3.1 Disaster Prediction Model

For disaster type classification, we trained a logistic regression on all 207 color-based features after scaling with `StandardScaler`. We built a `StandardScalerTransformer` wrapper implementing a `Transformer` interface for state management consistency with model classes.

We classify images from `midwest-flooding` (class 0) and `socal-fire` (class 1). The logistic regression uses tabular features extracted via preprocessing, while a CNN learns directly from raw pixels downsampled to 64×64 . This downscaling risks losing structural information like edges and textures, but larger networks would require more data to avoid overfitting from many trainable parameters across deep layers.

With limited samples, the CNN achieved only 90-95% accuracy. To address data sparsity, we applied augmentations (rotations at different angles, horizontal and vertical flips; see 19), increasing dataset size 9-fold. CNNs do not require explicit standardization.

3.3.2 Damage Prediction Model

Based on our disaster type classification results, where logistic models excelled, we began with logistic regression for the damage prediction task. We briefly experimented with linear regression but quickly returned to logistic models due to their substantially better performance.

Our multiclass logistic regression model classifies `hurricane-matthew` images into four damage levels (0-3). Unlike the disaster prediction model, we trained on both pixel-based (207) and structure-based (20) features for a total of 227 features. All features were scaled using our `StandardScaler` transformer.

We also evaluated LightGBM on the same 227-dimensional feature set. LightGBM's ability to capture nonlinear feature interactions makes it a strong baseline for the more subtle damage prediction problem.

For transfer learning, we train a shallow MLP on standardized 384-dimensional DinoV2 embeddings. The MLP architecture consists of one or two hidden layers (e.g., [128] or [128, 64]) followed by a linear output layer. Balanced upsampling further improves performance on the underrepresented damage level 2 class. We experimented with segmenting the full 1024×1024 into smaller regions that can be used for CNN inference independently, but didn't pursue this further as aggregation of multiple patches turned out challenging given the low <30% accuracy of patch level predictions.

3.4 Training Scheme

To effectively train, regularize, and validate our models, we designed a pipeline based on cross-validation, which we applied for both prediction tasks. We trained our final model on the full dataset that was provided to us, excluding the evaluation set. A graphical representation of the scheme can be found in 20.

3.4.1 Regularization

To find optimal regularization in logistic regression, we train models with different complexity regularization values ($C = 0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 200.0, 500.0, 1000.0, 10^4, 10^5, 10^6$) for both classification tasks. For each C value, we perform 5-fold cross validation and compute accuracy, precision, recall, and F1-score from confusion matrices on validation sets, with precision, recall, and F1-score macro-averaged across classes. This allows us to assess generalization of the model and gives us control over the bias variance tradeoff (more regularization \rightarrow less overfitting). Performance metrics are averaged across folds to assess each C value.

For the binary disaster type task, we additionally sweep probability thresholds between 0.05 and 0.95 in increments of 0.01 to optimize the decision boundary. This confirmed that the standard 0.5 threshold performs near-optimally.

3.4.2 Grid Search and Hyperparameter Tuning

Hyperparameter tuning is performed through systematic grid search across all model families. For each hyperparameter configuration in the grid, we split the training data into 5 stratified folds using `sklearn's StratifiedKFold`, ensuring each fold has equal distributions for all classes. We then train the model on 4 folds and evaluate on the held-out fold, computing performance metrics (accuracy for disaster type, macro F1-score for damage assessment) on each validation fold. The metrics are averaged across all 5 folds to obtain a robust estimate of generalization performance for that configuration. This process is repeated for all configurations in the grid, and we select the configuration achieving the best mean performance across folds.

For logistic regression, we search over regularization strengths C ranging from 0.01 to 10^6 . For LightGBM, we explore combinations of tree depth (6, 8, 10), number of leaves (31, 50, 63), minimum samples per leaf (1, 5, 10), and class-weighting strategies (balanced vs. none). For CNNs, we search over learning rates (10^{-4} , $5 \cdot 10^{-5}$), batch sizes (16, 32, 64), numbers of epochs (8, 12, 16, 32), and channel configurations (e.g., [64, 32] or [64, 64, 32]). For DinoV2+MLP, we evaluate learning rates ($5 \cdot 10^{-5}$, 10^{-4} , $5 \cdot 10^{-4}$), hidden-layer widths ([128], [256], [128, 64], [64, 32]), batch sizes (32, 64, 128), numbers of epochs (32, 48, 64). We also experimented with the following loss function types: `nn.CrossEntropyLoss`, `nn.SoftF1Loss`, and `nn.CrossEntropyLoss` weighted by the original class distribution in order to compensate for the disadvantage of minority classes in the sampling process.

3.4.3 Sampling

Due to class imbalance, we built a `Sampler` class for upsampling or downsampling. It can either match all classes to the smallest class size or sample (with replacement) to reach predefined class sizes. For disaster type classification with logistic regression, we avoid upsampling because logistic regression is sensitive to duplicate samples and the given dataset is already balanced. However, for damage assessment, balanced upsampling consistently improves minority-class performance despite this sensitivity and is therefore applied. For CNNs on the disaster type task, we use balanced upsampling as CNNs are less sensitive to duplicate samples than linear models.

3.4.4 Ensemble Models

In an effort to leverage the inherent stochasticity of model training, we incorporated an ensemble strategy into our pipeline. By averaging predictions from multiple models trained with different random seeds but similar parameters and on the same data, we achieved more robust performance.

3.5 Training and Inference Pipelines

3.5.1 Disaster prediction

Both pipelines (Figure 17) share a similar structure: data loading and preprocessing, then model-specific stages. The logistic regression pipeline standardizes tabular features, optional hyper-parameter tuning, and trains via regularized logistic loss. The CNN pipeline converts inputs to 64×64 images, applies augmentations, and trains using PyTorch. During inference, both load saved transformers and models to generate predictions on holdout data, exporting results as CSV files.

3.5.2 Damage level prediction

The damage-level prediction pipeline mirrors Figure 17, following the same stages (`load_disaster_data`, `preprocess_data`, optional hyper-parameter tuning, training, and artifact persistence) but with different features and labels and an additional upsampling step. Instead of `feature_pipeline_flood_fire_logistic_regression` and `disaster-type labels`, it uses `feature_pipeline_hurricane_logistic_regression` to extract task-specific features and predicts damage severity class. For DinoV2 models, it uses `feature_pipeline_hurricane_dinov2` to extract embeddings for the images after manipulating them for compatibility and consistency with DinoV2's training data and optionally augmenting them. Inference reuses the same artifact-handling and prediction logic, again exporting results as CSV files.

4 Results

4.1 Evaluation Setup

We evaluate all models using stratified 5-fold cross-validation on the labeled training data. We also use an additional test holdout split to not let the hyperparameter search in disaster prediction leak into the accuracy estimate. Our

`k_fold_cross_validation(...)` helper accepts feature–label pairs, a model wrapped in our Trainer interface, and optional Sampler, Transformer, and Augmenter modules for preprocessing and augmentation. For the disaster type task, we use overall accuracy as the primary metric, since the task is binary and classes are relatively balanced. For the damage level task, we use macro-averaged F1-score (unweighted mean across classes) to account for the severe class imbalance, along with macro precision and recall for completeness.

4.2 Disaster Type Classification

Table 1 and Figure 32 summarize wildfire vs. flooding performance. Both logistic regression and the CNN achieve near-perfect accuracy, with logistic regression performing slightly better.

| Model | Accuracy | Macro Precision | Macro Recall | Macro F1 | Evaluation Accuracy |
|---------------------|----------|-----------------|--------------|----------|---------------------|
| Logistic Regression | 99.50 | 99.51 | 99.50 | 99.50 | 100 |
| CNN | 98.75 | 98.79 | 98.75 | 98.75 | 100 |

Table 1: 5-fold cross-validation results for disaster type classification. The evaluation score was computed on the test set provided to us on Pensieve. All values are in %.

These results confirm the patterns identified in our EDA: the 207 color-based features, especially color ratios and HSV distributions, nearly linearly separate wildfire and flooding imagery. Logistic regression exploits this structure effectively, slightly outperforming the CNN (trained with C equal to 0.05), whose additional capacity provides little benefit in this small-data regime.

4.3 Damage Level Assessment

Damage prediction is substantially more challenging, and results vary across model families (Table 2 and Figure 32).

| Model | Accuracy | Macro Precision | Macro Recall | Macro F1 | Evaluation F1 |
|-------------------------------|----------|-----------------|--------------|----------|---------------|
| Logistic Regression (tabular) | 60.00 | 45.16 | 43.66 | 43.66 | 53.61 |
| LightGBM (tabular) | 72.50 | 51.09 | 49.23 | 49.43 | +71 |
| CNN (pixels, 64×64) | 53.00 | 44.91 | 40.63 | 40.00 | 65.62 |
| DINOv2 + MLP (embeddings) | 66.95 | 46.74 | 47.59 | 46.82 | 76.61 |

Table 2: 5-fold cross-validation results for damage level identification. Precision, recall, and F1 are macro-averaged across the four classes. The evaluation score was computed on the test set on Pensieve. All values are in %. DinoV2 + MLP achieved the strongest performance on the evaluation set.

The logistic regression baseline ($\text{macro F1} \approx 42.71\%$) shows that hand-crafted features contain useful but incomplete signal. LightGBM improves to about 49% (71% on the evaluation set) by modeling nonlinear interactions among features. The CNN trained on downsampled pixels reaches only 40.00% macro F1 (65% on the evaluation set), reflecting the difficulty of learning spatial representations from just 200 images. The DINOv2+MLP model achieves performance comparable to LightGBM (around 47% macro F1, but around 77% on the evaluation set), suggesting that while pre-trained embeddings provide rich representations, they do not automatically solve the extreme class-imbalance problem in this small dataset. The different loss functions we implemented (weighted cross entropy loss and soft F1 loss) did not yield better results.

4.4 Difference in Performance Between Methods

The performance differences across models reflect the interaction between dataset size and task complexity and the different inductive biases of the models.

In disaster type classification, color statistics already provide a nearly linearly separable representation. Logistic regression achieves the highest accuracy because a linear decision boundary is sufficient, and complex models add unnecessary variance. CNNs do not provide additional benefit and can overfit in small-data regimes. We thus find classical models to be sufficient for disaster type classification.

In damage level assessment, prediction requires understanding spatial structure (collapsed roofs, debris patterns, texture changes) that global handcrafted statistics cannot fully capture. Logistic regression (43.66% macro F1) is limited by its linear nature. LightGBM (49.43%) captures nonlinear feature interactions and improves meaningfully. CNNs (40.00%) have the right inductive bias but insufficient data to learn strong filters from scratch. Finally, DINOv2+MLP (46.82%) provides a rich pre-trained representation but does not outperform LightGBM, likely due to extreme class imbalance (level 2 has only 6 samples) and insufficient training data to effectively fine-tune the classifier head. The results suggest that in severely data-constrained settings with extreme imbalance, traditional gradient-boosted trees can match or exceed transfer-learning approaches, possibly because they handle sparse regions of feature space more robustly.

5 Discussion

5.1 Evaluation of Approach

Our systematic comparison of classical and deep learning methods reveals clear trade-offs between model complexity and performance. For disaster type classification, logistic regression achieved near-perfect performance (99-100%) with minimal hyperparameter tuning, demonstrating that classical methods remain highly competitive when features strongly separate classes. This validates our EDA-guided feature engineering process.

For damage assessment, the progressive increase in model sophistication, from logistic regression and LightGBM to CNNs and DinoV2+ML, revealed that LightGBM and DinoV2 embeddings achieve the best performance. The comparable results between classical and transfer-learning approaches suggest that in severely data-constrained settings with extreme class imbalance, traditional ensemble methods can match deep learning methods. The unified ModelTrainer abstraction was crucial for enabling fair comparison across model families by standardizing preprocessing, cross-validation, and evaluation procedures.

5.2 Limitations

5.2.1 Dataset Size and Class Imbalance

Our dataset is small by deep learning standards: only 200 images for damage assessment and severe class imbalance. The CNN achieved only 40.00% F1 over cross-validation despite heavy augmentation (9 \times), suggesting that even aggressive augmentation cannot fully compensate for insufficient training data. The extreme imbalance in level 2 limits all models' ability to learn reliable decision boundaries for that class.

5.2.2 Feature Engineering Limitations

The exclusion of Sobel edge detection and connected component statistics from the final feature set highlights the difficulty of feature engineering for distribution-shifted test data. These features showed promise during EDA but caused models to fail on the holdout set (predicting only a single class), likely due to distributional differences between training and test images.

5.2.3 Evaluation Constraints

Our holdout set evaluation is limited by the lack of ground truth labels. Additionally, visual inspection suggests the holdout set may be sorted by disaster type, which could inflate apparent performance. Without true random sampling, our performance estimates may not generalize to new disaster events.

5.3 Discoveries

5.3.1 Transfer Learning Did Not Outperform Classical Methods

We expected DinoV2 embeddings to substantially improve damage assessment performance, but the model achieved only 46.82% F1 in cross-validation, comparable to LightGBM. This counterintuitive result suggests that in extremely small, imbalanced datasets, traditional gradient-boosted trees may be more robust than deep learning approaches, possibly because they handle sparse regions of feature space more effectively.

5.3.2 Classical Methods Competitive for Simple Tasks

The near-perfect performance of logistic regression on disaster type classification (99-100%) exceeded expectations. The strong discriminative power of color features, particularly RGB ratios and HSV distributions, made the problem

nearly linearly separable. This reinforces the value of thorough EDA and feature engineering before moving to complex models.

5.3.3 CNN Underperformance Despite Augmentation

Despite applying $9\times$ data augmentation using geometric transformation, the CNN (40.00% F1) for damage assessment achieved lower performance than LightGBM. This suggests that geometric augmentations (rotations, flips) may not create the diversity needed for effective CNN training on small datasets, and 200 images provide insufficient examples of damage patterns.

5.4 Extensions

5.4.1 Multi-Scale Patch Analysis

Split images into smaller subregions (e.g., 32×32 grids), classify each independently, and aggregate predictions via majority voting or logical OR—this mirrors AlexNet’s augmentation strategy [18], increasing training data while capturing finer-grained local patterns lost in full-image classification.

5.4.2 Building Detection Preprocessing

Currently, we analyze entire images, but damage indicators appear primarily in residential areas. Forests, water, and empty terrain add noise with no visible damage patterns in examined examples. Using object detection (e.g., YOLO) to identify housing regions, then computing features only on those areas, should significantly improve performance by filtering irrelevant pixels. This work could proceed in parallel by researching open-source pretrained residential detection models suited to our data.

5.4.3 Temporal Change Detection

When pre-disaster imagery is available, comparing before and after images could provide more direct damage signals than single-image classification. Computing difference features (e.g., structural changes, color shifts) between temporal pairs might capture damage more reliably.

5.4.4 Further Exploration into Transfer Learning

Leverage additional models pretrained on ImageNet (ResNet, EfficientNet) or satellite datasets (Sentinel-2, Landsat). Models like SatMAE [14] or Prithvi [15, 21] already understand infrastructure and land-use patterns relevant to poverty. We would fine-tune classification heads for our tasks.

5.4.5 Model-based Augmentations

Xu et al. [22] present augmentation strategies based on Generative Adversarial Networks (GANs) to mitigate class imbalance and limited data availability in deep learning. These include ImbCGAN, which applies Conditional GAN to imbalanced classes, and GAN-MBD (Multi-Branch Discriminator), which translates images to match other labels. Specifically for satellite imagery classification, Adedeji et al. [23] suggest that Deep Convolutional GANs (DCGANs) and Wasserstein GANs with Gradient Penalty (WGAN-GP) improve results compared to non-augmentation, especially when combined with geometric augmentations.

5.4.6 Ensemble Methods

Combining predictions from DinoV2+MLP and LightGBM through weighted voting or stacking could potentially leverage complementary strengths: DinoV2’s high-level visual understanding and LightGBM’s nonlinear feature interactions.

5.5 Implications for Practice

For disaster type classification, logistic regression’s over 99% accuracy, computational efficiency, and interpretability make it suitable for real-time deployment.

For damage assessment, LightGBM’s 49.43% macro F1 suggests it is among the best performing methods we tested and offers a practical balance between accuracy and computational efficiency. While 49% F1 is modest, it could still provide value for initial triage and priority mapping when combined with human review. The model could flag potential high-damage areas for immediate inspection while allowing faster coverage of large affected regions.

5.6 Societal Benefits to Automated Disaster Assessment

The ability to identify natural disasters and the level of damage they cause in (near-)real-time is critical for effective and fast natural disaster response. It allows emergency responders to prioritize rescue actions and to understand where help is needed most. Our work extracting critical information from satellite imagery is a step toward carefully and cost-effectively monitoring disaster-prone locations at scale, including the regions which are harder to reach. Especially when resources are scarce, automatic natural disaster assessments can provide governments and stakeholders with a deeper understanding of disaster damage and leads as to how to provide help efficiently.

5.7 Ethical Concerns

Despite the numerous societal benefits associated with disaster monitoring using satellite imagery, there are some ethical concerns too.

First, space monitoring using high-resolution imagery may invade individuals' privacy. Image data may contain sensitive information about individuals and should therefore be handled with care. This highlights the importance of regulatory frameworks that protect against malicious surveillance or any other misuse using satellite images.

Second, the models may be biased, potentially leading to consistently incorrect predictions for specific locations. For example, if the model structurally underestimates the damage level for a particular region, that region might receive structurally less support. Over-reliance on this technology, especially in a safety-critical context like disaster response, can lead to serious harm.

Lastly, we recognize the environmental impact of large-scale monitoring due to the high computational cost associated with training models, extracting features, and classifying satellite images.

6 Conclusion

Our work demonstrates that automated satellite image analysis is feasible for disaster type classification but remains challenging for fine-grained damage assessment in small-data regimes. Traditional methods (LightGBM) provide similar results to classification models based on DINOv2 embeddings for damage level prediction, while simple logistic models based on hand-crafted features suffice for disaster type classification.

A Appendix

A.1 Exploratory Data Analysis

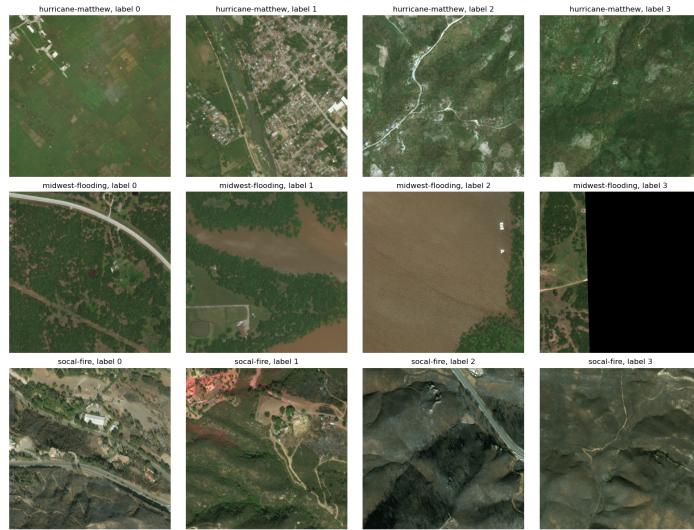


Figure 1: Twelve 1024×1024 satellite image, one for each combination of disaster type and level of damage, illustrating distinct color and texture patterns that differentiate flooding from wildfire and hurricane damage.

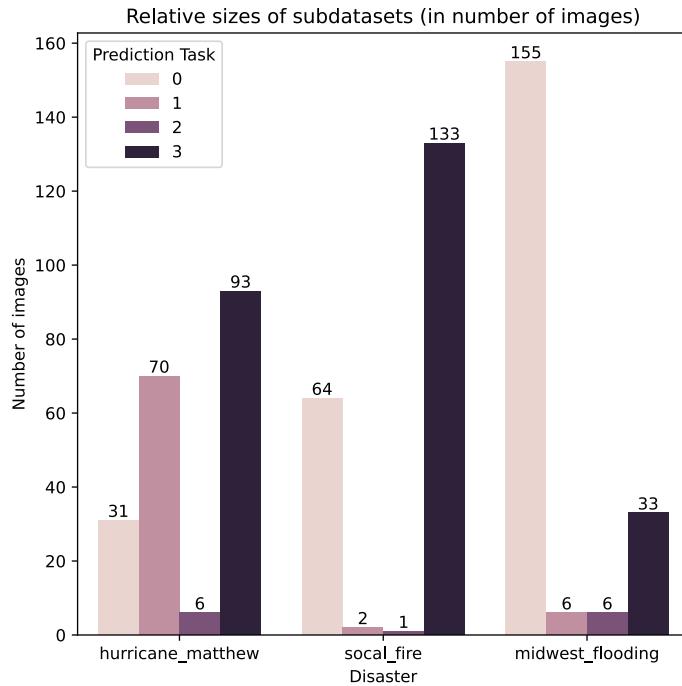


Figure 2: Distribution of image counts across four damage levels (0-3) for three distinct disaster types: hurricane-matthew, socal-fire, and midwest-flooding. The chart reveals significant class imbalance, with Damage Level 3 being most prevalent for hurricane-matthew and socal-fire, while Damage Level 0 dominates midwest-flooding. Damage Levels 1 and 2 are generally underrepresented across all disaster types (except for hurricane-matthew, where a fair amount of Damage 1 is observed).

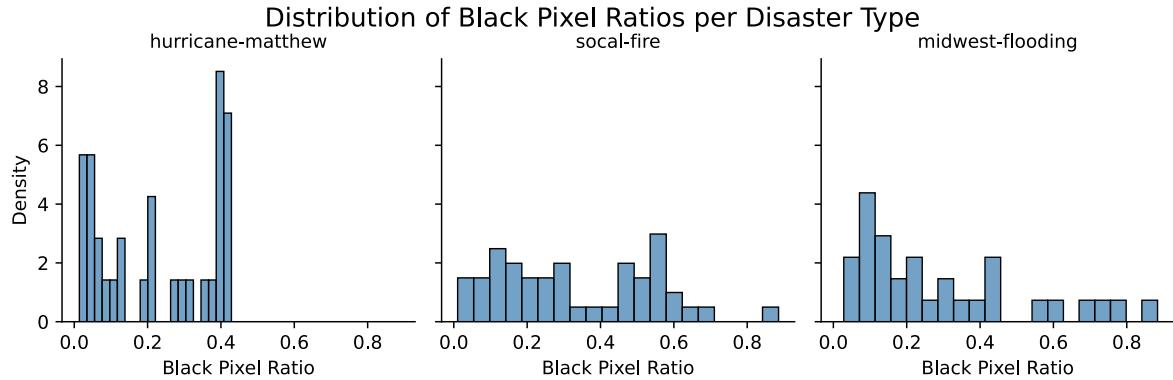


Figure 3: Density distributions illustrating the distribution of black pixel ratios for the three distinct disaster types. The hurricane-matthew distribution is notably bimodal, with prominent peaks around 0.05 and 0.40. The socal-fire distribution is more spread out across the range of black pixel ratios, showing multiple smaller peaks. In contrast, midwest-flooding exhibits a right-skewed distribution, with the majority of black pixel ratios concentrated at lower values, primarily below 0.20.

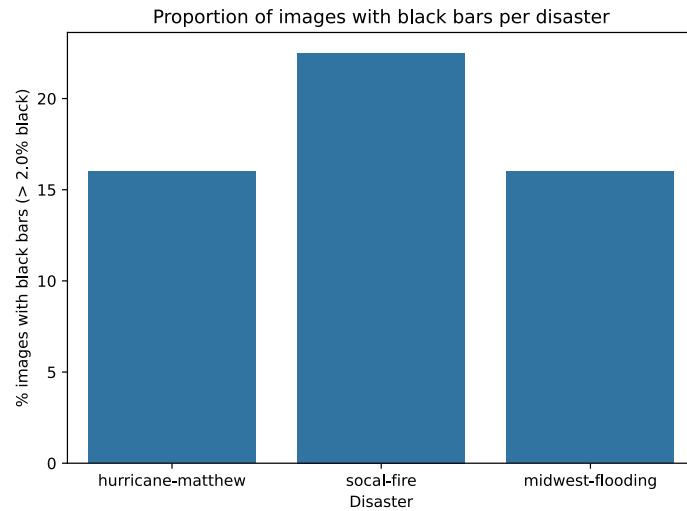


Figure 4: There is a relatively similar proportion (20%) of images for each class having black bars.



Figure 5: An image with a black bar, covering about 50% of the image. In the second image, the black pixels are replaced by the mean color. In the third, half of the image is mirrored onto the other half to fill the black bar.

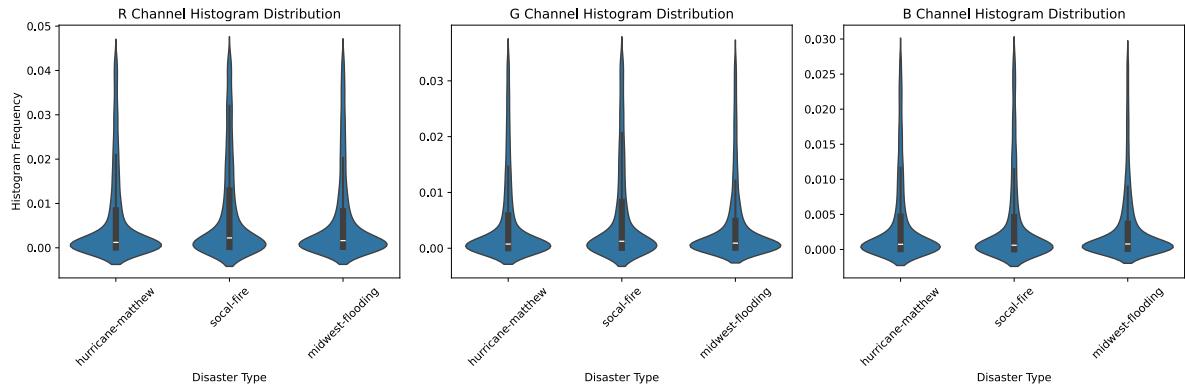


Figure 6: RGB histogram distributions condensed to violin plots (IQR filtered) to better compare color patterns across disaster types. We drop outliers beyond deviating more than 10% from the IQR.

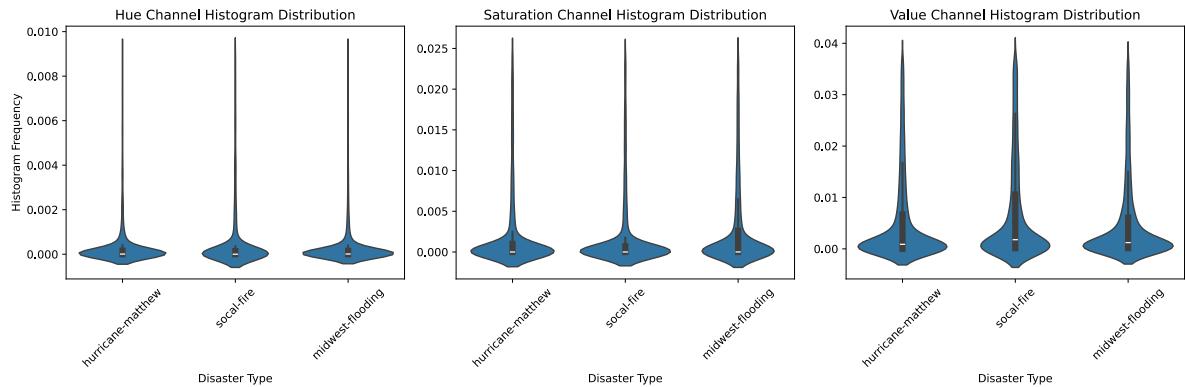


Figure 7: HSV histogram distributions condensed to violin plots (IQR filtered) to better compare color patterns across disaster types. We drop outliers beyond deviating more than 10% from the IQR.

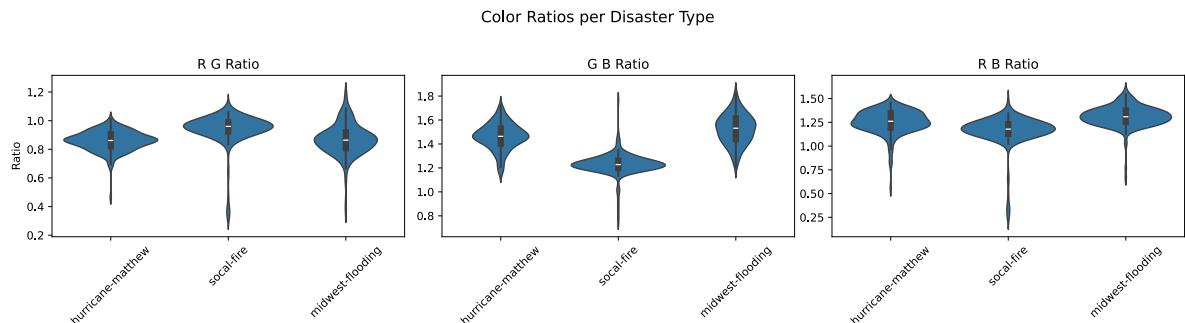


Figure 8: The distributions of the color channel ratios as violin plots, one for each disaster type. Outliers (150% deviation from IQR) were removed.

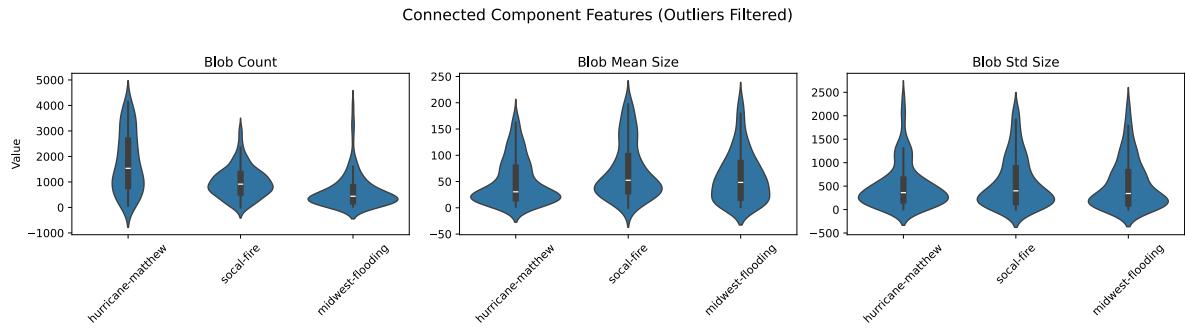


Figure 9: Distribution of connected component statistics (blob count, mean size, and size standard deviation) across disaster types, shown as violin plots with outliers removed to better visualize differences between disaster categories. Outliers (150% deviation from IQR) were removed.

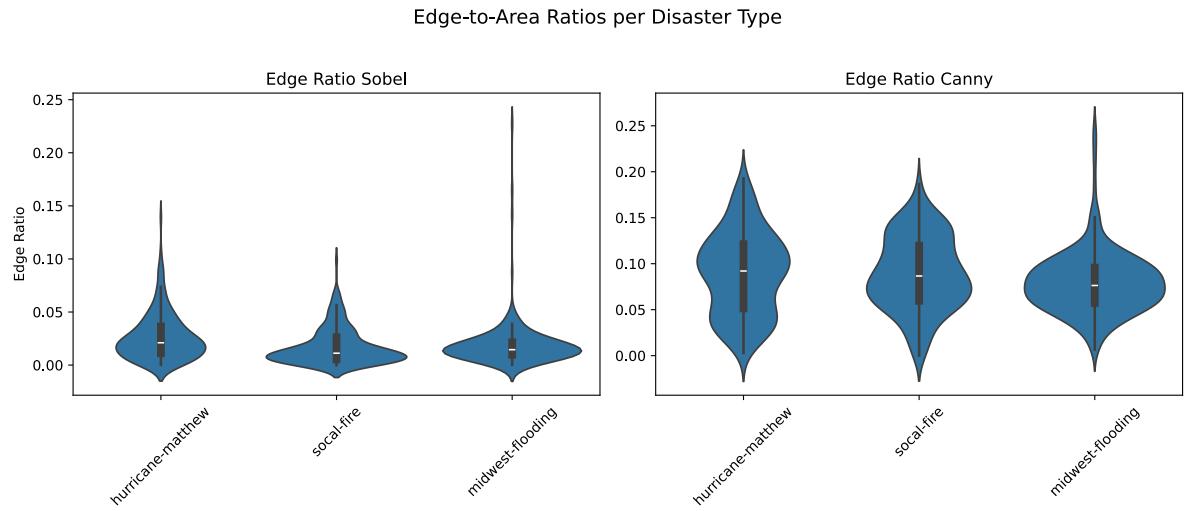


Figure 10: The ratios between the number of edges and the number of non-edges (after edge detection using the Sobel and Canny filters) as violin plots, one for each disaster type. Outliers (150% deviation from IQR) were removed.

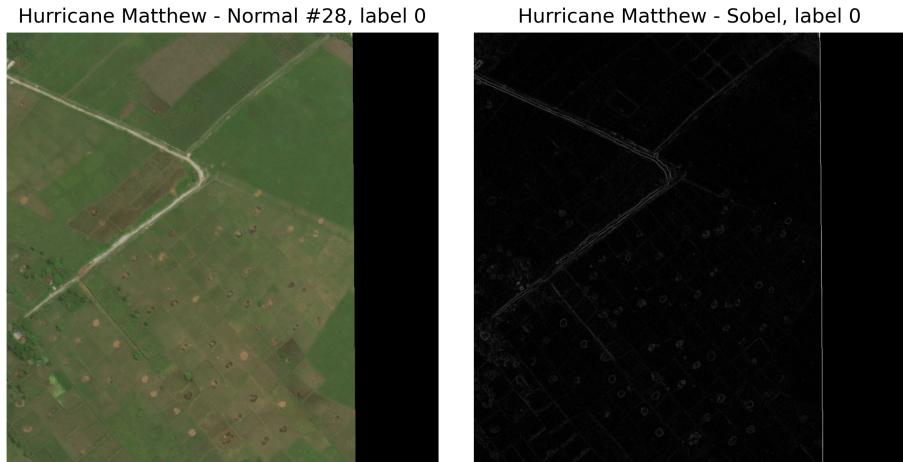


Figure 11: The right image is the left image after transformation using the Sobel filter.

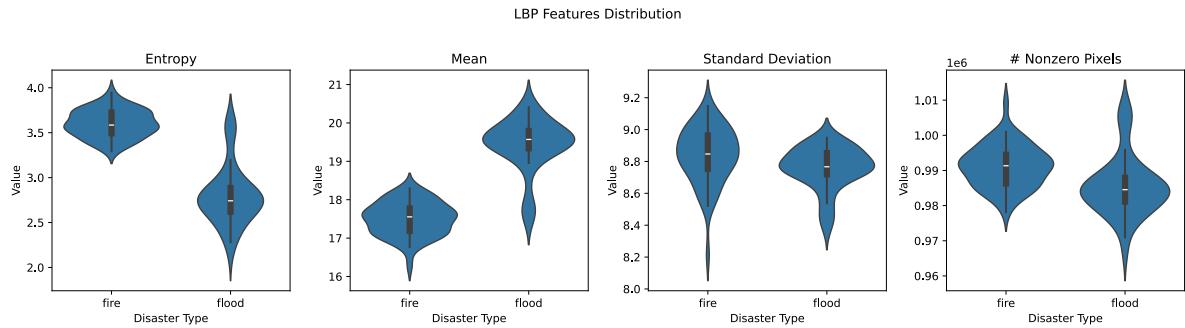


Figure 12: The mean, entropy, and number of nonzero pixels after application of Local Binary Pattern filtering in both classes seem to be indicators for the disaster class.

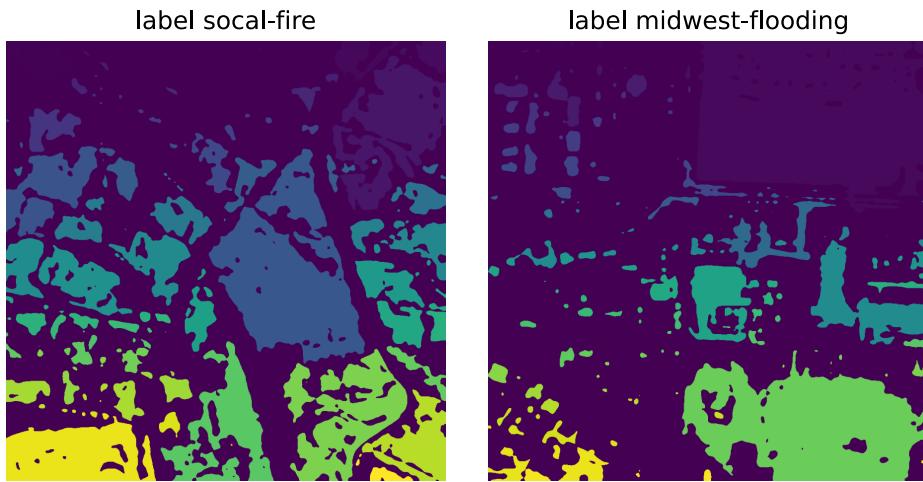


Figure 13: These are two images processed using the Haralick filter. Midwest-floodings seem to have larger, brain-like patterns, motivating us to include the number of non-zero values as an additional structure-based feature to the design matrix.

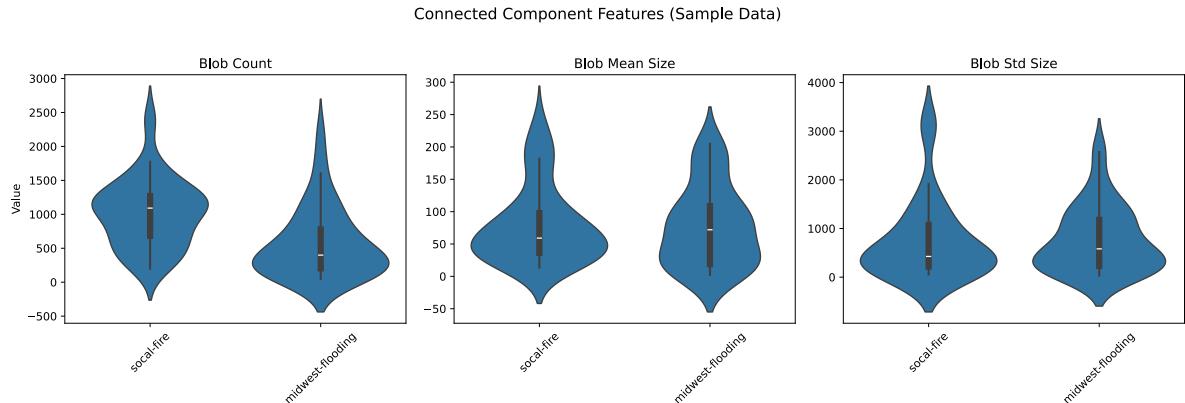


Figure 14: Violin plots showing the distribution of the connected component features (blob count, blob mean size, and blob size standard deviation) over 50 images of each class. Remarkably, there are outliers for each statistic, but specifically for the blob count, all outliers are from the flooding class. Outliers (150% deviation from IQR) were removed.

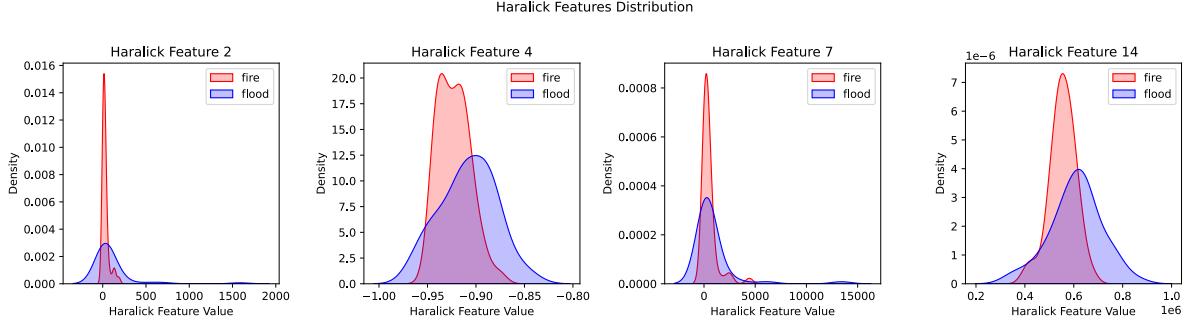


Figure 15: Some of the thirteen Haralick features show potential to help differentiate between fire and flooding images. The distributions show differing local densities in the right tails. The fourteenth feature is derived from the number of non-zero values after application of Haralick filtering.

A.2 Data Preprocessing



Figure 16: Preprocessing result of image with black bar using mirroring to fill black missing regions.

A.3 Training and inference pipeline

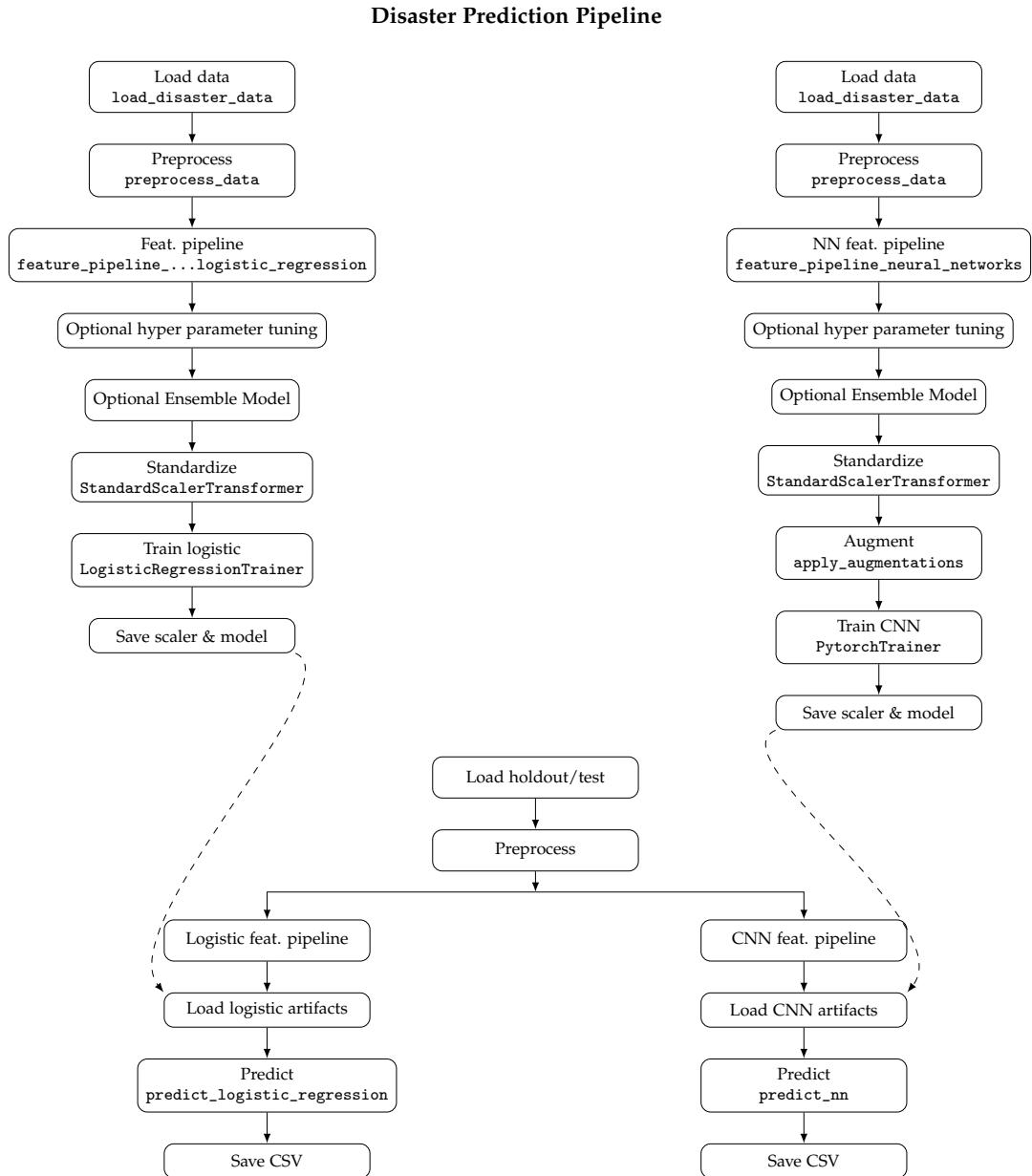


Figure 17: Overview of the disaster prediction training and inference pipeline. Both models follow similar data loading and preprocessing steps but diverge into separate training paths for the Logistic Regression and CNN models. Dashed arrows indicate artifact reuse during prediction.

A.4 Downscaling

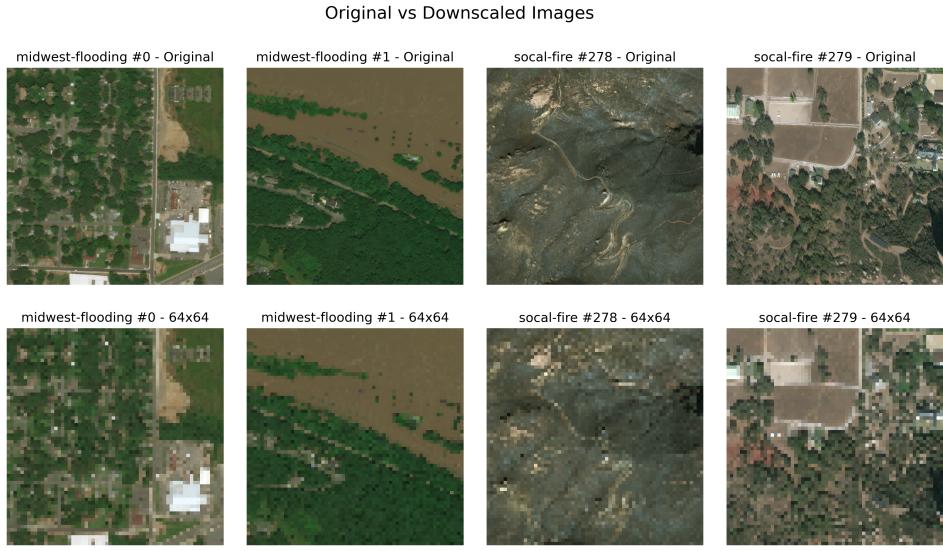


Figure 18: The plot shows 4 original images and their downscaled version used for training of the CNN.

A.5 Augmentation

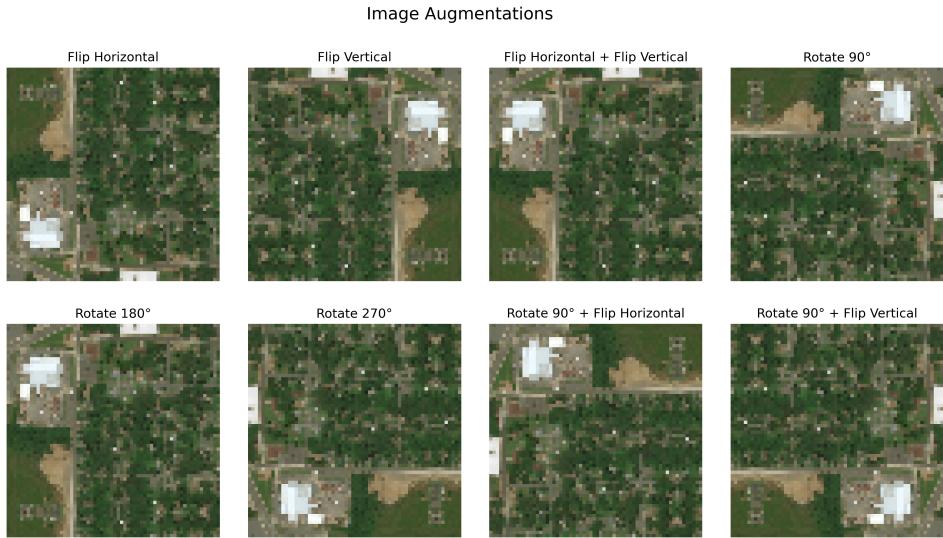


Figure 19: The plot shows eight transformations based on the first image in 18 illustrating the augmentation used to increase the number of training samples for our CNN models.

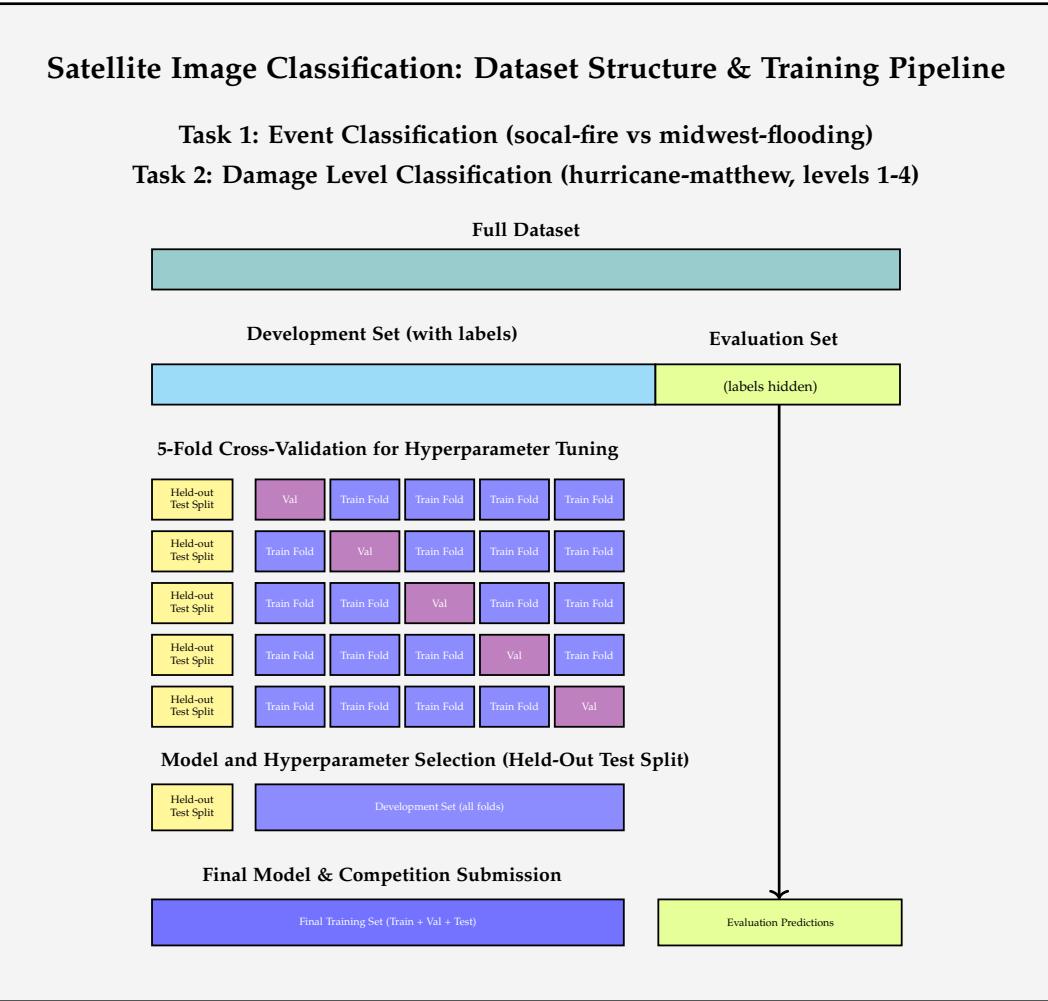


Figure 20: Overall dataset and model training pipeline for satellite image classification. We tune models using 5-fold cross-validation on the development set, with each fold maintaining a validation split. For the damage prediction task we opted against a dedicated test set for the time being given our already limited data (6 samples in minority class) - the samples are more valuable for training. This is reasonable as our best performing models (lightgbm / dinov2 + mlp) work well in the default configuration. The final model is retrained on all available development data before generating predictions on the hidden evaluation set.

A.6 Dataset Structure & Training Pipeline

A.7 Results

A.7.1 Grid Search

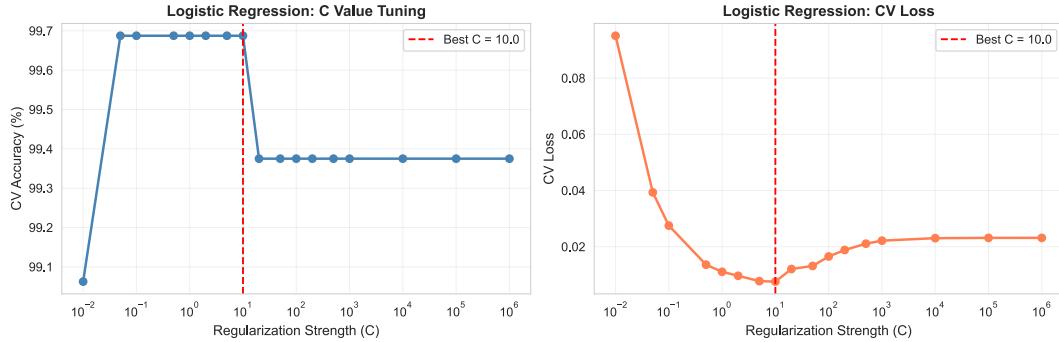


Figure 21: Hyperparameter tuning for Logistic Regression in disaster type classification, showing cross-validation accuracy and loss as functions of regularization strength C . The optimal value achieves the highest accuracy and lowest loss on the validation set.

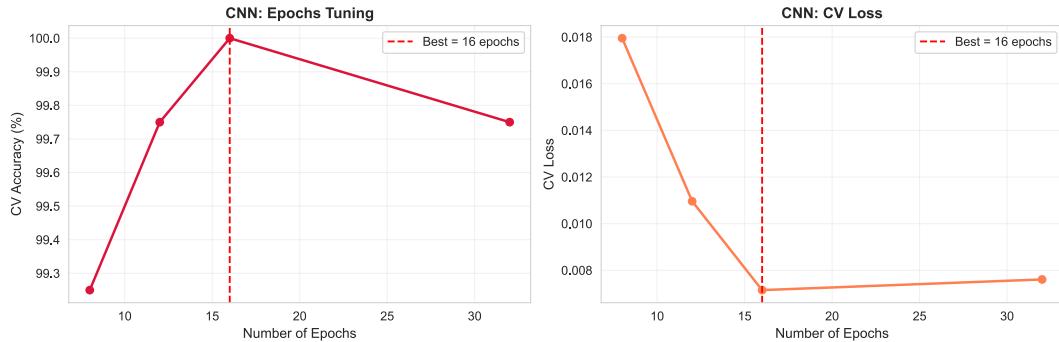


Figure 22: Hyperparameter tuning for CNN in disaster type classification, showing cross-validation accuracy and loss as functions of the number of training epochs. The optimal number of epochs achieves the highest accuracy and lowest loss on the validation set.

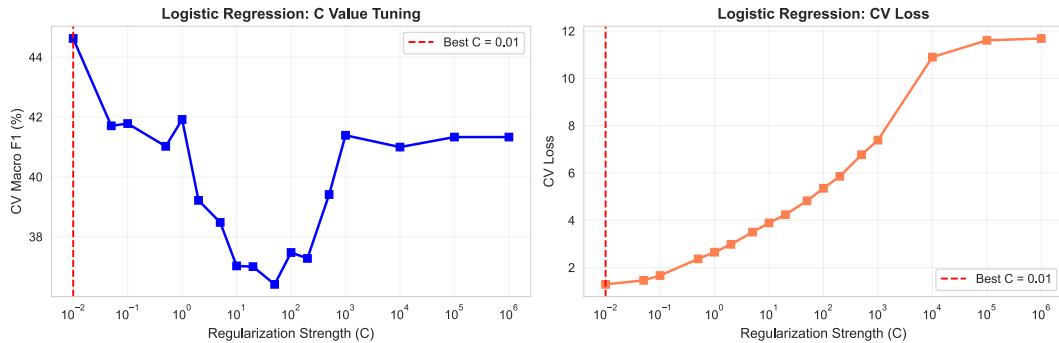


Figure 23: Hyperparameter tuning for Logistic Regression in damage level prediction, showing cross-validation macro F1 score and loss as functions of regularization strength C . The optimal value achieves the highest F1 score and lowest loss on the validation set.

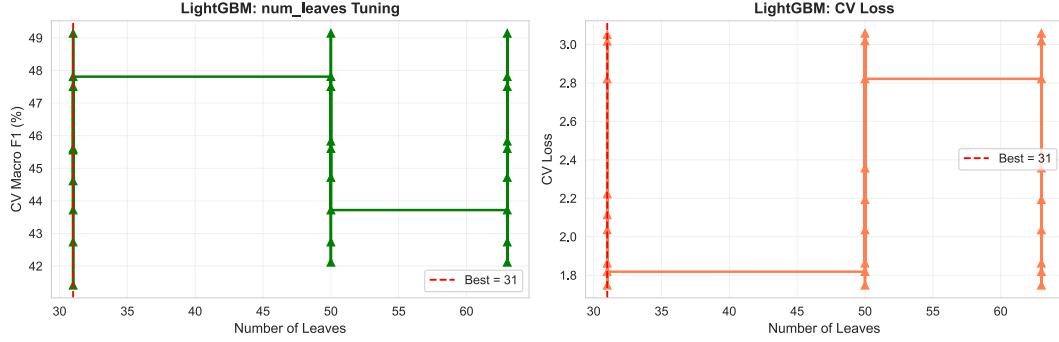


Figure 24: Hyperparameter tuning for LightGBM in damage level prediction, showing cross-validation macro F1 score and loss as functions of the number of leaves. The optimal number of leaves achieves the highest F1 score and lowest loss on the validation set.

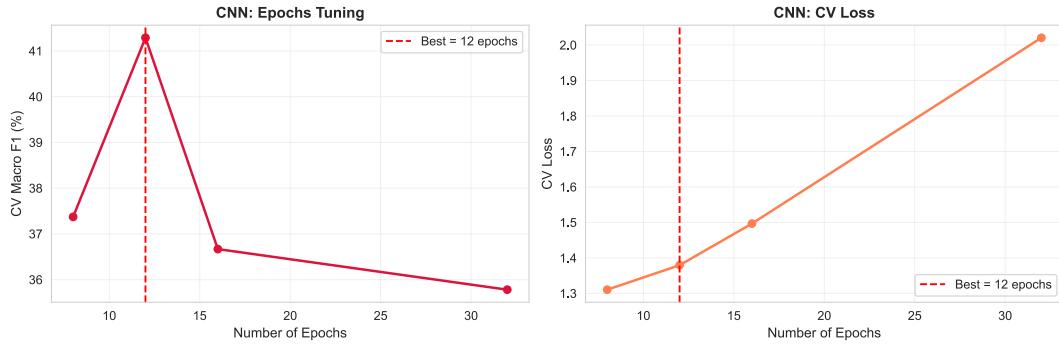


Figure 25: Hyperparameter tuning for CNN in damage level prediction, showing cross-validation macro F1 score and loss as functions of the number of training epochs. The optimal number of epochs achieves the highest F1 score and lowest loss on the validation set.

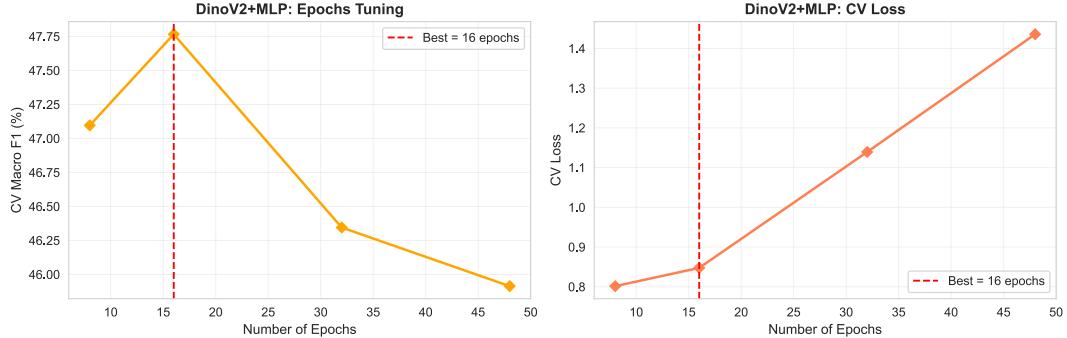


Figure 26: Hyperparameter tuning for DinoV2+MLP in damage level prediction, showing cross-validation macro F1 score and loss as functions of the number of training epochs. The optimal number of epochs achieves the highest F1 score and lowest loss on the validation set.

A.7.2 Disaster Type Classification

Logistic Regression

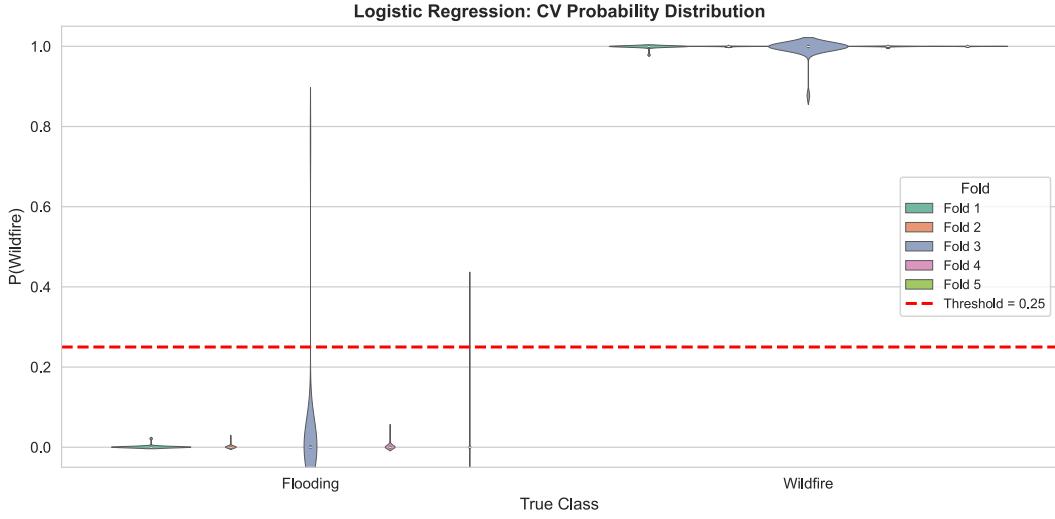


Figure 27: Probability distribution of predicted probabilities by true class label across 5-fold cross-validation for the logistic regression model. Each violin plot shows the distribution for one fold, with colors distinguishing different folds. The red dashed line indicates the decision threshold. Both classes show high predicted probabilities, indicating strong model confidence.

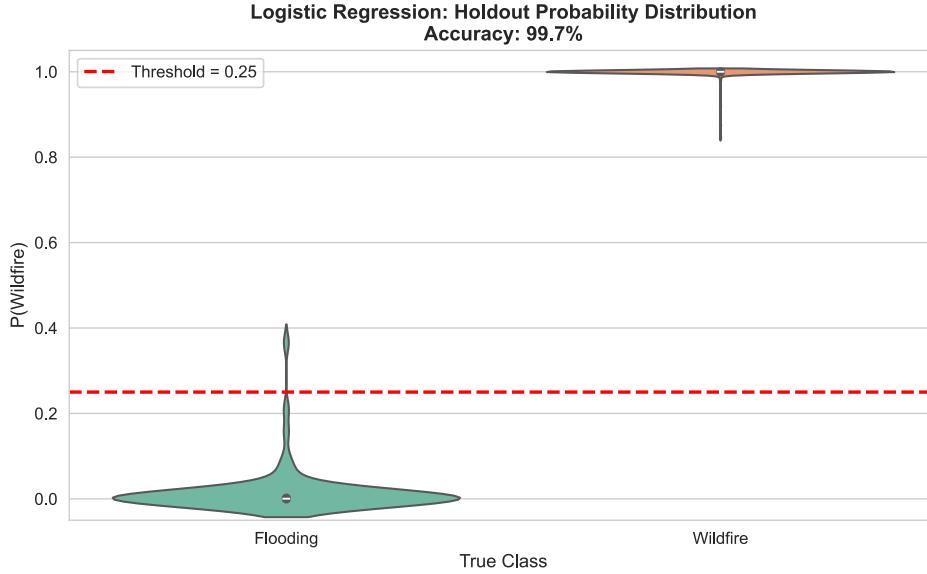


Figure 28: Probability distribution of predicted probabilities by true class label on the holdout set for the logistic regression model. Both classes show concentrated distributions, indicating model confidence. The red dashed line indicates the decision threshold.

| Metric | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) |
|------------------|--------------|---------------|------------|--------------|
| Macro Average | 100.00 | 100.00 | 100.00 | 100.00 |
| Midwest-flooding | 100.00 | 100.00 | 100.00 | 100.00 |
| Socal-fire | 100.00 | 100.00 | 100.00 | 100.00 |

Table 3: Disaster classification scores per class at the best decision threshold (at 77%).

| Metric | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) |
|----------------|--------------|---------------|------------|--------------|
| Macro Average | 85.00 | 55.56 | 54.63 | 54.31 |
| Damage Level 0 | 83.75 | 49.50 | 52.00 | 48.76 |
| Damage Level 1 | 76.88 | 70.89 | 58.79 | 64.03 |
| Damage Level 2 | 94.37 | 20.00 | 20.00 | 20.00 |
| Damage Level 3 | 85.00 | 81.84 | 87.71 | 84.46 |

Table 4: Damage level classification scores per class at the decision threshold 0.7.

| Metric | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) |
|-----------|--------------|---------------|------------|--------------|
| Overall | 98.75 | 98.16 | 99.38 | 98.76 |
| MACRO AVG | 98.75 | 98.77 | 98.75 | 98.75 |
| Class 0 | 98.75 | 99.38 | 98.12 | 98.74 |
| Class 1 | 98.75 | 98.16 | 99.38 | 98.76 |

Table 5: Classification scores after mitigating black bars using color channel pixel value means (Best decision threshold = 0.17).

| Metric | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) |
|-----------|--------------|---------------|------------|--------------|
| Overall | 99.06 | 98.79 | 99.38 | 99.07 |
| MACRO AVG | 99.06 | 99.09 | 99.06 | 99.06 |
| Class 0 | 99.06 | 99.39 | 98.75 | 99.06 |
| Class 1 | 99.06 | 98.79 | 99.38 | 99.07 |

Table 6: Classification scores after mitigating black bars using color channel pixel value medians (Best decision threshold = 0.16).

| Metric | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) |
|-----------|--------------|---------------|------------|--------------|
| Overall | 100 | 100 | 100 | 100 |
| MACRO AVG | 100 | 100 | 100 | 100 |
| Class 0 | 100 | 100 | 100 | 100 |
| Class 1 | 100 | 100 | 100 | 100 |

Table 7: Classification scores using mirroring (Best decision threshold = 0.77).

| Metric | Accuracy (%) | Precision (%) | Recall (%) | F1-score (%) |
|-----------|--------------|---------------|------------|--------------|
| Overall | 99.38 | 99.39 | 99.38 | 99.37 |
| MACRO AVG | 99.38 | 99.39 | 99.38 | 99.37 |
| Class 0 | 99.38 | 99.39 | 99.38 | 99.37 |
| Class 1 | 99.38 | 99.39 | 99.38 | 99.37 |

Table 8: Classification scores without preprocessing (Best decision threshold = 0.46). Due to the class

CNN

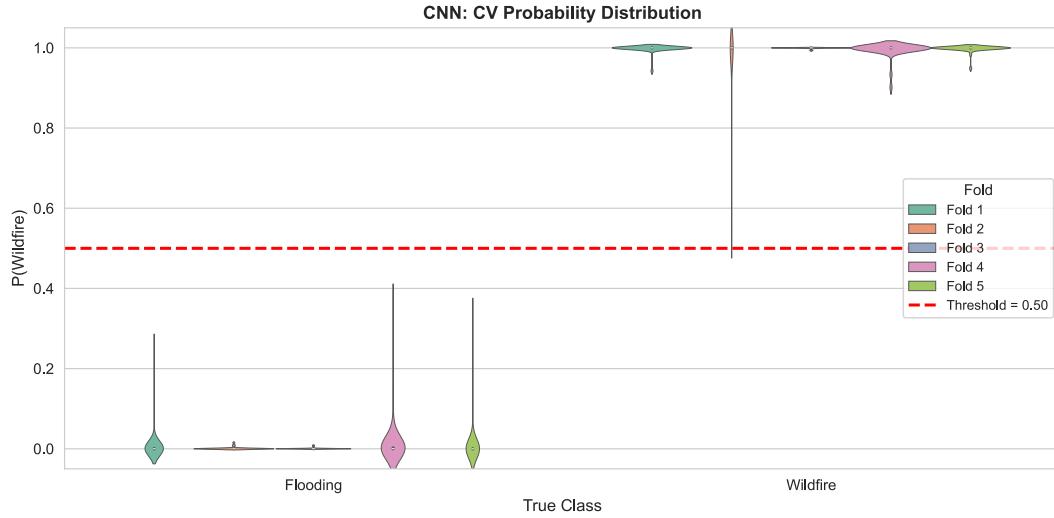


Figure 29: Probability distribution of predicted probabilities by true class label across 5-fold cross-validation for the CNN model. Each violin plot shows the distribution for one fold, with colors distinguishing different folds. The red dashed line indicates the decision threshold at 0.5. Both classes (0: Flooding, 1: Wildfire) show high predicted probabilities near 1.0, indicating strong model confidence.

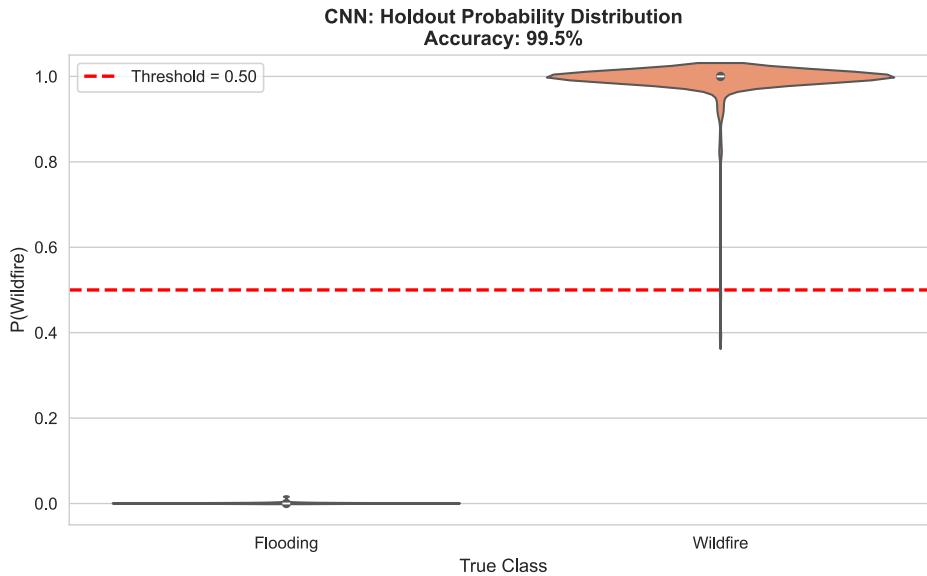


Figure 30: Probability distribution of predicted probabilities by true class label on the holdout set for the CNN model. Both classes show highly concentrated distributions near probability 1.0, indicating strong model confidence. The red dashed line indicates the decision threshold at 0.5.

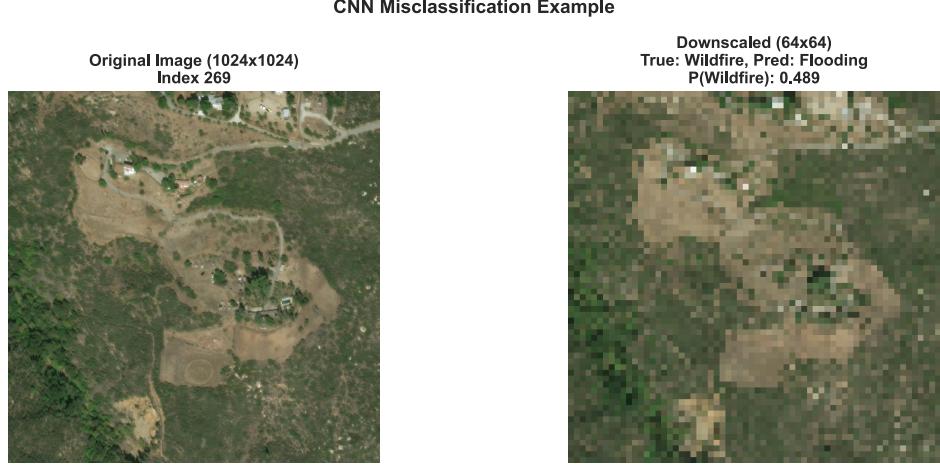


Figure 31: Side-by-side comparison of the original (1024×1024) and downscaled (64×64) images for a sample where the model's prediction differs from the assumed true label. We note that we only assume this is a misclassification, as the holdout set appears to be sorted by disaster type, and we do not have ground truth labels for verification.

A.7.3 Confusion Matrix

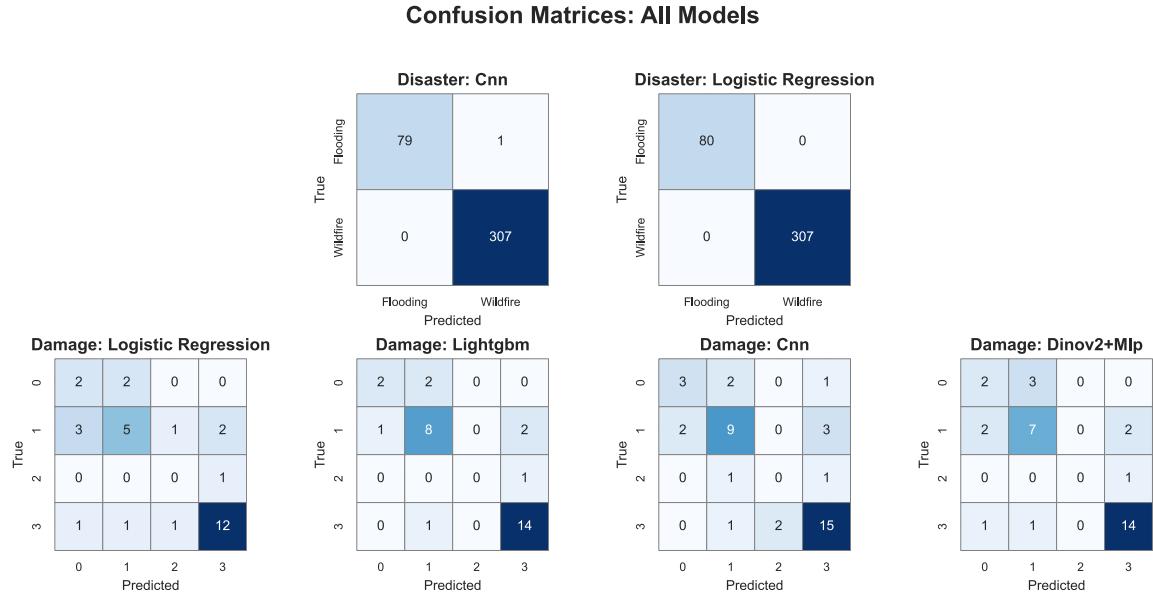


Figure 32: Confusion matrices for all models across disaster type prediction (top row) and damage level prediction (bottom row). For damage prediction models, confusion matrices are aggregated by averaging confusion matrices across all cross-validation folds of the best hyperparameter.

Bibliography

- [1] S. R. Institute, "Shifting sands: Global economic and insurance market outlook 2026-27," Swiss Re Institute, sigma report 5/2025, 2025.
- [2] NOAA, "Wildfire climate connection," 2025.
- [3] N. H. Center, "National hurricane center tropical cyclone report: Hurricane matthew," 2017.
- [4] SCOR, "2017 california wildfire season," 2017.
- [5] A. Gupta et al., "Xbd: A dataset for assessing building damage from satellite imagery," *arXiv preprint arXiv:1911.09296*, 2019.
- [6] R. M. Haralick, K. Shanmugam, and I. Dinstein, "Textural features for image classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, no. 6, pp. 610–621, 1973. doi: [10.1109/TSMC.1973.4309314](https://doi.org/10.1109/TSMC.1973.4309314).
- [7] I. Sobel and G. Feldman, "A 3x3 isotropic gradient operator for image processing," *Stanford Artificial Intelligence Project (SAIL) Technical Report*, 1968, Presented at the Stanford Artificial Intelligence Project (SAIL) 1968.
- [8] J. F. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986. doi: [10.1109/TPAMI.1986.4767851](https://doi.org/10.1109/TPAMI.1986.4767851).
- [9] P. Berezina and D. Liu, "Hurricane damage assessment using coupled convolutional neural networks: A case study of hurricane michael," *Geomatics, Natural Hazards and Risk*, vol. 13, no. 1, pp. 414–431, 2022. doi: [10.1080/19475705.2022.2030414](https://doi.org/10.1080/19475705.2022.2030414). eprint: <https://doi.org/10.1080/19475705.2022.2030414>.
- [10] I. Alisjahbana, J. Li, Ben, Strong, and Y. Zhang, *Deepdamagenet: A two-step deep-learning model for multi-disaster building damage segmentation and classification using satellite imagery*, 2024. arXiv: [2405.04800 \[cs.CV\]](https://arxiv.org/abs/2405.04800).
- [11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. doi: [10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848).
- [12] E. Weber and H. Kané, *Building disaster damage assessment in satellite imagery with multi-temporal fusion*, 2020. arXiv: [2004.05525 \[cs.CV\]](https://arxiv.org/abs/2004.05525).
- [13] M. Oquab, T. Dariset, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski, *Dinov2: Learning robust visual features without supervision*, 2024. arXiv: [2304.07193 \[cs.CV\]](https://arxiv.org/abs/2304.07193).
- [14] Y. Cong, S. Khanna, C. Meng, P. Liu, E. Rozi, Y. He, M. Burke, D. B. Lobell, and S. Ermon, *Satmae: Pre-training transformers for temporal and multi-spectral satellite imagery*, 2023. arXiv: [2207.08051 \[cs.CV\]](https://arxiv.org/abs/2207.08051).
- [15] J. Jakubik, L. Chu, P. Fraccaro, C. Gomes, G. Nyirjesy, R. Bangalore, D. Lambhate, K. Das, D. Oliveira Borges, D. Kimura, N. Simumba, D. Szwarcman, M. Muszynski, K. Weldemariam, B. Zadrozny, R. Ganti, C. Costa, C. Edwards Blair & Watson, K. Mukkavilli, H. Schmude Johannes & Hamann, P. Robert, S. Roy, C. Phillips, K. Ankur, M. Ramasubramanian, I. Gurung, W. J. Leong, R. Avery, R. Ramachandran, M. Maskey, P. Olofossen, E. Fancher, T. Lee, K. Murphy, D. Duffy, M. Little, H. Alemohammad, M. Cecil, S. Li, S. Khallaghi, D. Godwin, M. Ahmadi, F. Kordi, B. Saux, N. Pastick, P. Doucette, R. Fleckenstein, D. Luanga, A. Corvin, and E. Granger, *Prithvi-100M*, Aug. 2023. doi: [10.57967/hf/0952](https://doi.org/10.57967/hf/0952).
- [16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.
- [17] T. Moutakanni, M. Oquab, M. Szafraniec, M. Vakalopoulou, and P. Bojanowski, "You don't need data augmentation in self-supervised learning," *arXiv preprint arXiv:2406.09294*, 2024.
- [18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, Includes the data augmentation strategies (translation, reflection, scaling) that improved CNN generalization, vol. 25, 2012. doi: [10.1145/3065386](https://doi.org/10.1145/3065386).
- [19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

- [20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035.
- [21] J. Jakubik, S. Roy, C. E. Phillips, P. Fraccaro, D. Godwin, B. Zadrozny, D. Szwarcman, C. Gomes, G. Nyirjesy, B. Edwards, D. Kimura, N. Simumba, L. Chu, S. K. Mukkavilli, D. Lambhate, K. Das, R. Bangalore, D. Oliveira, M. Muszynski, K. Ankur, M. Ramasubramanian, I. Gurung, S. Khallaghi, H. Li, M. Cecil, M. Ahmadi, F. Kordi, H. Alemdohammad, M. Maskey, R. Ganti, K. Weldemariam, and R. Ramachandran, "Foundation Models for Generalist Geospatial Artificial Intelligence," *Preprint Available on arxiv:2310.18660*, Oct. 2023.
- [22] M. Xu, S. Yoon, A. Fuentes, and D. S. Park, "A comprehensive survey of image augmentation techniques for deep learning," *Pattern Recognition*, vol. 137, p. 109347, 2023, issn: 0031-3203. doi: [10.1016/j.patcog.2023.109347](https://doi.org/10.1016/j.patcog.2023.109347).
- [23] O. Adedeji, P. Owoade, O. Ajayi, and O. Arowolo, "Image augmentation for satellite images," *arXiv preprint arXiv:2207.14580*, 2022.

Exploratory Data Analysis

This notebook explores the satellite imagery dataset for disaster classification and damage assessment.

Contents:

- 0. Setup & Data Loading
- 1. Dataset Overview
- 2. Missing Data Analysis (Black Bars)
- 3. Color-based Features
- 4. Texture & Edge Features
- 5. Structural Features

0. Setup & Data Loading

Import required libraries and load the satellite imagery dataset.

```
In [1]: # Setup paths for notebook to run from analysis/ folder
import sys
from pathlib import Path

ROOT = Path("..").resolve()
sys.path.insert(0, str(ROOT))

# Standard imports
import json
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import plotly.express as px
from skimage.transform import integral_image
from skimage.feature import haar_like_feature

# Project imports - Constants
from data200.const.const import DATA_DIR, FIGURES_DIR, DISASTER_LIST
from data200.const.schema import DatasetSample
from data200.utils.data import get_images, get_labels

# Project imports - Data loading and preprocessing
from data200.io.load import load_disaster_data
from data200.pipeline.preprocess import preprocess_data
from data200.preprocess.patch import fill_black_bars, fill_black_bars_by_mirror, add_alpha_from_black_bars

# Project imports - Plotting utilities
from data200.plot.image import render_image, render_images, render_image_samples, render_image_interactive
from data200.plot.color import (
    plot_rgb_decomposition,
    render_multiple_black_bar_comparisons,
    plot_black_ratios,
    plot_color_intensity_hist,
    plot_black_ratio_distribute_per_disaster,
    plot_image_region,
    plot_feature_violin_by_disaster,
    plot_histogram_violin_by_disaster,
)

# Project imports - Feature extraction
from data200.utils.feature import (
    get_sobel_features,
    get_gabor_features,
    generate_gabor_kernel,
    get_local_binary_pattern,
    extract_features_by_disaster,
)
from data200.feature.black_pixels import (
    patch_dataset,
    black_bar_ratio_per_disaster,
    find_black_ratios,
    has_black_bars,
    gen_black_ratios_df,
)
from data200.feature.color_based_features import (
    get_color_moments,
    get_color_histogram,
    get_color_entropy,
    get_hsv_histogram,
    get_color_ratios,
)
from data200.feature.structured_features import (
    get_edge_to_area_ratio_sobel,
```

```

        get_edge_to_area_ratio_canny,
        get_connected_component_stats,
        get_corner_count,
    )
from data200.feature.texture_based_features import get_wavelet_features, get_haralick_features, get_haar_features
from data200.sample.sample import interleave

# Ensure output directories exist
(FIGURES_DIR / "eda").mkdir(parents=True, exist_ok=True)

print(f"Data directory: {DATA_DIR}")
print(f"Figures directory: {FIGURES_DIR}")

%load_ext autoreload
%autoreload 2

```

Data directory: /Users/robinholzi/git/ucb/data200/data200-grad-project/data/satellite-image-data-new
 Figures directory: /Users/robinholzi/git/ucb/data200/data200-grad-project/figures

0.1 Load Dataset

Load the satellite imagery dataset containing three disaster types: Hurricane Matthew, SoCal Fire, and Midwest Flooding.

```
In [2]: data = load_disaster_data(path=DATA_DIR)

Loading train images and labels for hurricane-matthew dataset...
Loading train images and labels for socal-fire dataset...
Loading train images and labels for midwest-flooding dataset...
```

0.2 Verify Image Dimensions

Confirm all images have consistent dimensions (1024×1024×3).

```
In [3]: print(f"image shape: {data.hurricane_matthew.images[0].shape}")

wrong = 0
for image in data.hurricane_matthew.images:
    if image.shape != (1024,1024,3):
        wrong += 1
for image in data.socal_fire.images:
    if image.shape != (1024,1024,3):
        wrong += 1
for image in data.midwest_flooding.images:
    if image.shape != (1024,1024,3):
        wrong += 1
print(wrong)

image shape: (1024, 1024, 3)
0
```

0.3 Sample Image Visualization

Visualize sample images from each disaster type to understand the data.

```
In [4]: print(f"image shape: {data.hurricane_matthew.images[0].shape}, dtype", f"{data.hurricane_matthew.images[0].dtype}")

image shape: (1024, 1024, 3), dtype uint8

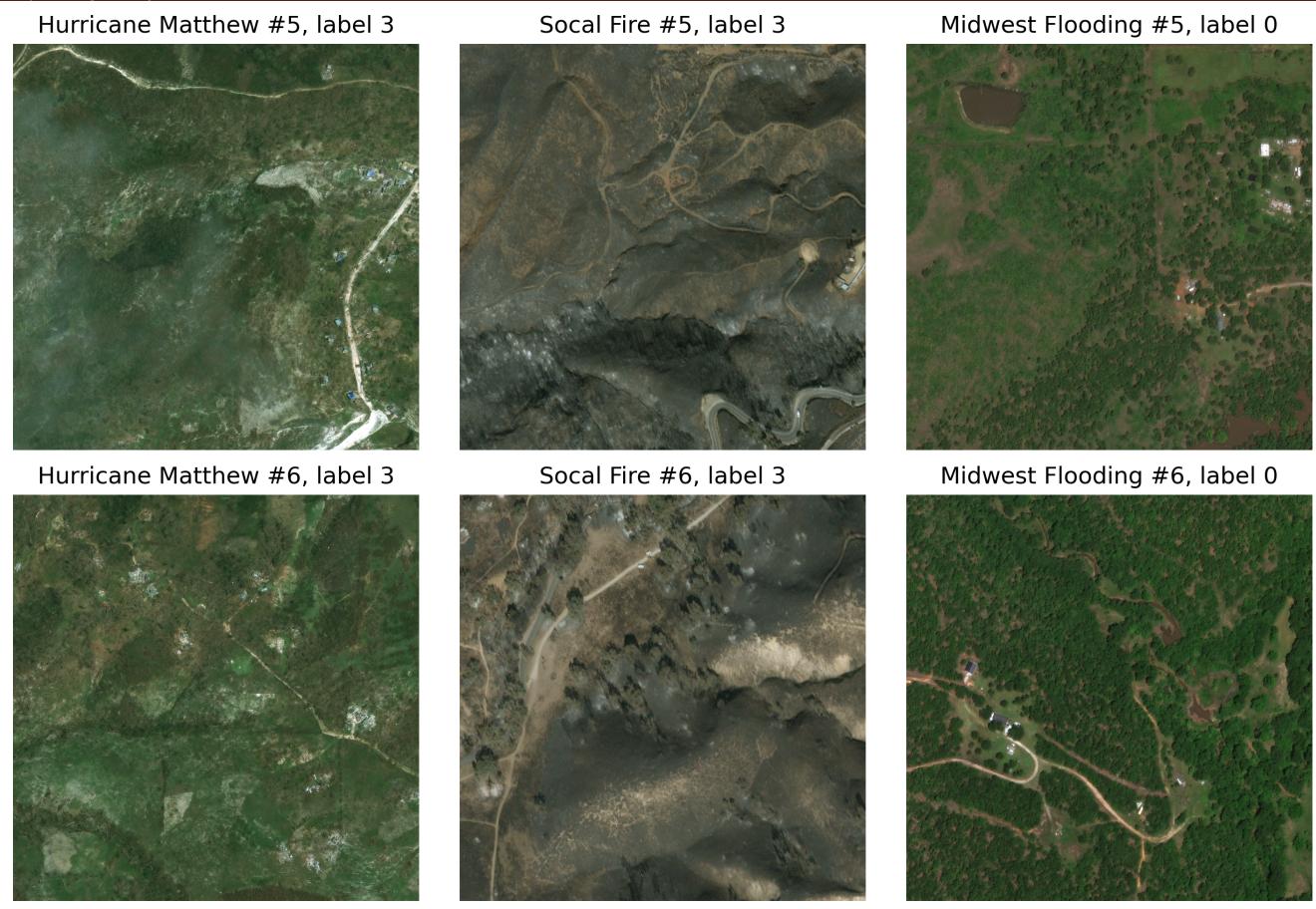
In [5]: long = False
if long:
    num_images_per_disaster = 100
    start = 0
else:
    num_images_per_disaster = 2
    start = 5

sample_indexes = {
    "hurricane-matthew": list(range(start, start+num_images_per_disaster)),
    "midwest-flooding": list(range(start, start+num_images_per_disaster)),
    "socal-fire": list(range(start, start+num_images_per_disaster)),
}
assert all(
    len(sample_indexes[disaster]) == num_images_per_disaster
    for disaster in DISASTER_LIST
)

# interleave disasters round-robin
samples = [
    data.get(disaster).get_sample(sample_indexes[disaster][idx], dataset=disaster.replace("-", " ").title())
    for idx in range(num_images_per_disaster)
    for disaster in DISASTER_LIST
]

render_image_samples(samples).savefig(FIGURES_DIR / "eda" / "disaster_samples_latex.pdf", dpi=300)
```

```
/Users/robinholzr/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout(), so results might be incorrect.  
plt.tight_layout()
```



1. Dataset Overview

Explore the basic statistics, class distributions, and characteristics of the satellite imagery dataset.

1.1 Basic Statistics

Summary of dataset dimensions and image properties.

```
In [6]: structure = data.structure()  
overview = data.get_overview()  
  
# All images are 1024x1024x3  
structure
```

```
Out[6]:
```

| | disaster | prediction_task | num_images | width | height | channels | pixels | values |
|---|-------------------|-----------------|------------|-------|--------|----------|---------|---------|
| 0 | hurricane-matthew | damage level | 200 | 1024 | 1024 | 3 | 1048576 | 3145728 |
| 1 | socal-fire | disaster type | 200 | 1024 | 1024 | 3 | 1048576 | 3145728 |
| 2 | midwest-flooding | disaster type | 200 | 1024 | 1024 | 3 | 1048576 | 3145728 |

```
In [7]: (  
    structure["width"].unique(),  
    structure["height"].unique(),  
    structure["channels"].unique()  
)
```

```
Out[7]: (array([1024]), array([1024]), array([3]))
```

1.2 Images per Disaster Type

Compare the number of images available for each disaster type.

```
In [8]: plt.figure(figsize=(6, 6))
```

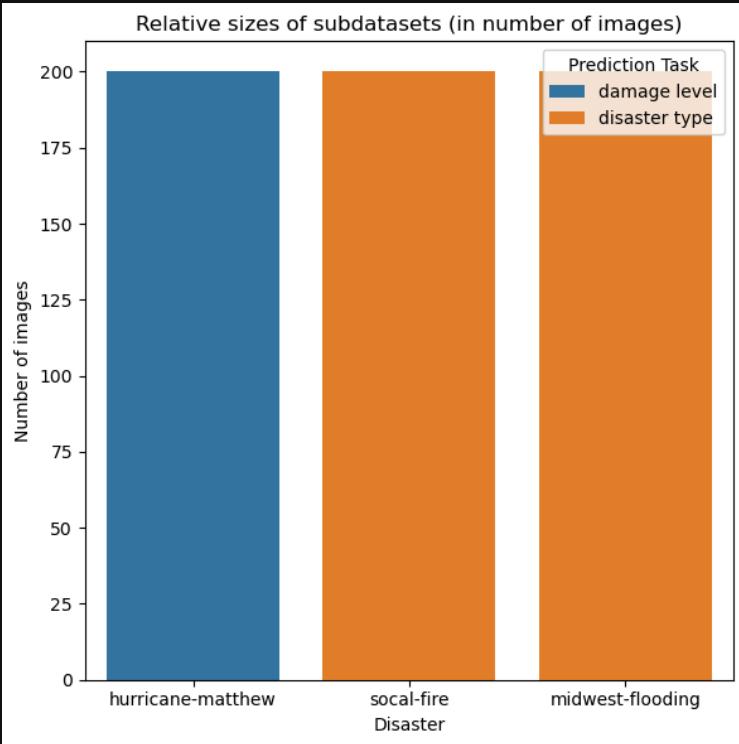
```

# Relative sizes of subdatasets
sns.barplot(
    structure,
    x="disaster",
    y="num_images",
    hue="prediction_task",
)

plt.title("Relative sizes of subdatasets (in number of images)")
plt.xlabel("Disaster")
plt.ylabel("Number of images")
plt.tight_layout()
plt.legend(title='Prediction Task')

# Save as png
plt.savefig(
    FIGURES_DIR / "eda" / "relative_sizes_subdatasets.pdf",
    dpi=300
)

```



1.3 Damage Label Distribution

Visualize the distribution of damage levels across disaster types.

```

In [9]: plt.figure(figsize=(6, 6))

ax = sns.countplot(
    overview,
    x="disaster_label",
    hue="damage_label",
)

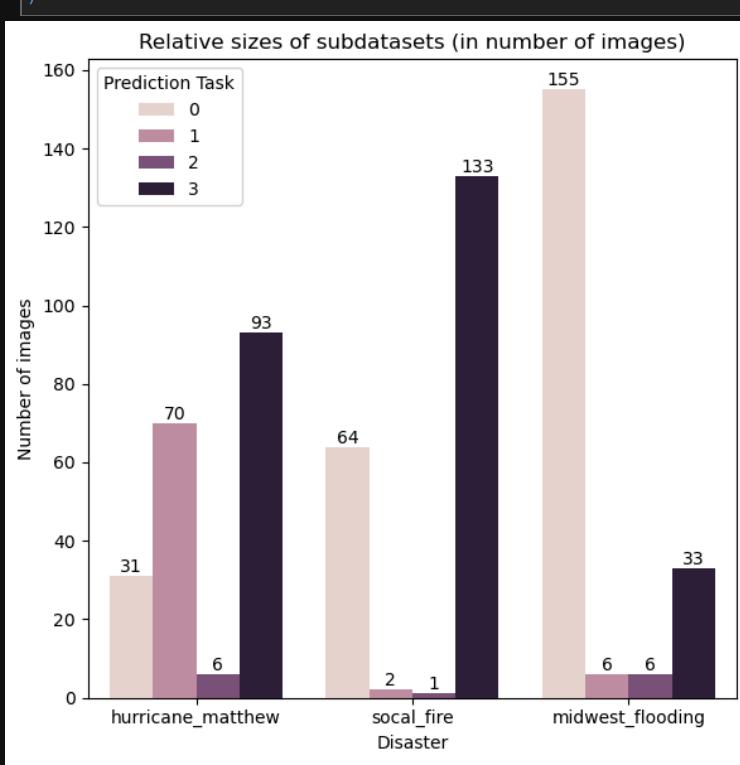
for container in ax.containers:
    ax.bar_label(container)

plt.title("Relative sizes of subdatasets (in number of images)")
plt.xlabel("Disaster")
plt.ylabel("Number of images")
plt.tight_layout()
plt.legend(title='Damage Level')

plt.title("Relative sizes of subdatasets (in number of images)")
plt.xlabel("Disaster")
plt.ylabel("Number of images")
plt.tight_layout()
plt.legend(title='Prediction Task')

# Save as png
plt.savefig(
    FIGURES_DIR / "eda" / "label_distribution_per_subdataset.pdf",
    dpi=300
)

```



1.4 Dataset Size Considerations

The datasets are quite small (200 samples per disaster type), which presents challenges for deep learning.

Potential solutions:

1. **Data augmentation:** Random crops, rotations, color jittering, flips, noise addition
2. **Transfer learning:** Use pre-trained models (not pursued to avoid external dependencies)
3. **Classical ML models:** Use hand-crafted features with SVM, Boosted Trees, Random Forests as baselines

2. Missing Data Analysis (Black Bars)

Analyze and handle missing data represented as black bars in satellite images.

2.1 The Problem

Many images have black bars (missing pixels) on the sides, likely due to sensor failure or preprocessing (cloud removal). Different disaster types have different black bar ratios.

2.2 Handling Options

| Option | Description | Decision |
|-----------------------|------------------------------------------|-------------------------------------------------|
| Drop images | Remove images with black bars | Rejected: Would lose 15-20% of data |
| Fill with mean | Replace black pixels with image mean | Considered: Simple baseline |
| Mirror fill | Reflect image content into black regions | Chosen: Reflects image content naturally |
| Mask channel | Add 4th channel indicating valid pixels | Rejected: More complex implementation |

2.3 Feature Extraction Strategy

After patching, we extract the following classical features:

Color-based: Color moments, histograms, entropy, HSV, color ratios

Texture-based: Gabor filters, LBP, Sobel edges, Haralick, wavelets

Structural: Edge-to-area ratio, connected components, corner detection

2.4 Black Bar Detection

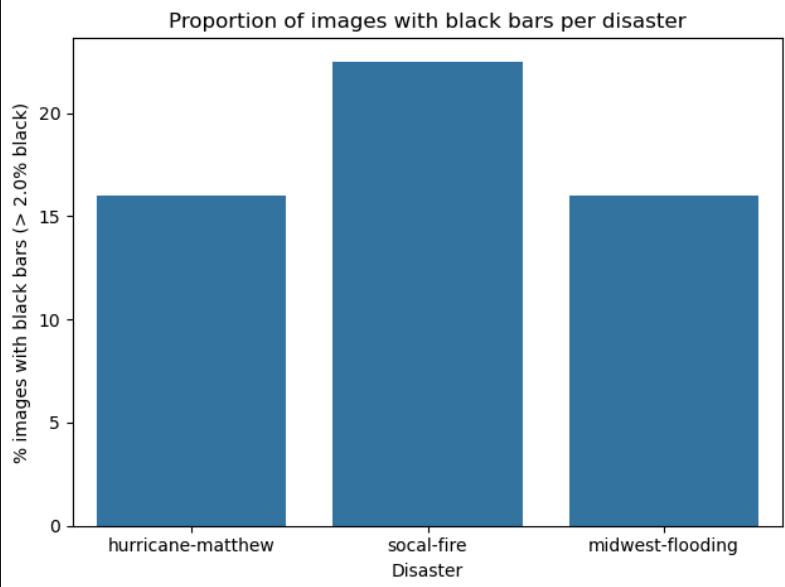
Analyze the proportion of images with black bars per disaster type.

```
In [10]: black_bar_ratios_per_disaster = {
    disaster: find_black_ratios(data.get(disaster).images)
    for disaster in DISASTER_LIST
}
black_ratios_df = gen_black_ratios_df(black_bar_ratios_per_disaster)
black_ratios_df.head(2)
```

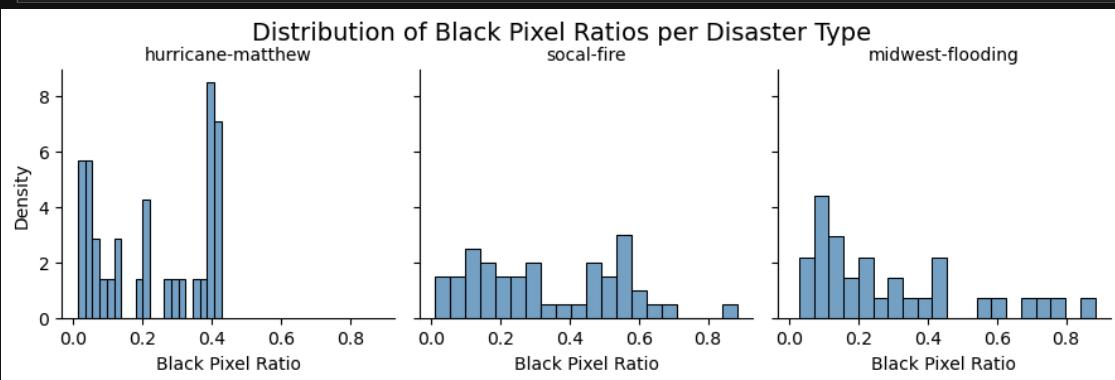
```
Out[10]:      disaster  sample_idx  black_ratio
0  hurricane-matthew        0        0.0
1  hurricane-matthew        1        0.0
```

```
In [11]: plot_black_ratios(black_bar_ratios_per_disaster)
print(
    "Example black ratio sample 0:",
    find_black_ratios(data.hurricane_matthew.images)[0],
)
print(
    "Example black ratio sample 2:",
    find_black_ratios(data.hurricane_matthew.images)[2],
)
```

Example black ratio sample 0: 0.0
Example black ratio sample 2: 0.0



```
In [12]: plot_black_ratio_distribute_per_disaster(black_ratios_df).savefig(
    FIGURES_DIR / "eda" / "black_ratio_distribution_per_disaster.pdf",
    dpi=300
)
```



```
In [13]: black_bar_samples = np.array([
    data.socal_fire.images[i]
    for i, ratio in enumerate(black_bar_ratios_per_disaster["socal-fire"])
    if ratio >= 0.1
])
```

```
In [14]: offset = 28
num_samples = 1
```

```

subset = black_bar_samples[offset:offset+num_samples]
filled_mean = fill_black_bars(
    subset, method="mean"
)
filled_mirror = fill_black_bars(
    subset, method="mirror"
)
render_multiple_black_bar_comparisons(
    subset,
    filled_mean,
    filled_mirror,
    offset=0,
    num_samples=num_samples
).savefig(
    FIGURES_DIR / "eda" / "black_bar_filling_comparison.pdf",
    dpi=300,
    bbox_inches="tight"
)
render_multiple_black_bar_comparisons(
    subset,
    None,
    filled_mirror,
    offset=0,
    num_samples=num_samples
).savefig(
    FIGURES_DIR / "eda" / "black_bar_filling_comparison_nemean.pdf",
    dpi=300,
    bbox_inches="tight"
)

```

/Users/robinholzi/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
`plt.tight_layout()`

Black Bar Filling Comparisons

Original



Filled (mean)



Filled (mirror)



/Users/robinholzi/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
`plt.tight_layout()`

Black Bar Filling Comparisons

Original Filled (mirror)



```
In [15]: # Use an image to demonstrate the alpha channel effect
sample_with_black_bars = black_bar_samples[28]

image_with_black_bar_alpha = add_alpha_from_black_bars(
    np.array([sample_with_black_bars]))
)

fig = render_images([
    sample_with_black_bars,
    image_with_black_bar_alpha[0],
], labels=["Original", "With Alpha Channel"], ncols=2)
plt.show()
```

/Users/robinholzi/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
plt.tight_layout()

label Original



label With Alpha Channel



2.5 Apply Patching

Apply mirror-fill patching to all images. All subsequent analysis uses this patched dataset.

```
In [16]: patched_data = preprocess_data(data)
```

3. Color-based Features

Extract and visualize color-based features that may help distinguish disaster types and damage levels.

3.1 Color Moments

Extract mean, standard deviation, and skewness for each RGB channel to capture color distribution characteristics.

```
In [17]: # Extract color moments from one sample image
sample_features = get_color_moments(patched_data.hurricane_matthew.images[0])
pd.DataFrame([sample_features])
```

```
Out[17]:      r_mean    r_std   r_skew    g_mean    g_std   g_skew    b_mean    b_std   b_skew
0  80.946669  25.259553  1.687784  95.179172  23.573383  1.552585  64.224064  21.924001  2.26661
```

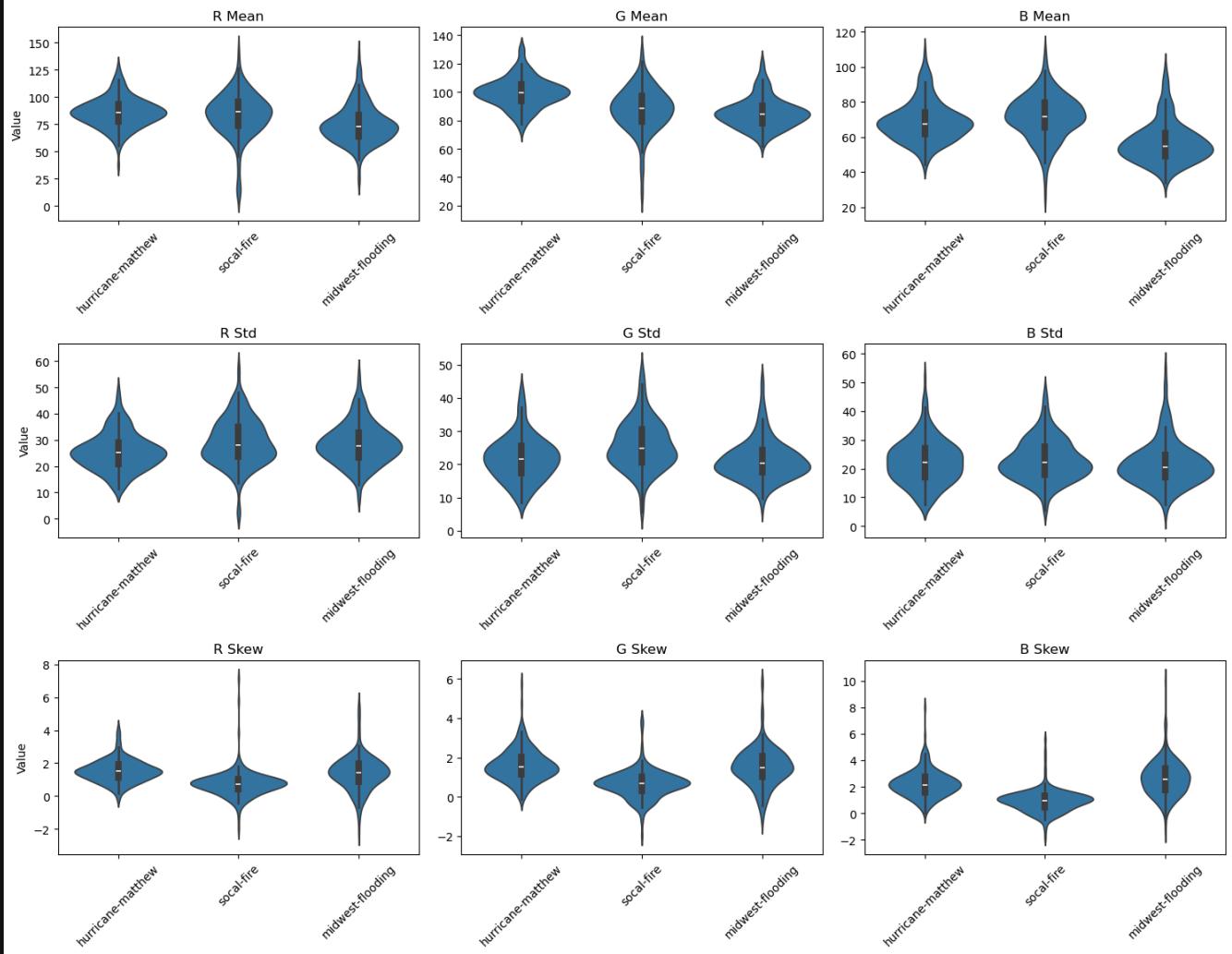
```
In [18]: # Extract color moments for all disasters
df_color_moments = extract_features_by_disaster(patched_data, get_color_moments)
print(f"Extracted features from {len(df_color_moments)} images")
df_color_moments.head()
```

Extracting features: 100%|██████████| 600/600 [00:16<00:00, 36.52it/s]
Extracted features from 600 images

```
Out[18]:      r_mean    r_std   r_skew    g_mean    g_std   g_skew    b_mean    b_std   b_skew  disaster  label
0  80.946669  25.259553  1.687784  95.179172  23.573383  1.552585  64.224064  21.924001  2.266610  hurricane-matthew  1
1  82.295654  32.486160  1.438341  94.330956  31.130185  1.220090  75.778777  30.406815  1.927453  hurricane-matthew  3
2  96.923233  25.478084  1.496852  104.998304  18.720188  2.213880  71.095031  20.712719  2.505107  hurricane-matthew  1
3  80.765044  11.577415  3.832558  106.896722  8.965433  1.356518  83.332263  8.721872  3.763137  hurricane-matthew  3
4  75.076754  34.162408  1.329201  91.805744  27.319979  2.139990  71.761678  27.859699  2.467230  hurricane-matthew  1
```

```
In [19]: # Visualize color moments for all disasters (3x3 grid)
color_features = ['r_mean', 'g_mean', 'b_mean', 'r_std', 'g_std', 'b_std', 'r_skew', 'g_skew', 'b_skew']
plot_feature_violin_by_disaster(
    df_color_moments,
    features=color_features,
    nrows=3,
    suptitle="Color Moments per Disaster Type",
    figsize=(15, 12),
)
plt.show()
```

Color Moments per Disaster Type



3.2 Color Histograms

Analyze the distribution of RGB pixel intensities across disaster types.

```
In [20]: rebuild_hist = False
if rebuild_hist:
    color_intensity_histogram = plot_color_intensity_hist(patched_data, bins=100, images_per_disaster=1000)
    color_intensity_histogram.write_image(
        FIGURES_DIR / "eda" / "color_intensity_histogram.pdf", scale=10.0
    )
```

```
In [21]: # Extract color histogram from one sample image
sample_img = patched_data.hurricane_matthew.images[0]
sample_features = get_color_histogram(sample_img, bins=32)
pd.DataFrame([sample_features])
```

```
Out[21]:   r_hist_0  r_hist_1  r_hist_2  r_hist_3  r_hist_4  r_hist_5  r_hist_6  r_hist_7  r_hist_8  r_hist_9 ... b_hist_22  b_hist_23  b_hist_24  b_hist_2
0  0.000002      0.0  0.000002  0.000225  0.004338  0.020159  0.062877  0.131206  0.18758  0.183578 ...  0.000932  0.000514  0.000279  0.00018
```

1 rows × 96 columns

```
In [22]: # Extract color histogram for all disasters
df_color_histogram = extract_features_by_disaster(
    patched_data, lambda img: get_color_histogram(img, bins=32)
)
print(f"Extracted features from {len(df_color_histogram)} images")
df_color_histogram.head()
```

Extracting features: 100%|██████████| 600/600 [00:11<00:00, 51.59it/s]

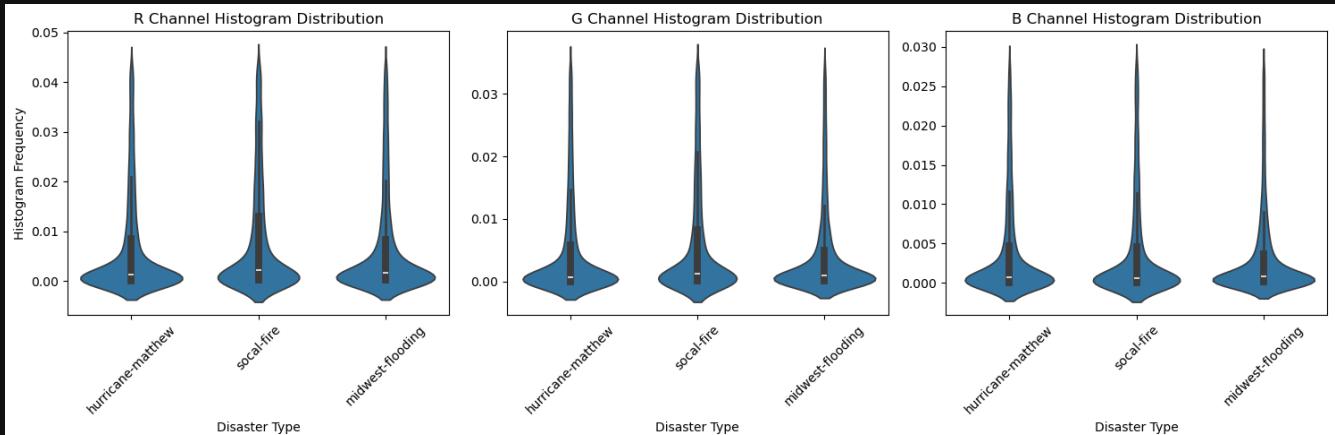
Extracted features from 600 images

| Out [22]: | r_hist_0 | r_hist_1 | r_hist_2 | r_hist_3 | r_hist_4 | r_hist_5 | r_hist_6 | r_hist_7 | r_hist_8 | r_hist_9 | ... | b_hist_24 | b_hist_25 | b_hist_26 |
|-----------|----------|----------|----------|--------------|----------|----------|----------|----------|----------|----------|-----|-----------|-----------|--------------|
| 0 | 0.000002 | 0.000000 | 0.000002 | 2.250671e-04 | 0.004338 | 0.020159 | 0.062877 | 0.131206 | 0.187580 | 0.183578 | ... | 0.000279 | 0.000185 | 1.544952e-04 |
| 1 | 0.000002 | 0.000000 | 0.000008 | 4.100800e-05 | 0.001923 | 0.062041 | 0.140810 | 0.114287 | 0.126428 | 0.135360 | ... | 0.003969 | 0.002934 | 1.928329e-03 |
| 2 | 0.000000 | 0.000000 | 0.000000 | 0.000000e+00 | 0.000000 | 0.000000 | 0.000018 | 0.005012 | 0.056083 | 0.227024 | ... | 0.000842 | 0.000485 | 3.986359e-04 |
| 3 | 0.000000 | 0.000000 | 0.000000 | 9.536743e-07 | 0.000022 | 0.000424 | 0.001400 | 0.005514 | 0.087260 | 0.414827 | ... | 0.000028 | 0.000013 | 9.536743e-07 |
| 4 | 0.000033 | 0.000014 | 0.000103 | 1.011648e-01 | 0.066694 | 0.022664 | 0.044926 | 0.103224 | 0.161695 | 0.147786 | ... | 0.003044 | 0.002165 | 1.008034e-03 |

5 rows × 98 columns



```
In [23]: # Visualize RGB histogram distributions per disaster type using violin plots
plot_histogram_violin_by_disaster(
    df_color_histogram,
    channels=['r', 'g', 'b'],
    hist_prefix='',
    num_bins=32,
    filter_outliers=True,
    iqr_multiplier=0.1,
    save_path=FIGURES_DIR / "eda" / "average_rgb_histograms_per_disaster.pdf",
)
plt.show()
```



3.3 Color Entropy

Measure color diversity entropy (diversity) for each RGB channel.

```
In [24]: # Extract color entropy from one sample image
sample_features = get_color_entropy(patched_data.hurricane_matthew.images[0])
pd.DataFrame([sample_features])
```

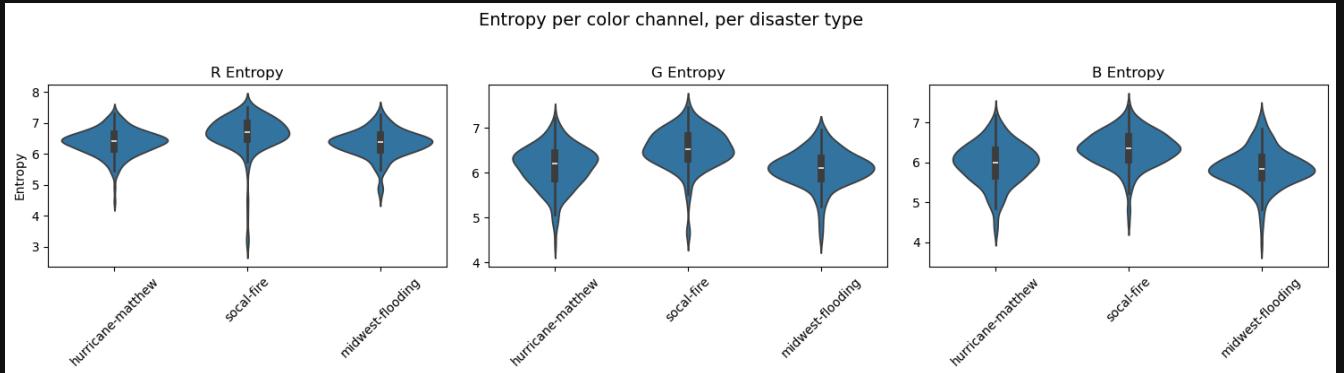
```
Out [24]:   r_entropy  g_entropy  b_entropy
0  6.428279  6.361284  6.036096
```

```
In [25]: # Extract color entropy for all disasters
df_color_entropy = extract_features_by_disaster(patched_data, get_color_entropy)
print(f"Extracted features from {len(df_color_entropy)} images")
df_color_entropy.head()
```

Extracting features: 100%|██████████| 600/600 [00:11<00:00, 53.25it/s]
Extracted features from 600 images

```
Out [25]:   r_entropy  g_entropy  b_entropy      disaster  label
0  6.428279  6.361284  6.036096  hurricane-matthew  1
1  6.721464  6.702422  6.448170  hurricane-matthew  3
2  6.307734  5.850019  5.765490  hurricane-matthew  1
3  4.903666  4.417693  4.279366  hurricane-matthew  3
4  6.673090  6.248124  6.113640  hurricane-matthew  1
```

```
In [26]: # Visualize color entropy for all disasters
plot_feature_violin_by_disaster(
    df_color_entropy,
    features=['r_entropy', 'g_entropy', 'b_entropy'],
    ylabel='Entropy',
    suptitle="Entropy per color channel, per disaster type",
    save_path=FIGURES_DIR / "eda" / "color_entropy_violin.pdf",
)
plt.show()
```



3.4 HSV Histograms

Extract histograms in HSV color space (Hue, Saturation, Value).

```
In [27]: # Extract HSV histogram from one sample image
sample_img = patched_data.hurricane_matthew.images[0]
sample_features = get_hsv_histogram(sample_img, bins=32)
pd.DataFrame([sample_features])
```

```
Out[27]:   h_hist_0  h_hist_1  h_hist_2  h_hist_3  h_hist_4  h_hist_5  h_hist_6  h_hist_7  h_hist_8  h_hist_9 ...  v_hist_22  v_hist_23  v_hist_24  v_hist_2
0  0.000013  0.000007  0.000105  0.001437  0.011891  0.051447  0.17396  0.305024  0.332318  0.117805 ...  0.004493  0.003452  0.002263  0.00
```

1 rows × 96 columns

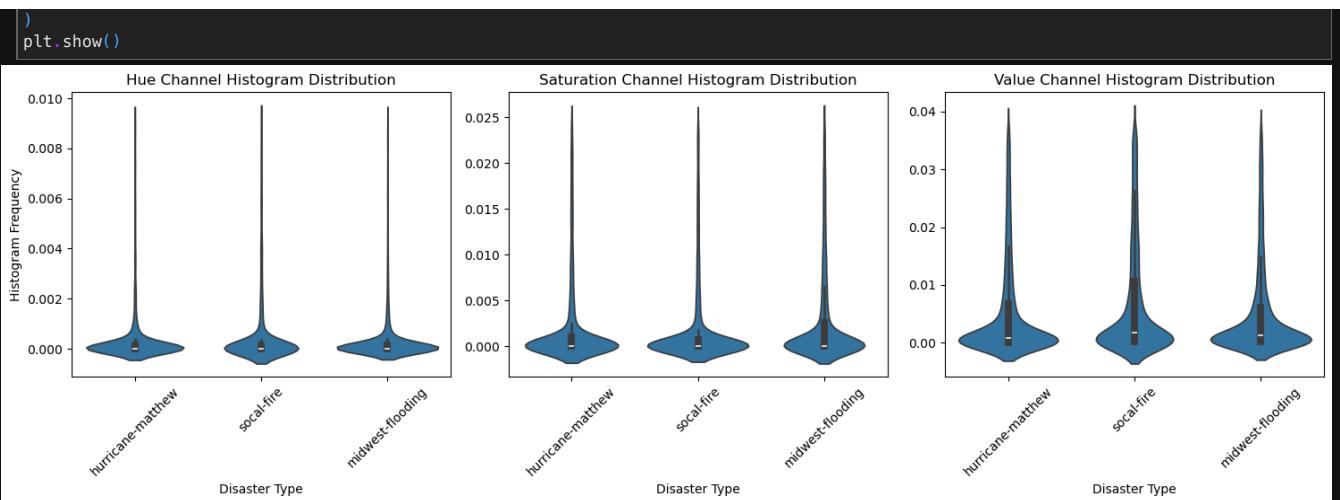
```
In [28]: # Extract HSV histogram for all disasters
df_hsv_histogram = extract_features_by_disaster(
    patched_data, lambda img: get_hsv_histogram(img, bins=32)
)
print(f"Extracted features from {len(df_hsv_histogram)} images")
df_hsv_histogram.head()
```

Extracting features: 100%|██████████| 600/600 [00:58<00:00, 10.24it/s]

```
Extracted features from 600 images
Out[28]:   h_hist_0  h_hist_1  h_hist_2  h_hist_3  h_hist_4  h_hist_5  h_hist_6  h_hist_7  h_hist_8  h_hist_9 ...  v_hist_24  v_hist_25  v_hist_26  v_hist
0  0.000013  0.000007  0.000105  0.001437  0.011891  0.051447  0.173960  0.305024  0.332318  0.117805 ...  0.002263  0.001000  0.000452  0.00
1  0.000059  0.000096  0.000547  0.030729  0.135286  0.118480  0.079616  0.060322  0.083125  0.112291 ...  0.005197  0.004213  0.002840  0.00
2  0.000412  0.000663  0.000646  0.042691  0.129465  0.131012  0.190145  0.395461  0.107595  0.001067 ...  0.002372  0.001224  0.000794  0.00
3  0.000005  0.000000  0.000051  0.000583  0.005885  0.012086  0.019405  0.022624  0.028752  0.047679 ...  0.000062  0.000037  0.000013  0.00
4  0.000320  0.000208  0.000467  0.002690  0.010624  0.026950  0.110952  0.238033  0.243384  0.106144 ...  0.005957  0.004189  0.002993  0.001
```

5 rows × 98 columns

```
In [29]: # Visualize HSV histogram distributions per disaster type using violin plots
plot_histogram_violin_by_disaster(
    df_hsv_histogram,
    channels=['h', 's', 'v'],
    hist_prefix='',
    num_bins=32,
    channel_labels=['Hue', 'Saturation', 'Value'],
    filter_outliers=True,
    iqr_multiplier=0.1,
    save_path=FIGURES_DIR / "eda" / "average_hsv_histograms_per_disaster.pdf",
```



3.5 Color Ratios

Extract R/G, G/B, and R/B ratios.

```
In [30]: # Extract color ratios from one sample image
sample_features = get_color_ratios(patched_data.hurricane_matthew.images[0])
pd.DataFrame([sample_features])
```

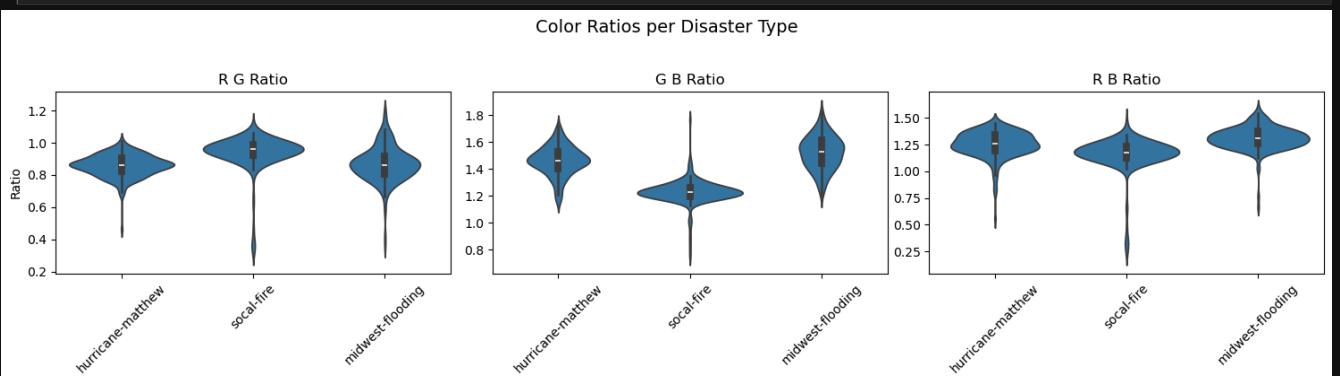
```
Out[30]:   r_g_ratio  g_b_ratio  r_b_ratio
0    0.850466   1.481986   1.260379
```

```
In [31]: # Extract color ratios for all disasters
df_color_ratios = extract_features_by_disaster(patched_data, get_color_ratios)
print(f"Extracted features from {len(df_color_ratios)} images")
df_color_ratios.head()
```

Extracting features: 100%|██████████| 600/600 [00:00<00:00, 847.57it/s]
Extracted features from 600 images

```
Out[31]:   r_g_ratio  g_b_ratio  r_b_ratio      disaster  label
0    0.850466   1.481986   1.260379  hurricane-matthew    1
1    0.872414   1.244820   1.085999  hurricane-matthew    3
2    0.923093   1.476873   1.363291  hurricane-matthew    1
3    0.755543   1.282777   0.969193  hurricane-matthew    3
4    0.817778   1.279314   1.046196  hurricane-matthew    1
```

```
In [32]: # Visualize color ratios for all disasters
plot_feature_violin_by_disaster(
    df_color_ratios,
    features=['r_g_ratio', 'g_b_ratio', 'r_b_ratio'],
    ylabel='Ratio',
    suptitle="Color Ratios per Disaster Type",
    save_path=FIGURES_DIR / "eda" / "color_ratios_per_disaster.pdf",
)
plt.show()
```



4. Texture & Edge Features

Extract texture and edge-based features that capture structural patterns in the images.

```
In [33]: sample_dc20 = patched_data.get_random_sample_disaster_classification(10)
sample_dc100 = patched_data.get_random_sample_disaster_classification(50)
sample_dc100 = interleave(sample_dc100, 2)
```

4.1 Gabor Filters

```
In [34]: sample_img = patched_data.hurricane_matthew.images[0]

theta = 0
sigma = 1.0
frequency = 0.1

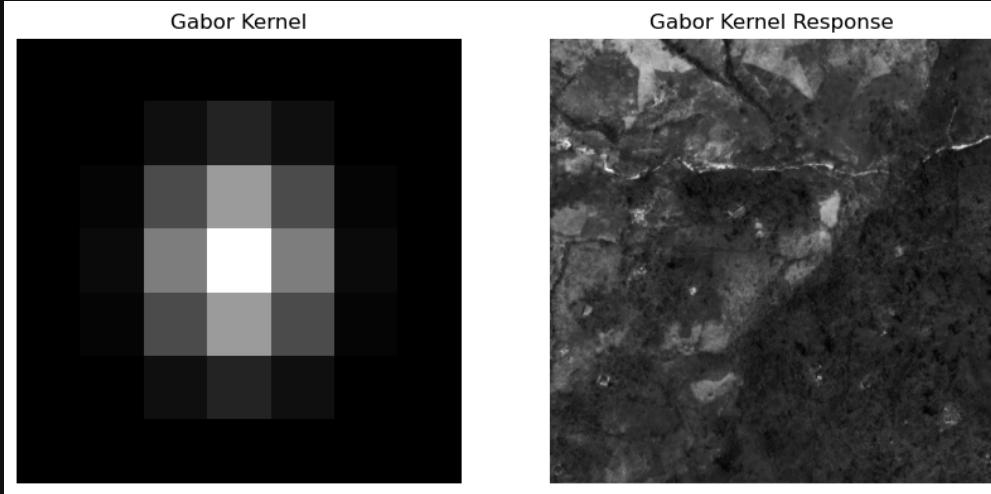
kernel = generate_gabor_kernel(theta, sigma, frequency)
gabor = get_gabor_features(sample_img, kernel)

fig, axs = plt.subplots(1, 2, figsize=(10, 5)) # 1 row, 2 columns

# Plot Gabor Kernel
axs[0].imshow(kernel, cmap="gray")
axs[0].axis("off") # Remove axis
axs[0].set_title("Gabor Kernel")

# Plot Gabor Kernel Response
axs[1].imshow(gabor, cmap="gray")
axs[1].axis("off") # Remove axis
axs[1].set_title("Gabor Kernel Response")

plt.show()
```



4.2 Sobel Edge Detection

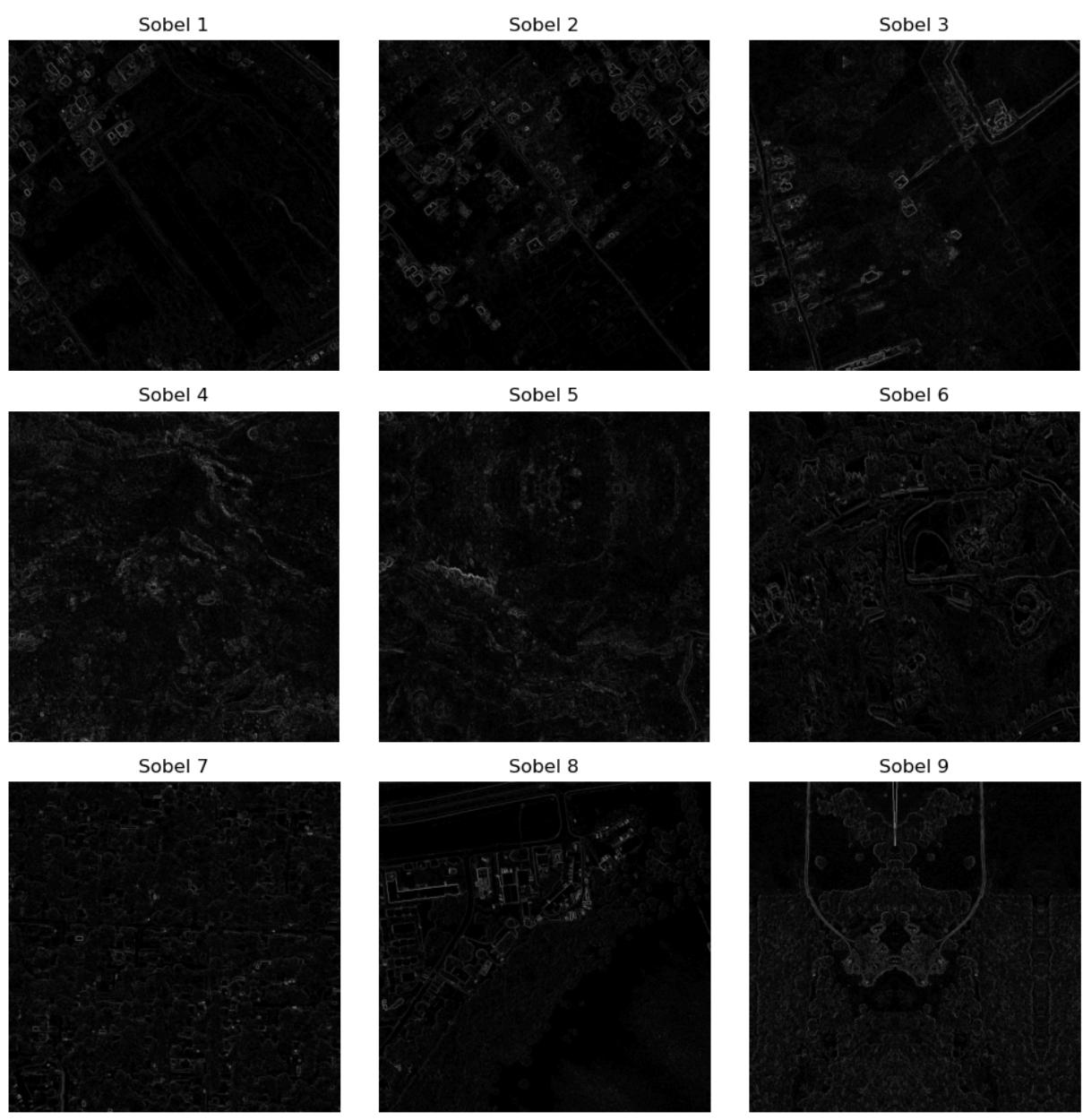
Detect edges using Sobel operators to capture structural boundaries.

```
In [35]: fig, axes = plt.subplots(3,3, figsize=(10,10))

for idx, img in enumerate(patched_data.get_random_sample_across_disasters(3)):
    row, col = divmod(idx, 3) # convert index to row/col
    edges = get_sobel_features(img.image)

    axes[row, col].imshow(edges, cmap="gray")
    axes[row, col].set_title(f"Sobel {idx+1}")
    axes[row, col].axis("off")

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



```
In [36]: images_sobel = []
features_fire = []
features_flooding = []

for idx, img in enumerate(sample_dc100):
    sobel = get_sobel_features(img.image)
    images_sobel.append(sobel)

    hist, _ = np.histogram(sobel.ravel(), bins=int(sobel.max() + 1), range=(0, sobel.max() + 1))
    hist = hist / hist.sum()
    img_entropy = -np.sum(hist[hist > 0] * np.log2(hist[hist > 0]))
    img_mean = sobel.mean()
    img_std = sobel.std()
    nonzero_count = np.count_nonzero(sobel)

    fv = np.array([img_entropy, img_mean, img_std, nonzero_count])

    fv = np.append(fv, nonzero_count)
    if idx % 2 == 0:
        features_fire.append(fv)
    else:
        features_flooding.append(fv)

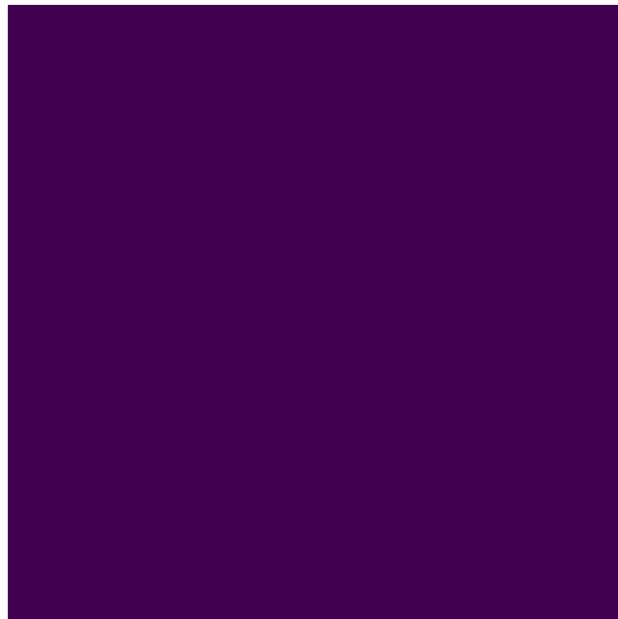
render_images(images_sobel[0:2], ncols=2, labels=['socal-fire', 'midwest-flooding']);
df_flood = pd.DataFrame(features_flooding)
df_fire = pd.DataFrame(features_fire)

results = pd.DataFrame()
```

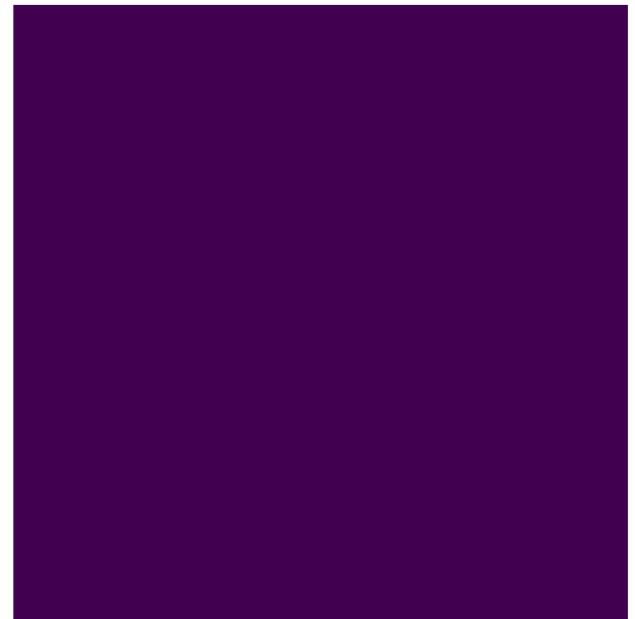
```
results['mean flood'] = df_flood.mean()
results['mean fire'] = df_fire.mean()
results['std flood'] = df_flood.std()
results['std fire'] = df_fire.std()
display(results)
```

```
/Users/robinholzi/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
plt.tight_layout()
```

label socal-fire



label midwest-flooding



| | mean flood | mean fire | std flood | std fire |
|---|--------------|--------------|------------|------------|
| 0 | 0.000000e+00 | 0.000000e+00 | 0.000000 | 0.000000 |
| 1 | 2.385613e-02 | 2.447254e-02 | 0.004938 | 0.006279 |
| 2 | 2.405656e-02 | 2.123228e-02 | 0.005085 | 0.006318 |
| 3 | 1.048316e+06 | 1.048418e+06 | 535.279601 | 650.734204 |
| 4 | 1.048316e+06 | 1.048418e+06 | 535.279601 | 650.734204 |

4.3 Local Binary Patterns (LBP)

Capture texture patterns using local binary pattern descriptors.

```
In [37]: lbp = get_local_binary_pattern(sample_dc100[0].image, radius=3)
plt.imshow(lbp, cmap="gray")
plt.axis("off")
plt.title("Local Binary Pattern (LBP)")
plt.show()
```

Local Binary Pattern (LBP)



```
In [38]: images_lbp = []
features_fire = []
features_flooding = []

for idx, img in enumerate(sample_dc100):
    lbp = get_local_binary_pattern(img.image, radius=3)
    images_lbp.append(lbp)

    hist, _ = np.histogram(lbp.ravel(), bins=int(lbp.max() + 1), range=(0, lbp.max() + 1))
    hist = hist / hist.sum()
    img_entropy = -np.sum(hist[hist > 0] * np.log2(hist[hist > 0]))
    img_mean = lbp.mean()
    img_std = lbp.std()
    nonzero_count = np.count_nonzero(lbp)

    fv = np.array([img_entropy, img_mean, img_std, nonzero_count])

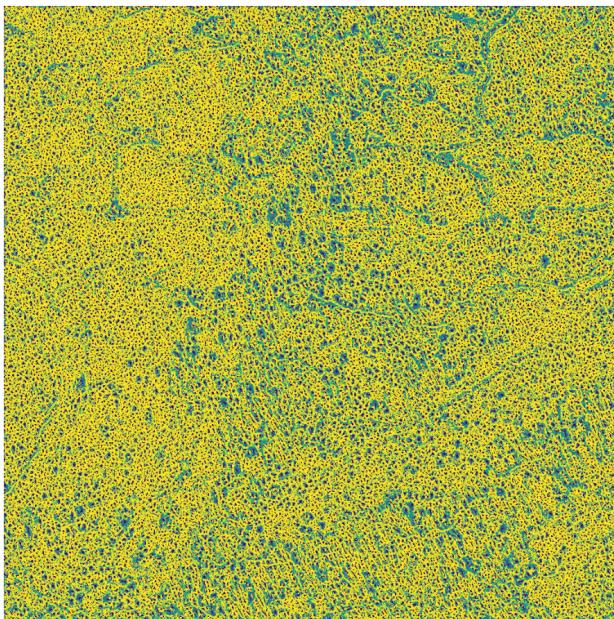
    fv = np.append(fv, nonzero_count)
    if idx % 2 == 0:
        features_fire.append(fv)
    else:
        features_flooding.append(fv)

render_images(images_lbp[0:2], ncols=2, labels=['socal-fire', 'midwest-flooding']);
df_flood = pd.DataFrame(features_flooding)
df_fire = pd.DataFrame(features_fire)

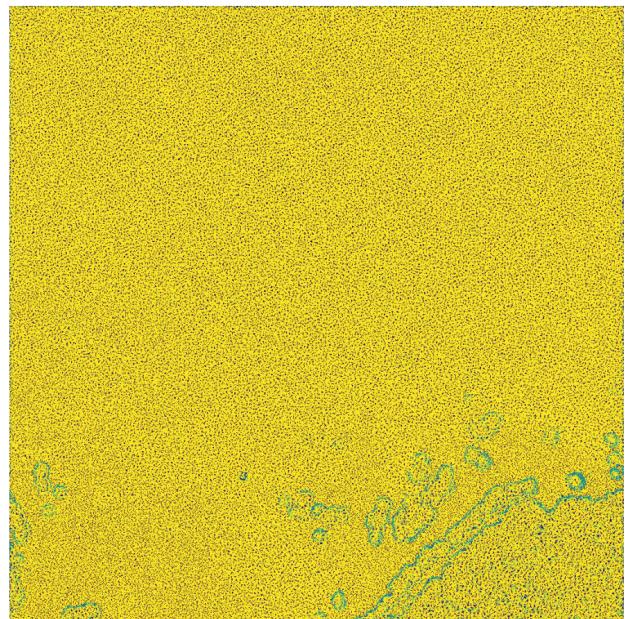
results = pd.DataFrame()
results['mean flood'] = df_flood.mean()
results['mean fire'] = df_fire.mean()
results['std flood'] = df_flood.std()
results['std fire'] = df_fire.std()
display(results)

/Users/robinholzi/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
plt.tight_layout()
```

label socal-fire



label midwest-flooding

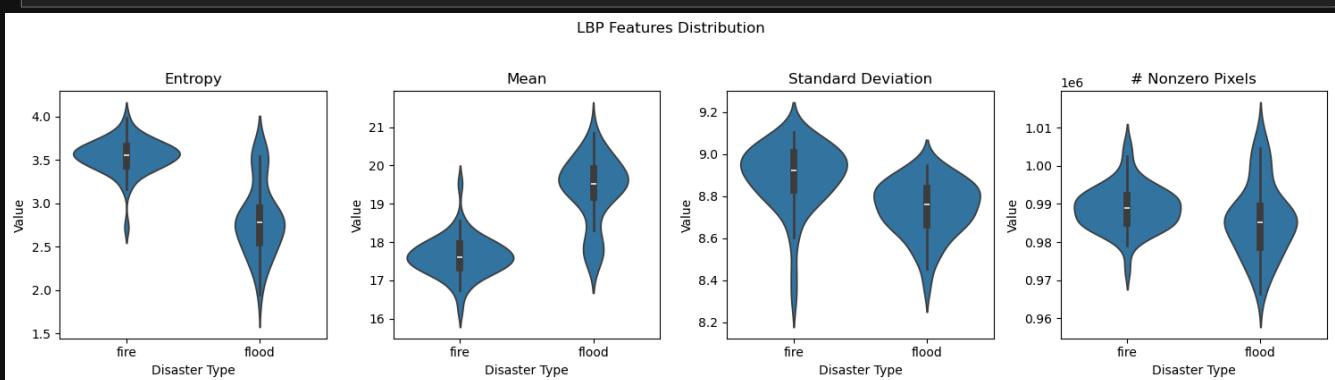


| | mean flood | mean fire | std flood | std fire |
|---|---------------|---------------|-------------|-------------|
| 0 | 2.813866 | 3.544877 | 0.413570 | 0.197876 |
| 1 | 19.366262 | 17.620881 | 0.857580 | 0.513790 |
| 2 | 8.741378 | 8.892735 | 0.130338 | 0.155433 |
| 3 | 985719.180000 | 988952.760000 | 9519.055183 | 5859.863569 |
| 4 | 985719.180000 | 988952.760000 | 9519.055183 | 5859.863569 |

```
In [39]: fig, axes = plt.subplots(1,4, figsize=(15, 4))
axes = axes.flatten()
features = range(4)
labels = ['Entropy', 'Mean', 'Standard Deviation', '# Nonzero Pixels']

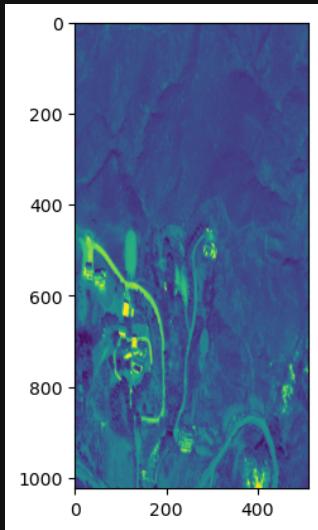
for ax, feature_nr, lbl in zip(axes, features, labels):
    df_plot = pd.DataFrame({
        'value': list(df_fire[feature_nr]) + list(df_flood[feature_nr]),
        'disaster': ['fire'] * len(df_fire) + ['flood'] * len(df_flood)
    })
    sns.violinplot(data=df_plot, x='disaster', y='value', ax=ax)
    ax.set_title(f'{lbl}')
    ax.set_xlabel('Disaster Type')
    ax.set_ylabel('Value')

fig.suptitle("LBP Features Distribution", y=1.03)
plt.tight_layout()
fig.savefig(
    FIGURES_DIR / "eda" / "lbp_features_distribution.pdf",
    dpi=300,
    bbox_inches="tight"
)
plt.show()
```



Extract multi-scale frequency information using wavelet transforms.

```
In [40]: wvf = get_wavelet_features(patched_data.socal_fire.images[2])
cA=wvf['CA']
cA = cA.mean(axis=2)
plt.imshow(np.array(cA));
```



4.5 Haralick Features

Compute texture statistics from gray-level co-occurrence matrices.

```
In [41]: images_hrk = []
features_fire = []
features_flooding = []

for idx, img in enumerate(sample_dc100):
    labeled_img, feature = get_haralick_features(img.image)
    images_hrk.append(labeled_img)

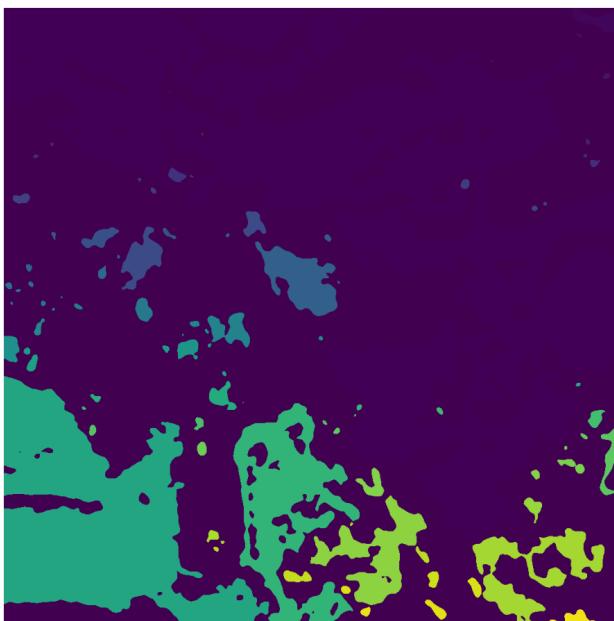
    nonzero_count = labeled_img.size - np.count_nonzero(labeled_img)
    haralick_keys = sorted([k for k in feature.keys() if k.startswith("haralick_")])
    fv = np.array([feature[k] for k in haralick_keys])
    fv = np.append(fv, nonzero_count)
    if idx % 2 == 0:
        features_fire.append(fv)
    else:
        features_flooding.append(fv)

fig = render_images(images_hrk[2:4], ncols=2, labels=['socal-fire', 'midwest-flooding'])
fig.savefig(
    FIGURES_DIR / "eda" / "haralick_labeled_images.pdf",
    dpi=300,
    bbox_inches="tight"
)
df_flood = pd.DataFrame(features_flooding)
df_fire = pd.DataFrame(features_fire)

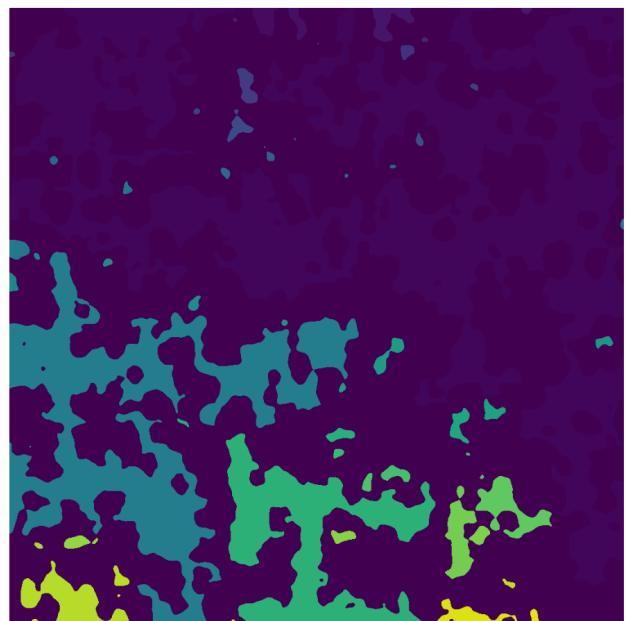
results = pd.DataFrame()
results['mean flood'] = df_flood.mean()
results['mean fire'] = df_fire.mean()
results['std flood'] = df_flood.std()
results['std fire'] = df_fire.std()
display(results)
```

```
/Users/robinholzi/git/ucb/data200/data200-grad-project/data200/plot/image.py:96: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
plt.tight_layout()
```

label socal-fire



label midwest-flooding



| | mean flood | mean fire | std flood | std fire |
|----|---------------|---------------|---------------|--------------|
| 0 | 0.454409 | 0.388047 | 0.069457 | 0.050189 |
| 1 | 99.531738 | 32.088856 | 268.337908 | 32.597979 |
| 2 | 0.182221 | 0.178960 | 0.081719 | 0.047351 |
| 3 | -0.909922 | -0.920013 | 0.025407 | 0.018664 |
| 4 | 0.967283 | 0.979501 | 0.019157 | 0.014279 |
| 5 | 0.948192 | 0.962132 | 0.019432 | 0.014038 |
| 6 | 825.004538 | 434.567148 | 2126.196713 | 480.977739 |
| 7 | 0.985714 | 0.985416 | 0.007083 | 0.004187 |
| 8 | 15.965200 | 19.652894 | 20.870706 | 13.379647 |
| 9 | 3200.486413 | 1706.181738 | 8239.515114 | 1893.349856 |
| 10 | 1.731700 | 1.977741 | 0.403311 | 0.382089 |
| 11 | 1.761782 | 2.005946 | 0.416324 | 0.388485 |
| 12 | 0.013581 | 0.014707 | 0.010758 | 0.025148 |
| 13 | 589915.220000 | 550730.860000 | 124682.030144 | 58716.232298 |

The different features have different distributions.

```
In [42]: component_stats_data = []

for idx, (s, transf) in enumerate(zip(sample_dc100, images_hrk)):
    features = get_connected_component_stats(s.image)
    disaster = "socal-fire" if idx % 2 == 0 else "midwest-flooding"
    features['disaster'] = disaster
    features['label'] = s.label
    component_stats_data.append(features)

df_component_stats = pd.DataFrame(component_stats_data)
print(f"Extracted features from {len(df_component_stats)} images")
```

Extracted features from 100 images

```
In [43]: df_component_stats.head(100)
```

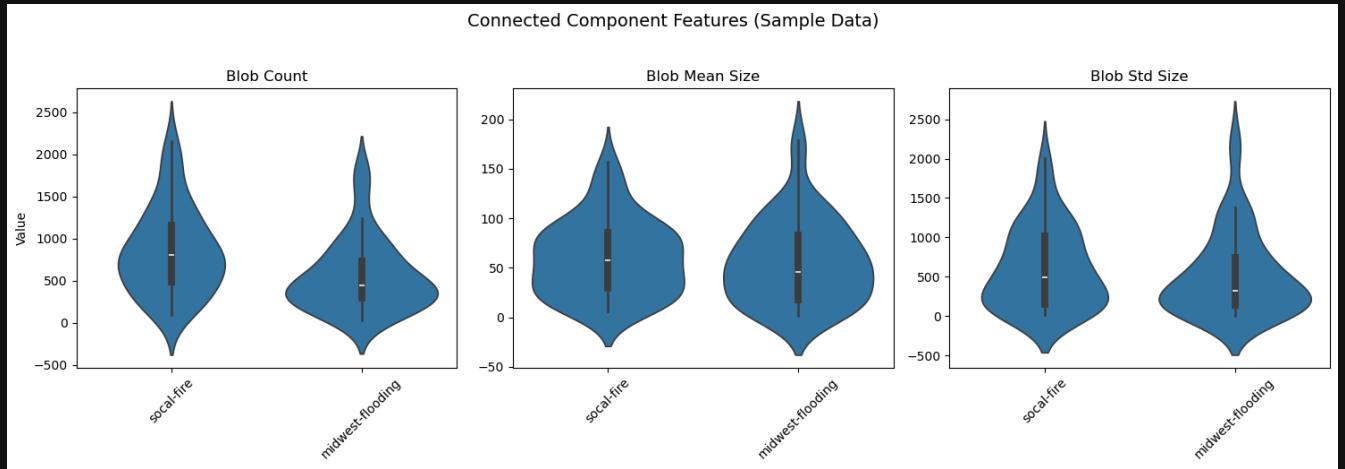
Out [43]:

| | blob_count | blob_mean_size | blob_std_size | disaster | label |
|-----|------------|----------------|---------------|------------------|-------|
| 0 | 808 | 16.227723 | 60.891277 | socal-fire | 3 |
| 1 | 5129 | 1.283291 | 1.596165 | midwest-flooding | 3 |
| 2 | 858 | 38.500000 | 230.216272 | socal-fire | 3 |
| 3 | 698 | 46.277937 | 142.962746 | midwest-flooding | 0 |
| 4 | 921 | 74.819761 | 487.684954 | socal-fire | 3 |
| ... | ... | ... | ... | ... | ... |
| 95 | 788 | 18.039340 | 116.051351 | midwest-flooding | 3 |
| 96 | 149 | 17.530201 | 55.688438 | socal-fire | 3 |
| 97 | 239 | 117.158996 | 1308.493001 | midwest-flooding | 0 |
| 98 | 1232 | 124.956981 | 845.383393 | socal-fire | 0 |
| 99 | 913 | 63.210296 | 863.530019 | midwest-flooding | 0 |

100 rows × 5 columns

In [44]:

```
# Visualize connected component stats (sample data from Haralick exploration)
fig = plot_feature_violin_by_disaster(
    df_component_stats,
    features=['blob_count', 'blob_mean_size', 'blob_std_size'],
    ylabel='Value',
    suptitle="Connected Component Features (Sample Data)",
    figsize=(15, 5),
    filter_outliers=True,
)
fig.savefig(
    FIGURES_DIR / "eda" / "connected_component_features.pdf",
    dpi=300,
    bbox_inches="tight"
)
plt.show()
```

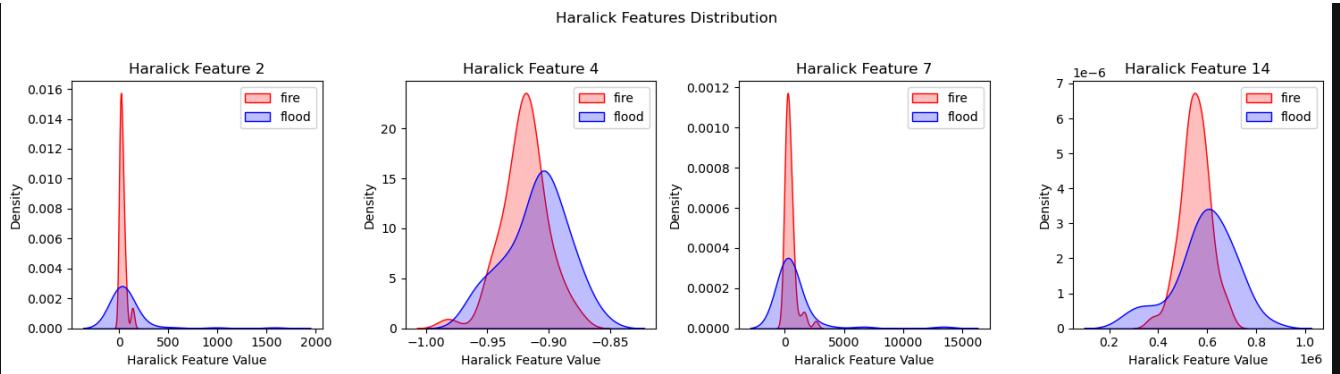


In [45]:

```
fig, axes = plt.subplots(1, 4, figsize=(15, 4))
axes = axes.flatten()
features = [1, 3, 6, 13]

for ax, feature_nr in zip(axes, features):
    sns.kdeplot(df_fire[feature_nr], ax=ax, fill=True, color='red', label='fire')
    sns.kdeplot(df_flood[feature_nr], ax=ax, fill=True, color='blue', label='flood')
    ax.set_title(f'Haralick Feature {feature_nr+1}')
    ax.set_xlabel(f'Haralick Feature Value')
    ax.set_ylabel('Density')
    ax.legend()

fig.suptitle("Haralick Features Distribution", y=1.03)
plt.tight_layout()
fig.savefig(
    FIGURES_DIR / "eda" / "haralick_features_distribution.pdf",
    dpi=300,
    bbox_inches="tight"
)
plt.show()
```



4.6 Haar-like Features

Extract rectangular Haar-like features commonly used in object detection.

```
In [46]: # Extract Haar-like features from a random sample (computationally expensive)
# Note: Images are downsampled to 32x32 patches for efficiency
n_samples_per_disaster = 15 # Use subset for speed

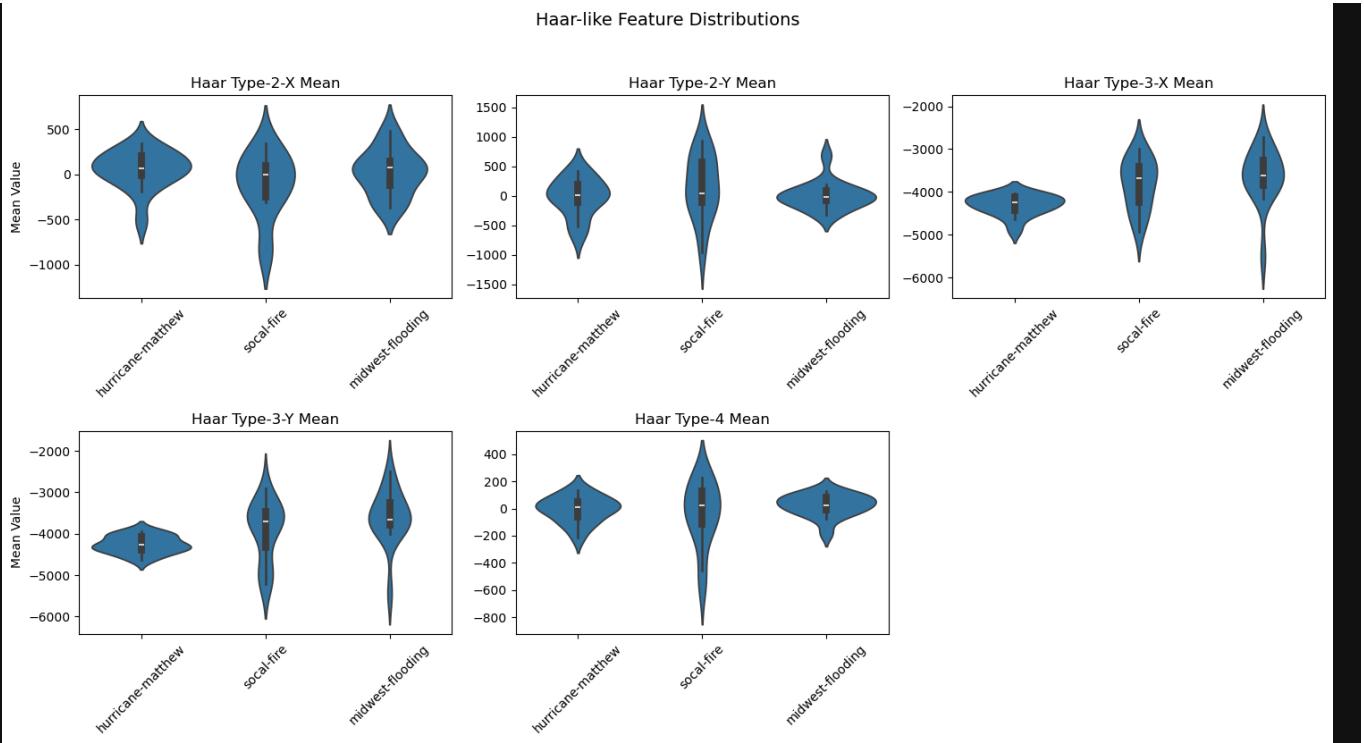
haar_data = []
for disaster in DISASTER_LIST:
    images = patched_data.get(disaster).images
    labels = patched_data.get(disaster).labels
    # Random sample
    indices = np.random.choice(len(images), min(n_samples_per_disaster, len(images)), replace=False)
    for idx in indices:
        features = get_haar_features(images[idx])
        features['disaster'] = disaster
        features['label'] = labels[idx]
        haar_data.append(features)

df_haar = pd.DataFrame(haar_data)
print(f"Extracted Haar features from {len(df_haar)} images ({n_samples_per_disaster} per disaster)")
df_haar.head()
```

Extracted Haar features from 45 images (15 per disaster)

| | haar_type-2-x_mean | haar_type-2-x_std | haar_type-2-x_max | haar_type-2-y_mean | haar_type-2-y_std | haar_type-2-y_max | haar_type-3-x_mean | haar_type-3-x_std | haar_type-3-x_max | haar_type-3-y_mean | haar_type-3-y_std | ha |
|---|--------------------|-------------------|-------------------|--------------------|-------------------|-------------------|--------------------|-------------------|-------------------|--------------------|-------------------|----|
| 0 | 341.863281 | 867.197165 | 6277 | 282.422289 | 672.137130 | 4629 | -4095.517723 | 4260.740924 | -53 | -3939.690220 | 4064.747393 | |
| 1 | 99.169907 | 387.740891 | 1828 | -0.157301 | 258.384937 | 1024 | -4312.193434 | 4278.900704 | -70 | -4376.840863 | 4417.625267 | |
| 2 | -25.989864 | 633.122256 | 4114 | 11.664551 | 494.141581 | 2277 | -4128.510239 | 4210.132836 | -17 | -4014.817723 | 3990.367666 | |
| 3 | 181.402654 | 613.565523 | 4421 | 39.354063 | 229.898003 | 1687 | -4428.174334 | 4602.728089 | -76 | -4312.975115 | 4335.256133 | |
| 4 | 258.843424 | 534.792263 | 3589 | 382.704812 | 915.182142 | 5738 | -4166.074242 | 4196.309824 | -66 | -4243.226768 | 4359.765578 | |

```
In [47]: # Visualize Haar feature distributions (mean statistics)
haar_mean_features = [col for col in df_haar.columns if col.startswith('haar_') and '_mean' in col]
plot_feature_violin_by_disaster(
    df_haar,
    features=haar_mean_features,
    ylabel='Mean Value',
    suptitle="Haar-like Feature Distributions",
    nrows=2,
    save_path=FIGURES_DIR / "eda" / "haar_features_per_disaster.pdf",
)
plt.show()
```



5. Structural Features

Extract features that capture the structural composition and geometric properties of images.

5.1 Edge-to-Area Ratio

Measure proportion of edge pixels (Sobel and Canny variants).

```
In [48]: # Extract edge ratios from one sample image
sample_sobel = get_edge_to_area_ratio_sobel(patched_data.hurricane_matthew.images[0])
sample_canny = get_edge_to_area_ratio_canny(patched_data.hurricane_matthew.images[0])
sample_features = {**sample_sobel, **sample_canny}
pd.DataFrame([sample_features])
```

```
Out[48]:   edge_ratio_sobel  edge_ratio_canny
0          0.02152        0.105304
```

```
In [49]: # Extract edge ratios for all disasters
def get_edge_ratios(img):
    """Combine Sobel and Canny edge ratio features."""
    return {**get_edge_to_area_ratio_sobel(img), **get_edge_to_area_ratio_canny(img)}

df_edge_ratio = extract_features_by_disaster(patched_data, get_edge_ratios)
print(f"Extracted features from {len(df_edge_ratio)} images")
df_edge_ratio.head()
```

Extracting features: 100%|██████████| 600/600 [00:39<00:00, 15.12it/s]

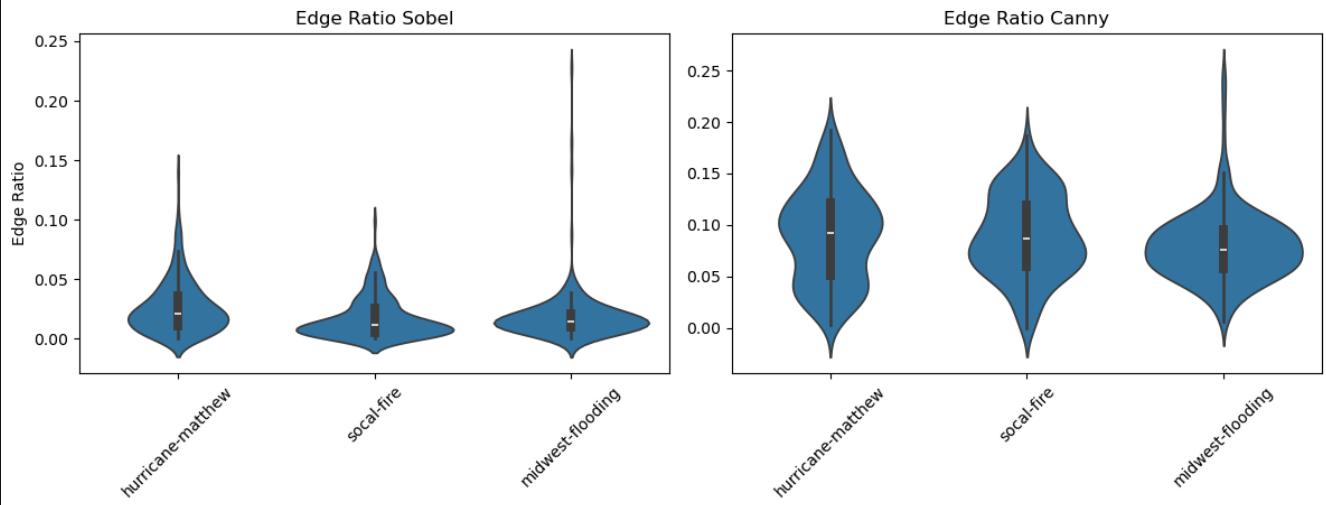
Extracted features from 600 images

```
Out[49]:   edge_ratio_sobel  edge_ratio_canny      disaster  label
0          0.021520        0.105304  hurricane-matthew  1
1          0.033147        0.090958  hurricane-matthew  3
2          0.014235        0.044187  hurricane-matthew  1
3          0.004337        0.015178  hurricane-matthew  3
4          0.033848        0.090000  hurricane-matthew  1
```

```
In [50]: # Visualize edge ratios for all disasters
plot_feature_violin_by_disaster(
    df_edge_ratio,
    features=['edge_ratio_sobel', 'edge_ratio_canny'],
    ylabel='Edge Ratio',
    suptitle="Edge-to-Area Ratios per Disaster Type",
    figsize=(12, 5),
    save_path=FIGURES_DIR / "eda" / "edge_ratios_per_disaster.pdf",
```

```
)  
plt.show()
```

Edge-to-Area Ratios per Disaster Type



5.2 Connected Components

Count and measure blobs/connected components.

```
In [51]: # Extract connected component stats from one sample image  
sample_features = get_connected_component_stats(patched_data.hurricane_matthew.images[0])  
pd.DataFrame([sample_features])
```

```
Out[51]: blob_count blob_mean_size blob_std_size  
0 3580 22.222067 312.727558
```

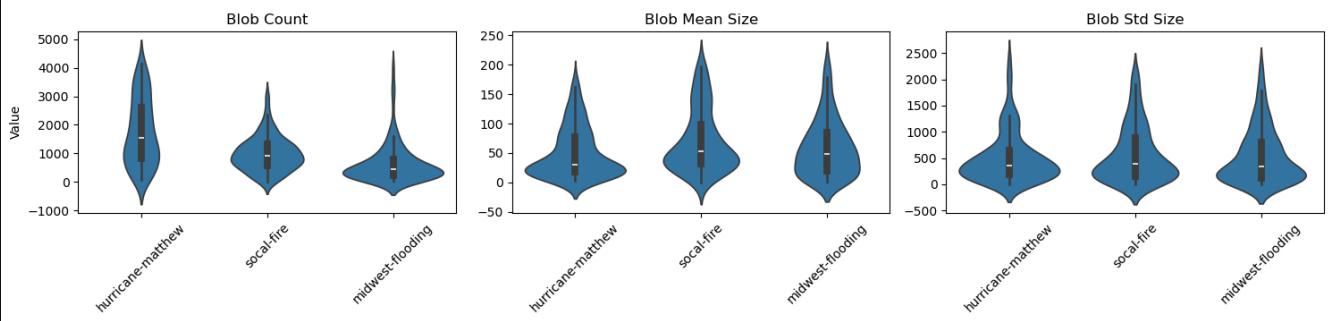
```
In [52]: # Extract connected component stats for all disasters  
df_component_stats = extract_features_by_disaster(patched_data, get_connected_component_stats)  
print(f"Extracted features from {len(df_component_stats)} images")  
df_component_stats.head()
```

```
Extracting features: 100%|██████████| 600/600 [00:16<00:00, 36.77it/s]  
Extracted features from 600 images
```

```
Out[52]: blob_count blob_mean_size blob_std_size disaster label  
0 3580 22.222067 312.727558 hurricane-matthew 1  
1 3774 25.851086 727.705226 hurricane-matthew 3  
2 725 118.084138 1266.159183 hurricane-matthew 1  
3 356 47.426966 471.685583 hurricane-matthew 3  
4 1025 86.920976 1335.470351 hurricane-matthew 1
```

```
In [53]: # Visualize connected component stats for all disasters (with outlier filtering)  
plot_feature_violin_by_disaster(  
    df_component_stats,  
    features=['blob_count', 'blob_mean_size', 'blob_std_size'],  
    ylabel='Value',  
    subtitle="Connected Component Features (Outliers Filtered)",  
    filter_outliers=True,  
    save_path=FIGURES_DIR / "eda" / "connected_component_stats_per_disaster.pdf",  
)  
plt.show()
```

Connected Component Features (Outliers Filtered)



5.3 Corner Detection

Count corners using `goodFeaturesToTrack`.

```
In [54]: # Extract corner count from one sample image
sample_features = get_corner_count(patched_data.hurricane_matthew.images[0])
pd.DataFrame([sample_features])
```

```
Out[54]:    corner_count
0            3872
```

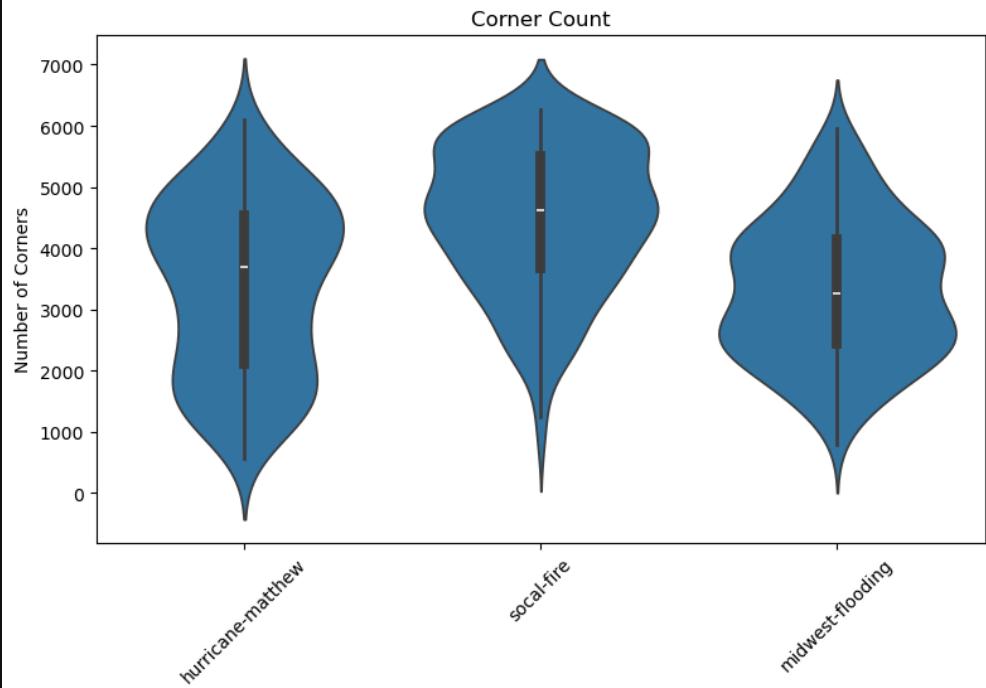
```
In [55]: # Extract corner counts for all disasters
df_corner_count = extract_features_by_disaster(patched_data, get_corner_count)
print(f"Extracted features from {len(df_corner_count)} images")
df_corner_count.head()
```

```
Extracting features: 100%|██████████| 600/600 [00:07<00:00, 78.50it/s]
Extracted features from 600 images
```

```
Out[55]:    corner_count    disaster  label
0            3872  hurricane-matthew    1
1            3015  hurricane-matthew    3
2            2608  hurricane-matthew    1
3             566  hurricane-matthew    3
4            1684  hurricane-matthew    1
```

```
In [56]: # Visualize corner counts for all disasters
plot_feature_violin_by_disaster(
    df_corner_count,
    features=['corner_count'],
    ylabel='Number of Corners',
    suptitle="Corner Count Comparison Across Disasters",
    figsize=(8, 6),
)
plt.show()
```

Corner Count Comparison Across Disasters



disaster_prediction.py

```
1  """
2  Training and prediction pipeline for disaster type prediction.
3
4  Run via `pixi run disaster-prediction --help`
5
6  E.g.:
7
8  # With hyper-parameter tuning (model selection: logistic_regression or cnn)
9  pixi run disaster-prediction fit \
10     --model logistic_regression \
11     --no-skip-cross-validation
12
13 # For prediction on unlabeled test set
14 pixi run disaster-prediction predict \
15     --model logistic_regression
16 """
17
18 import json
19 import warnings
20 from pathlib import Path
21 from typing import Literal, get_args
22
23 import numpy as np
24 import torch
25 import torch.nn as nn
26 import torch.optim as optim
27 import typer
28 from data200.const.const import (
29     CNN_AUGMENTATIONS,
30     DATA_DIR,
31     DISASTER_TYPE_PREDICTIONS_PATH,
32     MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION,
33 )
34 from data200.evaluate.cross_validation import k_fold_cross_validation
35 from data200.evaluate.evaluator import BasicCrossValidationEvaluator
36 from data200.evaluate.grid_search import (
37     grid_search_cross_validation,
38     grid_search_decision_threshold,
39 )
40 from data200.feature.transform import (
41     IdentityTransformer,
42     StandardScalerTransformer,
43     convert_to_image_tensor,
44     convert_to_trainable_tensor,
45 )
46 from data200.io.load import (
47     load_disaster_data,
48 )
49 from data200.io.save import save_predictions
50 from data200.model.classic.logistic import LogisticRegressionTrainer
51 from data200.model.nn.architecture.cnn import CNNClassifier
52 from data200.model.nn.ensemble import EnsembleTrainer
53 from data200.model.nn.pytorch import PytorchTrainer, PytorchTrainingConfig
54 from data200.pipeline.config import NNConfig
55 from data200.pipeline.feature import (
56     feature_pipeline_flood_fire_logistic_regression,
57     feature_pipeline_flood_fire_neural_networks,
58 )
59 from data200.pipeline.preprocess import preprocess_data
60 from data200.preprocess.augmentation import apply_augmentations
61 from data200.sample.sample import BalancedUpsampler, IdentitySampler
62 from data200.utils.device import get_device
63 from data200.utils.utils import set_seed
64 from pydantic import BaseModel
65
66 app = typer.Typer()
67
68 DisasterTypeModel = Literal["logistic_regression", "cnn"]
69 DISASTER_TYPE_MODELS = list(get_args(DisasterTypeModel))
70
71 SEED = 42
72
73 TRANSFORMER_PATHS: dict[DisasterTypeModel, Path] = {
```

```

74     model: MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION / f"{model}_scaler.json"
75     for model in DISASTER_TYPE_MODELS
76   }
77 TRAINER_CONFIG_PATHS = {
78   model: MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION / f"{model}_trainer_config.json"
79   for model in DISASTER_TYPE_MODELS
80 }
81 MODEL_PATHS = {
82   model: MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION / f"{model}_model.json"
83   for model in DISASTER_TYPE_MODELS
84 }
85 EVAL_RESULTS_PATHS = {
86   model: MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION
87   / f"{model}_grid_search_eval_results.json"
88   for model in DISASTER_TYPE_MODELS
89 }
90
91 # ----- TRAINING -----
92 # -----
93 # -----
94
95 # ----- LOGISTIC -----
96 # -----
97
98
99 def fit_logistic_regression(
100   data_path: Path = DATA_DIR,
101   skip_cross_validation: bool = True,
102   train_ensemble: bool = True, # More robust 99.5% - 100% accuracy
103 ):
104   """Fit logistic regression model for disaster type prediction."""
105   model = "logistic_regression"
106
107   # Load the data
108   data = load_disaster_data(path=data_path)
109
110   # Preprocess the data, overwrite to save memory
111   data = preprocess_data(data)
112
113   # ----- FEATURE ENGINEERING -----
114
115   trainable_dataset = feature_pipeline_flood_fire_logistic_regression(data)
116   del data
117
118   # ----- CROSS VALIDATION AND TRAINING -----
119
120   print(f"Training {model} model...")
121   transformer = StandardScalerTransformer()
122
123   # logistic regression is sensitive to duplicates, so we use identity sampler
124   sampler = IdentitySampler()
125
126   trainer_grid = {
127     # smaller C = stronger regularization
128     C: LogisticRegressionTrainer(C=C)
129     for C in [
130       0.01,
131       0.05,
132       0.1,
133       0.5,
134       1.0,
135       2.0,
136       5.0,
137       10.0,
138       20.0,
139       50.0,
140       100.0,
141       200.0,
142       500.0,
143       1000.0,
144       10_000.0,
145       100_000.0,
146       1_000_000.0,
147     ]
148   }
149
150   if not skip_cross_validation:

```

```

151     # Tune hyperparameters via grid search cross-validation
152     eval_results, (best_name, best_trainer), test_metrics, _ = (
153         grid_search_cross_validation(
154             data=trainable_dataset,
155             trainer_grid=trainer_grid,
156             # loss: lower is better
157             fn_find_best_metric=lambda m: min(m.keys(), key=lambda t: m[t].loss),
158             k_folds=5,
159             test_split=0.2,
160             sampler=sampler,
161             transformer=transformer,
162             fn_augment=None,
163             seed=SEED,
164         )
165     )
166
167     # Manual override to select middle of indifferent Cs
168     best_name = 10.0
169     best_trainer = trainer_grid[best_name]
170
171     # Tune decision-threshold via grid search cross-validation
172     thresholds_grid = list(np.arange(0.05, 0.96, 0.01))
173     eval_results_dt, prob_distributions, best_threshold, test_metrics_dt = (
174         grid_search_decision_threshold(
175             data=trainable_dataset,
176             trainer=best_trainer,
177             thresholds_grid=thresholds_grid,
178             fn_find_best_metric=lambda m: max(
179                 m.keys(),
180                 key=lambda t: m[t].accuracy, # accuracy: higher is better
181             ),
182             k_folds=5,
183             test_split=0.2,
184             sampler=sampler,
185             transformer=transformer,
186             fn_augment=None,
187             seed=SEED,
188         )
189     )
190
191     # Manual override to select reasonable threshold
192     best_threshold = 0.5
193
194     # Save eval results
195     EVAL_RESULTS_PATHS[model].write_text(
196         json.dumps(
197             {
198                 "hyperparameter_tuning": {
199                     name: eval_results[name].model_dump()
200                     for name in eval_results.keys()
201                 },
202                 "best_hyperparameters": best_name,
203                 "test_metrics": test_metrics.model_dump(),
204                 "decision_threshold_tuning": {
205                     thresh: eval_results_dt[thresh].model_dump()
206                     for thresh in eval_results_dt.keys()
207                 },
208                 "best_decision_threshold": best_threshold,
209                 "test_metrics_dt": test_metrics_dt.model_dump(),
210                 "prob_distributions_cv": {
211                     int(cls): [probs.tolist() for probs in fold_probs]
212                     for cls, fold_probs in prob_distributions.items()
213                 },
214             }
215         )
216     )
217
218 else:
219     print("Skipping cross-validation as requested.")
220
221     # hard coded from previous grid searches
222     best_name = 10.0
223     best_trainer = trainer_grid[best_name]
224
225 if train_ensemble:
226     # Ensemble by averaging predictions from 5 models with different seeds
227     print("Ensembling via averaging predictions from 5 models...")

```

```

228
229     ensemble_trainer = EnsembleTrainer(
230         submodels=[LogisticRegressionTrainer(C=best_name) for _ in range(5)],
231         bootstrap_samples=True, # needed for classical models
232         stratified_bootstrap=True,
233     )
234
235     else:
236         print("Not ensembling, using best trainer directly.")
237         ensemble_trainer = best_trainer
238
239     # Final validation through cross validation
240     # Validating ensemble performance (average over 5 folds)
241     evaluator = BasicCrossValidationEvaluator(
242         loss_func="cross_entropy", track_prob_distributions=True
243     )
244     k_fold_cross_validation(
245         data=trainable_dataset,
246         index_subset=trainable_dataset.original_indexes, # all data
247         trainer=ensemble_trainer,
248         k=5,
249         sampler=sampler,
250         transformer=transformer,
251         fn_augment=None,
252         evaluator=evaluator,
253         seed=SEED,
254         print_folds=True,
255     )
256
257     # Train final model on all data for best generalization
258     set_seed(SEED)
259     ensemble_trainer.reset_model()
260
261     # Use full trainable dataset
262     X, y = trainable_dataset.X, trainable_dataset.y
263
264     X_sampled, y_sampled = sampler.sample(X, y, seed=SEED)
265     class_counts = np.bincount(y_sampled)
266
267     # Standardize features (typically done after up/down sampling)
268     transformer.reset()
269     X_sampled_scaled = transformer.fit_transform(X_sampled)
270
271     ensemble_trainer.fit(X_sampled_scaled, y_sampled)
272     print(f"Trained final model on {X.shape[0]} samples [{class_counts}]")
273
274     # ----- PERSIST ARTIFACTS ----- #
275
276     print("Saving trained artifacts...")
277     MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION.mkdir(parents=True, exist_ok=True)
278     transformer.save(TRANSFORMER_PATHS[model])
279     ensemble_trainer.save_model(MODEL_PATHS[model])
280
281
282     # ----- CNN ----- #
283     class PytorchTrainerConfig(BaseModel):
284         """Configuration for PyTorch trainer."""
285
286         target_image_size: int = 64
287         num_epochs: int = 4
288         batch_size: int = 32
289         lr: float = 0.001
290         conv_channels: list[int] = [64, 32]
291
292
293     def build_trainer(config: PytorchTrainerConfig) -> PytorchTrainer:
294         """Build a PyTorch trainer based on the given configuration."""
295
296         def model_factory() -> nn.Module:
297             return CNNClassifier(
298                 in_channels=3,
299                 num_classes=2,
300                 conv_channels=config.conv_channels,
301             )
302
303         return PytorchTrainer(
304             model_factory=model_factory,

```

```

305     optimizer_factory=lambda m: optim.Adam(m.parameters(), lr=config.lr),
306     criterion=nn.CrossEntropyLoss(),
307     config=PytorchTrainingConfig(
308         num_epochs=config.num_epochs,
309         batch_size=config.batch_size,
310         device=get_device(),
311     ),
312 )
313
314
315 def fit_cnn_model(
316     data_path: Path = DATA_DIR,
317     skip_cross_validation: bool = True,
318     train_ensemble: bool = True, # bumps from ~99% - 100%
319 ):
320     """Fit CNN model for disaster type prediction."""
321     model = "cnn"
322
323     # Load the data
324     data = load_disaster_data(path=data_path)
325
326     # Preprocess the data, overwrite to save memory
327     data = preprocess_data(data)
328
329     # ----- FEATURE ENGINEERING ----- #
330
331     target_image_size = 64
332     trainable_dataset = feature_pipeline_flood_fire_neural_networks(
333         data,
334         nn_config=NNConfig(
335             image_size=target_image_size,
336             in_channels=3,
337             num_classes=2,
338         ),
339     )
340     del data
341
342     def fn_augment(X_tr, y_tr) -> tuple[torch.Tensor, torch.Tensor]:
343         X_tr, y_tr = apply_augmentations(
344             images=convert_to_image_tensor(X_tr).numpy(),
345             labels=y_tr.numpy(),
346             augmentations=CNN_AUGMENTATIONS,
347         )[2:]
348         return convert_to_trainable_tensor(torch.tensor(X_tr)), torch.tensor(y_tr)
349
350     # ----- TRAINING ----- #
351
352     print("Training CNN model...")
353
354     # CNNs don't need standardization
355     transformer = IdentityTransformer()
356     sampler = BalancedUpsampler()
357
358     if not skip_cross_validation:
359         trainer_config_grid = {
360             repr(
361                 (
362                     num_epochs,
363                     batch_size,
364                     lr,
365                     conv_channels,
366                 )
367             ): PytorchTrainerConfig(
368                 target_image_size=target_image_size,
369                 num_epochs=num_epochs,
370                 batch_size=batch_size,
371                 lr=lr,
372                 conv_channels=conv_channels,
373             )
374             for (num_epochs, batch_size, lr, conv_channels) in (
375                 [
376                     (num_epochs, batch_size, lr, conv_channels)
377                     for num_epochs in [8, 12, 16, 32]
378                     for batch_size in [16, 32, 64]
379                     for lr in [0.0005, 0.0001, 0.0005]
380                     for conv_channels in [[64, 32], [128, 64], [64, 64, 32]]
381                 ]
382             )
383         }

```

```

382         )
383     }
384
385     trainer_grid = {
386         name: build_trainer(config) for name, config in trainer_config_grid.items()
387     }
388
389     eval_results, (best_name, best_trainer), test_metrics, prob_distributions = (
390         grid_search_cross_validation(
391             data=trainable_dataset,
392             trainer_grid=trainer_grid,
393             fn_find_best_metric=lambda m: min(m.keys(), key=lambda t: m[t].loss),
394             k_folds=5,
395             # no separate test holdout needed for decision threshold tuning
396             test_split=None,
397             sampler=sampler,
398             transformer=transformer,
399             fn_augment=fn_augment,
400             seed=SEED,
401         )
402     )
403     best_trainer_config = trainer_config_grid[best_name]
404
405     # Save eval results
406     eval_results_dict = {
407         name: eval_results[name].model_dump() for name in eval_results.keys()
408     }
409     if prob_distributions:
410         eval_results_dict["prob_distributions_cv"] = {
411             int(cls): [probs.tolist() for probs in fold_probs]
412             for cls, fold_probs in prob_distributions.items()
413         }
414     EVAL_RESULTS_PATHS[model].write_text(json.dumps(eval_results_dict))
415
416     # Save best trainer config
417     TRAINER_CONFIG_PATHS[model].write_text(best_trainer_config.model_dump_json())
418
419 else:
420     print("Skipping cross-validation as requested.")
421
422     # Load best trainer config from previous grid search
423     best_trainer_config = PytorchTrainerConfig.model_validate_json(
424         TRAINER_CONFIG_PATHS[model].read_text()
425     )
426     best_trainer = build_trainer(best_trainer_config)
427
428 if train_ensemble:
429     # Ensemble by averaging predictions from 5 models with different seeds
430     print("Ensembling via averaging predictions from 5 models...")
431
432     ensemble_trainer = EnsembleTrainer(
433         submodels=[build_trainer(best_trainer_config) for _ in range(5)],
434         bootstrap_samples=False, # seeding is usually enough for deep NN
435     )
436
437 else:
438     print("Not ensembling, using best trainer directly.")
439     ensemble_trainer = best_trainer
440
441 # Final validation through cross validation
442 # Validating ensemble performance (average over 5 folds)
443 evaluator = BasicCrossValidationEvaluator(
444     loss_func="cross_entropy", track_prob_distributions=True
445 )
446 k_fold_cross_validation(
447     data=trainable_dataset,
448     index_subset=trainable_dataset.original_indexes, # all data
449     trainer=ensemble_trainer,
450     k=5,
451     sampler=sampler,
452     transformer=transformer,
453     fn_augment=None,
454     evaluator=evaluator,
455     seed=SEED,
456     print_folds=True,
457 )
458

```

```

459     # Train final model on all data for best generalization
460     set_seed(SEED)
461     ensemble_trainer.reset_model()
462
463     # Use full trainable dataset
464     X, y = trainable_dataset.X, trainable_dataset.y
465
466     # Apply augmentations to all data
467     X, y = fn_augment(X, y)
468
469     # Sampling to balance classes
470     # X_sampled, y_sampled = sampler.sample(X, y, seed=SEED)
471     X_sampled, y_sampled = X, y
472     class_counts = np.bincount(y_sampled)
473
474     # Standardize features (typically done after up/down sampling)
475     transformer.reset()
476     X_sampled_scaled = transformer.fit_transform(X_sampled)
477
478     ensemble_trainer.fit(X_sampled_scaled, y_sampled, X_val=None, y_val=None)
479     print(f"Trained final model on {X.shape[0]} samples [{class_counts}]")
480
481     # ----- PERSIST ARTIFACTS -----
482
483     print("Saving trained artifacts...")
484     MODEL_ARTIFACTS_DIR_DISASTER_PREDICTION.mkdir(parents=True, exist_ok=True)
485     transformer.save(TRANSFORMER_PATHS[model])
486     ensemble_trainer.save_model(MODEL_PATHS[model])
487
488     # ----- PREDICT -----
489
490     # ----- LOGISTIC -----
491
492
493     # ----- FEATURE ENGINEERING -----
494
495
496     def predict_logistic_regression(data_path: Path = DATA_DIR):
497         """Predict disaster types using logistic regression model."""
498         model = "logistic_regression"
499
500         # Load model
501         print("Loading trained artifacts...")
502         transformer = StandardScalerTransformer()
503         transformer.load(TRANSFORMER_PATHS[model])
504         decision_threshold = 0.5 # Tuned in training
505
506         pred_model = EnsembleTrainer(
507             submodels=[LogisticRegressionTrainer() for _ in range(5)],
508             bootstrap_samples=True,
509             stratified_bootstrap=True,
510         )
511         pred_model.load_model(MODEL_PATHS[model])
512
513         # Load holdout data
514         print("Loading holdout data...")
515         data = load_disaster_data(path=data_path, split="test")
516
517         # Preprocess holdout data, overwrite to save memory
518         data = preprocess_data(data)
519
520         # ----- RUN INFERENCE -----
521
522         predictable_dataset = feature_pipeline_flood_fire_logistic_regression(data)
523
524         # Training was on balanced data, but output needs to reflect original class imbalance
525         y_prob_holdout = pred_model.predict_proba(X_holdout_scaled)[:, 1]
526         y_pred_holdout = (y_prob_holdout >= decision_threshold).astype(np.uint8)
527
528         # Save results to CSV
529         save_predictions(
530             y_pred_holdout,

```

```

536         DISASTER_TYPE_PREDICTIONS_PATH.with_name(
537             f"{DISASTER_TYPE_PREDICTIONS_PATH.stem}_{model}.csv",
538         ),
539     )
540     print("Holdout predictions saved")
541
542 # ----- CNN ----- #
543
544
545 def predict_cnn(data_path: Path = DATA_DIR):
546     """Predict disaster types using CNN model."""
547     model = "cnn"
548
549     # Load model
550     print("Loading trained artifacts...")
551     transformer = IdentityTransformer()
552     transformer.load(TRANSFORMER_PATHS[model])
553
554     best_trainer_config = PytorchTrainerConfig.model_validate_json(
555         TRAINER_CONFIG_PATHS[model].read_text()
556     )
557     pred_model = EnsembleTrainer(
558         submodels=[build_trainer(best_trainer_config) for _ in range(5)],
559         bootstrap_samples=False, # seeding is usually enough for deep NN
560     )
561     pred_model.load_model(MODEL_PATHS[model])
562
563     # Load holdout data
564     print("Loading holdout data...")
565     data = load_disaster_data(path=data_path, split="test")
566
567     # Preprocess holdout data, overwrite to save memory
568     data = preprocess_data(data)
569
570 # ----- FEATURE ENGINEERING ----- #
571
572 predictable_dataset = feature_pipeline_flood_fire_neural_networks(
573     data,
574     nn_config=NNConfig(
575         image_size=64,
576         in_channels=3,
577         num_classes=2,
578     ),
579 )
580
581 # ----- RUN INFERENCE ----- #
582
583 X_holdout = predictable_dataset.X
584 X_holdout_scaled = transformer.transform(X_holdout)
585
586 # TODO: test time augmentation
587 y_prob_holdout = pred_model.predict_proba(X_holdout_scaled)[:, 1]
588 y_pred_holdout = (y_prob_holdout >= 0.5).astype(np.uint8)
589
590 # Save results to CSV
591 save_predictions(
592     y_pred_holdout,
593     DISASTER_TYPE_PREDICTIONS_PATH.with_name(
594         f"{DISASTER_TYPE_PREDICTIONS_PATH.stem}_{model}.csv",
595     ),
596 )
597     print("Holdout predictions saved")
598
599
600 # ----- CLI ----- #
601 # -----
602 # -----
603 # -----
604
605 @app.command()
606 def fit(
607     data_path: Path = DATA_DIR,
608     model: DisasterTypeModel = "logistic_regression",
609     skip_cross_validation: bool = True,
610 ):
611     """Fit disaster type prediction model."""

```

```
613
614     if model == "logistic_regression":
615         fit_logistic_regression(
616             data_path=data_path, skip_cross_validation=skip_cross_validation
617         )
618     else:
619         fit_cnn_model(data_path=data_path, skip_cross_validation=skip_cross_validation)
620
621
622 @app.command()
623 def predict(
624     data_path: Path = DATA_DIR,
625     model: DisasterTypeModel = "logistic_regression",
626 ):
627     """Predict disaster types using trained model."""
628
629     if model == "logistic_regression":
630         predict_logistic_regression(data_path=data_path)
631     else:
632         predict_cnn(data_path=data_path)
633
634
635 if __name__ == "__main__":
636     warnings.filterwarnings("ignore", category=UserWarning)
637
638     print("Using device:", get_device())
639     app()
640
```

damage_prediction.py

```
1  """
2  Training and prediction pipeline for damage level prediction.
3
4  Run via `pixi run damage-prediction --help`
5
6  E.g.:
7
8  # With hyper-parameter tuning (model selection: logistic_regression or cnn)
9  pixi run damage-prediction fit \
10    --model logistic_regression \
11    --no-skip-cross-validation
12
13 # For prediction on unlabeled test set
14 pixi run damage-prediction predict \
15    --model logistic_regression
16 """
17
18 import json
19 from pathlib import Path
20 from typing import Literal, get_args
21
22 import numpy as np
23 import torch
24 import torch.nn as nn
25 import torch.optim as optim
26 import typer
27 from data200.const.const import (
28     CNN_AUGMENTATIONS,
29     DAMAGE_LEVEL_PREDICTIONS_PATH,
30     DATA_DIR,
31     MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION,
32 )
33 from data200.evaluate.cross_validation import k_fold_cross_validation
34 from data200.evaluate.evaluator import BasicCrossValidationEvaluator
35 from data200.evaluate.grid_search import (
36     grid_search_cross_validation,
37 )
38 from data200.evaluate.loss import SoftF1Loss
39 from data200.feature.transform import (
40     IdentityTransformer,
41     StandardScalerTransformer,
42     convert_to_image_tensor,
43     convert_to_trainable_tensor,
44 )
45 from data200.io.load import (
46     load_disaster_data,
47 )
48 from data200.io.save import save_predictions
49 from data200.model.classic.lightgbm import LightGBMTrainer
50 from data200.model.classic.logistic import LogisticRegressionTrainer
51 from data200.model.nn.architecture.cnn import CNNClassifier
52 from data200.model.nn.architecture.mlp import MLPClassifier
53 from data200.model.nn.ensemble import EnsembleTrainer
54 from data200.model.nn.pytorch import PytorchTrainer, PytorchTrainingConfig
55 from data200.pipeline.config import NNConfig
56 from data200.pipeline.feature import (
57     feature_pipeline_dinov2,
58     feature_pipeline_hurricane_logistic_regression,
59     feature_pipeline_hurricane_neural_networks,
60 )
61 from data200.pipeline.preprocess import preprocess_data
62 from data200.preprocess.augmentation import apply_augmentations
63 from data200.sample.sample import BalancedUpsampler
64 from data200.utils.device import get_device
65 from data200.utils.utils import set_seed
66 from pydantic import BaseModel
67
68 app = typer.Typer()
69
70 DamageLevelModel = Literal["logistic_regression", "lightgbm", "dinov2+mlp", "cnn"]
71 DAMAGE_LEVEL_MODELS = list(get_args(DamageLevelModel))
72
73 TRANSFORMER_PATHS: dict[DamageLevelModel, Path] = {
```

```

74     model: MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION / f"{model}_scaler.json"
75     for model in DAMAGE_LEVEL_MODELS
76   }
77 TRAINER_CONFIG_PATHS = {
78   model: MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION / f"{model}_trainer_config.json"
79   for model in DAMAGE_LEVEL_MODELS
80 }
81 MODEL_PATHS = {
82   model: MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION / f"{model}_model.json"
83   for model in DAMAGE_LEVEL_MODELS
84 }
85 EVAL_RESULTS_PATHS = {
86   model: MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION
87   / f"{model}_grid_search_eval_results.json"
88   for model in DAMAGE_LEVEL_MODELS
89 }
90
91 SEED = 42
92
93
94 def serialize_prob_distributions(prob_dist: dict) -> dict:
95     """Convert prob_distributions dict to JSON-serializable format."""
96     if prob_dist is None:
97         return None
98     return {int(k): [arr.tolist() for arr in v] for k, v in prob_dist.items()}
99
100
101 # ----- #
102 #          TRAINING          #
103 # ----- #
104
105
106 class PytorchTrainerConfig(BaseModel):
107     """Configuration for PyTorch trainer."""
108
109     target_image_size: int = 64
110     num_epochs: int = 4
111     batch_size: int = 32
112     lr: float = 0.001
113
114     conv_channels: list[int] = [64, 32]
115     """Channels per conv layer in CNN model."""
116
117     hidden_layers: list[int] = [128]
118     """Hidden layer sizes for MLP model."""
119
120     num_classes: int = 4
121     loss: Literal[
122         "cross_entropy", "balanced_cross_entropy", "weighted_cross_entropy"
123     ] = "cross_entropy"
124
125
126     def build_trainer(
127         kind: Literal["dinov2+mlp", "cnn"],
128         config: PytorchTrainerConfig,
129         class_weights: torch.Tensor | None = None,
130     ) -> PytorchTrainer:
131         """Build a PyTorch trainer based on the given configuration."""
132
133     def model_factory() -> nn.Module:
134         if kind == "dinov2+mlp":
135             return MLPClassifier(
136                 in_channels=1,
137                 height=config.target_image_size,
138                 width=1,
139                 num_classes=config.num_classes,
140                 hidden_layers=config.hidden_layers,
141             )
142         elif kind == "cnn":
143             return CNNClassifier(
144                 in_channels=3,
145                 num_classes=config.num_classes,
146                 conv_channels=config.conv_channels,
147             )
148         else:
149             raise ValueError(f"Unknown model kind: {kind}")
150

```

```

151     device = get_device()
152
153     if config.loss == "cross_entropy":
154         criterion = nn.CrossEntropyLoss()
155     elif config.loss == "balanced_cross_entropy":
156         criterion = SoftF1Loss()
157     elif config.loss == "weighted_cross_entropy":
158         if class_weights is None:
159             raise ValueError(
160                 "class_weights must be provided for weighted_cross_entropy"
161             )
162         normalized_weights = class_weights / class_weights.sum()
163         criterion = nn.CrossEntropyLoss(weight=normalized_weights.float().to(device))
164     else:
165         raise ValueError(f"Unknown loss type: {config.loss}")
166
167     return PytorchTrainer(
168         model_factory=model_factory,
169         optimizer_factory=lambda m: optim.Adam(m.parameters(), lr=config.lr),
170         criterion=criterion,
171         config=PytorchTrainingConfig(
172             num_epochs=config.num_epochs, batch_size=config.batch_size, device=device
173         ),
174     )
175
176 # ----- LOGISTIC ----- #
177
178
179
180 def fit_logistic_regression(
181     data_path: Path = DATA_DIR,
182     skip_cross_validation: bool = True,
183     train_ensemble: bool = True, # Consistently +5-10% F1 score on every fold
184 ):
185     """Fit logistic regression model for damage level prediction."""
186
187     model = "logistic_regression"
188     print(f"Training {model} model...")
189
190     # Load the data
191     data = load_disaster_data(path=data_path)
192
193     # Preprocess the data, overwrite to save memory
194     data = preprocess_data(data)
195
196     # ----- FEATURE ENGINEERING ----- #
197
198     trainable_dataset = feature_pipeline_hurricane_logistic_regression(data)
199     del data
200
201     # ----- CROSS VALIDATION AND TRAINING ----- #
202
203     print(f"Training {model} model...")
204     transformer = StandardScalerTransformer()
205
206     # Even though logistic regression is sensitive to duplicate samples,
207     # empirically it seems to help with class imbalance here.
208     sampler = BalancedUpsampler()
209
210     trainer_grid = {
211         # smaller C = stronger regularization
212         C: LogisticRegressionTrainer(C=C, max_iter=1000)
213         for C in [
214             0.01,
215             0.05,
216             0.1,
217             0.5,
218             1.0,
219             2.0,
220             5.0,
221             10.0,
222             20.0,
223             50.0,
224             100.0,
225             200.0,
226             500.0,
227             1000.0,

```

```

228         10_000.0,
229         100_000.0,
230         1_000_000.0,
231     ]
232 }
233
234 if not skip_cross_validation:
235     # Tune hyperparameters via grid search cross-validation
236     eval_results, (best_name, best_trainer), _, prob_distributions = (
237         grid_search_cross_validation(
238             data=trainable_dataset,
239             trainer_grid=trainer_grid,
240             # f1: higher is better
241             fn_find_best_metric=lambda m: max(
242                 m.keys(), key=lambda t: m[t].f1_score_macro
243             ),
244             k_folds=5,
245             # 20% test / 20% val split removes too many samples
246             # of the minority classes (only 6 samples in class 2)
247             test_split=None,
248             sampler=sampler,
249             transformer=transformer,
250             fn_augment=None,
251             seed=42,
252         )
253     )
254
255     # Manual override
256     # best_name =
257     # best_trainer = trainer_grid[best_name]
258
259     # Save eval results
260     EVAL_RESULTS_PATHS[model].write_text(
261         json.dumps(
262             {
263                 "hyperparameter_tuning": {
264                     name: eval_results[name].model_dump()
265                     for name in eval_results.keys()
266                 },
267                 "best_hyperparameters": best_name,
268                 "test_metrics": None,
269                 "prob_distributions_cv": serialize_prob_distributions(
270                     prob_distributions
271                 ),
272             }
273         )
274     )
275
276 else:
277     print("Skipping cross-validation as requested.")
278
279     # hard coded from previous grid searches
280     best_name = 0.05
281     best_trainer = trainer_grid[best_name]
282
283 if train_ensemble:
284     # Ensemble by averaging predictions from 5 models with different seeds
285     print("Ensembling via averaging predictions from 5 models...")
286
287     ensemble_trainer = EnsembleTrainer(
288         submodels=[
289             LogisticRegressionTrainer(C=best_name, max_iter=1000) for _ in range(5)
290         ],
291         bootstrap_samples=True, # needed for classical models
292         stratified_bootstrap=True,
293     )
294
295 else:
296     print("Not ensembling, using best trainer directly.")
297     ensemble_trainer = best_trainer
298
299 # Final validation through cross validation
300 # Validating ensemble performance (average over 5 folds)
301 evaluator = BasicCrossValidationEvaluator(
302     loss_func="cross_entropy", track_prob_distributions=True
303 )
304 k_fold_cross_validation(

```

```

305     data=trainable_dataset,
306     index_subset=trainable_dataset.original_indexes, # all data
307     trainer=ensemble_trainer,
308     k=5,
309     sampler=sampler,
310     transformer=transformer,
311     fn_augment=None,
312     evaluator=evaluator,
313     seed=SEED,
314     print_folds=True,
315 )
316
317 # Train final model on all data for best generalization
318 # Use ensemble for final model
319 set_seed(SEED)
320 ensemble_trainer.reset_model()
321
322 # Use full trainable dataset
323 X, y = trainable_dataset.X, trainable_dataset.y
324
325 X_sampled, y_sampled = sampler.sample(X, y, seed=SEED)
326 class_counts = np.bincount(y_sampled)
327
328 # Standardize features (typically done after up/down sampling)
329 transformer.reset()
330 X_sampled_scaled = transformer.fit_transform(X_sampled)
331
332 ensemble_trainer.fit(X_sampled_scaled, y_sampled)
333 print(f"Trained final model on {X.shape[0]} samples [{class_counts}]")
334
335 # ----- PERSIST ARTIFACTS -----
336
337 print("Saving trained artifacts...")
338 MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION.mkdir(parents=True, exist_ok=True)
339 transformer.save(TRANSFORMER_PATHS[model])
340 ensemble_trainer.save_model(MODEL_PATHS[model])
341
342 # ----- LIGHTGBM -----
343
344
345 def fit_lightgbm(
346     data_path: Path = DATA_DIR,
347     skip_cross_validation: bool = True,
348     train_ensemble: bool = True, # Consistently +5-10% F1 score on every fold
349 ):
350     """Fit LightGBM model for damage level prediction."""
351
352     model = "lightgbm"
353     print(f"Training {model} model...")
354
355     # Load the data
356     data = load_disaster_data(path=data_path)
357
358     # Preprocess the data, overwrite to save memory
359     data = preprocess_data(data)
360
361     # ----- FEATURE ENGINEERING -----
362
363     trainable_dataset = feature_pipeline_hurricane_logistic_regression(data)
364     del data
365
366     # ----- CROSS VALIDATION AND TRAINING -----
367
368     print(f"Training {model} model...")
369     transformer = StandardScalerTransformer()
370
371     sampler = BalancedUpsampler()
372
373     trainer_grid = {
374         (
375             "lightgbm_" + str(params).strip().replace(" ", "").replace("'", ""),
376         ): LightGBMTrainer(num_classes=len(np.unique(trainable_dataset.y)), **params)
377     }
378     for params in [
379         {"objective": "multiclassova", "class_weight": "balanced"}, # balanced
380         {"objective": "multiclassova", "class_weight": None}, # None
381         *[ # [
```

```

382         {
383             "objective": "multiclassova",
384             "class_weight": "balanced",
385             "num_leaves": num_leaves,
386             "max_depth": max_depth,
387             "min_child_samples": min_child_samples,
388         }
389         for num_leaves in [31, 50, 63]
390         for max_depth in [6, 8, 10]
391         for min_child_samples in [1, 5, 10]
392     ],
393 ]
394 }
395
396 if not skip_cross_validation:
397     # Tune hyperparameters via grid search cross-validation
398     eval_results, (best_name, best_trainer), _, prob_distributions = (
399         grid_search_cross_validation(
400             data=trainable_dataset,
401             trainer_grid=trainer_grid,
402             # f1: higher is better
403             fn_find_best_metric=lambda m: max(
404                 m.keys(), key=lambda t: m[t].f1_score_macro
405             ),
406             k_folds=5,
407             # 20% test / 20% val split removes too many samples
408             # of the minority classes (only 6 samples in class 2)
409             test_split=None,
410             sampler=sampler,
411             transformer=transformer,
412             fn_augment=None,
413             seed=42,
414         )
415     )
416
417     # Manual override
418     # best_name =
419     # best_trainer = trainer_grid[best_name]
420
421     # Save eval results
422     EVAL_RESULTS_PATHS[model].write_text(
423         json.dumps(
424             {
425                 "hyperparameter_tuning": {
426                     name: eval_results[name].model_dump()
427                     for name in eval_results.keys()
428                 },
429                 "best_hyperparameters": best_name,
430                 "test_metrics": None,
431                 "prob_distributions_cv": serialize_prob_distributions(
432                     prob_distributions
433                 ),
434             }
435         )
436     )
437
438 else:
439     print("Skipping cross-validation as requested.")
440
441     # hard coded from previous grid searches
442     best_name = "lightgbm_{objective:multiclassova,class_weight:balanced}"
443     best_trainer = trainer_grid[best_name]
444
445 if train_ensemble:
446     # Ensemble by averaging predictions from 5 models with different seeds
447     print("Ensembling via averaging predictions from 5 models...")
448
449     ensemble_trainer = EnsembleTrainer(
450         submodels=[
451             LightGBMTrainer(
452                 num_classes=len(np.unique(trainable_dataset.y)),
453                 objective="multiclassova",
454                 class_weight="balanced",
455             )
456             for _ in range(5)
457         ],
458         bootstrap_samples=True, # needed for classical models

```

```

459         stratified_bootstrap=True,
460     )
461
462 else:
463     print("Not ensembling, using best trainer directly.")
464     ensemble_trainer = best_trainer
465
466 # Final validation through cross validation
467 # Validating ensemble performance (average over 5 folds)
468 evaluator = BasicCrossValidationEvaluator(
469     loss_func="cross_entropy", track_prob_distributions=True
470 )
471 k_fold_cross_validation(
472     data=trainable_dataset,
473     index_subset=trainable_dataset.original_indexes, # all data
474     trainer=ensemble_trainer,
475     k=5,
476     sampler=sampler,
477     transformer=transformer,
478     fn_augment=None,
479     evaluator=evaluator,
480     seed=SEED,
481     print_folds=True,
482 )
483
484 # Train final model on all data for best generalization
485 # Use ensemble for final model
486 set_seed(SEED)
487 ensemble_trainer.reset_model()
488
489 # Train final model on all data for best generalization
490 set_seed(SEED)
491 ensemble_trainer.reset_model()
492
493 # Use full trainable dataset
494 X, y = trainable_dataset.X, trainable_dataset.y
495
496 X_sampled, y_sampled = sampler.sample(X, y, seed=SEED)
497 class_counts = np.bincount(y_sampled)
498
499 # Standardize features (typically done after up/down sampling)
500 transformer.reset()
501 X_sampled_scaled = transformer.fit_transform(X_sampled)
502
503 ensemble_trainer.fit(X_sampled_scaled, y_sampled)
504 print(f"Trained final model on {X.shape[0]} samples [{class_counts}]")
505
506 # ----- PERSIST ARTIFACTS -----
507
508 print("Saving trained artifacts...")
509 MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION.mkdir(parents=True, exist_ok=True)
510 transformer.save(TRANSFORMER_PATHS[model])
511 ensemble_trainer.save_model(MODEL_PATHS[model])
512
513 # ----- DINOV2+MLP -----
514
515
516
517 def fit_mlp_dinov2(
518     data_path: Path = DATA_DIR,
519     skip_cross_validation: bool = True,
520     augment: bool = True,
521     train_ensemble: bool = True,
522 ):
523     """Fit dinov2+mlp model for damage level prediction."""
524     model = "dinov2+mlp"
525     print(f"Training {model} model on top of dinov2 embeddings...")
526
527     # Load the data
528     data = load_disaster_data(path=data_path)
529
530     # Preprocess the data, overwrite to save memory
531     data = preprocess_data(data)
532
533 # ----- FEATURE ENGINEERING -----
534
535 trainable_dataset = feature_pipeline_hurricane_dinov2(

```

```

536     dataset_dir=data_path, data=data, augment=augment
537 )
538 del data
539
540 # ----- CROSS VALIDATION AND TRAINING ----- #
541
542 sampler = BalancedUpsampler()
543 num_classes = len(np.unique(trainable_dataset.y))
544
545 # With standardization performs significantly better here than without
546 transformer = StandardScalerTransformer()
547
548 if not skip_cross_validation:
549     trainer_config_grid = {
550         repr(
551             (num_epochs, batch_size, lr, hidden_layers, loss)
552         ): PytorchTrainerConfig(
553             target_image_size=trainable_dataset.X.shape[1],
554             num_epochs=num_epochs,
555             batch_size=batch_size,
556             lr=lr,
557             hidden_layers=hidden_layers,
558             num_classes=num_classes,
559             loss=loss,
560         )
561         for (num_epochs, batch_size, lr, hidden_layers, loss) in (
562             [
563                 (num_epochs, batch_size, lr, hidden_layers, loss)
564                 for num_epochs in [8, 16, 32, 48]
565                 for batch_size in [32, 64]
566                 for lr in [0.0001, 0.00005]
567                 for hidden_layers in [[128], [128, 64], [256], [64, 32]]
568                 for loss in ["cross_entropy"]
569             ]
570         )
571     }
572
573     trainer_grid = {
574         name: build_trainer(
575             model, config, class_weights=torch.tensor([31, 70, 6, 93])
576         )
577         for name, config in trainer_config_grid.items()
578     }
579
580     eval_results, (best_name, best_trainer), _, prob_distributions = (
581         grid_search_cross_validation(
582             data=trainable_dataset,
583             trainer_grid=trainer_grid,
584             fn_find_best_metric=lambda m: max(
585                 m.keys(), key=lambda t: m[t].f1_score_macro
586             ),
587             k_folds=5,
588             # 20% test / 20% val split removes too many samples
589             # of the minority classes (only 6 samples in class 2)
590             test_split=None,
591             sampler=sampler,
592             transformer=transformer,
593             # Augmentation already applied before during feature pipeline
594             fn_augment=None,
595             seed=42,
596         )
597     )
598     best_trainer_config = trainer_config_grid[best_name]
599
600     # Save eval results
601     EVAL_RESULTS_PATHS[model].write_text(
602         json.dumps(
603             {
604                 "hyperparameter_tuning": {
605                     name: eval_results[name].model_dump()
606                     for name in eval_results.keys()
607                 },
608                 "best_hyperparameters": best_name,
609                 "prob_distributions_cv": serialize_prob_distributions(
610                     prob_distributions
611                 ),
612             }
613         )
614     )

```

```

613         )
614     )
615
616     # Save best trainer config
617     TRAINER_CONFIG_PATHS[model].write_text(best_trainer_config.model_dump_json())
618
619 else:
620     print("Skipping cross-validation as requested.")
621
622     # Load best trainer config from previous grid search
623     best_trainer_config = PytorchTrainerConfig.model_validate_json(
624         TRAINER_CONFIG_PATHS[model].read_text()
625     )
626     best_trainer = build_trainer(model, best_trainer_config)
627
628 if train_ensemble:
629     # Ensemble by averaging predictions from 5 models with different seeds
630     print("Ensembling via averaging predictions from 5 models...")
631
632     ensemble_trainer = EnsembleTrainer(
633         submodels=[build_trainer(model, best_trainer_config) for _ in range(5)],
634         bootstrap_samples=True,
635         stratified_bootstrap=True,
636     )
637
638 else:
639     print("Not ensembling, using best trainer directly.")
640     ensemble_trainer = best_trainer
641
642     # Final validation through cross validation
643     # Validating ensemble performance (average over 5 folds)
644     evaluator = BasicCrossValidationEvaluator(
645         loss_func="cross_entropy", track_prob_distributions=True
646     )
647     k_fold_cross_validation(
648         data=trainable_dataset,
649         index_subset=trainable_dataset.original_indexes, # all data
650         trainer=ensemble_trainer,
651         k=5,
652         sampler=sampler,
653         transformer=transformer,
654         fn_augment=None,
655         evaluator=evaluator,
656         seed=SEED,
657         print_folds=True,
658     )
659
660     # Train final model on all data for best generalization
661     set_seed(SEED)
662     ensemble_trainer.reset_model()
663
664     # Use full trainable dataset
665     X, y = trainable_dataset.X, trainable_dataset.y
666
667     # No additional data augmentation
668
669     # Sampling to balance classes
670     X_sampled, y_sampled = sampler.sample(X, y, seed=SEED)
671     class_counts = np.bincount(y_sampled)
672
673     # Standardize features (typically done after up/down sampling)
674     transformer.reset()
675     X_sampled_scaled = transformer.fit_transform(X_sampled)
676
677     ensemble_trainer.fit(X_sampled_scaled, y_sampled, X_val=None, y_val=None)
678     print(
679         f"Trained final model on {X_sampled_scaled.shape[0]} samples; [{class_counts}]"
680     )
681
682     # ----- PERSIST ARTIFACTS -----
683
684     print("Saving trained artifacts...")
685     MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION.mkdir(parents=True, exist_ok=True)
686     transformer.save(TRANSFORMER_PATHS[model])
687     ensemble_trainer.save_model(MODEL_PATHS[model])
688
689

```

```

690  # ----- CNN ----- #
691
692
693 def fit_cnn_model(
694     data_path: Path = DATA_DIR,
695     skip_cross_validation: bool = True,
696     train_ensemble: bool = True, # Consistently +1-2% F1 score average, better stability with bootstrap sampling
697 ):
698     """Fit CNN model for damage level prediction."""
699
700     model = "cnn"
701
702     # Load the data
703     data = load_disaster_data(path=data_path)
704
705     # Preprocess the data, overwrite to save memory
706     data = preprocess_data(data)
707
708     # ----- FEATURE ENGINEERING ----- #
709
710     target_image_size = 64
711     trainable_dataset = feature_pipeline_hurricane_neural_networks(
712         data,
713         nn_config=NNConfig(
714             image_size=target_image_size,
715             in_channels=3,
716             num_classes=2,
717         ),
718     )
719     del data
720
721     def fn_augment(X_tr, y_tr) -> tuple[torch.Tensor, torch.Tensor]:
722         X_tr, y_tr = apply_augmentations(
723             images=convert_to_image_tensor(X_tr).numpy(),
724             labels=y_tr.numpy(),
725             augmentations=CNN_AUGMENTATIONS,
726         )[2:]
727         return convert_to_trainable_tensor(torch.tensor(X_tr)), torch.tensor(y_tr)
728
729     # ----- TRAINING ----- #
730
731     print("Training CNN model...")
732
733     # CNNs don't need standardization
734     transformer = IdentityTransformer()
735     sampler = BalancedUpsampler()
736
737     if not skip_cross_validation:
738         trainer_config_grid = {
739             repr(
740                 (
741                     num_epochs,
742                     batch_size,
743                     lr,
744                     conv_channels,
745                 )
746             ): PytorchTrainerConfig(
747                 target_image_size=target_image_size,
748                 num_epochs=num_epochs,
749                 batch_size=batch_size,
750                 lr=lr,
751                 conv_channels=conv_channels,
752             )
753             for (num_epochs, batch_size, lr, conv_channels) in (
754                 [
755                     (num_epochs, batch_size, lr, conv_channels)
756                     for num_epochs in [8, 12, 16, 32]
757                     for batch_size in [16, 32, 64]
758                     for lr in [0.0005, 0.0001, 0.0005]
759                     for conv_channels in [[64, 32], [128, 64], [64, 64, 32]]
760                 ]
761             )
762         }
763
764         trainer_grid = {
765             name: build_trainer(model, config)
766             for name, config in trainer_config_grid.items()
767         }

```

```

767     }
768
769     eval_results, (best_name, best_trainer), _, prob_distributions =
770         grid_search_cross_validation(
771             data=trainable_dataset,
772             trainer_grid=trainer_grid,
773             fn_find_best_metric=lambda m: max(
774                 m.keys(), key=lambda t: m[t].f1_score_macro
775             ),
776             k_folds=5,
777             # 20% test / 20% val split removes too many samples
778             # of the minority classes (only 6 samples in class 2)
779             test_split=None,
780             sampler=sampler,
781             transformer=transformer,
782             fn_augment=fn_augment,
783             seed=42,
784         )
785     )
786     best_trainer_config = trainer_config_grid[best_name]
787
788     # Save eval results
789     EVAL_RESULTS_PATHS[model].write_text(
790         json.dumps(
791             {
792                 "hyperparameter_tuning": {
793                     name: eval_results[name].model_dump()
794                     for name in eval_results.keys()
795                 },
796                 "best_hyperparameters": best_name,
797                 "prob_distributions_cv": serialize_prob_distributions(
798                     prob_distributions
799                 ),
800             }
801         )
802     )
803
804     # Save best trainer config
805     TRAINER_CONFIG_PATHS[model].write_text(best_trainer_config.model_dump_json())
806
807 else:
808     print("Skipping cross-validation as requested.")
809
810     # Load best trainer config from previous grid search
811     best_trainer_config = PytorchTrainerConfig.model_validate_json(
812         TRAINER_CONFIG_PATHS[model].read_text()
813     )
814     best_trainer = build_trainer(model, best_trainer_config)
815
816 if train_ensemble:
817     # Ensemble by averaging predictions from 5 models with different seeds
818     print("Ensembling via averaging predictions from 5 models...")
819
820     ensemble_trainer = EnsembleTrainer(
821         submodels=[build_trainer(model, best_trainer_config) for _ in range(5)],
822         bootstrap_samples=True,
823         stratified_bootstrap=True,
824     )
825
826 else:
827     print("Not ensembling, using best trainer directly.")
828     ensemble_trainer = best_trainer
829
830 # Final validation through cross validation
831 # Validating ensemble performance (average over 5 folds)
832 evaluator = BasicCrossValidationEvaluator(
833     loss_func="cross_entropy", track_prob_distributions=True
834 )
835 k_fold_cross_validation(
836     data=trainable_dataset,
837     index_subset=trainable_dataset.original_indexes, # all data
838     trainer=ensemble_trainer,
839     k=5,
840     sampler=sampler,
841     transformer=transformer,
842     fn_augment=None,
843     evaluator=evaluator,

```

```

844     seed=SEED,
845     print_folds=True,
846 )
847
848 # Train final model on all data for best generalization
849 # Use ensemble for final model
850 set_seed(SEED)
851 ensemble_trainer.reset_model()
852
853 # Use full trainable dataset
854 X, y = trainable_dataset.X, trainable_dataset.y
855
856 # Apply augmentations to all data
857 X, y = fn_augment(X, y)
858
859 # Sampling to balance classes
860 # X_sampled, y_sampled = sampler.sample(X, y, seed=SEED)
861 X_sampled, y_sampled = X, y
862 class_counts = np.bincount(y_sampled)
863
864 # Standardize features (typically done after up/down sampling)
865 transformer.reset()
866 X_sampled_scaled = transformer.fit_transform(X_sampled)
867
868 ensemble_trainer.fit(X_sampled_scaled, y_sampled, X_val=None, y_val=None)
869 print(f"Trained final model on {X.shape[0]} samples {class_counts}")
870
871 # ----- PERSIST ARTIFACTS -----
872
873 print("Saving trained artifacts...")
874 MODEL_ARTIFACTS_DIR_DAMAGE_PREDICTION.mkdir(parents=True, exist_ok=True)
875 transformer.save(TRANSFORMER_PATHS[model])
876 ensemble_trainer.save_model(MODEL_PATHS[model])
877
878
879 # ----- PREDICT -----
880
881 # ----- LOGISTIC -----
882
883 # ----- FEATURE ENGINEERING -----
884
885 def predict_logistic_regression(data_path: Path = DATA_DIR):
886     """Predict damage level using logistic regression model."""
887     model = "logistic_regression"
888
889     # Load model
890     print("Loading trained artifacts...")
891     transformer = StandardScalerTransformer()
892     transformer.load(TRANSFORMER_PATHS[model])
893
894     pred_model = EnsembleTrainer(
895         submodels=[LogisticRegressionTrainer() for _ in range(5)],
896         bootstrap_samples=True,
897         stratified_bootstrap=True,
898     )
899     pred_model.load_model(MODEL_PATHS[model])
900
901     # Load holdout data
902     print("Loading holdout data...")
903     data = load_disaster_data(path=data_path, split="test")
904
905     # Preprocess holdout data, overwrite to save memory
906     data = preprocess_data(data)
907
908     # ----- RUN INFERENCE -----
909
910     predictable_dataset = feature_pipeline_hurricane_logistic_regression(data)
911
912     # ----- RUN INFERENCE -----
913
914     X_holdout = predictable_dataset.X
915     X_holdout_scaled = transformer.fit_transform(X_holdout)
916     y_pred_holdout = pred_model.predict(X_holdout_scaled)
917
918     # Save results to CSV
919     save_predictions(

```

```

921     y_pred_holdout,
922     DAMAGE_LEVEL_PREDICTIONS_PATH.with_name(
923         f"{DAMAGE_LEVEL_PREDICTIONS_PATH.stem}_{model}.csv",
924     ),
925 )
926 print("Holdout predictions saved")
927
928
929 def predict_lightgbm(data_path: Path = DATA_DIR):
930     """Predict damage level using LightGBM model."""
931
932     model = "lightgbm"
933
934     # Load model
935     print("Loading trained artifacts...")
936     transformer = StandardScalerTransformer()
937     transformer.load(TRANSFORMER_PATHS[model])
938
939     pred_model = EnsembleTrainer(
940         submodels=[LightGBMTrainer(num_classes=4) for _ in range(5)],
941         bootstrap_samples=True,
942         stratified_bootstrap=True,
943     )
944     pred_model.load_model(MODEL_PATHS[model])
945
946     # Load holdout data
947     print("Loading holdout data...")
948     data = load_disaster_data(path=data_path, split="test")
949
950     # Preprocess holdout data, overwrite to save memory
951     data = preprocess_data(data)
952
953     # ----- FEATURE ENGINEERING -----
954
955     predictable_dataset = feature_pipeline_hurricane_logistic_regression(data)
956
957     # ----- RUN INFERENCE -----
958
959     X_holdout = predictable_dataset.X
960     X_holdout_scaled = transformer.fit_transform(X_holdout)
961     y_pred_holdout = pred_model.predict(X_holdout_scaled)
962
963     # Save results to CSV
964     save_predictions(
965         y_pred_holdout,
966         DAMAGE_LEVEL_PREDICTIONS_PATH.with_name(
967             f"{DAMAGE_LEVEL_PREDICTIONS_PATH.stem}_{model}.csv",
968         ),
969     )
970     print("Holdout predictions saved")
971
972
973 def predict_mlp_dinov2(data_path: Path = DATA_DIR):
974     """Predict damage level using dinov2+mlp model."""
975
976     model = "dinov2+mlp"
977
978     # Load model
979     print("Loading trained artifacts...")
980     transformer = StandardScalerTransformer()
981     transformer.load(TRANSFORMER_PATHS[model])
982
983     best_trainer_config = PytorchTrainerConfig.model_validate_json(
984         TRAINER_CONFIG_PATHS[model].read_text()
985     )
986     pred_model = EnsembleTrainer(
987         submodels=[
988             build_trainer(
989                 model, best_trainer_config, class_weights=torch.tensor([31, 70, 6, 93])
990             )
991             for _ in range(5)
992         ],
993         bootstrap_samples=True,
994         stratified_bootstrap=True,
995     )
996     pred_model.load_model(MODEL_PATHS[model])
997

```

```

998     # Load holdout data
999     print("Loading holdout data...")
1000    data = load_disaster_data(path=data_path, split="test")
1001
1002    # Preprocess holdout data, overwrite to save memory
1003    data = preprocess_data(data)
1004
1005    # ----- FEATURE ENGINEERING ----- #
1006
1007    predictable_dataset = feature_pipeline_hurricane_dinov2(
1008        dataset_dir=data_path, data=data, augment=False
1009    )
1010
1011    # ----- RUN INFERENCE ----- #
1012
1013    X_holdout = predictable_dataset.X
1014    X_holdout_scaled = transformer.transform(X_holdout)
1015
1016    y_pred_holdout = pred_model.predict(X_holdout_scaled)
1017
1018    # Save results to CSV
1019    save_predictions(
1020        y_pred_holdout,
1021        DAMAGE_LEVEL_PREDICTIONS_PATH.with_name(
1022            f"{DAMAGE_LEVEL_PREDICTIONS_PATH.stem}_{model}.csv",
1023        ),
1024    )
1025    print(
1026        f"Holdout predictions saved to {DAMAGE_LEVEL_PREDICTIONS_PATH.stem}_{model}.csv"
1027    )
1028
1029
1030 def predict_cnn(data_path: Path = DATA_DIR):
1031     """Predict damage level using CNN model."""
1032
1033     model = "cnn"
1034
1035     # Load model
1036     print("Loading trained artifacts...")
1037     transformer = IdentityTransformer()
1038     transformer.load(TRANSFORMER_PATHS[model])
1039
1040     best_trainer_config = PytorchTrainerConfig.model_validate_json(
1041         TRAINER_CONFIG_PATHS[model].read_text()
1042     )
1043     pred_model = EnsembleTrainer(
1044         submodels=[build_trainer(model, best_trainer_config) for _ in range(5)],
1045         bootstrap_samples=True,
1046         stratified_bootstrap=True,
1047     )
1048     pred_model.load_model(MODEL_PATHS[model])
1049
1050     # Load holdout data
1051     print("Loading holdout data...")
1052     data = load_disaster_data(path=data_path, split="test")
1053
1054     # Preprocess holdout data, overwrite to save memory
1055     data = preprocess_data(data)
1056
1057     # ----- FEATURE ENGINEERING ----- #
1058
1059     predictable_dataset = feature_pipeline_hurricane_neural_networks(
1060         data,
1061         nn_config=NNConfig(
1062             image_size=64,
1063             in_channels=3,
1064             num_classes=2,
1065         ),
1066     )
1067
1068     # ----- RUN INFERENCE ----- #
1069
1070     X_holdout = predictable_dataset.X
1071     X_holdout_scaled = transformer.transform(X_holdout)
1072
1073     # TODO: test time augmentation
1074     y_pred_holdout = pred_model.predict(X_holdout_scaled)76

```

```

1075     # Save results to CSV
1076     save_predictions(
1077         y_pred_holdout,
1078         DAMAGE_LEVEL_PREDICTIONS_PATH.with_name(
1079             f"{DAMAGE_LEVEL_PREDICTIONS_PATH.stem}_{model}.csv",
1080         ),
1081     ),
1082 )
1083 print("Holdout predictions saved")
1084
1085
1086 # ----- #
1087 # CLI #
1088 # ----- #
1089
1090
1091 @app.command()
1092 def fit(
1093     data_path: Path = DATA_DIR,
1094     model: DamageLevelModel = "logistic_regression",
1095     skip_cross_validation: bool = True,
1096 ):
1097     """Fit damage level prediction model."""
1098
1099     if model == "logistic_regression":
1100         fit_logistic_regression(
1101             data_path=data_path, skip_cross_validation=skip_cross_validation
1102         )
1103     elif model == "lightgbm":
1104         fit_lightgbm(data_path=data_path, skip_cross_validation=skip_cross_validation)
1105     elif model == "dinv2+mlp":
1106         fit_mlp_dinov2(data_path=data_path, skip_cross_validation=skip_cross_validation)
1107     elif model == "cnn":
1108         fit_cnn_model(data_path=data_path, skip_cross_validation=skip_cross_validation)
1109     else:
1110         raise NotImplementedError(f"Training for model {model} not implemented.")
1111
1112
1113 @app.command()
1114 def predict(
1115     data_path: Path = DATA_DIR, model: DamageLevelModel = "logistic_regression"
1116 ):
1117     """Predict damage levels using trained model."""
1118
1119     if model == "logistic_regression":
1120         predict_logistic_regression(data_path=data_path)
1121     elif model == "lightgbm":
1122         predict_lightgbm(data_path=data_path)
1123     elif model == "dinv2+mlp":
1124         predict_mlp_dinov2(data_path=data_path)
1125     elif model == "cnn":
1126         predict_cnn(data_path=data_path)
1127     else:
1128         raise NotImplementedError(f"Prediction for model {model} not implemented.")
1129
1130
1131 if __name__ == "__main__":
1132     print("Using device:", get_device())
1133     app()
1134

```