ChatGPT

Machine Learning Data Preparation and Annotation System for Fiber Optic End-Face Images

Overview

Fiber optic connector end-face inspection is crucial to ensure signal integrity, as even tiny defects (dust, scratches, pits, etc.) can cause signal loss or back-reflection and degrade performance [1]. We propose a modular **data preparation and annotation system** that processes a folder of fiber end-face images and produces labeled datasets for training machine learning models (segmentation and classification) in later steps. The system performs fully **unsupervised analysis** – detecting anomalies and clustering them into defect types without any initial labels. Defects are broadly categorized into **permanent defects** (e.g. scratches, cracks, pits/chips) and **contamination** (transient defects like dust, oil or residue) [2]. No model training is done in this stage – we only generate structured annotations and features ready for model training. The key components of the system include:

- **Zone Segmentation:** Identify the fiber's core, cladding, and ferrule contact regions in each image, producing zone masks and geometric metrics (core/cladding radii, core-to-cladding offset, etc.).
- **Unsupervised Defect Detection:** Use image processing and clustering to **detect anomalies** (scratches, digs/pits, blobs, contamination) in the image without pre-trained models or labels. Defect regions are segmented and characterized.
- **Feature Extraction:** Compute a rich set of features at pixel, defect-region, and whole-zone levels (intensity statistics, entropy, gradients, texture, morphological features, etc.). These features feed into the clustering and also provide data for classification datasets.
- **Defect Clustering & Labeling:** Apply unsupervised clustering on defect features to group similar defects. Assign each defect region a label (cluster ID or inferred type) and estimate severity metrics (e.g. size, length, intensity contrast).
- **Dataset Assembly:** Organize the outputs into multiple datasets for model training: (1) pixel-level segmentation labels for each image (region zones and defect classes per pixel), (2) region-level features with anomaly labels for each zone, and (3) image-level labels indicating overall defect presence/type.
- **Defect Library:** Collect all detected defect instances into a library (database or folder) indexed by defect location in image, defect type, severity, and source image reference. This library serves as a reference of defect examples (e.g. image patches of defects) for future analysis or model validation.

This system is implemented as a **modular Python codebase** with clear separation of tasks, configurable parameters, and thorough documentation. The following sections describe each component and the code organization in detail.

Directory Structure and Configuration

The codebase is organized into logical directories for inputs, outputs, and processing modules to maximize reusability and clarity. An example project structure is outlined below:

```
fiber_defect_inspection/
├── config.yaml                # Configuration file for parameters and paths
├── src/                       # Python source modules for each functionality
│   ├── main.py                # Orchestrates the end-to-end data preparation
│   ├── zone_segmentation.py   # Functions for core/cladding/ferrule
segmentation
│   ├── defect_detection.py    # Functions for unsupervised defect finding
│   ├── feature_extraction.py  # Functions to compute pixel, region, defect
features
│   ├── clustering.py          # Functions for clustering and labeling defects
│   ├── dataset_builder.py     # Functions to assemble and save datasets
│   └── utils.py               # Utility functions (image I/O, visualization,
etc.)
├── data/
│   ├── input_images/          # Folder of input fiber end-face images (to be
processed)
│   ├── output/                # Folder for all output datasets and results
│   │   ├── pixel_level/       # Pixel-level segmentation labels (e.g.
masks, .npy arrays)
│   │   ├── region_level/      # Region-level feature tables (e.g. CSV/Parquet
files)
│   │   ├── image_level/       # Image-level labels (e.g. CSV/Parquet summary)
│   │   └── segmentation_masks/ # Visualization outputs (e.g. overlaid defect
masks)
│   └── defect_library/        # Collected defect patches or info files
organized by type
└── README.md                  # Documentation for usage and module descriptions
```

- **Input:** The `input_images/` directory contains the raw end-face images to process. The system accepts a path to this folder (configured in `config.yaml` or via command-line argument). Images can be of various formats (JPEG, PNG, etc.), assumed to be consistent magnification and orientation.
- **Output:** All processed results are saved under `output/` in structured form. We create subfolders for each dataset type:
- `pixel_level/` for pixel-wise classification data (e.g. pairs of image and mask files, or NumPy arrays of image data and label masks).
- `region_level/` for zone-level data (tabular features and labels per zone, saved as `.csv` or `.parquet` for efficiency).
- `image_level/` for image-level labels (a summary CSV/Parquet mapping image filenames to overall defect labels or pass/fail status).

- `segmentation_masks/` for any diagnostic images (optional) such as images with zones outlined or defects highlighted, to visually verify the segmentation results.
- **Defect Library:** The `defect_library/` stores extracted defect instances. This could be organized into subfolders by defect type (e.g. `scratches/`, `pits/`, `contamination/`) each containing image patches of those defects, or a database/file that indexes all defects. Each defect entry in the library includes metadata: its image of origin, location (e.g. bounding box or mask), assigned type label, and severity. This library is useful for quickly retrieving examples of specific defect types and severities.
- **Configuration:** A central `config.yaml` (or JSON) file holds configurable parameters, making the pipeline flexible. Examples of configurations include: image processing thresholds, expected fiber dimensions (to guide zone detection), clustering parameters (number of clusters or distance thresholds), and file path settings. By adjusting `config.yaml`, users can tweak the defect detection sensitivity or feature set without modifying code.

All source modules in `src/` are documented with clear docstrings and organized by functionality, enabling reuse (for example, one could import the `zone_segmentation` module in a different script if needed). The `README.md` provides instructions on how to run the pipeline (e.g. `python src/main.py --config config.yaml`) and explains each output.

Image Preprocessing and Zone Segmentation

The first step is to **segment the end-face image into its key zones**: the fiber core, the cladding, and the ferrule contact surface around the fiber. Proper zone segmentation is essential because industry standards define different acceptance criteria for defects in different zones [3] [4]. We will identify: - **Zone A (Core):** the central fiber core region (small innermost circle, typically ~5–10 µm radius for SMF fibers). This zone is most critical; even tiny defects here cause failures [3].
- **Zone B (Cladding):** the ring of fiber cladding around the core (outer glass, ~125 µm diameter for standard fibers). Small defects in cladding are somewhat tolerable up to a limit [5].
- *(Optionally)* **Zone C (Adhesive):** a narrow ring just outside the cladding where epoxy bonding occurs inside the ferrule [6]. Often this zone is ignored in automated analysis.
- **Zone D (Ferrule Contact):** the outer ferrule surface area (polished ceramic around the fiber) [4]. Scratches or large debris here can matter if above certain size, but many small defects are acceptable on the ferrule.

Illustration of fiber end-face zones (A: Core, B: Cladding, C: Adhesive, D: Ferrule). Different defect size thresholds apply per zone in standard pass/fail analysis [3] [4].

Zone Detection Algorithm: For each image, we use image processing (likely via OpenCV or scikit-image) to find the circular fiber areas: - **Cladding Boundary:** The outer edge of the fiber cladding (boundary between Zone B and Zone D) usually appears as a strong circle in the image (the contrast between glass fiber and ceramic ferrule). We can detect this via a Hough Circle Transform or edge detection + circle fitting. This gives the fiber **center coordinates** and **cladding radius** in pixels [7]. The fiber center serves as the reference for defining all zones.
- **Core Region:** Depending on the imaging method, the fiber core may appear as a discernible inner circle. For example, with proper illumination the fiber core might be visible as a darker or brighter spot at the center [8] [9]. We attempt to detect the core by looking for an inner circle or intensity gradient at the center. Techniques include thresholding (if core is brighter/darker than cladding) or looking for a small

concentric circle using Hough Transform (with radius constraint based on expected core size). If the core is not directly visible, we can approximate the core zone as a circle of a fixed radius (from known fiber specs) centered at the fiber center.
- **Ferrule Region:** The ferrule contact zone (Zone D) is essentially the image area outside the cladding circle up to a certain radius. Typically, the field of view is limited, so we define the ferrule zone as the area between the cladding edge and the image border (or a fixed multiple of cladding radius). In practice, we may limit analysis to a diameter that covers the polished ferrule face that contacts other connectors.
- **Adhesive Zone:** If needed, define a narrow ring (e.g. 10 μm width annulus) just outside the cladding to mark the epoxy region [6], though this can be ignored in defect analysis per standards.

Using the detected center and radii, we create **binary masks** for each zone: e.g. `mask_core`, `mask_cladding`, `mask_ferrule` (and possibly `mask_adhesive`). These masks label pixels by zone membership. We also calculate **geometric metrics**: - **Core Radius & Cladding Radius:** measured in pixels (which can be converted to microns if calibration is known). This helps verify fiber geometry and identify any polishing issues (e.g., cladding diameter deviations).
- **Core-to-Cladding Offset:** the distance between the core center and cladding center. Ideally this is zero (core perfectly centered in cladding); any offset (eccentricity) beyond ~1–2 μm may indicate a manufacturing defect [10]. We compute this by finding the core circle center and subtracting the cladding circle center coordinates.
- **Concentricity Error:** (related to offset) could be reported as a percentage of core radius or absolute microns, as a quality metric.
- **Zone Area (pixel count):** the number of pixels in each zone mask (useful for normalization of defect sizes per zone).
- **Zone Boundaries (pixel locations):** the coordinates of the boundary curve for each zone (e.g., circle perimeter points). This can be stored if needed for visualization or further analysis (for instance, to check if the cladding boundary is perfectly circular or chipped).

All these values are saved for each image. The `zone_segmentation.py` module encapsulates this logic, providing a function like `segment_zones(image) -> masks, metrics` which returns the masks and a dictionary of metrics (radii, offset, etc.). These metrics will later be included in the region-level feature dataset.

Unsupervised Defect Detection and Segmentation

With zones identified, the system performs **unsupervised defect detection** – locating scratches, digs, blobs, and contamination on the fiber end-face **without any pre-labeled examples**. This relies on image processing and outlier detection techniques rather than supervised classifiers. The goal is to produce a **defect mask** or a set of defect region proposals for each image, which we can then label via clustering.

1. Preprocessing and Noise Reduction: First, we apply smoothing and normalization to reduce noise and illumination unevenness. For example, a **Gaussian blur** can remove high-frequency noise while preserving actual defect features [11]. We might also use background subtraction techniques: e.g., a **median filter** or morphological open/close to estimate the smooth background, then subtract it from the original to highlight anomalies (this is similar in spirit to the “difference of min-max ranking filter (DO2MR)” approach which enhances defect regions by removing smooth background variations [12] [13]). Uneven lighting often causes the image corners or ferrule to appear darker [14]; background normalization addresses this so that defects stand out by intensity.

2. Detecting Contaminants and Pits (Region-Based Defects): After preprocessing, we identify **region-based defects** such as dust particles, oil blotches, pits or chips on the fiber surface. These typically appear as isolated blobs or spots of differing intensity. We can use a combination of thresholding and morphological operations: - Compute a **residual image** = original - blurred (or use a high-pass filter) so that flat regions become ~0 and only local intensity deviations remain [13]. Defects will appear as bright or dark spots in the residual. - Apply an **adaptive threshold** or local peak detection on the residual to create a binary map of candidate defect pixels [15]. We might use Otsu's method or simply pick a threshold like mean + n·std of residual. The goal is to mark all potential defects, even if it over-segments (we will refine later). - Use **connected component analysis** on this binary map to group adjacent defect pixels into defect regions. Each region (a blob of connected pixels) is one defect candidate. We filter out too-small regions (could be noise) by size or area threshold. - For each region, calculate basic properties: area (pixel count), average intensity difference from background, and shape descriptors (e.g. bounding box, aspect ratio, circularity). This helps later differentiate types. Even tiny pits should be captured by this sensitive filtering [16] [17].

3. Detecting Scratches: Scratches are challenging because they are typically **elongated, low-contrast lines** across the surface [18]. They may not have a strong intensity difference like a dust speck, so the above blob detection might miss them. We take a separate approach tailored to linear features [19]: - Use **edge detection** (e.g. Canny filter) or a **directional filter** to highlight linear structures. We might apply a set of elongated kernel filters (line detectors) at various angles (0°, 45°, 90°, etc.) to produce responses where line-like features are present [20]. The "Linear Enhancement Inspector (LEI)" is an example of such a morphology-based scratch detector [21] – essentially accumulating contrast along a line element to boost scratch visibility [20]. - Another method: apply a **Sobel or Scharr gradient filter** to get gradient magnitude. Scratches often produce contiguous gradient along their length. Thin elongated connected gradients could indicate a scratch. - The result is a rough mask of linear features. We then use morphological thinning or skeletonization to get the centerline of scratches if needed, and connect broken segments if a scratch is discontinuous. - Group line pixels into line segments (e.g. Hough transform could also detect straight line segments if scratches are mostly straight). Compute each segment's length, width (perhaps using the thickness or blur around the line), and orientation.

4. Combining Defect Candidates: We now have two sets of defect regions: blob-like regions (contaminants, pits) and line-like regions (scratches). We merge these into a unified defect list. Some overlap might occur (e.g. a long scratch might have been partially detected in blob detection); we can union overlapping regions or associate line segments with blob regions if they intersect. Each defect candidate region at this stage has a mask or pixel list, and some preliminary features (size, shape, etc.). We assign each region an ID and also note **which zone** it falls in (by checking overlap with core/cladding/ferrule masks). Zone context is important for severity and later dataset labeling (e.g. a defect in Zone A vs Zone D).

This entire unsupervised detection runs **without human-provided labels**, instead relying on general assumptions about defect appearance (brightness difference or edge continuity). The output of this stage for each image is a **defect mask image** (all detected defect areas marked) and a **list of defect regions** with their geometric shapes. The `defect_detection.py` module implements this logic, providing functions like `detect_defects(image, core_mask, cladding_mask) -> defect_mask, defect_regions`. Each entry in `defect_regions` might be a dict containing coordinates, size, etc., as well as a placeholder for a defect label (to be filled by clustering later). We also calculate immediate **severity metrics** per defect here (before clustering), such as: - *Defect Area* (in μm^2 or pixel count). - *Defect Length* (for scratches). - *Defect Intensity Contrast*: difference between the defect region's intensity and the local background. - *Count of defect regions per zone*: e.g. how many in core, cladding, ferrule (could indicate overall cleanliness).

These metrics can feed into severity scoring (e.g. a scratch length of 200 µm is high severity, a dust speck of 1 µm is low severity) and will later be used in the feature dataset.

Feature Extraction for Zones and Defects

Once defects are detected and segmented, we compute a comprehensive set of **features** at multiple levels (pixel-level features, defect-region-level features, and zone-level aggregates). These features quantitatively describe the image and defect characteristics, and will be used both for clustering (unsupervised labeling) and for training data in the structured datasets.

Pixel-Level Feature Maps: Although we won't individually list features for every pixel in the final dataset (due to volume), pixel-level features are used internally for defect detection and could be saved for further analysis or model input. Examples include: - **Intensity**: Raw grayscale or color intensity at each pixel, possibly normalized. - **Gradient Magnitude and Orientation**: Per-pixel edge strength (from Sobel/Canny) and orientation which helped find scratches. - **Local Entropy**: Entropy in a small window around each pixel (high entropy might indicate texture or contamination). - **Local Binary Patterns (LBP)** or similar texture codes at each pixel for capturing micro-texture in the ferrule surface. - **Pixel's Zone**: We can produce a zone label map so each pixel "knows" whether it's in core, cladding, etc., which is effectively the zone mask.

These pixel-level maps are intermediate; we may store them as image files or arrays in the `features/` directory for debugging, but typically we will aggregate them into region and global features rather than outputting every pixel's data to the final user.

Defect Region Features: For each defect region identified (scratch or blob), we compute a rich feature vector: - **Size/Area:** in pixels and approximate microns^2 (using calibration if available). - **Shape Descriptors:** e.g. aspect ratio (major axis/minor axis of the region's bounding ellipse), roundness/circularity, solidity (area/convex hull area), and Hu moments to characterize shape. Scratches should show high aspect ratio (long and thin) whereas blobs (dirt) are more round.
- **Location Features:** the zone in which the defect lies (core/cladding/ferrule), distance of the defect from the fiber center (useful for contamination likely accumulating toward edges, etc.), and maybe angular position (for ferrule scratches orientation). - **Intensity/Contrast Features:** mean and standard deviation of pixel intensities in the defect vs the surrounding local background. For example, dust might appear dark on a bright background (negative contrast), whereas a pit may appear as a dark spot in core area. We capture min, max intensity in defect region, and maybe a signal-to-noise ratio of the defect spot. - **Texture Features:** if the defect region is large enough, compute texture metrics inside it like entropy, or gray-level co-occurrence matrix (GLCM) features (contrast, homogeneity) to distinguish, say, oily smear (which might have a texture) vs a hard particle. - **Gradient Features:** average gradient magnitude along the edges of the defect region (scratches will have edges along their length), or for scratches specifically, the straightness of the skeleton (a measure of how linear the defect is). - **Morphological Features:** number of separate sub-regions (if defect region is complex), length (for scratches, the length of skeleton or bounding box length), width (for scratches, maybe average width), orientation (angle of scratch relative to horizontal). For contamination, maybe count of holes in the region if any (some contamination might surround fiber leaving holes).

Each defect's feature vector is saved (likely in a list or temporary DataFrame) and later will be appended with the cluster label (defect type) after clustering. These serve as the training features for any future defect classification model.

Zone-Level Features: In addition to individual defects, we summarize features for each zone (core, cladding, ferrule) in an image: - **Cleanliness Metrics:** number of defect regions in the zone, total defect area fraction (defect area / zone area * 100%), largest defect size in zone, etc. - **Intensity/Uniformity:** average brightness of the zone, standard deviation of intensity in the zone (a very high std could indicate lots of texture or contamination versus a clean polished surface which is more uniform). - **Gradient/Texture in Zone:** e.g. average gradient magnitude in that zone (a scratched ferrule will have higher average gradient due to many edges), or entropy of the zone image. - **Core-Cladding Alignment:** for core zone specifically, the offset metric (in microns) to indicate alignment quality, which we computed earlier. - **Other Geometry:** fiber cladding radius measured (should be ~consistent if all good; any deviation might indicate the image is out of focus or the fiber end is chipped).

These zone features, combined with whether any defect exists in the zone, will allow training a model to predict pass/fail for that zone or classify if the zone is defective. We compile these into a **region-level table** later.

All feature computations are implemented in `feature_extraction.py`, which provides utility functions like `extract_defect_features(defect_region, image, masks)` and `compute_zone_features(image, masks, defect_list)`. This separation ensures we can easily adjust or add features. For example, if we decide to use a pre-trained CNN (like MobileNet) to extract an embedding for each defect patch for richer representation, we could plug that in here as an additional feature vector per defect (this could improve clustering of subtle defect patterns, using deep features [22]). The system is flexible to incorporate such changes via config (e.g., a flag to enable deep feature extraction).

Unsupervised Clustering for Defect Labeling

After feature extraction, we have a set of unlabeled defect instances (each with a feature vector). The next step is to **cluster these defect features** to group similar defects and infer their label (scratch, dig, contamination, etc.) automatically. Clustering is done across all defect instances from all images (assuming the folder may contain many images covering various defect types). The goal is to find natural groupings in feature space corresponding to defect categories.

Clustering Algorithm: We can use a clustering method like **k-means**, **DBSCAN**, or **Agglomerative clustering**: - *K-means:* If we roughly know how many defect categories we expect (e.g. 4 categories: {scratch, pit, contamination, other}), we could set k=4. However, this requires some guess and might force groupings even if data doesn't naturally cluster into exactly 4. - *DBSCAN:* This density-based clustering can find clusters of arbitrary shape and also identify outliers that don't belong to any cluster (which could be noise or novel defect types). It requires tuning a distance epsilon and minPoints. Useful if the number of defect instances is large and some defects are very distinct. - *Agglomerative (hierarchical):* This can give a dendrogram of how defects cluster, and one can cut it at a certain number of clusters or distance. It's more flexible but more computationally heavy if many instances.

For a first pass, we might choose k-means with k around 3–5 (the config can allow this to be adjusted). Since we know **scratches** vs **contamination** are distinct, and possibly **pits/chips** might form another cluster, we expect at least these groups. We run clustering on the scaled feature vectors of defects. Before clustering, feature normalization is done (each feature scaled to comparable range, e.g. zero-mean/unit-variance) so that no single feature (like area which might have large numeric values) dominates the distance metric.

Label Inference: Once clusters are formed, each defect region gets a cluster ID. We then **assign a descriptive label** to each cluster by interpreting the cluster's feature centroid: - One cluster may show high aspect ratio, large length, low width – we infer this cluster = **"Scratch"**. - Another cluster may show small area, near-core or on fiber, dark contrast – likely **"Pit/Dig"** in fiber. - Another cluster may have larger irregular areas, high entropy, often in ferrule zone – likely **"Contamination"** (dust/oil). - If a cluster appears but doesn't match known patterns (e.g., if it's maybe fiber **chips at the edge** or polishing residue), we label it accordingly or mark as "Other".

This mapping from cluster to label can either be done manually by inspecting a few examples from each cluster (since unsupervised clustering won't name them for us), or via simple rules on the features (as a sanity check). For instance, we can automatically designate the cluster with highest average aspect_ratio as scratches, the cluster with smallest average area as maybe pits, etc., based on known defect characteristics [2]. The system could output a small summary of cluster properties to help an engineer confirm the labeling.

Severity Assignment: Using cluster information and existing metrics, we assign each defect an approximate severity level. Severity can be a continuous metric (like size in microns, or scratch length) and/or a categorical tier (low/medium/high). For example: - For contamination blobs: severity could correlate with area (a dust speck of 1 μm^2 = low, a big smear covering 1000 μm^2 = high severity). - For scratches: severity might relate to length (long scratch across the core = very severe) or count (many small scratches). Also, location plays a role: a medium scratch in Zone A (core) could be marked higher severity than a longer scratch in Zone D (ferrule), because core scratches are more critical. We incorporate zone weighting into severity scoring. - We can define thresholds in config (e.g. >10 μm particle = severe for core/cladding, >50 μm scratch length = severe, etc., aligning with standards [4]).

Each defect entry thus gets fields: {cluster_label, defect_type, severity_score, severity_level}. This information is added to the `defect_regions` list for each image.

Finally, we update our **defect mask** for each image: instead of a generic "defect" mask, we can produce a **label mask** where different defects have different label values. For instance, label pixels belonging to scratches with code 1, contamination with code 2, etc. This creates a multi-class segmentation mask array for the image, which will be part of the pixel-level dataset.

The clustering and labeling logic resides in `clustering.py`. It might provide a function `label_defects_all_images(defect_feature_list) -> labeled_defects` which returns all defects with assigned labels, and possibly the cluster model itself (for analysis). By isolating clustering here, we could swap in a different unsupervised method in the future (even a one-class anomaly detector per defect type [23] if we had known-good images as reference).

Assembly of Training Datasets

With all images processed (zones segmented, defects detected and labeled, features computed), the system compiles the results into **structured datasets** ready for consumption by machine learning training pipelines. We create three main datasets as requested: pixel-level, region-level, and image-level. Each dataset is saved in an **optimized format** (NumPy arrays for images/masks, Parquet/CSV for tabular data, and organized directories for any image files) for efficient loading during model training.

Pixel-Level Segmentation Dataset

This dataset is for training convolutional neural networks to perform segmentation (pixel-wise classification). We need to pair each input image with a label mask of the same dimensions, where each pixel's value represents its class. Given the nature of our problem, we have a few ways to structure the classes: - **Option 1:** Multi-class mask that encodes both zone and defect information in one label. For example, we could assign codes: 0 = background (no defect on ferrule), 1 = core defect, 2 = cladding defect, 3 = ferrule defect, 4 = core clean, 5 = cladding clean, 6 = ferrule clean. However, this approach can be cumbersome (it mixes two concepts).
- **Option 2:** Two separate masks for each image: one for zone segmentation (classifying pixels as core/cladding/ferrule/background), and one for defect segmentation (classifying pixels as defect vs no defect, or even defect type categories). A model could be trained separately on each, or a multi-headed model could output both. This might be clearer to implement.
- **Option 3:** A combined multi-channel label: e.g., one channel image of zone labels, another channel of defect labels. Some segmentation frameworks allow multi-output.

For simplicity, we will provide the data for both **zone segmentation** and **defect segmentation**. Specifically: - **Zone mask:** Each pixel labeled as {Core, Cladding, Ferrule, Outside} – essentially an annotation of the fiber structure. This can be generated directly from our `mask_core`, `mask_cladding`, etc. (We might label adhesive zone same as ferrule or as separate if needed, but as per standards, we might ignore adhesive for modeling). - **Defect mask:** Each pixel labeled as {No Defect, Scratch, Contamination, OtherDefect}. This is derived from the defect regions we found. Pixels that are part of a defect region take that region's defect type label; all other pixels are no-defect. (If defects overlap, which is rare except maybe a scratch overlapping a contaminated area, we might prioritize one label or mark that as combined – but typically they won't perfectly overlap in physical reality).

We save these masks aligned with the input image. The output format could be: - **Image files:** e.g. under `output/pixel_level/images/` and `output/pixel_level/masks/`. We can keep the original image (or a preprocessed version) and a corresponding mask image (maybe stored as a PNG with indexed colors or as a NumPy `.npy` array). The filenames will match (e.g., `IMG_001.png` and `IMG_001_mask.png`). This is a common format for segmentation tasks and can be directly loaded by many frameworks. - **NumPy arrays:** Alternatively or additionally, we might save all images in a single `.npy` or HDF5 file for faster loading. For instance, a numpy array of shape (N, H, W, C) for images and (N, H, W) for masks, where N is number of images. This could be memory heavy if images are large, so often the on-disk image+mask pairs approach is used. We can document both options in the code, letting the user decide via config (e.g. `save_masks_as_images: True/False`).

The pixel-level dataset enables training a **segmentation CNN** to automatically identify fiber zones and defects pixel-by-pixel. Note that because our defect detection was unsupervised, these masks are **pseudo-labels** – they might not be perfect. However, they provide a starting point that can be refined later with active learning or manual corrections if needed.

Region-Level (Zone-Level) Classification Dataset

This dataset treats each **zone in each image as a data sample** described by features and labeled as "clean" or "defective". The idea is to train models (e.g. an ML classifier or a neural network on tabular data) to

predict if a given zone will pass or fail inspection based on its features, which could mimic industry standards or provide an automated QC tool.

We construct a table where each row corresponds to one zone of one image. Columns include: - **Image ID** (or file name) and **Zone ID** (e.g. "IMG_001, Zone A-Core"). These can serve as composite index or identification. - **Zone Type** (categorical feature: Core, Cladding, Ferrule). We may one-hot encode this if using as input to a model, or simply know that different zone types may have different threshold criteria.
- **Features:** All the computed zone-level features (as discussed earlier): - geometric (e.g. core_offset for core zone, fiber_diameter for cladding zone), - cleanliness (defect_count, defect_area_frac, etc. for that zone), - intensity/texture metrics (mean_intensity, intensity_std, avg_gradient, entropy_in_zone, etc.). - Possibly aggregated defect info: e.g. if defects exist, what is the worst defect type in that zone (e.g. "scratch" or "contamination"), and severity of that worst defect. - **Label/Target:** A binary or categorical label for the zone's condition. Since this is unsupervised, we derive the label from our detected defects: e.g., **Defective vs Clean**. We can mark *Defective* if any defect above a severity threshold is present in the zone. For example, if Zone A (Core) had **no defects**, label it as "Clean"; if it had even a small defect, label "Defective" (since core should be pristine [3]). For Zone D (Ferrule), maybe require a large defect >10 μm to label as "Defective" as per standards [4] . These criteria can be encoded from config or standards. We can also include a more fine-grained label indicating the type of defect causing failure (e.g. "Fail-scratch" vs "Fail-dirt"), or a severity category ("Clean", "Minor Defect", "Major Defect"). The simplest approach is binary classification (pass/fail per zone), but the dataset can support multi-class if needed.

This region-level dataset is saved as a **CSV or Parquet** file (`output/region_level/zone_defect_features.parquet` for example). Parquet is useful for large tables as it's compressed and efficient for columnar access. Each row has all numeric features and the label. If needed, we may also save a separate CSV mapping image and zone to the defect type(s) found for interpretability.

By training a model on this dataset, one could predict zone cleanliness from the features, essentially learning to mimic the unsupervised detection (or even improve on it if refined labels are provided later).

Image-Level Classification Dataset

At the highest level, we provide labels per entire image indicating its overall defect status or classification. This could be used to train a model that directly classifies a whole end-face image as "Good/Pass" or "Bad/Fail" or even into categories like "Scratch vs Contamination". There are a few ways to structure image-level labels: - **Binary pass/fail:** Label each image 0 = Pass (no significant defects) or 1 = Fail (has defects beyond allowed limits). This could be determined by looking at the zone labels – e.g. if any zone is "Fail", then the image fails. In our unsupervised context, if any core/cladding defect was found, likely it's a fail; small ferrule-only defects might still be a pass. We can implement logic per standards or a conservative approach (any defect -> fail, except tiny ones in ferrule). - **Multi-class by primary defect type:** For example, classify the image by what kind of defect it has (if any): "No defect", "Scratch", "Contamination", "Both" etc. If multiple types are present, we could choose the one deemed most severe or just label multi-label (see next). - **Multi-label:** Use separate boolean flags for each defect type presence. E.g., columns: *has_scratch*, *has_contamination*, *has_pit*. An image could have more than one true. This is useful if we want to train a model to detect multiple defect types simultaneously. Many modern classification networks can output multiple labels.

Our system can output a CSV that includes for each image: the image filename, perhaps the overall pass/fail, and the presence/absence of each defect category. We favor a **multi-label CSV**, because it's the most informative: one can always derive a pass/fail from it (just see if any defect column is true and if in a critical zone). For example:

Image	PassFail	Scratch	Contamination	Pit/Chip	Notes
IMG_001.png	Fail	True	False	False	Scratch in cladding
IMG_002.png	Fail	True	True	False	Scratch + dust
IMG_003.png	Pass	False	False	False	Clean end-face
…	…	…	…	…	…

This table is saved as `output/image_level/image_labels.csv`. **Parquet** could also be used if we have many images or want efficient slicing.

Additionally, to integrate with image classification training pipelines, some users like to have images sorted into folder by class. For binary classification, one could copy or symlink images into `output/image_level/Pass/` and `output/image_level/Fail/`. For multi-class single-label, folders by defect type could be made (though if multi-label, folder-per-class doesn't directly apply). We will primarily rely on the CSV labeling, but the code can optionally output a folder structure if configured (for example, a config flag `export_images_by_label: True` that, when True and if classes are mutually exclusive, creates such folders).

This image-level dataset allows training of a CNN to predict overall connector cleanliness, which could serve as a quick screening model. However, since defects can be small, image-level modeling is harder – that's why we have the detailed segmentation and region analysis as well.

Defect Library Construction

In addition to the structured datasets, the system builds a **Defect Library** that indexes each defect instance detected across the dataset. This library is essentially a collection of defect **samples** (like a mini-dataset of defects) which can be used for analysis, visualization, or even as training data for a focused defect classifier.

Each entry in the defect library includes: - **Defect ID:** a unique identifier for the defect (could be composite like ImageID_regionIndex or a global counter). - **Source Image:** reference to the original image file name (and potentially the zone). - **Location:** the position of the defect on the image, e.g. bounding box coordinates (x, y, width, height) and/or the polygon mask of the defect region. This helps in locating it back on the image. - **Defect Type Label:** the category assigned (from clustering, e.g. Scratch, Contamination, Pit, Other). - **Severity:** the severity score or level assigned. - **Zone:** which zone (core, cladding, ferrule) the defect is in. - **Feature Vector:** we can also store the features computed for that defect (this could be useful if someone wants to use this library to train a model purely on defect patches).

The library can be stored in two complementary ways: 1. **CSV/Parquet Index:** A file `defect_library_index.csv` with columns [DefectID, Image, Zone, Type, Severity, x, y, width, height, etc.]. This is a master table for reference. 2. **Image Patches:** Optionally, we save a cropped image of each

defect region (with some padding) in a directory structure. For example: - `defect_library/scratch/` folder contains images of all scratches, named by their ID or original image reference. - `defect_library/contamination/` contains all contamination patches, etc. Within each, we might further sub-divide by severity (e.g. subfolders `low/`, `high/`) or by zone if that's useful ("core_scratch/" vs "ferrule_scratch/"), depending on what is most insightful. The patch images are useful for quickly browsing defect examples or feeding into a dedicated model (like training a classifier on patch images to see if clustering was accurate).

The defect library serves as a **knowledge base of defects**. For example, an engineer can review all "Scratch" patches to verify the clustering indeed grouped correctly. If any are miscategorized, adjustments can be made (this might feed back into re-tuning clustering or manually relabeling some clusters).

It's important that the library is indexed by **location and source**, meaning we can trace a defect back to where it was on the connector. If later a model flags a defect, one can cross-reference it in this library for more context.

Building this library is integrated in the pipeline after clustering and labeling. The code will iterate over each detected defect, save the patch image (using the stored mask/bounding box on the original image), and append an entry to the index. This happens in the `dataset_builder.py` or a dedicated `defect_library.py` module.

Code Modules and Workflow

All the above functionality is divided into reusable modules. Below is the typical workflow executed in `main.py` (or an orchestrator function), showing how each module is used in sequence:

```
# Pseudo-code outline of main workflow
config = load_config("config.yaml")
initialize_output_dirs(config)

all_defect_features = []   # to collect for clustering
all_zone_records = []      # to collect for region-level dataset
all_image_labels = []      # to collect for image-level dataset
defect_id_counter = 0

for img_file in list_images(config.input_folder):
    image = load_image(img_file)
    # 1. Segment zones
    masks = segment_zones(image, config)  # returns core_mask, cladding_mask,
ferrule_mask
    zone_metrics = compute_zone_geometry(masks)  # radii, offsets, etc.
    # 2. Detect defects
    defect_mask, defect_regions = detect_defects(image, masks, config)
    # 3. Extract features
    for defect in defect_regions:
        features = extract_defect_features(defect, image, masks)
        features['zone'] = defect['zone']
```

```
        features['image'] = img_file.name
        features['id']   = defect_id_counter
        all_defect_features.append(features)
        defect_id_counter += 1
    zone_features = extract_zone_features(image, masks, defect_regions,
zone_metrics)
    # Determine zone labels (clean/defective) from defect_regions present:
    zone_records = assign_zone_labels(zone_features, defect_regions, config)
    all_zone_records.extend(zone_records)
    # Save pixel-level masks for this image
    save_mask_image(defect_mask, out_path=..., filename=img_file.name +
"_mask.png")
    save_mask_image(zone_mask_from(masks), out_path=..., filename=img_file.name
+ "_zones.png")
    # 4. Image-level label
    image_label = determine_image_label(zone_records)
    all_image_labels.append({ 'image': img_file.name, **image_label })
```

After processing all images:

```
# 5. Cluster defects and assign labels
clusters = cluster_defect_features(all_defect_features, config)
labeled_defects = assign_cluster_labels(all_defect_features, clusters)
# Update defect library and datasets with the labels:
for defect in labeled_defects:
    save_defect_patch(defect, image_cache[defect['image']], out_dir=...)
record_defect_library_index(labeled_defects, out_path=...)
# 6. Finalize datasets
save_region_level_table(all_zone_records, labeled_defects, out_path=...)
save_image_level_table(all_image_labels, out_path=...)
```

In the above pseudo-code: - We loop through each image, perform segmentation, defect detection, feature extraction, and *temporarily store* defect features and zone info. - Only after processing all images do we perform clustering on the collected defect features (`cluster_defect_features`), then tag each defect with its cluster label (`assign_cluster_labels`). - We then save the defect library and update zone records or image labels if the detailed defect type is needed in those (e.g. we might augment zone records with a “main_defect_type” field from the cluster labels). - We save all outputs in bulk at the end (though some intermediate saving, like pixel masks, was done per image to avoid holding too many large images in memory).

The configuration (`config`) is passed around as needed, containing parameters like cluster method and number of clusters, defect detection thresholds (e.g. for binary thresholding, minimum defect size to consider, etc.), morphological kernel sizes, etc., making the system adaptable to different image qualities or standards. For example, if dealing with multi-mode fibers with 50 µm core, the config can specify that core radius for segmentation.

Each function in modules has documentation explaining inputs/outputs and any assumptions. The code is written to be **reusable**; e.g., one could import `zone_segmentation.segment_zones` in a separate script to just get the fiber center from an image for some other purpose.

Documentation and Configurability

Throughout the codebase, we maintain clear documentation. Each module has a header explaining its purpose, and complex logic (like the clustering or the scratch detection) is explained in comments or docstrings with references to relevant research or standards if applicable. For instance, in `defect_detection.py`, the docstring can cite the idea from Mei *et al.* about using a min-max filter for contamination detection [12] and a linear detector for scratches [19], to justify our approach.

The `README.md` or a separate docs folder provides guidance on: - How to install required libraries (e.g. OpenCV, scikit-image, numpy, pandas, scikit-learn). - How to run the pipeline on a new dataset. - How to adjust config parameters (for example, increasing the sensitivity might involve lowering a threshold in `config.yaml` for defect detection). - The structure of output files and how to interpret them. For example, explaining that in `zone_defect_features.parquet`, each row is a zone with features and a `label=1` means zone fail.

Furthermore, we ensure the system is **scalable** (can handle dozens or hundreds of images). Time-consuming steps like clustering or feature extraction can be vectorized or parallelized if needed (the design could incorporate parallel processing for independent per-image computations, controlled by config to utilize multi-core processors).

Finally, by separating the concerns (segmentation, detection, feature engineering, etc.), the codebase allows improvements in one area without affecting others. For example, if a more advanced segmentation method (like a pre-trained U-Net for finding the fiber core) becomes available, one can replace the implementation in `zone_segmentation.py` and the rest of the pipeline remains unchanged, simply consuming the new masks. Similarly, one could experiment with different clustering algorithms in `clustering.py` without altering how features are computed.

In summary, this modular data preparation system will produce: - **Pixel-wise annotated images** ready to train segmentation models for both fiber zones and defects. - **Structured tables** of features and labels at zone and image level for training classifiers or doing statistical analysis. - **A defect example library** for visualization or further model development (like focused classification of defect patches).

By automating unsupervised defect labeling, this system jump-starts the model training process, providing rich annotated data without the costly step of manual labeling. It sets the stage for subsequent supervised learning where models can be trained on these annotations, and it can be iteratively refined as more data or feedback becomes available.

Sources: The design takes inspiration from industry standards for fiber inspection zones [3] [4] and known image processing techniques for defect detection [12] [19]. The defect definitions (scratches vs contamination) follow common descriptions in fiber optic maintenance guides [2]. This ensures the system's outputs are meaningful and aligned with real-world criteria for fiber end-face quality.

---

1 7 11 12 13 14 15 16 17 18 19 20 21 Automated Inspection of Defects in Optical Fiber Connector End Face Using Novel Morphology Approaches
https://www.mdpi.com/1424-8220/18/5/1408

2 8 9 Visual Scratch-Defect Fiber End Face Inspection System
https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=15398

3 4 5 6 Knowledge Article View - Knowledge Portal
https://knowledge.exfo.com/kb?id=kb_article_view&sysparm_article=KB0010238

10 [PDF] Active Core Alignment (ACA) process and evolution - diamond usa
https://www.diausa.com/fileadmin/user_upload/Content/Extranet/Documents_and_Downloads/White_Papers_Product_release_and_Newsletter/English/WP_Active_Core_Alignment.pdf

22 23 ML in Manufacturing: Detecting Defects with Unsupervised Learning and Image Feature Extraction – The Official Blog of BigML.com
https://blog.bigml.com/2021/12/03/ml-in-manufacturing-detecting-defects-with-unsupervised-learning-and-image-feature-extraction/