Report Kernel Rebooters: Vision Division

https://github.com/kekko7072/Final_Project_Kernel_Rebooters/tree/main

Marco Carraro
marco.carraro.23@studenti.unipd.it

Luca Pellegrini
luca.pellegrini.6@studenti.unipd.it

Francesco Vezzani
francesco.vezzani.1@studenti.unipd.it

February 21, 2026

1 Introduction

The overall goal of this project is to build a complete Computer Vision pipeline that can classify flower images - without the use of deep learning methods - and that can support performance evaluation across different feature-based approaches. In practical terms, the system is designed around a clear workflow: images are loaded from a structured dataset, visual descriptors are extracted, test images are compared against training references, and final predictions are produced.

The project includes multiple classification branches with a common evaluation pipeline: SIFT, SURF (optional at compile time), ORB, Template Matching, HOG, and BoW. While each branch uses a different representation and matching strategy, all of them are integrated in the same application flow and rely on the same metrics infrastructure for accuracy, confusion matrix, timing statistics, and recap export.

This report is therefore organized by module, first documenting architecture and implementation decisions for each algorithm, then summarizing the shared metrics system and experimental results.

2 Preprocessing

The preprocessing steps are implemented in the `main` function:

- Command line parsing: a `cv::CommandLineParser` is created. If a path to a valid dataset was not provided as a command line argument, the default relative path `../Final_project_proposal/` is used.
- Validation of the dataset path: the `main` function checks that the data path is valid, i.e. non-empty and containg the subdirectories "test_photos", "train_healthy_photos" and "train_diseased_photos".
- Loading test and train images: the `loadImages` function is responsible for loading the images from the dataset, and store them in custom `FlowerImageContainer` objects.
- Loading of template images: the `loadTemplates` function is responsible for loading the template images used by the Template Matching classifier.
- Results directory creation: the `main` function creates the output directory where the results of each classifier will be stored, if it does not exist yet.

2.1 Custom data types

Custom classes `FlowerImage`, `FlowerImageContainer`, and `FlowerTemplate` (derived from FlowerImage) are used to store the images and their associated metadata.

A FlowerImage object stores two `cv::Mat` objects, with both the BGR color version and the single-channel grayscale version of the image, together with some metadata extracted from the image filename and location within the dataset.

A FlowerImageContainer object stores a vector of FlowerImage objects, and keeps a record of which FlowerImage elements belong to any of the possible types of `FlowerType`, so that it can return a vector of only those elements of a certain flower class in the `getImagesByFlowerType` function (time complexity O(n), where n is the number of images returned).

A FlowerTemplate object is a specialization of the FlowerImage class, in that it stores a template (BGR color image) and its associated mask (single-channel grayscale image).

3 SIFT

3.1 Code Structure Decision

SIFT logic is organized into:

- `SIFTExtractor` (`include/sift.h`, `src/sift.cpp`) for low-level feature extraction, descriptor matching, and match filtering operations.
- `sift_processing` module (`include/sift_processing.h`, `src/sift_processing.cpp`) for dataset-level training and testing workflows.

This separation maintains architectural consistency with other modules: the extractor class handles algorithmic operations while the processing module orchestrates dataset traversal, descriptor aggregation, and prediction logic. As a result, `main.cpp` remains focused on high-level coordination, and the SIFT components can be independently tested, optimized, or replaced.

The `SIFTExtractor` interface exposes essential operations: `extract(...)`, `matchDescriptors(...)`, `filterMatches(...)`, and `matchAndFilter(...)`. Timing instrumentation is included to support performance analysis during development and evaluation phases.

3.2 Feature Extraction Choices

- Input images are in grayscale to focus on gradient information rather than color variation.
- SIFT parameters use OpenCV defaults with optional tuning: configurable number of features, octave layers, contrast threshold, edge threshold, and Gaussian sigma.
- The default configuration (`nfeatures=0`) extracts all detected keypoints, providing maximum detail for the training phase.

The decision to use grayscale processing aligns with SIFT's design philosophy: the algorithm was originally developed for intensity-based matching, and color information typically adds minimal discriminative value while increasing computational cost.

3.3 Training Pipeline

The training workflow follows a clear sequence:

1. Extract SIFT descriptors from all healthy and diseased (if `use_diseased` flag is set) training images.
2. Aggregate descriptors per class using vertical concatenation (`cv::vconcat`).

3. Store the combined descriptor matrices in a class-indexed map for efficient test-time retrieval.

This pipeline choice reflects a bag-of-features approach: all descriptors from a given class are pooled together without explicit spatial structure. The rationale is simplicity and robustness the classifier relies on the statistical distribution of local features rather than their precise geometric arrangement.

An important implementation detail is the use of `cv::vconcat` for descriptor combination, which is significantly faster than iterative concatenation and produces a single contiguous memory block per class.

3.4 Matching and Classification Strategy

For each test image:

1. Extract SIFT descriptors.
2. Match against each class's descriptor pool using FLANN-based matching.
3. Filter matches based on distance threshold: $d_{match} < threshold \times d_{min}$.
4. Classify as the class with the maximum number of good matches.

3.5 Matcher Configuration

Two matcher implementations were tested:

- `BFMatcher` with L2 norm: exhaustive search, guaranteed optimal matches, but slower for large descriptor sets (around 25 minutes for a classification).
- `FlannBasedMatcher`: approximate nearest neighbor search using KD-trees, significantly faster with same accuracy (around 5 minutes for a classification).

The FLANN matcher was selected as the default after benchmarking showed 5x speedup with negligible impact on classification accuracy for this dataset.

3.6 Robustness Choices

- Empty image check before extraction with early return and error message.
- Empty descriptor validation before matching operations.
- Graceful handling of images with zero detected keypoints (skipped in testing loop).
- Timing instrumentation for extraction and matching phases to identify performance bottlenecks.

3.7 Performance Considerations

The implementation includes several deliberate trade-offs:

- **Descriptor aggregation**: All training descriptors are stored in memory. For very large datasets (thousands of images per class), this could exceed available memory. A potential mitigation would be descriptor subsampling or clustering-based aggregation.
- **Exhaustive matching**: Each test descriptor is matched against all training descriptors for a given class. This scales as $O(N_{test} \times N_{train})$. FLANN provides partial relief.

These trade-offs were chosen to maintain code clarity and establish a functional baseline before introducing more complex optimization strategies.

3.8 Tested Configurations

Several parameters were evaluated during development:

- **Use of diseased descriptors**: Including diseased training samples improved classification accuracy.
- **Distance threshold**: Values between 1.5 and 2.5 were tested for the ratio test-inspired filtering, with 2.0 providing a good balance between precision and recall.
- **Matcher choice**: FLANN-based matching provided a significant speedup with no loss in accuracy compared to brute-force matching.
- **Nearest Neighbor Distance Ratio**: The original Lowe's ratio test was implemented but found to be too restrictive for this dataset, leading to very few matches. The distance-based thresholding approach provided a more flexible filtering mechanism that retained more valid matches while still reducing false positives.
- **Limits in train descriptor**: To mitigate the number of descriptors difference between classes, a maximum number of descriptors per class was tested (first 15000, then 30000). The sampling of those descriptors was done in multiple ways: random sampling, uniform sampling and k-means clustering. The best results were obtained with random sampling but this method was not consistent across runs. The other methods did not provide any improvement in accuracy and, expecially k-means was very time consuming. At the end, there is non limit on the number of train descriptors.

The final configuration selected for the evaluation phase was FLANN-based matching with a distance threshold of 1.7 (better in vlab simulation) and no limit on the number of training descriptors, as this provided the best overall accuracy while maintaining reasonable processing time.

In the table below (see Table 1), the results of the different tested configurations are reported, showing the impact of each parameter choice on both accuracy and processing time. Those tests has been performed on pc, not on vlab.

Matcher	Dataset	Thresh.	Train Descriptor	Time	Acc.	Daisy	Dand.	Rose	Sunf.	Tulip	NoFl.
BF	Full	2,5	NULL	26:15:996	20,31%	0,00%	41,67%	8,33%	16,67%	41,67%	0,00%
BF	Healthy	2,5	NULL	03:04:159	15,63%	8,33%	41,67%	0,00%	25,00%	8,33%	0,00%
FLANN	Full	2,5	NULL	06:54:950	20,31%	0,00%	41,67%	8,33%	16,67%	41,67%	0,00%
FLANN	Full	2,0	NULL	05:25:163	23,44%	8,33%	41,67%	16,67%	16,67%	41,67%	0,00%
FLANN	Full	2,5	15k (rnd sample)	04:17:885	20,31%	0,00%	33,33%	8,33%	25,00%	41,67%	0,00%
FLANN	Full	1,5	15k (rnd sample)	03:57:923	25,00%	16,67%	41,67%	8,33%	33,33%	33,33%	0,00%
FLANN	Full	1,5	NULL	05:29:062	18,75%	16,67%	33,33%	8,33%	25,00%	16,67%	0,00%
FLANN+NNDR	Full	0,8	NULL	05:33:180	17,19%	25,00%	16,67%	8,33%	16,67%	25,00%	0,00%
FLANN+NNDR	Full	0,7	NULL	05:23:491	23,44%	33,33%	33,33%	0,00%	16,67%	41,67%	0,00%
FLANN+NNDR	Full	0,7	15k (unif sample)	02:59:029	20,31%	41,67%	16,67%	8,33%	33,33%	8,33%	0,00%
FLANN+NNDR	Full	0,7	15k (k-means)	>30' train	18,75%	41,67%	25,00%	0,00%	25,00%	8,33%	0,00%
FLANN	Full	2,0	30k (unif sample)	04:02:434	20,31%	8,33%	41,67%	8,33%	25,00%	25,00%	0,00%

Table 1: SIFT performance evaluation

4 SURF

4.1 Code Structure Decision

SURF logic follows the same architectural pattern established for SIFT:

- `SURFExtractor` (`include/surf.h`, `src/surf.cpp`) for low-level feature extraction, descriptor matching, and match filtering operations.

- `surf_processing` module (`include/surf_processing.h`, `src/surf_processing.cpp`) for dataset-level training and testing workflows.

This parallel structure serves multiple purposes: it maintains consistency across the codebase, allows direct code comparison between SIFT and SURF implementations in `main.cpp`.

The `SURFExtractor` interface mirrors `SIFTExtractor`, exposing: `extract(...)`, `matchDescriptors(...)`, `filterMatches(...)`, and `matchAndFilter(...)`. This symmetry was a deliberate design decision to reduce cognitive load when working with multiple feature extractors and to facilitate comparative analysis.

4.2 Conditional Compilation

A critical implementation detail is that SURF support is optional and controlled through the `ENABLE_SURF` preprocessor flag. This is necessary because SURF is part of OpenCV's `xfeatures2d` module, which requires separate installation and is not included in OpenCV's default distribution due to patent restrictions.

The conditional compilation pattern is implemented using:

```
#ifdef ENABLE_SURF
// SURF implementation
#endif // ENABLE_SURF
```

This approach allows the project to compile and run without SURF when `xfeatures2d` is unavailable, while still supporting it when the module is properly installed. Users can enable SURF by configuring CMake with `-DCONFIG_ENABLE_SURF=ON`.

4.3 Feature Extraction Choices

- Input images are used in grayscale, consistent with SURF's original design for intensity-based features.
- SURF parameters use sensible defaults: Hessian threshold 400.0, 4 octaves, 3 octave layers.
- The `extended` flag (defaulting to `false`) controls descriptor dimensionality: 64 dimensions for standard SURF, 128 for extended SURF.
- The `upright` flag (defaulting to `false`) determines rotation invariance: when false, descriptors are rotation-invariant but computationally more expensive.

The Hessian threshold is the primary parameter controlling the number of detected keypoints. Higher values detect fewer but more distinctive keypoints, while lower values increase keypoint count at the cost of potentially including less stable features. The default is set to 100.0 but it has been increased to 400.0 to reduce the number of keypoints, speed up processing and increasing accuracy.

4.4 Training Pipeline

The training workflow is architecturally identical to SIFT:

1. Extract SURF descriptors from healthy training images.
2. Optionally extract descriptors from diseased training images.
3. Aggregate descriptors per class using `cv::vconcat`.
4. Store combined descriptor matrices indexed by flower class.

This structural equivalence was intentional, it ensures that performance differences between SIFT and SURF can be attributed solely to the feature extraction algorithm rather than implementation variations in the classification pipeline.

4.5 Matching Strategy

The matching and classification strategy is identical to SIFT:

1. Extract SURF descriptors from test image.
2. Match against each class's descriptor pool using FLANN-based matching.
3. Filter matches: $d_{match} < \tau \cdot d_{min}$.
4. Classify as the class with maximum good matches.

Using the same threshold parameter τ across SIFT and SURF allows for direct comparison, though optimal threshold values may differ between the two algorithms due to differences in descriptor distance distributions.

5 ORB

5.1 Code Structure Decision

ORB follows the established architectural pattern:

- `ORBExtractor` (`include/orb.h`, `src/orb.cpp`) for low-level feature extraction, descriptor matching, and match filtering operations.
- `orb_processing` module (`include/orb_processing.h`, `src/orb_processing.cpp`) for dataset-level training and testing workflows.

This structural consistency was intentional: by maintaining identical interfaces and workflows across SIFT, SURF, and ORB modules, the codebase remains maintainable and direct algorithm comparisons become straightforward. The only algorithmic differences are encapsulated within the respective extractor classes.

5.2 Feature Extraction Choices

- Input images are used in grayscale, consistent with ORB's design for binary pattern matching.
- ORB parameters are set to default values apart from `nfeatures`.
- `nfeatures` is set to 1500, with threshold values adjusted accordingly for optimal classification performance.

A critical discovery during implementation was that ORB's `nfeatures` parameter behaves fundamentally differently from SIFT's equivalent. Setting `nfeatures=0` results in no keypoint detection, whereas SIFT interprets this as "extract all detected keypoints." This behavior necessitated explicit parameter specification and careful tuning to achieve acceptable performance.

5.3 Training Pipeline

The training workflow mirrors SIFT and SURF:

1. Extract ORB descriptors from healthy training images.
2. Optionally extract descriptors from diseased training images.
3. Aggregate descriptors per class using vertical concatenation.
4. Store combined descriptor matrices indexed by flower class.

This structural equivalence ensures that performance differences can be attributed solely to the feature extraction algorithm rather than implementation variations.

5.4 Matching Strategy

ORB matching follows the same high-level strategy as SIFT/SURF but with a critical difference in the underlying matcher:

1. Extract ORB descriptors from test image.
2. Match against each class's descriptor pool using BFMatcher with Hamming distance. FLANN-based matching is possible using LSH (Locality Sensitive Hashing) but typically slower than BFMatcher for binary descriptors.
3. Filter matches: $d_{match} < \tau \cdot d_{min}$.
4. Classify as the class with maximum good matches.

The classification decision remains:

$$\hat{y}(x) = \arg\max_{c} |\text{good\_matches}(x, c)|$$

5.5 Parameter Tuning Process

Initial implementation with `nfeatures=0` resulted in zero keypoint detection across all images. This revealed a fundamental difference between ORB and SIFT parameter interpretation:

- SIFT: `nfeatures=0` $\rightarrow$ extract all detected keypoints
- ORB: `nfeatures=0` $\rightarrow$ extract exactly zero keypoints
- `nfeatures` ≥ 1000 to ensure sufficient descriptor density per image

6 Template Matching

The Template Matching method uses hand-picked template images (5 out of the 15 train healthy images available for every flower class) to classify the test images using the standard `cv::matchTemplate` function.

Template Matching relies of a simple idea: slide the template over one test image, and compute a similarity score for each relative position between the template and the image. Depending on the similarity score chosen (OpenCV provides several options), the minimum or maximun score achieved will correspond to the best match between template and image.

6.1 Code Structure Decision

Template Matching logic is split into:

- `processImage` function: compares one test images with one set of templates (of one of the classes)
- `template_match` function: complete wrapper of the Template Matching routine, iterates over all test images and compares similarity scores achieved with different classes to assign a final prediction

6.2 Feature Extraction Choices

Since the Template Matching method is heavily dependent on the size of both the template and the image been processed, and since templates images within each class have different sizes, all template images and associated masks are resized, with `cv::resize`, to a size of 400x300. This step is necessary to allow comparison of matching scores between templates, as the use of templates of different sizes would lead to matching scores being not normalized.

To improve classification, every test image is processed twice, in two different sizes (1200x900 and 800x600), and the best matching score for every class is kept.

6.3 Matching Choices

The `cv::matchTemplate` function allows a mask to be specified only if either TM_SQDIFF or TM_CCORR_NORMED are used as similarity scores.

Both similarity scores lead to similar classification results with our setup, so the TM_CCORR_NORMED (normalized cross-correlation) score was chosen.

6.4 Robustness Choices

- Empty input image check in `template_match(...)`.
- Empty input template or mask in `processImage(...)`

6.5 Trade-off

The implementation is intentionally simple and readable, with time complexity $O(N_{test} \cdot N_{template\_per\_class} \cdot N_{classes})$.

7 HOG

7.1 Objective

The HOG branch was introduced to add a global shape-and-gradient based descriptor that is easy to interpret and relatively stable in many real-world image conditions. The idea is simple: each test image is converted into one feature vector, then compared with feature vectors extracted from training images. The predicted class is taken from the nearest training sample.

From an engineering perspective, the objective was not to create a heavily optimized model, but to produce a transparent baseline that is easy to debug, easy to validate, and fully consistent with the project architecture.

7.2 Code Structure Decision

HOG logic is split into:

- `HOGExtractor` (`include/hog.h`, `src/hog.cpp`) for low-level feature extraction and distance computation.
- `hog(...)` wrapper (`src/matching.cpp`) for dataset-level loop and prediction print.

This separation is important because it keeps responsibilities clean: `HOGExtractor` handles only algorithmic operations, while `hog(...)` handles dataset traversal and prediction flow. As a result, `main.cpp` remains focused on orchestration, and the module can be reused or replaced with minimal impact on the rest of the system.

In the current version, `HOGExtractor` was simplified to a minimal API: `extract(...)` returns `bool` and `matchDescriptors(...)` returns the L2 distance. Timing fields and related getters were removed because they were not used by the application flow.

7.3 Feature Extraction Choices

- Input image is converted to grayscale if needed.
- The image is resized to a fixed window (`64x128`).
- HOG parameters are basic OpenCV defaults: block size `16x16`, block stride `8x8`, cell size `8x8`, 9 bins.

The fixed resize step is a key practical decision: HOG vectors can only be compared directly when they share the same dimensionality. Using a fixed window gives deterministic descriptor length and avoids shape-dependent edge cases in matching.

7.4 Matching Choices

- For each test descriptor, all train descriptors are scanned.
- Similarity score is Euclidean distance (L2).
- Predicted label is the train sample with minimum distance.

This results in a straightforward nearest-neighbor baseline:

$$\hat{y}(x) = \arg\min_i \|h(x) - h(x_i^{train})\|_2$$

where $h(\cdot)$ is the HOG descriptor.

The benefit of this choice is interpretability: every prediction can be traced back to one specific training sample and one explicit distance value, which is very useful during qualitative inspection.

7.5 Robustness Choices

- Empty input image check in `extract(...)`.
- Empty or incompatible descriptor check before matching.
- Sample is skipped by the wrapper when `extract(...)` returns `false`.

7.6 Trade-off

The implementation is intentionally simple and readable, but the exhaustive comparison step has cost $O(N_{test} \cdot N_{train})$. This is acceptable as a baseline and for medium dataset sizes, but it is a known scalability limitation for larger datasets.

8 BoW

8.1 Objective

While HOG focuses on global gradient structure, BoW was introduced to capture local visual patterns through keypoints. The goal is to convert a variable number of local ORB descriptors into a fixed-size global representation, so images can be compared in a compact and uniform way.

In other words, BoW bridges local detail and global matching: it keeps the discriminative power of local descriptors but produces one standardized vector per image.

8.2 Code Structure Decision

BoW logic is split into:

- `BoWExtractor` (`include/bow.h`, `src/bow.cpp`) for vocabulary building, histogram extraction, and histogram distance.
- `bow(...)` wrapper (`src/matching.cpp`) for train/test loop and prediction print.

This separation follows the same architectural rule used across the project: core algorithm in an extractor class, orchestration in a lightweight wrapper. This improves readability and allows future experiments (different vocabularies, different local features, different distance metrics) without touching application flow.

The current version also simplifies the BoW extractor interface to three public operations: `buildVocabulary(...)`, `extract(...)`, and `matchDescriptors(...)`. Unused timing/keypoint getter APIs were removed to keep the code shorter and easier to maintain.

8.3 Pipeline Choices

The implemented BoW pipeline is:

1. extract ORB descriptors from all train images,
2. convert descriptors to `CV_32F`,
3. run k-means to build a visual vocabulary (default size: 20 words),
4. assign each descriptor to the closest visual word,
5. build one histogram per image and normalize it,
6. compare test and train histograms with L2 distance.

This pipeline is intentionally classical and explicit. Every stage can be inspected independently (descriptors, vocabulary, histogram), making debugging and incremental improvement easier for the team.

8.4 Why ORB + K-means

- ORB is already present in the project and is computationally light.
- K-means gives a direct and standard way to form visual words.
- Fixed vocabulary keeps the implementation deterministic and easy to tune.

Choosing ORB also reduces integration friction, since ORB-based utilities were already available and familiar in the codebase. K-means, despite being simple, provides a clear semantic interpretation: each cluster center is treated as a visual word.

8.5 Histogram and Matching Formula

Each image is represented by a normalized histogram $b(x) \in R^K$, where K is vocabulary size. Prediction follows:

$$\hat{y}(x) = \arg\min_i \|b(x) - b(x_i^{train})\|_2$$

Normalization reduces sensitivity to raw keypoint count differences between images, so comparison focuses more on visual word distribution than on absolute descriptor count. This is especially useful when images produce very different numbers of detected keypoints.

8.6 Robustness Choices

- Vocabulary availability check before extraction.
- Empty descriptor/histogram checks.
- Skip logic in `bow(...)` when `extract(...)` returns `false`.

8.7 Trade-off

The current implementation favors clarity over optimization:

- vocabulary is rebuilt at each run,
- nearest-neighbor matching is exhaustive,
- no TF-IDF or advanced scoring yet.

These trade-offs are deliberate at this stage: the code remains concise and easy to reason about, and it provides a reliable baseline before introducing acceleration or more advanced weighting schemes.

9 Metrics implementation

9.1 Code Structure Decision

The metrics system is organized into three components:

- `metrics.h/cpp` (`include/metrics.h`, `src/metrics.cpp`) for core data structures and computation functions.
- `Metrics` struct as the central data container, passed by reference throughout the pipeline.
- `print_stats.h/cpp` (`include/print_stats.h`, `src/print_stats.cpp`) for formatted output and reporting.

This separation permits modular development: `metrics.cpp` handles numerical computations and data management, while `print_stats.cpp` focuses exclusively on presentation formatting. This design makes it easy to add new output formats without modifying the core metrics logic.

9.2 Core Data Structure

The `Metrics` struct encapsulates all evaluation data:

```
struct Metrics {
 int num_classes;
 int total_samples;
 int correct_predictions;
 std::vector<std::vector<int>> confusion_matrix;
 std::vector<double> processing_times;
};
```

Key design decisions:

- **Confusion matrix**: Stored as `vector<vector<int>>` rather than a flat array for intuitive indexing: `confusion_matrix[true_class][predicted_class]`.
- **Processing times**: Stored as individual measurements rather than pre-computed aggregates, allowing flexible statistical analysis (mean, median, percentiles) without data loss.
- **Counts vs rates**: Raw counts (`correct_predictions`, `total_samples`) are stored; accuracy is computed on-demand to avoid synchronization issues during incremental updates.

9.3 Classification Record System

To support detailed analysis of individual predictions, a parallel tracking system was implemented:

```
using ClassificationRecord = std::array<std::string, 3>;
using ClassificationRecap = std::vector<ClassificationRecord>;
```

Each `ClassificationRecord` contains:

1. **Filename**: Original test image filename for traceability
2. **True class**: Ground truth label from dataset
3. **Predicted class**: Algorithm's classification output

This structure enables post processing analysis of misclassified images, identification of systematic error patterns and comparison between algorithms behavior.

9.4 Accuracy Computation

Multiple accuracy metrics are supported:

Overall accuracy:

$$\text{accuracy} = \frac{\text{correct\_predictions}}{\text{total\_samples}}$$

Per-class accuracy:

$$\text{accuracy}_c = \frac{\text{confusion\_matrix}[c][c]}{\sum_j \text{confusion\_matrix}[c][j]}$$

Per-class accuracy is computed by dividing the diagonal element (correct predictions for class c) by the sum of the entire row (all predictions where the true class was c). This metric reveals which classes are well-distinguished and which are frequently confused.

The implementation includes proper handling of edge cases: zero-sample classes return 0.0 accuracy rather than causing division by zero.

9.5 Timing Statistics

Processing time analysis provides four key metrics:

- **Total time**: Sum of all per-image processing times.
- **Mean time**: Average processing time per image.
- **Min/Max time**: Identifies best-case and worst-case performance.

These metrics are computed as:

$$\text{mean} = \frac{1}{n}\sum_{i=1}^{n} t_i$$

$$\text{min} = \min_i t_i, \quad \text{max} = \max_i t_i$$

9.6 Reporting Functions

The `print_stats` module provides hierarchical reporting:
Component functions:

- `printConfusionMatrix(...)`: Formatted confusion matrix table.
- `printTimingStats(...)`: Processing time summary.
- `printPerClassAccuracy(...)`: Per-class accuracy breakdown.

Composite function:

- `printClassificationReport(...)`: Combines all three components into a comprehensive summary with optional algorithm name header.

This modular structure allows flexible reporting: debugging sessions might print only timing stats while final evaluation uses the full classification report.

9.7 File Export System

The metrics system includes comprehensive file export functionality for persistent storage and offline analysis. The `saveClassificationRecap` function generates a detailed text report for a single algorithm containing:

- **Per-image classification table**: Lists every test image with its true label, predicted label, and correctness indicator (V/X)
- **Summary statistics**: Total samples, correct predictions, overall accuracy

- **Timing analysis**: Mean, min, max, and total processing times
- **Per-class accuracy**: Accuracy percentage for each flower class
- **Confusion matrix**: Full confusion matrix for detailed error analysis

This comprehensive export ensures that all experimental results are preserved for later analysis and comparison between algorithms.

9.8 Design Patterns

Several design patterns enhance usability:
Const-correctness: Query functions take `const Metrics&` to communicate that they perform read-only operations:

```
double totalAccuracy(const Metrics& metrics);
```

Reference parameters: Update functions take `Metrics&` to enable efficient in-place modification:

```
void addPrediction(Metrics& metrics, int true_class, int predicted_class);
```

9.9 Extensibility

The metrics system is designed for easy extension:

- Adding new metrics: Extend the `Metrics` struct and add corresponding computation functions.
- New output formats: Add new functions to `print_stats.cpp` or create separate modules.
- Comparative analysis: Multiple algorithm results can be combined for side-by-side comparison and statistical significance testing.

9.10 Performance Considerations

The metrics system adds minimal overhead:

- Per-prediction update: $O(1)$ for counter increments and confusion matrix access.
- Time recording: $O(1)$ vector push_back operation.
- Accuracy computation: $O(c^2)$ where c is number of classes, negligible compared to feature extraction time.

9.11 Integration with Feature Extractors

All feature extraction modules (SIFT, SURF, ORB, TM, HOG, BoW) use this common metrics infrastructure, ensuring:

- Consistent evaluation methodology across algorithms.
- Comparable timing measurements (all include feature extraction through final prediction).
- Uniform reporting format for easy comparison.

This standardization is critical for fair performance comparisons and for making informed decisions about which algorithm is best suited for the flower classification task.

10 Flower Health Detection

10.1 Objective and Challenges

The initial project scope included a binary health classifier to distinguish between healthy and diseased flowers. However, implementation was not feasible due to two critical limitations:

- Absence of ground truth labels
- Training data quality issues

10.1.1 Absence of Ground Truth

The test dataset lacks health status labels entirely:

- Test image filenames do not encode health status
- No separate directories or metadata for healthy vs diseased test images

Without ground truth, it is impossible to compute accuracy, generate confusion matrices or validate classifier performance. This fundamentally prevents quantitative evaluation of any health classification system.

10.1.2 Training Data Quality Issues

Analysis of the training set revealed significant problems:

- **Visual similarity**: Many diseased images show only subtle differences from healthy ones (minor discoloration, early-stage infections)
- **Grayscale conversion loss**: Plant diseases manifest primarily through color changes (yellowing, browning, spots), but the current pipeline discards color information

10.2 System Architecture Support

The codebase was designed with health classification extension in mind:

- Training functions include `use_diseased` parameter for easy toggling
- Clear separation between `train_healthy` and `train_diseased` containers
- Modular design allows adding `trainHealthClassifier` and `testHealthClassifier` functions without architectural changes

Flower health detection was not implemented due to absent test labels and inadequate training data quality. The system architecture supports this extension once suitable data becomes available, but successful implementation will likely require color-preserving feature extraction or learning-based approaches designed specifically for disease symptom detection.

11 Performance Evaluation

Table 2 presents a comprehensive comparison of all implemented algorithms on the flower classification task. The evaluation metrics include total processing time, overall classification accuracy and per-class accuracy for each of the six flower categories. All experiments were conducted on identical hardware (vlab environment) using the same train/test split to ensure fair comparison. Obviusly the time result may vary on different runs.

Algorithm	Time (mm::ss::ms)	Acc.	Daisy	Dand.	Rose	Sunf.	Tulip	NoFl.
SIFT	02:36:875	21,88%	16,67%	41,67%	8,33%	25,00%	25,00%	0,00%
SURF	01:36:980	29,69%	33,33%	58,33%	25,00%	16,67%	25,00%	0,00%
ORB	01:04:009	21,88%	25,00%	16,67%	25,00%	25,00%	25,00%	0,00%
TM	08:04:068	20,31%	0,00%	41,67%	16,67%	0,00%	50,00%	0,00%
HOG	00:00:285	31,25%	16,67%	66,67%	8,33%	16,67%	58,33%	0,00%
BoW	00:01:875	34,38%	33,33%	41,67%	16,67%	50,00%	41,67%	0,00%

Table 2: Performance evaluation

12 Conclusion

12.0.1 Overall Performance Ranking

The algorithms can be ranked by overall accuracy as follows:

1. **BoW (34.38%)**: Best overall performer, demonstrating the effectiveness of vocabulary-based approaches for this task.
2. **HOG (31.25%)**: Second best, achieving competitive accuracy with dramatically lower computational cost.
3. **SURF (29.69%)**: Best among local feature descriptors, balancing speed and accuracy effectively.
4. **SIFT/ORB (21.88%)**: Tied performance but with significantly different computational profiles.
5. **TM (20.31%)**: Lowest accuracy, limited by its template-based approach.

12.0.2 Speed vs Accuracy Tradeoff

The results reveal distinct speed-accuracy tradeoffs:

HOG achieves the best computational efficiency, processing the entire pipeline in under 1 second while maintaining competitive accuracy (31.25%).

BoW provides the highest accuracy (34.38%) at a modest computational cost (1.875 seconds), representing an excellent balance for offline analysis.

SURF demonstrates the advantage of approximation strategies in local feature matching, achieving 8% higher accuracy than SIFT while requiring only 60% of the processing time. This validates the design choice of using Fast Hessian detection and reduced descriptor dimensionality.

ORB, despite being the fastest local feature method (64 seconds), does not achieve accuracy gains over SIFT. The binary descriptor's computational advantages are offset by reduced discriminative power for this particular task.

Template Matching exhibits the worst speed-accuracy tradeoff, requiring over 8 minutes while achieving only 20.31% accuracy. This poor performance is expected given the high variability in flower appearance, scale and viewpoint across images.

12.0.3 Per-Class Performance Patterns

Dandelion is the easiest class to classify, with most algorithms achieving 40-65% accuracy. HOG achieves the best performance (66.67%), likely due to dandelions' distinctive radial gradient structure that HOG captures effectively.

Rose is the most challenging class, with accuracies ranging from 8-25%. The low performance across all methods suggests high intra-class variability (different rose colors, bloom stages, viewing angles) that confuses gradient-based and local feature approaches alike.

Sunflower shows algorithm-dependent performance: BoW achieves 50% accuracy while SIFT/SURF/ORB remain at 16-25%. This suggests that vocabulary-based encoding better captures the distinctive but complex texture patterns of sunflower centers compared to direct descriptor matching.

Tulip performance varies widely (25-58%), with HOG achieving the best result. This variation suggests that shape-based features (captured by HOG) are more discriminative than texture-based features for tulips.

Daisy shows moderate and consistent performance (16-33%) across methods, suggesting it has neither distinctive shape nor texture features that any single approach can exploit effectively.

NoFlower achieves 0% accuracy across all algorithms, indicating systematic failure to recognize negative examples. This is a critical limitation due to the lack of sufficient NoFlower diversity in the training set and all algorithms are biased toward predicting one of the five flower classes thanks to the winner-takes-all classification strategy.

12.0.4 Critical Observations

NoFlower failure is the most significant finding: 0% accuracy across all six algorithms indicates a fundamental problem. Potential causes include:

- Insufficient negative examples in training set
- Class imbalance favoring flower classes
- Lack of explicit "reject" threshold in winner-takes-all classification
- Distribution mismatch between training and test NoFlower images

Rose classification difficulty (consistently lowest per-class accuracy) suggests that roses exhibit the highest visual variability among flower classes, making them difficult to model with any single feature representation.

BoW superiority indicates that mid-level feature representations (visual vocabularies) are better suited to this task than raw local descriptors or global gradient statistics, possibly due to better handling of intra-class variation.

12.1 Limitations and Conclusions

The relatively low absolute accuracies (20-34%) indicate that traditional feature-based methods struggle with the inherent complexity of flower classification.

Despite these limitations, the implemented system provides a solid baseline, comprehensive evaluation framework and modular architecture that facilitates future improvements and extensions.

13 Work Distribution

The project workload was distributed among team members in the following way:

- Marco Carraro: SIFT, SURF, ORB and metrics system.
- Luca Pellegrini: preprocessing pipeline, image container infrastructure, template matching.
- Francesco Vezzani: HOG and BoW implementation, matching wrappers,

All team members participated in code reviews, integration testing, and collaborative debugging sessions to ensure consistency across modules and resolve interface compatibility issues.

Hours worked	Marco Carraro	Luca Pellegrini	Francesco Vezzani
Total	**∼40**	**∼40**	**∼20**

Table 3: Approximate hours worked per team member

References / Links

- OpenCV SIFT documentation: `https://docs.opencv.org/4.5.4/d7/d60/classcv_1_1SIFT.html#a94ee0141f77675822e574bbd2c079811`
- OpenCV SURF documentation: `https://docs.opencv.org/4.6.0/d5/df7/classcv_1_1xfeatures2d_1_1SURF.html`
- OpenCV FlannBasedMatcher documentation: `https://docs.opencv.org/4.5.4/dc/de2/classcv_1_1FlannBasedMatcher.html`
- OpenCV HOGDescriptor documentation: `https://docs.opencv.org/4.x/d5/d33/structcv_1_1HOGDescriptor.html`
- OpenCV ORB documentation: `https://docs.opencv.org/4.x/db/d95/classcv_1_1ORB.html`
- OpenCV kmeans documentation: `https://docs.opencv.org/4.x/d1/d5c/tutorial_py_kmeans_opencv.html`