PyUCC User Manual

Table of Contents

Document Version: 1.0
Application: PyUCC (Python Unified Code Counter)

---

1. Introduction

PyUCC is an advanced static code analysis tool. Its primary objective is to provide quantitative metrics on software development and, crucially, to track code evolution over time through a powerful **Differing** system.

What is it for?

1. **Counting:** Knowing exactly how many lines of code, comments, and blank lines make up your project.
2. **Measuring:** Calculating software complexity and maintainability.
3. **Comparing:** Understanding exactly what changed between two versions (added/removed/modified files and how complexity has shifted).

---

2. Core Concepts

Before starting, it is useful to understand the key terms used in the application.

2.1 Baseline

A **Baseline** is an instant "snapshot" of your project at a specific moment in time.
* When you create a baseline, PyUCC saves a copy of the files and calculates all metrics.
* Baselines serve as reference points (benchmarks) for future comparisons.

2.2 Supported Metrics

- **SLOC (Source Lines of Code):**
  - *Physical Lines:* Total lines in the file.
  - *Code Lines:* Lines containing executable code.
  - *Comment Lines:* Documentation lines.
  - *Blank Lines:* Empty lines (often used for formatting).
- **Cyclomatic Complexity (CC):** Measures the complexity of the control flow (how many `if`, `for`, `while` statements, etc.). **Lower is better.**
- **Maintainability Index (MI):** An index from 0 to 100 estimating how easy the code is to maintain. **Higher is better** (above 85 is excellent, below 65 is problematic).

2.3 Profile

A **Profile** is a saved configuration that tells PyUCC:
* Which folders to analyze.
* Which languages to include (e.g., Python and C++ only).
* What to ignore (e.g., `venv`, `build` folders, temporary files).

---

3. User Interface (GUI)

The interface is divided into functional zones to keep the workflow organized.

1. **Top Bar:**
   - **Profile** selection.
   - Access to **Settings** and Profile Manager (**Manage**).
2. **Actions Bar:** The main buttons to start operations (`Scan`, `Countings`, `Metrics`, `Differing`).
3. **Progress Area:** Progress bar and file counter.
4. **Results Table:** The large central table where data appears.
5. **Log & Status:** At the bottom, a log panel to see what is happening and a status bar monitoring system resources (CPU/RAM).

---

4. Configuration and Settings

4.1 Configuration Files Location

PyUCC stores all configuration in the application directory (where the executable or source code is located):

- **`profiles.json`** - Stores all your analysis profiles (project paths, filters, ignore patterns)

- **`settings.json`** - Stores application settings (baseline directory, retention policy, duplicates parameters)

Location:
- If running from source: in the repository root folder (e.g., `C:\src\PyUcc\`)
- If running from compiled exe: in the same folder as the executable

Advantages:
- Fully portable application - copy the entire folder to move your setup
- Easy to backup - just backup the application folder
- No hidden files in user's home directory

4.2 Application Settings (Settings.json)

You can configure PyUCC's behavior through the □□ **Settings** menu in the top bar.

Available Settings:

1. **Baseline Directory**
2. **What it is:** Where PyUCC stores baseline snapshots.
3. **Default:** `baseline/` subfolder in the application directory.
4. **Recommendation:** Use the default, or set a custom path if you want to store baselines on a network drive or separate disk.
5. **Example:** `D:\ProjectBaselines\` or `\\server\share\baselines\`
6. **Max Baselines to Keep**
7. **What it is:** Maximum number of baseline snapshots to retain per profile.
8. **Default:** 5
9. **Behavior:** When exceeded, PyUCC automatically deletes the oldest baselines.
10. **Recommendation:**
    - 3-5 for small projects
    - 10+ for critical projects requiring long history
    - 20+ if disk space is not a concern
11. **Zip Baselines**
12. **What it is:** Whether to compress baseline snapshots as `.zip` files.
13. **Default:** `false` (disabled)
14. **Advantages when enabled:**
    - Saves disk space (50-80% reduction for source code)
    - Faster to transfer/backup
15. **Disadvantages:**
    - Slightly slower to create/compare (compression overhead)
    - Cannot browse snapshot files directly
16. **Recommendation:** Enable for large projects (>10,000 files) or when disk space is limited.
17. **Duplicates Settings** (stored automatically)
18. Threshold, k-gram size, winnowing window
19. These are saved when you use the Duplicates feature
20. See Section 9 for detailed explanation

How to Configure:

1. Click □□ **Settings** in the top bar.
2. Set your preferred baseline directory (or leave default).
3. Set max baselines to keep.
4. Check "Zip baselines" if desired.
5. Click **Save**.

First-Time Setup Recommendation:

At first run, PyUCC will use sensible defaults:
- Baselines stored in `baseline/` subfolder
- Keep last 5 baselines
- No compression

You should configure Settings if:
- You want baselines on a different drive (e.g., network storage, external disk)
- You need to keep more baseline history
- You're running out of disk space and want compression

4.3 First Run & Profile Configuration

The first thing to do upon opening PyUCC is to define *what* to analyze.

1. Click on □□ **Manage...** in the top bar.
2. Click on □ **New** to clear the fields.
3. Enter a **Name** for the profile (e.g., "My Backend Project").
4. In the **Paths** section, use **Add Folder** to select your code's root directory.
5. In the **Filter Extensions** section, select the languages you are interested in (e.g., Python, Java).
6. In the **Ignore patterns** box, you can keep the defaults (which already exclude `.git`, `__pycache__`, etc.).
7. Click □ **Save**.

4.4 Simple Analysis (Scan, Countings, Metrics)

If you only want to analyze the current state without comparisons:

- □ **Scan:** Simply verifies which files are found based on the profile filters. Useful to check if you are including the right files.
- □ **Countings:** Analyzes every file and reports how many code, comment, and blank lines exist.
- □ **Metrics:** Calculates Cyclomatic Complexity and Maintainability Index for each file.

> **Tip:** You can double-click on a file in the results table to open it in the built-in **File Viewer**, which provides syntax highlighting and a colored minimap (blue=code, green=comments).

4.5 The "Differing" Workflow (Comparison)

This is PyUCC's most powerful feature.

Step A: Create the First Baseline
1. Select your profile.
2. Click on □ **Differing**.
3. If this is the first time you analyze this project, PyUCC will notify you: *"No baseline found"*.
4. Confirm creation. PyUCC will take a "snapshot" of the project (Baseline).

Step B: Work on the Code
Now you can close PyUCC and work on your code (modify files, add new ones, delete old ones).

Step C: Compare
1. Reopen PyUCC and select the same profile.
2. Click on □ **Differing**.
3. This time, PyUCC detects an existing previous Baseline and asks which one to compare against (if you have multiple).
4. The result will be a table with specific color coding:
* **Green:** Added files or improved metrics.
* **Red:** Deleted files or worsened metrics (e.g., increased complexity).
* **Yellow/Orange:** Modified files.
* **Δ (Delta) Columns:** Show numerical differences (e.g., `+50` code lines, `-2` complexity).

> **Diff Viewer:** If you double-click a row in the Differing results, a window will open showing the two files side-by-side, highlighting exactly which lines changed.

---

5. Exemplary Use Cases

Case 1: Refactoring

- **Goal:** You want to clean up code and ensure you haven't increased complexity.
- **Action:** Create a Baseline before starting. Perform refactoring. Run *Differing*.
- **Verification:** Check the **Δ avg_cc** column. If it is negative (e.g., `-0.5`), great! You reduced complexity. If **Δ comment_lines** is positive, you improved documentation.

Case 2: Code Review

- **Goal:** A colleague added a new feature. What changed?
- **Action:** Run *Differing* against the previous master/main version.
- **Verification:** Sort by "Status". Immediately see **Added** (A) and **Modified** (M) files. Open the Diff Viewer on modified files to inspect specific lines.

---

6. Development Philosophy (For Developers)

PyUCC was built following rigorous software engineering principles, reflected in its stability and usage.

6.1 Clean Code & PEP8 Standards

The code adheres to the Python **PEP8** standard. This ensures that if you ever want to extend the tool or write automation scripts using the `core` modules, you will find readable, standardized, and predictable code.

6.2 Separation of Concerns (SoC)

The application is strictly divided into two parts:
1. **Core (`pyucc.core`):** Contains pure logic (scanning, metric calculation, diff algorithms). It knows nothing about the GUI.
2. **GUI (`pyucc.gui`):** Handles only visualization and user interaction.
Philosophy: This allows changing the interface without breaking the logic, or using the logic via command line without launching the GUI.

6.3 Non-Blocking UI (Worker Manager)

You may notice the interface never freezes, even when analyzing thousands of files.
This is thanks to the **WorkerManager**. All heavy operations are executed in separate background threads. The GUI receives updates via a thread-safe `queue`.
* **User Benefit:** You can always press "Cancel" if an operation takes too long.

6.4 Intelligent Matching Algorithm (Gale-Shapley)

In *Differing*, PyUCC doesn't just check if filenames are identical. It uses an algorithm inspired by the "Stable Marriage Problem" (Gale-Shapley) combined with Levenshtein distance on paths.
* **Philosophy:** If you move a file from one folder to another, the system attempts to recognize it as the *same* file moved, rather than marking one as "Deleted" and one as "Added".

6.5 Determinism

The system uses content hashing (SHA1/MD5) to optimize calculations (caching) and to determine if a file has *truly* changed, ignoring the filesystem modification timestamp if the content remains identical.

---

7. Troubleshooting Common Issues

- **Program finds no files:** Check the Profile Manager to see if the file extension is selected in the language list or if the folder is covered by "Ignore patterns".
- **Extreme slowness:** If you included folders with thousands of small non-code files (e.g., `node_modules` or image assets), add them to "Ignore patterns".
- **Empty Diff Viewer:** Ensure the source files still exist on disk. If you deleted the project folder after creating the baseline, the viewer cannot display the current file.

---

8. New Features (Since v1.0)

This release adds several capabilities that improve code-quality analysis, reproducibility of baselines, and duplicate detection across a codebase. Below is a concise description of what changed and how to use the new features.

8.1 Duplicate Detection (GUI + CLI)

- **What it does:** Finds exact and fuzzy duplicates across the project. Exact duplicates are detected by content hashing (SHA1). Fuzzy duplicates use k-gram fingerprinting with a winnowing step to create fingerprints, and a Jaccard similarity score to rank likely duplicates.
- **Parameters:** `k` (k-gram size), `window` (winnowing window), and `threshold` (percent similarity). Defaults are chosen for balanced precision/recall but can be adjusted.
- **How to run (GUI):** Use the new **Duplicates** button in the Actions bar (it appears before the Differ button). A dialog lets you choose extensions, the similarity threshold, and fingerprinting parameters. Settings persist between runs.
- **How to run (CLI):** `python -m pyucc duplicates <path> --threshold 5.0 --ext .py .c` prints a JSON structure with duplicates found.
- **Exports:** Results can be exported to CSV and to a UCC-style textual report placed inside baseline folders (when run during baseline creation).

8.2 UCC-style Duplicate and Differ Reports

- **Compact UCC-style table:** Differ now produces a compact table compatible with UCC-like output, including additional Δ (delta) columns: `ΔCode`, `ΔComm`, `ΔBlank`, `ΔFunc`, `ΔAvgCC`, `ΔMI`. This helps quickly see numeric changes in code, comments, blank lines, number of functions, average cyclomatic complexity and maintainability.
- **Duplicates report:** A textual `duplicates_report.txt` is generated (when requested) that lists duplicate groups with pairwise percent similarity and the parameters used to generate them. Baselines store the parameters so results are reproducible.

Example (compact UCC-style snippet):

```
File                                  Code   Comm  Blank  Func  AvgCC   MI   ΔCode  ΔComm  ΔBlank  ΔFunc  ΔAvgCC  ΔMI
-------------------------------------------------------------------------------------------------------------------
src/module/a.py                        120     10      8      5     2.3    78   +10     -1      0       +0     -0.1    +2
src/module/b_copy.py                   118      8     10      5     2.4    76   -2      -2      +2      0      +0.1    -2
```

8.3 Scanner & Baseline Improvements

- **Centralized scanning:** The `scanner` is the canonical provider of the file list. Heavy modules (Differ, Duplicates finder) accept a `file_list` produced by the scanner to avoid rescanning and to ensure consistent filtering.
- **Ignore pattern normalization:** Ignore entries like `.bak` are normalized to `*.bak` and matching is case-insensitive by default; this prevents accidental inclusion of backup files in baselines.
- **Baseline reproducibility:** Baselines now store the duplicates parameters and the file list snapshot. When a baseline is re-created or analyzed later, PyUCC attempts to re-run per-file function analysis (if `lizard` is available) so that function-level metrics in older baselines remain useful.

8.4 Notes on Dependencies

- Function-level metrics (number of functions, per-function CC) rely on `lizard`. If `lizard` is not installed, PyUCC will still produce SLOC and coarse metrics but function details may be missing. Baseline creation records this state and will re-run function analysis if `lizard` becomes available later.

---

9. Duplicate Detection: Algorithms and Technical Details

This section provides a deeper understanding of how PyUCC identifies duplicate code, what the algorithms do, and how to interpret the results.

9.1 Exact Duplicate Detection

How it works:
- PyUCC normalizes each file (strips leading/trailing whitespace from each line, converts to lowercase optionally).
- Computes a SHA1 hash of the normalized content.
- Files with identical hashes are considered exact duplicates.

Use case: Finding files that were copy-pasted with no or minimal changes (e.g., `utils.py` and `utils_backup.py`).

What you'll see:
- In the GUI table: pairs of files marked as "exact" duplicates with 100% similarity.
- In the report: listed under "Exact duplicates" section.

9.2 Fuzzy Duplicate Detection (Advanced)

Fuzzy detection identifies files that are *similar* but not identical. This is useful for finding:
- Code that was copy-pasted and then slightly modified.
- Refactored modules that share large blocks of logic.
- Experimental branches or "almost-duplicates" that should be merged.

Algorithm Overview:

1. **K-gram Hashing (Rolling Hash with Rabin-Karp):**
2. Each file is divided into overlapping sequences of `k` consecutive lines (k-grams).
3. A rolling hash (Rabin-Karp polynomial hash) is computed for each k-gram.
4. This produces a large set of hash values representing all k-grams in the file.
5. **Winnowing (Fingerprint Selection):**
6. To reduce the number of hashes (and improve performance), PyUCC applies a "winnowing" technique.
7. A sliding window of size `w` moves over the hash sequence.
8. In each window, the minimum hash value is selected as a fingerprint.
9. This creates a compact set of representative fingerprints for the file.
10. **Key property:** If two files share a substring of at least `k + w - 1` lines, they will share at least one fingerprint.
11. **Inverted Index:**
12. All fingerprints from all files are stored in an inverted index: `{fingerprint -> [list of files containing it]}`.
13. This allows fast lookup of which files share fingerprints.
14. **Jaccard Similarity:**

15. For each pair of files that share at least one fingerprint, PyUCC computes the Jaccard similarity:
    `Jaccard(A, B) = |A ∩ B| / |A ∪ B|`
16. Where A and B are the sets of fingerprints for the two files.
17. If the Jaccard score is above the threshold (default: 0.85, meaning 85% similarity), the pair is flagged as a fuzzy duplicate.
18. **Percent Change Calculation:**
19. PyUCC also estimates the percentage of lines that differ between the two files.
20. If `pct_change <= threshold` (e.g., ≤5%), the files are considered duplicates.

Parameters you can adjust:

- **`k` (k-gram size):** Number of consecutive lines in each k-gram. Default: 25.
- Larger `k` → fewer false positives, but may miss small duplicates.
- Smaller `k` → more sensitive, but may produce false positives.
- **`window` (winnowing window size):** Size of the window for selecting fingerprints. Default: 4.
- Larger window → fewer fingerprints, faster processing, but may miss some matches.
- Smaller window → more fingerprints, slower, but more thorough.
- **`threshold` (percent change threshold):** Maximum allowed difference (in %) to still consider two files duplicates. Default: 5.0%.
- Lower threshold → stricter matching (only very similar files).
- Higher threshold → more lenient (catches files with more differences).

Recommended settings:

Use Case	k	window	threshold
Strict duplicate finding (only near-identical files)	30	5	3.0%
Balanced (default)	25	4	5.0%
Loose matching (catch refactored code)	20	3	10.0%
Very aggressive (experimental)	15	2	15.0%

9.3 Understanding Duplicate Reports

GUI Table Columns:

- **File A / File B:** The two files being compared.
- **Match Type:** "exact" or "fuzzy".
- **Similarity (%):** For fuzzy matches, the Jaccard similarity score (0-100%).
- **Pct Change (%):** Estimated percentage of lines that differ.

Textual Report (duplicates_report.txt):

The report is divided into two sections:

1. **Exact Duplicates:**
   ```
   Exact duplicates: 3

src/utils.py <=> src/backup/utils_old.py
src/module/helper.py <=> src/module/helper - Copy.py
```

1. **Fuzzy Duplicates:**
   ```
   Fuzzy duplicates (threshold): 5

src/processor.py <=> src/processor_v2.py
Similarity: 92.5% | Pct Change: 3.2%

src/core/engine.py <=> src/experimental/engine_new.py
Similarity: 88.0% | Pct Change: 4.8%
```

Interpretation:

- **High similarity (>95%):** Strong candidates for deduplication. Consider keeping only one version or merging.
- **Medium similarity (85-95%):** Review manually. May indicate refactored code or intentional variations.
- **Threshold violations:** Files that exceed the `pct_change` threshold won't appear in the report, even if they share some fingerprints.

10. Reading and Interpreting Differ Reports

The Differ functionality produces several types of output. Understanding each helps you track code evolution accurately.

10.1 Compact UCC-Style Table

When you run *Differing*, PyUCC generates a compact summary table similar to the original UCC tool:

Example:

```
File                                  Code   Comm  Blank  Func  AvgCC   MI   ΔCode  ΔComm  ΔBlank  ΔFunc  ΔAvgCC  ΔMI
-------------------------------------------------------------------------------------------------------------------
src/module/a.py                        120     10      8      5     2.3    78   +10     -1      0       +0     -0.1    +2
src/module/b.py                        118      8     10      5     2.4    76   -2      -2      +2      0      +0.1   -2
src/new_feature.py                      45      5      3      2     1.8    82   +45     +5      +3      +2     +1.8   +82
src/old_code.py                         --     --     --     --     --    --   -30     -5      -2      -1     -2.1   -75
```

Column Meanings:

Column	Meaning
File	Relative path to the file
Code	Current number of code lines
Comm	Current number of comment lines
Blank	Current number of blank lines
Func	Number of functions detected (requires `lizard`)
AvgCC	Average cyclomatic complexity per function
MI	Maintainability Index (0-100, higher is better)
ΔCode	Change in code lines (current - baseline)
ΔComm	Change in comment lines
ΔBlank	Change in blank lines
ΔFunc	Change in function count
ΔAvgCC	Change in average cyclomatic complexity
ΔMI	Change in maintainability index

Color Coding (GUI):

- **Green rows:** New files (Added) or improved metrics (e.g., ΔAvgCC < 0, ΔMI > 0).
- **Red rows:** Deleted files or worsened metrics (e.g., ΔAvgCC > 0, ΔMI < 0).
- **Yellow/Orange rows:** Modified files with mixed changes.
- **Gray rows:** Unmodified files (identical to baseline).

What to look for:

- **ΔCode >> 0:** Significant code expansion. Is it justified by new features?
- **ΔComm < 0:** Documentation decreased. Consider adding more comments.
- **ΔAvgCC > 0:** Complexity increased. May indicate need for refactoring.
- **ΔMI < 0:** Maintainability worsened. Review the changes.
- **New files with high AvgCC:** New code is already complex. Flag for review.

10.2 Detailed Diff Report (diff_report.txt)

A textual report is saved in the baseline folder:

Structure:

```
PyUCC Baseline Comparison Report
================================
Baseline ID: MyProject__20251205T143022_local
Snapshot timestamp: 2025-12-05 14:30:22

Summary:
  New files: 3
  Deleted files: 1
  Modified files: 12
  Unchanged files: 45

Metric Changes:
  Total Code Lines: +150
  Total Comments: -5
  Average CC: +0.2 (slight increase in complexity)
```

```
  Average MI: -1.5 (slight decrease in maintainability)

[Compact UCC-style table here]

Legend:
  A = Added file
  D = Deleted file
  M = Modified file
  U = Unchanged file
  ...
```

10.3 CSV Exports

You can export any result table to CSV for further analysis in Excel, pandas, or BI tools.

Columns include:
- File path
- All SLOC metrics (code, comment, blank lines)
- Complexity metrics (CC, MI, function count)
- Deltas (if from a Differ operation)
- Status flags (A/D/M/U)

Use cases:
- Trend analysis over multiple baselines.
- Generating charts (e.g., complexity over time).
- Feeding into CI/CD quality gates.

---

11. Practical Use Cases and Workflows

Use Case 1: Detecting Copy-Paste Code Before Code Review

Scenario: Your team is developing a new module. You suspect some developers copy-pasted existing code instead of refactoring.

Workflow:
1. Create a profile for your project.
2. Click **Duplicates** button.
3. Set threshold to 5% (strict).
4. Review the results table.
5. For each fuzzy duplicate pair:
- Double-click to open both files in the diff viewer (if implemented).
- Assess whether the duplication is intentional or should be refactored into a shared utility.
6. Export to CSV and share with the team for discussion.

Expected outcome: You identify 3-5 near-duplicate files and create tickets to consolidate them.

Use Case 2: Tracking Complexity During a Refactoring Sprint

Scenario: Your team plans a 2-week refactoring sprint to reduce technical debt.

Workflow:
1. **Before the sprint:** Create a baseline ("Pre-Refactor").
- Click **Differing** → Create baseline.
- Name it "PreRefactor_Sprint5".
2. **During the sprint:** Developers refactor code, extract functions, add comments.
3. **After the sprint:** Run **Differing** against the baseline.
4. Review the compact table:
- Check ΔAvgCC: Should be negative (complexity reduced).
- Check ΔMI: Should be positive (maintainability improved).
- Check ΔComm: Should be positive (more documentation).
5. Generate a diff report and attach to sprint retrospective.

Expected outcome: Quantitative proof that refactoring worked: "We reduced average CC by 15% and increased MI by 8 points."

Use Case 3: Ensuring New Features Don't Degrade Quality

Scenario: You're adding a new feature to a mature codebase. You want to ensure the new code doesn't introduce excessive complexity.

Workflow:
1. Create a baseline before starting feature development.
2. Develop the feature in a branch.

3. Before merging to main:
- Run **Differing** to compare current state vs. baseline.
- Filter for new files (status = "A").
- Check AvgCC and MI of new files.
- If AvgCC > 5 or MI < 70, flag for refactoring before merge.
4. Use **Duplicates** to ensure new code doesn't duplicate existing utilities.

Expected outcome: New feature code passes quality gates before merge.

Use Case 4: Generating Compliance Reports for Audits

Scenario: Your organization requires periodic code quality audits.

Workflow:
1. Create baselines monthly (e.g., "Audit_2025_01", "Audit_2025_02", ...).
2. Each baseline automatically generates:
- `countings_report.txt`
- `metrics_report.txt`
- `duplicates_report.txt`
3. Archive these reports in a compliance folder.
4. For the audit, provide:
- Trend of total SLOC over time.
- Trend of average CC and MI.
- Number of duplicates detected and resolved each month.

Expected outcome: Auditors see measurable improvement in code quality metrics over time.

Use Case 5: Onboarding New Developers with Code Metrics

Scenario: A new developer joins the team and needs to understand the codebase.

Workflow:
1. Run **Metrics** on the entire codebase.
2. Export to CSV.
3. Sort by AvgCC (descending) to identify the most complex modules.
4. Share the list with the new developer:
- "These 5 files have the highest complexity. Be extra careful when modifying them."
- "These modules have low MI. They're candidates for refactoring—good learning exercises."
5. Use **Duplicates** to show which parts of the code have redundancy (explain why).

Expected outcome: New developer understands code hotspots and quality issues faster.

---

12. Tips for Effective Use

12.1 Profile Management

- **Create separate profiles** for different subprojects or components.
- Use **ignore patterns** aggressively to exclude:
- `node_modules`, `venv`, `.venv`
- Build outputs (`build/`, `dist/`, `bin/`)
- Generated code
- Test fixtures or mock data

12.2 Baseline Strategy

- **Naming convention:** Use descriptive names with dates or version tags:
- `Release_v1.2.0_20251201`
- `PreRefactor_Sprint10`
- `BeforeMerge_FeatureX`
- **Frequency:** Create baselines at key milestones:
- End of each sprint
- Before/after major refactorings
- Before releases
- **Retention:** Keep at least 3-5 recent baselines. Archive older ones.

12.3 Interpreting Metrics

Cyclomatic Complexity (CC):

- **1-5:** Simple, low risk.
- **6-10:** Moderate complexity, acceptable.
- **11-20:** High complexity, review recommended.
- **21+:** Very high complexity, refactoring strongly recommended.

Maintainability Index (MI):
- **85-100:** Highly maintainable (green zone).
- **70-84:** Moderately maintainable (yellow zone).
- **Below 70:** Low maintainability (red zone), needs attention.

12.4 Duplicate Detection Best Practices

- Start with **default parameters** (k=25, window=4, threshold=5%).
- If you get too many false positives, **increase k** or **decrease threshold**.
- If you suspect duplicates are being missed, **decrease k** or **increase threshold**.
- Always **review fuzzy duplicates manually**—not all similarities are bad (e.g., interface implementations).

---

13. Troubleshooting and FAQs

Q: Duplicates detection is slow on large codebases.

A:
- Use profile filters to limit the file types analyzed.
- Increase `k` and `window` to reduce the number of fingerprints processed.
- Exclude large auto-generated files or test fixtures.

Q: Why are some files missing function-level metrics?

A:
- Function-level analysis requires `lizard`. Install it: `pip install lizard`.
- Some languages may not be fully supported by `lizard`.

Q: Differ shows files as "Modified" but I didn't change them.

A:
- Check if line endings changed (CRLF ↔ LF).
- Verify the file wasn't reformatted by an auto-formatter.
- PyUCC uses content hashing—any byte-level change triggers "Modified" status.

Q: How do I reset all baselines?

A:
- Baselines are stored in the `baseline/` folder (default).
- Delete the baseline folder or specific baseline subdirectories to reset.

Q: Can I run PyUCC in CI/CD pipelines?

A:
- Yes! Use the CLI mode:
`bash python -m pyucc differ create /path/to/repo python -m pyucc differ diff <baseline_id> /path/to/repo python -m pyucc duplicates /path/to/repo --threshold 5.0`
- Parse the JSON output or text reports in your pipeline scripts.
