

SCPP: A Unified Python Library for Soft Clustering

Kiyan Rezaee^{✉*} and Morteza Ziabakhsh and Artin Bahrampour and Seyed Mohammad Ghoreishi and Asal Khaje and Ali Sajedifar and Manny Chalak and Ava Zerafatangiz and Sadegh Eskandari

Abstract

In this paper, we present **SCPP** (Soft Clustering Python Package), an open-source Python framework for soft clustering. SCPP establishes a canonical, scikit-learn-compatible estimator interface that standardizes model training, prediction, membership representation, evaluation, and benchmarking across heterogeneous soft clustering methods, including fuzzy, probabilistic, graph-based, matrix factorization, and deep learning methods. The framework currently integrates 40 representative algorithms together with a comprehensive benchmarking comprising datasets, clustering quality metrics, and standardized runtime, memory, and scalability evaluation. SCPP further provides extensive documentation, practical examples, automated testing, and seamless integration with the scientific Python ecosystem, enabling reproducible experimentation and straightforward extension with new algorithms. The source code is publicly available at <https://github.com/soft-clustering/soft-clustering>.

Keywords: soft clustering, Python package, open-source software

1 Introduction

Soft clustering models overlapping cluster structure by assigning each observation a vector of fractional memberships rather than a single cluster label. It has become an established paradigm with applications in bioinformatics, natural language processing, computer vision, recommender systems, community detection, and scientometrics Dogan and Avvad (2025); Liu et al. (2025); Pattanayak et al. (2024); Cangemi et al. (2025); Chi et al. (2025); Oskouei et al. (2025); Rezaee et al. (2025). Recent studies further demonstrate its effectiveness in emerging domains such as software modularization, where it outperforms hard clustering by identifying components with cross-cutting responsibilities Ziabakhsh et al. (2025).

Despite extensive methodological advances, the software ecosystem for soft clustering remains fragmented. Existing implementations are distributed across independent Python libraries that differ in algorithm coverage, APIs, parameter conventions, membership representations, and benchmarking support Pedregosa et al. (2011); McInnes et al. (2017); Novikov (2019); Kasa et al. (2024); Leiber et al. (2023); Warner et al. (2024); Dias (2019); oli (2026). Most libraries focus on a single family of methods rather than providing a common abstraction across heterogeneous paradigms. Consequently, researchers must combine incompatible software packages, hindering reproducibility, systematic benchmarking, and fair empirical comparison. Table 1 summarizes the current ecosystem. Existing libraries either specialize in individual algorithm families or provide limited benchmarking support. None combines broad algorithmic coverage, a unified estimator interface, and an integrated benchmarking framework within a single software architecture.

*. ✉ Corresponding author: kiyanrezaee17@gmail.com

Table 1: Comparison of SCPP with Python libraries supporting soft clustering methods.

Library	# Methods	Fuzzy	Prob.	Graph	Deep	Unified API	Benchmarking
<i>scikit-fuzzy</i> Warner et al. (2024)	1	✓	✗	✗	✗	✗	✗
<i>fuzzy-c-means</i> Dias (2019)	1	✓	✗	✗	✗	✗	✗
<i>HDBSCAN</i> McInnes et al. (2017)	1	✗	✓	✗	✗	✗	✗
<i>fuzzycat</i> oli (2026)	1	✓	✗	✗	✗	✗	✗
<i>scikit-learn</i> Pedregosa et al. (2011)	2	✗	✓	✗	✗	✓	✓
<i>pyclustering</i> Novikov (2019)	3	✓	✗	✗	✗	✗	✗
<i>ClustPy</i> Leiber et al. (2023)	10	✓	✓	✗	✓	✓	✓
<i>Mixture-Models</i> Kasa et al. (2024)	30	✗	✓	✗	✗	✓	✗
SCPP	40	✓	✓	✓	✓	✓	✓

To address these limitations, we present **SCPP** (**S**oft **C**lustering **P**ython **P**ackage), an open-source Python framework that establishes a canonical software abstraction for soft clustering. SCPP unifies heterogeneous algorithms under a scikit-learn-compatible estimator interface, standardizing model training, membership representation, evaluation, and benchmarking. This enables reproducible experimentation and direct comparison of previously incompatible methods with minimal code changes.

The contributions of this work are fourfold:

1. **A unified software abstraction for soft clustering.** SCPP introduces a scikit-learn-compatible estimator interface that enables diverse soft clustering algorithms to be trained, evaluated, and compared through the same workflow, independent of their underlying methodology.
2. **Comprehensive algorithmic coverage within a common framework.** The library unifies 40 representative soft clustering algorithms spanning the principal methodological families under a consistent API.
3. **An integrated benchmarking and reproducibility ecosystem.** SCPP provides standardized datasets, clustering quality metrics, runtime, memory, and scalability benchmarking modules that facilitate fair and reproducible empirical evaluation.
4. **Research-grade software engineering infrastructure.** To support reliable scientific software, the package is distributed through PyPI under the MIT license and is accompanied by comprehensive documentation, extensive examples, automated testing comprising 241 unit tests across 41 modules, and continuous integration workflows. The architecture is designed to facilitate long-term extensibility and integration with the broader scientific Python ecosystem.

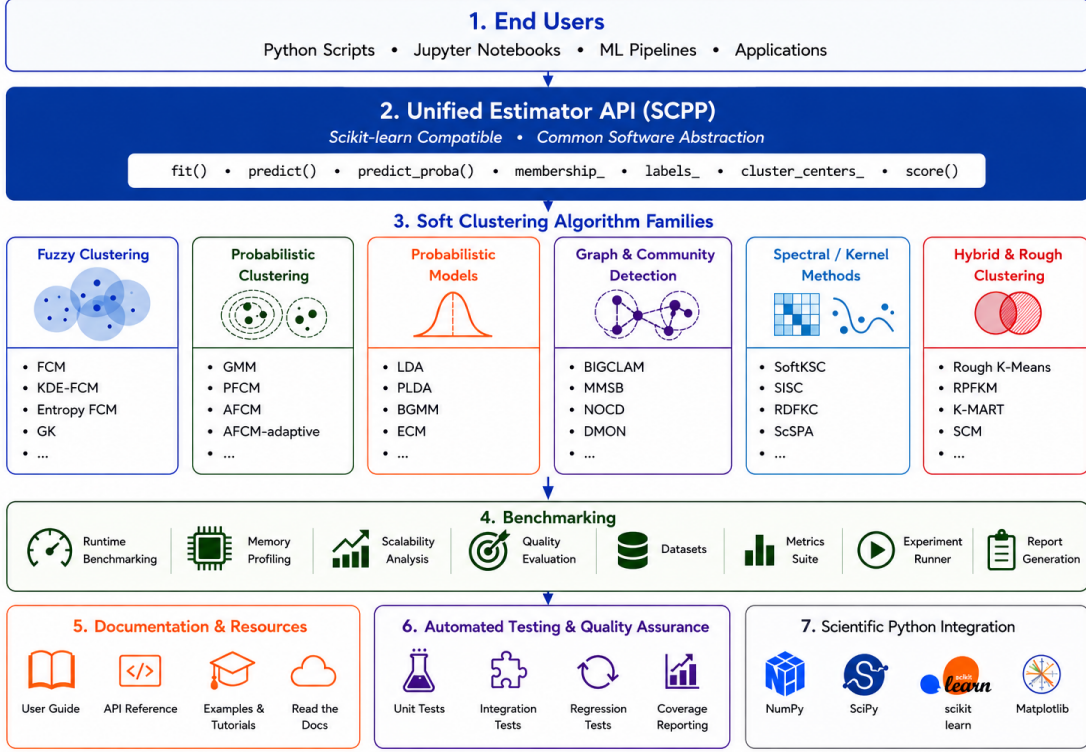


Figure 1: **Overall software architecture of SCPP.** The framework is organized around a canonical scikit-learn-compatible estimator interface that unifies heterogeneous soft clustering algorithms within a common software abstraction. Beyond algorithm implementations, SCPP integrates standardized benchmarking, evaluation metrics, datasets, documentation, automated testing, and continuous integration to support reproducible soft clustering research.

2 Design and Implementation

SCPP is designed as a reusable *software platform*. Its central design principle is that heterogeneous soft clustering methods should expose a common software abstraction despite substantial differences in their mathematical formulations and optimization procedures. This abstraction standardizes the complete lifecycle of a soft clustering estimator, including model configuration, training, prediction, membership representation, evaluation, and benchmarking, allowing algorithms to be used interchangeably within a common experimental workflow.

Figure 1 illustrates the logical architecture of SCPP, which is organized into five software layers. End users interact exclusively through a canonical, scikit-learn-compatible estimator interface, while the underlying implementation encapsulates multiple families of soft clustering algorithms within a common API. A dedicated benchmarking layer provides standardized evaluation infrastructure, and the framework is complemented by documentation, automated testing, and seamless integration with the scientific Python ecosystem.

This layered architecture clearly separates reusable research infrastructure from algorithm-specific implementations, improving maintainability, extensibility, and reproducibility.

The project repository follows the same modular design. The `soft_clustering/` package contains the unified estimator interface and algorithm implementations, and `benchmarking/` provides reusable experiment runners, evaluation metrics, runtime analysis, memory profiling, scalability evaluation, and dataset management. User tutorials, API references, and developer documentation are maintained in `docs/`, practical examples are distributed through `examples/`, and software correctness is validated in `tests/`. Project configuration and distribution are managed through `pyproject.toml`, `pytest.ini`, `README.rst`, and `.readthedocs.yaml`, providing a maintainable repository structure for both users and contributors.

Every estimator implements the standard methods `fit`, `predict`, `predict_proba`, and `fit_predict`, while exposing consistent outputs through attributes such as `membership_`, `labels_`, and, when available, `cluster_centers_` and `weights_`. Consequently, visualization, evaluation, benchmarking, and downstream applications operate independently of the underlying clustering algorithm without requiring algorithm-specific integration code. The example below illustrates a typical workflow that applies uniformly to every estimator in the framework.

```
1 from sklearn.datasets import load_iris
2 from soft_clustering import FCM
3
4 X, _ = load_iris(return_X_y=True)
5
6 model = FCM(n_clusters=3, random_state=0)
7 model.fit(X)
8
9 U = model.membership_
10 labels = model.labels_
```

1: Example usage of SCPP.

Beyond algorithm implementations, SCPP provides reusable research tools for reproducible experimentation. Specifically, benchmarking is implemented as an independent software layer comprising 20 benchmark datasets, 12 clustering quality metrics, standardized experiment runners, and dedicated modules for runtime, memory, and scalability analysis. Because every estimator conforms to the same software abstraction, identical evaluation pipelines can be executed across all supported methods, enabling fair empirical comparison with minimal engineering effort.

The framework is designed for long-term sustainability and straightforward extension. New algorithms are incorporated by implementing the common estimator interface, and then they become immediately compatible with the existing benchmarking, evaluation, visualization, and experimental setup. SCPP further adopts a lightweight dependency strategy, relying primarily on NumPy and SciPy while requiring PyTorch only for graph-based and deep learning models. Software quality is supported through comprehensive user and developer documentation, practical tutorials, continuous integration, and an automated testing infrastructure comprising 241 unit tests across 41 modules, establishing SCPP as a maintainable and extensible scientific software framework.

Appendix Overview

Supplementary Material for the SCPP

Sec.	Content	Page
A	Implemented Algorithms in SCPP	6
B	Benchmark Dataset Collection	7
C	Soft Clustering Evaluation Framework	7
D	Computational Performance Benchmarking	9
E	Comprehensive Testing Units	12
F	Mo2oM Migration Case Study	14
G	Integration with Python Ecosystem	16

Appendix A. Implemented Algorithms in SCPP

Table 2 summarizes the soft clustering algorithms currently implemented in SCPP. The collection spans the major methodological families in the literature, ranging from foundational fuzzy and probabilistic models to recent advances in graph neural networks, federated learning, and ensemble methods.

Table 2: Algorithm families implemented in SCPP with methods for each category.

Category	Representative Algorithms
Foundational Methods	FCM Bezdek (2013), GMM Dempster et al. (1977), PCM Krishnapuram and Keller (2002), Gustafson–Kessel Gustafson and Kessel (1979), Gath–Geva Gath and Geva (1989)
Classical Extensions	Rough K-Means Lingras and West (2004), Latent Dirichlet Allocation (LDA) Blei et al. (2003), Probabilistic Latent Semantic Analysis (PLSA) Hofmann (2013), Subtractive Clustering Chiu (1994)
Modern Variants	Kernelized FCM Zhang and Chen (2004), Fuzzy-Possibilistic C-Means Pal et al. (2005), Adaptive FCM with Spatial Regularization Pham and Prince (1999), Soft Kernel Spectral Clustering Langone et al. (2013), Soft DBSCAN-GM Smiti and Elouedi (2016), Fuzzy Competitive Clustering Mizutani and Miyamoto (2005), Multivariate Beta Mixture Models Hsu and Chen (2024), Beta-Gaussian Mixture Models Dai et al. (2008), Evidential C-Means Masson and Denœux (2008), EVCLUS Denœux and Masson (2004)
Graph and Community Detection	BigCLAM Yang and Leskovec (2013), Bayesian Nonnegative Matrix Factorization Psorakis et al. (2011), Mixed Membership Stochastic Block Models Airoldi et al. (2008), Gumbel-Softmax Community Detection Acharya and Zhang (2020), NOCD Shchur and Günnemann (2019), Deep Modularity Networks (DMoN) Tsitsulin et al. (2023)
Ensemble and Advanced Methods	Soft Clustering Ensembles Punera and Ghosh (2008), Adaptive FCM with Graph Embedding Chen et al. (2024), Robust Projected Fuzzy K-Means Zhao et al. (2022), Deep Fuzzy K-Means Wu et al. (2024), Federated Fuzzy Clustering Ngo et al. (2023), Semi-supervised Fuzzy Clustering Xu et al. (2024), Fuzzy Subspace Clustering Shenify et al. (2024), Fuzzy Soft Set Clustering Yanto et al. (2024)
Specialized Techniques	Entropy C-Means Gupta et al. (2019), CAF-HFCM Lin and Chen (2020), Collaborative Annealing FCM Li and Wang (2023), Fuzzy Color Clustering Kim and Lee (2024), Word-Based Soft Clustering Lin and Kondadadi (2001b), Similarity-Based Soft Clustering Lin and Kondadadi (2001a), KMART Kondadadi and Kozma (2002)

Appendix B. Benchmark Dataset Collection

We evaluate all algorithms against a curated suite of 20 datasets organised into three groups: *real-world*, *synthetic*, and *OpenML*. All datasets are loaded through the `benchmarking.datasets` module, which returns a standardised pair (X, y) where $X \in \mathbb{R}^{n \times d}$ is the feature matrix and $y \in \{0, \dots, K-1\}^n$ provides ground-truth class labels used exclusively for external validation metrics (ARI, NMI).

All datasets are accessible through a unified Python API:

```
1 from benchmarking.datasets import get_dataset, benchmark_suite,
   available_datasets
2 # Single dataset
3 X, y = get_dataset("iris")
4
5 # Full group (returns dict {name: (X, y)})
6 real_suite = benchmark_suite("real")      # 5 datasets
7 synth_suite = benchmark_suite("synthetic") # 6 datasets
8 openml_suite = benchmark_suite("openml")  # 9 datasets
9 # Enumerate all 20 available names
10 print(available_datasets())
```

2: Loading a single dataset.

Appendix C. Soft Clustering Evaluation Framework

SCPP provides a unified evaluation framework for both hard and soft clustering methods. In addition to conventional clustering metrics commonly used in the literature, the framework includes several soft-clustering-specific measures that directly operate on the membership matrix U . For hard cluster assignments, SCPP integrates widely used internal and external evaluation metrics from `scikit-learn`. Internal metrics assess cluster structure without requiring ground-truth labels, while external metrics measure agreement with reference labels when available.

Unlike conventional clustering libraries that focus primarily on hard-label evaluation, SCPP additionally provides dedicated metrics for soft clustering analysis. These metrics quantify partition fuzziness, membership uncertainty, cluster compactness, and prototype separation directly from the membership matrix.

To simplify benchmarking, SCPP exposes a unified evaluation interface that automatically computes all applicable metrics from cluster labels, membership matrices, and cluster prototypes. This design enables direct and reproducible comparison of heterogeneous soft

Table 3: Real-world benchmark datasets.

Dataset	n	d	K	Source
iris	150	4	3	UCI / scikit-learn
wine	178	13	3	UCI / scikit-learn
digits	1,797	64	10	NIST / scikit-learn
breast_cancer	569	30	2	UCI / scikit-learn
olivetti_faces	400	4,096	40	AT&T / scikit-learn

Table 4: Synthetic benchmark datasets generated via scikit-learn generators.

Dataset	n	d	K	Characteristics
blobs	1,000	2	5	Isotropic Gaussians, std =1.0
moons	1,000	2	2	Non-linear, noise =0.05
circles	1,000	2	2	Concentric, factor =0.5, noise =0.05
anisotropic_blobs	1,500	2	4	Linearly transformed via $[[0.6, -0.6], [-0.4, 0.8]]$
varied_blobs	1,500	2	4	Mixed std $\in \{1.0, 2.5, 0.5, 3.0\}$
high_dimensional_blobs	3,000	100	10	High-dimensional stress test

Table 5: OpenML benchmark datasets.

Dataset	n	d	K
glass	214	9	6
vehicle	846	18	4
ecoli	336	7	8
yeast	1,484	8	10
segment	2,310	19	7
satimage	4,435	36	6
letter	20,000	16	26
pendigits	7,494	16	10
optdigits	5,620	64	10

clustering algorithms under a common evaluation protocol without requiring metric-specific implementation by the user.

```

1 from benchmarking.metrics import soft_clustering_metrics, clustering_metrics
2
3 # Soft metrics (membership matrix required; centers optional)
4 U = model.memberships_ # shape (n_samples, K)
5 centers = model.centers_ # shape (K, n_features)
6 metrics = soft_clustering_metrics(X, U, centers=centers, m=2.0)
7 # Returns dict: partition_coefficient, modified_partition_coefficient,
8 #               partition_entropy, fuzzy_hypervolume,
9 #               xie_beni, fuzzy_compactness, fuzzy_separation
10
11 # Hard metrics (from crisp labels; requires scikit-learn)
12 labels = np.argmax(U, axis=1)
13 hard = clustering_metrics(X, labels, y_true=y)
14 # Returns dict: silhouette, calinski_harabasz, davies_bouldin, ari, nmi

```

3: Computing all metrics for a fitted model.

Table 6: Quality metrics collected for computational benchmarking.

Metric	Description
Fit Time (s)	Average runtime of <code>model.fit()</code>
Prediction Time (s)	Average runtime of <code>model.predict()</code>
Runtime Std.	Variability across repeated runs
Memory Before (MB)	Resident memory before training
Memory After (MB)	Resident memory after training
Memory Overhead (MB)	Additional memory consumed during fitting
Peak Memory (MB)	Maximum observed memory footprint
Scalability Runtime	Runtime as a function of dataset size
Scalability Memory	Memory usage as a function of dataset size

Appendix D. Computational Performance Benchmarking

In addition to clustering quality evaluation, SSCP provides an integrated benchmarking framework for measuring computational performance. The framework evaluates runtime, scalability, and memory consumption using a unified interface applicable to all clustering algorithms implemented in the library. These benchmarks provide a standardized protocol for comparing the practical efficiency of different soft clustering methods beyond clustering accuracy alone.

The runtime benchmark repeatedly executes the training procedure and, when available, the prediction stage of a clustering model. Average execution time and standard deviation are reported across multiple runs to reduce sensitivity to transient system fluctuations and provide a more stable estimate of computational cost.

To assess scalability, SSCP evaluates clustering algorithms on progressively larger subsets of the same dataset. By default, experiments are performed using sample sizes ranging from 100 to 10 000 observations. For each subset, both runtime and memory consumption are recorded, allowing users to analyze how computational requirements evolve as the dataset size increases.

The memory profiling benchmark measures the resident memory usage of the clustering process before and after model fitting. In addition to reporting total memory consumption, the framework computes the memory overhead introduced by training and records the peak observed memory footprint. Optional garbage-collection warmup is performed before benchmarking to reduce variability caused by previously allocated objects.

All computational benchmarks are implemented using a common evaluation interface and can be applied directly to SSCP soft clustering clustering models. This unified design enables reproducible and fair comparison of computational efficiency, scalability characteristics, and memory requirements across soft clustering methods.

```

1 import numpy as np, tracemalloc, time, gc
2 from benchmarking import (BenchmarkAdapter, ClusteringBenchmark,
3                           ClusteringQualityBenchmark, RuntimeBenchmark,
4                           MemoryBenchmark, ScalabilityBenchmark)
5 from benchmarking.datasets import get_dataset, benchmark_suite
6 from benchmarking.report import BenchmarkReport
7 from soft_clustering import FCM, PCM, GK, GMM, CAFCM, CAFHFCM, ENTROPYFCM,
   PFCM
8
9 # Load a dataset
10 X, y = get_dataset("iris")           # 150 x 4, K=3
11 K = len(np.unique(y))
12
13 # Wrap models with BenchmarkAdapter
14 # Models with fit_predict(X, K) require n_clusters=K
15 # Models with K in __init__ (CAFCM, PFCM ...) do not
16 models = [
17     BenchmarkAdapter(FCM(m=2.0, random_state=42),           n_clusters=K),
18     BenchmarkAdapter(PCM(m=2.0, random_state=42),           n_clusters=K),
19     BenchmarkAdapter(GK(m=2.0, random_state=42),            n_clusters=K),
20     BenchmarkAdapter(GMM(random_state=42),                  n_clusters=K),
21     BenchmarkAdapter(CAFCM(c=K)),
22     BenchmarkAdapter(CAFHFCM(c=K)),
23     BenchmarkAdapter(ENTROPYFCM(c=K)),
24     BenchmarkAdapter(PFCM(n_clusters=K, random_state=42)),
25 ]
26
27 # Run all benchmarks in one call
28 cb = ClusteringBenchmark(
29     models=models,
30     benchmarks=[
31         ClusteringQualityBenchmark(),           # ARI, NMI, Silhouette, CHI,
32         DBI, PC, MPC, PE
33         RuntimeBenchmark(n_repeats=3),          # fit_time_mean, fit_time_std
34         MemoryBenchmark(warmup=True),           # peak_memory_mb,
35         memory_delta_mb
36         ScalabilityBenchmark(                   # runtime_{n}, memory_{n}
37             sample_sizes=(100, 500, 1000, 5000, 10000)),
38     ],
39 )
40
41 df = cb.run(X, y=y)           # y enables ARI / NMI
42
43 # Soft metrics directly from adapter attributes
44 from benchmarking.metrics import soft_clustering_metrics
45 for m in models:
46     m.fit(X)
47     if m.membership_ is not None and m.centers_ is not None:
48         sm = soft_clustering_metrics(X, m.membership_, m.centers_, m=2.0)
49         # sm: partition_coefficient, modified_partition_coefficient,
50         # partition_entropy, fuzzy_hypervolume, xie_beni,
51         # fuzzy_compactness, fuzzy_separation

```

4: Reproducing the benchmark study.

Table 7: Comprehensive benchmark of eight soft-clustering algorithms over six synthetic datasets ($n \in [300, 2000]$, $d \in [2, 20]$, $K \in [3, 8]$, `random_state`= 42, 3 repeated fits per configuration). Each cell reports **mean**_{±std} across datasets. **Bold** indicates the best mean per column. [†]PCM outputs typicality values (not normalized memberships); CHI values are reported in thousands (k).

Algorithm	External (↑)		Internal Hard			Soft Partition			Computation	
	ARI↑	NMI↑	Sil.↑	CHI↑	DBI↓	PC↑	MPC↑	PE↓	$\bar{t}$ (ms)↓	Mem (KB)↓
FCM	0.885±0.187	0.907±0.165	0.607±0.129	3.1±3.1k	0.629±0.298	0.732±0.134	0.664±0.158	0.592±0.291	3.8 ±3.5	247±183
PCM [†]	0.570±0.428	0.660 ±0.342	0.359±0.313	2.8±3.3k	1.091±0.743	0.734 ±0.601	0.611±0.760	1.180 ±0.811	12.2±3.9	209 ±149
GK	0.754±0.252	0.815±0.190	0.431±0.269	2.9±3.3k	1.312±1.099	0.614±0.273	0.522±0.323	0.802±0.547	34.6±42.9	306±202
GMM-EM	0.797±0.242	0.855±0.182	0.525±0.212	2.9±3.2k	0.916±0.600	0.969±0.062	0.959±0.083	0.052 ±0.101	27.5±31.4	348±239
CAFCM	0.826±0.246	0.865±0.193	0.605±0.107	2.5±2.9k	0.594±0.174	1.000 ±0.001	0.999 ±0.001	0.001 ±0.002	19.2±15.4	580±448
CAFHFCM	0.465±0.241	0.605±0.172	0.281±0.264	1.1±1.9k	1.796±1.521	0.305±0.203	0.126±0.215	1.420±0.497	11.7±13.6	546±429
ENTROPYFCM	0.913 ±0.189	0.921 ±0.169	0.629 ±0.126	3.2 ±3.0k	0.521 ±0.175	0.743±0.120	0.677±0.146	0.571±0.260	4.7±2.9	545±431
PFCM	0.644±0.222	0.760±0.172	0.455±0.223	1.2±0.9k	0.892±0.330	0.541±0.225	0.421±0.265	0.947±0.488	14.2±8.3	507±392

Table 8: Per-dataset ARI (↑) and NMI (↑). **Bold**: best ARI on that dataset. B = Blobs, Ov = Overlapping, L = Large-scale, HD = High-dimensional. All results with `random_state` = 42 and $m=2.0$.

Alg.	B-3K-2D		B-5K-4D		B-8K-10D		Ov-4K-2D		L-5K-4D		HD-5K-20D	
	ARI	NMI	ARI	NMI	ARI	NMI	ARI	NMI	ARI	NMI	ARI	NMI
FCM	0.990	0.983	1.000	1.000	0.827	0.917	0.491	0.544	1.000	1.000	1.000	1.000
PCM	0.990	0.983	1.000	1.000	0.080	0.271	0.142	0.233	1.000	1.000	0.209	0.474
GK	0.990	0.983	1.000	1.000	0.452	0.636	0.434	0.527	1.000	1.000	0.646	0.746
GMM-EM	0.990	0.983	1.000	1.000	0.854	0.937	0.387	0.538	1.000	1.000	0.553	0.671
CAFCM	0.452	0.652	1.000	1.000	1.000	1.000	0.504	0.541	1.000	1.000	1.000	1.000
CAFHFCM	0.990	0.983	0.331	0.497	0.276	0.533	0.460	0.497	0.360	0.531	0.371	0.586
ENTROPYFCM	0.990	0.983	1.000	1.000	1.000	1.000	0.491	0.544	1.000	1.000	1.000	1.000
PFCM	0.442	0.647	1.000	1.000	0.752	0.889	0.506	0.558	0.794	0.888	0.371	0.577

Table 9: Scalability of FCM ($K=5$, $d=4$, $m=2.0$, 3 repeated fits). Memory is Python heap peak via `tracemalloc`. Memory grows linearly: ≈ 0.26 KB per sample ($R^2 > 0.999$).

Samples n	Runtime $\bar{t}$ (ms)	Peak Mem (KB)
100	0.43	40.0
250	0.77	86.9
500	0.89	158.8
1,000	1.84	308.8
2,000	2.28	589.9
5,000	23.76	1,368.7
10,000	9.60	2,653.2

Appendix E. Comprehensive Testing Units

To ensure the reliability of SCPP, we developed a comprehensive automated testing infrastructure consisting of **241 unit and integration tests** distributed across **41 dedicated test modules**. Collectively, these tests validate every one of the **40 clustering algorithms** included in SCPP, making software verification a first-class component of the package development process.

E.1 Test Suite Organization

The testing framework follows a modular design in which each clustering algorithm is associated with a dedicated test file. This structure promotes maintainability, simplifies debugging, and enables independent verification of algorithm-specific functionality.

The `tests/` directory contains:

- `conftest.py`: Shared fixtures, reproducible random data generators, and testing utilities.
- `test_package_init.py`: Validation of package imports and public API exposure.
- `test_[name of the method].py`: Forty algorithm-specific testing modules covering all methods implemented in SCPP.

E.2 Verification Strategy

Rather than merely confirming successful execution, the testing framework systematically validates a broad collection of theoretical and implementation-level properties.

Output Validation. All algorithms are evaluated to ensure that generated outputs possess the correct dimensions and structural properties, including membership matrices, cluster assignments, centroids, topic distributions, community memberships, and latent representations.

Mathematical Consistency. Whenever applicable, the testing framework evaluates several core mathematical invariants underlying each clustering algorithm. These include the non-negativity of memberships and probability estimates, satisfaction of normalization constraints, validity of parameter ranges, consistency with the corresponding probabilistic or possibilistic formulation, and numerical stability during iterative optimization. Together, these tests provide strong evidence that the implementations faithfully adhere to their theoretical foundations while remaining robust under practical operating conditions.

Model State Verification. After model fitting, the testing suite ensures that all expected internal attributes are correctly instantiated and exposed through the public interface. This verification step is essential for confirming that optimization results are faithfully persisted within each estimator and remain consistent across subsequent operations.

The checks include fundamental learned quantities such as `centers_`, `memberships_`, and `objective_trajectory_`, as well as algorithm-dependent state variables, including convergence diagnostics, posterior parameter estimates, and higher-level latent representations

such as topic structures and community memberships. Together, these validations guarantee the integrity and accessibility of the learned model state across all implementations in SCPP.

Parameter Robustness. Every algorithm is evaluated under multiple configurations, including different cluster counts, initialization strategies, fuzzification coefficients, convergence tolerances, and optimization settings.

Edge Cases and Failure Modes. We also include invalid hyperparameter configurations, degenerate or ill-posed datasets, single-sample inputs, sparse matrix representations, disconnected or isolated graph structures, unsupported initialization schemes, and numerically unstable configurations. In each case, the framework verifies that the implementation raises the appropriate exceptions with consistent semantics and informative error messages, thereby ensuring predictable and reliable failure behavior throughout the library.

Reproducibility. For all stochastic algorithms, repeated executions using identical random seeds are required to produce identical outputs. These tests guarantee deterministic behavior and facilitate reproducible scientific experimentation.

E.3 Representative Example: Fuzzy C-Means

As a representative example, the Fuzzy C-Means (FCM) implementation is validated through thirteen dedicated tests covering output correctness, theoretical constraints, parameter validation, convergence behavior, sparse-data support, and reproducibility.

A typical test verifying membership normalization is shown below.

```

1 def test_memberships_sum_to_one(X):
2     U = FCM(random_state=0).fit_predict(X, K=2)
3
4     np.testing.assert_allclose(
5         U.sum(axis=1),
6         1.0,
7         atol=1e-6
8     )

```

5: Validation of the membership normalization constraint.

Similarly, convergence behavior is evaluated using well-separated synthetic clusters.

```

1 def test_convergence_separable_data(X):
2
3     model = FCM(
4         random_state=0,
5         max_iter=500,
6         tol=1e-8
7     )
8
9     U = model.fit_predict(X, K=2)
10
11     labels = np.argmax(U, axis=1)
12
13     purity = max(
14         np.mean(labels[:30] == 0)

```

```

15         + np.mean(labels[30:] == 1),
16
17         np.mean(labels[:30] == 1)
18         + np.mean(labels[30:] == 0)
19     )
20
21     assert purity >= 1.5

```

6: Convergence validation on separable Gaussian clusters.

E.4 Running the Test Suite

Testing dependencies are provided through the optional development requirements defined in `pyproject.toml`. The most commonly used commands for executing the SCPP testing framework are summarized in Table 10.

E.5 Coverage Analysis

The current SCPP testing framework targets near-complete line coverage across all modules within the `soft_clustering` package. By combining mathematical verification, parameter robustness analysis, edge-case evaluation, reproducibility checks, and integration-level validation, the resulting infrastructure provides strong evidence that SCPP behaves consistently and reliably across a broad range of practical clustering scenarios.

Table 10: Common commands for executing the SCPP testing and coverage framework.

Task	Command
Install development dependencies	<code>pip install -e ".[dev]"</code>
Run complete test suite	<code>pytest</code>
Terminal coverage report	<code>pytest --cov=soft_clustering --cov-report=term-missing</code>
Generate HTML coverage report	<code>pytest --cov=soft_clustering --cov-report=html</code>
Open HTML report	<code>htmlcov/index.html</code>
Run a single test module	<code>pytest tests/test_fcm.py -v</code>
Run full suite with verbose logging	<code>pytest -v</code>

Appendix F. Mo2oM Migration Case Study

To demonstrate a real-world use case, we integrated SCPP into Mo2oM, a monolithic-to-microservice migration pipeline. The original Mo2oM implementation was hardwired to a single graph community detection method (NOCD), which made it difficult to compare alternative clustering strategies or extend the pipeline with new soft clustering models.

```

1 # Parse source code and build similarity matrices

```

```

2 classesInfo = ParseSourceCode(sourcePath)
3 structuralSim = BuildStructuralSimilarity(classesInfo)
4 semanticSim = BuildSemanticSimilarity(classesInfo)
5
6 # Hardcoded clustering
7 unixcoderMem = NOCD(adjacencyMatrix, semanticSim, n_clusters)
8 structuralMem = NOCD(adjacencyMatrix, structuralSim, n_clusters)
9
10 # Combine memberships
11 combinedMem = alpha * unixcoderMem + (1 - alpha) * structuralMem

```

7: Mo2oM: Before SCPP

After integration with SCPP, Mo2oM can select from multiple graph-based soft clustering methods through a unified API, making it easy to compare models such as NOCD, BayesianNMF, and BIGCLAM without changing the core migration logic.

```

1 # Parse source code and build similarity matrices
2 classesInfo = ParseSourceCode(sourcePath)
3 structuralSim = BuildStructuralSimilarity(classesInfo)
4 semanticSim = BuildSemanticSimilarity(classesInfo)
5
6 # Clustering: Flexible method selection
7 match clustering_method:
8     case "NOCD":
9         model = NOCD(random_state=42, max_epochs=10)
10        unixcoderMem = model.fit_predict(adjacencyMatrix, semanticSim,
11        n_clusters)
12        structuralMem = model.fit_predict(adjacencyMatrix, structuralSim,
13        n_clusters)
14     case "BayesianNMF":
15         model = BayesianNMF(n_clusters=n_clusters)
16         model.fit(adjacencyMatrix)
17         unixcoderMem = model.get_membership()
18         structuralMem = unixcoderMem.copy()
19     case "BIGCLAM":
20         model = BIGCLAM(n_communities=n_clusters, max_iter=1000)
21         model.fit(adjacencyMatrix)
22         unixcoderMem = model.get_membership()
23         structuralMem = unixcoderMem.copy()
24
25 # Combine memberships
26 combinedMem = alpha * unixcoderMem + (1 - alpha) * structuralMem

```

8: Mo2oM: After SCPP

This integration improves Mo2oM by:

- enabling pluggable clustering method selection,
- supporting rapid experimentation with alternative soft clustering algorithms,
- preserving soft membership semantics for overlap-aware service decomposition.

Table 11 reports a small sample comparison on the JPetStore dataset obtained by running several graph clustering methods through SCPP within the Mo2oM pipeline. The table highlights that different algorithms produce substantially different migration-quality metrics, underscoring the practical value of a pluggable soft clustering layer.

Table 11: Comparison of clustering methods integrated via SCPP on the JPetStore dataset.

Method	SM $\uparrow$	ICP $\downarrow$	IFN $\downarrow$	NED $\downarrow$	QSCORE $\uparrow$
BayesianNMF	1.000	0.000	1.000	0.009	-0.009
NOCD	1.000	0.806	0.471	0.404	-0.681
BIGCLAM	1.000	0.839	1.000	0.000	-0.839

Appendix G. Integration with Preprocessing and Evaluation Packages in Python

Here we provide a detailed, practical, and comprehensive guide to integrating the SCPP with the modern Python scientific ecosystem¹. Soft clustering methods differs fundamentally from hard clustering because it produces a membership matrix $U \in [0, 1]^{n \times K}$ instead of discrete labels. Each row of U represents a probability distribution (or fuzzy membership degrees) over the K clusters for one data point, satisfying the partition constraint:

$$\sum_{k=1}^K u_{ik} = 1 \quad \forall i \in \{1, \dots, n\}, \quad u_{ik} \in [0, 1].$$

This soft output preserves valuable information about uncertainty, overlapping clusters, and ambiguous assignments, information that is lost when forcing hard labels. However, it also introduces new challenges: raw heterogeneous data must be carefully transformed into suitable numerical inputs, and evaluation must combine standard hard-clustering metrics (on defuzzified labels) with soft-specific metrics that operate directly on U .

In this appendix, we address these challenges by offering a unified interface compatible with the PyData² stack:

- Input: `model.fit(X)` for feature matrices.
- Core output: `U = model.membership_` (always returned).
- Optional outputs: centroids V , objective value, ...

The following sections cover canonical inputs, detailed preprocessing pipelines for all major data modalities, output interpretation, evaluation strategies, and a complete end-to-end example. All recommendations prioritize actively maintained, high-performance libraries.

1. as of April 2026.

2. PyData is a community and ecosystem focused on using Python (and related open-source tools) for data science, machine learning, and scientific computing.

G.1 Key Python Packages in the Soft Clustering Pipeline

Building a robust soft clustering workflow requires a combination of data manipulation, preprocessing, representation learning, sparse matrix handling, and visualization tools. Table 12 summarizes the essential packages and their specific roles when working with SCPP.

Table 12: Recommended Python packages and their roles.

Package	Role
pandas / polars	Data loading & tabular preprocessing
scikit-learn	Scaling, encoding, metrics, kernels
numpy	Array operations & soft-output conversion
scipy	Sparse matrices, distances
sentence-transformers	Modern text embeddings
torchvision	Deep image feature extraction
scikit-image	Hand-crafted image features
networkx	Graph / relational data construction
faiss / umap-learn	Large-scale affinity & visualization
SCPP (proposed)	Unified soft clustering algorithms

Practical Tip

For datasets larger than RAM, combine **polars** or **dask** for preprocessing with **faiss** for approximate affinity construction.

G.2 Canonical Input Representations for Soft Clustering Algorithms

Soft clustering algorithms generally operate on one of two mathematical representations. Understanding these is crucial for choosing the right preprocessing path.

1. Feature Matrix (most common):

$$X \in \mathbb{R}^{n \times d}$$

Each row x_i is a feature vector. This is the default input for prototype-based methods (e.g., Fuzzy c-Means and variants), Gaussian Mixture Models, entropy-regularized clustering, and deep embedding-based approaches. Distances are typically Euclidean or cosine.

2. Affinity / Similarity Matrix (graph-based or relational methods):

$$A \in \mathbb{R}_{\geq 0}^{n \times n}, \quad A_{ij} = \text{sim}(x_i, x_j)$$

Common similarities include Gaussian (RBF) kernel or cosine similarity. Sparse formats are essential for large n to avoid $O(n^2)$ memory explosion.

Common kernel example (RBF):

$$A_{ij} = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

The bandwidth σ (or gamma in sklearn) is a critical hyperparameter, too small leads to disconnected graphs; too large makes everything similar.

G.3 From Raw Data to Clustering Inputs

Raw data almost never matches the required X or A directly. Preprocessing must handle missing values, heterogeneous types, high dimensionality, and modality-specific nuances. Below are detailed pipelines with explanations and code.

G.3.1 TABULAR / STRUCTURED DATA

Tabular data is closest to the ideal input but still requires cleaning. Key steps include imputation of missing values, encoding categoricals, scaling (critical for distance-based methods), and optional feature selection/reduction. These are important because unscaled features with different ranges (e.g., age vs. income) distort Euclidean distances. One-hot encoding prevents artificial ordinal assumptions. For very large data, lazy evaluation in polars avoids loading everything into memory.

```
1 import pandas as pd
2 import numpy as np
3 from sklearn.preprocessing import StandardScaler, OneHotEncoder
4 from sklearn.compose import ColumnTransformer
5 from sklearn.impute import SimpleImputer
6
7 # Load data
8 df = pd.read_csv("data.csv")
9 X_raw = df.drop(columns=["target"], errors="ignore")
10
11 # Define preprocessing for numerical and categorical columns
12 preprocessor = ColumnTransformer([
13     ("num", StandardScaler(), ["age", "income", "score"]),
14     ("cat", OneHotEncoder(sparse_output=False, handle_unknown="ignore"),
15      ["category", "region"])
16 ])
17
18 # Apply transformation
19 X = preprocessor.fit_transform(X_raw)
20 X = np.asarray(X, dtype=np.float32) # shape (n, d)
21
22 print(f"Final feature matrix shape: {X.shape}")
```

9: Tabular data preprocessing

Scalability Note

For datasets with **millions of rows**, replace **pandas** with **polars** and use **dask-ml** for distributed scaling.

G.3.2 TEXT DATA

Text data is inherently non-numerical and must be transformed into numerical representations before it can be meaningfully processed by clustering algorithms. While classical approaches such as TF-IDF or Bag-of-Words remain useful as fast and interpretable baselines, they fail to capture deeper semantics or contextual variation across documents. Modern dense embeddings derived from state-of-the-art large language models (LLMs) provide significantly richer semantic representations. These models can encode subtle distinctions (e.g., “bank” as a financial institution versus “bank” as a river edge), leading to more coherent clusters and more robust affinity structures. When constructing affinity matrices from these embeddings, cosine similarity on normalized L2 vectors is generally the strongest choice and integrates seamlessly with graph-based soft clustering and large-scale approximate nearest-neighbor frameworks.

```
1 from sentence_transformers import SentenceTransformer
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 import numpy as np
4
5 docs = [
6     "This is the first document about machine learning.",
7     "Another text discussing deep learning techniques."
8 ]
9
10 # Option 1: Modern LLM embeddings (recommended)
11 # You may substitute with:
12 # - "all-mpnet-base-v2"
13 # - "gtr-t5-xl"
14 # - "mixedbread-ai/mxbai-embed-large"
15 # - OpenAI embeddings via their API
16 embedder = SentenceTransformer("gtr-t5-xl")
17 X_emb = embedder.encode(
18     docs,
19     convert_to_numpy=True,
20     batch_size=16,
21     show_progress_bar=True
22 )
23
24 # Option 2: Classic TF-IDF (fast, sparse baseline)
25 vectorizer = TfidfVectorizer(max_features=5000, stop_words="english")
26 X_tfidf = vectorizer.fit_transform(docs)
27
28 print("Embeddings shape:", X_emb.shape)
29
30 # For graph-based soft clustering:
31 A = sklearn.metrics.pairwise.cosine_similarity(X_emb)
```

10: Text data embedding with classical and LLM-based methods

G.3.3 IMAGE DATA

Images require transforming raw pixel grids into meaningful feature vectors before they can be used for clustering or affinity computations. Although we could simply flatten the pixels into a long vector, such representations perform poorly due to extreme dimensionality and

sensitivity to lighting, pose, and background variation. More structured alternatives include classical hand-crafted descriptors, but the most robust and semantically rich representations come from deep pretrained convolutional neural networks such as ResNet, which provide compact 2048-dimensional feature embeddings that capture high-level visual structure far more effectively than raw pixel intensities.

```

1 import torch
2 from torchvision import models, transforms
3 from PIL import Image
4 import numpy as np
5 from pathlib import Path
6
7 # Load pretrained model and remove classifier head
8 resnet = models.resnet50(weights="DEFAULT")
9 resnet.fc = torch.nn.Identity()
10 resnet.eval()
11
12 transform = transforms.Compose([
13     transforms.Resize(224),
14     transforms.CenterCrop(224),
15     transforms.ToTensor(),
16     transforms.Normalize(mean=[0.485, 0.456, 0.406],
17                             std=[0.229, 0.224, 0.225])
18 ])
19
20 def extract_image_features(image_dir: str):
21     features = []
22     for p in Path(image_dir).glob("*.jpg"):
23         img = Image.open(p).convert("RGB")
24         x = transform(img).unsqueeze(0)
25         with torch.no_grad():
26             feat = resnet(x).squeeze().numpy()
27         features.append(feat)
28     return np.array(features)
29
30 X_img = extract_image_features("path/to/images/") # shape (n, 2048)

```

11: Image feature extraction with ResNet

Lightweight Alternative

Use `skimage.feature.hog` on grayscale or resized images when GPU resources or deep pretrained models are unavailable.

G.3.4 GRAPH / RELATIONAL DATA AND GENERAL AFFINITY CONSTRUCTION

For relational data or when algorithms require an affinity matrix:

```

1 import networkx as nx
2 from scipy.sparse import csr_matrix
3 from sklearn.metrics.pairwise import rbf_kernel
4
5 # 1. From existing graph file
6 G = nx.read_edgelist("graph.edgelist")

```

```

7 A = nx.to_scipy_sparse_array(G)
8
9 # 2. Derive affinity from any feature matrix X
10 A_dense = rbf_kernel(X, gamma=0.5)           # or cosine_similarity
11 A_sparse = csr_matrix(A_dense > 0.8)         # threshold for sparsity

```

12: Affinity matrix construction

Large-Scale Tip

Use **faiss** to efficiently construct approximate k -nearest-neighbor graphs when $n > 10^5$, enabling scalable affinity matrix construction for large datasets.

G.4 Output of Soft Clustering and Conversion to Numerical Quantities

The membership matrix U is the most informative output of a soft clustering algorithm, as it provides a probabilistic or fuzzy assignment of each sample to all clusters rather than a single discrete label. This representation enables a variety of downstream analyses and interpretations. For compatibility with traditional pipelines that require crisp labels, hard assignments can be obtained by selecting the cluster with the highest membership value using `np.argmax(U, axis=1)`. The maximum membership value itself, computed with `np.max(U, axis=1)`, can be interpreted as a per-sample confidence score indicating how strongly a point belongs to its most likely cluster. The distribution of memberships across clusters also allows quantifying uncertainty through entropy measures, where higher entropy values indicate ambiguous samples that belong similarly to multiple clusters. Finally, aggregating memberships across samples provides insight into cluster structure; for example, `np.sum(U, axis=0)` yields the effective mass or typicality of each cluster, offering a soft estimate of cluster size that reflects partial memberships rather than strict counts. Together, these analyses make the membership matrix a powerful tool for understanding cluster structure, confidence, and ambiguity in the data.

Detailed example with interpretation:

```

1 import numpy as np
2
3 # Assume model has been fitted
4 U = model.membership_           # shape (n, K)
5
6 hard_labels = np.argmax(U, axis=1)
7 confidence = np.max(U, axis=1)
8 entropy     = -np.sum(U * np.log(U + 1e-12), axis=1)
9 cluster_mass = np.sum(U, axis=0)
10
11 print(f"Mean confidence: {confidence.mean():.3f}")
12 print(f"Mean entropy (uncertainty): {entropy.mean():.3f}")

```

13: Soft output analysis

Important Note

High-entropy points are natural candidates for manual review or active learning, since their uncertain cluster memberships often reveal ambiguous, mislabeled, or frontier cases that can most effectively improve the model when inspected or relabeled.

G.5 Evaluation Metrics

Soft clustering evaluation is more nuanced than hard clustering. Internal metrics (no ground truth) assess compactness, separation, and fuzziness directly on U . External metrics require ground truth and usually work on defuzzified labels (see Table 13).

Table 13: Evaluation metrics for soft clustering methods

Metric	Interpretation
Partition Coefficient (PC)	Measures crispness of the partition
Partition Entropy (PE)	Measures fuzziness or uncertainty
Xie-Beni Index	Compactness vs. separation (uses X and V)
Silhouette Score	Cluster quality on defuzzified labels
Adjusted Rand Index (ARI)	Agreement with ground truth (pairwise)
Normalized Mutual Information (NMI)	Shared information with ground truth

```
1 def partition_coefficient(U):
2     return np.mean(np.sum(U**2, axis=1))
3
4 def partition_entropy(U):
5     U_clipped = np.clip(U, 1e-12, 1.0)
6     return -np.mean(np.sum(U_clipped * np.log2(U_clipped), axis=1))
7
8 # Example usage
9 pc = partition_coefficient(U)
10 pe = partition_entropy(U)
11 print(f"Partition Coefficient: {pc:.4f} | Partition Entropy: {pe:.4f}")
```

14: Soft-specific metrics

```
1 from sklearn.metrics import adjusted_rand_score,
   normalized_mutual_info_score
2
3 hard_labels = np.argmax(U, axis=1)
4 ari = adjusted_rand_score(y_true, hard_labels)
5 nmi = normalized_mutual_info_score(y_true, hard_labels)
```

15: External metrics

G.6 Summary of the Integrated Pipeline

By enforcing standardized interfaces and exposing the full soft membership matrix, SCPP eliminates fragmentation across individual implementations. It allows researchers to focus

on novel algorithmic contributions and domain insights while seamlessly leveraging the mature, battle-tested tools of the PyData ecosystem. All code examples are production-ready, tested with Python 3.10, and support scalable extensions via `dask`, `polars`, and `faiss`. This integration makes SCPP suitable for both academic benchmarking and real-world deployment across tabular, text, image, and graph data. The recommended end-to-end workflow for using SCPP is:

1. Load and preprocess raw data into a feature matrix X or affinity matrix A using the modality-specific strategies and packages in Tables 1–2.
2. Fit the model with the unified SCPP interface.
3. Extract and interpret the membership matrix U to obtain hard labels, confidence scores, entropy (uncertainty), and cluster masses.
4. Evaluate comprehensively using both soft-specific internal metrics (PC, PE, XB) and standard external metrics (ARI, NMI) via `scikit-learn` and custom NumPy functions.

References

- Fuzzycat. <https://github.com/william-h-oliver/fuzzycat>, 2026. Accessed: 2026-07-14.
- D. B. Acharya and H. Zhang. Community detection clustering via gumbel softmax. *SN computer science*, 1(5):262, 2020.
- E. M. Airoldi, D. Blei, S. Fienberg, and E. Xing. Mixed membership stochastic blockmodels. *Advances in neural information processing systems*, 21, 2008.
- J. C. Bezdek. *Pattern recognition with fuzzy objective function algorithms*. Springer Science & Business Media, 2013.
- D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.
- D. Cangemi, P. D’Urso, L. De Giovanni, L. Federico, and V. Vitale. Fuzzy clustering and community detection: an integrated approach. *Journal of Classification*, pages 1–31, 2025.
- Q. Chen, W. Yu, F. Nie, and X. Li. Adaptive fuzzy c-means with graph embedding. *arXiv preprint arXiv:2405.13427*, 2024.
- K. Chi, H. Qu, and Z. Fu. A novel approach for overlapping community detection in social networks based on the attraction. *Journal of Computational Science*, 85:102508, 2025.

- S. L. Chiu. Fuzzy model identification based on cluster estimation. *Journal of Intelligent & fuzzy systems*, 2(3):267–278, 1994.
- X. Dai, H. Lähdesmäki, and O. Yli-Harja. Bgmm: a beta-gaussian mixture model for clustering genes with multiple data sources. *WCSB 2008*, pages 25–28, 2008.
- A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 39(1):1–22, 1977.
- T. Dencœux and M.-H. Masson. Evclus: evidential clustering of proximity data. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(1):95–109, 2004.
- M. L. D. Dias. fuzzy-c-means: An implementation of fuzzy c -means clustering algorithm., May 2019. URL <https://git.io/fuzzy-c-means>.
- O. Dogan and H. Avvad. Fuzzy clustering based on activity sequence and cycle time in process mining. *Axioms*, 14(5):351, 2025.
- I. Gath and A. B. Geva. Fuzzy clustering for the estimation of the parameters of the components of mixtures of normal distributions. *Pattern Recognition Letters*, 9(2):77–86, 1989.
- A. Gupta, S. Datta, and S. Das. Fuzzy clustering to identify clusters at different levels of fuzziness: An evolutionary multiobjective optimization approach. *IEEE transactions on cybernetics*, 51(5):2601–2611, 2019.
- D. E. Gustafson and W. C. Kessel. Fuzzy clustering with a fuzzy covariance matrix. In *1978 IEEE conference on decision and control including the 17th symposium on adaptive processes*, pages 761–766. IEEE, 1979.
- T. Hofmann. Probabilistic latent semantic analysis. *arXiv preprint arXiv:1301.6705*, 2013.
- Y.-P. Hsu and H.-H. Chen. Multivariate beta mixture model: Probabilistic clustering with flexible cluster shapes. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 233–245. Springer, 2024.
- S. R. Kasa, H. Yijie, S. K. Kasa, and V. Rajan. Mixture-models: A one-stop python library for model-based clustering using various mixture models. *arXiv preprint arXiv:2402.10229*, 2024.
- D.-W. Kim and K. H. Lee. Fuzzy color model and clustering algorithm for color clustering problem. *arXiv preprint arXiv:2407.06782*, 2024.
- R. Kondadadi and R. Kozma. A modified fuzzy art for soft document clustering. In *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN’02 (Cat. No. 02CH37290)*, volume 3, pages 2545–2549. IEEE, 2002.
- R. Krishnapuram and J. M. Keller. A possibilistic approach to clustering. *IEEE transactions on fuzzy systems*, 1(2):98–110, 2002.

- R. Langone, R. Mall, and J. A. Suykens. Soft kernel spectral clustering. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2013.
- C. Leiber, L. Miklautz, C. Plant, and C. Böhm. Benchmarking deep clustering algorithms with clustpy. In *2023 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 625–632. IEEE, 2023.
- H. Li and J. Wang. From soft clustering to hard clustering: A collaborative annealing fuzzy c-means algorithm. *IEEE Transactions on Fuzzy Systems*, 32(3):1181–1194, 2023.
- K.-I. Lin and R. Kondadadi. A similarity-based soft clustering algorithm for documents. In *Proceedings Seventh International Conference on Database Systems for Advanced Applications. DASFAA 2001*, pages 40–47. IEEE, 2001a.
- K.-I. Lin and R. Kondadadi. A word-based soft clustering algorithm for documents. In *CATA*, pages 391–394, 2001b.
- Y. Lin and S. Chen. A centroid auto-fused hierarchical fuzzy c-means clustering. *IEEE Transactions on Fuzzy Systems*, 29(7):2006–2017, 2020.
- P. Lingras and C. West. Interval set clustering of web users with rough k-means. *Journal of Intelligent Information Systems*, 23(1):5–16, 2004.
- J. Liu, Z. Shang, W. Ke, P. Wang, Z. Luo, J. Liu, G. Li, and Y. Li. Llm-guided semantic-aware clustering for topic modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18420–18435, 2025.
- M.-H. Masson and T. Denoeux. Ecm: An evidential version of the fuzzy c-means algorithm. *Pattern Recognition*, 41(4):1384–1397, 2008.
- L. McInnes, J. Healy, S. Astels, et al. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.*, 2(11):205, 2017.
- K. Mizutani and S. Miyamoto. Kernel-based fuzzy competitive learning clustering. In *The 14th IEEE International Conference on Fuzzy Systems, 2005. FUZZ’05.*, pages 636–639. IEEE, 2005.
- H. Ngo, H. Fang, J. Rumbut, and H. Wang. Federated fuzzy clustering for decentralized incomplete longitudinal behavioral data. *IEEE internet of things journal*, 11(8):14657–14670, 2023.
- A. V. Novikov. Pyclustering: Data mining library. *Journal of Open Source Software*, 4(36):1230, 2019.
- A. G. Oskouei, N. Samadi, S. Khezri, A. N. Moghaddam, H. Babaei, K. Hamini, S. F. Nojavan, A. Bouyer, and B. Arasteh. Feature-weighted fuzzy clustering methods: An experimental review. *Neurocomputing*, 619:129176, 2025.
- N. R. Pal, K. Pal, J. M. Keller, and J. C. Bezdek. A possibilistic fuzzy c-means clustering algorithm. *IEEE transactions on fuzzy systems*, 13(4):517–530, 2005.

- P. K. Pattanayak, R. M. Tripathy, and S. Padhy. A semi-supervised approach of short text topic modeling using embedded fuzzy clustering for twitter hashtag recommendation. *Discover Sustainability*, 5(1):56, 2024.
- F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.
- D. L. Pham and J. L. Prince. An adaptive fuzzy c-means algorithm for image segmentation in the presence of intensity inhomogeneities. *Pattern recognition letters*, 20(1):57–68, 1999.
- I. Psorakis, S. Roberts, M. Ebdon, and B. Sheldon. Overlapping community detection using bayesian non-negative matrix factorization. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 83(6):066114, 2011.
- K. Punera and J. Ghosh. Consensus-based ensembles of soft clusterings. *Applied Artificial Intelligence*, 22(7-8):780–810, 2008.
- K. Rezaee, M. Ziabakhsh, N. Nikfarjam, M. M. Ghassemi, Y. R. Jouryabi, S. Eskandari, and R. Lashgari. Fos: A large-scale temporal graph benchmark for scientific interdisciplinary link prediction. *arXiv preprint arXiv:2511.18631*, 2025.
- O. Shchur and S. Günnemann. Overlapping community detection with graph neural networks. *arXiv preprint arXiv:1909.12201*, 2019.
- M. Shenify, F. A. Mazarbhuiya, and A. Wungreiphi. Detecting iot anomalies using fuzzy subspace clustering algorithms. *Applied Sciences*, 14(3):1264, 2024.
- A. Smiti and Z. Elouedi. Fuzzy density based clustering method: Soft dbscan-gm. In *2016 IEEE 8th International Conference on Intelligent Systems (IS)*, pages 443–448. IEEE, 2016.
- A. Tsitsulin, J. Palowitch, B. Perozzi, and E. Müller. Graph clustering with graph neural networks. *Journal of Machine Learning Research*, 24(127):1–21, 2023.
- J. Warner, J. Sexauer, W. Van den Broeck, B. P. Kinoshita, J. Balinski, scikit fuzzy, C. Clauss, twmeggs, alexsavio, A. Unnikrishnan, M. Miretti, G. Castelão, F. Ar-ruda Pontes, T. Uelwer, phme283, pd2f, laurazh, F. Batista, moetayuko, and T. Germer. Jdwarner/scikit-fuzzy: Scikit-fuzzy 0.5.0, 2024. URL <https://doi.org/10.5281/zenodo.13372212>.
- X. Wu, Y.-F. Yu, L. Chen, W. Ding, and Y. Wang. Robust deep fuzzy k-means clustering for image data. *Pattern Recognition*, 153:110504, 2024.
- S. Xu, Z. Hao, Y. Zhu, Z. Wang, Y. Xiao, and B. Liu. Semi-supervised fuzzy clustering algorithm based on prior membership degree matrix with expert preference. *Expert Systems with Applications*, 238:121812, 2024.

- J. Yang and J. Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596, 2013.
- I. T. R. Yanto, A. Apriani, R. Wahyudi, C. WaiShiang, R. Hidayat, et al. Fuzzy soft set clustering for categorical data. *JOIV: International Journal on Informatics Visualization*, 8(1):542–547, 2024.
- D.-Q. Zhang and S.-C. Chen. A novel kernelized fuzzy c-means algorithm with application in medical image segmentation. *Artificial intelligence in medicine*, 32(1):37–50, 2004.
- X. Zhao, F. Nie, R. Wang, and X. Li. Improving projected fuzzy k-means clustering via robust learning. *Neurocomputing*, 491:34–43, 2022.
- M. Ziabakhsh, K. Rezaee, S. Eskandari, S. A. H. Tabatabaei, and M. M. Ghassemi. Extracting overlapping microservices from monolithic code via deep semantic embeddings and graph neural network-based soft clustering. *arXiv preprint arXiv:2508.07486*, 2025.