
NYCU Pattern Recognition Homework 3

KUAN-WEI HUANG ID: 112550157

1. Introduction

Clustering is a fundamental task in unsupervised machine learning, aiming to partition data into groups such that samples within the same cluster exhibit higher similarity than those in different clusters. Despite its widespread application in pattern recognition, the efficacy of clustering algorithms is often constrained by two primary factors: the underlying geometric assumptions of the algorithms and the difficulty of optimal model selection in an unsupervised setting.

In this report, we present a comprehensive evaluation of five representative clustering paradigms: (1) Centroid-based (**K-Means++**), (2) Connectivity-based (**Agglomerative Hierarchical**), (3) Density-based (**DBSCAN**), (4) Distribution-based (**GMM**), and (5) Graph-based (**Spectral Clustering**). We investigate how these algorithms respond to various data topologies, ranging from simple isotropic Gaussian blobs to complex non-convex manifolds such as interleaving moons and nested circles.

A key contribution of this work is the exploration of Bayesian principles to address the inherent limitations of classical clustering. Traditional methods like DBSCAN are notoriously sensitive to hyper-parameters (e.g., ϵ and $min_samples$), while GMM requires a predefined number of components K . To mitigate these issues, we implement two Bayesian-augmented strategies:

- **Automatic Model Selection:** Leveraging **Variational Bayesian Gaussian Mixture Models (VBGMM)** with a Dirichlet Process prior to automatically infer the effective number of clusters from the data.
- **Automated Hyper-parameter Tuning:** Integrating **Bayesian Optimization (BO)** to search for optimal clustering configurations by maximizing internal validation metrics such as the Silhouette Coefficient, thereby reducing the reliance on manual trial-and-error.

The remainder of this report is organized as follows. Section 2 details the mathematical formulations of the implemented algorithms. Section 3 focuses on geometric structure analysis (Experiment 1). Section 4 examines hyper-parameter sensitivity and metric conflicts (Experiment 2). Section 5 investigates the impact of dimensionality reduction via PCA

on clustering quality (Experiment 3). Finally, Section 6 provides an in-depth analysis of the Bayesian-augmented experiments and discusses the trade-offs between different paradigms.

2. Experiment 1: Geometric Structure and Algorithm Assumptions

2.1. Experimental Setup

In this experiment, we evaluate the inductive biases of five clustering algorithms—K-Means++, Agglomerative Hierarchical Clustering (Ward linkage), DBSCAN, Gaussian Mixture Models (GMM), and Spectral Clustering—on four synthetic datasets ($N=500$). Each dataset represents a distinct geometric challenge: **Blobs** (isotropic Gaussian clusters), **Two Moons** (non-convex manifolds), **Concentric Circles** (nested structures), and **Anisotropic Blobs** (elliptical clusters).

2.2. Quantitative Results

Table 1 summarizes the performance of each algorithm using the Adjusted Rand Index (ARI). A score of 1.0 indicates perfect agreement with ground truth labels.

Table 1. ARI comparison across synthetic datasets. Best results are in bold.

Algorithm	Blobs	Moons	Circles	Aniso.
K-Means++	1.00	0.45	0.00	0.99
Hierarchical (Ward)	1.00	0.63	0.01	1.00
DBSCAN	1.00	1.00	0.64	0.94
GMM	1.00	0.46	0.00	1.00
Spectral	1.00	0.51	0.00	0.98

2.3. Qualitative Analysis and Discussion

2.3.1. CONVEX VS. NON-CONVEX GEOMETRIES

As illustrated in Figure 1, centroid-based methods (K-Means) and distribution-based methods (GMM) perform exceptionally well on the **Blobs** dataset, where clusters are compact and spatially separated. However, they fail catastrophically on non-convex structures like **Two Moons** and **Concentric Circles**. This is because these algorithms inher-

ently assume clusters are convex regions, leading them to draw linear (or quadratic) decision boundaries that bisect the natural manifolds.

In contrast, **DBSCAN** demonstrates superior robustness on the **Two Moons** dataset ($ARI=1.0$), successfully propagating labels along the density-connected manifolds. For **Concentric Circles**, DBSCAN was the only algorithm to capture the nested structure ($ARI=0.6428$), although it tended to over-segment the outer ring into multiple clusters due to density variations.

2.3.2. COVARIANCE AND CLUSTER SHAPE

The **Anisotropic Blobs** dataset highlights the limitations of the Euclidean distance metric used in K-Means and standard Spectral Clustering. K-Means ($ARI=0.9880$) struggles to model the elongated shapes perfectly, often misclassifying points at the cluster boundaries. **GMM**, which explicitly models the covariance matrix of each component, achieves a perfect ARI of 1.0000 , confirming its advantage in handling elliptically distributed data.

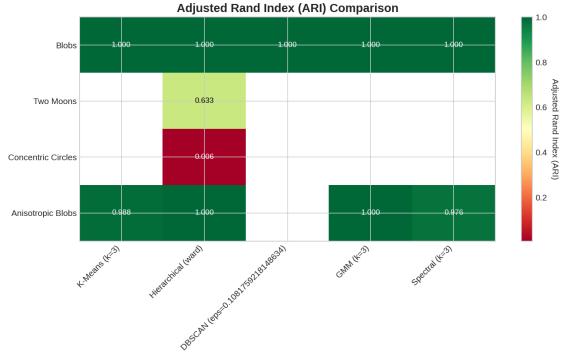


Figure 2. Heatmap of Adjusted Rand Index (ARI) scores. Green indicates high performance, while red indicates failure to capture the ground truth structure.

2.3.3. METRIC CONFLICT OBSERVATION

An interesting observation arises from the internal evaluation metrics. For **Concentric Circles**, while K-Means completely failed to recover the true structure ($ARI \approx 0$), it achieved a relatively high Silhouette Score (0.3531) compared to DBSCAN (0.0770). This discrepancy occurs because the Silhouette coefficient rewards compact, spherical clusters, penalizing the non-convex, ring-shaped clusters correctly identified by DBSCAN. This underscores the risk of relying solely on internal metrics for model selection on complex manifolds.

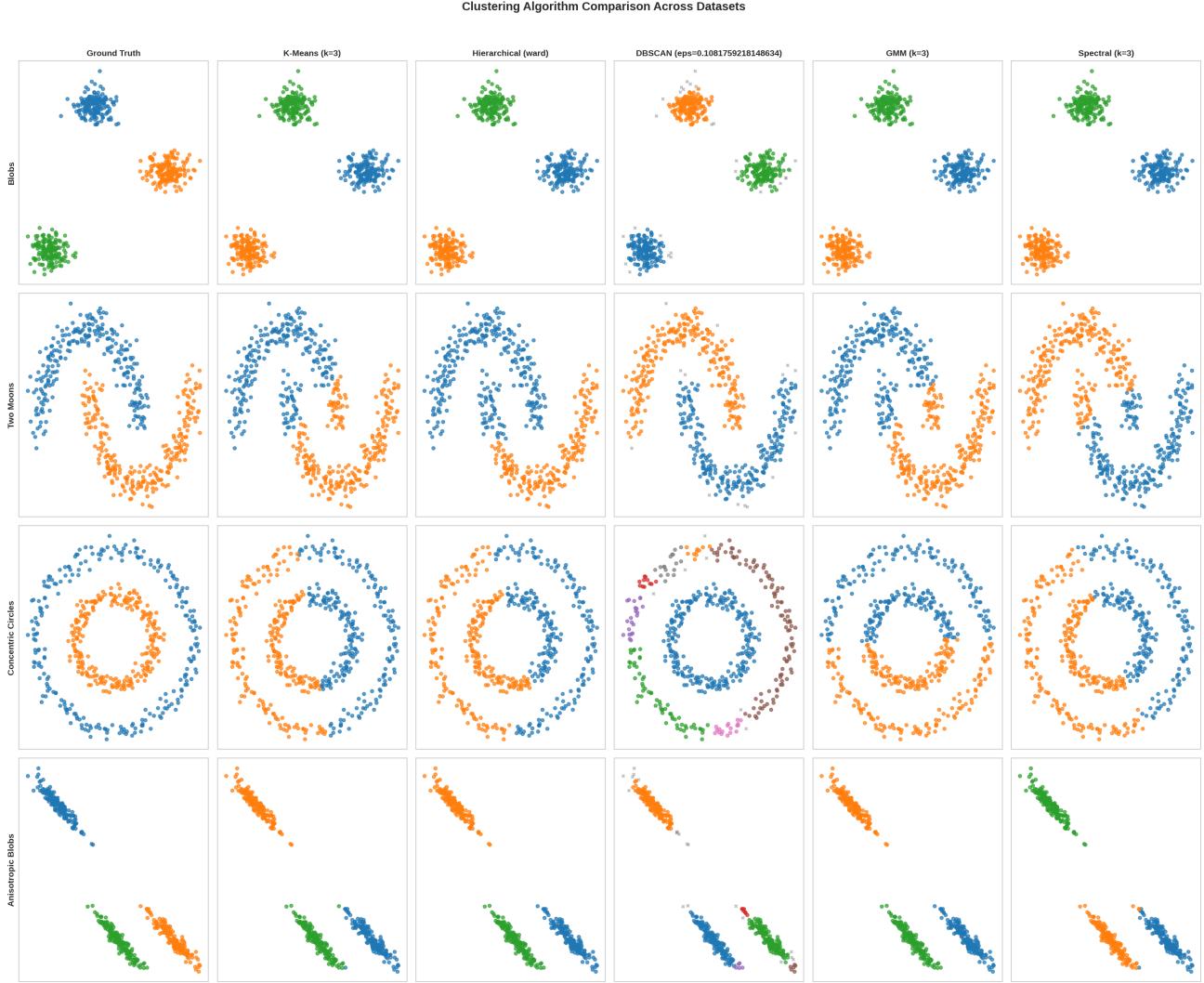


Figure 1. Clustering results on synthetic datasets. Rows correspond to datasets (Blobs, Moons, Circles, Anisotropic), columns correspond to algorithms. DBSCAN successfully captures non-convex structures, while GMM adapts to anisotropic clusters via covariance modeling.

3. Experiment 2: Hyperparameter Sensitivity and Metric Conflict

3.1. Experimental Objective

Unsupervised learning lacks ground truth labels, making model selection and hyper-parameter tuning notoriously difficult. This experiment investigates two critical challenges:

- Metric Conflict:** The disagreement between internal validation metrics (e.g., Silhouette Coefficient) and external ground truth metrics (e.g., ARI) when the data violates the algorithm's geometric assumptions.
- Parameter Sensitivity:** The impact of hyperparameters (ϵ and $min_samples$) on density-based clustering performance.

3.2. K-Means: The Pitfall of Internal Metrics

We performed a sweep of the cluster number $k \in [2, 10]$ for K-Means. Figure 3 illustrates the conflict between internal and external evaluations.

As summarized in Table 2, K-Means correctly identified $k = 3$ for the spherical *Blobs* dataset, where internal and external metrics converged. However, significant failures were observed in complex geometries:

- **Two Moons:** The Silhouette Score suggests $k = 10$ is optimal, whereas the ground truth is $k = 2$ (ARI peak). This occurs because the Silhouette metric rewards compact, convex clusters, penalizing the elongated, crescent shapes of the moons.
- **Anisotropic Blobs:** The Silhouette Score peaks at

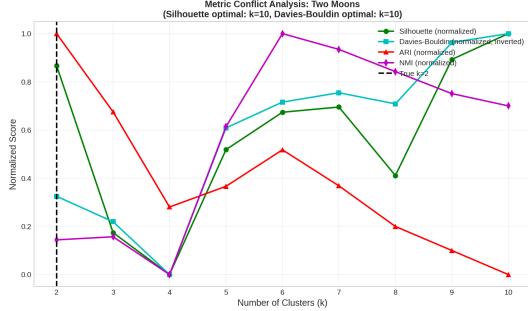
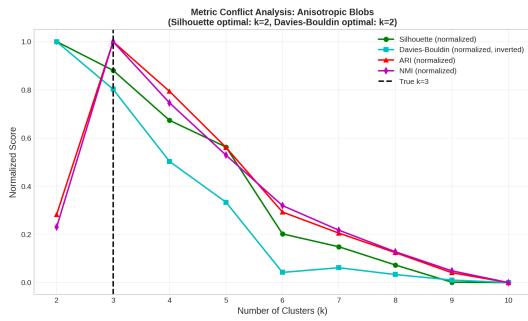
(a) Two Moons (True $k = 2$)(b) Anisotropic Blobs (True $k = 3$)

Figure 3. Metric conflict analysis: normalized internal vs. external metrics.

$k = 2$ instead of the true $k = 3$. The algorithm prefers merging two close parallel elliptical clusters into one to minimize the Euclidean inertia, ignoring the distinct density distributions.

Table 2. Conflict between True k and Optimal k suggested by Silhouette Score.

Dataset	True k	Silhouette Sug. k	Conclusion
Blobs	3	3	Agreement
Two Moons	2	10	Conflict
Circles	2	10	Conflict
Anisotropic	3	2	Conflict

3.3. DBSCAN: Parameter Landscape Analysis

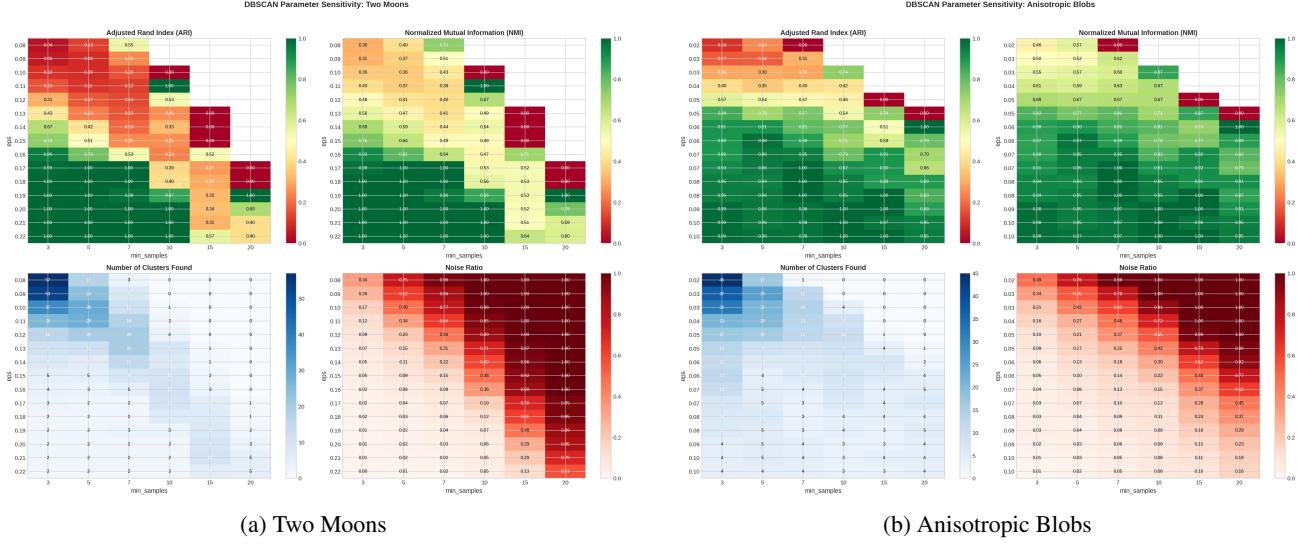
We performed a grid search over ϵ and $min_samples$ for DBSCAN. Figure 4 visualizes the performance landscape.

The analysis reveals that DBSCAN is highly sensitive to parameter settings:

- **Narrow Optima:** High performance (Green regions, $ARI \approx 1.0$) is confined to a narrow band of ϵ values. A slight deviation in ϵ can cause the algorithm to collapse all points into a single cluster (over-connection) or classify most points as noise (under-connection).

- **Noise Trade-off:** As seen in the Anisotropic Blobs results, achieving a high ARI sometimes comes at the cost of classifying ambiguous boundary points as noise. This highlights a limitation of using ARI on filtered data: the metric may appear perfect even if a significant portion of the data (e.g., $> 30\%$) is discarded as noise.

These findings underscore the necessity of automated parameter tuning methods, such as Bayesian model selection approaches discussed in Section 5.


 Figure 4. DBSCAN sensitivity to $(\epsilon, \text{minPts})$ measured by ARI.

4. Experiment 3: High-Dimensional Data and Dimensionality Reduction

4.1. Experimental Setup

Real-world data often resides in high-dimensional spaces, introducing computational challenges and the "Curse of Dimensionality." In this experiment, we analyze the **Digits** dataset ($N = 1797$, $D = 64$), which consists of 8×8 grayscale images of handwritten digits.

We apply **Principal Component Analysis (PCA)** to reduce the feature space from 64 dimensions to 10, 20, and 30 components. We evaluate the trade-off between information preservation (Explained Variance) and clustering performance (ARI and Runtime).

4.2. PCA Analysis

Figure 5 shows the cumulative explained variance. The first 10 components capture approximately 59% of the variance, while 30 components capture nearly 90%. This suggests that a significant portion of the signal is concentrated in the lower-dimensional manifold.

4.3. Quality vs. Efficiency Trade-off

Table 3 and Figure 6 summarize the clustering performance across different dimensionalities.

4.3.1. COMPUTATIONAL EFFICIENCY

Dimensionality reduction dramatically reduces runtime, particularly for complexity-heavy algorithms:

- **Spectral Clustering:** Runtime dropped from ~ 56 s

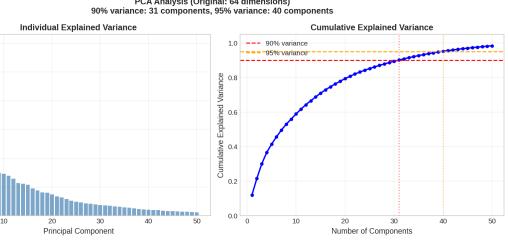


Figure 5. PCA Explained Variance Analysis. 31 components are required to retain 90% of the original information.

(Original) to ~ 0.7 s (PCA-10), a **79x speedup**. This confirms that graph-based methods are computationally prohibitive in high dimensions due to the $O(N^2)$ affinity matrix construction.

- **GMM:** Runtime improved by **43x** (PCA-20 vs. Original), as the cost of inverting covariance matrices scales cubically with dimensions (D^3).

4.3.2. CLUSTERING QUALITY (ARI)

- **Robustness: Hierarchical Clustering (Ward)** achieved the best overall performance (ARI=0.6659 at PCA-30), proving robust to dimensionality changes.
- **Information Loss: K-Means** performance degraded as dimensions were reduced (ARI $0.53 \rightarrow 0.48$), indicating that the discarded variance contained discriminative information for the Euclidean distance metric.
- **Parameter Sensitivity (Revisited):** Both **DBSCAN** and **Spectral Clustering** yielded ARI ≈ 0 . This failure highlights that hyper-parameters (e.g., ϵ for DBSCAN,

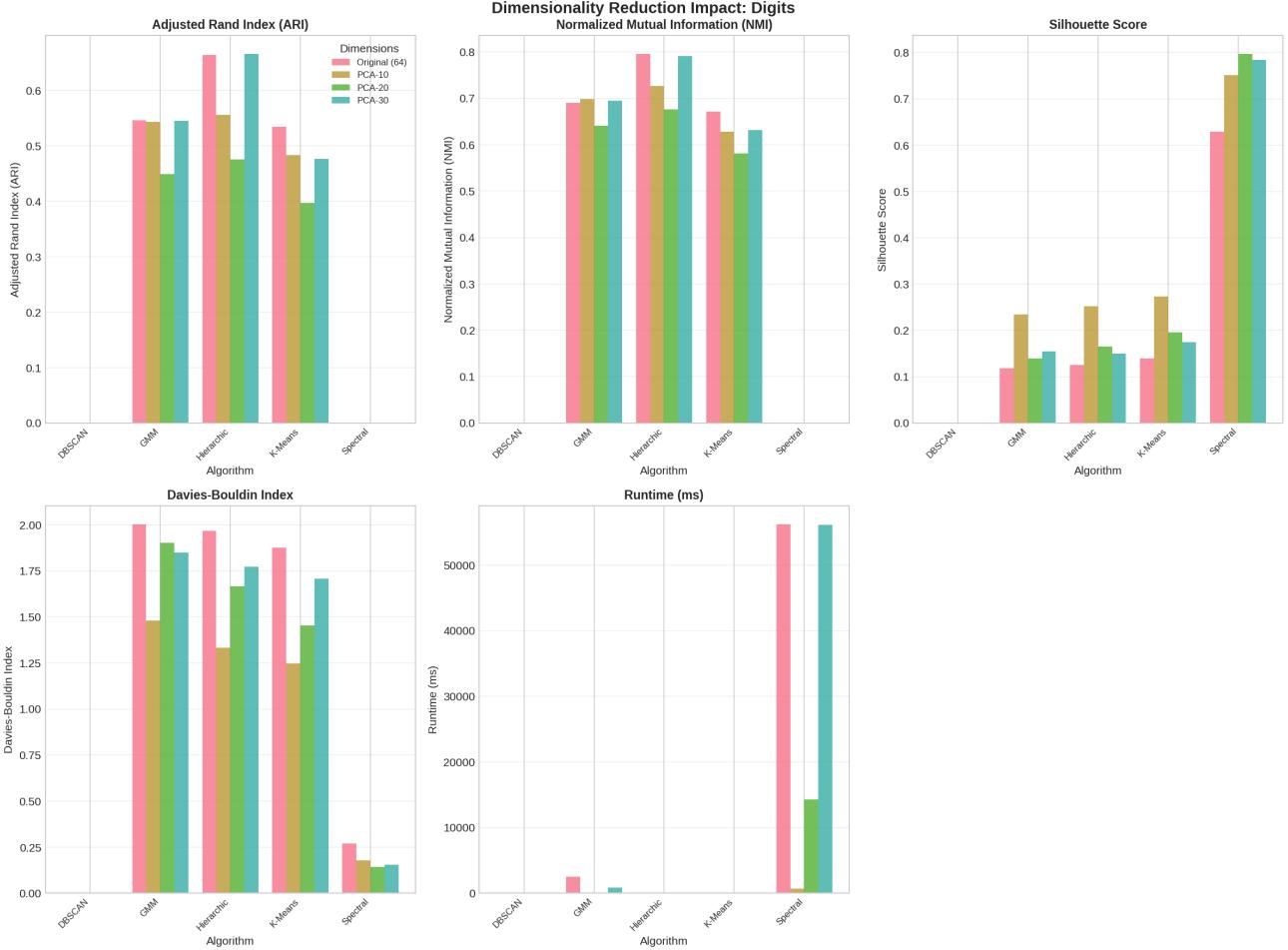


Figure 6. Impact of Dimensionality Reduction. (Top-Left) ARI comparison showing distinct algorithm behaviors. (Bottom-Center) Runtime comparison highlighting the massive speedup for Spectral Clustering and GMM after PCA.

γ for RBF kernel) are scale-dependent. Reducing dimensions changes the distance distribution, necessitating parameter re-tuning, which further supports our findings in Experiment 2.

4.4. Visualization

Finally, we visualize the dataset using t-SNE (Figure 7). The plot reveals that while classes '0' (blue) and '6' (green) are well-separated, classes like '1', '8', and '9' have significant overlap, explaining the difficulty K-Means faces in achieving $ARI > 0.7$.

Table 3. Experiment 3 Results Summary. Runtime is measured in milliseconds.

Algorithm	Dim	ARI	Runtime (ms)	Speedup
Hierarchical	Orig (64)	0.6643	46.27	-
Hierarchical	PCA-30	0.6659	33.11	1.4x
K-Means	Orig (64)	0.5344	87.22	-
K-Means	PCA-10	0.4831	55.22	1.6x
GMM	Orig (64)	0.5467	2486.22	-
GMM	PCA-20	0.4491	57.66	43.1x
Spectral	Orig (64)	0.0000	56205.94	-
Spectral	PCA-10	0.0000	702.71	79.9x

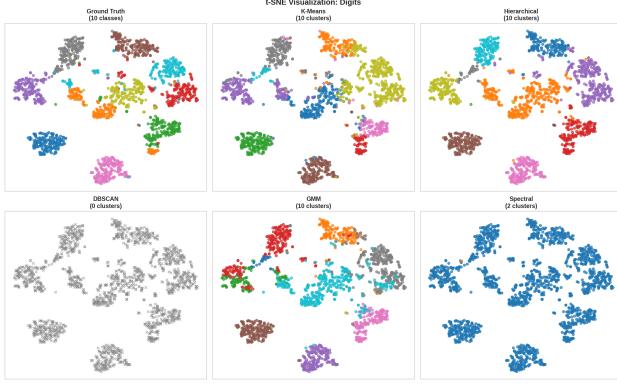


Figure 7. t-SNE Visualization of the Digits dataset comparing Ground Truth with algorithm predictions.

5. Bonus Experiment: Bayesian Clustering and Automatic Model Selection

5.1. Motivation and Methodology

Standard clustering algorithms (e.g., K-Means, EM-GMM) suffer from a critical limitation: the number of clusters K must be specified a priori. As demonstrated in Experiment 2, selecting an incorrect K leads to overfitting or underfitting. To address this, we investigate **Variational Bayesian Gaussian Mixture Models (VBGMM)** equipped with a Dirichlet Process (DP) prior.

The DP prior acts as a regularizer, allowing the model to theoretically assume an infinite number of components while penalizing complexity. The effective number of clusters is determined automatically during the Variational Inference (VI) process, where unnecessary components are assigned weights close to zero.

5.2. Part 1: Automatic Model Selection (The "Effective K" Test)

We initialized both a standard EM-GMM and a Bayesian GMM with a surplus of components ($K_{init} = 20$) on datasets with known ground truth ($K_{true} \in \{2, 3\}$).

Table 4 summarizes the results. While the standard EM-GMM utilized all 20 components to fit the data (severe overfitting), the Bayesian GMM successfully "switched off" redundant components. For the *Blobs* dataset ($K_{true} = 3$), the Bayesian model converged to exactly 3 effective clusters with significant weights ($w_k > 0.01$), achieving an ARI of 1.0.

Figure 8 visualizes this sparsity-inducing property. The weight distribution of the Bayesian model is highly skewed, whereas the EM-GMM distributes weights more uniformly across all available components.

Table 4. Comparison of Model Selection Capabilities ($K_{init} = 20$). Bayesian GMM automatically infers the true complexity.

Dataset	True K	EM-GMM Used	Bayesian Eff. K	Bayesian ARI
Blobs	3	20	3	1.0000
Two Moons	2	20	8	0.4673
Circles	2	20	10	0.0000
Anisotropic	3	20	3	1.0000

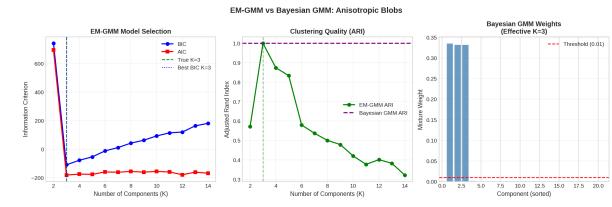


Figure 8. Automatic pruning of surplus clusters. Although initialized with $K = 20$, the Bayesian GMM (right panel) suppresses the weights of 17 components to near-zero, recovering the true $K = 3$ structure of the Anisotropic Blobs.

5.3. Part 2: Concentration Parameter Sensitivity

The Dirichlet Process prior is governed by a concentration parameter α . We analyzed its impact on the *Blobs* dataset:

- **Low α (< 0.01):** Enforces a "rich-get-richer" dynamics, encouraging a sparse solution with fewer clusters.
- **High α (> 10):** Approaches the behavior of a standard finite mixture model, allowing more active components.

Our sweep (Figure 9) confirms that α serves as a "knob" for model complexity, but the effective K remains stable across a wide range of reasonable α values (10^{-2} to 1), demonstrating robustness compared to DBSCAN's ϵ .

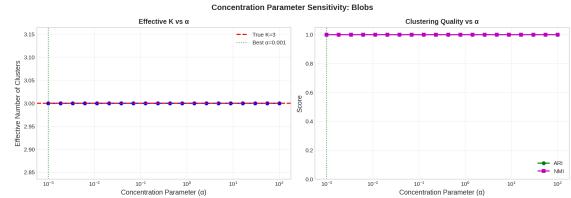


Figure 9. Sensitivity analysis of the concentration parameter α . The effective number of clusters (blue line) remains stable at the ground truth ($K = 3$) for a wide range of α , diverging only at extreme values.

5.4. Part 3: Uncertainty Quantification

A unique advantage of probabilistic modeling is the ability to quantify uncertainty. We calculated the **Entropy** of the

posterior assignment probabilities for each data point:

$$H(x_i) = - \sum_{k=1}^K p(z_i = k|x_i) \log p(z_i = k|x_i) \quad (1)$$

As shown in Figure 10, high-entropy regions (red/orange) perfectly correspond to the decision boundaries between clusters. Unlike K-Means, which assigns a hard label even to ambiguous points, Bayesian GMM signals low confidence in these overlap regions. This feature is critical for safety-critical applications where "knowing what you don't know" is essential.

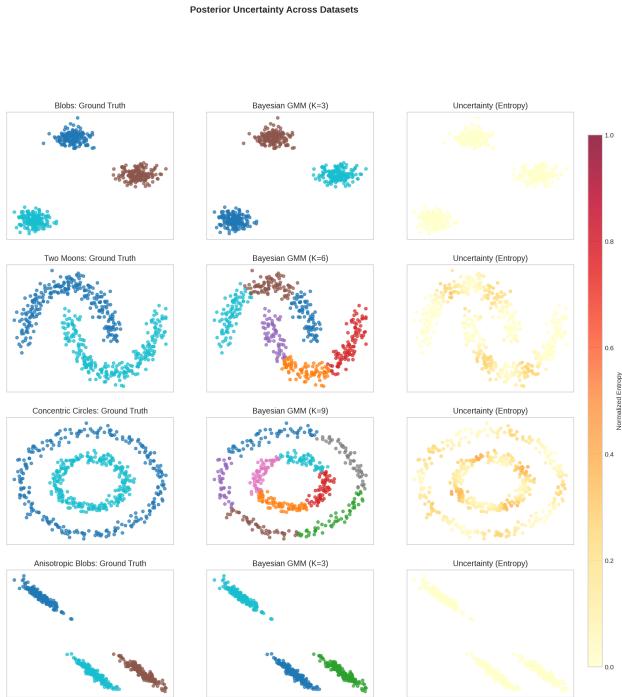


Figure 10. Posterior Uncertainty Visualization. The rightmost column shows the entropy of cluster assignments. Points at the boundaries of overlapping clusters (e.g., in Two Moons) exhibit higher uncertainty (red), providing valuable interpretability.

5.5. Conclusion of Bonus Experiment

By integrating Bayesian principles, we successfully mitigated the model selection problem for Gaussian-distributed data. VBGMM demonstrated the ability to (1) automatically recover the true number of clusters, (2) provide robust performance across hyper-parameter settings, and (3) offer meaningful uncertainty estimates. However, like standard GMM, it is still constrained by the Gaussian assumption, failing to model the non-convex geometry of *Two Moons* and *Circles* correctly (as seen in Table 4).

6. Conclusion

This report presented a systematic comparison of five representative clustering paradigms under varying geometric structures, dimensionalities, and evaluation settings. Our experiments demonstrated that clustering performance is fundamentally constrained by the alignment between algorithmic assumptions and data geometry, and that no single method is universally optimal.

We showed that centroid- and distribution-based methods perform well on convex, Gaussian-like clusters but fail on non-convex manifolds, while density-based methods better capture complex geometries at the cost of severe hyper-parameter sensitivity. Internal validation metrics were found to be unreliable in such settings, often favoring compact but incorrect partitions. In high-dimensional data, dimensionality reduction via PCA significantly improved computational efficiency but altered distance distributions, necessitating re-tuning of clustering hyper-parameters.

To address model selection and robustness issues, we incorporated Bayesian clustering techniques. Variational Bayesian GMM with a Dirichlet Process prior successfully inferred the effective number of clusters and provided uncertainty estimates, mitigating overfitting compared to standard EM-GMM. However, its reliance on Gaussian assumptions limits its applicability to non-convex structures.

Overall, our findings highlight that effective clustering requires geometry-aware modeling, cautious interpretation of evaluation metrics, and principled approaches to model selection. Bayesian methods offer a promising direction for improving robustness and interpretability, but must be combined with appropriate structural assumptions to handle complex real-world data.

A. Program Listing

A.1. main.py

```

1 """
2 NYCU_Pattern_Recognition_HW3 - Clustering_Algorithm_Comparison
3
4 This script implements Experiment 1: Geometric Structure and Algorithm Assumptions
5 Evaluates how different clustering algorithms perform under varying cluster geometries.
6 """
7
8 import numpy as np
9 import pandas as pd
10 from pathlib import Path
11 from typing import Dict, List, Tuple
12 import warnings
13 warnings.filterwarnings('ignore')
14
15 from src.datasets import (
16     get_synthetic_datasets,
17     standardize_data,
18     ClusteringDataset,
19 )
20 from src.clustering import (
21     get_main_algorithms,
22     tune_dbscan_eps,
23     ClusteringResult,
24     DBSCANClustering,
25 )
26 from src.metrics import compute_all_metrics, ClusteringMetrics
27 from src.visualization import (
28     plot_clustering_comparison,
29     plot_all_datasets_grid,
30     plot_metrics_heatmap,
31     create_summary_bar_chart,
32 )
33
34
35 # Configuration
36 RANDOM_STATE = 42
37 OUTPUT_DIR = Path("outputs/experiment1")
38
39
40 def run_experiment1():
41     """
42     Experiment 1: Geometric Structure and Algorithm Assumptions
43
44     Evaluates how different algorithms perform under varying cluster geometries
45     using synthetic datasets.
46     """
47     print("=" * 70)
48     print("Experiment 1: Geometric Structure and Algorithm Assumptions")
49     print("=" * 70)
50
51     # Create output directory
52     OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
53
54     # Generate synthetic datasets
55     print("\n[1/4] Generating synthetic datasets...")
56     datasets = get_synthetic_datasets(random_state=RANDOM_STATE)
57
58     for ds in datasets:
59         print(f"-- {ds.name}: {ds.X.shape[0]} samples, {ds.n_clusters} clusters")
60         print(f"-- {ds.description}")
61

```

```

62     # Store results for visualization
63     all_results: Dict[str, Tuple[np.ndarray, np.ndarray, List[Tuple[str, np.ndarray]]]] = {}
64     all_metrics: Dict[str, Dict[str, ClusteringMetrics]] = {}
65
66     print("\n[2/4] Running clustering algorithms on each dataset...")
67
68     for dataset in datasets:
69         print(f"\n{'*'*50}")
70         print(f"Dataset: {dataset.name}")
71         print(f"{'*'*50}")
72
73         # Standardize features (z-score normalization)
74         X_scaled, _ = standardize_data(dataset.X)
75
76         # Get algorithms with correct number of clusters
77         algorithms = get_main_algorithms(
78             n_clusters=dataset.n_clusters,
79             random_state=RANDOM_STATE
80         )
81
82         # Tune DBSCAN eps for this specific dataset
83         # Replace the default DBSCAN with tuned version
84         tuned_eps = tune_dbSCAN_eps(X_scaled, dataset.n_clusters)
85         for i, algo in enumerate(algorithms):
86             if isinstance(algo, DBSCANClustering):
87                 algorithms[i] = DBSCANClustering(eps=tuned_eps, min_samples=5)
88                 print(f"{'*'*50} Tuned_DBSCAN_eps: {tuned_eps:.3f}{'*'*50}")
89
90         dataset_results = []
91         dataset_metrics = {}
92
93         for algo in algorithms:
94             # Fit and predict
95             result = algo.fit_predict(X_scaled)
96
97             # Compute metrics
98             metrics = compute_all_metrics(
99                 X_scaled,
100                 result.labels,
101                 labels_true=dataset.y
102             )
103
104             dataset_results.append((result.algorithm_name, result.labels))
105             dataset_metrics[result.algorithm_name] = metrics
106
107             print(f"\n{result.algorithm_name}:")
108             print(f"    Clusters found: {result.n_clusters_found}")
109             print(metrics)
110
111         # Store for visualization
112         all_results[dataset.name] = (X_scaled, dataset.y, dataset_results)
113         all_metrics[dataset.name] = dataset_metrics
114
115         # Generate visualizations
116         print("\n[3/4] Generating visualizations...")
117
118         # Individual dataset comparison plots
119         for dataset_name, (X, y_true, results) in all_results.items():
120             fig = plot_clustering_comparison(
121                 X=X,
122                 labels_true=y_true,
123                 results=results,
124                 dataset_name=dataset_name,

```

```

125         save_path=OUTPUT_DIR / f"{dataset_name.lower().replace('_', '_')}_comparison.
126             png"
127     )
128     plt.close(fig)
129
130     # All datasets grid
131     fig = plot_all_datasets_grid(
132         datasets_results=all_results,
133         save_path=OUTPUT_DIR / "all_datasets_grid.png"
134     )
135     plt.close(fig)
136
137     # Metrics heatmaps
138     metric_names = ['silhouette', 'ari', 'nmi', 'davies_bouldin']
139     display_names = {
140         'silhouette': 'Silhouette_Score',
141         'ari': 'Adjusted_Rand_Index_(ARI)',
142         'nmi': 'Normalized_Mutual_Information_(NMI)',
143         'davies_bouldin': 'Davies-Bouldin_Index'
144     }
145
146     for metric in metric_names:
147         metrics_data = {}
148         for dataset_name, algo_metrics in all_metrics.items():
149             metrics_data[dataset_name] = {}
150             for algo_name, m in algo_metrics.items():
151                 value = getattr(m, metric, None)
152                 metrics_data[dataset_name][algo_name] = value
153
154         dataset_names = list(metrics_data.keys())
155         algorithm_names = list(next(iter(metrics_data.values()))).keys()
156
157         fig = plot_metrics_heatmap(
158             metrics_data=metrics_data,
159             dataset_names=dataset_names,
160             algorithm_names=algorithm_names,
161             metric_name=display_names[metric],
162             save_path=OUTPUT_DIR / f"heatmap_{metric}.png"
163         )
164         plt.close(fig)
165
166     # Create summary table
167     print("\n[4/4] Creating_summary_tables...")
168
169     summary_data = []
170     for dataset_name, algo_metrics in all_metrics.items():
171         for algo_name, metrics in algo_metrics.items():
172             row = {
173                 'Dataset': dataset_name,
174                 'Algorithm': algo_name,
175                 'Silhouette': f"{metrics.silhouette:.4f}" if metrics.silhouette else "N/A",
176
177                 'ARI': f"{metrics.ari:.4f}" if metrics.ari else "N/A",
178                 'NMI': f"{metrics.nmi:.4f}" if metrics.nmi else "N/A",
179                 'Davies-Bouldin': f"{metrics.davies_bouldin:.4f}" if metrics.
180                     davies_bouldin else "N/A",
181             }
182             summary_data.append(row)
183
184     df_summary = pd.DataFrame(summary_data)
185
186     # Save to CSV
187     csv_path = OUTPUT_DIR / "experiment1_results.csv"
188     df_summary.to_csv(csv_path, index=False)
189     print(f"\nResults saved to: {csv_path}")

```

```

187     # Print summary table
188     print("\n" + "=" * 70)
189     print("EXPERIMENT_1_RESULTS_SUMMARY")
190     print("=" * 70)
191     print(df_summary.to_string(index=False))
192
193     # Analysis and observations
194     print_analysis(all_metrics)
195
196
197     print(f"\n    All_visualizations_saved_to:{OUTPUT_DIR}/")
198     print("        Experiment_1_completed_successfully!")
199
200     return all_results, all_metrics
201
202
203 def print_analysis(all_metrics: Dict[str, Dict[str, ClusteringMetrics]]) :
204     """Print_analysis_and_observations_based_on_results."""
205     print("\n" + "=" * 70)
206     print("ANALYSIS_AND_OBSERVATIONS")
207     print("=" * 70)
208
209     # Find best algorithm for each dataset
210     print("\n    Best_Algorithm_by_ARI_(per_dataset):")
211     for dataset_name, algo_metrics in all_metrics.items():
212         best_algo = None
213         best_ari = -1
214         for algo_name, metrics in algo_metrics.items():
215             if metrics.ari is not None and metrics.ari > best_ari:
216                 best_ari = metrics.ari
217                 best_algo = algo_name
218         if best_algo:
219             print(f"    {dataset_name}: {best_algo} (ARI={best_ari:.4f})")
220
221     # Key observations
222     print("\n    Expected_Observations:")
223     print("    1. K-Means_performs_well_on_Blobs_but_fails_on_Moons_and_Circles")
224     print("    2. DBSCAN_and_Spectral_Clustering_succeed_on_non-convex_structures")
225     print("    3. GMM_outperforms_K-Means_on_Anisotropic_Blobs_(elliptical_clusters)")
226     print("    4. Hierarchical_(Ward)_behaves_similarly_to_K-Means_on_spherical_data")
227
228
229 # Import matplotlib at module level for visualization
230 import matplotlib.pyplot as plt
231
232
233 def main():
234     """Main_entry_point."""
235     import sys
236
237     # Parse command line arguments
238     if len(sys.argv) > 1:
239         experiment = sys.argv[1].lower()
240     else:
241         experiment = "all"
242
243     if experiment == "1" or experiment == "exp1":
244         run_experiment1()
245     elif experiment == "2" or experiment == "exp2":
246         from src.experiment2 import run_experiment2
247         run_experiment2()
248     elif experiment == "3" or experiment == "exp3":
249         from src.experiment3 import run_experiment3
250         run_experiment3()
251     elif experiment == "bonus" or experiment == "bayesian":
```

```

252     from src.bonus_bayesian import run_all_bonus_experiments
253     run_all_bonus_experiments()
254 elif experiment == "all":
255     print("\n" + "=" * 70)
256     print("Running All Experiments (including Bonus)")
257     print("=" * 70 + "\n")
258
259     run_experiment1()
260
261     print("\n" + "=" * 70 + "\n")
262
263     from src.experiment2 import run_experiment2
264     run_experiment2()
265
266     print("\n" + "=" * 70 + "\n")
267
268     from src.experiment3 import run_experiment3
269     run_experiment3()
270
271     print("\n" + "=" * 70 + "\n")
272
273     from src.bonus_bayesian import run_all_bonus_experiments
274     run_all_bonus_experiments()
275 elif experiment == "main":
276     # Run only main experiments (1, 2, 3) without bonus
277     print("\n" + "=" * 70)
278     print("Running Main Experiments (1, 2, 3)")
279     print("=" * 70 + "\n")
280
281     run_experiment1()
282
283     print("\n" + "=" * 70 + "\n")
284
285     from src.experiment2 import run_experiment2
286     run_experiment2()
287
288     print("\n" + "=" * 70 + "\n")
289
290     from src.experiment3 import run_experiment3
291     run_experiment3()
292 else:
293     print(f"Unknown experiment: {experiment}")
294     print("Usage: python main.py [1|2|3|bonus|main|all]")
295     print("...1 or exp1: Run Experiment 1 (Geometric Structure)")
296     print("...2 or exp2: Run Experiment 2 (Hyperparameter Sensitivity)")
297     print("...3 or exp3: Run Experiment 3 (Dimensionality Reduction)")
298     print("...bonus: Run Bonus Experiments (Bayesian Clustering)")
299     print("...main: Run main experiments (1, 2, 3) without bonus")
300     print("...all: Run all experiments including bonus (default)")
301     sys.exit(1)
302
303
304 if __name__ == "__main__":
305     main()

```

A.2. src/__init__.py

```

1 # NYCU Pattern Recognition HW3 - Clustering Algorithms
2 # Package initialization
3
4 from . import datasets
5 from . import clustering
6 from . import metrics
7 from . import visualization

```

```

8 | from . import experiment2
9 | from . import experiment3
10| from . import bonus_bayesian

```

A.3. src/bonus_bayesian.py

```

1 """
2 BonusExperiment: BayesianClustering
3
4 This module demonstrates how Bayesian modeling:
5 - Handles uncertainty
6 - Controls model complexity
7 - Reduces reliance on manual selection of cluster number
8
9 Three sub-experiments:
10 1. Effective Number of Clusters: EM-GMM vs Bayesian GMM
11 2. Concentration Parameter Sensitivity: _sweep
12 3. Posterior Uncertainty Visualization: Entropy-based analysis
13 """
14
15 import numpy as np
16 import pandas as pd
17 import matplotlib.pyplot as plt
18 import seaborn as sns
19 from pathlib import Path
20 from typing import Dict, List, Tuple, Optional
21 from dataclasses import dataclass
22 from sklearn.mixture import GaussianMixture, BayesianGaussianMixture
23 from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score
24 from scipy.stats import entropy
25
26 from .datasets import (
27     ClusteringDataset,
28     standardize_data,
29     get_synthetic_datasets,
30     generate_blobs,
31 )
32
33
34 @dataclass
35 class BayesianGMMResult:
36     """Results from Bayesian GMM analysis."""
37     weights: np.ndarray
38     effective_k: int
39     labels: np.ndarray
40     responsibilities: np.ndarray
41     ari: float
42     nmi: float
43     model: BayesianGaussianMixture
44
45
46 @dataclass
47 class EMGMMResult:
48     """Results from EM-GMM analysis."""
49     k: int
50     labels: np.ndarray
51     ari: float
52     nmi: float
53     bic: float
54     aic: float
55     model: GaussianMixture
56
57
58 def fit_em_gmm(

```

```

59     X: np.ndarray,
60     y_true: np.ndarray,
61     n_components: int,
62     random_state: int = 42,
63 ) -> EMGMMResult:
64     """
65     Fit_EM-GMM_with_specified_number_of_components.
66
67     Args:
68         X: Feature_matrix
69         y_true: Ground_truth_labels
70         n_components: Number_of_Gaussian_components
71         random_state: Random_seed
72
73     Returns:
74         EMGMMResult_with_fitted_model_and_metrics
75     """
76     model = GaussianMixture(
77         n_components=n_components,
78         covariance_type='full',
79         random_state=random_state,
80         n_init=5,
81     )
82     labels = model.fit_predict(X)
83
84     return EMGMMResult(
85         k=n_components,
86         labels=labels,
87         ari=adjusted_rand_score(y_true, labels),
88         nmi=normalized_mutual_info_score(y_true, labels),
89         bic=model.bic(X),
90         aic=model.aic(X),
91         model=model,
92     )
93
94
95 def fit_bayesian_gmm(
96     X: np.ndarray,
97     y_true: np.ndarray,
98     n_components: int = 20,
99     weight_concentration_prior: float = 0.01,
100    weight_concentration_prior_type: str = 'dirichlet_process',
101    weight_threshold: float = 0.01,
102    random_state: int = 42,
103 ) -> BayesianGMMResult:
104     """
105     Fit_Bayesian_GMM_with_Dirichlet_Process_prior.
106
107     Args:
108         X: Feature_matrix
109         y_true: Ground_truth_labels
110         n_components: Maximum_number_of_components_(Kmax)
111         weight_concentration_prior: Concentration_parameter_
112         weight_concentration_prior_type: 'dirichlet_process' or 'dirichlet_distribution'
113         weight_threshold: Threshold_for_counting_effective_clusters
114         random_state: Random_seed
115
116     Returns:
117         BayesianGMMResult_with_fitted_model_and_metrics
118     """
119     model = BayesianGaussianMixture(
120         n_components=n_components,
121         covariance_type='full',
122         weight_concentration_prior_type=weight_concentration_prior_type,
123         weight_concentration_prior=weight_concentration_prior,

```

```

124         random_state=random_state,
125         n_init=3,
126         max_iter=200,
127     )
128
129     labels = model.fit_predict(X)
130     responsibilities = model.predict_proba(X)
131
132     # Count effective clusters (weights above threshold)
133     effective_k = np.sum(model.weights_ > weight_threshold)
134
135     return BayesianGMMResult(
136         weights=model.weights_,
137         effective_k=effective_k,
138         labels=labels,
139         responsibilities=responsibilities,
140         ari=adjusted_rand_score(y_true, labels),
141         nmi=normalized_mutual_info_score(y_true, labels),
142         model=model,
143     )
144
145
146 def compute_point_entropy(responsibilities: np.ndarray) -> np.ndarray:
147     """
148     Compute entropy of posterior responsibilities for each data point.
149
150     High entropy indicates uncertainty in cluster assignment.
151
152     Args:
153         responsibilities: Posterior probabilities (n_samples, n_components)
154
155     Returns:
156         Entropy values for each data point
157     """
158     # Add small epsilon to avoid log(0)
159     eps = 1e-10
160     probs = np.clip(responsibilities, eps, 1.0)
161
162     # Compute entropy: -sum(p * log(p))
163     point_entropy = -np.sum(probs * np.log(probs), axis=1)
164
165     # Normalize by max possible entropy (uniform distribution)
166     n_components = responsibilities.shape[1]
167     max_entropy = np.log(n_components)
168
169     return point_entropy / max_entropy # Normalized to [0, 1]
170
171
172 # =====
173 # Bonus Experiment 1: Effective Number of Clusters
174 # =====
175
176 def run_bonus_experiment1(
177     datasets: List[ClusteringDataset],
178     k_range: range = range(2, 15),
179     kmax_bayesian: int = 20,
180     output_dir: Path = Path("outputs/bonus"),
181     random_state: int = 42,
182 ) -> Dict:
183     """
184     Bonus Experiment 1: Compare EM-GMM and Bayesian GMM on cluster number selection.
185
186     Procedure:
187     1. Fit EM-GMM with varying K
188     2. Fit Bayesian GMM with large Kmax

```

```

189     3. Compare learned mixture weights
190     4. Count effective clusters
191 """
192     print("\n" + "=" * 70)
193     print("Bonus Experiment 1: Effective Number of Clusters")
194     print("=" * 70)
195
196     output_dir.mkdir(parents=True, exist_ok=True)
197     results = {}
198
199     for dataset in datasets:
200         print(f"\nDataset: {dataset.name} (True_k={dataset.n_clusters})")
201         print("-" * 50)
202
203         X_scaled, _ = standardize_data(dataset.X)
204
205         # =====
206         # Part A: EM-GMM with varying K
207         # =====
208         print("Fitting EM-GMM with varying K...")
209         em_results = []
210         for k in k_range:
211             result = fit_em_gmm(X_scaled, dataset.y, k, random_state)
212             em_results.append(result)
213             print(f"K={k}: ARI={result.ari:.4f}, BIC={result.bic:.1f}")
214
215         # Find best K by BIC
216         best_em = min(em_results, key=lambda r: r.bic)
217         print(f"EM-GMM best K (by BIC): {best_em.k}")
218
219         # =====
220         # Part B: Bayesian GMM
221         # =====
222         print(f"Fitting Bayesian GMM (Kmax={kmax_bayesian})...")
223         bayesian_result = fit_bayesian_gmm(
224             X_scaled, dataset.y,
225             n_components=kmax_bayesian,
226             weight_concentration_prior=0.01,
227             random_state=random_state,
228         )
229         print(f"Bayesian GMM effective K: {bayesian_result.effective_k}")
230         print(f"Bayesian GMM ARI: {bayesian_result.ari:.4f}")
231
232         results[dataset.name] = {
233             'em_results': em_results,
234             'bayesian_result': bayesian_result,
235             'best_em': best_em,
236             'true_k': dataset.n_clusters,
237         }
238
239         # =====
240         # Visualizations
241         # =====
242
243         # Plot 1: EM-GMM model selection (BIC/AIC)
244         fig, axes = plt.subplots(1, 3, figsize=(15, 5))
245
246         # BIC/AIC plot
247         ax1 = axes[0]
248         k_vals = [r.k for r in em_results]
249         bic_vals = [r.bic for r in em_results]
250         aic_vals = [r.aic for r in em_results]
251         ax1.plot(k_vals, bic_vals, 'b-o', linewidth=2, label='BIC')
252         ax1.plot(k_vals, aic_vals, 'r-s', linewidth=2, label='AIC')
253         ax1.axvline(x=dataset.n_clusters, color='green', linestyle='--',

```

```

254     label=f'True_K={dataset.n_clusters}')
255     ax1.axvline(x=best_em.k, color='blue', linestyle=':',
256                 label=f'Best_BIC_K={best_em.k}')
257     ax1.set_xlabel('Number_of_Components_(K)', fontsize=11)
258     ax1.set_ylabel('Information_Criterion', fontsize=11)
259     ax1.set_title('EM-GMM_Model_Selection', fontsize=12, fontweight='bold')
260     ax1.legend()
261     ax1.grid(True, alpha=0.3)
262
263 # ARI comparison
264 ax2 = axes[1]
265 ari_vals = [r.ari for r in em_results]
266 ax2.plot(k_vals, ari_vals, 'g-o', linewidth=2, label='EM-GMM_ARI')
267 ax2.axhline(y=bayesian_result.ari, color='purple', linestyle='--',
268               linewidth=2, label=f'Bayesian_GMM_ARI')
269 ax2.axvline(x=dataset.n_clusters, color='green', linestyle='--', alpha=0.5)
270 ax2.set_xlabel('Number_of_Components_(K)', fontsize=11)
271 ax2.set_ylabel('Adjusted_Rand_Index', fontsize=11)
272 ax2.set_title('Clustering_Quality_(ARI)', fontsize=12, fontweight='bold')
273 ax2.legend()
274 ax2.grid(True, alpha=0.3)
275
276 # Bayesian GMM weights
277 ax3 = axes[2]
278 weights = bayesian_result.weights
279 sorted_weights = np.sort(weights) [::-1]
280 colors = ['steelblue' if w > 0.01 else 'lightgray' for w in sorted_weights]
281 ax3.bar(range(1, len(weights) + 1), sorted_weights, color=colors, alpha=0.8)
282 ax3.axhline(y=0.01, color='red', linestyle='--', label='Threshold_(0.01)')
283 ax3.set_xlabel('Component_(sorted)', fontsize=11)
284 ax3.set_ylabel('Mixture_Weight', fontsize=11)
285 ax3.set_title(f'Bayesian_GMM_Weights\n(Effective_K={bayesian_result.effective_k})',
286               fontsize=12, fontweight='bold')
287 ax3.legend()
288 ax3.grid(True, alpha=0.3, axis='y')
289
290 plt.suptitle(f'EM-GMM_vs_Bayesian_GMM:{dataset.name}',
291               fontsize=14, fontweight='bold')
292 plt.tight_layout()
293
294 save_path = output_dir / f"bonus1_effective_k_{dataset.name.lower().replace(' ', '_')}.png"
295 fig.savefig(save_path, dpi=150, bbox_inches='tight')
296 print(f"__Saved:{save_path}")
297 plt.close(fig)
298
299 # Create summary table
300 summary_data = []
301 for ds_name, data in results.items():
302     row = {
303         'Dataset': ds_name,
304         'True_K': data['true_k'],
305         'EM-GMM_Best_K_(BIC)': data['best_em'].k,
306         'EM-GMM_ARI': f'{data["best_em"].ari:.4f}',
307         'Bayesian_GMM_Effective_K': data['bayesian_result'].effective_k,
308         'Bayesian_GMM_ARI': f'{data["bayesian_result"].ari:.4f}',
309     }
310     summary_data.append(row)
311
312 df_summary = pd.DataFrame(summary_data)
313 df_summary.to_csv(output_dir / "bonus1_summary.csv", index=False)
314
315 print("\n" + "=" * 70)
316 print("BONUS_EXPERIMENT_1_SUMMARY")

```

```

317     print("=" * 70)
318     print(df_summary.to_string(index=False))
319
320     return results
321
322
323 # =====
324 # Bonus Experiment 2: Concentration Parameter Sensitivity
325 # =====
326
327 def run_bonus_experiment2(
328     datasets: List[ClusteringDataset],
329     alpha_range: np.ndarray = None,
330     kmax: int = 20,
331     output_dir: Path = Path("outputs/bonus"),
332     random_state: int = 42,
333 ) -> Dict:
334     """
335     BonusExperiment_2: Analyze_effect_of_concentration_parameter .
336
337     Procedure:
338     1. Use Dirichlet Process-style Bayesian GMM
339     2. Sweep concentration parameter
340     3. Record effective number of clusters and ARI
341     4. Plot Effective K vs and ARI vs
342     """
343     print("\n" + "=" * 70)
344     print("BonusExperiment_2: Concentration_Parameter_Sensitivity")
345     print("=" * 70)
346
347     output_dir.mkdir(parents=True, exist_ok=True)
348
349     if alpha_range is None:
350         # Log-spaced alpha values from very small to large
351         alpha_range = np.logspace(-3, 2, 20) # 0.001 to 100
352
353     results = {}
354
355     for dataset in datasets:
356         print(f"\nDataset: {dataset.name} (True_k={dataset.n_clusters})")
357         print("-" * 50)
358
359         X_scaled, _ = standardize_data(dataset.X)
360
361         effective_k_list = []
362         ari_list = []
363         nmi_list = []
364
365         for alpha in alpha_range:
366             result = fit_bayesian_gmm(
367                 X_scaled, dataset.y,
368                 n_components=kmax,
369                 weight_concentration_prior=alpha,
370                 random_state=random_state,
371             )
372             effective_k_list.append(result.effective_k)
373             ari_list.append(result.ari)
374             nmi_list.append(result.nmi)
375
376             print(f"  ={alpha:.4f}: Effective_K={result.effective_k}, ARI={result.ari:.4
377 f}")
378
379     results[dataset.name] = {
380         'alpha_range': alpha_range,
381         'effective_k': np.array(effective_k_list),

```

```

381         'ari': np.array(ari_list),
382         'nmi': np.array(nmi_list),
383         'true_k': dataset.n_clusters,
384     }
385
386     # Visualization
387     fig, axes = plt.subplots(1, 2, figsize=(14, 5))
388
389     # Effective K vs
390     ax1 = axes[0]
391     ax1.semilogx(alpha_range, effective_k_list, 'b-o', linewidth=2, markersize=6)
392     ax1.axhline(y=dataset.n_clusters, color='red', linestyle='--',
393                  linewidth=2, label=f'True_K={dataset.n_clusters}')
394     ax1.set_xlabel('Concentration_Parameter_( )', fontsize=12)
395     ax1.set_ylabel('Effective_Number_of_Clusters', fontsize=12)
396     ax1.set_title('Effective_K_vs_ ', fontsize=12, fontweight='bold')
397     ax1.legend()
398     ax1.grid(True, alpha=0.3)
399
400     # ARI vs
401     ax2 = axes[1]
402     ax2.semilogx(alpha_range, ari_list, 'g-o', linewidth=2, markersize=6, label='ARI')
403     ax2.semilogx(alpha_range, nmi_list, 'm-s', linewidth=2, markersize=6, label='NMI')
404     ax2.set_xlabel('Concentration_Parameter_( )', fontsize=12)
405     ax2.set_ylabel('Score', fontsize=12)
406     ax2.set_title('Clustering_Quality_vs_ ', fontsize=12, fontweight='bold')
407     ax2.legend()
408     ax2.grid(True, alpha=0.3)
409     ax2.set_ylim([0, 1.05])
410
411     # Find best
412     best_idx = np.argmax(ari_list)
413     best_alpha = alpha_range[best_idx]
414     ax1.axvline(x=best_alpha, color='green', linestyle=':', alpha=0.7,
415                  label=f'Best_ ={best_alpha:.3f}')
416     ax2.axvline(x=best_alpha, color='green', linestyle=':', alpha=0.7)
417     ax1.legend()
418
419     plt.suptitle(f'Concentration_Parameter_Sensitivity:{dataset.name}', fontweight='bold', fontsize=14)
420     plt.tight_layout()
421
422     save_path = output_dir / f"bonus2_alpha_sensitivity_{dataset.name.lower().replace(' ', '_')}.png"
423     fig.savefig(save_path, dpi=150, bbox_inches='tight')
424     print(f"Saved:{save_path}")
425     plt.close(fig)
426
427     # Summary analysis
428     print("\n" + "=" * 70)
429     print("ANALYSIS: Concentration_Parameter_Effects")
430     print("=" * 70)
431     print("""
432     Key_Observations:
433
434 1. Small_(<0.1):
435     Encourages_fewer_clusters
436     Prior_concentrates_probability_on_few_components
437     May_underfit_if_true_K_is_large
438
439 2. Large_(>10):
440     Encourages_more_clusters
441     More_uniform_prior_over_components
442     Approaches_EM-GMM_behavior
443

```

```

445     3. Optimal :
446         - Dataset-dependent
447         - Typically in range [0.01, 1] for moderate cluster numbers
448         - Different from geometry-based parameters (e.g., DBSCAN eps)
449 """
450
451     return results
452
453
454 # =====
455 # Bonus Experiment 3: Posterior Uncertainty Visualization
456 # =====
457
458 def run_bonus_experiment3(
459     datasets: List[ClusteringDataset] = None,
460     output_dir: Path = Path("outputs/bonus"),
461     random_state: int = 42,
462 ) -> Dict:
463     """
464     Bonus Experiment 3: Visualize posterior uncertainty via entropy.
465
466     Procedure:
467     1. Compute posterior responsibilities
468     2. Calculate entropy per data point
469     3. Visualize entropy on 2D toy datasets
470
471     High entropy indicates ambiguous/boundary points.
472 """
473     print("\n" + "=" * 70)
474     print("Bonus Experiment 3: Posterior Uncertainty Visualization")
475     print("=" * 70)
476
477     output_dir.mkdir(parents=True, exist_ok=True)
478
479     # Use 2D synthetic datasets for visualization
480     if datasets is None:
481         datasets = get_synthetic_datasets(random_state=random_state)
482
483     results = {}
484
485     for dataset in datasets:
486         if dataset.X.shape[1] != 2:
487             print(f"Skipping {dataset.name} (not 2D)")
488             continue
489
490         print(f"\nDataset: {dataset.name}")
491         print("-" * 50)
492
493         X_scaled, _ = standardize_data(dataset.X)
494
495         # Fit Bayesian GMM
496         bayesian_result = fit_bayesian_gmm(
497             X_scaled, dataset.y,
498             n_components=15,
499             weight_concentration_prior=0.1,
500             random_state=random_state,
501         )
502
503         # Compute point-wise entropy
504         point_entropy = compute_point_entropy(bayesian_result.responsibilities)
505
506         results[dataset.name] = {
507             'X': X_scaled,
508             'y_true': dataset.y,
509             'labels': bayesian_result.labels,

```

```

510     'entropy': point_entropy,
511     'responsibilities': bayesian_result.responsibilities,
512     'effective_k': bayesian_result.effective_k,
513 }
514
515 print(f"Effective_K:{bayesian_result.effective_k}")
516 print(f"Mean_entropy:{np.mean(point_entropy):.4f}")
517 print(f"High_uncertainty_points:(entropy>0.5):{np.sum(point_entropy>0.5)}")
518
519 # Visualization
520 fig, axes = plt.subplots(1, 3, figsize=(15, 5))
521
522 # Ground truth
523 ax1 = axes[0]
524 scatter1 = ax1.scatter(X_scaled[:, 0], X_scaled[:, 1], c=dataset.y,
525                         cmap='tab10', s=30, alpha=0.7)
526 ax1.set_title(f'Ground_Truth\n{n_clusters} clusters',
527                 fontsize=12, fontweight='bold')
528 ax1.set_xlabel('Feature_1')
529 ax1.set_ylabel('Feature_2')
530
531 # Bayesian GMM clustering
532 ax2 = axes[1]
533 scatter2 = ax2.scatter(X_scaled[:, 0], X_scaled[:, 1], c=bayesian_result.labels,
534                         cmap='tab10', s=30, alpha=0.7)
535 ax2.set_title(f'Bayesian_GMM_Clustering\n(Effective_K={bayesian_result.effective_k})',
536                 fontsize=12, fontweight='bold')
537 ax2.set_xlabel('Feature_1')
538 ax2.set_ylabel('Feature_2')
539
540 # Entropy visualization
541 ax3 = axes[2]
542 scatter3 = ax3.scatter(X_scaled[:, 0], X_scaled[:, 1], c=point_entropy,
543                         cmap='YlOrRd', s=30, alpha=0.8, vmin=0, vmax=1)
544 cbar = plt.colorbar(scatter3, ax=ax3)
545 cbar.set_label('Normalized_Entropy', fontsize=10)
546 ax3.set_title('Posterior_Uncertainty\n(High_entropy=_ambiguous)',
547                 fontsize=12, fontweight='bold')
548 ax3.set_xlabel('Feature_1')
549 ax3.set_ylabel('Feature_2')
550
551 plt.suptitle(f'Posterior_Uncertainty_Analysis:{dataset.name}',
552                 fontsize=14, fontweight='bold')
553 plt.tight_layout()
554
555 save_path = output_dir / f"bonus3_uncertainty_{dataset.name.lower().replace(' ', '_')}.png"
556 fig.savefig(save_path, dpi=150, bbox_inches='tight')
557 print(f"Saved:{save_path}")
558 plt.close(fig)
559
560 # Create combined visualization
561 n_datasets = len([d for d in datasets if d.X.shape[1] == 2])
562 if n_datasets > 0:
563     fig, axes = plt.subplots(n_datasets, 3, figsize=(15, 4 * n_datasets))
564     if n_datasets == 1:
565         axes = axes.reshape(1, -1)
566
567     row_idx = 0
568     for dataset in datasets:
569         if dataset.X.shape[1] != 2:
570             continue
571
572         data = results[dataset.name]

```

```

573     X = data['X']
574
575     # Ground truth
576     axes[row_idx, 0].scatter(X[:, 0], X[:, 1], c=data['y_true'],
577                               cmap='tab10', s=20, alpha=0.7)
578     axes[row_idx, 0].set_title(f'{dataset.name}: _Ground_Truth')
579     axes[row_idx, 0].set_xticks([])
580     axes[row_idx, 0].set_yticks([])
581
582     # Clustering
583     axes[row_idx, 1].scatter(X[:, 0], X[:, 1], c=data['labels'],
584                               cmap='tab10', s=20, alpha=0.7)
585     axes[row_idx, 1].set_title(f'Bayesian_GMM_(K={data["effective_k"]})')
586     axes[row_idx, 1].set_xticks([])
587     axes[row_idx, 1].set_yticks([])
588
589     # Entropy
590     sc = axes[row_idx, 2].scatter(X[:, 0], X[:, 1], c=data['entropy'],
591                               cmap='YlOrRd', s=20, alpha=0.8, vmin=0, vmax=1)
592     axes[row_idx, 2].set_title('Uncertainty_(Entropy)')
593     axes[row_idx, 2].set_xticks([])
594     axes[row_idx, 2].set_yticks([])
595
596     row_idx += 1
597
598     # Add colorbar
599     fig.subplots_adjust(right=0.92)
600     cbar_ax = fig.add_axes([0.94, 0.15, 0.02, 0.7])
601     cbar = fig.colorbar(sc, cax=cbar_ax)
602     cbar.set_label('Normalized_Entropy', fontsize=10)
603
604     plt.suptitle('Posterior_Uncertainty_Across_Datasets',
605                  fontsize=14, fontweight='bold', y=1.02)
606
607     save_path = output_dir / "bonus3_uncertainty_all_datasets.png"
608     fig.savefig(save_path, dpi=150, bbox_inches='tight')
609     print(f"\nSaved_combined_visualization:{save_path}")
610     plt.close(fig)
611
612     print("\n" + "=" * 70)
613     print("ANALYSIS:_Posterior_Uncertainty")
614     print("==" * 70)
615     print("""
616     ↳ Interpretation:
617
618     1._Low_Uncertainty_(Yellow):
619     ↳ ↳ High_confidence_in_cluster_assignment
620     ↳ ↳ Points_clearly_belong_to_one_cluster
621
622     2._High_Uncertainty_(Red):
623     ↳ ↳ Uncertain_cluster_assignment
624     ↳ ↳ Typically_boundary_points_or_overlapping_regions
625     ↳ ↳ Model_acknowledges_ambiguity
626
627     3._Bayesian_Advantage:
628     ↳ ↳ Probabilistic_assignments_capture_uncertainty
629     ↳ ↳ Hard_clustering_(K-Means)_cannot_express_this
630     ↳ ↳ Useful_for_downstream_decision-making
631     """
632
633     return results
634
635
636     # =====
637     # Main Bonus Experiment Runner

```

```

638 # =====
639
640 def run_all_bonus_experiments(
641     output_dir: Path = Path("outputs/bonus"),
642     random_state: int = 42,
643 ) -> Dict:
644     """
645     Run all bonus experiments for Bayesian clustering.
646
647     Args:
648         output_dir: Directory to save outputs
649         random_state: Random seed
650
651     Returns:
652         Dictionary containing all results
653     """
654     print("\n" + "=" * 70)
655     print("BONUS: Bayesian Gaussian Mixture Model Experiments")
656     print("=" * 70)
657
658     output_dir.mkdir(parents=True, exist_ok=True)
659
660     # Get synthetic datasets
661     datasets = get_synthetic_datasets(random_state=random_state)
662
663     results = {}
664
665     # Bonus Experiment 1: Effective Number of Clusters
666     results['experiment1'] = run_bonus_experiment1(
667         datasets=datasets,
668         output_dir=output_dir,
669         random_state=random_state,
670     )
671
672     # Bonus Experiment 2: Concentration Parameter Sensitivity
673     results['experiment2'] = run_bonus_experiment2(
674         datasets=datasets,
675         output_dir=output_dir,
676         random_state=random_state,
677     )
678
679     # Bonus Experiment 3: Posterior Uncertainty Visualization
680     results['experiment3'] = run_bonus_experiment3(
681         datasets=datasets,
682         output_dir=output_dir,
683         random_state=random_state,
684     )
685
686     # Final summary
687     print("\n" + "=" * 70)
688     print("BONUS EXPERIMENTS COMPLETED")
689     print("=" * 70)
690     print(f"""
691     All bonus experiment results saved to: {output_dir}/
692
693     Generated Files:
694     - bonus1_effective_k_*.png: EM-GMM_vs_Bayesian_GMM_comparison
695     - bonus1_summary.csv: Effective_K_comparison_table
696     - bonus2_alpha_sensitivity_*.png: Concentration_parameter_analysis
697     - bonus3_uncertainty_*.png: Posterior_entropy_visualization
698     - bonus3_uncertainty_all_datasets.png: Combined_uncertainty_view
699
700     Key Takeaways:
701
702 1. Bayesian GMM automatically infers cluster number:

```

```

703     """- No need to manually specify K
704     """- Dirichlet Process prior controls complexity
705
706     2. Concentration parameter controls model complexity:
707     """- Small  ↗ fewer clusters
708     """- Large ↗ more clusters
709     """- Optimal is dataset-dependent
710
711     3. Posterior uncertainty provides valuable information:
712     """- Identifies ambiguous/boundary points
713     """- Enables principled decision-making under uncertainty
714     """- Not available in hard clustering methods
715     """
716
717     return results

```

A.4. src/clustering.py

```

1 """
2 Clustering algorithm wrappers for unified interface.
3 """
4
5 import numpy as np
6 from sklearn.cluster import (
7     KMeans,
8     AgglomerativeClustering,
9     DBSCAN,
10    SpectralClustering,
11)
12 from sklearn.mixture import GaussianMixture, BayesianGaussianMixture
13 from dataclasses import dataclass
14 from typing import Optional, Dict, Any, Callable
15 from abc import ABC, abstractmethod
16
17
18 @dataclass
19 class ClusteringResult:
20     """Container for clustering results."""
21     algorithm_name: str
22     labels: np.ndarray
23     n_clusters_found: int
24     params: Dict[str, Any]
25     model: Any # The fitted model object
26
27
28 class ClusteringAlgorithm(ABC):
29     """Abstract base class for clustering algorithms."""
30
31     @property
32     @abstractmethod
33     def name(self) -> str:
34         """Algorithm name for display."""
35         pass
36
37     @abstractmethod
38     def fit_predict(self, X: np.ndarray) -> ClusteringResult:
39         """Fit the model and return clustering results."""
40         pass
41
42
43 class KMeansClustering(ClusteringAlgorithm):
44     """K-Means / K-Means++ clustering."""
45
46     def __init__(


```

```

47     self,
48     n_clusters: int,
49     init: str = "k-means++",
50     n_init: int = 10,
51     random_state: int = 42,
52 ) :
53     self.n_clusters = n_clusters
54     self.init = init
55     self.n_init = n_init
56     self.random_state = random_state
57
58 @property
59 def name(self) -> str:
60     return f"K-Means_{n_clusters})"
61
62 def fit_predict(self, X: np.ndarray) -> ClusteringResult:
63     model = KMeans(
64         n_clusters=self.n_clusters,
65         init=self.init,
66         n_init=self.n_init,
67         random_state=self.random_state,
68     )
69     labels = model.fit_predict(X)
70     return ClusteringResult(
71         algorithm_name=self.name,
72         labels=labels,
73         n_clusters_found=self.n_clusters,
74         params={
75             "n_clusters": self.n_clusters,
76             "init": self.init,
77             "n_init": self.n_init,
78         },
79         model=model,
80     )
81
82
83 class HierarchicalClustering(ClusteringAlgorithm):
84     """Agglomerative_Hierarchical_Clustering."""
85
86     def __init__(
87         self,
88         n_clusters: int,
89         linkage: str = "ward",
90     ) :
91         self.n_clusters = n_clusters
92         self.linkage = linkage
93
94     @property
95     def name(self) -> str:
96         return f"Hierarchical_{self.linkage})"
97
98     def fit_predict(self, X: np.ndarray) -> ClusteringResult:
99         model = AgglomerativeClustering(
100             n_clusters=self.n_clusters,
101             linkage=self.linkage,
102         )
103         labels = model.fit_predict(X)
104         return ClusteringResult(
105             algorithm_name=self.name,
106             labels=labels,
107             n_clusters_found=self.n_clusters,
108             params={
109                 "n_clusters": self.n_clusters,
110                 "linkage": self.linkage,
111             },
112

```

```

112         model=model,
113     )
114
115
116 class DBSCANClustering(ClusteringAlgorithm):
117     """DBSCAN_density-based_clustering."""
118
119     def __init__(
120         self,
121         eps: float = 0.5,
122         min_samples: int = 5,
123     ):
124         self.eps = eps
125         self.min_samples = min_samples
126
127     @property
128     def name(self) -> str:
129         return f"DBSCAN_(eps={self.eps})"
130
131     def fit_predict(self, X: np.ndarray) -> ClusteringResult:
132         model = DBSCAN(
133             eps=self.eps,
134             min_samples=self.min_samples,
135         )
136         labels = model.fit_predict(X)
137         # Count clusters (excluding noise label -1)
138         n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
139         return ClusteringResult(
140             algorithm_name=self.name,
141             labels=labels,
142             n_clusters_found=n_clusters,
143             params={
144                 "eps": self.eps,
145                 "min_samples": self.min_samples,
146             },
147             model=model,
148         )
149
150
151 class GMMClustering(ClusteringAlgorithm):
152     """Gaussian_Mixture_Model_clustering_(EM_algorithm)."""
153
154     def __init__(
155         self,
156         n_components: int,
157         covariance_type: str = "full",
158         random_state: int = 42,
159     ):
160         self.n_components = n_components
161         self.covariance_type = covariance_type
162         self.random_state = random_state
163
164     @property
165     def name(self) -> str:
166         return f"GMM_(k={self.n_components})"
167
168     def fit_predict(self, X: np.ndarray) -> ClusteringResult:
169         model = GaussianMixture(
170             n_components=self.n_components,
171             covariance_type=self.covariance_type,
172             random_state=self.random_state,
173         )
174         labels = model.fit_predict(X)
175         return ClusteringResult(
176             algorithm_name=self.name,

```

```

177         labels=labels,
178         n_clusters_found=self.n_components,
179         params={
180             "n_components": self.n_components,
181             "covariance_type": self.covariance_type,
182         },
183         model=model,
184     )
185
186
187 class SpectralClusteringWrapper(ClusteringAlgorithm):
188     """Spectral_Clustering_with_RBF_affinity."""
189
190     def __init__(
191         self,
192         n_clusters: int,
193         affinity: str = "rbf",
194         gamma: float = 1.0,
195         random_state: int = 42,
196     ):
197         self.n_clusters = n_clusters
198         self.affinity = affinity
199         self.gamma = gamma
200         self.random_state = random_state
201
202     @property
203     def name(self) -> str:
204         return f"Spectral_(k={self.n_clusters})"
205
206     def fit_predict(self, X: np.ndarray) -> ClusteringResult:
207         model = SpectralClustering(
208             n_clusters=self.n_clusters,
209             affinity=self.affinity,
210             gamma=self.gamma,
211             random_state=self.random_state,
212             assign_labels="kmeans",
213         )
214         labels = model.fit_predict(X)
215         return ClusteringResult(
216             algorithm_name=self.name,
217             labels=labels,
218             n_clusters_found=self.n_clusters,
219             params={
220                 "n_clusters": self.n_clusters,
221                 "affinity": self.affinity,
222                 "gamma": self.gamma,
223             },
224             model=model,
225         )
226
227
228 class BayesianGMMClustering(ClusteringAlgorithm):
229     """Bayesian_Gaussian_Mixture_Model_(Variational_Inference)."""
230
231     def __init__(
232         self,
233         n_components: int = 10,
234         weight_concentration_prior_type: str = "dirichlet_process",
235         weight_concentration_prior: float = 0.01,
236         covariance_type: str = "full",
237         random_state: int = 42,
238     ):
239         self.n_components = n_components
240         self.weight_concentration_prior_type = weight_concentration_prior_type
241         self.weight_concentration_prior = weight_concentration_prior

```

```

242         self.covariance_type = covariance_type
243         self.random_state = random_state
244
245     @property
246     def name(self) -> str:
247         return f"Bayesian_GMM_(K_max={self.n_components})"
248
249     def fit_predict(self, X: np.ndarray) -> ClusteringResult:
250         model = BayesianGaussianMixture(
251             n_components=self.n_components,
252             weight_concentration_prior_type=self.weight_concentration_prior_type,
253             weight_concentration_prior=self.weight_concentration_prior,
254             covariance_type=self.covariance_type,
255             random_state=self.random_state,
256         )
257         labels = model.fit_predict(X)
258         # Count effective clusters (components with significant weight)
259         effective_clusters = np.sum(model.weights_ > 0.01)
260         return ClusteringResult(
261             algorithm_name=self.name,
262             labels=labels,
263             n_clusters_found=effective_clusters,
264             params={
265                 "n_components": self.n_components,
266                 "weight_concentration_prior_type": self.weight_concentration_prior_type,
267                 "weight_concentration_prior": self.weight_concentration_prior,
268                 "weights": model.weights_,
269             },
270             model=model,
271         )
272
273
274     def get_main_algorithms(n_clusters: int, random_state: int = 42) -> list[ClusteringAlgorithm]:
275         """
276         Get the five main clustering algorithms for Experiment 1.
277
278         Args:
279             n_clusters: Target number of clusters (from ground truth)
280             random_state: Random seed for reproducibility
281
282         Returns:
283             List of configured clustering algorithms
284         """
285         return [
286             KMeansClustering(n_clusters=n_clusters, random_state=random_state),
287             HierarchicalClustering(n_clusters=n_clusters, linkage="ward"),
288             DBSCANClustering(eps=0.5, min_samples=5),
289             GMMClustering(n_components=n_clusters, random_state=random_state),
290             SpectralClusteringWrapper(n_clusters=n_clusters, random_state=random_state),
291         ]
292
293
294     def get_hierarchical_variants(n_clusters: int) -> list[ClusteringAlgorithm]:
295         """Get all hierarchical clustering linkage variants."""
296         return [
297             HierarchicalClustering(n_clusters=n_clusters, linkage="single"),
298             HierarchicalClustering(n_clusters=n_clusters, linkage="complete"),
299             HierarchicalClustering(n_clusters=n_clusters, linkage="ward"),
300         ]
301
302
303     def tune_dbscan_eps(X: np.ndarray, target_n_clusters: int) -> float:
304         """
305         Heuristic to tune DBSCAN eps parameter based on data.

```

```

306
307     """Uses k-distance graph approach: compute distance to k-th nearest neighbor
308     and use a percentile of those distances.
309     """
310     from sklearn.neighbors import NearestNeighbors
311
312     k = min(5, X.shape[0] - 1)
313     nn = NearestNeighbors(n_neighbors=k)
314     nn.fit(X)
315     distances, _ = nn.kneighbors(X)
316     k_distances = distances[:, -1]
317
318     # Use the knee point (elbow) in the k-distance graph
319     # Approximate by taking a percentile
320     eps = np.percentile(k_distances, 90)
321     return eps

```

A.5. src/datasets.py

```

1 """
2 Synthetic and real-world dataset generation for clustering experiments.
3 """
4
5 import numpy as np
6 from sklearn.datasets import (
7     make_blobs,
8     make_moons,
9     make_circles,
10    load_iris,
11    load_wine,
12    load_digits,
13 )
14 from sklearn.preprocessing import StandardScaler
15 from dataclasses import dataclass
16 from typing import Optional, Tuple
17
18
19 @dataclass
20 class ClusteringDataset:
21     """Container for clustering dataset with metadata."""
22     name: str
23     X: np.ndarray
24     y: np.ndarray
25     n_clusters: int
26     description: str
27
28
29 def generate_blobs(
30     n_samples: int = 500,
31     n_features: int = 2,
32     centers: int = 3,
33     cluster_std: float = 1.0,
34     random_state: int = 42,
35 ) -> ClusteringDataset:
36     """Generate isotropic Gaussian blobs (spherical clusters).
37
38     This is a baseline sanity check dataset where K-Means should perform well.
39     """
40     X, y = make_blobs(
41         n_samples=n_samples,
42         n_features=n_features,
43         centers=centers,
44         cluster_std=cluster_std,

```

```

46         random_state=random_state,
47     )
48     return ClusteringDataset(
49         name="Blobs",
50         X=X,
51         y=Y,
52         n_clusters=centers,
53         description="Spherical Gaussian clusters - baseline for centroid-based methods",
54     )
55
56
57 def generate_moons(
58     n_samples: int = 500,
59     noise: float = 0.1,
60     random_state: int = 42,
61 ) -> ClusteringDataset:
62     """
63     Generate two interleaving half circles (moons).
64
65     This dataset has non-convex structure that challenges K-Means.
66     """
67     X, y = make_moons(
68         n_samples=n_samples,
69         noise=noise,
70         random_state=random_state,
71     )
72     return ClusteringDataset(
73         name="Two_Moons",
74         X=X,
75         y=y,
76         n_clusters=2,
77         description="Non-convex interleaving crescents - challenges centroid methods",
78     )
79
80
81 def generate_circles(
82     n_samples: int = 500,
83     noise: float = 0.05,
84     factor: float = 0.5,
85     random_state: int = 42,
86 ) -> ClusteringDataset:
87     """
88     Generate concentric circles (nested cluster structure).
89
90     This dataset has nested structure that requires density or graph-based methods.
91     """
92     X, y = make_circles(
93         n_samples=n_samples,
94         noise=noise,
95         factor=factor,
96         random_state=random_state,
97     )
98     return ClusteringDataset(
99         name="Concentric_Circles",
100        X=X,
101        y=y,
102        n_clusters=2,
103        description="Nested circular clusters - requires non-linear methods",
104    )
105
106
107 def generate_anisotropic(
108     n_samples: int = 500,
109     centers: int = 3,
110     random_state: int = 42,

```

```

111 ) -> ClusteringDataset:
112     """
113     Generate_anisotropic_(elliptical)_Gaussian_blobs.
114
115     Created_by_applying_a_linear_transformation_to_standard_blobs.
116     This_challenges_K-Means_which Assumes_spherical_clusters.
117     """
118     X, y = make_blobs(
119         n_samples=n_samples,
120         n_features=2,
121         centers=centers,
122         cluster_std=1.0,
123         random_state=random_state,
124     )
125     # Apply linear transformation to create elongated clusters
126     transformation = np.array([[0.6, -0.6], [-0.4, 0.8]])
127     X = X @ transformation
128
129     return ClusteringDataset(
130         name="Anisotropic_Blobs",
131         X=X,
132         y=y,
133         n_clusters=centers,
134         description="Elliptical_clusters_via_linear_transformation_GMM_advantage",
135     )
136
137
138 def load_iris_dataset() -> ClusteringDataset:
139     """Load_Iris_dataset_for_clustering_evaluation."""
140     data = load_iris()
141     return ClusteringDataset(
142         name="Iris",
143         X=data.data,
144         y=data.target,
145         n_clusters=3,
146         description="Classic_4-feature_flower_dataset_(150_samples,_3_classes)",
147     )
148
149
150 def load_wine_dataset() -> ClusteringDataset:
151     """Load_Wine_dataset_for_clustering_evaluation."""
152     data = load_wine()
153     return ClusteringDataset(
154         name="Wine",
155         X=data.data,
156         y=data.target,
157         n_clusters=3,
158         description="Wine_recognition_dataset_(178_samples,_13_features,_3_classes)",
159     )
160
161
162 def load_digits_dataset() -> ClusteringDataset:
163     """Load_Digits_dataset_for_high-dimensional_clustering_evaluation."""
164     data = load_digits()
165     return ClusteringDataset(
166         name="Digits",
167         X=data.data,
168         y=data.target,
169         n_clusters=10,
170         description="8x8_handwritten_digits_(1797_samples,_64_features,_10_classes)",
171     )
172
173
174 def get_synthetic_datasets(random_state: int = 42) -> list[ClusteringDataset]:
175     """Generate_all_synthetic_datasets_for_Experiment_1."""

```

```

176     return [
177         generate_blobs(random_state=random_state),
178         generate_moons(random_state=random_state),
179         generate_circles(random_state=random_state),
180         generate_anisotropic(random_state=random_state),
181     ]
182
183
184 def get_real_datasets() -> list[ClusteringDataset]:
185     """Load all real-world datasets."""
186     return [
187         load_iris_dataset(),
188         load_wine_dataset(),
189         load_digits_dataset(),
190     ]
191
192
193 def standardize_data(X: np.ndarray) -> Tuple[np.ndarray, StandardScaler]:
194     """
195     Apply z-score normalization to features.
196
197     Returns:
198     Tuple of (standardized_data, fitted_scaler)
199     """
200     scaler = StandardScaler()
201     X_scaled = scaler.fit_transform(X)
202     return X_scaled, scaler

```

A.6. src/experiment2.py

```

1 """
2 Experiment_2: Hyperparameter_Sensitivity_and_Metric_Conflict
3
4 This module analyzes how sensitive clustering results are to hyperparameter choices
5 and reveals conflicts between internal and external evaluation metrics.
6
7 Two sub-experiments:
8 2.1 K-Means: Choice of Number of Clusters (k_sweep from 2 to 10)
9 2.2 DBSCAN: Density Parameters (eps and min_samples sweep)
10 """
11
12 import numpy as np
13 import pandas as pd
14 import matplotlib.pyplot as plt
15 import seaborn as sns
16 from pathlib import Path
17 from typing import Dict, List, Tuple, Optional
18 from dataclasses import dataclass
19 from sklearn.cluster import KMeans, DBSCAN
20 from sklearn.metrics import (
21     silhouette_score,
22     calinski_harabasz_score,
23     davies_bouldin_score,
24     adjusted_rand_score,
25     normalized_mutual_info_score,
26 )
27
28 from .datasets import ClusteringDataset, standardize_data, get_synthetic_datasets
29
30
31 @dataclass
32 class KMeansAnalysisResult:
33     """Results from K-Means k-sweep analysis."""
34     k_values: np.ndarray

```

```

35     sse_values: np.ndarray # Sum of Squared Errors (inertia)
36     silhouette_values: np.ndarray
37     calinski_harabasz_values: np.ndarray
38     davies_bouldin_values: np.ndarray
39     ari_values: Optional[np.ndarray] = None
40     nmi_values: Optional[np.ndarray] = None
41
42
43 @dataclass
44 class DBSCANAnalysisResult:
45     """Results from DBSCAN parameter sweep analysis."""
46     eps_values: np.ndarray
47     min_samples_values: np.ndarray
48     ari_matrix: np.ndarray # Shape: (len(eps), len(min_samples))
49     nmi_matrix: np.ndarray
50     n_clusters_matrix: np.ndarray
51     noise_ratio_matrix: np.ndarray
52
53
54 def analyze_kmeans_k_selection(
55     X: np.ndarray,
56     y_true: Optional[np.ndarray] = None,
57     k_range: range = range(2, 11),
58     random_state: int = 42,
59 ) -> KMeansAnalysisResult:
60     """
61     Analyze K-Means performance across different values of k.
62
63     Args:
64         X: Feature matrix
65         y_true: Ground truth labels (optional, for external metrics)
66         k_range: Range of k values to test
67         random_state: Random seed for reproducibility
68
69     Returns:
70         KMeansAnalysisResult with all computed metrics
71     """
72     k_values = np.array(list(k_range))
73     n_k = len(k_values)
74
75     sse_values = np.zeros(n_k)
76     silhouette_values = np.zeros(n_k)
77     calinski_harabasz_values = np.zeros(n_k)
78     davies_bouldin_values = np.zeros(n_k)
79     ari_values = np.zeros(n_k) if y_true is not None else None
80     nmi_values = np.zeros(n_k) if y_true is not None else None
81
82     for i, k in enumerate(k_values):
83         kmeans = KMeans(n_clusters=k, random_state=random_state, n_init=10)
84         labels = kmeans.fit_predict(X)
85
86         # SSE (Sum of Squared Errors) - inertia
87         sse_values[i] = kmeans.inertia_
88
89         # Internal metrics
90         silhouette_values[i] = silhouette_score(X, labels)
91         calinski_harabasz_values[i] = calinski_harabasz_score(X, labels)
92         davies_bouldin_values[i] = davies_bouldin_score(X, labels)
93
94         # External metrics (if ground truth available)
95         if y_true is not None:
96             ari_values[i] = adjusted_rand_score(y_true, labels)
97             nmi_values[i] = normalized_mutual_info_score(y_true, labels)
98
99     return KMeansAnalysisResult(

```

```

100     k_values=k_values,
101     sse_values=sse_values,
102     silhouette_values=silhouette_values,
103     calinski_harabasz_values=calinski_harabasz_values,
104     davies_bouldin_values=davies_bouldin_values,
105     ari_values=ari_values,
106     nmi_values=nmi_values,
107 )
108
109
110 def analyze_dbSCAN_parameters(
111     X: np.ndarray,
112     y_true: np.ndarray,
113     eps_range: np.ndarray = None,
114     min_samples_range: np.ndarray = None,
115 ) -> DBSCANAnalysisResult:
116     """
117     Analyze_DBSCAN_performance_across_different_eps_and_min_samples_combinations.
118
119     Args:
120         X: Feature matrix
121         y_true: Ground truth labels
122         eps_range: Array of eps values to test
123         min_samples_range: Array of min samples values to test
124
125     Returns:
126         DBSCANAnalysisResult with all computed metrics
127     """
128     # Default ranges based on data statistics
129     if eps_range is None:
130         # Use k-distance based heuristic
131         from sklearn.neighbors import NearestNeighbors
132         nn = NearestNeighbors(n_neighbors=5)
133         nn.fit(X)
134         distances, _ = nn.kneighbors(X)
135         k_dist = np.sort(distances[:, -1])
136         eps_min = np.percentile(k_dist, 10)
137         eps_max = np.percentile(k_dist, 95)
138         eps_range = np.linspace(eps_min, eps_max, 15)
139
140     if min_samples_range is None:
141         min_samples_range = np.array([3, 5, 7, 10, 15, 20])
142
143     n_eps = len(eps_range)
144     n_min_samples = len(min_samples_range)
145
146     ari_matrix = np.zeros((n_eps, n_min_samples))
147     nmi_matrix = np.zeros((n_eps, n_min_samples))
148     n_clusters_matrix = np.zeros((n_eps, n_min_samples))
149     noise_ratio_matrix = np.zeros((n_eps, n_min_samples))
150
151     for i, eps in enumerate(eps_range):
152         for j, min_samples in enumerate(min_samples_range):
153             dbSCAN = DBSCAN(eps=eps, min_samples=int(min_samples))
154             labels = dbSCAN.fit_predict(X)
155
156             # Count clusters (excluding noise)
157             n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
158             n_clusters_matrix[i, j] = n_clusters
159
160             # Noise ratio
161             noise_ratio = np.sum(labels == -1) / len(labels)
162             noise_ratio_matrix[i, j] = noise_ratio
163
164             # External metrics (filter noise for fair comparison)

```

```

165     mask = labels != -1
166     if mask.sum() > 1 and n_clusters >= 1:
167         # Only compute ARI/NMI if we have valid clusters
168         if n_clusters >= 2:
169             ari_matrix[i, j] = adjusted_rand_score(y_true[mask], labels[mask])
170             nmi_matrix[i, j] = normalized_mutual_info_score(y_true[mask], labels[
171                 mask])
172         else:
173             ari_matrix[i, j] = 0
174             nmi_matrix[i, j] = 0
175     else:
176         ari_matrix[i, j] = -1 # Invalid clustering
177         nmi_matrix[i, j] = -1
178
179     return DBSCANAnalysisResult(
180         eps_values=eps_range,
181         min_samples_values=min_samples_range,
182         ari_matrix=ari_matrix,
183         nmi_matrix=nmi_matrix,
184         n_clusters_matrix=n_clusters_matrix,
185         noise_ratio_matrix=noise_ratio_matrix,
186     )
187
188 def plot_kmeans_elbow(
189     result: KMeansAnalysisResult,
190     dataset_name: str,
191     true_k: Optional[int] = None,
192     save_path: Optional[Path] = None,
193     figsize: Tuple[int, int] = (12, 8),
194 ) -> plt.Figure:
195     """
196     Plot_K-Means_elbow_analysis_with_multiple_metrics.
197
198     Creates a 2x2 subplot:
199     - SSE (Elbow Method)
200     - Silhouette Score
201     - ARI/NMI (external metrics)
202     - Davies-Bouldin / Calinski-Harabasz
203     """
204     fig, axes = plt.subplots(2, 2, figsize=figsize)
205
206     k_values = result.k_values
207
208     # Plot 1: SSE (Elbow Method)
209     ax1 = axes[0, 0]
210     ax1.plot(k_values, result.sse_values, 'b-o', linewidth=2, markersize=8)
211     ax1.set_xlabel('Number_of_Clusters_(k)', fontsize=11)
212     ax1.set_ylabel('SSE_(Inertia)', fontsize=11)
213     ax1.set_title('Elbow_Method_(SSE)', fontsize=12, fontweight='bold')
214     ax1.grid(True, alpha=0.3)
215     if true_k:
216         ax1.axvline(x=true_k, color='r', linestyle='--', label=f'True_k={true_k}')
217         ax1.legend()
218
219     # Plot 2: Silhouette Score
220     ax2 = axes[0, 1]
221     ax2.plot(k_values, result.silhouette_values, 'g-o', linewidth=2, markersize=8)
222     ax2.set_xlabel('Number_of_Clusters_(k)', fontsize=11)
223     ax2.set_ylabel('Silhouette_Score', fontsize=11)
224     ax2.set_title('Silhouette_Score', fontsize=12, fontweight='bold')
225     ax2.grid(True, alpha=0.3)
226     # Mark optimal k (max silhouette)
227     optimal_k_sil = k_values[np.argmax(result.silhouette_values)]
228     ax2.axvline(x=optimal_k_sil, color='orange', linestyle='--',

```

```

229         label=f'Optimal_k={optimal_k_sil})')
230     if true_k:
231         ax2.axvline(x=true_k, color='r', linestyle='--', label=f'True_k={true_k}')
232     ax2.legend()
233
234 # Plot 3: External Metrics (ARI/NMI)
235 ax3 = axes[1, 0]
236 if result.ari_values is not None:
237     ax3.plot(k_values, result.ari_values, 'r-o', linewidth=2, markersize=8, label='ARI')
238     ax3.plot(k_values, result.nmi_values, 'm-s', linewidth=2, markersize=8, label='NMI')
239     ax3.set_xlabel('Number_of_Clusters_(k)', fontsize=11)
240     ax3.set_ylabel('Score', fontsize=11)
241     ax3.set_title('External_Metrics_(ARI_&_NMI)', fontsize=12, fontweight='bold')
242     ax3.legend()
243     ax3.grid(True, alpha=0.3)
244     if true_k:
245         ax3.axvline(x=true_k, color='r', linestyle='--', alpha=0.5)
246     else:
247         ax3.text(0.5, 0.5, 'No_ground_truth_available', ha='center', va='center',
248                 transform=ax3.transAxes, fontsize=12)
249     ax3.set_title('External_Metrics_(N/A)', fontsize=12, fontweight='bold')
250
251 # Plot 4: Davies-Bouldin and Calinski-Harabasz
252 ax4 = axes[1, 1]
253 ax4_twin = ax4.twinx()
254
255 line1, = ax4.plot(k_values, result.davies_bouldin_values, 'c-o',
256                     linewidth=2, markersize=8, label='Davies-Bouldin_( )')
257 ax4.set_xlabel('Number_of_Clusters_(k)', fontsize=11)
258 ax4.set_ylabel('Davies-Bouldin_Index', color='c', fontsize=11)
259 ax4.tick_params(axis='y', labelcolor='c')
260
261 line2, = ax4_twin.plot(k_values, result.calinski_harabasz_values, 'y-s',
262                         linewidth=2, markersize=8, label='Calinski-Harabasz_( )')
263 ax4_twin.set_ylabel('Calinski-Harabasz_Index', color='y', fontsize=11)
264 ax4_twin.tick_params(axis='y', labelcolor='y')
265
266 ax4.set_title('Other_Internal_Metrics', fontsize=12, fontweight='bold')
267 ax4.legend([line1, line2], ['Davies-Bouldin_( )', 'Calinski-Harabasz_( )'], loc='upper_right')
268 ax4.grid(True, alpha=0.3)
269
270 plt.suptitle(f'K-Means_Hyperparameter_Analysis:{dataset_name}',
271             fontsize=14, fontweight='bold', y=1.02)
272 plt.tight_layout()
273
274 if save_path:
275     save_path.parent.mkdir(parents=True, exist_ok=True)
276     fig.savefig(save_path, dpi=150, bbox_inches='tight')
277     print(f"Saved: {save_path}")
278
279 return fig
280
281
282 def plot_dbscan_heatmaps(
283     result: DBSCANAnalysisResult,
284     dataset_name: str,
285     save_path: Optional[Path] = None,
286     figsize: Tuple[int, int] = (16, 12),
287 ) -> plt.Figure:
288     """
289     Plot_DBSCAN_parameter_sensitivity_as_heatmaps.
290

```

```

291     """Creates a 2x2 subplot:
292     - ARI heatmap
293     - NMI heatmap
294     - Number_of_clusters heatmap
295     - Noise_ratio heatmap
296     """
297     fig, axes = plt.subplots(2, 2, figsize=figsize)
298
299     eps_labels = [f'{e:.2f}' for e in result.eps_values]
300     ms_labels = [str(int(m)) for m in result.min_samples_values]
301
302     # ARI Heatmap
303     ax1 = axes[0, 0]
304     # Mask invalid values
305     ari_masked = np.ma.masked_where(result.ari_matrix < 0, result.ari_matrix)
306     im1 = ax1.imshow(ari_masked, cmap='RdYlGn', aspect='auto', vmin=0, vmax=1)
307     ax1.set_xticks(np.arange(len(ms_labels)))
308     ax1.set_yticks(np.arange(len(eps_labels)))
309     ax1.set_xticklabels(ms_labels)
310     ax1.set_yticklabels(eps_labels)
311     ax1.set_xlabel('min_samples', fontsize=11)
312     ax1.set_ylabel('eps', fontsize=11)
313     ax1.set_title('Adjusted_Rand_Index_(ARI)', fontsize=12, fontweight='bold')
314     plt.colorbar(im1, ax=ax1)
315
316     # Add text annotations
317     for i in range(len(eps_labels)):
318         for j in range(len(ms_labels)):
319             val = result.ari_matrix[i, j]
320             if val >= 0:
321                 color = 'white' if val < 0.3 or val > 0.7 else 'black'
322                 ax1.text(j, i, f'{val:.2f}', ha='center', va='center',
323                         color=color, fontsize=8)
324
325     # NMI Heatmap
326     ax2 = axes[0, 1]
327     nmi_masked = np.ma.masked_where(result.nmi_matrix < 0, result.nmi_matrix)
328     im2 = ax2.imshow(nmi_masked, cmap='RdYlGn', aspect='auto', vmin=0, vmax=1)
329     ax2.set_xticks(np.arange(len(ms_labels)))
330     ax2.set_yticks(np.arange(len(eps_labels)))
331     ax2.set_xticklabels(ms_labels)
332     ax2.set_yticklabels(eps_labels)
333     ax2.set_xlabel('min_samples', fontsize=11)
334     ax2.set_ylabel('eps', fontsize=11)
335     ax2.set_title('Normalized_Mutual_Information_(NMI)', fontsize=12, fontweight='bold')
336     plt.colorbar(im2, ax=ax2)
337
338     for i in range(len(eps_labels)):
339         for j in range(len(ms_labels)):
340             val = result.nmi_matrix[i, j]
341             if val >= 0:
342                 color = 'white' if val < 0.3 or val > 0.7 else 'black'
343                 ax2.text(j, i, f'{val:.2f}', ha='center', va='center',
344                         color=color, fontsize=8)
345
346     # Number of Clusters Heatmap
347     ax3 = axes[1, 0]
348     im3 = ax3.imshow(result.n_clusters_matrix, cmap='Blues', aspect='auto')
349     ax3.set_xticks(np.arange(len(ms_labels)))
350     ax3.set_yticks(np.arange(len(eps_labels)))
351     ax3.set_xticklabels(ms_labels)
352     ax3.set_yticklabels(eps_labels)
353     ax3.set_xlabel('min_samples', fontsize=11)
354     ax3.set_ylabel('eps', fontsize=11)
355     ax3.set_title('Number_of_Clusters_Found', fontsize=12, fontweight='bold')

```

```

356     plt.colorbar(im3, ax=ax3)
357
358     for i in range(len(eps_labels)):
359         for j in range(len(ms_labels)):
360             val = int(result.n_clusters_matrix[i, j])
361             ax3.text(j, i, str(val), ha='center', va='center',
362                     color='white' if val > 5 else 'black', fontsize=8)
363
364     # Noise Ratio Heatmap
365     ax4 = axes[1, 1]
366     im4 = ax4.imshow(result.noise_ratio_matrix, cmap='Reds', aspect='auto', vmin=0, vmax
367                     =1)
368     ax4.set_xticks(np.arange(len(ms_labels)))
369     ax4.set_yticks(np.arange(len(eps_labels)))
370     ax4.set_xticklabels(ms_labels)
371     ax4.set_yticklabels(eps_labels)
372     ax4.set_xlabel('min_samples', fontsize=11)
373     ax4.set_ylabel('eps', fontsize=11)
374     ax4.set_title('Noise_Ratio', fontsize=12, fontweight='bold')
375     plt.colorbar(im4, ax=ax4)
376
377     for i in range(len(eps_labels)):
378         for j in range(len(ms_labels)):
379             val = result.noise_ratio_matrix[i, j]
380             color = 'white' if val > 0.5 else 'black'
381             ax4.text(j, i, f'{val:.2f}', ha='center', va='center',
382                     color=color, fontsize=8)
383
384     plt.suptitle(f'DBSCAN_Parameter_Sensitivity:{dataset_name}',
385                  fontsize=14, fontweight='bold', y=1.02)
386     plt.tight_layout()
387
388     if save_path:
389         save_path.parent.mkdir(parents=True, exist_ok=True)
390         fig.savefig(save_path, dpi=150, bbox_inches='tight')
391         print(f"Saved:{save_path}")
392
393     return fig
394
395 def plot_metric_conflict_analysis(
396     kmeans_result: KMeansAnalysisResult,
397     dataset_name: str,
398     true_k: int,
399     save_path: Optional[Path] = None,
400     figsize: Tuple[int, int] = (10, 6),
401 ) -> plt.Figure:
402     """
403     Visualize conflicts between internal and external metrics.
404
405     Shows cases where internal metrics suggest different k than external metrics.
406     """
407     fig, ax = plt.subplots(figsize=figsize)
408
409     k_values = kmeans_result.k_values
410
411     # Normalize all metrics to [0, 1] for comparison
412     def normalize(arr, higher_is_better=True):
413         arr = np.array(arr)
414         if not higher_is_better:
415             arr = -arr
416         return (arr - arr.min()) / (arr.max() - arr.min() + 1e-10)
417
418     # Plot normalized metrics
419     ax.plot(k_values, normalize(kmeans_result.silhouette_values),

```

```

420     'g-o', linewidth=2, label='Silhouette_(normalized)')
421 ax.plot(k_values, normalize(kmeans_result.davies_bouldin_values, higher_is_better=
422     False),
423         'c-s', linewidth=2, label='Davies-Bouldin_(normalized,_inverted)')
424
425 if kmeans_result.ari_values is not None:
426     ax.plot(k_values, normalize(kmeans_result.ari_values),
427             'r^', linewidth=2, label='ARI_(normalized)')
428     ax.plot(k_values, normalize(kmeans_result.nmi_values),
429             'm-d', linewidth=2, label='NMI_(normalized)')
430
431 # Mark true k
432 ax.axvline(x=true_k, color='black', linestyle='--', linewidth=2,
433             label=f'True_k={true_k}')
434
435 # Find optimal k for each metric
436 optimal_sil = k_values[np.argmax(kmeans_result.silhouette_values)]
437 optimal_db = k_values[np.argmin(kmeans_result.davies_bouldin_values)]
438
439 ax.set_xlabel('Number_of_Clusters_(k)', fontsize=12)
440 ax.set_ylabel('Normalized_Score', fontsize=12)
441 ax.set_title(f'Metric_Conflict_Analysis_{dataset_name}\n'
442             f'(Silhouette_optimal:_k={optimal_sil}, Davies-Bouldin_optimal:_k={'
443                 optimal_db})',
444             fontsize=12, fontweight='bold')
445 ax.legend(loc='upper_right')
446 ax.grid(True, alpha=0.3)
447 ax.set_xticks(k_values)
448
449 plt.tight_layout()
450
451 if save_path:
452     save_path.parent.mkdir(parents=True, exist_ok=True)
453     fig.savefig(save_path, dpi=150, bbox_inches='tight')
454     print(f"Saved: {save_path}")
455
456 return fig
457
458
459 def run_experiment2(
460     datasets: List[ClusteringDataset] = None,
461     output_dir: Path = Path("outputs/experiment2"),
462     random_state: int = 42,
463 ) -> Dict:
464     """
465     Run_Experiment_2:_Hyperparameter_Sensitivity_and_Metric_Conflict.
466
467     Args:
468         datasets: List_of_datasets_to_use_(default:_synthetic_datasets)
469         output_dir: Directory_to_save_outputs
470         random_state: Random_seed
471
472     Returns:
473         Dictionary_containing_all_results
474     """
475     print("=" * 70)
476     print("Experiment_2:_Hyperparameter_Sensitivity_and_Metric_Conflict")
477     print("=" * 70)
478
479     output_dir.mkdir(parents=True, exist_ok=True)
480
481     # Use synthetic datasets if not provided
482     if datasets is None:
483         datasets = get_synthetic_datasets(random_state=random_state)
484

```

```

483     results = {
484         'kmeans': {},
485         'dbSCAN': {}
486     }
487
488     # =====
489     # 2.1 K-Means: Choice of Number of Clusters
490     # =====
491     print("\n[2.1] K-Means: Number_of_Clusters_Analysis")
492     print("-" * 50)
493
494     for dataset in datasets:
495         print(f"\nDataset: {dataset.name}")
496
497         # Standardize data
498         X_scaled, _ = standardize_data(dataset.X)
499
500         # Analyze k selection
501         kmeans_result = analyze_kmeans_k_selection(
502             X=X_scaled,
503             y_true=dataset.y,
504             k_range=range(2, 11),
505             random_state=random_state,
506         )
507         results['kmeans'][dataset.name] = kmeans_result
508
509         # Find optimal k values
510         optimal_sil = kmeans_result.k_values[np.argmax(kmeans_result.silhouette_values)]
511         optimal_db = kmeans_result.k_values[np.argmin(kmeans_result.davies_bouldin_values)]
512         ]
513         optimal_ari = kmeans_result.k_values[np.argmax(kmeans_result.ari_values)] if
514             kmeans_result.ari_values is not None else None
515
516         print(f"True_k: {dataset.n_clusters}")
517         print(f"Optimal_k_(Silhouette): {optimal_sil}")
518         print(f"Optimal_k_(Davies-Bouldin): {optimal_db}")
519         if optimal_ari:
520             print(f"Optimal_k_(ARI): {optimal_ari}")
521
522         # Check for metric conflict
523         if optimal_sil != dataset.n_clusters:
524             print(f"Silhouette suggests k={optimal_sil}, but true_k={dataset.
525                 n_clusters}")
526
527         # Plot elbow analysis
528         fig = plot_kmeans_elbow(
529             result=kmeans_result,
530             dataset_name=dataset.name,
531             true_k=dataset.n_clusters,
532             save_path=output_dir / f"kmeans_elbow_{dataset.name.lower().replace(' ', '_')}.
533                 png",
534         )
535         plt.close(fig)
536
537         # Plot metric conflict analysis
538         fig = plot_metric_conflict_analysis(
539             kmeans_result=kmeans_result,
540             dataset_name=dataset.name,
541             true_k=dataset.n_clusters,
542             save_path=output_dir / f"metric_conflict_{dataset.name.lower().replace(' ', '_')}.
543                 png",
544         )
545         plt.close(fig)
546
547     # =====

```

```

543 # 2.2 DBSCAN: Density Parameters
544 # =====
545 print("\n[2.2]_DBSCAN:_Density_Parameters_Analysis")
546 print("-" * 50)
547
548 for dataset in datasets:
549     print(f"\nDataset:{dataset.name}")
550
551     # Standardize data
552     X_scaled, _ = standardize_data(dataset.X)
553
554     # Analyze DBSCAN parameters
555     dbscan_result = analyze_dbSCAN_parameters(
556         X=X_scaled,
557         y_true=dataset.y,
558     )
559     results['dbSCAN'][dataset.name] = dbscan_result
560
561     # Find best parameters
562     best_idx = np.unravel_index(np.argmax(dbscan_result.ari_matrix), dbscan_result.
563         ari_matrix.shape)
564     best_eps = dbscan_result.eps_values[best_idx[0]]
565     best_min_samples = int(dbscan_result.min_samples_values[best_idx[1]])
566     best_ari = dbscan_result.ari_matrix[best_idx]
567
568     print(f"__Best_params:__eps={best_eps:.3f},__min_samples={best_min_samples}")
569     print(f"__Best_ARI:__{best_ari:.4f}")
570     print(f"__Clusters_at_best_params:__{int(dbscan_result.n_clusters_matrix[best_idx])}
571         ")
572     print(f"__Noise_ratio_at_best_params:__{dbscan_result.noise_ratio_matrix[best_idx]
573         }:.2%")
574
575     # Plot DBSCAN heatmaps
576     fig = plot_dbSCAN_heatmaps(
577         result=dbscan_result,
578         dataset_name=dataset.name,
579         save_path=output_dir / f"dbSCAN_heatmap_{dataset.name.lower().replace(' ', '_')}.png",
580     )
581     plt.close(fig)
582
583     # =====
584     # Create summary table
585     # =====
586     print("\n[Summary]_Creating_summary_tables...")
587
588     # K-Means summary
589     kmeans_summary = []
590     for ds_name, result in results['kmeans'].items():
591         row = {
592             'Dataset': ds_name,
593             'True_k': next(d.n_clusters for d in datasets if d.name == ds_name),
594             'Optimal_k_(Silhouette)': int(result.k_values[np.argmax(result.
595                 silhouette_values)]),
596             'Optimal_k_(DB)': int(result.k_values[np.argmin(result.davies_bouldin_values)])
597             ],
598             'Optimal_k_(ARI)': int(result.k_values[np.argmax(result.ari_values)]) if
599             result.ari_values is not None else 'N/A',
600             'Max_ARI': f"{np.max(result.ari_values):.4f}" if result.ari_values is not None
601             else 'N/A',
602         }
603         kmeans_summary.append(row)
604
605     df_kmeans = pd.DataFrame(kmeans_summary)
606     df_kmeans.to_csv(output_dir / "kmeans_k_selection_summary.csv", index=False)

```

```

600
601     # DBSCAN summary
602     dbscan_summary = []
603     for ds_name, result in results['dbscan'].items():
604         best_idx = np.unravel_index(np.argmax(result.ari_matrix), result.ari_matrix.shape)
605         row = {
606             'Dataset': ds_name,
607             'Best_eps': f'{result.eps_values[best_idx[0]]:.3f}',
608             'Best_min_samples': int(result.min_samples_values[best_idx[1]]),
609             'Best_ARI': f'{result.ari_matrix[best_idx]:.4f}',
610             'Clusters_Found': int(result.n_clusters_matrix[best_idx]),
611             'Noise_Ratio': f'{result.noise_ratio_matrix[best_idx]:.2%}'}
612         }
613         dbscan_summary.append(row)
614
615     df_dbscan = pd.DataFrame(dbscan_summary)
616     df_dbscan.to_csv(output_dir / "dbscan_parameter_summary.csv", index=False)
617
618     # Print summaries
619     print("\n" + "=" * 70)
620     print("K-MEANS_K_SELECTION_SUMMARY")
621     print("=" * 70)
622     print(df_kmeans.to_string(index=False))
623
624     print("\n" + "=" * 70)
625     print("DBSCAN_PARAMETER_SENSITIVITY_SUMMARY")
626     print("=" * 70)
627     print(df_dbscan.to_string(index=False))
628
629     # Analysis
630     print("\n" + "=" * 70)
631     print("ANALYSIS_AND_OBSERVATIONS")
632     print("=" * 70)
633     print("""
634     ↴Key Findings:
635
636 1. K-Means_k_Selection:
637     ↴Internal_metrics_(Silhouette,_DB)_may_disagree_with_external_metrics_(ARI,_NMI)
638     ↴The_elbow_method_often_suggests_different_k_than_ground_truth
639     ↴This_highlights_the_challenge_of_unsupervised_model_selection
640
641 2. DBSCAN_Parameter_Sensitivity:
642     ↴DBSCAN_is_highly_sensitive_to_eps_and_min_samples
643     ↴Small_eps_may_noise_points_(all_noise_in_extreme_cases)
644     ↴Large_eps_single_cluster_(everything_merged)
645     ↴Optimal_parameter_region_is_often_narrow
646     ↴Different_datasets_require_different_parameter_tuning
647
648 3. Metric_Conflict_Implications:
649     ↴Internal_metrics_optimize_cluster_compactness/separation
650     ↴External_metrics_measure_alignment_with_ground_truth
651     ↴In_practice,_ground_truth_is_unavailable,_making_model_selection_difficult
652     """
653
654     print(f"\n    ↴All_results_saved_to:{output_dir}/")
655     print("        ↴Experiment_2_completed_successfully!")
656
657     return results

```

A.7. `src/experiment3.py`

```

1 """
2 Experiment_3:_High-Dimensional_Data_and_Dimensionality_Reduction
3

```

```

4 This_module_investigates_the_impact_of_dimensionality_on_clustering_quality_and_efficiency
5 .
6 Procedure:
7 1. Apply_clustering_algorithms_directly_to_original_features
8 2. Apply_PCA_to_reduce_dimensionality_(e.g.,_10_and_20_dimensions)
9 3. Repeat_clustering_on_reduced_representations
10 4. Compare:_Runtime,_Internal_metrics,_External_metrics
11 5. Optional:_t-SNE_plots_for_qualitative_inspection
12 """
13
14 import numpy as np
15 import pandas as pd
16 import matplotlib.pyplot as plt
17 import seaborn as sns
18 from pathlib import Path
19 from typing import Dict, List, Tuple, Optional
20 from dataclasses import dataclass
21 import time
22 from sklearn.decomposition import PCA
23 from sklearn.manifold import TSNE
24
25 from .datasets import (
26     ClusteringDataset,
27     standardize_data,
28     load_digits_dataset,
29     load_iris_dataset,
30     load_wine_dataset,
31 )
32 from .clustering import (
33     get_main_algorithms,
34     ClusteringAlgorithm,
35     ClusteringResult,
36 )
37 from .metrics import compute_all_metrics, ClusteringMetrics
38
39
40 @dataclass
41 class DimensionalityExperimentResult:
42     """Results_from_a_single_clustering_run_with_timing."""
43     algorithm_name: str
44     n_dimensions: int
45     dimension_label: str # e.g., "Original (64)", "PCA-10", "PCA-20"
46     runtime_seconds: float
47     metrics: ClusteringMetrics
48     labels: np.ndarray
49
50
51 def apply_pca(
52     X: np.ndarray,
53     n_components: int,
54     random_state: int = 42,
55 ) -> Tuple[np.ndarray, PCA, float]:
56     """
57     Apply_PCA_dimensionality_reduction.
58
59     Args:
60         X: Feature_matrix
61         n_components: Number_of_principal_components
62         random_state: Random_seed
63
64     Returns:
65         Tuple_of_(transformed_data, fitted_PCA_object, explained_variance_ratio)
66     """
67     pca = PCA(n_components=n_components, random_state=random_state)

```

```

68     X_reduced = pca.fit_transform(X)
69     explained_variance = np.sum(pca.explained_variance_ratio_)
70     return X_reduced, pca, explained_variance
71
72
73 def run_clustering_with_timing(
74     algorithm: ClusteringAlgorithm,
75     X: np.ndarray,
76     y_true: np.ndarray,
77     n_dimensions: int,
78     dimension_label: str,
79 ) -> DimensionalityExperimentResult:
80     """
81     Run a clustering algorithm and measure execution time.
82
83     Args:
84         algorithm: Clustering algorithm to run
85         X: Feature matrix
86         y_true: Ground truth labels
87         n_dimensions: Number of dimensions in X
88         dimension_label: Label for this dimension setting
89
90     Returns:
91         DimensionalityExperimentResult with timing and metrics
92     """
93     # Measure runtime
94     start_time = time.perf_counter()
95     result = algorithm.fit_predict(X)
96     end_time = time.perf_counter()
97
98     runtime = end_time - start_time
99
100    # Compute metrics
101    metrics = compute_all_metrics(X, result.labels, y_true)
102
103    return DimensionalityExperimentResult(
104        algorithm_name=result.algorithm_name,
105        n_dimensions=n_dimensions,
106        dimension_label=dimension_label,
107        runtime_seconds=runtime,
108        metrics=metrics,
109        labels=result.labels,
110    )
111
112
113 def plot_pca_explained_variance(
114     X: np.ndarray,
115     max_components: int = None,
116     save_path: Optional[Path] = None,
117     figsize: Tuple[int, int] = (12, 5),
118 ) -> plt.Figure:
119     """
120     Plot PCA explained variance analysis.
121
122     Creates two subplots:
123     - Individual explained variance ratio per component
124     - Cumulative explained variance ratio
125     """
126     if max_components is None:
127         max_components = min(X.shape[1], 50)
128
129     pca = PCA(n_components=max_components)
130     pca.fit(X)
131
132     fig, axes = plt.subplots(1, 2, figsize=figsize)

```

```

133
134     # Individual variance
135     ax1 = axes[0]
136     ax1.bar(range(1, max_components + 1), pca.explained_variance_ratio_,
137             alpha=0.7, color='steelblue')
138     ax1.set_xlabel('Principal Component', fontsize=11)
139     ax1.set_ylabel('Explained Variance Ratio', fontsize=11)
140     ax1.set_title('Individual Explained Variance', fontsize=12, fontweight='bold')
141     ax1.grid(True, alpha=0.3)
142
143     # Cumulative variance
144     ax2 = axes[1]
145     cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
146     ax2.plot(range(1, max_components + 1), cumulative_variance,
147             'b-o', linewidth=2, markersize=4)
148     ax2.axhline(y=0.9, color='r', linestyle='--', label='90% variance')
149     ax2.axhline(y=0.95, color='orange', linestyle='--', label='95% variance')
150     ax2.set_xlabel('Number of Components', fontsize=11)
151     ax2.set_ylabel('Cumulative Explained Variance', fontsize=11)
152     ax2.set_title('Cumulative Explained Variance', fontsize=12, fontweight='bold')
153     ax2.legend()
154     ax2.grid(True, alpha=0.3)
155     ax2.set_ylim([0, 1.05])
156
157     # Find components for 90% and 95% variance
158     n_90 = np.argmax(cumulative_variance >= 0.9) + 1
159     n_95 = np.argmax(cumulative_variance >= 0.95) + 1
160     ax2.axvline(x=n_90, color='r', linestyle=':', alpha=0.5)
161     ax2.axvline(x=n_95, color='orange', linestyle=':', alpha=0.5)
162
163     plt.suptitle(f'PCA Analysis (Original: {X.shape[1]} dimensions)\n'
164                  f'90% variance: {n_90} components, 95% variance: {n_95} components',
165                  fontsize=12, fontweight='bold')
166     plt.tight_layout()
167
168     if save_path:
169         save_path.parent.mkdir(parents=True, exist_ok=True)
170         fig.savefig(save_path, dpi=150, bbox_inches='tight')
171         print(f"Saved: {save_path}")
172
173     return fig
174
175
176 def plot_tsne_visualization(
177     X: np.ndarray,
178     y_true: np.ndarray,
179     labels_dict: Dict[str, np.ndarray],
180     dataset_name: str,
181     save_path: Optional[Path] = None,
182     figsize: Tuple[int, int] = (16, 10),
183     random_state: int = 42,
184 ) -> plt.Figure:
185     """
186     Create t-SNE visualizations comparing ground truth and clustering results.
187
188     Args:
189         X: Feature matrix (will be reduced to 2D via t-SNE)
190         y_true: Ground truth labels
191         labels_dict: Dictionary mapping algorithm names to predicted labels
192         dataset_name: Name of dataset for title
193         save_path: Path to save figure
194         figsize: Figure size
195         random_state: Random seed for t-SNE
196
197     Returns:

```

```

198     """matplotlib.Figure object
199     """
200     # Apply t-SNE for 2D visualization
201     print("...Computing t-SNE embedding (this may take a moment)...")
202     tsne = TSNE(n_components=2, random_state=random_state, perplexity=30, n_iter=1000)
203     X_tsne = tsne.fit_transform(X)
204
205     n_plots = len(labels_dict) + 1 # +1 for ground truth
206     n_cols = min(3, n_plots)
207     n_rows = (n_plots + n_cols - 1) // n_cols
208
209     fig, axes = plt.subplots(n_rows, n_cols, figsize=figsize)
210     axes = np.atleast_2d(axes).flatten()
211
212     colors = plt.cm.tab10(np.linspace(0, 1, 10))
213
214     def plot_scatter(ax, X_2d, labels, title):
215         unique_labels = np.unique(labels)
216         for label in unique_labels:
217             mask = labels == label
218             if label == -1:
219                 ax.scatter(X_2d[mask, 0], X_2d[mask, 1], c='gray', marker='x',
220                             s=20, alpha=0.5, label='Noise')
221             else:
222                 color = colors[label % len(colors)]
223                 ax.scatter(X_2d[mask, 0], X_2d[mask, 1], c=[color],
224                             s=20, alpha=0.7)
225         ax.set_title(title, fontsize=11, fontweight='bold')
226         ax.set_xticks([])
227         ax.set_yticks([])
228
229     # Plot ground truth
230     plot_scatter(axes[0], X_tsne, y_true, f'Ground Truth\n({len(np.unique(y_true))} classes)')
231
232     # Plot each algorithm's results
233     for idx, (algo_name, labels) in enumerate(labels_dict.items(), start=1):
234         n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
235         plot_scatter(axes[idx], X_tsne, labels, f'{algo_name}\n({n_clusters} clusters)')
236
237     # Hide unused subplots
238     for idx in range(n_plots, len(axes)):
239         axes[idx].set_visible(False)
240
241     plt.suptitle(f't-SNE Visualization: {dataset_name}', fontsize=14, fontweight='bold')
242     plt.tight_layout()
243
244     if save_path:
245         save_path.parent.mkdir(parents=True, exist_ok=True)
246         fig.savefig(save_path, dpi=150, bbox_inches='tight')
247         print(f"Saved: {save_path}")
248
249     return fig
250
251
252     def plot_dimensionality_comparison(
253         results: List[DimensionalityExperimentResult],
254         metric_name: str,
255         dataset_name: str,
256         save_path: Optional[Path] = None,
257         figsize: Tuple[int, int] = (12, 6),
258     ) -> plt.Figure:
259         """
260         Plot comparison of a metric across different dimensionalities.
261         """

```

```

262     # Organize data by algorithm and dimension
263     algorithms = sorted(set(r.algorithm_name for r in results))
264     dimensions = sorted(set(r.dimension_label for r in results),
265                         key=lambda x: (0 if 'Original' in x else int(x.split('-')[1])) if '-'
266                           in x else 999))
267
268     # Create grouped bar chart
269     fig, ax = plt.subplots(figsize=figsize)
270
271     x = np.arange(len(algorithms))
272     width = 0.8 / len(dimensions)
273
274     for i, dim_label in enumerate(dimensions):
275         values = []
276         for algo in algorithms:
277             result = next((r for r in results
278                            if r.algorithm_name == algo and r.dimension_label == dim_label),
279                            None)
280
281             if result:
282                 value = getattr(result.metrics, metric_name, None)
283                 values.append(value if value is not None else 0)
284             else:
285                 values.append(0)
286
287         offset = (i - len(dimensions)/2 + 0.5) * width
288         bars = ax.bar(x + offset, values, width, label=dim_label, alpha=0.8)
289
290         ax.set_xlabel('Algorithm', fontsize=12)
291         ax.set_ylabel(metric_name.replace('_', '_').title(), fontsize=12)
292         ax.set_title(f'{metric_name.replace("_", "_").title()} by Dimensionality:{dataset_name}', fontsize=12, fontweight='bold')
293         ax.set_xticks(x)
294         ax.set_xticklabels([a.split('(')[0].strip() for a in algorithms], rotation=45, ha='right')
295         ax.legend(title='Dimensions')
296         ax.grid(True, alpha=0.3, axis='y')
297
298         plt.tight_layout()
299
300     if save_path:
301         save_path.parent.mkdir(parents=True, exist_ok=True)
302         fig.savefig(save_path, dpi=150, bbox_inches='tight')
303         print(f"Saved: {save_path}")
304
305     return fig
306
307 def plot_runtime_comparison(
308     results: List[DimensionalityExperimentResult],
309     dataset_name: str,
310     save_path: Optional[Path] = None,
311     figsize: Tuple[int, int] = (12, 6),
312 ) -> plt.Figure:
313     """
314     Plot runtime comparison across dimensionalities.
315     """
316     algorithms = sorted(set(r.algorithm_name for r in results))
317     dimensions = sorted(set(r.dimension_label for r in results),
318                         key=lambda x: (0 if 'Original' in x else int(x.split('-')[1])) if '-'
319                           in x else 999))
320
321     fig, ax = plt.subplots(figsize=figsize)
322
323     x = np.arange(len(algorithms))

```

```

322     width = 0.8 / len(dimensions)
323
324     for i, dim_label in enumerate(dimensions):
325         values = []
326         for algo in algorithms:
327             result = next((r for r in results
328                           if r.algorithm_name == algo and r.dimension_label == dim_label),
329                           None)
330             values.append(result.runtime_seconds * 1000 if result else 0) # Convert to ms
331
332         offset = (i - len(dimensions)/2 + 0.5) * width
333         bars = ax.bar(x + offset, values, width, label=dim_label, alpha=0.8)
334
335     ax.set_xlabel('Algorithm', fontsize=12)
336     ax.set_ylabel('Runtime_(ms)', fontsize=12)
337     ax.set_title(f'Runtime Comparison by Dimensionality:_{dataset_name}',
338                  fontsize=12, fontweight='bold')
339     ax.set_xticks(x)
340     ax.set_xticklabels([a.split('(')[0].strip() for a in algorithms], rotation=45, ha='right')
341     ax.legend(title='Dimensions')
342     ax.grid(True, alpha=0.3, axis='y')
343
344     plt.tight_layout()
345
346     if save_path:
347         save_path.parent.mkdir(parents=True, exist_ok=True)
348         fig.savefig(save_path, dpi=150, bbox_inches='tight')
349         print(f"Saved:{save_path}")
350
351
352 def plot_comprehensive_summary(
353     results: List[DimensionalityExperimentResult],
354     dataset_name: str,
355     save_path: Optional[Path] = None,
356     figsize: Tuple[int, int] = (16, 12),
357 ) -> plt.Figure:
358     """
359     Create_a_comprehensive_summary_plot_with_multiple_metrics.
360     """
361     fig, axes = plt.subplots(2, 3, figsize=figsize)
362
363     algorithms = sorted(set(r.algorithm_name for r in results))
364     dimensions = sorted(set(r.dimension_label for r in results),
365                         key=lambda x: (0 if 'Original' in x else int(x.split('-')[1]) if '-'
366                                         in x else 999))
367
368     x = np.arange(len(algorithms))
369     width = 0.8 / len(dimensions)
370
371     metrics_to_plot = [
372         ('ari', 'Adjusted_Rand_Index_(ARI)', True),
373         ('nmi', 'Normalized_Mutual_Information_(NMI)', True),
374         ('silhouette', 'Silhouette_Score', True),
375         ('davies_bouldin', 'Davies-Bouldin_Index', False), # Lower is better
376         ('runtime_seconds', 'Runtime_(ms)', False),
377     ]
378
379     for ax_idx, (metric_key, metric_title, higher_is_better) in enumerate(metrics_to_plot):
380         :
381             ax = axes.flatten()[ax_idx]
382
383             for i, dim_label in enumerate(dimensions):

```

```

383     values = []
384     for algo in algorithms:
385         result = next((r for r in results
386                         if r.algorithm_name == algo and r.dimension_label ==
387                         dim_label), None)
388         if result:
389             if metric_key == 'runtime_seconds':
390                 value = result.runtime_seconds * 1000 # Convert to ms
391             else:
392                 value = getattr(result.metrics, metric_key, None)
393             values.append(value if value is not None else 0)
394         else:
395             values.append(0)
396
397         offset = (i - len(dimensions)/2 + 0.5) * width
398         ax.bar(x + offset, values, width, label=dim_label, alpha=0.8)
399
400         ax.set_xlabel('Algorithm', fontsize=10)
401         ax.set_ylabel(metric_title, fontsize=10)
402         ax.set_title(metric_title, fontsize=11, fontweight='bold')
403         ax.set_xticks(x)
404         ax.set_xticklabels([a.split('(')[0].strip()[:10] for a in algorithms],
405                           rotation=45, ha='right', fontsize=8)
406         ax.grid(True, alpha=0.3, axis='y')
407         if ax_idx == 0:
408             ax.legend(title='Dimensions', fontsize=8)
409
410     # Hide the last subplot (we have 5 metrics, 6 subplots)
411     axes.flatten()[5].set_visible(False)
412
413 plt.suptitle(f'Dimensionality_Reduction_Impact:{dataset_name}',
414               fontsize=14, fontweight='bold')
415 plt.tight_layout()
416
417 if save_path:
418     save_path.parent.mkdir(parents=True, exist_ok=True)
419     fig.savefig(save_path, dpi=150, bbox_inches='tight')
420     print(f"Saved:{save_path}")
421
422 return fig
423
424 def run_experiment3(
425     dataset: ClusteringDataset = None,
426     pca_dimensions: List[int] = [10, 20, 30],
427     output_dir: Path = Path("outputs/experiment3"),
428     random_state: int = 42,
429 ) -> Dict:
430     """
431     Run_Experiment_3: High-Dimensional_Data_and_Dimensionality_Reduction.
432
433     Args:
434         dataset: Dataset_to_use_(default:Digits_dataset)
435         pca_dimensions: List_of_PCA_dimensions_to_test
436         output_dir: Directory_to_save_outputs
437         random_state: Random_seed
438
439     Returns:
440         Dictionary_containing_all_results
441     """
442     print("=" * 70)
443     print("Experiment_3: High-Dimensional_Data_and_Dimensionality_Reduction")
444     print("=" * 70)
445
446     output_dir.mkdir(parents=True, exist_ok=True)

```

```

447     # Use Digits dataset if not provided (recommended in README)
448     if dataset is None:
449         dataset = load_digits_dataset()
450
451
452     print(f"\nDataset: {dataset.name}")
453     print(f"Samples: {dataset.X.shape[0]}")
454     print(f"Original dimensions: {dataset.X.shape[1]}")
455     print(f"Classes: {dataset.n_clusters}")
456     print(f" {dataset.description}")
457
458     # Standardize data
459     X_scaled, _ = standardize_data(dataset.X)
460
461     # =====
462     # PCA Analysis
463     # =====
464     print("\n[1/4] PCA Explained Variance Analysis")
465     print("-" * 50)
466
467     fig = plot_pca_explained_variance(
468         X=X_scaled,
469         save_path=output_dir / "pca_explained_variance.png",
470     )
471     plt.close(fig)
472
473     # Show variance explained by chosen dimensions
474     for n_dim in pca_dimensions:
475         _, var_explained = apply_pca(X_scaled, n_dim, random_state)
476         print(f"PCA-{n_dim}: {var_explained:.2%} variance explained")
477
478     # =====
479     # Run Clustering at Different Dimensionalities
480     # =====
481     print("\n[2/4] Running Clustering at Different Dimensionalities")
482     print("-" * 50)
483
484     all_results: List[DimensionalityExperimentResult] = []
485
486     # Prepare data at different dimensionalities
487     data_versions = [
488         (X_scaled, dataset.X.shape[1], f"Original-{dataset.X.shape[1]}"),
489     ]
490
491     for n_dim in pca_dimensions:
492         X_pca, _, var = apply_pca(X_scaled, n_dim, random_state)
493         data_versions.append((X_pca, n_dim, f"PCA-{n_dim}"))
494
495     # Get algorithms
496     algorithms = get_main_algorithms(
497         n_clusters=dataset.n_clusters,
498         random_state=random_state,
499     )
500
501     # Run clustering for each data version and algorithm
502     for X_data, n_dim, dim_label in data_versions:
503         print(f"\n{dim_label}:")
504
505         for algo in algorithms:
506             # Create fresh algorithm instance (some algorithms cache state)
507             algo_fresh = type(algo).new_(type(algo))
508             algo_fresh.__dict__.update(algo.__dict__)
509
510             result = run_clustering_with_timing(
511                 algorithm=algo_fresh,

```

```

512         X=X_data,
513         y_true=dataset.y,
514         n_dimensions=n_dim,
515         dimension_label=dim_label,
516     )
517     all_results.append(result)
518
519     ari = result.metrics.ari if result.metrics.ari else 0
520     print(f"\n{result.algorithm_name}: ARI={ari:.4f}, "
521           f"Runtime={result.runtime_seconds*1000:.2f}ms")
522
523 # =====
524 # Generate Visualizations
525 # =====
526 print("\n[3/4] Generating Visualizations")
527 print("-" * 50)
528
529 # Comprehensive summary
530 fig = plot_comprehensive_summary(
531     results=all_results,
532     dataset_name=dataset.name,
533     save_path=output_dir / "comprehensive_summary.png",
534 )
535 plt.close(fig)
536
537 # Individual metric comparisons
538 for metric in ['ari', 'nmi', 'silhouette']:
539     fig = plot_dimensionality_comparison(
540         results=all_results,
541         metric_name=metric,
542         dataset_name=dataset.name,
543         save_path=output_dir / f"comparison_{metric}.png",
544     )
545     plt.close(fig)
546
547 # Runtime comparison
548 fig = plot_runtime_comparison(
549     results=all_results,
550     dataset_name=dataset.name,
551     save_path=output_dir / "runtime_comparison.png",
552 )
553 plt.close(fig)
554
555 # t-SNE visualization (on PCA-reduced data for speed)
556 print("\nGenerating t-SNE visualization...")
557 X_pca_30, _, _ = apply_pca(X_scaled, 30, random_state)
558
559 # Get labels from original dimension clustering
560 labels_dict = {}
561 for result in all_results:
562     if "Original" in result.dimension_label:
563         # Simplify algorithm name for display
564         simple_name = result.algorithm_name.split('(')[0].strip()
565         labels_dict[simple_name] = result.labels
566
567 fig = plot_tsne_visualization(
568     X=X_pca_30, # Use PCA-reduced for faster t-SNE
569     y_true=dataset.y,
570     labels_dict=labels_dict,
571     dataset_name=dataset.name,
572     save_path=output_dir / "tsne_visualization.png",
573     random_state=random_state,
574 )
575 plt.close(fig)
576

```

```

577 # =====
578 # Create Summary Tables
579 # =====
580 print("\n[4/4] Creating_Summary_Tables")
581 print("-" * 50)
582
583 # Create detailed results table
584 summary_data = []
585 for result in all_results:
586     row = {
587         'Algorithm': result.algorithm_name,
588         'Dimensions': result.dimension_label,
589         'ARI': f"{result.metrics.ari:.4f}" if result.metrics.ari else "N/A",
590         'NMI': f"{result.metrics.nmi:.4f}" if result.metrics.nmi else "N/A",
591         'Silhouette': f"{result.metrics.silhouette:.4f}" if result.metrics.silhouette
592             else "N/A",
593         'Davies-Bouldin': f"{result.metrics.davies_bouldin:.4f}" if result.metrics.
594             davies_bouldin else "N/A",
595         'Runtime_(ms)': f"{result.runtime_seconds*1000:.2f}",
596     }
597     summary_data.append(row)
598
599 df_summary = pd.DataFrame(summary_data)
600 df_summary.to_csv(output_dir / "experiment3_results.csv", index=False)
601
602 # Create comparison table (algorithms as rows, dimensions as columns for ARI)
603 pivot_ari = df_summary.pivot(index='Algorithm', columns='Dimensions', values='ARI')
604 pivot_ari.to_csv(output_dir / "ari_by_dimension.csv")
605
606 pivot_runtime = df_summary.pivot(index='Algorithm', columns='Dimensions', values='
607     Runtime_(ms)')
608 pivot_runtime.to_csv(output_dir / "runtime_by_dimension.csv")
609
610 # Print summaries
611 print("\n" + "=" * 70)
612 print("EXPERIMENT_3_RESULTS_SUMMARY")
613 print("=" * 70)
614 print(df_summary.to_string(index=False))
615
616 print("\n" + "=" * 70)
617 print("ARI_BY_DIMENSIONALITY")
618 print("=" * 70)
619 print(pivot_ari.to_string())
620
621 # Analysis
622 print("\n" + "=" * 70)
623 print("ANALYSIS_AND_OBSERVATIONS")
624 print("=" * 70)
625
626 # Find best performing dimension for each algorithm
627 print("\n Best_Dimensionality_by_Algorithm_(ARI):")
628 for algo in set(r.algorithm_name for r in all_results):
629     algo_results = [r for r in all_results if r.algorithm_name == algo]
630     best = max(algo_results, key=lambda r: r.metrics.ari if r.metrics.ari else 0)
631     print(f"\n{algo}: {best.dimension_label}(ARI={best.metrics.ari:.4f})")
632
633 # Runtime improvement
634 print("\n Runtime_Improvement_with_Dimensionality_Reduction:")
635 for algo in set(r.algorithm_name for r in all_results):
636     algo_results = [r for r in all_results if r.algorithm_name == algo]
637     original = next((r for r in algo_results if "Original" in r.dimension_label), None)
638     pca_best = min([r for r in algo_results if "PCA" in r.dimension_label],
639                   key=lambda r: r.runtime_seconds)
640     if original and pca_best:

```

```

638         speedup = original.runtime_seconds / pca_best.runtime_seconds
639         print(f"\n{algo.split('(')[0].strip()}: {speedup:.2f}x faster with {pca_best.
640             dimension_label}")
640
641     print("""
642     Key Findings:
643
644     1. Curse of Dimensionality:
645         - High-dimensional data leads to distance concentration
646         - Points become equidistant, making clustering harder
647         - Dimensionality reduction can recover meaningful structure
648
649     2. PCA Benefits:
650         - Reduces noise by removing low-variance components
651         - Improves computational efficiency
652         - May improve clustering quality if signal is in top components
653
654     3. PCA Risks:
655         - May discard discriminative information in lower components
656         - Not all cluster structure aligns with principal components
657         - Trade-off between variance explained and cluster separability
658
659     4. Algorithm-Specific Observations:
660         - K-Means and GMM benefit from noise reduction via PCA
661         - Spectral Clustering may be robust to dimensionality with proper kernel
662         - DBSCAN requires density parameter re-tuning after PCA
663     """
664
665     print(f"\n    All results saved to: {output_dir}/")
666     print("    Experiment 3 completed successfully!")
667
668     return {
669         'results': all_results,
670         'summary': df_summary,
671         'pivot_ari': pivot_ari,
672         'pivot_runtime': pivot_runtime,
673     }

```

A.8. src/metrics.py

```

1 """
2 Evaluation metrics for clustering quality assessment.
3 """
4
5 import numpy as np
6 from sklearn.metrics import (
7     silhouette_score,
8     calinski_harabasz_score,
9     davies_bouldin_score,
10    adjusted_rand_score,
11    normalized_mutual_info_score,
12)
13 from dataclasses import dataclass
14 from typing import Optional, Dict
15
16
17 @dataclass
18 class ClusteringMetrics:
19     """Container for all clustering evaluation metrics."""
20     # Internal metrics (no ground truth required)
21     silhouette: Optional[float] = None
22     calinski_harabasz: Optional[float] = None
23     davies_bouldin: Optional[float] = None
24

```

```

25     # External metrics (requires ground truth)
26     ari: Optional[float] = None    # Adjusted Rand Index
27     nmi: Optional[float] = None   # Normalized Mutual Information
28
29     def to_dict(self) -> Dict[str, Optional[float]]:
30         """Convert metrics to dictionary."""
31         return {
32             "Silhouette": self.silhouette,
33             "Calinski-Harabasz": self.calinski_harabasz,
34             "Davies-Bouldin": self.davies_bouldin,
35             "ARI": self.ari,
36             "NMI": self.nmi,
37         }
38
39     def __str__(self) -> str:
40         """Pretty print metrics."""
41         lines = []
42         if self.silhouette is not None:
43             lines.append(f"Silhouette Score: {self.silhouette:.4f}")
44         if self.calinski_harabasz is not None:
45             lines.append(f"Calinski-Harabasz Index: {self.calinski_harabasz:.4f}")
46         if self.davies_bouldin is not None:
47             lines.append(f"Davies-Bouldin Index: {self.davies_bouldin:.4f}")
48         if self.ari is not None:
49             lines.append(f"Adjusted Rand Index: {self.ari:.4f}")
50         if self.nmi is not None:
51             lines.append(f"Normalized Mutual Information: {self.nmi:.4f}")
52         return "\n".join(lines)
53
54
55     def compute_internal_metrics(
56         X: np.ndarray,
57         labels: np.ndarray,
58     ) -> ClusteringMetrics:
59         """
60             Compute internal clustering metrics (no ground truth required).
61
62             Args:
63                 X: Feature matrix
64                 labels: Cluster assignments
65
66             Returns:
67                 ClusteringMetrics with internal metrics populated
68         """
69         metrics = ClusteringMetrics()
70
71         # Check if we have valid clustering (at least 2 clusters, not all noise)
72         unique_labels = set(labels)
73         n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)
74
75         if n_clusters < 2:
76             # Cannot compute metrics with less than 2 clusters
77             return metrics
78
79         # Filter out noise points for metric computation
80         mask = labels != -1
81         if mask.sum() < 2:
82             return metrics
83
84         X_valid = X[mask]
85         labels_valid = labels[mask]
86
87         # Silhouette Score: [-1, 1], higher is better
88         try:
89             metrics.silhouette = silhouette_score(X_valid, labels_valid)

```

```

90     except Exception:
91         pass
92
93     # Calinski-Harabasz Index: higher is better (no upper bound)
94     try:
95         metrics.calinski_harabasz = calinski_harabasz_score(X_valid, labels_valid)
96     except Exception:
97         pass
98
99     # Davies-Bouldin Index: lower is better (minimum 0)
100    try:
101        metrics.davies_bouldin = davies_bouldin_score(X_valid, labels_valid)
102    except Exception:
103        pass
104
105    return metrics
106
107
108 def compute_external_metrics(
109     labels_true: np.ndarray,
110     labels_pred: np.ndarray,
111 ) -> ClusteringMetrics:
112     """
113     Compute_external_clustering_metrics_(requires_ground_truth).
114
115     Args:
116         labels_true: Ground truth labels
117         labels_pred: Predicted cluster assignments
118
119     Returns:
120         ClusteringMetrics_with_external_metrics_populated
121     """
122     metrics = ClusteringMetrics()
123
124     # Filter out noise points (-1 labels) for fair comparison
125     mask = labels_pred != -1
126     if mask.sum() < 2:
127         return metrics
128
129     labels_true_valid = labels_true[mask]
130     labels_pred_valid = labels_pred[mask]
131
132     # Adjusted Rand Index: [-1, 1], 1 is perfect, 0 is random
133     try:
134         metrics.ari = adjusted_rand_score(labels_true_valid, labels_pred_valid)
135     except Exception:
136         pass
137
138     # Normalized Mutual Information: [0, 1], 1 is perfect
139     try:
140         metrics.nmi = normalized_mutual_info_score(
141             labels_true_valid, labels_pred_valid, average_method="arithmetic"
142         )
143     except Exception:
144         pass
145
146     return metrics
147
148
149 def compute_all_metrics(
150     X: np.ndarray,
151     labels_pred: np.ndarray,
152     labels_true: Optional[np.ndarray] = None,
153 ) -> ClusteringMetrics:
154     """

```

```

155     Compute_all_applicable_metrics.
156
157     Args:
158         X: Feature_matrix
159         labels_pred: Predicted_cluster_assignments
160         labels_true: Ground_truth_labels (optional)
161
162     Returns:
163         ClusteringMetrics with all applicable metrics populated
164         """
165         # Compute internal metrics
166         internal = compute_internal_metrics(X, labels_pred)
167
168         # Compute external metrics if ground truth is available
169         if labels_true is not None:
170             external = compute_external_metrics(labels_true, labels_pred)
171             internal.ari = external.ari
172             internal.nmi = external.nmi
173
174     return internal
175
176
177 def summarize_metrics(
178     results: list[tuple[str, ClusteringMetrics]],
179 ) -> str:
180     """
181     Create a summary table of metrics for multiple algorithms.
182
183     Args:
184         results: List of (algorithm_name, metrics) tuples
185
186     Returns:
187         Formatted_table_string
188         """
189     import pandas as pd
190
191     data = []
192     for name, metrics in results:
193         row = {"Algorithm": name}
194         row.update(metrics.to_dict())
195         data.append(row)
196
197     df = pd.DataFrame(data)
198     return df.to_string(index=False)
199
200
201 def rank_algorithms(
202     results: list[tuple[str, ClusteringMetrics]],
203     metric: str = "ari",
204 ) -> list[tuple[str, float]]:
205     """
206     Rank algorithms by a specific metric.
207
208     Args:
209         results: List of (algorithm_name, metrics) tuples
210         metric: Metric to rank by ('silhouette', 'ari', 'nmi', etc.)
211
212     Returns:
213         Sorted_list_of_(algorithm_name, metric_value) tuples
214         """
215     rankings = []
216     for name, metrics in results:
217         value = getattr(metrics, metric, None)
218         if value is not None:
219             rankings.append((name, value))

```

```

220     # Higher is better for most metrics, except Davies-Bouldin
221     reverse = metric != "davies_bouldin"
222     rankings.sort(key=lambda x: x[1], reverse=reverse)
223
224     return rankings

```

A.9. src/visualization.py

```

1 """
2 Visualization_utilities_for_clustering_experiments.
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7 import seaborn as sns
8 from typing import Optional, List, Tuple, Dict
9 from pathlib import Path
10
11 # Set style for publication-quality figures
12 plt.style.use('seaborn-v0_8-whitegrid')
13 sns.set_palette("husl")
14
15
16 def plot_clustering_comparison(
17     X: np.ndarray,
18     labels_true: np.ndarray,
19     results: List[Tuple[str, np.ndarray]],
20     dataset_name: str,
21     save_path: Optional[Path] = None,
22     figsize: Tuple[int, int] = None,
23 ) -> plt.Figure:
24     """
25     Create a comparison plot of clustering results across algorithms.
26
27     Args:
28         X: 2D feature matrix (n_samples, n_features)
29         labels_true: Ground truth labels
30         results: List of (algorithm_name, predicted_labels) tuples
31         dataset_name: Name of the dataset for title
32         save_path: Path to save figure (optional)
33         figsize: Figure size (width, height)
34
35     Returns:
36         matplotlib Figure object
37     """
38     n_algorithms = len(results)
39     n_cols = min(3, n_algorithms + 1)  # +1 for ground truth
40     n_rows = (n_algorithms + 1 + n_cols - 1) // n_cols
41
42     if figsize is None:
43         figsize = (5 * n_cols, 4 * n_rows)
44
45     fig, axes = plt.subplots(n_rows, n_cols, figsize=figsize)
46     axes = np.atleast_2d(axes)
47     axes = axes.flatten()
48
49     # Color palette
50     colors = plt.cm.tab10(np.linspace(0, 1, 10))
51
52     def plot_scatter(ax, X, labels, title):
53         unique_labels = np.unique(labels)

```

```

54     for label in unique_labels:
55         mask = labels == label
56         if label == -1:
57             # Noise points in DBSCAN
58             ax.scatter(X[mask, 0], X[mask, 1], c='gray', marker='x',
59                         s=20, alpha=0.5, label='Noise')
60         else:
61             color = colors[label % len(colors)]
62             ax.scatter(X[mask, 0], X[mask, 1], c=[color],
63                         s=30, alpha=0.7, edgecolors='white', linewidth=0.5)
64         ax.set_title(title, fontsize=12, fontweight='bold')
65         ax.set_xlabel('Feature_1')
66         ax.set_ylabel('Feature_2')
67
68     # Plot ground truth
69     plot_scatter(axes[0], X, labels_true, f"{dataset_name}\n(Ground_Truth)")
70
71     # Plot each algorithm's results
72     for idx, (name, labels) in enumerate(results, start=1):
73         plot_scatter(axes[idx], X, labels, name)
74
75     # Hide unused subplots
76     for idx in range(len(results) + 1, len(axes)):
77         axes[idx].set_visible(False)
78
79     plt.tight_layout()
80
81     if save_path:
82         save_path.parent.mkdir(parents=True, exist_ok=True)
83         fig.savefig(save_path, dpi=150, bbox_inches='tight')
84         print(f"Saved_figure_to_{save_path}")
85
86     return fig
87
88
89 def plot_metrics_heatmap(
90     metrics_data: Dict[str, Dict[str, float]],
91     dataset_names: List[str],
92     algorithm_names: List[str],
93     metric_name: str,
94     save_path: Optional[Path] = None,
95     figsize: Tuple[int, int] = (10, 6),
96 ) -> plt.Figure:
97     """
98     Create a heatmap of metric values across datasets and algorithms.
99
100    Args:
101        metrics_data: Nested dict {dataset: {algorithm: value}}
102        dataset_names: List of dataset names
103        algorithm_names: List of algorithm names
104        metric_name: Name of the metric for title
105        save_path: Path to save figure
106        figsize: Figure size
107
108    Returns:
109        matplotlib Figure object
110    """
111    # Create matrix
112    matrix = np.zeros((len(dataset_names), len(algorithm_names)))
113    for i, dataset in enumerate(dataset_names):
114        for j, algo in enumerate(algorithm_names):
115            value = metrics_data.get(dataset, {}).get(algo, np.nan)
116            matrix[i, j] = value if value is not None else np.nan
117
118    fig, ax = plt.subplots(figsize=figsize)

```

```

119     # Determine color map based on metric (DB is lower-is-better)
120     cmap = 'RdYlGn' if metric_name != 'Davies-Bouldin' else 'RdYlGn_r'
121
122     im = ax.imshow(matrix, cmap=cmap, aspect='auto')
123
124     # Add colorbar
125     cbar = ax.figure.colorbar(im, ax=ax)
126     cbar.ax.set_ylabel(metric_name, rotation=-90, va="bottom")
127
128     # Set ticks
129     ax.set_xticks(np.arange(len(algorithm_names)))
130     ax.set_yticks(np.arange(len(dataset_names)))
131     ax.set_xticklabels(algorithm_names, rotation=45, ha='right')
132     ax.set_yticklabels(dataset_names)
133
134     # Add text annotations
135     for i in range(len(dataset_names)):
136         for j in range(len(algorithm_names)):
137             value = matrix[i, j]
138             if not np.isnan(value):
139                 text = ax.text(j, i, f'{value:.3f}',
140                                ha='center', va='center',
141                                color='black' if 0.3 < (value - np.nanmin(matrix)) /
142                                              (np.nanmax(matrix) - np.nanmin(matrix) + 1e-10) < 0.7
143                                else 'white',
144                                fontsize=9)
145
146     ax.set_title(f'{metric_name}_Comparison', fontsize=14, fontweight='bold')
147     plt.tight_layout()
148
149     if save_path:
150         save_path.parent.mkdir(parents=True, exist_ok=True)
151         fig.savefig(save_path, dpi=150, bbox_inches='tight')
152         print(f"Saved figure to {save_path}")
153
154     return fig
155
156
157
158 def plot_metrics_table(
159     data: List[Dict],
160     title: str = "Clustering_Results",
161     save_path: Optional[Path] = None,
162     figsize: Tuple[int, int] = (12, 8),
163 ) -> plt.Figure:
164     """
165     Create a visual table of metrics.
166
167     Args:
168         data: List of dicts with metric data
169         title: Table title
170         save_path: Path to save figure
171         figsize: Figure size
172
173     Returns:
174         matplotlib Figure object
175     """
176     import pandas as pd
177
178     df = pd.DataFrame(data)
179
180     fig, ax = plt.subplots(figsize=figsize)
181     ax.axis('tight')
182     ax.axis('off')
183

```

```

184     table = ax.table(
185         cellText=df.values,
186         colLabels=df.columns,
187         cellLoc='center',
188         loc='center',
189     )
190     table.auto_set_font_size(False)
191     table.set_fontsize(9)
192     table.scale(1.2, 1.5)
193
194     # Style header
195     for i, key in enumerate(df.columns):
196         table[(0, i)].set_facecolor('#4472C4')
197         table[(0, i)].set_text_props(color='white', fontweight='bold')
198
199     ax.set_title(title, fontsize=14, fontweight='bold', pad=20)
200
201     if save_path:
202         save_path.parent.mkdir(parents=True, exist_ok=True)
203         fig.savefig(save_path, dpi=150, bbox_inches='tight')
204         print(f"Saved figure to {save_path}")
205
206     return fig
207
208
209 def plot_all_datasets_grid(
210     datasets_results: Dict[str, Tuple[np.ndarray, np.ndarray, List[Tuple[str, np.ndarray]]]],
211     save_path: Optional[Path] = None,
212     figsize: Tuple[int, int] = (20, 16),
213 ) -> plt.Figure:
214     """
215     Create a comprehensive grid showing all datasets and all algorithms.
216
217     Args:
218         datasets_results: Dict mapping dataset name to (X, y_true, [(algo_name, labels), ...])
219         save_path: Path to save figure
220         figsize: Figure size
221
222     Returns:
223         matplotlib Figure object
224     """
225     n_datasets = len(datasets_results)
226     dataset_names = list(datasets_results.keys())
227
228     # Get algorithm names from first dataset
229     first_data = list(datasets_results.values())[0]
230     algorithm_names = ['Ground Truth'] + [name for name, _ in first_data[2]]
231     n_algorithms = len(algorithm_names)
232
233     fig, axes = plt.subplots(n_datasets, n_algorithms, figsize=figsize)
234
235     colors = plt.cm.tab10(np.linspace(0, 1, 10))
236
237     for row_idx, (dataset_name, (X, y_true, results)) in enumerate(datasets_results.items()):
238         # Plot ground truth in first column
239         ax = axes[row_idx, 0]
240         for label in np.unique(y_true):
241             mask = y_true == label
242             ax.scatter(X[mask, 0], X[mask, 1], c=[colors[label % len(colors)]],
243                         s=15, alpha=0.7)
244         if row_idx == 0:
245             ax.set_title('Ground Truth', fontsize=10, fontweight='bold')

```

```

246     ax.set_ylabel(dataset_name, fontsize=10, fontweight='bold')
247     ax.set_xticks([])
248     ax.set_yticks([])
249
250     # Plot each algorithm
251     for col_idx, (algo_name, labels) in enumerate(results, start=1):
252         ax = axes[row_idx, col_idx]
253         unique_labels = np.unique(labels)
254         for label in unique_labels:
255             mask = labels == label
256             if label == -1:
257                 ax.scatter(X[mask, 0], X[mask, 1], c='gray', marker='x',
258                             s=10, alpha=0.5)
259             else:
260                 ax.scatter(X[mask, 0], X[mask, 1], c=[colors[label % len(colors)]],
261                             s=15, alpha=0.7)
262         if row_idx == 0:
263             ax.set_title(algo_name, fontsize=10, fontweight='bold')
264             ax.set_xticks([])
265             ax.set_yticks([])
266
267     plt.suptitle('Clustering_Algorithm_Comparison_Across_Datasets',
268                  fontsize=14, fontweight='bold', y=1.02)
269     plt.tight_layout()
270
271     if save_path:
272         save_path.parent.mkdir(parents=True, exist_ok=True)
273         fig.savefig(save_path, dpi=150, bbox_inches='tight')
274         print(f"Saved figure to {save_path}")
275
276     return fig
277
278
279 def create_summary_bar_chart(
280     metrics_data: Dict[str, Dict[str, float]],
281     metric_name: str,
282     save_path: Optional[Path] = None,
283     figsize: Tuple[int, int] = (12, 6),
284 ) -> plt.Figure:
285     """
286     Create a grouped bar chart comparing algorithms across datasets.
287
288     Args:
289         metrics_data: Dict[dataset: {algorithm: value}]
290         metric_name: Name of metric
291         save_path: Path to save figure
292         figsize: Figure size
293
294     Returns:
295         matplotlib_Figure_object
296     """
297     import pandas as pd
298
299     # Convert to DataFrame
300     df = pd.DataFrame(metrics_data).T
301
302     fig, ax = plt.subplots(figsize=figsize)
303
304     df.plot(kind='bar', ax=ax, width=0.8)
305
306     ax.set_xlabel('Dataset', fontsize=12)
307     ax.set_ylabel(metric_name, fontsize=12)
308     ax.set_title(f'{metric_name} by Algorithm and Dataset', fontsize=14, fontweight='bold')
309     ax.legend(title='Algorithm', bbox_to_anchor=(1.05, 1), loc='upper_left')

```

```
310     ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')
311
312     plt.tight_layout()
313
314     if save_path:
315         save_path.parent.mkdir(parents=True, exist_ok=True)
316         fig.savefig(save_path, dpi=150, bbox_inches='tight')
317         print(f"Saved figure to {save_path}")
318
319     return fig
```