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.

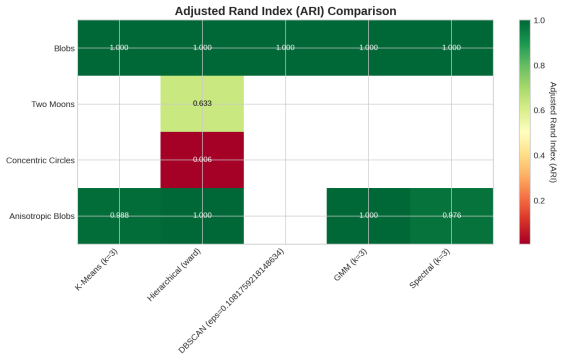


Figure 2. Heatmap of Adjusted Rand Index (ARI) scores. Green indicates high performance, while red indicates failure to capture the ground truth structure.

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 ≈ 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.

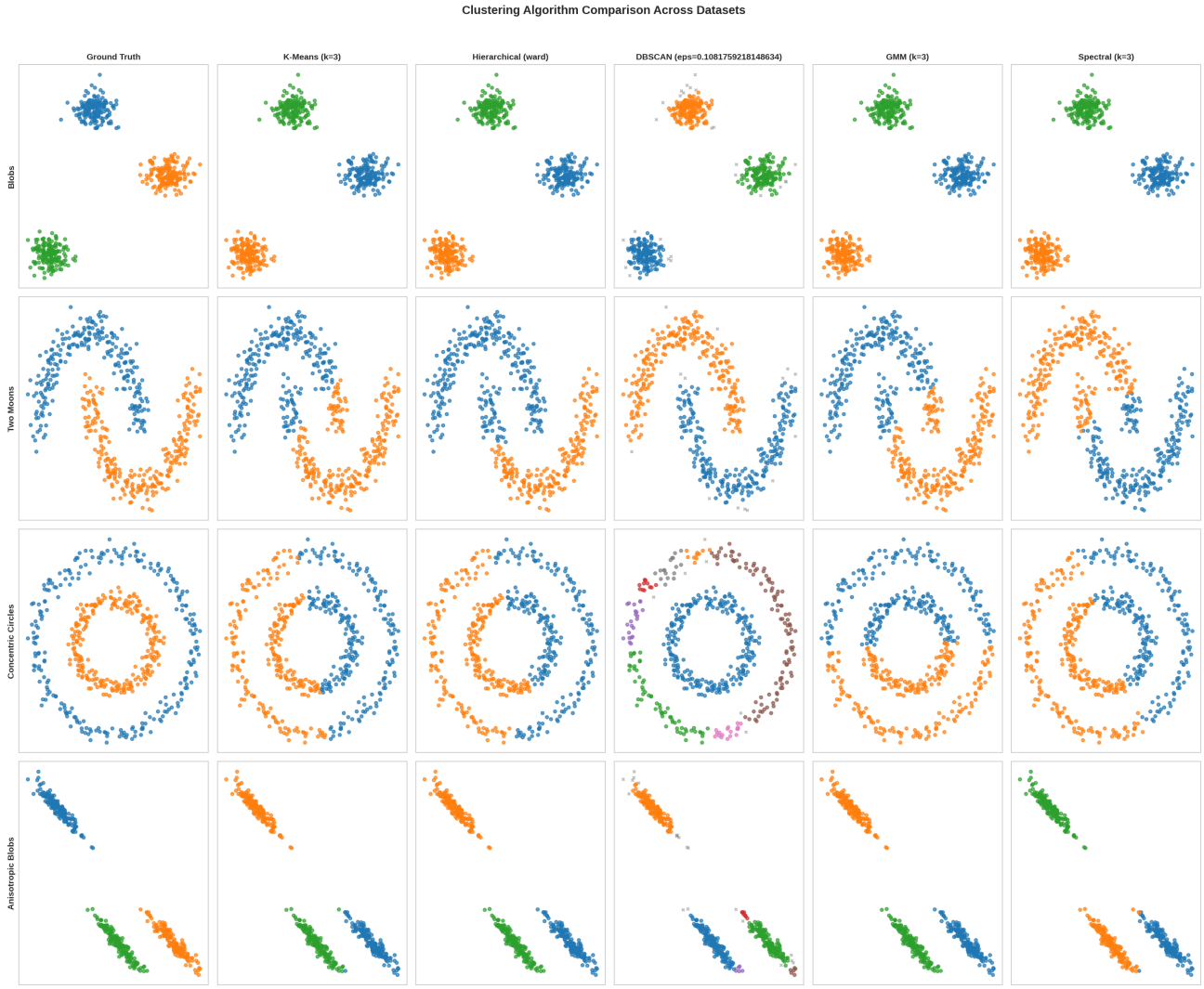


Figure 1. Clustering results on synthetic datasets. Rows correspond to datasets (Blobs, Moons, Circles, Anisotropic), columns correspond to algorithms. DBSCAN successfully captures non-convex structures, while GMM adapts to anisotropic clusters via covariance modeling.

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:

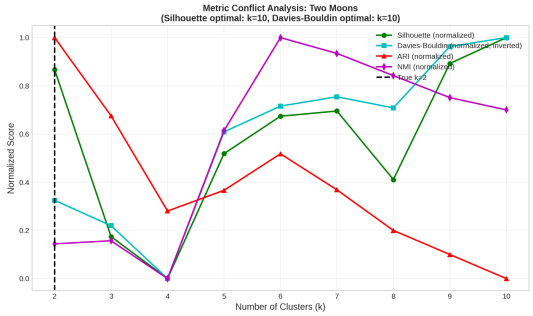
1. **Metric Conflict:** The disagreement between internal validation metrics (e.g., Silhouette Coefficient) and external ground truth metrics (e.g., ARI) when the data violates the algorithm's geometric assumptions.
2. **Parameter Sensitivity:** The impact of hyper-parameters (ϵ and $min\_samples$) on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

3.2. K-Means: The Pitfall of Internal Metrics

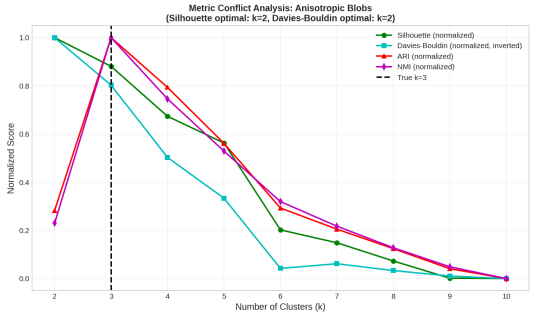
We performed a sweep of the cluster number $k \in [2, 10]$ for K-Means. Figure 3 illustrates the conflict between internal and external evaluations.

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**

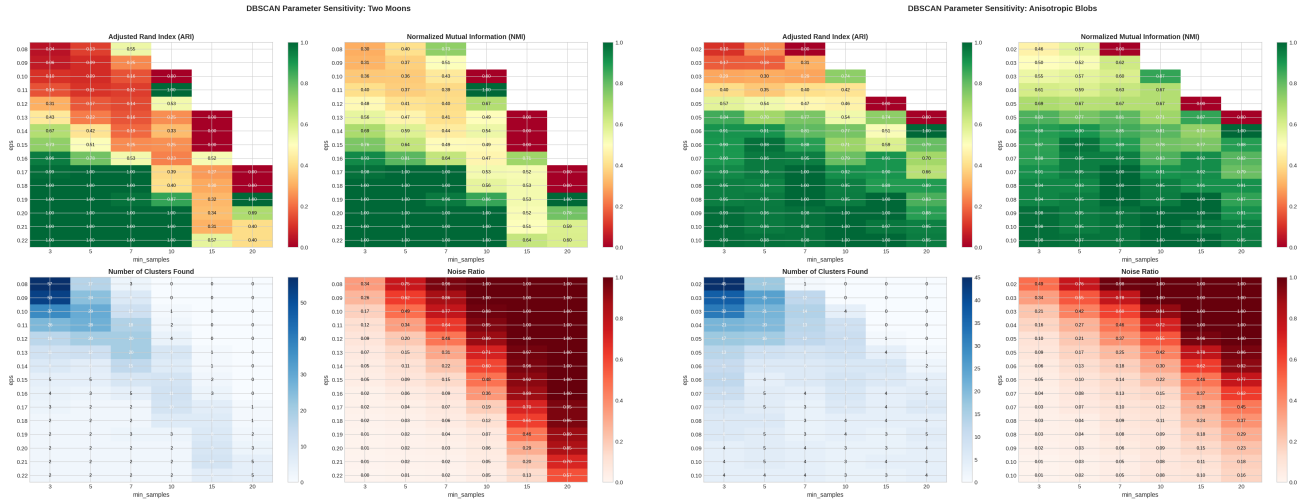
3.3. DBSCAN: Parameter Landscape Analysis

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 ≈ 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.



(a) Two Moons (b) Anisotropic Blobs

Figure 4. DBSCAN sensitivity to $(\varepsilon, \text{minPts})$ 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.

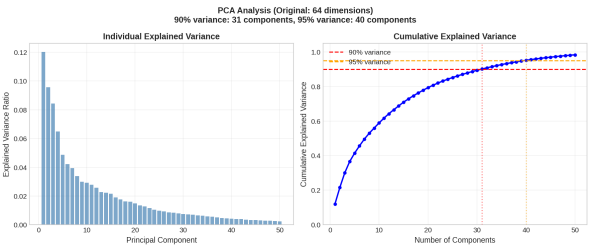


Figure 5. PCA Explained Variance Analysis. 31 components are required to retain 90% of the original information.

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 ∼56s (Original) to ∼0.7s (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 → 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,

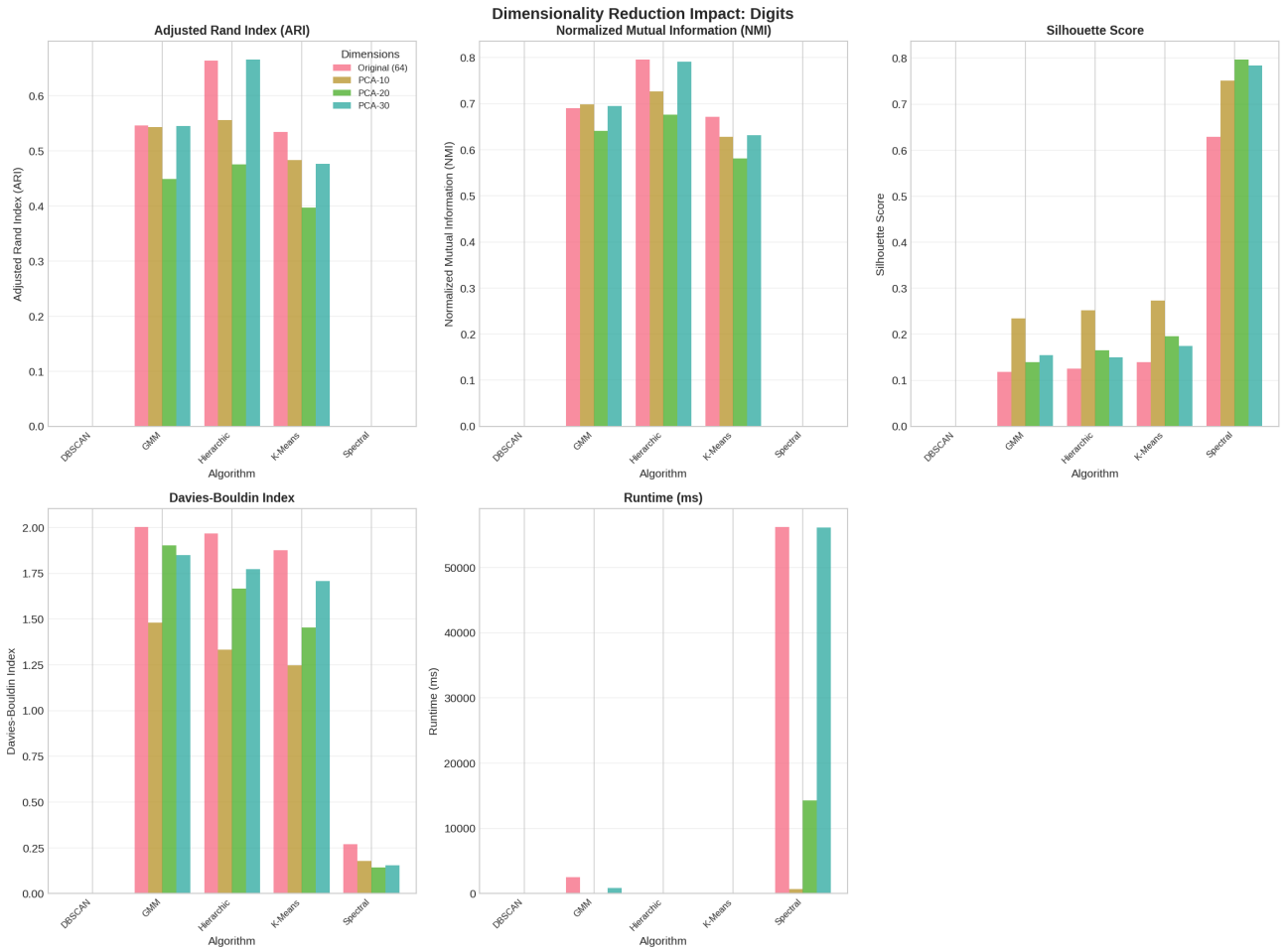


Figure 6. Impact of Dimensionality Reduction. (Top-Left) ARI comparison showing distinct algorithm behaviors. (Bottom-Center) Runtime comparison highlighting the massive speedup for Spectral Clustering and GMM after PCA.

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.

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**

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.

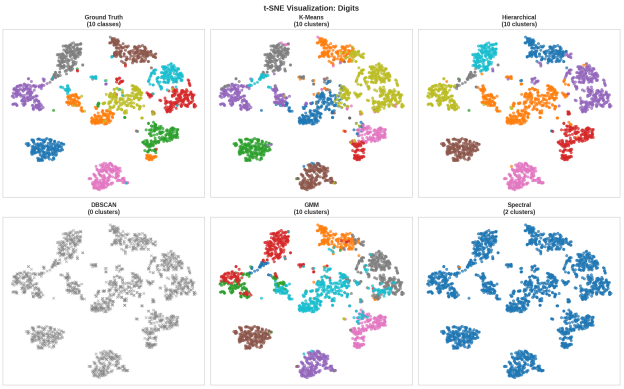


Figure 7. t-SNE Visualization of the Digits dataset comparing Ground Truth with algorithm predictions.

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

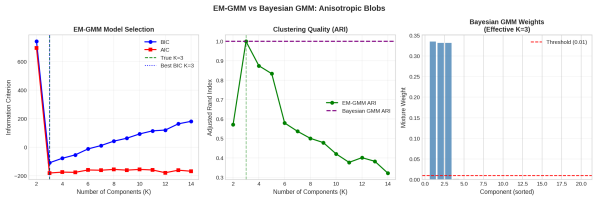


Figure 8. Automatic pruning of surplus clusters. Although initialized with $K = 20$, the Bayesian GMM (right panel) suppresses the weights of 17 components to near-zero, recovering the true $K = 3$ structure of the Anisotropic Blobs.

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.

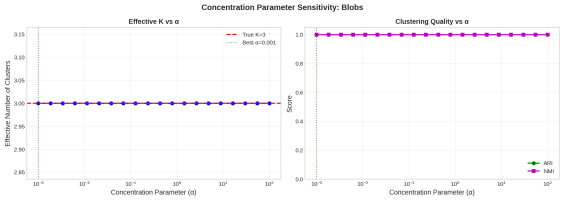


Figure 9. Sensitivity analysis of the concentration parameter α. The effective number of clusters (blue line) remains stable at the ground truth ($K = 3$) for a wide range of α, diverging only at extreme values.

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.

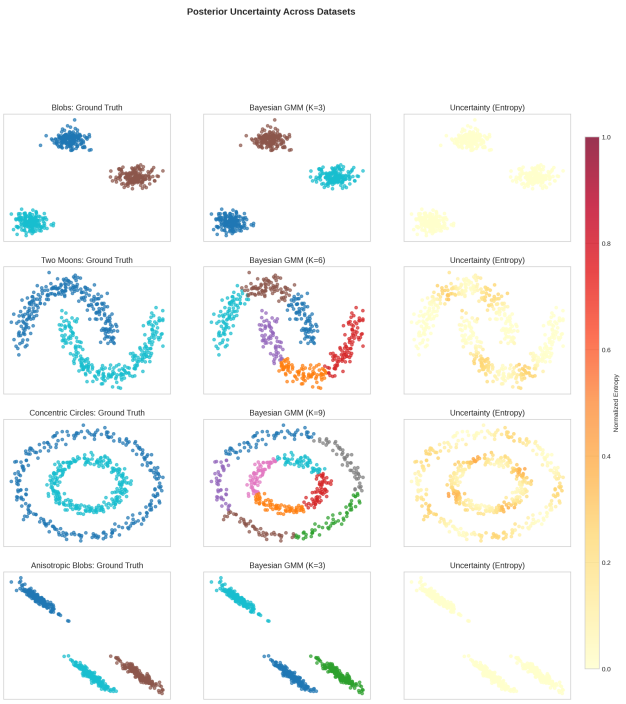


Figure 10. Posterior Uncertainty Visualization. The rightmost column shows the entropy of cluster assignments. Points at the boundaries of overlapping clusters (e.g., in Two Moons) exhibit higher uncertainty (red), providing valuable interpretability.

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`

```
"""
NYCU Pattern Recognition HW3 - Clustering Algorithm Comparison

This script implements Experiment 1: Geometric Structure and Algorithm Assumptions
Evaluates how different clustering algorithms perform under varying cluster geometries.
"""

import numpy as np
import pandas as pd
from pathlib import Path
from typing import Dict, List, Tuple
import warnings
warnings.filterwarnings('ignore')

from src.datasets import (
    get_synthetic_datasets,
    standardize_data,
    ClusteringDataset,
)
from src.clustering import (
    get_main_algorithms,
    tune_dbscan_eps,
    ClusteringResult,
    DBSCANClustering,
)
from src.metrics import compute_all_metrics, ClusteringMetrics
from src.visualization import (
    plot_clustering_comparison,
    plot_all_datasets_grid,
    plot_metrics_heatmap,
    create_summary_bar_chart,
)


# Configuration
RANDOM_STATE = 42
OUTPUT_DIR = Path("outputs/experiment1")


def run_experiment1():
    """
    Experiment 1: Geometric Structure and Algorithm Assumptions

    Evaluates how different algorithms perform under varying cluster geometries
    using synthetic datasets.
    """
    print("=" * 70)
    print("Experiment 1: Geometric Structure and Algorithm Assumptions")
    print("=" * 70)

    # Create output directory
    OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

    # Generate synthetic datasets
    print("\n[1/4] Generating synthetic datasets...")
    datasets = get_synthetic_datasets(random_state=RANDOM_STATE)

    for ds in datasets:
        print(f"  - {ds.name}: {ds.X.shape[0]} samples, {ds.n_clusters} clusters")
        print(f"    {ds.description}")

```

```python
    # Store results for visualization
    all_results: Dict[str, Tuple[np.ndarray, np.ndarray, List[Tuple[str, np.ndarray]]]] = {}
    all_metrics: Dict[str, Dict[str, ClusteringMetrics]] = {}

    print("\n[2/4] Running clustering algorithms on each dataset...")

    for dataset in datasets:
        print(f"\n{'  ' * 50}")
        print(f"Dataset: {dataset.name}")
        print(f"{'  ' * 50}")

        # Standardize features (z-score normalization)
        X_scaled, _ = standardize_data(dataset.X)

        # Get algorithms with correct number of clusters
        algorithms = get_main_algorithms(
            n_clusters=dataset.n_clusters,
            random_state=RANDOM_STATE
        )

        # Tune DBSCAN eps for this specific dataset
        # Replace the default DBSCAN with tuned version
        tuned_eps = tune_dbscan_eps(X_scaled, dataset.n_clusters)
        for i, algo in enumerate(algorithms):
            if isinstance(algo, DBSCANClustering):
                algorithms[i] = DBSCANClustering(eps=tuned_eps, min_samples=5)
                print(f"  Tuned DBSCAN eps: {tuned_eps:.3f}")

        dataset_results = []
        dataset_metrics = {}

        for algo in algorithms:
            # Fit and predict
            result = algo.fit_predict(X_scaled)

            # Compute metrics
            metrics = compute_all_metrics(
                X_scaled,
                result.labels,
                labels_true=dataset.y
            )

            dataset_results.append((result.algorithm_name, result.labels))
            dataset_metrics[result.algorithm_name] = metrics

            print(f"\n  {result.algorithm_name}:")
            print(f"    Clusters found: {result.n_clusters_found}")
            print(metrics)

        # Store for visualization
        all_results[dataset.name] = (X_scaled, dataset.y, dataset_results)
        all_metrics[dataset.name] = dataset_metrics

    # Generate visualizations
    print("\n[3/4] Generating visualizations...")

    # Individual dataset comparison plots
    for dataset_name, (X, y_true, results) in all_results.items():
        fig = plot_clustering_comparison(
            X=X,
            labels_true=y_true,
            results=results,
            dataset_name=dataset_name,
```

```
            save_path=OUTPUT_DIR / f"{dataset_name.lower().replace(' ','_')}_comparison.png"
        )
        plt.close(fig)

    # All datasets grid
    fig = plot_all_datasets_grid(
        datasets_results=all_results,
        save_path=OUTPUT_DIR / "all_datasets_grid.png"
    )
    plt.close(fig)

    # Metrics heatmaps
    metric_names = ['silhouette', 'ari', 'nmi', 'davies_bouldin']
    display_names = {
        'silhouette': 'Silhouette Score',
        'ari': 'Adjusted Rand Index (ARI)',
        'nmi': 'Normalized Mutual Information (NMI)',
        'davies_bouldin': 'Davies-Bouldin Index'
    }

    for metric in metric_names:
        metrics_data = {}
        for dataset_name, algo_metrics in all_metrics.items():
            metrics_data[dataset_name] = {}
            for algo_name, m in algo_metrics.items():
                value = getattr(m, metric, None)
                metrics_data[dataset_name][algo_name] = value

        dataset_names = list(metrics_data.keys())
        algorithm_names = list(next(iter(metrics_data.values())).keys())

        fig = plot_metrics_heatmap(
            metrics_data=metrics_data,
            dataset_names=dataset_names,
            algorithm_names=algorithm_names,
            metric_name=display_names[metric],
            save_path=OUTPUT_DIR / f"heatmap_{metric}.png"
        )
        plt.close(fig)

    # Create summary table
    print("\n[4/4] Creating summary tables...")

    summary_data = []
    for dataset_name, algo_metrics in all_metrics.items():
        for algo_name, metrics in algo_metrics.items():
            row = {
                'Dataset': dataset_name,
                'Algorithm': algo_name,
                'Silhouette': f"{metrics.silhouette:.4f}" if metrics.silhouette else "N/A",
                'ARI': f"{metrics.ari:.4f}" if metrics.ari else "N/A",
                'NMI': f"{metrics.nmi:.4f}" if metrics.nmi else "N/A",
                'Davies-Bouldin': f"{metrics.davies_bouldin:.4f}" if metrics.davies_bouldin else "N/A",
            }
            summary_data.append(row)

    df_summary = pd.DataFrame(summary_data)

    # Save to CSV
    csv_path = OUTPUT_DIR / "experiment1_results.csv"
    df_summary.to_csv(csv_path, index=False)
    print(f"\nResults saved to: {csv_path}")
```

```
187
188     # Print summary table
189     print("\n" + "=" * 70)
190     print("EXPERIMENT 1 RESULTS SUMMARY")
191     print("=" * 70)
192     print(df_summary.to_string(index=False))
193
194     # Analysis and observations
195     print_analysis(all_metrics)
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":
```

```
        from src.bonus_bayesian import run_all_bonus_experiments
        run_all_bonus_experiments()
    elif experiment == "all":
        print("\n" + "=" * 70)
        print("Running All Experiments (including Bonus)")
        print("=" * 70 + "\n")

        run_experiment1()

        print("\n" + "=" * 70 + "\n")

        from src.experiment2 import run_experiment2
        run_experiment2()

        print("\n" + "=" * 70 + "\n")

        from src.experiment3 import run_experiment3
        run_experiment3()

        print("\n" + "=" * 70 + "\n")

        from src.bonus_bayesian import run_all_bonus_experiments
        run_all_bonus_experiments()
    elif experiment == "main":
        # Run only main experiments (1, 2, 3) without bonus
        print("\n" + "=" * 70)
        print("Running Main Experiments (1, 2, 3)")
        print("=" * 70 + "\n")

        run_experiment1()

        print("\n" + "=" * 70 + "\n")

        from src.experiment2 import run_experiment2
        run_experiment2()

        print("\n" + "=" * 70 + "\n")

        from src.experiment3 import run_experiment3
        run_experiment3()
    else:
        print(f"Unknown experiment: {experiment}")
        print("Usage: python main.py [1|2|3|bonus|main|all]")
        print("  1 or exp1: Run Experiment 1 (Geometric Structure)")
        print("  2 or exp2: Run Experiment 2 (Hyperparameter Sensitivity)")
        print("  3 or exp3: Run Experiment 3 (Dimensionality Reduction)")
        print("  bonus:     Run Bonus Experiments (Bayesian Clustering)")
        print("  main:      Run main experiments (1, 2, 3) without bonus")
        print("  all:       Run all experiments including bonus (default)")
        sys.exit(1)


if __name__ == "__main__":
    main()
```

A.2. src/__init__.py

```
# NYCU Pattern Recognition HW3 - Clustering Algorithms
# Package initialization

from . import datasets
from . import clustering
from . import metrics
from . import visualization
```

```
from . import experiment2
from . import experiment3
from . import bonus_bayesian
```

A.3. src/bonus_bayesian.py

```
"""
Bonus Experiment: Bayesian Clustering

This module demonstrates how Bayesian modeling:
- Handles uncertainty
- Controls model complexity
- Reduces reliance on manual selection of cluster number

Three sub-experiments:
1. Effective Number of Clusters: EM-GMM vs Bayesian GMM
2. Concentration Parameter Sensitivity:   sweep
3. Posterior Uncertainty Visualization: Entropy-based analysis
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass
from sklearn.mixture import GaussianMixture, BayesianGaussianMixture
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score
from scipy.stats import entropy

from .datasets import (
    ClusteringDataset,
    standardize_data,
    get_synthetic_datasets,
    generate_blobs,
)


@dataclass
class BayesianGMMResult:
    """Results from Bayesian GMM analysis."""
    weights: np.ndarray
    effective_k: int
    labels: np.ndarray
    responsibilities: np.ndarray
    ari: float
    nmi: float
    model: BayesianGaussianMixture


@dataclass
class EMGMMResult:
    """Results from EM-GMM analysis."""
    k: int
    labels: np.ndarray
    ari: float
    nmi: float
    bic: float
    aic: float
    model: GaussianMixture


def fit_em_gmm(
```

```
    X: np.ndarray,
    y_true: np.ndarray,
    n_components: int,
    random_state: int = 42,
) -> EMGMMResult:
    """
␣␣␣␣Fit␣EM-GMM␣with␣specified␣number␣of␣components.

␣␣␣␣Args:
␣␣␣␣␣␣␣␣X:␣Feature␣matrix
␣␣␣␣␣␣␣␣y_true:␣Ground␣truth␣labels
␣␣␣␣␣␣␣␣n_components:␣Number␣of␣Gaussian␣components
␣␣␣␣␣␣␣␣random_state:␣Random␣seed

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣EMGMMResult␣with␣fitted␣model␣and␣metrics
␣␣␣␣"""
    model = GaussianMixture(
        n_components=n_components,
        covariance_type='full',
        random_state=random_state,
        n_init=5,
    )
    labels = model.fit_predict(X)

    return EMGMMResult(
        k=n_components,
        labels=labels,
        ari=adjusted_rand_score(y_true, labels),
        nmi=normalized_mutual_info_score(y_true, labels),
        bic=model.bic(X),
        aic=model.aic(X),
        model=model,
    )


def fit_bayesian_gmm(
    X: np.ndarray,
    y_true: np.ndarray,
    n_components: int = 20,
    weight_concentration_prior: float = 0.01,
    weight_concentration_prior_type: str = 'dirichlet_process',
    weight_threshold: float = 0.01,
    random_state: int = 42,
) -> BayesianGMMResult:
    """
␣␣␣␣Fit␣Bayesian␣GMM␣with␣Dirichlet␣Process␣prior.

␣␣␣␣Args:
␣␣␣␣␣␣␣␣X:␣Feature␣matrix
␣␣␣␣␣␣␣␣y_true:␣Ground␣truth␣labels
␣␣␣␣␣␣␣␣n_components:␣Maximum␣number␣of␣components␣(Kmax)
␣␣␣␣␣␣␣␣weight_concentration_prior:␣Concentration␣parameter␣
␣␣␣␣␣␣␣weight_concentration_prior_type:␣'dirichlet_process'␣or␣'dirichlet_distribution'
␣␣␣␣␣␣␣␣weight_threshold:␣Threshold␣for␣counting␣effective␣clusters
␣␣␣␣␣␣␣␣random_state:␣Random␣seed

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣BayesianGMMResult␣with␣fitted␣model␣and␣metrics
␣␣␣␣"""
    model = BayesianGaussianMixture(
        n_components=n_components,
        covariance_type='full',
        weight_concentration_prior_type=weight_concentration_prior_type,
        weight_concentration_prior=weight_concentration_prior,
```

```
        random_state=random_state,
        n_init=3,
        max_iter=200,
    )

    labels = model.fit_predict(X)
    responsibilities = model.predict_proba(X)

    # Count effective clusters (weights above threshold)
    effective_k = np.sum(model.weights_ > weight_threshold)

    return BayesianGMMResult(
        weights=model.weights_,
        effective_k=effective_k,
        labels=labels,
        responsibilities=responsibilities,
        ari=adjusted_rand_score(y_true, labels),
        nmi=normalized_mutual_info_score(y_true, labels),
        model=model,
    )


def compute_point_entropy(responsibilities: np.ndarray) -> np.ndarray:
    """
    Compute entropy of posterior responsibilities for each data point.

    High entropy indicates uncertainty in cluster assignment.

    Args:
        responsibilities: Posterior probabilities (n_samples, n_components)

    Returns:
        Entropy values for each data point
    """
    # Add small epsilon to avoid log(0)
    eps = 1e-10
    probs = np.clip(responsibilities, eps, 1.0)

    # Compute entropy: -sum(p * log(p))
    point_entropy = -np.sum(probs * np.log(probs), axis=1)

    # Normalize by max possible entropy (uniform distribution)
    n_components = responsibilities.shape[1]
    max_entropy = np.log(n_components)

    return point_entropy / max_entropy  # Normalized to [0, 1]


# ========================================
# Bonus Experiment 1: Effective Number of Clusters
# ========================================

def run_bonus_experiment1(
    datasets: List[ClusteringDataset],
    k_range: range = range(2, 15),
    kmax_bayesian: int = 20,
    output_dir: Path = Path("outputs/bonus"),
    random_state: int = 42,
) -> Dict:
    """
    Bonus Experiment 1: Compare EM-GMM and Bayesian GMM on cluster number selection.

    Procedure:
    1. Fit EM-GMM with varying K
    2. Fit Bayesian GMM with large Kmax
```

```
    3. Compare learned mixture weights
    4. Count effective clusters
    """
    print("\n" + "=" * 70)
    print("Bonus Experiment 1: Effective Number of Clusters")
    print("=" * 70)

    output_dir.mkdir(parents=True, exist_ok=True)
    results = {}

    for dataset in datasets:
        print(f"\nDataset: {dataset.name} (True k={dataset.n_clusters})")
        print("-" * 50)

        X_scaled, _ = standardize_data(dataset.X)

        # ========================================
        # Part A: EM-GMM with varying K
        # ========================================
        print("  Fitting EM-GMM with varying K...")
        em_results = []
        for k in k_range:
            result = fit_em_gmm(X_scaled, dataset.y, k, random_state)
            em_results.append(result)
            print(f"    K={k}: ARI={result.ari:.4f}, BIC={result.bic:.1f}")

        # Find best K by BIC
        best_em = min(em_results, key=lambda r: r.bic)
        print(f"  EM-GMM best K (by BIC): {best_em.k}")

        # ========================================
        # Part B: Bayesian GMM
        # ========================================
        print(f"  Fitting Bayesian GMM (Kmax={kmax_bayesian})...")
        bayesian_result = fit_bayesian_gmm(
            X_scaled, dataset.y,
            n_components=kmax_bayesian,
            weight_concentration_prior=0.01,
            random_state=random_state,
        )
        print(f"  Bayesian GMM effective K: {bayesian_result.effective_k}")
        print(f"  Bayesian GMM ARI: {bayesian_result.ari:.4f}")

        results[dataset.name] = {
            'em_results': em_results,
            'bayesian_result': bayesian_result,
            'best_em': best_em,
            'true_k': dataset.n_clusters,
        }

        # ========================================
        # Visualizations
        # ========================================

        # Plot 1: EM-GMM model selection (BIC/AIC)
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        # BIC/AIC plot
        ax1 = axes[0]
        k_vals = [r.k for r in em_results]
        bic_vals = [r.bic for r in em_results]
        aic_vals = [r.aic for r in em_results]
        ax1.plot(k_vals, bic_vals, 'b-o', linewidth=2, label='BIC')
        ax1.plot(k_vals, aic_vals, 'r-s', linewidth=2, label='AIC')
        ax1.axvline(x=dataset.n_clusters, color='green', linestyle='--',
```

```
                   label=f'True K={dataset.n_clusters}')
        ax1.axvline(x=best_em.k, color='blue', linestyle=':',
                   label=f'Best BIC K={best_em.k}')
        ax1.set_xlabel('Number of Components (K)', fontsize=11)
        ax1.set_ylabel('Information Criterion', fontsize=11)
        ax1.set_title('EM-GMM Model Selection', fontsize=12, fontweight='bold')
        ax1.legend()
        ax1.grid(True, alpha=0.3)

        # ARI comparison
        ax2 = axes[1]
        ari_vals = [r.ari for r in em_results]
        ax2.plot(k_vals, ari_vals, 'g-o', linewidth=2, label='EM-GMM ARI')
        ax2.axhline(y=bayesian_result.ari, color='purple', linestyle='--',
                   linewidth=2, label=f'Bayesian GMM ARI')
        ax2.axvline(x=dataset.n_clusters, color='green', linestyle='--', alpha=0.5)
        ax2.set_xlabel('Number of Components (K)', fontsize=11)
        ax2.set_ylabel('Adjusted Rand Index', fontsize=11)
        ax2.set_title('Clustering Quality (ARI)', fontsize=12, fontweight='bold')
        ax2.legend()
        ax2.grid(True, alpha=0.3)

        # Bayesian GMM weights
        ax3 = axes[2]
        weights = bayesian_result.weights
        sorted_weights = np.sort(weights)[::-1]
        colors = ['steelblue' if w > 0.01 else 'lightgray' for w in sorted_weights]
        ax3.bar(range(1, len(weights) + 1), sorted_weights, color=colors, alpha=0.8)
        ax3.axhline(y=0.01, color='red', linestyle='--', label='Threshold (0.01)')
        ax3.set_xlabel('Component (sorted)', fontsize=11)
        ax3.set_ylabel('Mixture Weight', fontsize=11)
        ax3.set_title(f'Bayesian GMM Weights\n(Effective K={bayesian_result.effective_k})'
            ,
                     fontsize=12, fontweight='bold')
        ax3.legend()
        ax3.grid(True, alpha=0.3, axis='y')

        plt.suptitle(f'EM-GMM vs Bayesian GMM: {dataset.name}',
                    fontsize=14, fontweight='bold')
        plt.tight_layout()

        save_path = output_dir / f"bonus1_effective_k_{dataset.name.lower().replace(' ','
            _')}.png"
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"  Saved: {save_path}")
        plt.close(fig)

    # Create summary table
    summary_data = []
    for ds_name, data in results.items():
        row = {
            'Dataset': ds_name,
            'True K': data['true_k'],
            'EM-GMM Best K (BIC)': data['best_em'].k,
            'EM-GMM ARI': f"{data['best_em'].ari:.4f}",
            'Bayesian GMM Effective K': data['bayesian_result'].effective_k,
            'Bayesian GMM ARI': f"{data['bayesian_result'].ari:.4f}",
        }
        summary_data.append(row)

    df_summary = pd.DataFrame(summary_data)
    df_summary.to_csv(output_dir / "bonus1_summary.csv", index=False)

    print("\n" + "=" * 70)
    print("BONUS EXPERIMENT 1 SUMMARY")
```

```
    print("=" * 70)
    print(df_summary.to_string(index=False))

    return results


# ========================================
# Bonus Experiment 2: Concentration Parameter Sensitivity
# ========================================

def run_bonus_experiment2(
    datasets: List[ClusteringDataset],
    alpha_range: np.ndarray = None,
    kmax: int = 20,
    output_dir: Path = Path("outputs/bonus"),
    random_state: int = 42,
) -> Dict:
    """
    Bonus Experiment 2: Analyze effect of concentration parameter  .

    Procedure:
    1. Use Dirichlet Process-style Bayesian GMM
    2. Sweep concentration parameter
    3. Record effective number of clusters and ARI
    4. Plot Effective K vs  and ARI vs
    """
    print("\n" + "=" * 70)
    print("Bonus Experiment 2: Concentration Parameter Sensitivity")
    print("=" * 70)

    output_dir.mkdir(parents=True, exist_ok=True)

    if alpha_range is None:
        # Log-spaced alpha values from very small to large
        alpha_range = np.logspace(-3, 2, 20)  # 0.001 to 100

    results = {}

    for dataset in datasets:
        print(f"\nDataset: {dataset.name} (True k={dataset.n_clusters})")
        print("-" * 50)

        X_scaled, _ = standardize_data(dataset.X)

        effective_k_list = []
        ari_list = []
        nmi_list = []

        for alpha in alpha_range:
            result = fit_bayesian_gmm(
                X_scaled, dataset.y,
                n_components=kmax,
                weight_concentration_prior=alpha,
                random_state=random_state,
            )
            effective_k_list.append(result.effective_k)
            ari_list.append(result.ari)
            nmi_list.append(result.nmi)

            print(f"  ={alpha:.4f}: Effective K={result.effective_k}, ARI={result.ari:.4
                f}")

        results[dataset.name] = {
            'alpha_range': alpha_range,
            'effective_k': np.array(effective_k_list),
```

```
            'ari': np.array(ari_list),
            'nmi': np.array(nmi_list),
            'true_k': dataset.n_clusters,
        }

        # Visualization
        fig, axes = plt.subplots(1, 2, figsize=(14, 5))

        # Effective K vs
        ax1 = axes[0]
        ax1.semilogx(alpha_range, effective_k_list, 'b-o', linewidth=2, markersize=6)
        ax1.axhline(y=dataset.n_clusters, color='red', linestyle='--',
                    linewidth=2, label=f'True K={dataset.n_clusters}')
        ax1.set_xlabel('Concentration Parameter (  )', fontsize=12)
        ax1.set_ylabel('Effective Number of Clusters', fontsize=12)
        ax1.set_title('Effective K vs  ', fontsize=12, fontweight='bold')
        ax1.legend()
        ax1.grid(True, alpha=0.3)

        # ARI vs
        ax2 = axes[1]
        ax2.semilogx(alpha_range, ari_list, 'g-o', linewidth=2, markersize=6, label='ARI')
        ax2.semilogx(alpha_range, nmi_list, 'm-s', linewidth=2, markersize=6, label='NMI')
        ax2.set_xlabel('Concentration Parameter (  )', fontsize=12)
        ax2.set_ylabel('Score', fontsize=12)
        ax2.set_title('Clustering Quality vs  ', fontsize=12, fontweight='bold')
        ax2.legend()
        ax2.grid(True, alpha=0.3)
        ax2.set_ylim([0, 1.05])

        # Find best
        best_idx = np.argmax(ari_list)
        best_alpha = alpha_range[best_idx]
        ax1.axvline(x=best_alpha, color='green', linestyle=':', alpha=0.7,
                    label=f'Best  ={best_alpha:.3f}')
        ax2.axvline(x=best_alpha, color='green', linestyle=':', alpha=0.7)
        ax1.legend()

        plt.suptitle(f'Concentration Parameter Sensitivity: {dataset.name}',
                     fontsize=14, fontweight='bold')
        plt.tight_layout()

        save_path = output_dir / f"bonus2_alpha_sensitivity_{dataset.name.lower().replace
            (' ','_')}.png"
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"  Saved: {save_path}")
        plt.close(fig)

    # Summary analysis
    print("\n" + "=" * 70)
    print("ANALYSIS: Concentration Parameter Effects")
    print("=" * 70)
    print("""
    Key Observations:

1. Small    (< 0.1):
   - Encourages fewer clusters
   - Prior concentrates probability on few components
   - May underfit if true K is large

2. Large    (> 10):
   - Encourages more clusters
   - More uniform prior over components
   - Approaches EM-GMM behavior

```

```
3. Optimal    :
   - Dataset-dependent
   - Typically in range [0.01, 1] for moderate cluster numbers
   - Different from geometry-based parameters (e.g., DBSCAN eps)
""")

    return results


# ========================================
# Bonus Experiment 3: Posterior Uncertainty Visualization
# ========================================

def run_bonus_experiment3(
    datasets: List[ClusteringDataset] = None,
    output_dir: Path = Path("outputs/bonus"),
    random_state: int = 42,
) -> Dict:
    """
    Bonus Experiment 3: Visualize posterior uncertainty via entropy.

    Procedure:
    1. Compute posterior responsibilities
    2. Calculate entropy per data point
    3. Visualize entropy on 2D toy datasets

    High entropy indicates ambiguous/boundary points.
    """
    print("\n" + "=" * 70)
    print("Bonus Experiment 3: Posterior Uncertainty Visualization")
    print("=" * 70)

    output_dir.mkdir(parents=True, exist_ok=True)

    # Use 2D synthetic datasets for visualization
    if datasets is None:
        datasets = get_synthetic_datasets(random_state=random_state)

    results = {}

    for dataset in datasets:
        if dataset.X.shape[1] != 2:
            print(f"  Skipping {dataset.name} (not 2D)")
            continue

        print(f"\nDataset: {dataset.name}")
        print("-" * 50)

        X_scaled, _ = standardize_data(dataset.X)

        # Fit Bayesian GMM
        bayesian_result = fit_bayesian_gmm(
            X_scaled, dataset.y,
            n_components=15,
            weight_concentration_prior=0.1,
            random_state=random_state,
        )

        # Compute point-wise entropy
        point_entropy = compute_point_entropy(bayesian_result.responsibilities)

        results[dataset.name] = {
            'X': X_scaled,
            'y_true': dataset.y,
            'labels': bayesian_result.labels,
```

```
            'entropy': point_entropy,
            'responsibilities': bayesian_result.responsibilities,
            'effective_k': bayesian_result.effective_k,
        }

        print(f"  Effective K: {bayesian_result.effective_k}")
        print(f"  Mean entropy: {np.mean(point_entropy):.4f}")
        print(f"  High uncertainty points (entropy > 0.5): {np.sum(point_entropy > 0.5)}")

        # Visualization
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        # Ground truth
        ax1 = axes[0]
        scatter1 = ax1.scatter(X_scaled[:, 0], X_scaled[:, 1], c=dataset.y,
                               cmap='tab10', s=30, alpha=0.7)
        ax1.set_title(f'Ground Truth\n({dataset.n_clusters} clusters)',
                      fontsize=12, fontweight='bold')
        ax1.set_xlabel('Feature 1')
        ax1.set_ylabel('Feature 2')

        # Bayesian GMM clustering
        ax2 = axes[1]
        scatter2 = ax2.scatter(X_scaled[:, 0], X_scaled[:, 1], c=bayesian_result.labels,
                               cmap='tab10', s=30, alpha=0.7)
        ax2.set_title(f'Bayesian GMM Clustering\n(Effective K={bayesian_result.effective_k
            })',
                      fontsize=12, fontweight='bold')
        ax2.set_xlabel('Feature 1')
        ax2.set_ylabel('Feature 2')

        # Entropy visualization
        ax3 = axes[2]
        scatter3 = ax3.scatter(X_scaled[:, 0], X_scaled[:, 1], c=point_entropy,
                               cmap='YlOrRd', s=30, alpha=0.8, vmin=0, vmax=1)
        cbar = plt.colorbar(scatter3, ax=ax3)
        cbar.set_label('Normalized Entropy', fontsize=10)
        ax3.set_title('Posterior Uncertainty\n(High entropy = ambiguous)',
                      fontsize=12, fontweight='bold')
        ax3.set_xlabel('Feature 1')
        ax3.set_ylabel('Feature 2')

        plt.suptitle(f'Posterior Uncertainty Analysis: {dataset.name}',
                     fontsize=14, fontweight='bold')
        plt.tight_layout()

        save_path = output_dir / f"bonus3_uncertainty_{dataset.name.lower().replace(' ','
            _')}.png"
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"  Saved: {save_path}")
        plt.close(fig)

    # Create combined visualization
    n_datasets = len([d for d in datasets if d.X.shape[1] == 2])
    if n_datasets > 0:
        fig, axes = plt.subplots(n_datasets, 3, figsize=(15, 4 * n_datasets))
        if n_datasets == 1:
            axes = axes.reshape(1, -1)

        row_idx = 0
        for dataset in datasets:
            if dataset.X.shape[1] != 2:
                continue

            data = results[dataset.name]
```

```
            X = data['X']

            # Ground truth
            axes[row_idx, 0].scatter(X[:, 0], X[:, 1], c=data['y_true'],
                                     cmap='tab10', s=20, alpha=0.7)
            axes[row_idx, 0].set_title(f'{dataset.name}: Ground Truth')
            axes[row_idx, 0].set_xticks([])
            axes[row_idx, 0].set_yticks([])

            # Clustering
            axes[row_idx, 1].scatter(X[:, 0], X[:, 1], c=data['labels'],
                                     cmap='tab10', s=20, alpha=0.7)
            axes[row_idx, 1].set_title(f'Bayesian GMM (K={data["effective_k"]})')
            axes[row_idx, 1].set_xticks([])
            axes[row_idx, 1].set_yticks([])

            # Entropy
            sc = axes[row_idx, 2].scatter(X[:, 0], X[:, 1], c=data['entropy'],
                                          cmap='YlOrRd', s=20, alpha=0.8, vmin=0, vmax=1)
            axes[row_idx, 2].set_title('Uncertainty (Entropy)')
            axes[row_idx, 2].set_xticks([])
            axes[row_idx, 2].set_yticks([])

            row_idx += 1

        # Add colorbar
        fig.subplots_adjust(right=0.92)
        cbar_ax = fig.add_axes([0.94, 0.15, 0.02, 0.7])
        cbar = fig.colorbar(sc, cax=cbar_ax)
        cbar.set_label('Normalized Entropy', fontsize=10)

        plt.suptitle('Posterior Uncertainty Across Datasets',
                     fontsize=14, fontweight='bold', y=1.02)

        save_path = output_dir / "bonus3_uncertainty_all_datasets.png"
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"\nSaved combined visualization: {save_path}")
        plt.close(fig)

    print("\n" + "=" * 70)
    print("ANALYSIS: Posterior Uncertainty")
    print("=" * 70)
    print("""
    Interpretation:

1. Low Entropy (Yellow):
   - High confidence in cluster assignment
   - Points clearly belong to one cluster

2. High Entropy (Red):
   - Uncertain cluster assignment
   - Typically boundary points or overlapping regions
   - Model acknowledges ambiguity

3. Bayesian Advantage:
   - Probabilistic assignments capture uncertainty
   - Hard clustering (K-Means) cannot express this
   - Useful for downstream decision-making
""")

    return results


# ======================================
# Main Bonus Experiment Runner
```

```python
# ========================================

def run_all_bonus_experiments(
    output_dir: Path = Path("outputs/bonus"),
    random_state: int = 42,
) -> Dict:
    """
    Run all bonus experiments for Bayesian clustering.

    Args:
        output_dir: Directory to save outputs
        random_state: Random seed

    Returns:
        Dictionary containing all results
    """
    print("\n" + "=" * 70)
    print("BONUS: Bayesian Gaussian Mixture Model Experiments")
    print("=" * 70)

    output_dir.mkdir(parents=True, exist_ok=True)

    # Get synthetic datasets
    datasets = get_synthetic_datasets(random_state=random_state)

    results = {}

    # Bonus Experiment 1: Effective Number of Clusters
    results['experiment1'] = run_bonus_experiment1(
        datasets=datasets,
        output_dir=output_dir,
        random_state=random_state,
    )

    # Bonus Experiment 2: Concentration Parameter Sensitivity
    results['experiment2'] = run_bonus_experiment2(
        datasets=datasets,
        output_dir=output_dir,
        random_state=random_state,
    )

    # Bonus Experiment 3: Posterior Uncertainty Visualization
    results['experiment3'] = run_bonus_experiment3(
        datasets=datasets,
        output_dir=output_dir,
        random_state=random_state,
    )

    # Final summary
    print("\n" + "=" * 70)
    print("BONUS EXPERIMENTS COMPLETED")
    print("=" * 70)
    print(f"""
   All bonus experiment results saved to: {output_dir}/

    Generated Files:
  - bonus1_effective_k_*.png: EM-GMM vs Bayesian GMM comparison
  - bonus1_summary.csv: Effective K comparison table
  - bonus2_alpha_sensitivity_*.png: Concentration parameter analysis
  - bonus3_uncertainty_*.png: Posterior entropy visualization
  - bonus3_uncertainty_all_datasets.png: Combined uncertainty view

    Key Takeaways:

1. Bayesian GMM automatically infers cluster number:
```

```
    - No need to manually specify K
    - Dirichlet Process prior controls complexity

2. Concentration parameter    controls model complexity:
    - Small        fewer clusters
    - Large        more clusters
    - Optimal      is dataset-dependent

3. Posterior uncertainty provides valuable information:
    - Identifies ambiguous/boundary points
    - Enables principled decision-making under uncertainty
    - Not available in hard clustering methods
""")

    return results
```

A.4. src/clustering.py

```
"""
Clustering algorithm wrappers for unified interface.
"""

import numpy as np
from sklearn.cluster import (
    KMeans,
    AgglomerativeClustering,
    DBSCAN,
    SpectralClustering,
)
from sklearn.mixture import GaussianMixture, BayesianGaussianMixture
from dataclasses import dataclass
from typing import Optional, Dict, Any, Callable
from abc import ABC, abstractmethod


@dataclass
class ClusteringResult:
    """Container for clustering results."""
    algorithm_name: str
    labels: np.ndarray
    n_clusters_found: int
    params: Dict[str, Any]
    model: Any  # The fitted model object


class ClusteringAlgorithm(ABC):
    """Abstract base class for clustering algorithms."""

    @property
    @abstractmethod
    def name(self) -> str:
        """Algorithm name for display."""
        pass

    @abstractmethod
    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        """Fit the model and return clustering results."""
        pass


class KMeansClustering(ClusteringAlgorithm):
    """K-Means / K-Means++ clustering."""

    def __init__(
```

```
        self,
        n_clusters: int,
        init: str = "k-means++",
        n_init: int = 10,
        random_state: int = 42,
    ):
        self.n_clusters = n_clusters
        self.init = init
        self.n_init = n_init
        self.random_state = random_state

    @property
    def name(self) -> str:
        return f"K-Means (k={self.n_clusters})"

    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        model = KMeans(
            n_clusters=self.n_clusters,
            init=self.init,
            n_init=self.n_init,
            random_state=self.random_state,
        )
        labels = model.fit_predict(X)
        return ClusteringResult(
            algorithm_name=self.name,
            labels=labels,
            n_clusters_found=self.n_clusters,
            params={
                "n_clusters": self.n_clusters,
                "init": self.init,
                "n_init": self.n_init,
            },
            model=model,
        )


class HierarchicalClustering(ClusteringAlgorithm):
    """Agglomerative Hierarchical Clustering."""

    def __init__(
        self,
        n_clusters: int,
        linkage: str = "ward",
    ):
        self.n_clusters = n_clusters
        self.linkage = linkage

    @property
    def name(self) -> str:
        return f"Hierarchical ({self.linkage})"

    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        model = AgglomerativeClustering(
            n_clusters=self.n_clusters,
            linkage=self.linkage,
        )
        labels = model.fit_predict(X)
        return ClusteringResult(
            algorithm_name=self.name,
            labels=labels,
            n_clusters_found=self.n_clusters,
            params={
                "n_clusters": self.n_clusters,
                "linkage": self.linkage,
            },
```

```
            model=model,
        )


class DBSCANClustering(ClusteringAlgorithm):
    """DBSCAN density-based clustering."""

    def __init__(
        self,
        eps: float = 0.5,
        min_samples: int = 5,
    ):
        self.eps = eps
        self.min_samples = min_samples

    @property
    def name(self) -> str:
        return f"DBSCAN (eps={self.eps})"

    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        model = DBSCAN(
            eps=self.eps,
            min_samples=self.min_samples,
        )
        labels = model.fit_predict(X)
        # Count clusters (excluding noise label -1)
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        return ClusteringResult(
            algorithm_name=self.name,
            labels=labels,
            n_clusters_found=n_clusters,
            params={
                "eps": self.eps,
                "min_samples": self.min_samples,
            },
            model=model,
        )


class GMMClustering(ClusteringAlgorithm):
    """Gaussian Mixture Model clustering (EM algorithm)."""

    def __init__(
        self,
        n_components: int,
        covariance_type: str = "full",
        random_state: int = 42,
    ):
        self.n_components = n_components
        self.covariance_type = covariance_type
        self.random_state = random_state

    @property
    def name(self) -> str:
        return f"GMM (k={self.n_components})"

    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        model = GaussianMixture(
            n_components=self.n_components,
            covariance_type=self.covariance_type,
            random_state=self.random_state,
        )
        labels = model.fit_predict(X)
        return ClusteringResult(
            algorithm_name=self.name,
```

```
            labels=labels,
            n_clusters_found=self.n_components,
            params={
                "n_components": self.n_components,
                "covariance_type": self.covariance_type,
            },
            model=model,
        )


class SpectralClusteringWrapper(ClusteringAlgorithm):
    """Spectral Clustering with RBF affinity."""

    def __init__(
        self,
        n_clusters: int,
        affinity: str = "rbf",
        gamma: float = 1.0,
        random_state: int = 42,
    ):
        self.n_clusters = n_clusters
        self.affinity = affinity
        self.gamma = gamma
        self.random_state = random_state

    @property
    def name(self) -> str:
        return f"Spectral (k={self.n_clusters})"

    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        model = SpectralClustering(
            n_clusters=self.n_clusters,
            affinity=self.affinity,
            gamma=self.gamma,
            random_state=self.random_state,
            assign_labels="kmeans",
        )
        labels = model.fit_predict(X)
        return ClusteringResult(
            algorithm_name=self.name,
            labels=labels,
            n_clusters_found=self.n_clusters,
            params={
                "n_clusters": self.n_clusters,
                "affinity": self.affinity,
                "gamma": self.gamma,
            },
            model=model,
        )


class BayesianGMMClustering(ClusteringAlgorithm):
    """Bayesian Gaussian Mixture Model (Variational Inference)."""

    def __init__(
        self,
        n_components: int = 10,
        weight_concentration_prior_type: str = "dirichlet_process",
        weight_concentration_prior: float = 0.01,
        covariance_type: str = "full",
        random_state: int = 42,
    ):
        self.n_components = n_components
        self.weight_concentration_prior_type = weight_concentration_prior_type
        self.weight_concentration_prior = weight_concentration_prior
```

```
        self.covariance_type = covariance_type
        self.random_state = random_state

    @property
    def name(self) -> str:
        return f"Bayesian GMM (K_max={self.n_components})"

    def fit_predict(self, X: np.ndarray) -> ClusteringResult:
        model = BayesianGaussianMixture(
            n_components=self.n_components,
            weight_concentration_prior_type=self.weight_concentration_prior_type,
            weight_concentration_prior=self.weight_concentration_prior,
            covariance_type=self.covariance_type,
            random_state=self.random_state,
        )
        labels = model.fit_predict(X)
        # Count effective clusters (components with significant weight)
        effective_clusters = np.sum(model.weights_ > 0.01)
        return ClusteringResult(
            algorithm_name=self.name,
            labels=labels,
            n_clusters_found=effective_clusters,
            params={
                "n_components": self.n_components,
                "weight_concentration_prior_type": self.weight_concentration_prior_type,
                "weight_concentration_prior": self.weight_concentration_prior,
                "weights": model.weights_,
            },
            model=model,
        )


def get_main_algorithms(n_clusters: int, random_state: int = 42) -> list[
    ClusteringAlgorithm]:
    """
    Get the five main clustering algorithms for Experiment 1.

    Args:
        n_clusters: Target number of clusters (from ground truth)
        random_state: Random seed for reproducibility

    Returns:
        List of configured clustering algorithms
    """
    return [
        KMeansClustering(n_clusters=n_clusters, random_state=random_state),
        HierarchicalClustering(n_clusters=n_clusters, linkage="ward"),
        DBSCANClustering(eps=0.5, min_samples=5),
        GMMClustering(n_components=n_clusters, random_state=random_state),
        SpectralClusteringWrapper(n_clusters=n_clusters, random_state=random_state),
    ]


def get_hierarchical_variants(n_clusters: int) -> list[ClusteringAlgorithm]:
    """Get all hierarchical clustering linkage variants."""
    return [
        HierarchicalClustering(n_clusters=n_clusters, linkage="single"),
        HierarchicalClustering(n_clusters=n_clusters, linkage="complete"),
        HierarchicalClustering(n_clusters=n_clusters, linkage="ward"),
    ]


def tune_dbscan_eps(X: np.ndarray, target_n_clusters: int) -> float:
    """
    Heuristic to tune DBSCAN eps parameter based on data.
```

```
    Uses k-distance graph approach: compute distance to k-th nearest neighbor
    and use a percentile of those distances.
    """
    from sklearn.neighbors import NearestNeighbors

    k = min(5, X.shape[0] - 1)
    nn = NearestNeighbors(n_neighbors=k)
    nn.fit(X)
    distances, _ = nn.kneighbors(X)
    k_distances = distances[:, -1]

    # Use the knee point (elbow) in the k-distance graph
    # Approximate by taking a percentile
    eps = np.percentile(k_distances, 90)
    return eps
```

A.5. src/datasets.py

```
"""
Synthetic and real-world dataset generation for clustering experiments.
"""

import numpy as np
from sklearn.datasets import (
    make_blobs,
    make_moons,
    make_circles,
    load_iris,
    load_wine,
    load_digits,
)
from sklearn.preprocessing import StandardScaler
from dataclasses import dataclass
from typing import Optional, Tuple


@dataclass
class ClusteringDataset:
    """Container for clustering dataset with metadata."""
    name: str
    X: np.ndarray
    y: np.ndarray
    n_clusters: int
    description: str


def generate_blobs(
    n_samples: int = 500,
    n_features: int = 2,
    centers: int = 3,
    cluster_std: float = 1.0,
    random_state: int = 42,
) -> ClusteringDataset:
    """
    Generate isotropic Gaussian blobs (spherical clusters).

    This is a baseline sanity check dataset where K-Means should perform well.
    """
    X, y = make_blobs(
        n_samples=n_samples,
        n_features=n_features,
        centers=centers,
        cluster_std=cluster_std,
```

```
        random_state=random_state,
    )
    return ClusteringDataset(
        name="Blobs",
        X=X,
        y=y,
        n_clusters=centers,
        description="Spherical Gaussian clusters - baseline for centroid-based methods",
    )


def generate_moons(
    n_samples: int = 500,
    noise: float = 0.1,
    random_state: int = 42,
) -> ClusteringDataset:
    """
    Generate two interleaving half circles (moons).

    This dataset has non-convex structure that challenges K-Means.
    """
    X, y = make_moons(
        n_samples=n_samples,
        noise=noise,
        random_state=random_state,
    )
    return ClusteringDataset(
        name="Two Moons",
        X=X,
        y=y,
        n_clusters=2,
        description="Non-convex interleaving crescents - challenges centroid methods",
    )


def generate_circles(
    n_samples: int = 500,
    noise: float = 0.05,
    factor: float = 0.5,
    random_state: int = 42,
) -> ClusteringDataset:
    """
    Generate concentric circles (nested cluster structure).

    This dataset has nested structure that requires density or graph-based methods.
    """
    X, y = make_circles(
        n_samples=n_samples,
        noise=noise,
        factor=factor,
        random_state=random_state,
    )
    return ClusteringDataset(
        name="Concentric Circles",
        X=X,
        y=y,
        n_clusters=2,
        description="Nested circular clusters - requires non-linear methods",
    )


def generate_anisotropic(
    n_samples: int = 500,
    centers: int = 3,
    random_state: int = 42,
```

```
) -> ClusteringDataset:
    """
    Generate anisotropic (elliptical) Gaussian blobs.

    Created by applying a linear transformation to standard blobs.
    This challenges K-Means which assumes spherical clusters.
    """
    X, y = make_blobs(
        n_samples=n_samples,
        n_features=2,
        centers=centers,
        cluster_std=1.0,
        random_state=random_state,
    )
    # Apply linear transformation to create elongated clusters
    transformation = np.array([[0.6, -0.6], [-0.4, 0.8]])
    X = X @ transformation

    return ClusteringDataset(
        name="Anisotropic Blobs",
        X=X,
        y=y,
        n_clusters=centers,
        description="Elliptical clusters via linear transformation - GMM advantage",
    )


def load_iris_dataset() -> ClusteringDataset:
    """Load Iris dataset for clustering evaluation."""
    data = load_iris()
    return ClusteringDataset(
        name="Iris",
        X=data.data,
        y=data.target,
        n_clusters=3,
        description="Classic 4-feature flower dataset (150 samples, 3 classes)",
    )


def load_wine_dataset() -> ClusteringDataset:
    """Load Wine dataset for clustering evaluation."""
    data = load_wine()
    return ClusteringDataset(
        name="Wine",
        X=data.data,
        y=data.target,
        n_clusters=3,
        description="Wine recognition dataset (178 samples, 13 features, 3 classes)",
    )


def load_digits_dataset() -> ClusteringDataset:
    """Load Digits dataset for high-dimensional clustering evaluation."""
    data = load_digits()
    return ClusteringDataset(
        name="Digits",
        X=data.data,
        y=data.target,
        n_clusters=10,
        description="8x8 handwritten digits (1797 samples, 64 features, 10 classes)",
    )


def get_synthetic_datasets(random_state: int = 42) -> list[ClusteringDataset]:
    """Generate all synthetic datasets for Experiment 1."""
```

```
    return [
        generate_blobs(random_state=random_state),
        generate_moons(random_state=random_state),
        generate_circles(random_state=random_state),
        generate_anisotropic(random_state=random_state),
    ]


def get_real_datasets() -> list[ClusteringDataset]:
    """Load all real-world datasets."""
    return [
        load_iris_dataset(),
        load_wine_dataset(),
        load_digits_dataset(),
    ]


def standardize_data(X: np.ndarray) -> Tuple[np.ndarray, StandardScaler]:
    """
    Apply z-score normalization to features.

    Returns:
        Tuple of (standardized data, fitted scaler)
    """
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    return X_scaled, scaler
```

A.6. src/experiment2.py

```
"""
Experiment 2: Hyperparameter Sensitivity and Metric Conflict

This module analyzes how sensitive clustering results are to hyperparameter choices
and reveals conflicts between internal and external evaluation metrics.

Two sub-experiments:
2.1 K-Means: Choice of Number of Clusters (k sweep from 2 to 10)
2.2 DBSCAN: Density Parameters (eps and min_samples sweep)
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass
from sklearn.cluster import KMeans, DBSCAN
from sklearn.metrics import (
    silhouette_score,
    calinski_harabasz_score,
    davies_bouldin_score,
    adjusted_rand_score,
    normalized_mutual_info_score,
)

from .datasets import ClusteringDataset, standardize_data, get_synthetic_datasets


@dataclass
class KMeansAnalysisResult:
    """Results from K-Means k sweep analysis."""
    k_values: np.ndarray
```

```
    sse_values: np.ndarray  # Sum of Squared Errors (inertia)
    silhouette_values: np.ndarray
    calinski_harabasz_values: np.ndarray
    davies_bouldin_values: np.ndarray
    ari_values: Optional[np.ndarray] = None
    nmi_values: Optional[np.ndarray] = None


@dataclass
class DBSCANAnalysisResult:
    """Results from DBSCAN parameter sweep analysis."""
    eps_values: np.ndarray
    min_samples_values: np.ndarray
    ari_matrix: np.ndarray  # Shape: (len(eps), len(min_samples))
    nmi_matrix: np.ndarray
    n_clusters_matrix: np.ndarray
    noise_ratio_matrix: np.ndarray


def analyze_kmeans_k_selection(
    X: np.ndarray,
    y_true: Optional[np.ndarray] = None,
    k_range: range = range(2, 11),
    random_state: int = 42,
) -> KMeansAnalysisResult:
    """
    Analyze K-Means performance across different values of k.

    Args:
        X: Feature matrix
        y_true: Ground truth labels (optional, for external metrics)
        k_range: Range of k values to test
        random_state: Random seed for reproducibility

    Returns:
        KMeansAnalysisResult with all computed metrics
    """
    k_values = np.array(list(k_range))
    n_k = len(k_values)

    sse_values = np.zeros(n_k)
    silhouette_values = np.zeros(n_k)
    calinski_harabasz_values = np.zeros(n_k)
    davies_bouldin_values = np.zeros(n_k)
    ari_values = np.zeros(n_k) if y_true is not None else None
    nmi_values = np.zeros(n_k) if y_true is not None else None

    for i, k in enumerate(k_values):
        kmeans = KMeans(n_clusters=k, random_state=random_state, n_init=10)
        labels = kmeans.fit_predict(X)

        # SSE (Sum of Squared Errors) - inertia
        sse_values[i] = kmeans.inertia_

        # Internal metrics
        silhouette_values[i] = silhouette_score(X, labels)
        calinski_harabasz_values[i] = calinski_harabasz_score(X, labels)
        davies_bouldin_values[i] = davies_bouldin_score(X, labels)

        # External metrics (if ground truth available)
        if y_true is not None:
            ari_values[i] = adjusted_rand_score(y_true, labels)
            nmi_values[i] = normalized_mutual_info_score(y_true, labels)

    return KMeansAnalysisResult(
```

```
        k_values=k_values,
        sse_values=sse_values,
        silhouette_values=silhouette_values,
        calinski_harabasz_values=calinski_harabasz_values,
        davies_bouldin_values=davies_bouldin_values,
        ari_values=ari_values,
        nmi_values=nmi_values,
    )


def analyze_dbscan_parameters(
    X: np.ndarray,
    y_true: np.ndarray,
    eps_range: np.ndarray = None,
    min_samples_range: np.ndarray = None,
) -> DBSCANAnalysisResult:
    """
    Analyze DBSCAN performance across different eps and min_samples combinations.

    Args:
        X: Feature matrix
        y_true: Ground truth labels
        eps_range: Array of eps values to test
        min_samples_range: Array of min_samples values to test

    Returns:
        DBSCANAnalysisResult with all computed metrics
    """
    # Default ranges based on data statistics
    if eps_range is None:
        # Use k-distance based heuristic
        from sklearn.neighbors import NearestNeighbors
        nn = NearestNeighbors(n_neighbors=5)
        nn.fit(X)
        distances, _ = nn.kneighbors(X)
        k_dist = np.sort(distances[:, -1])
        eps_min = np.percentile(k_dist, 10)
        eps_max = np.percentile(k_dist, 95)
        eps_range = np.linspace(eps_min, eps_max, 15)

    if min_samples_range is None:
        min_samples_range = np.array([3, 5, 7, 10, 15, 20])

    n_eps = len(eps_range)
    n_min_samples = len(min_samples_range)

    ari_matrix = np.zeros((n_eps, n_min_samples))
    nmi_matrix = np.zeros((n_eps, n_min_samples))
    n_clusters_matrix = np.zeros((n_eps, n_min_samples))
    noise_ratio_matrix = np.zeros((n_eps, n_min_samples))

    for i, eps in enumerate(eps_range):
        for j, min_samples in enumerate(min_samples_range):
            dbscan = DBSCAN(eps=eps, min_samples=int(min_samples))
            labels = dbscan.fit_predict(X)

            # Count clusters (excluding noise)
            n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
            n_clusters_matrix[i, j] = n_clusters

            # Noise ratio
            noise_ratio = np.sum(labels == -1) / len(labels)
            noise_ratio_matrix[i, j] = noise_ratio

            # External metrics (filter noise for fair comparison)
```

```
            mask = labels != -1
            if mask.sum() > 1 and n_clusters >= 1:
                # Only compute ARI/NMI if we have valid clusters
                if n_clusters >= 2:
                    ari_matrix[i, j] = adjusted_rand_score(y_true[mask], labels[mask])
                    nmi_matrix[i, j] = normalized_mutual_info_score(y_true[mask], labels[
                        mask])
                else:
                    ari_matrix[i, j] = 0
                    nmi_matrix[i, j] = 0
            else:
                ari_matrix[i, j] = -1  # Invalid clustering
                nmi_matrix[i, j] = -1

    return DBSCANAnalysisResult(
        eps_values=eps_range,
        min_samples_values=min_samples_range,
        ari_matrix=ari_matrix,
        nmi_matrix=nmi_matrix,
        n_clusters_matrix=n_clusters_matrix,
        noise_ratio_matrix=noise_ratio_matrix,
    )


def plot_kmeans_elbow(
    result: KMeansAnalysisResult,
    dataset_name: str,
    true_k: Optional[int] = None,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (12, 8),
) -> plt.Figure:
    """
    Plot K-Means elbow analysis with multiple metrics.

    Creates a 2x2 subplot:
    - SSE (Elbow Method)
    - Silhouette Score
    - ARI/NMI (external metrics)
    - Davies-Bouldin / Calinski-Harabasz
    """
    fig, axes = plt.subplots(2, 2, figsize=figsize)

    k_values = result.k_values

    # Plot 1: SSE (Elbow Method)
    ax1 = axes[0, 0]
    ax1.plot(k_values, result.sse_values, 'b-o', linewidth=2, markersize=8)
    ax1.set_xlabel('Number of Clusters (k)', fontsize=11)
    ax1.set_ylabel('SSE (Inertia)', fontsize=11)
    ax1.set_title('Elbow Method (SSE)', fontsize=12, fontweight='bold')
    ax1.grid(True, alpha=0.3)
    if true_k:
        ax1.axvline(x=true_k, color='r', linestyle='--', label=f'True k={true_k}')
        ax1.legend()

    # Plot 2: Silhouette Score
    ax2 = axes[0, 1]
    ax2.plot(k_values, result.silhouette_values, 'g-o', linewidth=2, markersize=8)
    ax2.set_xlabel('Number of Clusters (k)', fontsize=11)
    ax2.set_ylabel('Silhouette Score', fontsize=11)
    ax2.set_title('Silhouette Score', fontsize=12, fontweight='bold')
    ax2.grid(True, alpha=0.3)
    # Mark optimal k (max silhouette)
    optimal_k_sil = k_values[np.argmax(result.silhouette_values)]
    ax2.axvline(x=optimal_k_sil, color='orange', linestyle='--',
```

```
                label=f'Optimal k={optimal_k_sil}')
    if true_k:
        ax2.axvline(x=true_k, color='r', linestyle='--', label=f'True k={true_k}')
    ax2.legend()

    # Plot 3: External Metrics (ARI/NMI)
    ax3 = axes[1, 0]
    if result.ari_values is not None:
        ax3.plot(k_values, result.ari_values, 'r-o', linewidth=2, markersize=8, label='ARI')
        ax3.plot(k_values, result.nmi_values, 'm-s', linewidth=2, markersize=8, label='NMI')
        ax3.set_xlabel('Number of Clusters (k)', fontsize=11)
        ax3.set_ylabel('Score', fontsize=11)
        ax3.set_title('External Metrics (ARI & NMI)', fontsize=12, fontweight='bold')
        ax3.legend()
        ax3.grid(True, alpha=0.3)
        if true_k:
            ax3.axvline(x=true_k, color='r', linestyle='--', alpha=0.5)
    else:
        ax3.text(0.5, 0.5, 'No ground truth available', ha='center', va='center',
                 transform=ax3.transAxes, fontsize=12)
        ax3.set_title('External Metrics (N/A)', fontsize=12, fontweight='bold')

    # Plot 4: Davies-Bouldin and Calinski-Harabasz
    ax4 = axes[1, 1]
    ax4_twin = ax4.twinx()

    line1, = ax4.plot(k_values, result.davies_bouldin_values, 'c-o',
                      linewidth=2, markersize=8, label='Davies-Bouldin (   )')
    ax4.set_xlabel('Number of Clusters (k)', fontsize=11)
    ax4.set_ylabel('Davies-Bouldin Index', color='c', fontsize=11)
    ax4.tick_params(axis='y', labelcolor='c')

    line2, = ax4_twin.plot(k_values, result.calinski_harabasz_values, 'y-s',
                           linewidth=2, markersize=8, label='Calinski-Harabasz (   )')
    ax4_twin.set_ylabel('Calinski-Harabasz Index', color='y', fontsize=11)
    ax4_twin.tick_params(axis='y', labelcolor='y')

    ax4.set_title('Other Internal Metrics', fontsize=12, fontweight='bold')
    ax4.legend([line1, line2], ['Davies-Bouldin (   )', 'Calinski-Harabasz (   )'], loc='
        upper right')
    ax4.grid(True, alpha=0.3)

    plt.suptitle(f'K-Means Hyperparameter Analysis: {dataset_name}',
                 fontsize=14, fontweight='bold', y=1.02)
    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def plot_dbscan_heatmaps(
    result: DBSCANAnalysisResult,
    dataset_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (16, 12),
) -> plt.Figure:
    """
    Plot DBSCAN parameter sensitivity as heatmaps.

```

```
    Creates a 2x2 subplot:
    - ARI heatmap
    - NMI heatmap
    - Number of clusters heatmap
    - Noise ratio heatmap
    """
    fig, axes = plt.subplots(2, 2, figsize=figsize)

    eps_labels = [f'{e:.2f}' for e in result.eps_values]
    ms_labels = [str(int(m)) for m in result.min_samples_values]

    # ARI Heatmap
    ax1 = axes[0, 0]
    # Mask invalid values
    ari_masked = np.ma.masked_where(result.ari_matrix < 0, result.ari_matrix)
    im1 = ax1.imshow(ari_masked, cmap='RdYlGn', aspect='auto', vmin=0, vmax=1)
    ax1.set_xticks(np.arange(len(ms_labels)))
    ax1.set_yticks(np.arange(len(eps_labels)))
    ax1.set_xticklabels(ms_labels)
    ax1.set_yticklabels(eps_labels)
    ax1.set_xlabel('min_samples', fontsize=11)
    ax1.set_ylabel('eps', fontsize=11)
    ax1.set_title('Adjusted Rand Index (ARI)', fontsize=12, fontweight='bold')
    plt.colorbar(im1, ax=ax1)

    # Add text annotations
    for i in range(len(eps_labels)):
        for j in range(len(ms_labels)):
            val = result.ari_matrix[i, j]
            if val >= 0:
                color = 'white' if val < 0.3 or val > 0.7 else 'black'
                ax1.text(j, i, f'{val:.2f}', ha='center', va='center',
                        color=color, fontsize=8)

    # NMI Heatmap
    ax2 = axes[0, 1]
    nmi_masked = np.ma.masked_where(result.nmi_matrix < 0, result.nmi_matrix)
    im2 = ax2.imshow(nmi_masked, cmap='RdYlGn', aspect='auto', vmin=0, vmax=1)
    ax2.set_xticks(np.arange(len(ms_labels)))
    ax2.set_yticks(np.arange(len(eps_labels)))
    ax2.set_xticklabels(ms_labels)
    ax2.set_yticklabels(eps_labels)
    ax2.set_xlabel('min_samples', fontsize=11)
    ax2.set_ylabel('eps', fontsize=11)
    ax2.set_title('Normalized Mutual Information (NMI)', fontsize=12, fontweight='bold')
    plt.colorbar(im2, ax=ax2)

    for i in range(len(eps_labels)):
        for j in range(len(ms_labels)):
            val = result.nmi_matrix[i, j]
            if val >= 0:
                color = 'white' if val < 0.3 or val > 0.7 else 'black'
                ax2.text(j, i, f'{val:.2f}', ha='center', va='center',
                        color=color, fontsize=8)

    # Number of Clusters Heatmap
    ax3 = axes[1, 0]
    im3 = ax3.imshow(result.n_clusters_matrix, cmap='Blues', aspect='auto')
    ax3.set_xticks(np.arange(len(ms_labels)))
    ax3.set_yticks(np.arange(len(eps_labels)))
    ax3.set_xticklabels(ms_labels)
    ax3.set_yticklabels(eps_labels)
    ax3.set_xlabel('min_samples', fontsize=11)
    ax3.set_ylabel('eps', fontsize=11)
    ax3.set_title('Number of Clusters Found', fontsize=12, fontweight='bold')
```

```
    plt.colorbar(im3, ax=ax3)

    for i in range(len(eps_labels)):
        for j in range(len(ms_labels)):
            val = int(result.n_clusters_matrix[i, j])
            ax3.text(j, i, str(val), ha='center', va='center',
                     color='white' if val > 5 else 'black', fontsize=8)

    # Noise Ratio Heatmap
    ax4 = axes[1, 1]
    im4 = ax4.imshow(result.noise_ratio_matrix, cmap='Reds', aspect='auto', vmin=0, vmax
        =1)
    ax4.set_xticks(np.arange(len(ms_labels)))
    ax4.set_yticks(np.arange(len(eps_labels)))
    ax4.set_xticklabels(ms_labels)
    ax4.set_yticklabels(eps_labels)
    ax4.set_xlabel('min_samples', fontsize=11)
    ax4.set_ylabel('eps', fontsize=11)
    ax4.set_title('Noise Ratio', fontsize=12, fontweight='bold')
    plt.colorbar(im4, ax=ax4)

    for i in range(len(eps_labels)):
        for j in range(len(ms_labels)):
            val = result.noise_ratio_matrix[i, j]
            color = 'white' if val > 0.5 else 'black'
            ax4.text(j, i, f'{val:.2f}', ha='center', va='center',
                     color=color, fontsize=8)

    plt.suptitle(f'DBSCAN Parameter Sensitivity: {dataset_name}',
                 fontsize=14, fontweight='bold', y=1.02)
    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def plot_metric_conflict_analysis(
    kmeans_result: KMeansAnalysisResult,
    dataset_name: str,
    true_k: int,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (10, 6),
) -> plt.Figure:
    """
    Visualize conflicts between internal and external metrics.

    Shows cases where internal metrics suggest different k than external metrics.
    """
    fig, ax = plt.subplots(figsize=figsize)

    k_values = kmeans_result.k_values

    # Normalize all metrics to [0, 1] for comparison
    def normalize(arr, higher_is_better=True):
        arr = np.array(arr)
        if not higher_is_better:
            arr = -arr
        return (arr - arr.min()) / (arr.max() - arr.min() + 1e-10)

    # Plot normalized metrics
    ax.plot(k_values, normalize(kmeans_result.silhouette_values),
```

```
            'g-o', linewidth=2, label='Silhouette (normalized)')
    ax.plot(k_values, normalize(kmeans_result.davies_bouldin_values, higher_is_better=
        False),
            'c-s', linewidth=2, label='Davies-Bouldin (normalized, inverted)')

    if kmeans_result.ari_values is not None:
        ax.plot(k_values, normalize(kmeans_result.ari_values),
                'r-^', linewidth=2, label='ARI (normalized)')
        ax.plot(k_values, normalize(kmeans_result.nmi_values),
                'm-d', linewidth=2, label='NMI (normalized)')

    # Mark true k
    ax.axvline(x=true_k, color='black', linestyle='--', linewidth=2,
               label=f'True k={true_k}')

    # Find optimal k for each metric
    optimal_sil = k_values[np.argmax(kmeans_result.silhouette_values)]
    optimal_db = k_values[np.argmin(kmeans_result.davies_bouldin_values)]

    ax.set_xlabel('Number of Clusters (k)', fontsize=12)
    ax.set_ylabel('Normalized Score', fontsize=12)
    ax.set_title(f'Metric Conflict Analysis: {dataset_name}\n'
                 f'(Silhouette optimal: k={optimal_sil}, Davies-Bouldin optimal: k={
                     optimal_db})',
                 fontsize=12, fontweight='bold')
    ax.legend(loc='upper right')
    ax.grid(True, alpha=0.3)
    ax.set_xticks(k_values)

    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def run_experiment2(
    datasets: List[ClusteringDataset] = None,
    output_dir: Path = Path("outputs/experiment2"),
    random_state: int = 42,
) -> Dict:
    """
    Run Experiment 2: Hyperparameter Sensitivity and Metric Conflict.

    Args:
        datasets: List of datasets to use (default: synthetic datasets)
        output_dir: Directory to save outputs
        random_state: Random seed

    Returns:
        Dictionary containing all results
    """
    print("=" * 70)
    print("Experiment 2: Hyperparameter Sensitivity and Metric Conflict")
    print("=" * 70)

    output_dir.mkdir(parents=True, exist_ok=True)

    # Use synthetic datasets if not provided
    if datasets is None:
        datasets = get_synthetic_datasets(random_state=random_state)

```

```
    results = {
        'kmeans': {},
        'dbscan': {},
    }

    # ========================================
    # 2.1 K-Means: Choice of Number of Clusters
    # ========================================
    print("\n[2.1] K-Means: Number of Clusters Analysis")
    print("-" * 50)

    for dataset in datasets:
        print(f"\nDataset: {dataset.name}")

        # Standardize data
        X_scaled, _ = standardize_data(dataset.X)

        # Analyze k selection
        kmeans_result = analyze_kmeans_k_selection(
            X=X_scaled,
            y_true=dataset.y,
            k_range=range(2, 11),
            random_state=random_state,
        )
        results['kmeans'][dataset.name] = kmeans_result

        # Find optimal k values
        optimal_sil = kmeans_result.k_values[np.argmax(kmeans_result.silhouette_values)]
        optimal_db = kmeans_result.k_values[np.argmin(kmeans_result.davies_bouldin_values)
            ]
        optimal_ari = kmeans_result.k_values[np.argmax(kmeans_result.ari_values)] if
            kmeans_result.ari_values is not None else None

        print(f"  True k: {dataset.n_clusters}")
        print(f"  Optimal k (Silhouette): {optimal_sil}")
        print(f"  Optimal k (Davies-Bouldin): {optimal_db}")
        if optimal_ari:
            print(f"  Optimal k (ARI): {optimal_ari}")

        # Check for metric conflict
        if optimal_sil != dataset.n_clusters:
            print(f"    Silhouette suggests k={optimal_sil}, but true k={dataset.
                n_clusters}")

        # Plot elbow analysis
        fig = plot_kmeans_elbow(
            result=kmeans_result,
            dataset_name=dataset.name,
            true_k=dataset.n_clusters,
            save_path=output_dir / f"kmeans_elbow_{dataset.name.lower().replace(' ','_')
                }.png",
        )
        plt.close(fig)

        # Plot metric conflict analysis
        fig = plot_metric_conflict_analysis(
            kmeans_result=kmeans_result,
            dataset_name=dataset.name,
            true_k=dataset.n_clusters,
            save_path=output_dir / f"metric_conflict_{dataset.name.lower().replace(' ','_
                ')}.png",
        )
        plt.close(fig)

    # ========================================
```

```
    # 2.2 DBSCAN: Density Parameters
    # ========================================
    print("\n[2.2] DBSCAN: Density Parameters Analysis")
    print("-" * 50)

    for dataset in datasets:
        print(f"\nDataset: {dataset.name}")

        # Standardize data
        X_scaled, _ = standardize_data(dataset.X)

        # Analyze DBSCAN parameters
        dbscan_result = analyze_dbscan_parameters(
            X=X_scaled,
            y_true=dataset.y,
        )
        results['dbscan'][dataset.name] = dbscan_result

        # Find best parameters
        best_idx = np.unravel_index(np.argmax(dbscan_result.ari_matrix), dbscan_result.
            ari_matrix.shape)
        best_eps = dbscan_result.eps_values[best_idx[0]]
        best_min_samples = int(dbscan_result.min_samples_values[best_idx[1]])
        best_ari = dbscan_result.ari_matrix[best_idx]

        print(f"  Best params: eps={best_eps:.3f}, min_samples={best_min_samples}")
        print(f"  Best ARI: {best_ari:.4f}")
        print(f"  Clusters at best params: {int(dbscan_result.n_clusters_matrix[best_idx])
            }")
        print(f"  Noise ratio at best params: {dbscan_result.noise_ratio_matrix[best_idx
            ]:.2%}")

        # Plot DBSCAN heatmaps
        fig = plot_dbscan_heatmaps(
            result=dbscan_result,
            dataset_name=dataset.name,
            save_path=output_dir / f"dbscan_heatmap_{dataset.name.lower().replace(' ','_
                ')}.png",
        )
        plt.close(fig)

    # ========================================
    # Create summary table
    # ========================================
    print("\n[Summary] Creating summary tables...")

    # K-Means summary
    kmeans_summary = []
    for ds_name, result in results['kmeans'].items():
        row = {
            'Dataset': ds_name,
            'True k': next(d.n_clusters for d in datasets if d.name == ds_name),
            'Optimal k (Silhouette)': int(result.k_values[np.argmax(result.
                silhouette_values)]),
            'Optimal k (DB)': int(result.k_values[np.argmin(result.davies_bouldin_values)
                ]),
            'Optimal k (ARI)': int(result.k_values[np.argmax(result.ari_values)]) if
                result.ari_values is not None else 'N/A',
            'Max ARI': f"{np.max(result.ari_values):.4f}" if result.ari_values is not None
                 else 'N/A',
        }
        kmeans_summary.append(row)

    df_kmeans = pd.DataFrame(kmeans_summary)
    df_kmeans.to_csv(output_dir / "kmeans_k_selection_summary.csv", index=False)
```

```
    # DBSCAN summary
    dbscan_summary = []
    for ds_name, result in results['dbscan'].items():
        best_idx = np.unravel_index(np.argmax(result.ari_matrix), result.ari_matrix.shape)
        row = {
            'Dataset': ds_name,
            'Best eps': f"{result.eps_values[best_idx[0]]:.3f}",
            'Best min_samples': int(result.min_samples_values[best_idx[1]]),
            'Best ARI': f"{result.ari_matrix[best_idx]:.4f}",
            'Clusters Found': int(result.n_clusters_matrix[best_idx]),
            'Noise Ratio': f"{result.noise_ratio_matrix[best_idx]:.2%}",
        }
        dbscan_summary.append(row)

    df_dbscan = pd.DataFrame(dbscan_summary)
    df_dbscan.to_csv(output_dir / "dbscan_parameter_summary.csv", index=False)

    # Print summaries
    print("\n" + "=" * 70)
    print("K-MEANS K SELECTION SUMMARY")
    print("=" * 70)
    print(df_kmeans.to_string(index=False))

    print("\n" + "=" * 70)
    print("DBSCAN PARAMETER SENSITIVITY SUMMARY")
    print("=" * 70)
    print(df_dbscan.to_string(index=False))

    # Analysis
    print("\n" + "=" * 70)
    print("ANALYSIS AND OBSERVATIONS")
    print("=" * 70)
    print("""
    Key Findings:

1. K-Means k Selection:
   - Internal metrics (Silhouette, DB) may disagree with external metrics (ARI, NMI)
   - The elbow method often suggests different k than ground truth
   - This highlights the challenge of unsupervised model selection

2. DBSCAN Parameter Sensitivity:
   - DBSCAN is highly sensitive to eps and min_samples
   - Small eps    many noise points (all noise in extreme cases)
   - Large eps    single cluster (everything merged)
   - Optimal parameter region is often narrow
   - Different datasets require different parameter tuning

3. Metric Conflict Implications:
   - Internal metrics optimize cluster compactness/separation
   - External metrics measure alignment with ground truth
   - In practice, ground truth is unavailable, making model selection difficult
""")

    print(f"\ n   All results saved to: {output_dir}/")
    print("    Experiment 2 completed successfully!")

    return results
```

A.7. src/experiment3.py

```
"""
Experiment 3: High-Dimensional Data and Dimensionality Reduction

```

```python
"""
This module investigates the impact of dimensionality on clustering quality and efficiency.

Procedure:
1. Apply clustering algorithms directly to original features
2. Apply PCA to reduce dimensionality (e.g., 10 and 20 dimensions)
3. Repeat clustering on reduced representations
4. Compare: Runtime, Internal metrics, External metrics
5. Optional: t-SNE plots for qualitative inspection
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass
import time
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

from .datasets import (
    ClusteringDataset,
    standardize_data,
    load_digits_dataset,
    load_iris_dataset,
    load_wine_dataset,
)
from .clustering import (
    get_main_algorithms,
    ClusteringAlgorithm,
    ClusteringResult,
)
from .metrics import compute_all_metrics, ClusteringMetrics


@dataclass
class DimensionalityExperimentResult:
    """Results from a single clustering run with timing."""
    algorithm_name: str
    n_dimensions: int
    dimension_label: str  # e.g., "Original (64)", "PCA-10", "PCA-20"
    runtime_seconds: float
    metrics: ClusteringMetrics
    labels: np.ndarray


def apply_pca(
    X: np.ndarray,
    n_components: int,
    random_state: int = 42,
) -> Tuple[np.ndarray, PCA, float]:
    """
    Apply PCA dimensionality reduction.

    Args:
        X: Feature matrix
        n_components: Number of principal components
        random_state: Random seed

    Returns:
        Tuple of (transformed_data, fitted_PCA_object, explained_variance_ratio)
    """
    pca = PCA(n_components=n_components, random_state=random_state)
```

```
    X_reduced = pca.fit_transform(X)
    explained_variance = np.sum(pca.explained_variance_ratio_)
    return X_reduced, pca, explained_variance


def run_clustering_with_timing(
    algorithm: ClusteringAlgorithm,
    X: np.ndarray,
    y_true: np.ndarray,
    n_dimensions: int,
    dimension_label: str,
) -> DimensionalityExperimentResult:
    """
    Run a clustering algorithm and measure execution time.

    Args:
        algorithm: Clustering algorithm to run
        X: Feature matrix
        y_true: Ground truth labels
        n_dimensions: Number of dimensions in X
        dimension_label: Label for this dimension setting

    Returns:
        DimensionalityExperimentResult with timing and metrics
    """
    # Measure runtime
    start_time = time.perf_counter()
    result = algorithm.fit_predict(X)
    end_time = time.perf_counter()

    runtime = end_time - start_time

    # Compute metrics
    metrics = compute_all_metrics(X, result.labels, y_true)

    return DimensionalityExperimentResult(
        algorithm_name=result.algorithm_name,
        n_dimensions=n_dimensions,
        dimension_label=dimension_label,
        runtime_seconds=runtime,
        metrics=metrics,
        labels=result.labels,
    )


def plot_pca_explained_variance(
    X: np.ndarray,
    max_components: int = None,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (12, 5),
) -> plt.Figure:
    """
    Plot PCA explained variance analysis.

    Creates two subplots:
    - Individual explained variance ratio per component
    - Cumulative explained variance ratio
    """
    if max_components is None:
        max_components = min(X.shape[1], 50)

    pca = PCA(n_components=max_components)
    pca.fit(X)

    fig, axes = plt.subplots(1, 2, figsize=figsize)
```

```
    # Individual variance
    ax1 = axes[0]
    ax1.bar(range(1, max_components + 1), pca.explained_variance_ratio_,
            alpha=0.7, color='steelblue')
    ax1.set_xlabel('Principal Component', fontsize=11)
    ax1.set_ylabel('Explained Variance Ratio', fontsize=11)
    ax1.set_title('Individual Explained Variance', fontsize=12, fontweight='bold')
    ax1.grid(True, alpha=0.3)

    # Cumulative variance
    ax2 = axes[1]
    cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
    ax2.plot(range(1, max_components + 1), cumulative_variance,
             'b-o', linewidth=2, markersize=4)
    ax2.axhline(y=0.9, color='r', linestyle='--', label='90% variance')
    ax2.axhline(y=0.95, color='orange', linestyle='--', label='95% variance')
    ax2.set_xlabel('Number of Components', fontsize=11)
    ax2.set_ylabel('Cumulative Explained Variance', fontsize=11)
    ax2.set_title('Cumulative Explained Variance', fontsize=12, fontweight='bold')
    ax2.legend()
    ax2.grid(True, alpha=0.3)
    ax2.set_ylim([0, 1.05])

    # Find components for 90% and 95% variance
    n_90 = np.argmax(cumulative_variance >= 0.9) + 1
    n_95 = np.argmax(cumulative_variance >= 0.95) + 1
    ax2.axvline(x=n_90, color='r', linestyle=':', alpha=0.5)
    ax2.axvline(x=n_95, color='orange', linestyle=':', alpha=0.5)

    plt.suptitle(f'PCA Analysis (Original: {X.shape[1]} dimensions)\n'
                 f'90% variance: {n_90} components, 95% variance: {n_95} components',
                 fontsize=12, fontweight='bold')
    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def plot_tsne_visualization(
    X: np.ndarray,
    y_true: np.ndarray,
    labels_dict: Dict[str, np.ndarray],
    dataset_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (16, 10),
    random_state: int = 42,
) -> plt.Figure:
    """
    Create t-SNE visualizations comparing ground truth and clustering results.

    Args:
        X: Feature matrix (will be reduced to 2D via t-SNE)
        y_true: Ground truth labels
        labels_dict: Dictionary mapping algorithm names to predicted labels
        dataset_name: Name of dataset for title
        save_path: Path to save figure
        figsize: Figure size
        random_state: Random seed for t-SNE

    Returns:
```

```
        matplotlib Figure object
    """
    # Apply t-SNE for 2D visualization
    print("  Computing t-SNE embedding (this may take a moment)...")
    tsne = TSNE(n_components=2, random_state=random_state, perplexity=30, n_iter=1000)
    X_tsne = tsne.fit_transform(X)

    n_plots = len(labels_dict) + 1  # +1 for ground truth
    n_cols = min(3, n_plots)
    n_rows = (n_plots + n_cols - 1) // n_cols

    fig, axes = plt.subplots(n_rows, n_cols, figsize=figsize)
    axes = np.atleast_2d(axes).flatten()

    colors = plt.cm.tab10(np.linspace(0, 1, 10))

    def plot_scatter(ax, X_2d, labels, title):
        unique_labels = np.unique(labels)
        for label in unique_labels:
            mask = labels == label
            if label == -1:
                ax.scatter(X_2d[mask, 0], X_2d[mask, 1], c='gray', marker='x',
                           s=20, alpha=0.5, label='Noise')
            else:
                color = colors[label % len(colors)]
                ax.scatter(X_2d[mask, 0], X_2d[mask, 1], c=[color],
                           s=20, alpha=0.7)
        ax.set_title(title, fontsize=11, fontweight='bold')
        ax.set_xticks([])
        ax.set_yticks([])

    # Plot ground truth
    plot_scatter(axes[0], X_tsne, y_true, f'Ground Truth\n({len(np.unique(y_true))}
        classes)')

    # Plot each algorithm's results
    for idx, (algo_name, labels) in enumerate(labels_dict.items(), start=1):
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        plot_scatter(axes[idx], X_tsne, labels, f'{algo_name}\n({n_clusters} clusters)')

    # Hide unused subplots
    for idx in range(n_plots, len(axes)):
        axes[idx].set_visible(False)

    plt.suptitle(f't-SNE Visualization: {dataset_name}', fontsize=14, fontweight='bold')
    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def plot_dimensionality_comparison(
    results: List[DimensionalityExperimentResult],
    metric_name: str,
    dataset_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (12, 6),
) -> plt.Figure:
    """
    Plot comparison of a metric across different dimensionalities.
    """
```

```
    # Organize data by algorithm and dimension
    algorithms = sorted(set(r.algorithm_name for r in results))
    dimensions = sorted(set(r.dimension_label for r in results),
                        key=lambda x: (0 if 'Original' in x else int(x.split('-')[1]) if '-
                            ' in x else 999))

    # Create grouped bar chart
    fig, ax = plt.subplots(figsize=figsize)

    x = np.arange(len(algorithms))
    width = 0.8 / len(dimensions)

    for i, dim_label in enumerate(dimensions):
        values = []
        for algo in algorithms:
            result = next((r for r in results
                           if r.algorithm_name == algo and r.dimension_label == dim_label),
                               None)
            if result:
                value = getattr(result.metrics, metric_name, None)
                values.append(value if value is not None else 0)
            else:
                values.append(0)

        offset = (i - len(dimensions)/2 + 0.5) * width
        bars = ax.bar(x + offset, values, width, label=dim_label, alpha=0.8)

    ax.set_xlabel('Algorithm', fontsize=12)
    ax.set_ylabel(metric_name.replace('_', ' ').title(), fontsize=12)
    ax.set_title(f'{metric_name.replace("_"," ").title()} by Dimensionality: {
        dataset_name}',
                 fontsize=12, fontweight='bold')
    ax.set_xticks(x)
    ax.set_xticklabels([a.split('(')[0].strip() for a in algorithms], rotation=45, ha='
        right')
    ax.legend(title='Dimensions')
    ax.grid(True, alpha=0.3, axis='y')

    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def plot_runtime_comparison(
    results: List[DimensionalityExperimentResult],
    dataset_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (12, 6),
) -> plt.Figure:
    """
    Plot runtime comparison across dimensionalities.
    """
    algorithms = sorted(set(r.algorithm_name for r in results))
    dimensions = sorted(set(r.dimension_label for r in results),
                        key=lambda x: (0 if 'Original' in x else int(x.split('-')[1]) if '-
                            ' in x else 999))

    fig, ax = plt.subplots(figsize=figsize)

    x = np.arange(len(algorithms))
```

```
    width = 0.8 / len(dimensions)

    for i, dim_label in enumerate(dimensions):
        values = []
        for algo in algorithms:
            result = next((r for r in results
                           if r.algorithm_name == algo and r.dimension_label == dim_label),
                          None)
            values.append(result.runtime_seconds * 1000 if result else 0)  # Convert to ms

        offset = (i - len(dimensions)/2 + 0.5) * width
        bars = ax.bar(x + offset, values, width, label=dim_label, alpha=0.8)

    ax.set_xlabel('Algorithm', fontsize=12)
    ax.set_ylabel('Runtime (ms)', fontsize=12)
    ax.set_title(f'Runtime Comparison by Dimensionality: {dataset_name}',
                 fontsize=12, fontweight='bold')
    ax.set_xticks(x)
    ax.set_xticklabels([a.split('(')[0].strip() for a in algorithms], rotation=45, ha='right')
    ax.legend(title='Dimensions')
    ax.grid(True, alpha=0.3, axis='y')

    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def plot_comprehensive_summary(
    results: List[DimensionalityExperimentResult],
    dataset_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (16, 12),
) -> plt.Figure:
    """
    Create a comprehensive summary plot with multiple metrics.
    """
    fig, axes = plt.subplots(2, 3, figsize=figsize)

    algorithms = sorted(set(r.algorithm_name for r in results))
    dimensions = sorted(set(r.dimension_label for r in results),
                        key=lambda x: (0 if 'Original' in x else int(x.split('-')[1]) if '-' in x else 999))

    x = np.arange(len(algorithms))
    width = 0.8 / len(dimensions)

    metrics_to_plot = [
        ('ari', 'Adjusted Rand Index (ARI)', True),
        ('nmi', 'Normalized Mutual Information (NMI)', True),
        ('silhouette', 'Silhouette Score', True),
        ('davies_bouldin', 'Davies-Bouldin Index', False),  # Lower is better
        ('runtime_seconds', 'Runtime (ms)', False),
    ]

    for ax_idx, (metric_key, metric_title, higher_is_better) in enumerate(metrics_to_plot):
        ax = axes.flatten()[ax_idx]

        for i, dim_label in enumerate(dimensions):
```

```
            values = []
            for algo in algorithms:
                result = next((r for r in results
                               if r.algorithm_name == algo and r.dimension_label ==
                                   dim_label), None)
                if result:
                    if metric_key == 'runtime_seconds':
                        value = result.runtime_seconds * 1000  # Convert to ms
                    else:
                        value = getattr(result.metrics, metric_key, None)
                    values.append(value if value is not None else 0)
                else:
                    values.append(0)

            offset = (i - len(dimensions)/2 + 0.5) * width
            ax.bar(x + offset, values, width, label=dim_label, alpha=0.8)

        ax.set_xlabel('Algorithm', fontsize=10)
        ax.set_ylabel(metric_title, fontsize=10)
        ax.set_title(metric_title, fontsize=11, fontweight='bold')
        ax.set_xticks(x)
        ax.set_xticklabels([a.split('(')[0].strip()[:10] for a in algorithms],
                           rotation=45, ha='right', fontsize=8)
        ax.grid(True, alpha=0.3, axis='y')
        if ax_idx == 0:
            ax.legend(title='Dimensions', fontsize=8)

    # Hide the last subplot (we have 5 metrics, 6 subplots)
    axes.flatten()[5].set_visible(False)

    plt.suptitle(f'Dimensionality Reduction Impact: {dataset_name}',
                 fontsize=14, fontweight='bold')
    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved: {save_path}")

    return fig


def run_experiment3(
    dataset: ClusteringDataset = None,
    pca_dimensions: List[int] = [10, 20, 30],
    output_dir: Path = Path("outputs/experiment3"),
    random_state: int = 42,
) -> Dict:
    """
    Run Experiment 3: High-Dimensional Data and Dimensionality Reduction.

    Args:
        dataset: Dataset to use (default: Digits dataset)
        pca_dimensions: List of PCA dimensions to test
        output_dir: Directory to save outputs
        random_state: Random seed

    Returns:
        Dictionary containing all results
    """
    print("=" * 70)
    print("Experiment 3: High-Dimensional Data and Dimensionality Reduction")
    print("=" * 70)

    output_dir.mkdir(parents=True, exist_ok=True)
```

```
    # Use Digits dataset if not provided (recommended in README)
    if dataset is None:
        dataset = load_digits_dataset()

    print(f"\nDataset: {dataset.name}")
    print(f"  Samples: {dataset.X.shape[0]}")
    print(f"  Original dimensions: {dataset.X.shape[1]}")
    print(f"  Classes: {dataset.n_clusters}")
    print(f"  {dataset.description}")

    # Standardize data
    X_scaled, _ = standardize_data(dataset.X)

    # ========================================
    # PCA Analysis
    # ========================================
    print("\n[1/4] PCA Explained Variance Analysis")
    print("-" * 50)

    fig = plot_pca_explained_variance(
        X=X_scaled,
        save_path=output_dir / "pca_explained_variance.png",
    )
    plt.close(fig)

    # Show variance explained by chosen dimensions
    for n_dim in pca_dimensions:
        _, _, var_explained = apply_pca(X_scaled, n_dim, random_state)
        print(f"  PCA-{n_dim}: {var_explained:.2%} variance explained")

    # ========================================
    # Run Clustering at Different Dimensionalities
    # ========================================
    print("\n[2/4] Running Clustering at Different Dimensionalities")
    print("-" * 50)

    all_results: List[DimensionalityExperimentResult] = []

    # Prepare data at different dimensionalities
    data_versions = [
        (X_scaled, dataset.X.shape[1], f"Original ({dataset.X.shape[1]})"),
    ]

    for n_dim in pca_dimensions:
        X_pca, _, var = apply_pca(X_scaled, n_dim, random_state)
        data_versions.append((X_pca, n_dim, f"PCA-{n_dim}"))

    # Get algorithms
    algorithms = get_main_algorithms(
        n_clusters=dataset.n_clusters,
        random_state=random_state,
    )

    # Run clustering for each data version and algorithm
    for X_data, n_dim, dim_label in data_versions:
        print(f"\n  {dim_label}:")

        for algo in algorithms:
            # Create fresh algorithm instance (some algorithms cache state)
            algo_fresh = type(algo).__new__(type(algo))
            algo_fresh.__dict__.update(algo.__dict__)

            result = run_clustering_with_timing(
                algorithm=algo_fresh,
```

```
                X=X_data,
                y_true=dataset.y,
                n_dimensions=n_dim,
                dimension_label=dim_label,
            )
            all_results.append(result)

            ari = result.metrics.ari if result.metrics.ari else 0
            print(f"␣␣␣␣{result.algorithm_name}:␣ARI={ari:.4f},␣"
                  f"Runtime={result.runtime_seconds*1000:.2f}ms")

    # ========================================
    # Generate Visualizations
    # ========================================
    print("\n[3/4]␣Generating␣Visualizations")
    print("-" * 50)

    # Comprehensive summary
    fig = plot_comprehensive_summary(
        results=all_results,
        dataset_name=dataset.name,
        save_path=output_dir / "comprehensive_summary.png",
    )
    plt.close(fig)

    # Individual metric comparisons
    for metric in ['ari', 'nmi', 'silhouette']:
        fig = plot_dimensionality_comparison(
            results=all_results,
            metric_name=metric,
            dataset_name=dataset.name,
            save_path=output_dir / f"comparison_{metric}.png",
        )
        plt.close(fig)

    # Runtime comparison
    fig = plot_runtime_comparison(
        results=all_results,
        dataset_name=dataset.name,
        save_path=output_dir / "runtime_comparison.png",
    )
    plt.close(fig)

    # t-SNE visualization (on PCA-reduced data for speed)
    print("\n␣␣Generating␣t-SNE␣visualization...")
    X_pca_30, _, _ = apply_pca(X_scaled, 30, random_state)

    # Get labels from original dimension clustering
    labels_dict = {}
    for result in all_results:
        if "Original" in result.dimension_label:
            # Simplify algorithm name for display
            simple_name = result.algorithm_name.split('(')[0].strip()
            labels_dict[simple_name] = result.labels

    fig = plot_tsne_visualization(
        X=X_pca_30,  # Use PCA-reduced for faster t-SNE
        y_true=dataset.y,
        labels_dict=labels_dict,
        dataset_name=dataset.name,
        save_path=output_dir / "tsne_visualization.png",
        random_state=random_state,
    )
    plt.close(fig)

```

```
    # ======================================
    # Create Summary Tables
    # ======================================
    print("\n[4/4] Creating Summary Tables")
    print("-" * 50)

    # Create detailed results table
    summary_data = []
    for result in all_results:
        row = {
            'Algorithm': result.algorithm_name,
            'Dimensions': result.dimension_label,
            'ARI': f"{result.metrics.ari:.4f}" if result.metrics.ari else "N/A",
            'NMI': f"{result.metrics.nmi:.4f}" if result.metrics.nmi else "N/A",
            'Silhouette': f"{result.metrics.silhouette:.4f}" if result.metrics.silhouette
                else "N/A",
            'Davies-Bouldin': f"{result.metrics.davies_bouldin:.4f}" if result.metrics.
                davies_bouldin else "N/A",
            'Runtime (ms)': f"{result.runtime_seconds * 1000:.2f}",
        }
        summary_data.append(row)

    df_summary = pd.DataFrame(summary_data)
    df_summary.to_csv(output_dir / "experiment3_results.csv", index=False)

    # Create comparison table (algorithms as rows, dimensions as columns for ARI)
    pivot_ari = df_summary.pivot(index='Algorithm', columns='Dimensions', values='ARI')
    pivot_ari.to_csv(output_dir / "ari_by_dimension.csv")

    pivot_runtime = df_summary.pivot(index='Algorithm', columns='Dimensions', values='
        Runtime (ms)')
    pivot_runtime.to_csv(output_dir / "runtime_by_dimension.csv")

    # Print summaries
    print("\n" + "=" * 70)
    print("EXPERIMENT 3 RESULTS SUMMARY")
    print("=" * 70)
    print(df_summary.to_string(index=False))

    print("\n" + "=" * 70)
    print("ARI BY DIMENSIONALITY")
    print("=" * 70)
    print(pivot_ari.to_string())

    # Analysis
    print("\n" + "=" * 70)
    print("ANALYSIS AND OBSERVATIONS")
    print("=" * 70)

    # Find best performing dimension for each algorithm
    print("\ n  Best Dimensionality by Algorithm (ARI):")
    for algo in set(r.algorithm_name for r in all_results):
        algo_results = [r for r in all_results if r.algorithm_name == algo]
        best = max(algo_results, key=lambda r: r.metrics.ari if r.metrics.ari else 0)
        print(f"  {algo}: {best.dimension_label} (ARI={best.metrics.ari:.4f})")

    # Runtime improvement
    print("\ n  Runtime Improvement with Dimensionality Reduction:")
    for algo in set(r.algorithm_name for r in all_results):
        algo_results = [r for r in all_results if r.algorithm_name == algo]
        original = next((r for r in algo_results if "Original" in r.dimension_label), None
            )
        pca_best = min([r for r in algo_results if "PCA" in r.dimension_label],
                       key=lambda r: r.runtime_seconds)
        if original and pca_best:
```

```
            speedup = original.runtime_seconds / pca_best.runtime_seconds
            print(f"  {algo.split('(')[0].strip()}: {speedup:.2f}x faster with {pca_best.
                dimension_label}")

    print("""
    Key Findings:

1. Curse of Dimensionality:
   - High-dimensional data leads to distance concentration
   - Points become equidistant, making clustering harder
   - Dimensionality reduction can recover meaningful structure

2. PCA Benefits:
   - Reduces noise by removing low-variance components
   - Improves computational efficiency
   - May improve clustering quality if signal is in top components

3. PCA Risks:
   - May discard discriminative information in lower components
   - Not all cluster structure aligns with principal components
   - Trade-off between variance explained and cluster separability

4. Algorithm-Specific Observations:
   - K-Means and GMM benefit from noise reduction via PCA
   - Spectral Clustering may be robust to dimensionality with proper kernel
   - DBSCAN requires density parameter re-tuning after PCA
""")

    print(f"\n  All results saved to: {output_dir}/")
    print("  Experiment 3 completed successfully!")

    return {
        'results': all_results,
        'summary': df_summary,
        'pivot_ari': pivot_ari,
        'pivot_runtime': pivot_runtime,
    }
```

A.8. src/metrics.py

```
"""
Evaluation metrics for clustering quality assessment.
"""

import numpy as np
from sklearn.metrics import (
    silhouette_score,
    calinski_harabasz_score,
    davies_bouldin_score,
    adjusted_rand_score,
    normalized_mutual_info_score,
)
from dataclasses import dataclass
from typing import Optional, Dict


@dataclass
class ClusteringMetrics:
    """Container for all clustering evaluation metrics."""
    # Internal metrics (no ground truth required)
    silhouette: Optional[float] = None
    calinski_harabasz: Optional[float] = None
    davies_bouldin: Optional[float] = None

```

```python
    # External metrics (requires ground truth)
    ari: Optional[float] = None  # Adjusted Rand Index
    nmi: Optional[float] = None  # Normalized Mutual Information

    def to_dict(self) -> Dict[str, Optional[float]]:
        """Convert metrics to dictionary."""
        return {
            "Silhouette": self.silhouette,
            "Calinski-Harabasz": self.calinski_harabasz,
            "Davies-Bouldin": self.davies_bouldin,
            "ARI": self.ari,
            "NMI": self.nmi,
        }

    def __str__(self) -> str:
        """Pretty print metrics."""
        lines = []
        if self.silhouette is not None:
            lines.append(f"  Silhouette Score: {self.silhouette:.4f}")
        if self.calinski_harabasz is not None:
            lines.append(f"  Calinski-Harabasz Index: {self.calinski_harabasz:.4f}")
        if self.davies_bouldin is not None:
            lines.append(f"  Davies-Bouldin Index: {self.davies_bouldin:.4f}")
        if self.ari is not None:
            lines.append(f"  Adjusted Rand Index: {self.ari:.4f}")
        if self.nmi is not None:
            lines.append(f"  Normalized Mutual Information: {self.nmi:.4f}")
        return "\n".join(lines)


def compute_internal_metrics(
    X: np.ndarray,
    labels: np.ndarray,
) -> ClusteringMetrics:
    """
    Compute internal clustering metrics (no ground truth required).

    Args:
        X: Feature matrix
        labels: Cluster assignments

    Returns:
        ClusteringMetrics with internal metrics populated
    """
    metrics = ClusteringMetrics()

    # Check if we have valid clustering (at least 2 clusters, not all noise)
    unique_labels = set(labels)
    n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)

    if n_clusters < 2:
        # Cannot compute metrics with less than 2 clusters
        return metrics

    # Filter out noise points for metric computation
    mask = labels != -1
    if mask.sum() < 2:
        return metrics

    X_valid = X[mask]
    labels_valid = labels[mask]

    # Silhouette Score: [-1, 1], higher is better
    try:
        metrics.silhouette = silhouette_score(X_valid, labels_valid)
```

```
    except Exception:
        pass

    # Calinski-Harabasz Index: higher is better (no upper bound)
    try:
        metrics.calinski_harabasz = calinski_harabasz_score(X_valid, labels_valid)
    except Exception:
        pass

    # Davies-Bouldin Index: lower is better (minimum 0)
    try:
        metrics.davies_bouldin = davies_bouldin_score(X_valid, labels_valid)
    except Exception:
        pass

    return metrics


def compute_external_metrics(
    labels_true: np.ndarray,
    labels_pred: np.ndarray,
) -> ClusteringMetrics:
    """
␣␣␣␣Compute␣external␣clustering␣metrics␣(requires␣ground␣truth).

␣␣␣␣Args:
␣␣␣␣␣␣␣␣labels_true:␣Ground␣truth␣labels
␣␣␣␣␣␣␣␣labels_pred:␣Predicted␣cluster␣assignments

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣ClusteringMetrics␣with␣external␣metrics␣populated
␣␣␣␣"""
    metrics = ClusteringMetrics()

    # Filter out noise points (-1 labels) for fair comparison
    mask = labels_pred != -1
    if mask.sum() < 2:
        return metrics

    labels_true_valid = labels_true[mask]
    labels_pred_valid = labels_pred[mask]

    # Adjusted Rand Index: [-1, 1], 1 is perfect, 0 is random
    try:
        metrics.ari = adjusted_rand_score(labels_true_valid, labels_pred_valid)
    except Exception:
        pass

    # Normalized Mutual Information: [0, 1], 1 is perfect
    try:
        metrics.nmi = normalized_mutual_info_score(
            labels_true_valid, labels_pred_valid, average_method="arithmetic"
        )
    except Exception:
        pass

    return metrics


def compute_all_metrics(
    X: np.ndarray,
    labels_pred: np.ndarray,
    labels_true: Optional[np.ndarray] = None,
) -> ClusteringMetrics:
    """
```

```
    Compute all applicable metrics.

    Args:
        X: Feature matrix
        labels_pred: Predicted cluster assignments
        labels_true: Ground truth labels (optional)

    Returns:
        ClusteringMetrics with all applicable metrics populated
    """
    # Compute internal metrics
    internal = compute_internal_metrics(X, labels_pred)

    # Compute external metrics if ground truth is available
    if labels_true is not None:
        external = compute_external_metrics(labels_true, labels_pred)
        internal.ari = external.ari
        internal.nmi = external.nmi

    return internal


def summarize_metrics(
    results: list[tuple[str, ClusteringMetrics]],
) -> str:
    """
    Create a summary table of metrics for multiple algorithms.

    Args:
        results: List of (algorithm_name, metrics) tuples

    Returns:
        Formatted table string
    """
    import pandas as pd

    data = []
    for name, metrics in results:
        row = {"Algorithm": name}
        row.update(metrics.to_dict())
        data.append(row)

    df = pd.DataFrame(data)
    return df.to_string(index=False)


def rank_algorithms(
    results: list[tuple[str, ClusteringMetrics]],
    metric: str = "ari",
) -> list[tuple[str, float]]:
    """
    Rank algorithms by a specific metric.

    Args:
        results: List of (algorithm_name, metrics) tuples
        metric: Metric to rank by ('silhouette', 'ari', 'nmi', etc.)

    Returns:
        Sorted list of (algorithm_name, metric_value) tuples
    """
    rankings = []
    for name, metrics in results:
        value = getattr(metrics, metric, None)
        if value is not None:
            rankings.append((name, value))
```

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

    return rankings
```

A.9. src/visualization.py

```
"""
Visualization utilities for clustering experiments.
"""

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from typing import Optional, List, Tuple, Dict
from pathlib import Path

# Set style for publication-quality figures
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette("husl")


def plot_clustering_comparison(
    X: np.ndarray,
    labels_true: np.ndarray,
    results: List[Tuple[str, np.ndarray]],
    dataset_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = None,
) -> plt.Figure:
    """
    Create a comparison plot of clustering results across algorithms.

    Args:
        X: 2D feature matrix (n_samples, 2)
        labels_true: Ground truth labels
        results: List of (algorithm_name, predicted_labels) tuples
        dataset_name: Name of the dataset for title
        save_path: Path to save figure (optional)
        figsize: Figure size (width, height)

    Returns:
        matplotlib Figure object
    """
    n_algorithms = len(results)
    n_cols = min(3, n_algorithms + 1)  # +1 for ground truth
    n_rows = (n_algorithms + 1 + n_cols - 1) // n_cols

    if figsize is None:
        figsize = (5 * n_cols, 4 * n_rows)

    fig, axes = plt.subplots(n_rows, n_cols, figsize=figsize)
    axes = np.atleast_2d(axes)
    axes = axes.flatten()

    # Color palette
    colors = plt.cm.tab10(np.linspace(0, 1, 10))

    def plot_scatter(ax, X, labels, title):
        unique_labels = np.unique(labels)
```

```
        for label in unique_labels:
            mask = labels == label
            if label == -1:
                # Noise points in DBSCAN
                ax.scatter(X[mask, 0], X[mask, 1], c='gray', marker='x',
                           s=20, alpha=0.5, label='Noise')
            else:
                color = colors[label % len(colors)]
                ax.scatter(X[mask, 0], X[mask, 1], c=[color],
                           s=30, alpha=0.7, edgecolors='white', linewidth=0.5)
        ax.set_title(title, fontsize=12, fontweight='bold')
        ax.set_xlabel('Feature 1')
        ax.set_ylabel('Feature 2')

    # Plot ground truth
    plot_scatter(axes[0], X, labels_true, f"{dataset_name}\n(Ground Truth)")

    # Plot each algorithm's results
    for idx, (name, labels) in enumerate(results, start=1):
        plot_scatter(axes[idx], X, labels, name)

    # Hide unused subplots
    for idx in range(len(results) + 1, len(axes)):
        axes[idx].set_visible(False)

    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved figure to {save_path}")

    return fig


def plot_metrics_heatmap(
    metrics_data: Dict[str, Dict[str, float]],
    dataset_names: List[str],
    algorithm_names: List[str],
    metric_name: str,
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (10, 6),
) -> plt.Figure:
    """
    Create a heatmap of metric values across datasets and algorithms.

    Args:
        metrics_data: Nested dict {dataset: {algorithm: value}}
        dataset_names: List of dataset names
        algorithm_names: List of algorithm names
        metric_name: Name of the metric for title
        save_path: Path to save figure
        figsize: Figure size

    Returns:
        matplotlib Figure object
    """
    # Create matrix
    matrix = np.zeros((len(dataset_names), len(algorithm_names)))
    for i, dataset in enumerate(dataset_names):
        for j, algo in enumerate(algorithm_names):
            value = metrics_data.get(dataset, {}).get(algo, np.nan)
            matrix[i, j] = value if value is not None else np.nan

    fig, ax = plt.subplots(figsize=figsize)
```

```
    # Determine color map based on metric (DB is lower-is-better)
    cmap = 'RdYlGn' if metric_name != 'Davies-Bouldin' else 'RdYlGn_r'

    im = ax.imshow(matrix, cmap=cmap, aspect='auto')

    # Add colorbar
    cbar = ax.figure.colorbar(im, ax=ax)
    cbar.ax.set_ylabel(metric_name, rotation=-90, va="bottom")

    # Set ticks
    ax.set_xticks(np.arange(len(algorithm_names)))
    ax.set_yticks(np.arange(len(dataset_names)))
    ax.set_xticklabels(algorithm_names, rotation=45, ha='right')
    ax.set_yticklabels(dataset_names)

    # Add text annotations
    for i in range(len(dataset_names)):
        for j in range(len(algorithm_names)):
            value = matrix[i, j]
            if not np.isnan(value):
                text = ax.text(j, i, f'{value:.3f}',
                               ha='center', va='center',
                               color='black' if 0.3 < (value - np.nanmin(matrix)) /
                                     (np.nanmax(matrix) - np.nanmin(matrix) + 1e-10) < 0.7
                               else 'white',
                               fontsize=9)

    ax.set_title(f'{metric_name} Comparison', fontsize=14, fontweight='bold')
    plt.tight_layout()

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved figure to {save_path}")

    return fig


def plot_metrics_table(
    data: List[Dict],
    title: str = "Clustering Results",
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (12, 8),
) -> plt.Figure:
    """
    Create a visual table of metrics.

    Args:
        data: List of dicts with metric data
        title: Table title
        save_path: Path to save figure
        figsize: Figure size

    Returns:
        matplotlib Figure object
    """
    import pandas as pd

    df = pd.DataFrame(data)

    fig, ax = plt.subplots(figsize=figsize)
    ax.axis('tight')
    ax.axis('off')

```

```
    table = ax.table(
        cellText=df.values,
        colLabels=df.columns,
        cellLoc='center',
        loc='center',
    )
    table.auto_set_font_size(False)
    table.set_fontsize(9)
    table.scale(1.2, 1.5)

    # Style header
    for i, key in enumerate(df.columns):
        table[(0, i)].set_facecolor('#4472C4')
        table[(0, i)].set_text_props(color='white', fontweight='bold')

    ax.set_title(title, fontsize=14, fontweight='bold', pad=20)

    if save_path:
        save_path.parent.mkdir(parents=True, exist_ok=True)
        fig.savefig(save_path, dpi=150, bbox_inches='tight')
        print(f"Saved figure to {save_path}")

    return fig


def plot_all_datasets_grid(
    datasets_results: Dict[str, Tuple[np.ndarray, np.ndarray, List[Tuple[str, np.ndarray
        ]]]],
    save_path: Optional[Path] = None,
    figsize: Tuple[int, int] = (20, 16),
) -> plt.Figure:
    """
    Create a comprehensive grid showing all datasets and all algorithms.

    Args:
        datasets_results: Dict mapping dataset_name to (X, y_true, [(algo_name, labels), ...])
        save_path: Path to save figure
        figsize: Figure size

    Returns:
        matplotlib Figure object
    """
    n_datasets = len(datasets_results)
    dataset_names = list(datasets_results.keys())

    # Get algorithm names from first dataset
    first_data = list(datasets_results.values())[0]
    algorithm_names = ['Ground Truth'] + [name for name, _ in first_data[2]]
    n_algorithms = len(algorithm_names)

    fig, axes = plt.subplots(n_datasets, n_algorithms, figsize=figsize)

    colors = plt.cm.tab10(np.linspace(0, 1, 10))

    for row_idx, (dataset_name, (X, y_true, results)) in enumerate(datasets_results.items
        ()):
        # Plot ground truth in first column
        ax = axes[row_idx, 0]
        for label in np.unique(y_true):
            mask = y_true == label
            ax.scatter(X[mask, 0], X[mask, 1], c=[colors[label % len(colors)]],
                       s=15, alpha=0.7)
        if row_idx == 0:
            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
```