Unsupervised Learning - Advanced Handout

Machine Learning for Smarter Innovation

1 Unsupervised Learning - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations, production considerations)

1.1 Mathematical Foundations

1.1.1 Unsupervised Learning Framework

Unsupervised learning seeks to discover structure in data without labeled examples. Given observations $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$ drawn i.i.d. from an unknown distribution $P(X)$, the goal is to learn useful representations or groupings.

Key Tasks: - **Clustering**: Partition data into groups with similar characteristics - **Dimensionality Reduction**: Find lower-dimensional representations preserving relevant structure - **Density Estimation**: Model the underlying probability distribution $P(X)$ - **Anomaly Detection**: Identify observations deviating from expected patterns

Unlike supervised learning with a clear loss function, unsupervised learning requires defining what "structure" means - often through objectives like minimizing within-cluster variance, maximizing likelihood, or preserving pairwise distances.

1.1.2 K-Means: Optimization Perspective

K-means minimizes the within-cluster sum of squares (WCSS):

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2 = \sum_{i=1}^{n} \sum_{k=1}^{K} r_{ik} \|x_i - \mu_k\|^2$$

where $r_{ik} \in \{0, 1\}$ indicates cluster assignment and $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$ is the centroid.

Lloyd's Algorithm (EM interpretation):

1. **Initialize** centroids $\mu_1, \ldots, \mu_K$ (randomly or via K-means++)
2. **E-step (Assignment)**: For each point, assign to nearest centroid:

$$r_{ik} = \begin{cases} 1 & \text{if } k =_j \|x_i - \mu_j\|^2 \\ 0 & \text{otherwise} \end{cases}$$

3. **M-step (Update)**: Recompute centroids:

$$\mu_k = \frac{\sum_{i=1}^{n} r_{ik} x_i}{\sum_{i=1}^{n} r_{ik}}$$

4. **Repeat** until convergence (assignments unchanged)

Convergence Properties: - **Theorem**: Lloyd's algorithm is guaranteed to converge in finite iterations - **Proof sketch**: Each step decreases or maintains J; finite number of partitions implies termination - **Limitation**: Converges to local minimum, not necessarily global

K-means++ Initialization:

The initialization critically affects final solution quality. K-means++ provides a principled approach:

1. Choose first centroid uniformly at random from data
2. For each subsequent centroid, select x_i with probability:

$$P(x_i) = \frac{D(x_i)^2}{\sum_j D(x_j)^2}$$

   where $D(x_i)$ is distance to nearest existing centroid
3. Repeat until K centroids selected

Theorem (Arthur & Vassilvitskii, 2007): K-means++ achieves expected approximation ratio $O(\log K)$ to optimal solution.

1.1.3 Gaussian Mixture Models

GMMs generalize K-means to probabilistic soft clustering:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

where π_k are mixing coefficients ($\sum_k \pi_k = 1$, $\pi_k \geq 0$).

Expectation-Maximization Algorithm:

E-step: Compute responsibilities (posterior probabilities):

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)}$$

M-step: Update parameters:

$$N_k = \sum_{i=1}^{n} \gamma_{ik}$$

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik} x_i$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik} (x_i - \mu_k)(x_i - \mu_k)^T$$

$$\pi_k = \frac{N_k}{n}$$

Log-Likelihood:

$$\ell(\theta) = \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k) \right)$$

EM is guaranteed to increase $\ell(\theta)$ at each iteration but may converge to local maximum.

Model Selection: Use BIC (Bayesian Information Criterion) to select K:

$$\text{BIC} = -2\ell(\hat{\theta}) + p \log n$$

where p is the number of free parameters. Choose K minimizing BIC.

1.1.4 DBSCAN: Density-Based Clustering

DBSCAN discovers clusters as dense regions separated by sparse regions.

Formal Definitions:

ϵ-neighborhood: $N_\epsilon(p) = \{q \in \mathcal{D} : d(p, q) \leq \epsilon\}$

Core point: Point p is a core point if $|N_\epsilon(p)| \geq \text{MinPts}$

Directly density-reachable: Point q is directly density-reachable from p if: 1. p is a core point 2. $q \in N_\epsilon(p)$

Density-reachable: Point q is density-reachable from p if there exists a chain $p_1, p_2, \ldots, p_m$ where: - $p_1 = p$, $p_m = q$ - Each p_{i+1} is directly density-reachable from p_i

Density-connected: Points p and q are density-connected if there exists a point o such that both p and q are density-reachable from o.

Cluster: A cluster C is a non-empty subset satisfying: 1. **Maximality**: If $p \in C$ and q is density-reachable from p, then $q \in C$ 2. **Connectivity**: All points in C are mutually density-connected

Complexity: - Naive: $O(n^2)$ for distance computations - With spatial index (KD-tree, ball tree): $O(n \log n)$

Parameter Selection: - **MinPts**: Rule of thumb: $\text{MinPts} \geq d + 1$ where d is dimensionality; often $\text{MinPts} = 2d$ - ϵ: Examine k-distance graph (sorted distance to k-th nearest neighbor); look for "elbow"

1.1.5 Hierarchical Clustering Theory

Hierarchical clustering builds a tree (dendrogram) of nested clusters.

Agglomerative Approach (bottom-up): 1. Start with each point as singleton cluster 2. Repeatedly merge closest pair of clusters 3. Continue until single cluster remains

Linkage Functions:

Linkage	Formula	Properties
Single	$d(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$	Chaining effect, detects non-convex
Complete	$d(C_i, C_j) = \max_{x \in C_i, y \in C_j} d(x, y)$	Compact clusters, sensitive to outliers

() *	() *	() *
0.3	0.3	0.3
() *	() *	() *
0.3	0.3	0.3
er-age	$\frac{1}{\lvert C_i\rvert\lvert C_j\rvert}\sum_{x,y} d(x,y)$	UP-GMA
() *	() *	() *
0.3	0.3	0.3
	$\frac{\lvert C_i\rvert\lvert C_j\rvert}{\lvert C_i\rvert+\lvert C_j\rvert}\lVert\mu_i-\mu_j\rVert^2$	vari-ance in-crease

Lance-Williams Formula: General update for merged cluster distance:

$$d(C_k, C_i \cup C_j) = \alpha_i d(C_k, C_i) + \alpha_j d(C_k, C_j) + \beta d(C_i, C_j) + \gamma |d(C_k, C_i) - d(C_k, C_j)|$$

Different linkages correspond to different α, β, γ parameters.

Cophenetic Correlation: Measures how faithfully the dendrogram preserves pairwise distances:

$$r_c = \frac{\sum_{i<j}(d_{ij} - \bar{d})(t_{ij} - \bar{t})}{\sqrt{\sum_{i<j}(d_{ij} - \bar{d})^2 \sum_{i<j}(t_{ij} - \bar{t})^2}}$$

where d_{ij} is original distance and t_{ij} is cophenetic distance (height where i and j merge).

1.2 Principal Component Analysis

1.2.1 Mathematical Formulation

Given centered data matrix $X \in \mathbb{R}^{n \times d}$ (columns have zero mean):

Sample Covariance Matrix: $S = \frac{1}{n-1} X^T X$

Eigendecomposition: $S = V \Lambda V^T$ where: - $V = [v_1, v_2, \ldots, v_d]$ contains orthonormal eigenvectors - $\Lambda = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_d)$ with $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$

Principal Components: The k-th principal component is $z_k = X v_k$

Projection: Lower-dimensional representation using top k components:

$$Z = X V_k \in \mathbb{R}^{n \times k}$$

1.2.2 Variance Maximization Interpretation

Theorem: The first principal component v_1 maximizes variance of projected data:

$$v_1 =_{\|v\|=1} \text{Var}(Xv) =_{\|v\|=1} v^T S v$$

Proof: By Lagrangian with constraint $\|v\|^2 = 1$:

$$\mathcal{L}(v, \lambda) = v^T S v - \lambda(v^T v - 1)$$

$$\nabla_v \mathcal{L} = 2Sv - 2\lambda v = 0$$
$$Sv = \lambda v$$

Thus v must be an eigenvector of S. The variance is $v^T S v = \lambda v^T v = \lambda$, maximized by the largest eigenvalue.

1.2.3 Reconstruction Error Interpretation

PCA also minimizes reconstruction error:

$$\min_{V_k} \sum_{i=1}^{n} \|x_i - V_k V_k^T x_i\|^2$$

The optimal V_k consists of the top k eigenvectors of S.

1.2.4 Probabilistic PCA

Latent variable model:

$$z \sim \mathcal{N}(0, I_k)$$
$$x|z \sim \mathcal{N}(Wz + \mu, \sigma^2 I_d)$$

Maximum likelihood yields:

$$W_{ML} = U_k(\Lambda_k - \sigma^2 I)^{1/2} R$$

where U_k contains top k eigenvectors, Λ_k their eigenvalues, and R is arbitrary rotation.

Advantages: - Handles missing data via EM - Provides likelihood for model comparison - Connection to factor analysis

---

1.3 Advanced Dimensionality Reduction

1.3.1 t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE preserves local structure by matching probability distributions.

High-dimensional similarities (Gaussian):

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2/2\sigma_i^2)}$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

Low-dimensional similarities (Student-t with df=1):

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}}$$

Objective (KL divergence):

$$C = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

Gradient:

$$\frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

Perplexity: Controls effective number of neighbors; σ_i is set so that perplexity of P_i matches target:

$$\text{Perp}(P_i) = 2^{H(P_i)} = 2^{-\sum_j p_{j|i} \log_2 p_{j|i}}$$

Key Properties: - Heavy-tailed q_{ij} prevents crowding problem - Non-convex optimization; results vary with initialization - Distances in embedding not interpretable - Good for visualization, not for downstream ML

1.3.2 UMAP (Uniform Manifold Approximation and Projection)

UMAP provides theoretical grounding via topological data analysis.

Key differences from t-SNE: - Uses fuzzy simplicial sets instead of Gaussians - Different normalization (local vs global) - Cross-entropy loss instead of KL divergence - Faster computation via approximate nearest neighbors

High-dimensional graph: Fuzzy union of local fuzzy simplicial sets

$$\mu(x_i, x_j) = \exp\left(-\frac{d(x_i, x_j) - \rho_i}{\sigma_i}\right)$$

where ρ_i is distance to nearest neighbor.

Loss function:

$$\sum_{e \in E} \mu(e) \log \frac{\mu(e)}{\nu(e)} + (1 - \mu(e)) \log \frac{1 - \mu(e)}{1 - \nu(e)}$$

where ν is the low-dimensional membership strength.

1.4 Cluster Validation Theory

1.4.1 Internal Validation Metrics

Silhouette Coefficient:

For point i in cluster C_I: - $a(i) = \frac{1}{|C_I|-1} \sum_{j \in C_I, j \neq i} d(i,j)$ (mean intra-cluster distance) - $b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i,j)$ (mean nearest-cluster distance)

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \in [-1, 1]$$

Average silhouette: $\bar{s} = \frac{1}{n} \sum_{i=1}^{n} s(i)$

Calinski-Harabasz Index (variance ratio criterion):

$$CH = \frac{SS_B / (K-1)}{SS_W / (n-K)}$$

where: - $SS_B = \sum_{k=1}^{K} n_k \|\mu_k - \bar{x}\|^2$ (between-cluster variance) - $SS_W = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$ (within-cluster variance)

Higher is better; penalizes many small clusters.

Davies-Bouldin Index:

$$DB = \frac{1}{K} \sum_{i=1}^{K} \max_{j \neq i} \frac{s_i + s_j}{d_{ij}}$$

where s_i is average distance within cluster i and d_{ij} is centroid distance. Lower is better.

1.4.2 Gap Statistic

Compare clustering quality to null reference:

$$\text{Gap}_K = \mathbb{E}^*[\log W_K] - \log W_K$$

where $W_K = \sum_{k=1}^{K} \frac{1}{2n_k} \sum_{x_i, x_j \in C_k} d(x_i, x_j)^2$ and expectation is over reference distribution.

Procedure: 1. Cluster data for $K = 1, 2, \ldots, K_{max}$ 2. Generate B reference datasets (uniform in bounding box) 3. Compute Gap_K and standard error s_K 4. Choose smallest K such that $\text{Gap}_K \geq \text{Gap}_{K+1} - s_{K+1}$

1.4.3 External Validation (when labels available)

Adjusted Rand Index:

$$ARI = \frac{RI - E[RI]}{\max(RI) - E[RI]}$$

Corrects for chance; ranges from -1 to 1.

Normalized Mutual Information:

$$NMI = \frac{2 \cdot I(U; V)}{H(U) + H(V)}$$

where $I(U; V)$ is mutual information and H is entropy.

1.5 Production Implementation

1.5.1 Scalable Clustering Pipeline

```
import numpy as np
from sklearn.base import BaseEstimator, ClusterMixin
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MiniBatchKMeans, KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from typing import Optional, Dict, List, Tuple
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


class ProductionClusteringPipeline(BaseEstimator, ClusterMixin):
    """
    Production-grade clustering pipeline with automatic model selection,
    validation, and monitoring.
    """

    def __init__(
        self,
        k_range: Tuple[int, int] = (2, 10),
        algorithm: str = 'auto',
        validation_metric: str = 'silhouette',
        scale_features: bool = True,
        random_state: int = 42,
        n_init: int = 10,
        batch_size: int = 1000
```

```
    ):
        """
        Initialize clustering pipeline.

        Parameters
        ----------
        k_range : tuple
            Range of cluster numbers to evaluate (min, max)
        algorithm : str, {'auto', 'kmeans', 'minibatch_kmeans'}
            Clustering algorithm
        validation_metric : str, {'silhouette', 'calinski_harabasz', 'gap'}
            Metric for selecting optimal k
        scale_features : bool
            Whether to standardize features
        random_state : int
            Random seed for reproducibility
        n_init : int
            Number of initializations for k-means
        batch_size : int
            Batch size for mini-batch k-means
        """
        self.k_range = k_range
        self.algorithm = algorithm
        self.validation_metric = validation_metric
        self.scale_features = scale_features
        self.random_state = random_state
        self.n_init = n_init
        self.batch_size = batch_size

    def _select_algorithm(self, n_samples: int):
        """Select algorithm based on data size."""
        if self.algorithm == 'auto':
            if n_samples > 10000:
                return 'minibatch_kmeans'
            else:
                return 'kmeans'
        return self.algorithm

    def _compute_gap_statistic(
        self, X: np.ndarray, k_range: range, n_refs: int = 10
    ) -> Tuple[List[float], List[float]]:
        """Compute gap statistic for range of k values."""
        gaps = []
        stds = []

        for k in k_range:
            # Actual clustering
            km = KMeans(n_clusters=k, random_state=self.random_state, n_init=
    self.n_init)
            km.fit(X)
            wk = km.inertia_

            # Reference datasets
            ref_wks = []
            for _ in range(n_refs):
                X_ref = np.random.uniform(
                    X.min(axis=0), X.max(axis=0), X.shape
                )
                km_ref = KMeans(
                    n_clusters=k, random_state=self.random_state, n_init=self.
    n_init
                )
                km_ref.fit(X_ref)
```

```
                ref_wks.append(np.log(km_ref.inertia_))

            gap = np.mean(ref_wks) - np.log(wk)
            std = np.std(ref_wks) * np.sqrt(1 + 1/n_refs)

            gaps.append(gap)
            stds.append(std)

        return gaps, stds

    def _evaluate_k(self, X: np.ndarray, k: int) -> Dict[str, float]:
        """Evaluate clustering for specific k."""
        algorithm = self._select_algorithm(len(X))

        if algorithm == 'minibatch_kmeans':
            model = MiniBatchKMeans(
                n_clusters=k,
                random_state=self.random_state,
                n_init=self.n_init,
                batch_size=self.batch_size
            )
        else:
            model = KMeans(
                n_clusters=k,
                random_state=self.random_state,
                n_init=self.n_init
            )

        labels = model.fit_predict(X)

        # Compute validation metrics
        metrics = {
            'k': k,
            'inertia': model.inertia_,
            'silhouette': silhouette_score(X, labels) if k > 1 else 0,
            'calinski_harabasz': calinski_harabasz_score(X, labels) if k > 1
    else 0
        }

        return metrics, model, labels

    def fit(self, X: np.ndarray, y=None):
        """
        Fit clustering pipeline with automatic k selection.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            Training data

        Returns
        -------
        self
        """
        n_samples, n_features = X.shape
        logger.info(f"Fitting clustering pipeline: {n_samples} samples, {
    n_features} features")

        # Scale features if requested
        if self.scale_features:
            self.scaler_ = StandardScaler()
            X_scaled = self.scaler_.fit_transform(X)
        else:
```

```
            self.scaler_ = None
            X_scaled = X

        # Evaluate different k values
        k_range = range(self.k_range[0], self.k_range[1] + 1)
        self.evaluation_results_ = []

        for k in k_range:
            metrics, model, labels = self._evaluate_k(X_scaled, k)
            self.evaluation_results_.append({
                'metrics': metrics,
                'model': model,
                'labels': labels
            })
            logger.info(f"k={k}: silhouette={metrics['silhouette']:.3f}, "
                        f"CH={metrics['calinski_harabasz']:.1f}")

        # Select optimal k
        if self.validation_metric == 'silhouette':
            scores = [r['metrics']['silhouette'] for r in self.
    evaluation_results_]
            best_idx = np.argmax(scores)
        elif self.validation_metric == 'calinski_harabasz':
            scores = [r['metrics']['calinski_harabasz'] for r in self.
    evaluation_results_]
            best_idx = np.argmax(scores)
        elif self.validation_metric == 'gap':
            gaps, stds = self._compute_gap_statistic(X_scaled, k_range)
            # Gap statistic criterion
            for i in range(len(gaps) - 1):
                if gaps[i] >= gaps[i + 1] - stds[i + 1]:
                    best_idx = i
                    break
            else:
                best_idx = len(gaps) - 1
        else:
            raise ValueError(f"Unknown metric: {self.validation_metric}")

        # Store best model
        self.optimal_k_ = self.evaluation_results_[best_idx]['metrics']['k']
        self.model_ = self.evaluation_results_[best_idx]['model']
        self.labels_ = self.evaluation_results_[best_idx]['labels']

        logger.info(f"Selected k={self.optimal_k_}")

        # Compute cluster statistics
        self._compute_cluster_statistics(X_scaled)

        return self

    def _compute_cluster_statistics(self, X: np.ndarray):
        """Compute statistics for each cluster."""
        self.cluster_stats_ = {}

        for k in range(self.optimal_k_):
            mask = self.labels_ == k
            cluster_data = X[mask]

            self.cluster_stats_[k] = {
                'size': mask.sum(),
                'fraction': mask.mean(),
                'centroid': cluster_data.mean(axis=0),
                'std': cluster_data.std(axis=0),
```

```
                'min': cluster_data.min(axis=0),
                'max': cluster_data.max(axis=0)
            }

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict cluster labels for new data."""
        if self.scaler_ is not None:
            X = self.scaler_.transform(X)
        return self.model_.predict(X)

    def transform(self, X: np.ndarray) -> np.ndarray:
        """Transform to cluster-distance space."""
        if self.scaler_ is not None:
            X = self.scaler_.transform(X)
        return self.model_.transform(X)

    def fit_predict(self, X: np.ndarray, y=None) -> np.ndarray:
        """Fit and return cluster labels."""
        self.fit(X)
        return self.labels_

    def get_cluster_summary(self) -> Dict:
        """Get summary of clustering results."""
        return {
            'optimal_k': self.optimal_k_,
            'cluster_sizes': {k: s['size'] for k, s in self.cluster_stats_.
    items()},
            'validation_scores': {
                r['metrics']['k']: {
                    'silhouette': r['metrics']['silhouette'],
                    'calinski_harabasz': r['metrics']['calinski_harabasz']
                }
                for r in self.evaluation_results_
            }
        }


class ClusterMonitor:
    """
    Monitor cluster quality and detect drift in production.
    """

    def __init__(
        self,
        reference_model,
        reference_data: np.ndarray,
        drift_threshold: float = 0.05
    ):
        """
        Initialize cluster monitor.

        Parameters
        ----------
        reference_model : fitted clustering model
            Baseline model to compare against
        reference_data : array-like
            Reference dataset for drift detection
        drift_threshold : float
            P-value threshold for drift detection
        """
        self.reference_model = reference_model
        self.reference_data = reference_data
        self.drift_threshold = drift_threshold
```

```
        # Compute reference statistics
        self._compute_reference_stats()

    def _compute_reference_stats(self):
        """Compute reference distribution statistics."""
        self.reference_labels = self.reference_model.predict(self.
    reference_data)
        self.reference_distribution = np.bincount(
            self.reference_labels,
            minlength=self.reference_model.n_clusters
        ) / len(self.reference_labels)

        # Feature distributions per cluster
        self.reference_feature_stats = {}
        for k in range(self.reference_model.n_clusters):
            mask = self.reference_labels == k
            self.reference_feature_stats[k] = {
                'mean': self.reference_data[mask].mean(axis=0),
                'std': self.reference_data[mask].std(axis=0)
            }

    def check_distribution_drift(self, new_data: np.ndarray) -> Dict:
        """
        Check if cluster distribution has drifted.

        Uses chi-squared test to compare distributions.
        """
        from scipy import stats

        new_labels = self.reference_model.predict(new_data)
        new_distribution = np.bincount(
            new_labels,
            minlength=self.reference_model.n_clusters
        ) / len(new_labels)

        # Chi-squared test
        expected = self.reference_distribution * len(new_data)
        observed = new_distribution * len(new_data)

        # Avoid division by zero
        mask = expected > 0
        if not mask.all():
            expected = expected[mask]
            observed = observed[mask]

        chi2, p_value = stats.chisquare(observed, expected)

        return {
            'drift_detected': p_value < self.drift_threshold,
            'p_value': p_value,
            'chi2_statistic': chi2,
            'reference_distribution': self.reference_distribution,
            'current_distribution': new_distribution
        }

    def check_feature_drift(self, new_data: np.ndarray) -> Dict:
        """
        Check for feature-level drift within clusters.

        Uses Kolmogorov-Smirnov test per feature per cluster.
        """
        from scipy import stats
```

```
        new_labels = self.reference_model.predict(new_data)
        drift_results = {}

        for k in range(self.reference_model.n_clusters):
            ref_mask = self.reference_labels == k
            new_mask = new_labels == k

            if new_mask.sum() < 10:
                continue

            ref_cluster = self.reference_data[ref_mask]
            new_cluster = new_data[new_mask]

            feature_drifts = []
            for j in range(self.reference_data.shape[1]):
                stat, p_value = stats.ks_2samp(ref_cluster[:, j], new_cluster
    [:, j])
                feature_drifts.append({
                    'feature': j,
                    'ks_statistic': stat,
                    'p_value': p_value,
                    'drift_detected': p_value < self.drift_threshold
                })

            drift_results[k] = feature_drifts

        return drift_results

    def compute_quality_metrics(self, new_data: np.ndarray) -> Dict:
        """Compute clustering quality metrics for new data."""
        new_labels = self.reference_model.predict(new_data)

        return {
            'silhouette': silhouette_score(new_data, new_labels),
            'calinski_harabasz': calinski_harabasz_score(new_data, new_labels)
    ,
            'n_samples': len(new_data),
            'cluster_sizes': dict(zip(*np.unique(new_labels, return_counts=
    True)))
        }


# Usage demonstration
if __name__ == "__main__":
    from sklearn.datasets import make_blobs

    # Generate synthetic data
    X, y_true = make_blobs(n_samples=1000, n_features=10, centers=5,
    random_state=42)

    # Fit pipeline
    pipeline = ProductionClusteringPipeline(
        k_range=(2, 8),
        validation_metric='silhouette',
        scale_features=True
    )
    labels = pipeline.fit_predict(X)

    print("\nClustering Summary:")
    summary = pipeline.get_cluster_summary()
    print(f"Optimal k: {summary['optimal_k']}")
    print(f"Cluster sizes: {summary['cluster_sizes']}")
```

```
    # Setup monitoring
    monitor = ClusterMonitor(
        reference_model=pipeline.model_,
        reference_data=pipeline.scaler_.transform(X)
    )

    # Check new data
    X_new, _ = make_blobs(n_samples=200, n_features=10, centers=5,
    random_state=123)
    X_new_scaled = pipeline.scaler_.transform(X_new)

    drift_result = monitor.check_distribution_drift(X_new_scaled)
    print(f"\nDrift detected: {drift_result['drift_detected']}")
    print(f"P-value: {drift_result['p_value']:.4f}")
```

1.6 Practice Problems

1. **Derive** the EM update equations for GMM with diagonal covariance matrices
2. **Prove** that K-means is a special case of GMM with fixed spherical covariances
3. **Show** that single-linkage hierarchical clustering is equivalent to finding the minimum spanning tree
4. **Derive** the gradient of the t-SNE objective function
5. **Analyze** the computational complexity of different linkage methods
6. **Implement** the gap statistic with proper bootstrap confidence intervals
7. **Compare** UMAP and t-SNE theoretically and empirically on a high-dimensional dataset

---

1.7 References

1. MacQueen, J. (1967). "Some methods for classification and analysis of multivariate observations." *Proceedings of the 5th Berkeley Symposium*.
2. Arthur, D., & Vassilvitskii, S. (2007). "k-means++: The Advantages of Careful Seeding." *SODA*.
3. Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). "A Density-Based Algorithm for Discovering Clusters." *KDD*.
4. Ward, J. H. (1963). "Hierarchical Grouping to Optimize an Objective Function." *Journal of the American Statistical Association*.
5. Tibshirani, R., Walther, G., & Hastie, T. (2001). "Estimating the Number of Clusters in a Data Set via the Gap Statistic." *JRSS-B*.
6. van der Maaten, L., & Hinton, G. (2008). "Visualizing Data using t-SNE." *JMLR*.
7. McInnes, L., Healy, J., & Melville, J. (2018). "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction." *arXiv:1802.03426*.
8. Rousseeuw, P. J. (1987). "Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis." *Journal of Computational and Applied Mathematics*.
9. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. Chapter 9.
10. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer. Chapter 14.

Unsupervised learning reveals hidden structure in data, but interpretation requires domain expertise. Mathematical rigor ensures algorithms behave predictably, while production engineering ensures they scale reliably. Combine both perspectives for robust clustering systems.