Week 1 Handout 3: Advanced Clustering Theory Implementation

Machine Learning for Smarter Innovation

1 Week 1 Handout 3: Advanced Clustering Theory & Implementation

Target Audience: ML engineers and researchers **Duration**: 60 minutes reading + mathematical analysis **Level**: Advanced

---

1.1 Mathematical Foundations

1.1.1 K-Means Optimization Objective

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

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

Where: - k = number of clusters - C_i = cluster i - μ_i = centroid of cluster i - $||x - \mu_i||^2$ = squared Euclidean distance

1.1.2 Lloyd's Algorithm Convergence

Theorem: Lloyd's algorithm converges to a local minimum of the WCSS objective function.

Proof sketch: 1. Each assignment step reduces or maintains WCSS 2. Each update step reduces or maintains WCSS 3. WCSS is bounded below (0) 4. Therefore, algorithm converges to local minimum

Convergence rate: $O(n^{dk+1} \log n)$ worst case, but typically much faster in practice.

1.1.3 Distance Metrics and Their Properties

1. Euclidean Distance (L2 norm)

$$d(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

Properties: - Metric space properties: positive definite, symmetric, triangle inequality - Sensitive to scale and outliers - Assumes spherical clusters

2. Manhattan Distance (L1 norm)

$$d(x, y) = \sum_{i=1}^{d} |x_i - y_i|$$

Properties: - More robust to outliers than Euclidean - Better for high-dimensional sparse data - Creates diamond-shaped decision boundaries

3. Cosine Distance

$$d(x, y) = 1 - \frac{x \cdot y}{||x|| \cdot ||y||}$$

Properties: - Invariant to scaling - Measures angle between vectors - Ideal for text and high-dimensional data

---

1.2 Advanced Algorithmic Considerations

1.2.1 1. Initialization Strategies

K-means++ Initialization **Algorithm**: 1. Choose first centroid c_1 uniformly at random 2. For each subsequent centroid c_i: - Choose point x with probability $\propto D(x)^2$ - Where $D(x)$ = distance to nearest existing centroid

Theoretical Guarantee: K-means++ provides $O(\log k)$-approximation to optimal clustering in expectation.

```python
def kmeans_plus_plus_init(X, k, random_state=None):
    """Implement K-means++ initialization"""
    np.random.seed(random_state)
    n_samples, n_features = X.shape

    # Choose first center randomly
    centers = np.zeros((k, n_features))
    centers[0] = X[np.random.randint(n_samples)]

    for i in range(1, k):
        # Calculate distances to nearest center
        distances = np.array([min([np.linalg.norm(x - c)**2 for c in centers[:
   i]]) for x in X])

        # Choose next center with probability proportional to squared distance
        probabilities = distances / distances.sum()
        cumulative_probabilities = probabilities.cumsum()
        r = np.random.rand()

        for j, p in enumerate(cumulative_probabilities):
            if r < p:
                centers[i] = X[j]
                break

    return centers
```

Furthest First Initialization Choose centroids to maximize minimum distance between any two centroids.

Algorithm: 1. Choose first centroid randomly 2. For each subsequent centroid, choose point furthest from all existing centroids

1.3 Clustering Quality Metrics

1.3.1 1. Silhouette Analysis

For point i in cluster C_I:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

Where: - $a(i)$ = average distance to points in same cluster - $b(i)$ = minimum average distance to points in other clusters

Implementation with computational optimization:

```python
def optimized_silhouette_score(X, labels):
    """Compute silhouette score with optimized distance calculations"""
    n_samples = X.shape[0]
    unique_labels = np.unique(labels)

    # Precompute pairwise distances
    distances = np.zeros((n_samples, n_samples))
    for i in range(n_samples):
        for j in range(i+1, n_samples):
            dist = np.linalg.norm(X[i] - X[j])
            distances[i, j] = distances[j, i] = dist

    silhouette_values = np.zeros(n_samples)

    for i in range(n_samples):
        current_cluster = labels[i]

        # a(i): mean distance to points in same cluster
        same_cluster_indices = np.where(labels == current_cluster)[0]
        if len(same_cluster_indices) > 1:
            a_i = distances[i, same_cluster_indices].mean()
        else:
            a_i = 0

        # b(i): minimum mean distance to points in other clusters
        b_i = np.inf
        for other_cluster in unique_labels:
            if other_cluster != current_cluster:
                other_cluster_indices = np.where(labels == other_cluster)[0]
                if len(other_cluster_indices) > 0:
                    b_cluster = distances[i, other_cluster_indices].mean()
                    b_i = min(b_i, b_cluster)

        # Silhouette value
        if b_i == np.inf:
            silhouette_values[i] = 0
        else:
            silhouette_values[i] = (b_i - a_i) / max(a_i, b_i)

    return silhouette_values.mean()
```

1.3.2 2. Calinski-Harabasz Index

$$CH(k) = \frac{tr(B_k)/(k-1)}{tr(W_k)/(n-k)}$$

Where: - B_k = between-cluster scatter matrix - W_k = within-cluster scatter matrix - Higher values indicate better clustering

1.3.3 3. Davies-Bouldin Index

$$DB = \frac{1}{k}\sum_{i=1}^{k} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)} \right)$$

Where: - σ_i = average distance of points in cluster i to centroid c_i - $d(c_i, c_j)$ = distance between centroids i and j - Lower values indicate better clustering

---

1.4 Theoretical Limitations and Solutions

1.4.1 1. Local Optima Problem

Issue: K-means can converge to suboptimal solutions.

Solutions: - Multiple random initializations - Deterministic initialization (K-means++) - Global optimization approaches (genetic algorithms, simulated annealing)

1.4.2 2. Curse of Dimensionality

Issue: In high dimensions, distance measures become less meaningful.

Mathematical insight: For random vectors in $\mathbb{R}^d$:

$$\lim_{d \to \infty} \frac{d_{max} - d_{min}}{d_{min}} = 0$$

Solutions: - Dimensionality reduction (PCA, t-SNE, UMAP) - Feature selection - Distance metric learning

```python
def concentration_ratio(X, n_samples=1000):
    """Measure distance concentration in high dimensions"""
    n_features = X.shape[1]

    if X.shape[0] < n_samples:
        n_samples = X.shape[0]

    # Sample random pairs
    indices = np.random.choice(X.shape[0], (n_samples, 2), replace=True)
    distances = []

    for i, j in indices:
        if i != j:
            dist = np.linalg.norm(X[i] - X[j])
            distances.append(dist)

    distances = np.array(distances)
    concentration = (distances.max() - distances.min()) / distances.min()

    return concentration, distances.mean(), distances.std()
```

1.4.3 3. Non-Spherical Clusters

Issue: K-means assumes spherical clusters.

Alternative approaches:

Gaussian Mixture Models (GMM)

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

Where Σ_k allows for elliptical clusters.

```python
from sklearn.mixture import GaussianMixture

def compare_kmeans_gmm(X, n_components_range):
    """Compare K-means and GMM performance"""
    results = {}

    for n_comp in n_components_range:
        # K-means
        kmeans = KMeans(n_clusters=n_comp, random_state=42)
        kmeans_labels = kmeans.fit_predict(X)
        kmeans_score = silhouette_score(X, kmeans_labels) if n_comp > 1 else 0

        # GMM
        gmm = GaussianMixture(n_components=n_comp, random_state=42)
        gmm_labels = gmm.fit_predict(X)
        gmm_score = silhouette_score(X, gmm_labels) if n_comp > 1 else 0

        results[n_comp] = {
            ’kmeans_silhouette’: kmeans_score,
            ’gmm_silhouette’: gmm_score,
            ’gmm_aic’: gmm.aic(X),
            ’gmm_bic’: gmm.bic(X)
        }

    return results
```

DBSCAN for Arbitrary Shapes **Core point definition**: Point p is a core point if $|N_\epsilon(p)| \geq \text{minPts}$

Density reachability: Point q is density-reachable from p if there exists a chain of core points connecting them.

Cluster definition: Maximal set of density-connected points.

1.5 Advanced Implementation Techniques

1.5.1 1. Mini-Batch K-Means

For large datasets, use mini-batch gradient descent:

```python
def mini_batch_kmeans(X, k, batch_size=100, max_iter=100, random_state=None):
    """Implement mini-batch K-means for large datasets"""
    np.random.seed(random_state)
    n_samples, n_features = X.shape
    # Initialize centroids
    centroids = X[np.random.choice(n_samples, k, replace=False)]
    centroid_counts = np.ones(k)

    for iteration in range(max_iter):
        # Sample mini-batch
        batch_indices = np.random.choice(n_samples, min(batch_size, n_samples)
    , replace=False)
        batch = X[batch_indices]

        # Assign points to nearest centroids
        distances = np.sqrt(((batch - centroids[:, np.newaxis])**2).sum(axis
    =2))
        assignments = np.argmin(distances, axis=0)

        # Update centroids using moving average
        for i in range(k):
            mask = assignments == i
            if mask.any():
                # Learning rate based on count
                eta = 1.0 / centroid_counts[i]
                centroid_counts[i] += mask.sum()

                # Update centroid
                centroids[i] = (1 - eta) * centroids[i] + eta * batch[mask].
    mean(axis=0)

    return centroids, centroid_counts
```

1.5.2 2. Kernel K-Means

Transform data into higher-dimensional space using kernel trick:

$$\phi : \mathbb{R}^d \to \mathbb{R}^D, \quad D >> d$$

Objective in feature space:

$$\min \sum_{i=1}^{k} \sum_{x \in C_i} ||\phi(x) - \mu_i||^2$$

```
def gaussian_kernel(X, Y, gamma=1.0):
    """Compute Gaussian kernel matrix"""
    sq_dists = np.sum(X**2, axis=1).reshape(-1, 1) + \
               np.sum(Y**2, axis=1) - 2 * np.dot(X, Y.T)
    return np.exp(-gamma * sq_dists)

def kernel_kmeans(X, k, kernel_func, max_iter=100, random_state=None):
    """Implement kernel K-means"""
    np.random.seed(random_state)
    n_samples = X.shape[0]

    # Initialize cluster assignments randomly
    labels = np.random.randint(0, k, n_samples)

    # Precompute kernel matrix
    K = kernel_func(X, X)

    for iteration in range(max_iter):
        old_labels = labels.copy()

        # Update assignments
        for i in range(n_samples):
            best_cluster = 0
            min_distance = float(’inf’)

            for cluster in range(k):
                cluster_mask = labels == cluster
                if cluster_mask.any():
                    # Compute distance in feature space
                    distance = (K[i, i] -
                              2 * K[i, cluster_mask].mean() +
                              K[np.ix_(cluster_mask, cluster_mask)].mean())

                    if distance < min_distance:
                        min_distance = distance
                        best_cluster = cluster

            labels[i] = best_cluster

        # Check convergence
        if np.array_equal(labels, old_labels):
            break

    return labels
```

1.6 Stability Analysis and Model Selection

1.6.1 1. Consensus Clustering

Combine multiple clustering results to find stable partitions:

```
def consensus_clustering(X, k_range, n_iterations=50, sample_fraction=0.8):
    """Perform consensus clustering analysis"""
    n_samples = X.shape[0]
    consensus_matrices = {}

    for k in k_range:
        consensus_matrix = np.zeros((n_samples, n_samples))
        indicator_matrix = np.zeros((n_samples, n_samples))

        for iteration in range(n_iterations):
            # Bootstrap sample
            sample_size = int(n_samples * sample_fraction)
            sample_indices = np.random.choice(n_samples, sample_size, replace=
    False)
            X_sample = X[sample_indices]

            # Cluster sample
            kmeans = KMeans(n_clusters=k, random_state=iteration)
            labels = kmeans.fit_predict(X_sample)

            # Update consensus matrix
            for i in range(sample_size):
                for j in range(i+1, sample_size):
                    original_i, original_j = sample_indices[i], sample_indices
    [j]

                    # Update indicator matrix
                    indicator_matrix[original_i, original_j] += 1
                    indicator_matrix[original_j, original_i] += 1
                    # Update consensus matrix if same cluster
                    if labels[i] == labels[j]:
                        consensus_matrix[original_i, original_j] += 1
                        consensus_matrix[original_j, original_i] += 1

        # Normalize by number of times pairs appeared together
        consensus_matrix = np.divide(consensus_matrix, indicator_matrix,
                                    out=np.zeros_like(consensus_matrix),
                                    where=indicator_matrix!=0)

        consensus_matrices[k] = consensus_matrix

    return consensus_matrices

def stability_score(consensus_matrix):
    """Calculate stability score from consensus matrix"""
    # Remove diagonal (always 1)
    off_diagonal = consensus_matrix[np.triu_indices_from(consensus_matrix, k
   =1)]

    # Stability is proportion of pairs with consensus > 0.5
    stable_pairs = (off_diagonal > 0.5).sum()
    total_pairs = len(off_diagonal)

    return stable_pairs / total_pairs if total_pairs > 0 else 0
```

1.6.2 2. Information-Theoretic Measures

Mutual Information

$$MI(U,V) = \sum_{i,j} P(u_i, v_j) \log \frac{P(u_i, v_j)}{P(u_i)P(v_j)}$$

Adjusted Mutual Information (AMI)

$$AMI(U,V) = \frac{MI(U,V) - E[MI(U,V)]}{max(H(U),H(V)) - E[MI(U,V)]}$$

Where $E[MI(U,V)]$ is the expected mutual information under null hypothesis.

```
from sklearn.metrics import adjusted_mutual_info_score,
   normalized_mutual_info_score

def evaluate_clustering_stability(X, k, n_runs=20):
    """Evaluate clustering stability using multiple metrics"""
    labels_list = []

    # Generate multiple clustering results
    for run in range(n_runs):
        kmeans = KMeans(n_clusters=k, random_state=run)
        labels = kmeans.fit_predict(X)
        labels_list.append(labels)

    # Calculate pairwise AMI scores
    ami_scores = []
    for i in range(n_runs):
        for j in range(i+1, n_runs):
            ami = adjusted_mutual_info_score(labels_list[i], labels_list[j])
            ami_scores.append(ami)
    stability_metrics = {
        ’mean_ami’: np.mean(ami_scores),
        ’std_ami’: np.std(ami_scores),
        ’min_ami’: np.min(ami_scores),
        ’max_ami’: np.max(ami_scores)
    }

    return stability_metrics
```

1.7 Scalability and Production Considerations

1.7.1 1. Distributed K-Means

For massive datasets, implement distributed version:

```
def distributed_kmeans_step(data_partition, centroids):
    """Single step of distributed K-means on data partition"""
    # Assign points to nearest centroids
    distances = np.sqrt(((data_partition - centroids[:, np.newaxis])**2).sum(
   axis=2))
    assignments = np.argmin(distances, axis=0)

    # Compute local statistics
    local_stats = {}
    for k in range(len(centroids)):
        mask = assignments == k
        if mask.any():
            points = data_partition[mask]
            local_stats[k] = {
                ’sum’: points.sum(axis=0),
                ’count’: len(points),
                ’sse’: ((points - centroids[k])**2).sum()
            }
        else:
            local_stats[k] = {
                ’sum’: np.zeros(data_partition.shape[1]),
                ’count’: 0,
                ’sse’: 0
            }

    return local_stats

def merge_local_stats(stats_list):
    """Merge statistics from multiple partitions"""
    n_clusters = len(stats_list[0])
    merged_stats = {}

    for k in range(n_clusters):
        total_sum = np.zeros_like(stats_list[0][k][’sum’])
        total_count = 0
        total_sse = 0

        for stats in stats_list:
            total_sum += stats[k][’sum’]
            total_count += stats[k][’count’]
            total_sse += stats[k][’sse’]

        merged_stats[k] = {
            'centroid': total_sum / total_count if total_count > 0 else
    total_sum,
            'count': total_count,
            'sse': total_sse
        }

    return merged_stats
```

1.7.2 2. Online Clustering

For streaming data, implement online updates:

```
class OnlineKMeans:
    """Online K-means for streaming data"""

    def __init__(self, k, learning_rate=0.1):
        self.k = k
        self.learning_rate = learning_rate
        self.centroids = None
        self.counts = None
        self.initialized = False

    def partial_fit(self, X):
        """Update model with new batch of data"""
        if not self.initialized:
            self._initialize(X)

        # Assign points to nearest centroids
        distances = np.sqrt(((X - self.centroids[:, np.newaxis])**2).sum(axis
    =2))
        assignments = np.argmin(distances, axis=0)

        # Update centroids using exponential moving average
        for i in range(self.k):
            mask = assignments == i
            if mask.any():
                points = X[mask]
                self.counts[i] += len(points)

                # Adaptive learning rate based on count
                adaptive_lr = self.learning_rate / np.sqrt(self.counts[i])

                # Update centroid
                new_centroid = points.mean(axis=0)
                self.centroids[i] = ((1 - adaptive_lr) * self.centroids[i] +
                                   adaptive_lr * new_centroid)

    def _initialize(self, X):
        """Initialize centroids with first batch"""
        n_samples = X.shape[0]
        init_indices = np.random.choice(n_samples, min(self.k, n_samples),
    replace=False)
        self.centroids = X[init_indices].copy()
        self.counts = np.ones(self.k)
        self.initialized = True

    def predict(self, X):
        """Predict cluster labels"""
        distances = np.sqrt(((X - self.centroids[:, np.newaxis])**2).sum(axis
    =2))
        return np.argmin(distances, axis=0)
```

1.8 Research Frontiers and Advanced Topics

1.8.1 1. Deep Clustering

Combine representation learning with clustering:

$$\min_{\theta,Z} \mathcal{L}_{reconstruction} + \lambda \mathcal{L}_{clustering}$$

Where θ are neural network parameters and Z are cluster assignments.

1.8.2 2. Constrained Clustering

Incorporate domain knowledge through constraints: - **Must-link**: Points that must be in same cluster - **Cannot-link**: Points that cannot be in same cluster - **Size constraints**: Minimum/maximum cluster sizes

1.8.3 3. Multi-view Clustering

Cluster data with multiple representations:

$$\min \sum_{v=1}^{V} \alpha_v \mathcal{L}_v + \Omega(Z)$$

Where V is number of views and $\Omega(Z)$ enforces consistency across views.

1.9 Practical Implementation Guidelines

1.9.1 1. Hyperparameter Optimization

```
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import silhouette_score

def optimize_clustering_hyperparameters(X, param_grid):
    """Optimize clustering hyperparameters"""
    best_score = -1
    best_params = None
    results = []

    for params in ParameterGrid(param_grid):
        if params[’algorithm’] == ’kmeans’:
            model = KMeans(**{k: v for k, v in params.items() if k != ’
    algorithm’})
        elif params[’algorithm’] == ’dbscan’:
            model = DBSCAN(**{k: v for k, v in params.items() if k != ’
    algorithm’})

        labels = model.fit_predict(X)

        # Skip if only one cluster or all noise
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        if n_clusters > 1:
            score = silhouette_score(X, labels)
            results.append({**params, 'silhouette_score': score, 'n_clusters':
         n_clusters})

            if score > best_score:
                best_score = score
                best_params = params

    return best_params, best_score, results
```

1.9.2 2. Robust Error Handling

```
def robust_clustering_pipeline(X, algorithms=['kmeans', 'dbscan'], **kwargs):
    """Robust clustering pipeline with error handling"""
    results = {}

    for algorithm in algorithms:
        try:
            if algorithm == 'kmeans':
                model = KMeans(**kwargs.get('kmeans_params', {}))
                labels = model.fit_predict(X)

                # Additional validation
                n_clusters = len(set(labels))
                if n_clusters > 1:
                    score = silhouette_score(X, labels)
                    results[algorithm] = {
                        'labels': labels,
                        'score': score,
                        'n_clusters': n_clusters,
                        'model': model,
                        'status': 'success'
                    }
                else:
                    results[algorithm] = {
                        'status': 'failed',
                        'reason': 'Only one cluster found'
                    }

            elif algorithm == 'dbscan':
                model = DBSCAN(**kwargs.get('dbscan_params', {}))
                labels = model.fit_predict(X)

                n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
                if n_clusters > 1:
                    score = silhouette_score(X, labels)
                    results[algorithm] = {
                        'labels': labels,
                        'score': score,
                        'n_clusters': n_clusters,
                        'n_noise': list(labels).count(-1),
                        'model': model,
                        'status': 'success'
                    }
                else:
                    results[algorithm] = {
                        'status': 'failed',
                        'reason': f'Only {n_clusters} clusters found'
                    }

        except Exception as e:
            results[algorithm] = {
                'status': 'error',
                'error': str(e)
            }

    return results
```

1.10 Conclusion and Future Directions

1.10.1 Key Takeaways:

1. **Mathematical foundation** is crucial for understanding limitations
2. **Initialization matters** significantly for final results
3. **Validation requires multiple metrics** and stability analysis
4. **Scalability considerations** are essential for production systems
5. **Domain knowledge integration** improves practical results

1.10.2 Research Opportunities:

- **Interpretable clustering**: Making results more explainable
- **Temporal clustering**: Handling time-evolving data
- **Fairness in clustering**: Avoiding discriminatory partitions
- **Quantum clustering**: Leveraging quantum computing advantages

This advanced handout provides theoretical depth and practical implementation strategies for sophisticated clustering applications. Master these concepts to become a clustering expert.