Unsupervised Learning - Intermediate Handout

Machine Learning for Smarter Innovation

1 Unsupervised Learning - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.metrics import silhouette_score, calinski_harabasz_score,
    davies_bouldin_score
from sklearn.neighbors import NearestNeighbors
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
import warnings
warnings.filterwarnings('ignore')

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers the complete implementation workflow for unsupervised learning. Unsupervised methods discover hidden patterns in data without labeled outcomes. The three primary tasks are clustering (grouping similar items), dimensionality reduction (compressing features), and anomaly detection (finding outliers). Each requires different algorithms and validation approaches.

1.2 1. Data Preparation for Unsupervised Learning

1.2.1 Concept Overview

Unsupervised algorithms are highly sensitive to feature scales and data quality. Unlike supervised learning where labels can guide the model past minor data issues, unsupervised methods will find whatever patterns exist in the data, including patterns from noise, outliers, or scaling artifacts. Proper preparation is critical.

1.2.2 Creating Realistic Sample Data

```
# Generate synthetic customer data for clustering
np.random.seed(42)

# Create three distinct customer segments
n_samples = 500

# Segment 1: High-value frequent buyers
segment1 = np.random.multivariate_normal(
    mean=[300, 50, 5000],  # avg_order, purchase_freq, total_spend
    cov=[[2000, 100, 50000], [100, 50, 5000], [50000, 5000, 500000]],
    size=n_samples // 3
)

# Segment 2: Budget-conscious regular buyers
segment2 = np.random.multivariate_normal(
    mean=[50, 30, 800],
    cov=[[100, 20, 1000], [20, 30, 500], [1000, 500, 10000]],
    size=n_samples // 3
)

# Segment 3: Occasional big spenders
segment3 = np.random.multivariate_normal(
    mean=[500, 5, 2000],
    cov=[[5000, -50, 20000], [-50, 5, -100], [20000, -100, 100000]],
    size=n_samples // 3
)

# Combine into DataFrame
data = np.vstack([segment1, segment2, segment3])
df = pd.DataFrame(data, columns=['avg_order_value', 'purchase_frequency', '
    total_spend'])
df = df.clip(lower=0)  # Ensure no negative values

print(f"Dataset shape: {df.shape}")
print(f"\nFeature statistics:\n{df.describe().round(2)}")
```

1.2.3 Scaling Strategies

```
# Compare scaling methods
scalers = {
    'StandardScaler': StandardScaler(),
    'MinMaxScaler': MinMaxScaler(),
    'No Scaling': None
}

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
for ax, (name, scaler) in zip(axes, scalers.items()):
    if scaler:
        X_scaled = scaler.fit_transform(df)
    else:
        X_scaled = df.values

    ax.boxplot(X_scaled, labels=df.columns)
    ax.set_title(f'{name}')
    ax.tick_params(axis='x', rotation=45)
plt.tight_layout()
plt.savefig('scaling_comparison.pdf', bbox_inches='tight')
plt.show()
```

```
# Use StandardScaler for clustering (recommended for distance-based methods)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df)
print(f"Scaled data range: [{X_scaled.min():.2f}, {X_scaled.max():.2f}]")
```

1.2.4 Handling Outliers Before Clustering

```
from scipy import stats

# Detect outliers using Z-score method
z_scores = np.abs(stats.zscore(df))
outlier_mask = (z_scores > 3).any(axis=1)
print(f"Outliers detected: {outlier_mask.sum()} ({outlier_mask.mean():.1%})")

# Option 1: Remove outliers
df_clean = df[~outlier_mask].copy()
X_clean = scaler.fit_transform(df_clean)

# Option 2: Winsorize (cap extreme values)
from scipy.stats import mstats
df_winsor = df.copy()
for col in df.columns:
    df_winsor[col] = mstats.winsorize(df[col], limits=[0.01, 0.01])

# For this example, we'll use the original data to demonstrate DBSCAN outlier
    detection
```

1.3 2. K-Means Clustering

1.3.1 Concept Overview

K-Means partitions data into K clusters by iteratively assigning points to the nearest centroid and updating centroid positions. It minimizes within-cluster variance. K-Means is fast and scalable but requires specifying K in advance and assumes spherical, equally-sized clusters.

1.3.2 Basic Implementation

```
# Simple K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
labels = kmeans.fit_predict(X_scaled)

# Examine results
df['cluster'] = labels
print(f"Cluster distribution:\n{df['cluster'].value_counts().sort_index()}")
print(f"\nCluster centers (scaled):\n{kmeans.cluster_centers_.round(3)}")
print(f"\nInertia (within-cluster SS): {kmeans.inertia_:.2f}")
```

1.3.3 Finding Optimal K: Elbow and Silhouette Methods

```
def find_optimal_k(X, k_range=range(2, 11)):
    """
```

```
    Evaluate clustering quality across different K values.
    Returns inertia and silhouette scores for each K.
    """
    results = {'k': [], 'inertia': [], 'silhouette': [], 'calinski': []}

    for k in k_range:
        km = KMeans(n_clusters=k, random_state=42, n_init=10)
        labels = km.fit_predict(X)

        results['k'].append(k)
        results['inertia'].append(km.inertia_)
        results['silhouette'].append(silhouette_score(X, labels))
        results['calinski'].append(calinski_harabasz_score(X, labels))

    return pd.DataFrame(results)

# Run analysis
k_analysis = find_optimal_k(X_scaled)

# Visualize
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Elbow plot
axes[0].plot(k_analysis['k'], k_analysis['inertia'], 'bo-', linewidth=2)
axes[0].set_xlabel('Number of Clusters (K)')
axes[0].set_ylabel('Inertia')
axes[0].set_title('Elbow Method')

# Silhouette plot
axes[1].plot(k_analysis['k'], k_analysis['silhouette'], 'ro-', linewidth=2)
axes[1].set_xlabel('Number of Clusters (K)')
axes[1].set_ylabel('Silhouette Score')
axes[1].set_title('Silhouette Analysis')
axes[1].axhline(y=0.5, color='gray', linestyle='--', alpha=0.5)

# Calinski-Harabasz plot
axes[2].plot(k_analysis['k'], k_analysis['calinski'], 'go-', linewidth=2)
axes[2].set_xlabel('Number of Clusters (K)')
axes[2].set_ylabel('Calinski-Harabasz Score')
axes[2].set_title('Calinski-Harabasz Index')

plt.tight_layout()
plt.savefig('optimal_k_analysis.pdf', bbox_inches='tight')
plt.show()

print(f"\nOptimal K analysis:\n{k_analysis.round(3)}")
```

1.3.4 Silhouette Analysis Per Cluster

```
from sklearn.metrics import silhouette_samples

def plot_silhouette(X, labels, k):
    """Create detailed silhouette plot showing per-cluster quality."""
    silhouette_vals = silhouette_samples(X, labels)

    fig, ax = plt.subplots(figsize=(10, 6))
    y_lower = 10

    for i in range(k):
        cluster_silhouette = silhouette_vals[labels == i]
        cluster_silhouette.sort()
```

```
        y_upper = y_lower + len(cluster_silhouette)
        color = plt.cm.viridis(float(i) / k)
        ax.fill_betweenx(np.arange(y_lower, y_upper), 0, cluster_silhouette,
                         facecolor=color, alpha=0.7)
        ax.text(-0.05, y_lower + 0.5 * len(cluster_silhouette), str(i))
        y_lower = y_upper + 10

    avg_score = silhouette_score(X, labels)
    ax.axvline(x=avg_score, color='red', linestyle='--', label=f'Average: {
    avg_score:.3f}')
    ax.set_xlabel('Silhouette Coefficient')
    ax.set_ylabel('Cluster')
    ax.set_title(f'Silhouette Plot for K={k}')
    ax.legend()
    plt.tight_layout()
    plt.savefig(f'silhouette_k{k}.pdf', bbox_inches='tight')
    plt.show()

# Apply with optimal K
kmeans_final = KMeans(n_clusters=3, random_state=42, n_init=10)
labels_final = kmeans_final.fit_predict(X_scaled)
plot_silhouette(X_scaled, labels_final, 3)
```

1.4 3. DBSCAN (Density-Based Clustering)

1.4.1 Concept Overview

DBSCAN identifies clusters as dense regions separated by sparse areas. Unlike K-Means, it does not require specifying the number of clusters and can find arbitrarily shaped clusters. Points in sparse regions are labeled as noise (outliers). Two key parameters control behavior: eps (neighborhood radius) and min_samples (minimum points to form dense region).

1.4.2 Finding Optimal Epsilon

```
def find_optimal_eps(X, k=5):
    """
    Use k-distance graph to find optimal eps parameter.
    Look for the 'knee' in the sorted distance plot.
    """
    neighbors = NearestNeighbors(n_neighbors=k)
    neighbors.fit(X)
    distances, _ = neighbors.kneighbors(X)
    k_distances = np.sort(distances[:, -1])

    plt.figure(figsize=(10, 5))
    plt.plot(k_distances, linewidth=2)
    plt.xlabel('Points (sorted by distance)')
    plt.ylabel(f'{k}-NN Distance')
    plt.title('K-Distance Graph for Epsilon Selection')
    plt.grid(True, alpha=0.3)

    # Mark potential eps values
    percentiles = [90, 95, 99]
    for p in percentiles:
        eps_val = np.percentile(k_distances, p)
        plt.axhline(y=eps_val, linestyle='--', alpha=0.5,
```

```
                    label=f'{p}th percentile: {eps_val:.2f}')
    plt.legend()
    plt.tight_layout()
    plt.savefig('eps_selection.pdf', bbox_inches='tight')
    plt.show()

    return k_distances

k_distances = find_optimal_eps(X_scaled)
```

1.4.3 DBSCAN Implementation with Parameter Tuning

```
def evaluate_dbscan(X, eps_values, min_samples_values):
    """
    Grid search over DBSCAN parameters.
    Returns DataFrame with results.
    """
    results = []

    for eps in eps_values:
        for min_samples in min_samples_values:
            db = DBSCAN(eps=eps, min_samples=min_samples)
            labels = db.fit_predict(X)

            n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
            n_noise = (labels == -1).sum()
            noise_pct = n_noise / len(labels)

            # Silhouette only meaningful with 2+ clusters and some non-noise
    points
            if n_clusters >= 2 and n_noise < len(labels) - 2:
                # Calculate silhouette excluding noise points
                mask = labels != -1
                sil = silhouette_score(X[mask], labels[mask])
            else:
                sil = np.nan

            results.append({
                'eps': eps,
                'min_samples': min_samples,
                'n_clusters': n_clusters,
                'n_noise': n_noise,
                'noise_pct': noise_pct,
                'silhouette': sil
            })

    return pd.DataFrame(results)

# Parameter grid search
eps_values = [0.3, 0.5, 0.7, 1.0, 1.5]
min_samples_values = [3, 5, 10, 15]

dbscan_results = evaluate_dbscan(X_scaled, eps_values, min_samples_values)
print("DBSCAN Parameter Search Results:")
print(dbscan_results.sort_values('silhouette', ascending=False).head(10).round
    (3))
```

1.4.4 Apply Best DBSCAN Configuration

```
# Select best parameters (balancing clusters, noise, and silhouette)
best_params = dbscan_results.dropna().sort_values('silhouette', ascending=
    False).iloc[0]
print(f"\nBest DBSCAN params: eps={best_params['eps']}, min_samples={int(
    best_params['min_samples'])}")

dbscan = DBSCAN(eps=best_params['eps'], min_samples=int(best_params['
    min_samples']))
db_labels = dbscan.fit_predict(X_scaled)

# Analyze results
unique_labels = set(db_labels)
n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)
n_noise = (db_labels == -1).sum()

print(f"\nDBSCAN Results:")
print(f"  Clusters found: {n_clusters}")
print(f"  Noise points: {n_noise} ({n_noise/len(db_labels):.1%})")
print(f"  Cluster distribution: {pd.Series(db_labels).value_counts().
    sort_index().to_dict()}")

# Visualize with PCA
pca_2d = PCA(n_components=2)
X_2d = pca_2d.fit_transform(X_scaled)

plt.figure(figsize=(10, 6))
scatter = plt.scatter(X_2d[:, 0], X_2d[:, 1], c=db_labels, cmap='viridis',
    alpha=0.7)
plt.colorbar(scatter, label='Cluster (-1 = noise)')
plt.xlabel(f'PC1 ({pca_2d.explained_variance_ratio_[0]:.1%} variance)')
plt.ylabel(f'PC2 ({pca_2d.explained_variance_ratio_[1]:.1%} variance)')
plt.title('DBSCAN Clustering Results')
plt.tight_layout()
plt.savefig('dbscan_results.pdf', bbox_inches='tight')
plt.show()
```

1.5 4. Hierarchical Clustering

1.5.1 Concept Overview

Hierarchical clustering builds a tree of clusters through iterative merging (agglomerative) or splitting (divisive). The dendrogram visualization shows cluster relationships at all levels, allowing flexible choice of cluster count after the fact. Different linkage methods produce different cluster shapes.

1.5.2 Building and Visualizing Dendrograms

```
def plot_dendrogram_with_analysis(X, method='ward', max_display=50):
    """
    Create linkage matrix and plot dendrogram with analysis.
    """
    # Compute linkage
    linked = linkage(X, method=method)

    # Calculate cophenetic correlation (quality measure)
    from scipy.cluster.hierarchy import cophenet
    from scipy.spatial.distance import pdist
```

```
    coph_corr, _ = cophenet(linked, pdist(X))

    # Plot dendrogram
    plt.figure(figsize=(14, 6))
    dendrogram(
        linked,
        truncate_mode='lastp',
        p=max_display,
        leaf_rotation=90,
        leaf_font_size=8,
        show_contracted=True
    )
    plt.title(f'Hierarchical Clustering ({method} linkage)\nCophenetic
    Correlation: {coph_corr:.3f}')
    plt.xlabel('Sample Index or (Cluster Size)')
    plt.ylabel('Distance')
    plt.tight_layout()
    plt.savefig(f'dendrogram_{method}.pdf', bbox_inches='tight')
    plt.show()

    return linked, coph_corr

# Compare linkage methods
linkage_methods = ['ward', 'complete', 'average', 'single']
linkage_results = {}

for method in linkage_methods:
    linked, coph = plot_dendrogram_with_analysis(X_scaled, method=method)
    linkage_results[method] = {'linkage': linked, 'cophenetic': coph}

# Summary of linkage quality
print("\nLinkage Method Comparison (Cophenetic Correlation):")
for method, result in linkage_results.items():
    print(f"  {method}: {result['cophenetic']:.3f}")
```

1.5.3 Cutting the Dendrogram

```
# Use Ward linkage (typically best for compact clusters)
linked = linkage_results['ward']['linkage']

# Method 1: Cut at specific distance threshold
threshold = 5.0
labels_dist = fcluster(linked, t=threshold, criterion='distance')

# Method 2: Cut to get specific number of clusters
n_clusters = 3
labels_maxclust = fcluster(linked, t=n_clusters, criterion='maxclust')

print(f"Cutting at distance {threshold}: {len(set(labels_dist))} clusters")
print(f"Cutting for {n_clusters} clusters: {pd.Series(labels_maxclust).
    value_counts().sort_index().to_dict()}")

# Evaluate hierarchical clustering result
hier_sil = silhouette_score(X_scaled, labels_maxclust)
print(f"Silhouette score (hierarchical, k={n_clusters}): {hier_sil:.3f}")
```

1.5.4 Using AgglomerativeClustering

```
# Scikit-learn's AgglomerativeClustering for larger datasets
from sklearn.cluster import AgglomerativeClustering

agg = AgglomerativeClustering(n_clusters=3, linkage='ward')
agg_labels = agg.fit_predict(X_scaled)

print(f"\nAgglomerativeClustering results:")
print(f"Cluster distribution: {pd.Series(agg_labels).value_counts().sort_index
    ().to_dict()}")
print(f"Silhouette score: {silhouette_score(X_scaled, agg_labels):.3f}")
```

1.6 5. Dimensionality Reduction

1.6.1 PCA for Feature Compression

```
def analyze_pca(X, feature_names=None):
    """
    Perform PCA analysis with detailed component interpretation.
    """
    pca = PCA()
    X_pca = pca.fit_transform(X)

    # Explained variance
    cum_var = np.cumsum(pca.explained_variance_ratio_)

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

    # Scree plot
    axes[0].bar(range(1, len(pca.explained_variance_ratio_)+1),
                pca.explained_variance_ratio_, alpha=0.7, label='Individual')
    axes[0].plot(range(1, len(cum_var)+1), cum_var, 'ro-', label='Cumulative')
    axes[0].axhline(y=0.95, color='gray', linestyle='--', alpha=0.5)
    axes[0].set_xlabel('Principal Component')
    axes[0].set_ylabel('Explained Variance Ratio')
    axes[0].set_title('PCA Variance Explained')
    axes[0].legend()

    # Component loadings
    if feature_names is None:
        feature_names = [f'Feature {i}' for i in range(X.shape[1])]

    loadings = pd.DataFrame(
        pca.components_.T,
        columns=[f'PC{i+1}' for i in range(len(pca.components_))],
        index=feature_names
    )

    # Heatmap of loadings
    import seaborn as sns
    sns.heatmap(loadings, annot=True, fmt='.2f', cmap='RdBu_r', center=0, ax=
    axes[1])
    axes[1].set_title('PCA Component Loadings')

    plt.tight_layout()
    plt.savefig('pca_analysis.pdf', bbox_inches='tight')
    plt.show()

    # Determine optimal components (95% variance)
```

```
    n_components_95 = np.argmax(cum_var >= 0.95) + 1
    print(f"\nComponents for 95% variance: {n_components_95}")
    print(f"\nComponent loadings:\n{loadings.round(3)}")

    return pca, X_pca

pca, X_pca = analyze_pca(X_scaled, feature_names=df.columns[:3])
```

1.6.2 t-SNE for Visualization

```
def visualize_tsne(X, labels, perplexities=[5, 30, 50]):
    """
    Compare t-SNE visualizations with different perplexity values.
    Warning: t-SNE output should ONLY be used for visualization, never for
    clustering.
    """
    fig, axes = plt.subplots(1, len(perplexities), figsize=(5*len(perplexities
    ), 5))

    for ax, perp in zip(axes, perplexities):
        tsne = TSNE(n_components=2, perplexity=perp, random_state=42, n_iter
    =1000)
        X_tsne = tsne.fit_transform(X)

        scatter = ax.scatter(X_tsne[:, 0], X_tsne[:, 1], c=labels,
                             cmap='viridis', alpha=0.7, s=30)
        ax.set_title(f't-SNE (perplexity={perp})')
        ax.set_xlabel('t-SNE 1')
        ax.set_ylabel('t-SNE 2')

    plt.colorbar(scatter, ax=axes[-1], label='Cluster')
    plt.tight_layout()
    plt.savefig('tsne_comparison.pdf', bbox_inches='tight')
    plt.show()

# Visualize K-Means clusters with t-SNE
visualize_tsne(X_scaled, labels_final, perplexities=[10, 30, 50])
```

1.7 6. Cluster Profiling and Interpretation

1.7.1 Statistical Profiling

```
def profile_clusters(df, features, cluster_col='cluster'):
    """
    Generate comprehensive cluster profiles with statistics.
    """
    # Basic statistics per cluster
    profile = df.groupby(cluster_col)[features].agg(['mean', 'std', 'min', '
    max', 'count'])

    # Normalized means for comparison (z-scores relative to overall mean)
    overall_means = df[features].mean()
    overall_stds = df[features].std()

    normalized = df.groupby(cluster_col)[features].mean()
    for col in features:
```

```
        normalized[col] = (normalized[col] - overall_means[col]) /
    overall_stds[col]

    print("Cluster Sizes:")
    print(df[cluster_col].value_counts().sort_index())
    print(f"\nCluster Profiles (Original Scale):")
    print(df.groupby(cluster_col)[features].mean().round(2))
    print(f"\nNormalized Profiles (Z-scores vs overall):")
    print(normalized.round(2))

    return profile, normalized

# Apply to K-Means results
df['cluster'] = labels_final
features = ['avg_order_value', 'purchase_frequency', 'total_spend']
profile, normalized = profile_clusters(df, features)
```

1.7.2 Visualization: Radar Charts

```
def plot_cluster_radar(normalized_profiles, features):
    """
    Create radar chart showing cluster characteristics.
    """
    from math import pi

    categories = features
    N = len(categories)

    # Create angles for radar
    angles = [n / float(N) * 2 * pi for n in range(N)]
    angles += angles[:1]  # Complete the circle

    fig, ax = plt.subplots(figsize=(8, 8), subplot_kw=dict(polar=True))

    colors = plt.cm.viridis(np.linspace(0, 1, len(normalized_profiles)))

    for idx, (cluster, row) in enumerate(normalized_profiles.iterrows()):
        values = row.values.flatten().tolist()
        values += values[:1]
        ax.plot(angles, values, 'o-', linewidth=2, label=f'Cluster {cluster}',
     color=colors[idx])
        ax.fill(angles, values, alpha=0.1, color=colors[idx])

    ax.set_xticks(angles[:-1])
    ax.set_xticklabels(categories)
    ax.set_title('Cluster Profiles (Normalized)')
    ax.legend(loc='upper right', bbox_to_anchor=(1.3, 1.0))
    plt.tight_layout()
    plt.savefig('cluster_radar.pdf', bbox_inches='tight')
    plt.show()

plot_cluster_radar(normalized, features)
```

1.7.3 Parallel Coordinates Visualization

```
from pandas.plotting import parallel_coordinates

def plot_parallel_coordinates(df, features, cluster_col='cluster'):
```

```
    """
    Create parallel coordinates plot for cluster comparison.
    """
    # Normalize features for visualization
    df_plot = df[features + [cluster_col]].copy()
    for col in features:
        df_plot[col] = (df_plot[col] - df_plot[col].mean()) / df_plot[col].std
    ()

    plt.figure(figsize=(12, 6))
    parallel_coordinates(df_plot, cluster_col, colormap='viridis', alpha=0.3)
    plt.title('Cluster Profiles - Parallel Coordinates')
    plt.xlabel('Features')
    plt.ylabel('Normalized Value')
    plt.legend(title='Cluster', loc='upper right')
    plt.tight_layout()
    plt.savefig('parallel_coordinates.pdf', bbox_inches='tight')
    plt.show()

plot_parallel_coordinates(df, features)
```

1.8 7. Comprehensive Cluster Validation

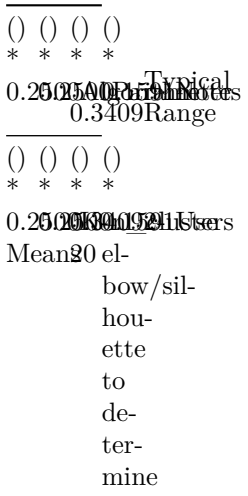
1.8.1 Internal Validation Metrics

```
def evaluate_clustering(X, labels, name='Clustering'):
    """
    Calculate all internal validation metrics.
    """
    # Filter out noise points for DBSCAN
    mask = labels != -1
    X_valid = X[mask]
    labels_valid = labels[mask]

    if len(set(labels_valid)) < 2:
        print(f"{name}: Insufficient clusters for evaluation")
        return None

    metrics = {
        'Silhouette': silhouette_score(X_valid, labels_valid),
        'Calinski-Harabasz': calinski_harabasz_score(X_valid, labels_valid),
        'Davies-Bouldin': davies_bouldin_score(X_valid, labels_valid)
    }

    print(f"\n{name} Validation Metrics:")
    print(f"  Silhouette Score: {metrics['Silhouette']:.3f} (higher is better,
     range -1 to 1)")
    print(f"  Calinski-Harabasz: {metrics['Calinski-Harabasz']:.1f} (higher is
     better)")
    print(f"  Davies-Bouldin: {metrics['Davies-Bouldin']:.3f} (lower is better
    )")

    return metrics

# Compare methods
kmeans_metrics = evaluate_clustering(X_scaled, labels_final, 'K-Means')
dbscan_metrics = evaluate_clustering(X_scaled, db_labels, 'DBSCAN')
hier_metrics = evaluate_clustering(X_scaled, agg_labels, 'Hierarchical')
```

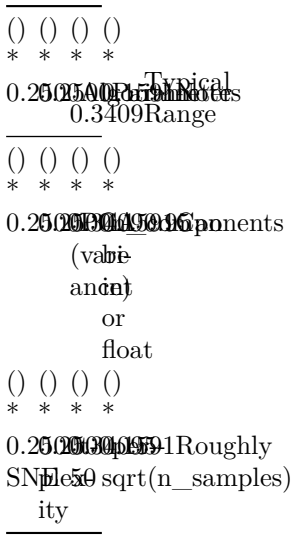
1.8.2 Stability Analysis

```python
def clustering_stability(X, n_clusters=3, n_iterations=10):
    """
    Assess clustering stability through multiple runs.
    """
    from sklearn.metrics import adjusted_rand_score

    all_labels = []
    for i in range(n_iterations):
        km = KMeans(n_clusters=n_clusters, random_state=i, n_init=10)
        all_labels.append(km.fit_predict(X))

    # Calculate pairwise ARI between runs
    ari_scores = []
    for i in range(n_iterations):
        for j in range(i+1, n_iterations):
            ari = adjusted_rand_score(all_labels[i], all_labels[j])
            ari_scores.append(ari)

    print(f"Clustering Stability (K={n_clusters}):")
    print(f"  Mean ARI: {np.mean(ari_scores):.3f}")
    print(f"  Std ARI: {np.std(ari_scores):.3f}")
    print(f"  Min ARI: {np.min(ari_scores):.3f}")

    return np.mean(ari_scores)

stability = clustering_stability(X_scaled, n_clusters=3)
```

1.9 Common Hyperparameters



() *	() *	() *	() *
0.25	5	20	Typical 0.3409 Range
0.25	5	20	More
Means	30	runs = more stable but slower	
0.25	5	20	User-
Means	500	ally converges before limit	
0.25	5	20	Use
SCAN		dependent distance graph	
0.25	5	20	2*
SCAN	20	mensions is common rule	
0.25	5	20	Ward,
er-age		complete,	for
ar-		av-	pact
chi-		er-	clus-
cal		age	ters
0.25	5	20	Al-
er-tan		dependent	threshold
ar-		na-	
chi-		tive	
cal		to n_clusters	



1.10 Practice Projects

1. **Customer Segmentation**: Load an e-commerce dataset, engineer RFM features (Recency, Frequency, Monetary), apply K-Means with elbow analysis, profile segments, and create actionable marketing recommendations for each segment.
2. **Anomaly Detection Pipeline**: Use DBSCAN on transaction data to identify outliers, tune eps using k-distance graphs, analyze characteristics of detected anomalies, and build a simple anomaly scoring function.
3. **Document Clustering**: Convert text documents to TF-IDF vectors, reduce dimensionality with PCA, cluster with multiple algorithms, use t-SNE for visualization, and extract representative terms for each cluster.
4. **Image Compression with PCA**: Load images as flattened arrays, apply PCA with varying component counts, reconstruct images, measure quality vs compression ratio, visualize the tradeoff curve.

1.11 Troubleshooting

()	()	()
*	*	*
0.3	4	2
()	()	()
*	*	*
0.3	4	2
Mean-shift		
gives dif- fer- ent re- sults each run		
()	()	()
*	*	*
0.3	4	2
points in one clus- ter (K-Means)		
()	()	()
*	*	*
0.3	4	2
SCAN la- bels ev- ery- thing as noise		
()	()	()
*	*	*
0.3	4	2
SCAN finds only one clus- ter		
()	()	()
*	*	*
0.3	4	2
hour- ett- scor- neg- a- tive		

Problem	Cause	Solution
Dendrogram too dense	Too many samples	Use truncate_mode='lastp'
t-SNE looks random	Perplexity wrong	Try values 5-50, ensure enough iterations
Memory error	Dataset too large	Use MiniBatchKMeans, subsample data
Features dominate others	Scale differences	Apply StandardScaler before clustering

---

1.12 Next Steps

- Read the advanced handout for mathematical foundations and production deployment
- Experiment with different feature combinations and transformations
- Apply clustering to real business problems with domain expert validation
- Explore ensemble clustering methods for improved stability
- Learn about semi-supervised approaches when partial labels exist

---

Clustering reveals natural structure, but structure must be validated. Good clusters are both statistically coherent and practically meaningful. Always combine metrics with domain expertise.