Clustering - Intermediate Handout

Machine Learning for Smarter Innovation

1 Clustering - 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
import seaborn as sns
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.impute import SimpleImputer
from sklearn.metrics import silhouette_score, calinski_harabasz_score,
    davies_bouldin_score
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import warnings
warnings.filterwarnings('ignore')

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

This handout covers practical clustering implementation for business applications, particularly focused on innovation and customer segmentation contexts. Clustering transforms raw data into actionable segments that drive strategic decisions. The techniques apply to customer segmentation, market analysis, portfolio categorization, and any domain where discovering natural groups provides business value.

1.2 1. Data Preparation for Clustering

1.2.1 Concept Overview

Clustering algorithms are sensitive to data quality, scale, and feature selection. Unlike supervised learning where labels guide the model through noisy data, clustering finds whatever structure exists including patterns from data artifacts. Proper preparation determines whether clusters reflect genuine business segments or statistical noise.

1.2.2 Creating Realistic Innovation Data

```
# Generate synthetic innovation project data
np.random.seed(42)
n_projects = 300

# Create different innovation profiles
def generate_innovation_data(n):
    data = {
        'total_funding': np.concatenate([
            np.random.lognormal(12, 1, n//3),      # High funding
            np.random.lognormal(10, 0.8, n//3),    # Medium funding
            np.random.lognormal(8, 0.5, n//3)      # Low funding
        ]),
        'team_size': np.concatenate([
            np.random.poisson(50, n//3),           # Large teams
            np.random.poisson(15, n//3),           # Medium teams
            np.random.poisson(5, n//3)             # Small teams
        ]),
        'market_size_estimate': np.concatenate([
            np.random.lognormal(22, 1.5, n//3),    # Large markets
            np.random.lognormal(20, 1, n//3),      # Medium markets
            np.random.lognormal(18, 0.8, n//3)     # Niche markets
        ]),
        'patent_count': np.concatenate([
            np.random.poisson(8, n//3),
            np.random.poisson(3, n//3),
            np.random.poisson(1, n//3)
        ]),
        'years_since_founding': np.concatenate([
            np.random.uniform(5, 15, n//3),
            np.random.uniform(2, 7, n//3),
            np.random.uniform(0, 3, n//3)
        ])
    }
    return pd.DataFrame(data)

df = generate_innovation_data(n_projects)
df['team_size'] = df['team_size'].clip(lower=1)

# Add some missing values (realistic scenario)
mask = np.random.choice([True, False], size=df.shape, p=[0.02, 0.98])
df = df.mask(mask)

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

1.2.3 Feature Engineering for Innovation Analysis

```
# Create derived features that capture business meaning
df_engineered = df.copy()

# Efficiency metrics
df_engineered['funding_per_employee'] = df['total_funding'] / df['team_size']
df_engineered['patents_per_year'] = df['patent_count'] / (df['
    years_since_founding'] + 0.5)

# Market metrics
```

```
df_engineered['funding_to_market_ratio'] = df['total_funding'] / df['
    market_size_estimate']

# Log transform skewed features
df_engineered['log_funding'] = np.log1p(df['total_funding'])
df_engineered['log_market_size'] = np.log1p(df['market_size_estimate'])

# Select features for clustering
feature_columns = [
    'log_funding', 'team_size', 'log_market_size',
    'patent_count', 'years_since_founding', 'funding_per_employee'
]

print("Correlation between features:")
print(df_engineered[feature_columns].corr().round(2))
```

1.2.4 Preprocessing Pipeline

```
def preprocess_for_clustering(df, feature_cols, scale_method='standard'):
    """
    Complete preprocessing pipeline for clustering.
    Returns scaled array and fitted transformers.
    """
    X = df[feature_cols].copy()

    # Handle missing values
    imputer = SimpleImputer(strategy='median')
    X_imputed = imputer.fit_transform(X)

    # Handle infinite values
    X_imputed = np.nan_to_num(X_imputed, nan=0, posinf=0, neginf=0)

    # Scale features
    if scale_method == 'standard':
        scaler = StandardScaler()
    elif scale_method == 'minmax':
        scaler = MinMaxScaler()
    else:
        scaler = StandardScaler()

    X_scaled = scaler.fit_transform(X_imputed)

    # Verify preprocessing
    print(f"Preprocessed shape: {X_scaled.shape}")
    print(f"Feature means (should be ~0 for StandardScaler): {X_scaled.mean(
    axis=0).round(3)}")
    print(f"Feature stds (should be ~1 for StandardScaler): {X_scaled.std(axis
    =0).round(3)}")

    return X_scaled, imputer, scaler

X_scaled, imputer, scaler = preprocess_for_clustering(df_engineered,
    feature_columns)
```

1.3 2. K-Means Clustering Implementation

1.3.1 Concept Overview

K-Means is the workhorse algorithm for business clustering. It partitions data into K groups by minimizing within-cluster variance. Fast and interpretable, it works well when clusters are roughly spherical and similarly sized. The main challenge is determining the optimal number of clusters.

1.3.2 Finding Optimal K

```
def comprehensive_k_analysis(X, k_range=range(2, 12)):
    """
    Evaluate multiple metrics across different K values.
    Returns DataFrame with all metrics for comparison.
    """
    results = {
        'k': [], 'inertia': [], 'silhouette': [],
        'calinski_harabasz': [], 'davies_bouldin': []
    }

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

        results['k'].append(k)
        results['inertia'].append(kmeans.inertia_)
        results['silhouette'].append(silhouette_score(X, labels))
        results['calinski_harabasz'].append(calinski_harabasz_score(X, labels)
    )
        results['davies_bouldin'].append(davies_bouldin_score(X, labels))

    return pd.DataFrame(results)

# Run analysis
k_analysis = comprehensive_k_analysis(X_scaled)

# Visualize all metrics
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Elbow plot
axes[0, 0].plot(k_analysis['k'], k_analysis['inertia'], 'bo-', linewidth=2)
axes[0, 0].set_xlabel('Number of Clusters (K)')
axes[0, 0].set_ylabel('Inertia')
axes[0, 0].set_title('Elbow Method')
axes[0, 0].grid(True, alpha=0.3)

# Silhouette
axes[0, 1].plot(k_analysis['k'], k_analysis['silhouette'], 'go-', linewidth=2)
axes[0, 1].set_xlabel('Number of Clusters (K)')
axes[0, 1].set_ylabel('Silhouette Score')
axes[0, 1].set_title('Silhouette Analysis (higher = better)')
axes[0, 1].grid(True, alpha=0.3)

# Calinski-Harabasz
axes[1, 0].plot(k_analysis['k'], k_analysis['calinski_harabasz'], 'ro-',
    linewidth=2)
axes[1, 0].set_xlabel('Number of Clusters (K)')
axes[1, 0].set_ylabel('Calinski-Harabasz Score')
axes[1, 0].set_title('Calinski-Harabasz Index (higher = better)')
axes[1, 0].grid(True, alpha=0.3)

# Davies-Bouldin
```

```
axes[1, 1].plot(k_analysis['k'], k_analysis['davies_bouldin'], 'mo-',
    linewidth=2)
axes[1, 1].set_xlabel('Number of Clusters (K)')
axes[1, 1].set_ylabel('Davies-Bouldin Score')
axes[1, 1].set_title('Davies-Bouldin Index (lower = better)')
axes[1, 1].grid(True, alpha=0.3)

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

print("\nK-Selection Analysis:")
print(k_analysis.round(3))

# Recommend K based on combined metrics
best_silhouette_k = k_analysis.loc[k_analysis['silhouette'].idxmax(), 'k']
best_calinski_k = k_analysis.loc[k_analysis['calinski_harabasz'].idxmax(), 'k'
    ]
print(f"\nRecommended K based on Silhouette: {best_silhouette_k}")
print(f"Recommended K based on Calinski-Harabasz: {best_calinski_k}")
```

1.3.3 Robust K-Means with Stability Testing

```
def robust_kmeans(X, n_clusters, n_runs=10):
    """
    Run K-Means multiple times and select the most stable result.
    Returns best model, score, and stability metrics.
    """
    all_labels = []
    all_scores = []
    best_model = None
    best_score = -1

    for run in range(n_runs):
        kmeans = KMeans(
            n_clusters=n_clusters,
            init='k-means++',
            n_init=10,
            max_iter=300,
            random_state=run
        )
        labels = kmeans.fit_predict(X)
        score = silhouette_score(X, labels)

        all_labels.append(labels)
        all_scores.append(score)

        if score > best_score:
            best_score = score
            best_model = kmeans

    # Calculate stability using Adjusted Rand Index
    from sklearn.metrics import adjusted_rand_score
    ari_scores = []
    for i in range(len(all_labels)):
        for j in range(i+1, len(all_labels)):
            ari_scores.append(adjusted_rand_score(all_labels[i], all_labels[j
    ]))

    stability = {
        'mean_silhouette': np.mean(all_scores),
```

```
        'std_silhouette': np.std(all_scores),
        'mean_ari': np.mean(ari_scores),
        'std_ari': np.std(ari_scores)
    }

    print(f"Stability Analysis for K={n_clusters}:")
    print(f"  Silhouette: {stability['mean_silhouette']:.3f} +/- {stability['
    std_silhouette']:.3f}")
    print(f"  Cross-run ARI: {stability['mean_ari']:.3f} +/- {stability['
    std_ari']:.3f}")

    return best_model, best_score, stability

# Apply robust clustering
optimal_k = 4  # Based on analysis above
final_model, final_score, stability_metrics = robust_kmeans(X_scaled,
    optimal_k)
print(f"\nFinal model silhouette score: {final_score:.3f}")
```

1.4 3. Cluster Visualization and Interpretation

1.4.1 PCA-Based Visualization

```
def visualize_clusters(X, labels, feature_names=None):
    """
    Create comprehensive cluster visualizations.
    """
    # PCA for 2D visualization
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

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

    # Scatter plot with clusters
    scatter = axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=labels,
                              cmap='viridis', alpha=0.7, s=50)
    axes[0].set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)
    ')
    axes[0].set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)
    ')
    axes[0].set_title('Clusters in PCA Space')
    plt.colorbar(scatter, ax=axes[0], label='Cluster')

    # Cluster size distribution
    unique, counts = np.unique(labels, return_counts=True)
    axes[1].bar(unique, counts, color=plt.cm.viridis(unique / max(unique)))
    axes[1].set_xlabel('Cluster')
    axes[1].set_ylabel('Number of Items')
    axes[1].set_title('Cluster Size Distribution')
    for i, (u, c) in enumerate(zip(unique, counts)):
        axes[1].annotate(f'{c} ({c/sum(counts)*100:.1f}%)',
                         xy=(u, c), ha='center', va='bottom')

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

    return pca
```

```
pca_model = visualize_clusters(X_scaled, final_model.labels_)
```

1.4.2 Cluster Profiling

```
def profile_clusters(df, labels, feature_cols, original_cols=None):
    """
    Create detailed cluster profiles with business interpretation.
    """
    df_clustered = df.copy()
    df_clustered['cluster'] = labels

    if original_cols is None:
        original_cols = feature_cols

    # Statistical summary per cluster
    profiles = []
    for cluster_id in sorted(df_clustered['cluster'].unique()):
        cluster_data = df_clustered[df_clustered['cluster'] == cluster_id]

        profile = {
            'cluster': cluster_id,
            'count': len(cluster_data),
            'percentage': len(cluster_data) / len(df_clustered) * 100
        }

        for col in original_cols:
            if col in cluster_data.columns:
                profile[f'{col}_mean'] = cluster_data[col].mean()
                profile[f'{col}_median'] = cluster_data[col].median()

        profiles.append(profile)

    profiles_df = pd.DataFrame(profiles)

    # Normalize for comparison (z-scores vs overall)
    normalized = df_clustered.groupby('cluster')[original_cols].mean()
    for col in original_cols:
        if col in df_clustered.columns:
            overall_mean = df_clustered[col].mean()
            overall_std = df_clustered[col].std()
            normalized[col] = (normalized[col] - overall_mean) / overall_std

    print("Cluster Sizes:")
    print(profiles_df[['cluster', 'count', 'percentage']].round(1))
    print("\nNormalized Cluster Profiles (Z-scores):")
    print(normalized.round(2))

    return profiles_df, normalized

# Apply profiling
original_features = ['total_funding', 'team_size', 'market_size_estimate',
                     'patent_count', 'years_since_founding']
profiles, normalized_profiles = profile_clusters(
    df_engineered, final_model.labels_, feature_columns, original_features
)
```

1.4.3 Business Interpretation Framework

```
def interpret_innovation_clusters(profiles_df, normalized_df, df_original):
    """
    Generate business interpretations for innovation clusters.
    """
    interpretations = {}

    for cluster_id in sorted(profiles_df['cluster'].unique()):
        cluster_profile = profiles_df[profiles_df['cluster'] == cluster_id].
    iloc[0]
        normalized_row = normalized_df.loc[cluster_id]

        # Determine characteristics based on z-scores
        characteristics = []

        if normalized_row.get('total_funding', 0) > 0.5:
            characteristics.append('High funding')
        elif normalized_row.get('total_funding', 0) < -0.5:
            characteristics.append('Low funding')

        if normalized_row.get('team_size', 0) > 0.5:
            characteristics.append('Large teams')
        elif normalized_row.get('team_size', 0) < -0.5:
            characteristics.append('Small teams')

        if normalized_row.get('market_size_estimate', 0) > 0.5:
            characteristics.append('Large market')
        elif normalized_row.get('market_size_estimate', 0) < -0.5:
            characteristics.append('Niche market')

        if normalized_row.get('years_since_founding', 0) > 0.5:
            characteristics.append('Mature')
        elif normalized_row.get('years_since_founding', 0) < -0.5:
            characteristics.append('Early stage')

        # Assign archetype based on characteristics
        char_str = ', '.join(characteristics)
        if 'High funding' in char_str and 'Large teams' in char_str:
            archetype = 'Established Innovators'
        elif 'Low funding' in char_str and 'Small teams' in char_str:
            archetype = 'Bootstrap Pioneers'
        elif 'Large market' in char_str and 'Early stage' in char_str:
            archetype = 'Market Disruptors'
        elif 'Niche market' in char_str:
            archetype = 'Niche Specialists'
        else:
            archetype = 'Balanced Growth'

        interpretations[cluster_id] = {
            'archetype': archetype,
            'characteristics': characteristics,
            'size': int(cluster_profile['count']),
            'percentage': cluster_profile['percentage']
        }

    # Print interpretations
    print("\n" + "="*60)
    print("CLUSTER INTERPRETATIONS")
    print("="*60)
    for cluster_id, info in interpretations.items():
        print(f"\nCluster {cluster_id}: {info['archetype']}")
        print(f"  Size: {info['size']} projects ({info['percentage']:.1f}%)")
        print(f"  Characteristics: {', '.join(info['characteristics'])}")
```

```
    return interpretations

interpretations = interpret_innovation_clusters(profiles, normalized_profiles,
    df_engineered)
```

1.5 4. Comparing Clustering Methods

1.5.1 DBSCAN Alternative

```
from sklearn.neighbors import NearestNeighbors

def find_optimal_dbscan(X, min_samples=5):
    """
    Find optimal eps for DBSCAN using k-distance graph.
    """
    neighbors = NearestNeighbors(n_neighbors=min_samples)
    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)')
    plt.ylabel(f'{min_samples}-NN Distance')
    plt.title('K-Distance Graph for DBSCAN eps Selection')
    plt.grid(True, alpha=0.3)
    plt.savefig('dbscan_eps_selection.pdf', bbox_inches='tight')
    plt.show()

    # Suggest eps at knee (90th percentile is often good starting point)
    suggested_eps = np.percentile(k_distances, 90)
    print(f"Suggested eps (90th percentile): {suggested_eps:.3f}")

    return k_distances, suggested_eps

k_distances, eps_suggestion = find_optimal_dbscan(X_scaled)

# Apply DBSCAN
dbscan = DBSCAN(eps=eps_suggestion, min_samples=5)
dbscan_labels = dbscan.fit_predict(X_scaled)

n_clusters_dbscan = len(set(dbscan_labels)) - (1 if -1 in dbscan_labels else
    0)
n_noise = (dbscan_labels == -1).sum()

print(f"\nDBSCAN Results:")
print(f"  Clusters found: {n_clusters_dbscan}")
print(f"  Noise points: {n_noise} ({n_noise/len(dbscan_labels)*100:.1f}%)")

if n_clusters_dbscan >= 2:
    mask = dbscan_labels != -1
    dbscan_silhouette = silhouette_score(X_scaled[mask], dbscan_labels[mask])
    print(f"  Silhouette score (excluding noise): {dbscan_silhouette:.3f}")
```

1.5.2 Method Comparison Summary

```
def compare_clustering_methods(X, methods_results):
    """
    Compare multiple clustering methods side by side.
    """
    comparison = []

    for name, labels in methods_results.items():
        mask = labels != -1 if -1 in labels else np.ones(len(labels), dtype=
    bool)
        n_clusters = len(set(labels[mask]))

        if n_clusters >= 2:
            metrics = {
                'Method': name,
                'N_Clusters': n_clusters,
                'Noise_Points': (labels == -1).sum() if -1 in labels else 0,
                'Silhouette': silhouette_score(X[mask], labels[mask]),
                'Calinski_Harabasz': calinski_harabasz_score(X[mask], labels[
    mask]),
                'Davies_Bouldin': davies_bouldin_score(X[mask], labels[mask])
            }
            comparison.append(metrics)

    return pd.DataFrame(comparison)

# Compare methods
methods_results = {
    'K-Means (K=4)': final_model.labels_,
    'DBSCAN': dbscan_labels
}

# Add hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
hier = AgglomerativeClustering(n_clusters=4, linkage='ward')
methods_results['Hierarchical (Ward)'] = hier.fit_predict(X_scaled)

comparison_df = compare_clustering_methods(X_scaled, methods_results)
print("\nMethod Comparison:")
print(comparison_df.round(3))
```

1.6 5. Production Pipeline

1.6.1 Complete Clustering Pipeline Class

```
class InnovationClusteringPipeline(BaseEstimator, TransformerMixin):
    """
    Production-ready clustering pipeline for innovation analysis.
    """

    def __init__(self, n_clusters=4, random_state=42):
        self.n_clusters = n_clusters
        self.random_state = random_state
        self.imputer = None
        self.scaler = None
        self.kmeans = None
        self.cluster_profiles = None
        self.feature_names = None
```

```
    def fit(self, X, feature_names=None):
        """Fit the complete pipeline."""
        self.feature_names = feature_names or [f'feature_{i}' for i in range(X
    .shape[1])]

        # Preprocessing
        self.imputer = SimpleImputer(strategy='median')
        X_imputed = self.imputer.fit_transform(X)

        self.scaler = StandardScaler()
        X_scaled = self.scaler.fit_transform(X_imputed)

        # Clustering
        self.kmeans = KMeans(
            n_clusters=self.n_clusters,
            init='k-means++',
            n_init=10,
            random_state=self.random_state
        )
        labels = self.kmeans.fit_predict(X_scaled)

        # Generate profiles
        self._generate_profiles(X, labels)

        return self

    def predict(self, X):
        """Predict cluster labels for new data."""
        X_imputed = self.imputer.transform(X)
        X_scaled = self.scaler.transform(X_imputed)
        return self.kmeans.predict(X_scaled)

    def fit_predict(self, X, feature_names=None):
        """Fit and return labels."""
        self.fit(X, feature_names)
        return self.kmeans.labels_

    def _generate_profiles(self, X, labels):
        """Generate cluster profiles."""
        df_temp = pd.DataFrame(X, columns=self.feature_names)
        df_temp['cluster'] = labels

        self.cluster_profiles = {}
        for cluster_id in sorted(np.unique(labels)):
            cluster_data = df_temp[df_temp['cluster'] == cluster_id]
            self.cluster_profiles[cluster_id] = {
                'size': len(cluster_data),
                'percentage': len(cluster_data) / len(df_temp) * 100,
                'means': cluster_data[self.feature_names].mean().to_dict(),
                'stds': cluster_data[self.feature_names].std().to_dict()
            }

    def get_cluster_summary(self):
        """Return cluster profiles."""
        return self.cluster_profiles

    def get_silhouette_score(self, X):
        """Calculate silhouette score for given data."""
        X_imputed = self.imputer.transform(X)
        X_scaled = self.scaler.transform(X_imputed)
        labels = self.kmeans.predict(X_scaled)
        return silhouette_score(X_scaled, labels)
```

```
# Usage example
pipeline = InnovationClusteringPipeline(n_clusters=4)
pipeline.fit(df_engineered[original_features].values, original_features)

# Predict on same data (or new data)
predictions = pipeline.predict(df_engineered[original_features].values)
print(f"Predictions: {np.unique(predictions, return_counts=True)}")

# Get summary
summary = pipeline.get_cluster_summary()
for cluster_id, profile in summary.items():
    print(f"Cluster {cluster_id}: {profile['size']} items ({profile['
    percentage']:.1f}%)")
```

1.7 Common Hyperparameters

() () () ()
* * * *
0.25025400.1591 Algorithm Parameter Typical Notes
0.3409 Range

() () () ()
* * * *
0.25020K30.1524 n_clusters Users
Means 2 20 elbow/silhouette analysis

() () () ()
* * * *
0.25020K30.1591 init Smart
Means means 'k-means++' initialization, use this

() () () ()
* * * *
0.25020K30.1591 n_init More
Means 3 0 runs = more stable

() *	() *	() *	() *
0.25	5	25	Typical 0.3409 Range
() *	() *	() *	() *
0.25	5	500	User-Means ally con-verges be-fore limit
() *	() *	() *	() *
0.25	5	20	Use SCAN lep-dependent distance graph
() *	() *	() *	() *
0.25	5	20	2 * n_dimensions Samples SCAN
() *	() *	() *	() *
0.25	5	20	Ward, er-age com-plete ar-chi-cal av-er-age pact clus-ters to-
() *	() *	() *	() *
0.25	5	20	Users er- use ar- dis-chi- tance_threshold cal
() *	() *	() *	() *
0.25	5	20	Mean dard- ter Scaler fea-tures at 0
() *	() *	() *	() *
0.25	5	20	Scale dard- to Scaler unit vari-ance

1.8 Practice Projects

1. **Customer Segmentation**: Load retail transaction data, create RFM features (Recency, Frequency, Monetary), apply K-Means, profile segments, and develop targeted marketing strategies for each segment.
2. **Innovation Portfolio Analysis**: Cluster startup or innovation projects by funding, team, and market characteristics. Create actionable archetypes and investment recommendations for each cluster.
3. **Product Categorization**: Cluster products by features, price, and customer behavior metrics. Use results to optimize product placement, bundling strategies, and inventory management.
4. **Geographic Market Segmentation**: Cluster geographic regions by demographic, economic, and behavioral variables. Identify priority markets for expansion.

1.9 Troubleshooting

[illegible]	[illegible]	Solution
[illegible] clusters	[illegible] initialization or too many clusters	Reduce K, increase n_init
[illegible] points in one cluster	[illegible] not scaled	Apply StandardScaler
[illegible] consistent results across runs	[illegible] initialization	Set random_state, increase n_init

()	()	()
*	*	*
0.3	4	2 Solution
()	()	()
*	*	*
0.3	4	2 Many silhouette scores near (<0.25)	different structure features, fewer clusters
()	()	()
*	*	*
0.3	4	2 Use SCAN only noise	k-distance graph, increase eps
()	()	()
*	*	*
0.3	4	2 Use memory error-sample with large data	MiniBatchK-Means
()	()	()
*	*	*
0.3	4	2 Many terrible not useful business meaningful	feature engineer features with domain input
()	()	()
*	*	*
0.3	4	2 Use balanced clusters terrible sizes	DBSCAN or GMM

1.10 Next Steps

- Read the advanced handout for mathematical foundations and enterprise deployment

- Apply clustering to your actual business data with domain expert validation
- Experiment with different feature combinations and transformations
- Implement monitoring for cluster drift in production systems
- Explore ensemble clustering for improved stability

Clustering transforms data into strategy. Good clusters are statistically valid, stable over time, and actionable for business decisions. Always validate with domain experts before deploying.