Week 1 Handout 2: Intermediate Clustering Implementation

Machine Learning for Smarter Innovation

1 Week 1 Handout 2: Intermediate Clustering Implementation

Target Audience: Practitioners with basic ML knowledge **Duration**: 45 minutes reading + coding **Level**: Intermediate

---

1.1 Technical Implementation Guide

This handout provides practical implementation details for clustering with Python and scikit-learn, including code examples and best practices.

1.1.1 Prerequisites

- Basic Python programming
- Familiarity with pandas and numpy
- Understanding of ML concepts
- Jupyter notebook or similar environment

1.2 Data Preparation Pipeline

1.2.1 1. Loading and Exploring Data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA

# Load data
df = pd.read_csv('innovation_dataset.csv')

# Explore structure
print(f"Shape: {df.shape}")
print(f"Columns: {list(df.columns)}")
print(f"Missing values:\n{df.isnull().sum()}")
print(f"Data types:\n{df.dtypes}")
```

1.2.2 2. Feature Engineering

```
# Create meaningful features
df['funding_per_employee'] = df['total_funding'] / df['team_size']
df['years_since_founding'] = 2025 - df['founding_year']
df['has_technical_founder'] = df['founder_background'].str.contains('Tech|
    Engineering')

# Handle categorical variables
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# For ordinal categories
le = LabelEncoder()
df['stage_encoded'] = le.fit_transform(df['development_stage'])

# For nominal categories (if few categories)
df_encoded = pd.get_dummies(df, columns=['industry_sector'], prefix='industry'
    )

# Select numerical features for clustering
feature_columns = [
    'total_funding', 'team_size', 'market_size_estimate',
    'funding_per_employee', 'years_since_founding', 'stage_encoded'
]
X = df[feature_columns]
```

1.2.3 3. Data Preprocessing

```
# Handle missing values
from sklearn.impute import SimpleImputer

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

# Scale features (CRITICAL for K-means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_imputed)

# Check scaling worked
print(f"Feature means: {X_scaled.mean(axis=0)}")
print(f"Feature stds: {X_scaled.std(axis=0)}")
```

1.3 Algorithm Implementation

1.3.1 1. Determining Optimal K

```
# Elbow method
def find_optimal_k(X, k_range=range(2, 11)):
    """Find optimal number of clusters using elbow method"""
    inertias = []
    silhouette_scores = []

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

```
        inertias.append(kmeans.inertia_)

        # Calculate silhouette score
        if k > 1:  # Silhouette requires at least 2 clusters
            sil_score = silhouette_score(X, kmeans.labels_)
            silhouette_scores.append(sil_score)
        else:
            silhouette_scores.append(0)

    return k_range, inertias, silhouette_scores

# Run analysis
k_range, inertias, sil_scores = find_optimal_k(X_scaled)

# Plot results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Elbow plot
ax1.plot(k_range, inertias, 'bo-')
ax1.set_xlabel('Number of Clusters (K)')
ax1.set_ylabel('Inertia')
ax1.set_title('Elbow Method')
ax1.grid(True)

# Silhouette plot
ax2.plot(k_range, sil_scores, 'ro-')
ax2.set_xlabel('Number of Clusters (K)')
ax2.set_ylabel('Silhouette Score')
ax2.set_title('Silhouette Analysis')
ax2.grid(True)

plt.tight_layout()
plt.show()

# Print recommendations
best_k = k_range[np.argmax(sil_scores)]
print(f"Recommended K based on silhouette score: {best_k}")
print(f"Silhouette score for K={best_k}: {max(sil_scores):.3f}")
```

1.3.2 2. Advanced K-Means Implementation

```
# Improved K-means with multiple initializations
def robust_kmeans(X, n_clusters, n_runs=10):
    """Run K-means multiple times and select best result"""
    best_kmeans = None
    best_score = -1

    for run in range(n_runs):
        kmeans = KMeans(
            n_clusters=n_clusters,
            init='k-means++',  # Smart initialization
            n_init=10,
            max_iter=300,
            random_state=run,
            algorithm='lloyd'  # Most stable algorithm
        )
        kmeans.fit(X)

        # Evaluate using silhouette score
        score = silhouette_score(X, kmeans.labels_)
```

```
        if score > best_score:
            best_score = score
            best_kmeans = kmeans

    return best_kmeans, best_score

# Apply robust clustering
optimal_k = 4  # Based on your analysis
final_kmeans, final_score = robust_kmeans(X_scaled, optimal_k)

print(f"Final silhouette score: {final_score:.3f}")
print(f"Cluster centers shape: {final_kmeans.cluster_centers_.shape}")
```

1.3.3 3. Alternative Clustering Methods

```
from sklearn.cluster import DBSCAN, AgglomerativeClustering
from sklearn.mixture import GaussianMixture

# DBSCAN for density-based clustering
def try_dbscan(X, eps_range=np.arange(0.3, 2.0, 0.1)):
    """Find optimal DBSCAN parameters"""
    best_eps = None
    best_score = -1
    best_labels = None

    for eps in eps_range:
        dbscan = DBSCAN(eps=eps, min_samples=5)
        labels = dbscan.fit_predict(X)

        # Skip if all points are noise or all in one cluster
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        if n_clusters < 2:
            continue

        score = silhouette_score(X, labels)
        if score > best_score:
            best_score = score
            best_eps = eps
            best_labels = labels

    return best_eps, best_score, best_labels

# Compare methods
methods = {
    'K-Means': final_kmeans.labels_,
    'DBSCAN': try_dbscan(X_scaled)[2] if try_dbscan(X_scaled)[2] is not None
    else None
}

for name, labels in methods.items():
    if labels is not None:
        score = silhouette_score(X_scaled, labels)
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        print(f"{name}: {n_clusters} clusters, silhouette = {score:.3f}")
```

1.4 Results Analysis and Visualization

1.4.1 1. Cluster Visualization

```
# PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Plot clusters
plt.figure(figsize=(10, 8))
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1],
                      c=final_kmeans.labels_,
                      cmap='viridis',
                      alpha=0.7)
plt.colorbar(scatter)
plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)')
plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)')
plt.title('Clusters in PCA Space')

# Add cluster centers
centers_pca = pca.transform(final_kmeans.cluster_centers_)
plt.scatter(centers_pca[:, 0], centers_pca[:, 1],
           c='red', marker='x', s=200, linewidths=3)
plt.show()
```

1.4.2 2. Cluster Profiling

```
# Add cluster labels to original dataframe
df['cluster'] = final_kmeans.labels_

# Profile each cluster
def profile_clusters(df, cluster_col='cluster', feature_cols=None):
    """Create detailed cluster profiles"""
    if feature_cols is None:
        feature_cols = df.select_dtypes(include=[np.number]).columns
        feature_cols = [col for col in feature_cols if col != cluster_col]

    profiles = []

    for cluster_id in sorted(df[cluster_col].unique()):
        cluster_data = df[df[cluster_col] == cluster_id]

        profile = {
            'cluster_id': cluster_id,
            'size': len(cluster_data),
            'percentage': len(cluster_data) / len(df) * 100
        }

        # Statistical summary for numerical features
        for col in feature_cols:
            if col in cluster_data.columns:
                profile[f'{col}_mean'] = cluster_data[col].mean()
                profile[f'{col}_median'] = cluster_data[col].median()
                profile[f'{col}_std'] = cluster_data[col].std()

        profiles.append(profile)

    return pd.DataFrame(profiles)

# Generate profiles
```

```
cluster_profiles = profile_clusters(df, feature_cols=feature_columns)
print(cluster_profiles.round(2))
```

1.4.3 3. Business Interpretation

```
# Create interpretation framework
def interpret_clusters(df, profiles):
    """Generate business interpretations of clusters"""
    interpretations = {}

    for idx, row in profiles.iterrows():
        cluster_id = int(row['cluster_id'])

        # Analyze key characteristics
        funding_level = 'High' if row['total_funding_mean'] > df['
    total_funding'].median() else 'Low'
        team_size = 'Large' if row['team_size_mean'] > df['team_size'].median
    () else 'Small'
        market_size = 'Big' if row['market_size_estimate_mean'] > df['
    market_size_estimate'].median() else 'Niche'

        # Generate description
        description = f"Cluster {cluster_id}: {funding_level} funding, {
    team_size} teams, {market_size} markets"

        # Suggest archetype name
        if funding_level == 'High' and team_size == 'Large':
            archetype = "Established Innovators"
        elif funding_level == 'Low' and team_size == 'Small':
            archetype = "Bootstrap Pioneers"
        elif market_size == 'Big':
            archetype = "Market Disruptors"
        else:
            archetype = "Niche Specialists"

        interpretations[cluster_id] = {
            'description': description,
            'archetype': archetype,
            'size': f"{row['size']} companies ({row['percentage']:.1f}%)"
        }

    return interpretations

# Apply interpretation
interpretations = interpret_clusters(df, cluster_profiles)

for cluster_id, info in interpretations.items():
    print(f"\n{info['archetype']} ({info['size']}):")
    print(f"  {info['description']}")
```

1.5 Validation and Quality Checks

1.5.1 1. Stability Testing

```
# Bootstrap validation
def bootstrap_clustering(X, n_clusters, n_iterations=50, sample_fraction=0.8):
```

```
    """Test clustering stability with bootstrap sampling"""
    stability_scores = []

    for i in range(n_iterations):
        # Bootstrap sample
        n_samples = int(len(X) * sample_fraction)
        indices = np.random.choice(len(X), n_samples, replace=True)
        X_bootstrap = X[indices]

        # Cluster
        kmeans = KMeans(n_clusters=n_clusters, random_state=i)
        labels = kmeans.fit_predict(X_bootstrap)

        # Calculate silhouette score
        if len(set(labels)) > 1:  # Ensure multiple clusters
            score = silhouette_score(X_bootstrap, labels)
            stability_scores.append(score)

    return np.array(stability_scores)

# Test stability
stability_scores = bootstrap_clustering(X_scaled, optimal_k)
print(f"Stability test results:")
print(f"  Mean silhouette: {stability_scores.mean():.3f}")
print(f"  Std silhouette: {stability_scores.std():.3f}")
print(f"  95% CI: [{np.percentile(stability_scores, 2.5):.3f}, {np.percentile(
    stability_scores, 97.5):.3f}]")

# Plot stability
plt.figure(figsize=(8, 4))
plt.hist(stability_scores, bins=15, alpha=0.7, edgecolor='black')
plt.axvline(stability_scores.mean(), color='red', linestyle='--', label=f'Mean
    : {stability_scores.mean():.3f}')
plt.xlabel('Silhouette Score')
plt.ylabel('Frequency')
plt.title('Clustering Stability Test')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

1.5.2 2. Feature Importance Analysis

```
# Analyze which features drive clustering
def feature_importance_clustering(X, labels, feature_names):
    """Calculate feature importance for clustering"""
    from sklearn.ensemble import RandomForestClassifier

    # Train classifier to predict cluster labels
    rf = RandomForestClassifier(n_estimators=100, random_state=42)
    rf.fit(X, labels)

    # Get feature importances
    importances = rf.feature_importances_

    # Create importance dataframe
    importance_df = pd.DataFrame({
        'feature': feature_names,
        'importance': importances
    }).sort_values('importance', ascending=False)

    return importance_df
```

```
# Analyze feature importance
feature_importance = feature_importance_clustering(
    X_scaled, final_kmeans.labels_, feature_columns
)

print("Feature importance for clustering:")
print(feature_importance)

# Visualize
plt.figure(figsize=(8, 6))
plt.barh(range(len(feature_importance)), feature_importance['importance'])
plt.yticks(range(len(feature_importance)), feature_importance['feature'])
plt.xlabel('Importance')
plt.title('Feature Importance for Clustering')
plt.tight_layout()
plt.show()
```

1.6 Production Implementation

1.6.1 1. Creating a Clustering Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin

class ClusteringPipeline(BaseEstimator, TransformerMixin):
    """Complete clustering pipeline for production use"""

    def __init__(self, n_clusters=4, random_state=42):
        self.n_clusters = n_clusters
        self.random_state = random_state
        self.pipeline = None
        self.cluster_profiles = None

    def fit(self, X, feature_columns=None):
        """Fit the clustering pipeline"""
        # Create pipeline
        self.pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler()),
            ('kmeans', KMeans(n_clusters=self.n_clusters,
                              random_state=self.random_state,
                              n_init=10))
        ])

        # Fit pipeline
        self.pipeline.fit(X)

        # Generate cluster profiles
        labels = self.pipeline.predict(X)
        self.cluster_profiles = self._create_profiles(X, labels,
    feature_columns)

        return self

    def predict(self, X):
        """Predict cluster labels for new data"""
        return self.pipeline.predict(X)
```

```
    def _create_profiles(self, X, labels, feature_columns):
        """Create cluster profiles"""
        if feature_columns is None:
            feature_columns = [f'feature_{i}' for i in range(X.shape[1])]

        df_temp = pd.DataFrame(X, columns=feature_columns)
        df_temp['cluster'] = labels

        profiles = {}
        for cluster_id in sorted(np.unique(labels)):
            cluster_data = df_temp[df_temp['cluster'] == cluster_id]

            profile = {
                'size': len(cluster_data),
                'percentage': len(cluster_data) / len(df_temp) * 100
            }

            for col in feature_columns:
                profile[f'{col}_mean'] = cluster_data[col].mean()
                profile[f'{col}_std'] = cluster_data[col].std()

            profiles[cluster_id] = profile

        return profiles

    def get_cluster_summary(self):
        """Get summary of all clusters"""
        return self.cluster_profiles

# Usage example
clustering_model = ClusteringPipeline(n_clusters=optimal_k)
clustering_model.fit(X, feature_columns)

# Predict new data
new_predictions = clustering_model.predict(X)
print(f"Predicted clusters: {new_predictions[:10]}")

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

1.7 Next Steps and Advanced Topics

1.7.1 Immediate Actions:

1. **Practice**: Apply this code to your own dataset
2. **Experiment**: Try different preprocessing steps
3. **Validate**: Test with domain experts
4. **Document**: Record what works for your use case

1.7.2 Week 2 Preview:

- **DBSCAN**: Density-based clustering for complex shapes
- **Gaussian Mixture Models**: Probabilistic clustering

- **Online clustering**: Real-time data streams
- **Text clustering**: NLP applications

1.7.3 Production Considerations:

- **Monitoring**: Track cluster stability over time
- **Retraining**: When to update your model
- **Scaling**: Handle larger datasets efficiently
- **Integration**: API deployment and batch processing

This handout provides a solid foundation for practical clustering implementation. Focus on understanding each step before moving to more advanced techniques.