Lecture 11

DBSCAN, BIRCH, SOM

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a powerful and flexible clustering algorithm commonly used in data mining and machine learning. Unlike K-Means, which assumes that clusters are spherical and equally sized, DBSCAN can discover clusters of arbitrary shapes and is robust to noise in the data. It works by identifying dense regions of data points separated by sparser regions. Here's how DBSCAN works:

1. Core Point: DBSCAN relies on two parameters: a distance metric (ε) and the minimum number of data points (MinPtsMinPts) required to form a dense region. A data point is considered a core point if there are at least MinPtsMinPts data points (including itself) within a distance of ε from it.

2. Border Point: A data point is considered a border point if it is within ε distance from a core point but does not meet the MinPtsMinPts requirement itself.

3. Noise Point: Data points that are neither core points nor border points are classified as noise points.

4. Cluster Formation: DBSCAN starts by selecting an arbitrary data point. If it's a core point, a new cluster is created. The algorithm expands the cluster by adding all directly reachable core points to it. The algorithm continues to expand the cluster until no more core points can be added. If a border point is reached, it is added to the cluster as well, but the expansion of the cluster stops in that direction. The algorithm then selects another unvisited data point and repeats the process until all data points are visited.

5. Handling Noise: Noise points are data points that do not belong to any cluster. They are often associated with outliers or noise in the dataset.

Key Considerations:

- The choice of ε and MinPtsMinPts is crucial. Smaller ε values result in more fine-grained clusters, while larger ε values may merge clusters. Setting MinPtsMinPts too low can lead to excessive noise, while setting it too high may result in large clusters.
- DBSCAN is capable of handling clusters of different shapes, sizes, and densities. It can identify clusters within clusters (nested clusters).
- The order in which data points are processed does not affect the final clustering result.
- The number of clusters is not predetermined; DBSCAN can discover an arbitrary number of clusters based on the data's density distribution.
- DBSCAN is less sensitive to the initial choice of parameters compared to K-Means.
- It's suitable for both numerical and categorical data, with appropriate distance metrics.
- DBSCAN is computationally efficient and can handle large datasets effectively.

DBSCAN is a versatile clustering algorithm used in various applications, including anomaly detection, image segmentation, customer segmentation, and geographic data analysis. Its ability to uncover complex cluster structures makes it particularly valuable in situations where other clustering methods may struggle.

An example of how to use the DBSCAN clustering algorithm in R. In this example, we'll generate synthetic data and apply DBSCAN to cluster the data points. For demonstration purposes, we'll use the popular dbscan package in R. To get started, make sure you have the dbscan package installed. You can install it using the following command if you haven't already:

Now, let's create a synthetic dataset and perform DBSCAN clustering:

```
# Load the dbscan library
library(dbscan)
# Generate synthetic data with two clusters and some noise
set.seed(123)
data <- rbind( matrix(rnorm(200, mean = c(2, 2), sd = 0.2), ncol = 2),
# Cluster 1
matrix(rnorm(200, mean = c(6, 6), sd = 0.2), ncol = 2), # Cluster 2
matrix(rnorm(20, mean = c(4, 4), sd = 2), ncol = 2) # Noise
)
# Perform DBSCAN clustering
dbscan_result <- dbscan(data, eps = 0.5, MinPts = 5)
# Display cluster assignments
cluster_assignments <- dbscan_result$cluster
print(cluster_assignments)
# Plot the data points colored by cluster
plot(data, col = cluster_assignments, pch = 19, main = "DBSCAN
Clustering Example", xlab = "X", ylab = "Y") legend("topright", legend
= unique(cluster_assignments), col = 1:max(cluster_assignments), pch =
19)
```

In this code, we: Load the dbscan library. Generate a synthetic dataset with two distinct clusters and some noise points. Perform DBSCAN clustering on the synthetic data using the dbscan function. We specify the distance parameter (eps) and the minimum number of points for a cluster (MinPts). Display the cluster assignments. Create a scatter plot of the data points, with each point colored according to its assigned cluster. We also add a legend for cluster labels.

When you run this code, you will see a scatter plot of the synthetic data points, each colored by its DBSCAN-assigned cluster label. The DBSCAN algorithm has effectively separated the two main clusters and identified the noise points as outliers.

You can experiment with different parameter values, such as eps and MinPts, to see how they affect the clustering results and the identification of noise points.

BIRCH, which stands for **Balanced Iterative Reducing and Clustering using Hierarchies**, is a hierarchical clustering algorithm designed for efficiently clustering large datasets with a focus on memory and speed efficiency. BIRCH is especially well-suited for datasets that do not fit entirely into memory, making it a scalable clustering technique. Here's how BIRCH works:

Basic Procedure:
Initialization: BIRCH begins by constructing a balanced tree structure called the CF (Clustering Feature) Tree. The CF Tree is used to represent the data distribution efficiently while minimizing memory usage.

Clustering Features: BIRCH relies on a set of clustering features, which are summarized information about the data points in the dataset. Each data point contributes to the clustering features, which include the count of data points, the linear sum, and the squared sum of data points within a subcluster. These features are stored in the CF Tree.

Insertion into the CF Tree: As data points are read sequentially, BIRCH inserts them into the CF Tree. The tree nodes maintain information about the current subcluster and its clustering features.

Splitting and Merging: The CF Tree is updated dynamically as data points are inserted. When a node reaches its capacity limit, it is split into two subclusters. Subclusters that are close to each other may be merged to maintain a balanced tree structure.

Global Clustering: Once all data points have been processed, BIRCH constructs global clusters by traversing the CF Tree, merging subclusters with similar clustering features and eliminating noise clusters.

Key Considerations:

- BIRCH is designed to be memory-efficient, as it stores summary information about data points in the CF Tree rather than the data points themselves.
- The number of clusters is determined based on the properties of the CF Tree and the desired clustering threshold. BIRCH can adapt to the underlying data distribution.
- BIRCH is useful for clustering large datasets where keeping the entire dataset in memory is not feasible.
- BIRCH is sensitive to the choice of parameters, such as the branching factor of the CF Tree and the clustering threshold.
- It is particularly effective in the preprocessing step for larger datasets before applying more detailed clustering algorithms, such as hierarchical or K-Means clustering.

Applications:
BIRCH is used in various applications, including text clustering, image segmentation, and data mining tasks where datasets are too large to fit in memory. It provides a memory-efficient way to perform preliminary clustering on such datasets, allowing for further analysis and exploration.

The R language does not have a built-in package for BIRCH clustering. The BIRCH algorithm is not as commonly implemented in R as some other clustering algorithms like K-Means or hierarchical clustering. However, I can provide you with a high-level description of how to use BIRCH clustering and the kind of steps you would follow if you were to implement it yourself in R.

Data Preparation: First, you would need to load and prepare your dataset. Ensure that your data is in a suitable format and that you've identified the features to cluster on.

BIRCH Parameters: Define the parameters for BIRCH, which typically include the branching factor (the maximum number of subclusters in each node) and the clustering threshold (a measure of cluster quality or similarity that determines whether two subclusters should be merged). You need to determine appropriate values for these parameters based on your data and clustering goals.

BIRCH Tree Construction: Implement the logic to construct the BIRCH tree data structure. As data points are read sequentially from your dataset, insert them into the tree, and handle node splitting and merging as necessary to maintain a balanced tree structure.

Global Clustering: Once you've inserted all data points and constructed the BIRCH tree, you would traverse the tree to perform global clustering. This involves merging subclusters with similar features and eliminating noise clusters.

Cluster Visualization and Analysis: After the clustering is complete, you can visualize and analyze the resulting clusters as needed for your specific application.

For those who want to experiment with BIRCH clustering, it's worth exploring whether other machine learning libraries or tools support BIRCH. Available R packages and their features are being updated all the time.

Self-Organizing Maps (SOM), also known as Kohonen maps, are a type of artificial neural network that is used for clustering, visualization, and dimensionality reduction of high-dimensional data. SOMs are particularly useful for identifying patterns and relationships in data. Here's how Self-Organizing Maps work:

Basic Structure: A SOM consists of a grid of nodes, typically organized as a two-dimensional grid, but other arrangements are possible. Each node in the grid is associated with a weight vector of the same dimension as the input data. These weight vectors are initialized randomly or with small random values. SOMs are unsupervised learning models, meaning they do not require labeled training data. They learn from the input data without external guidance.

Learning Process: Initialization: Start by initializing the SOM with a grid of nodes and random weight vectors.

Input Data Presentation: For each input data point in your dataset, the SOM finds the node with the closest weight vector to the input data point. This is done by computing the distance (e.g., Euclidean distance) between the input data point and the weight vectors of all nodes.

Winner Node Selection: The node with the closest weight vector is known as the "winner" or "best-matching unit" (BMU). The BMU is the node whose weight vector is most similar to the input data point.

Weight Vector Update: Update the weight vectors of the BMU and its neighboring nodes. The weight vectors of nodes near the BMU are adjusted to be more similar to the input data point. The magnitude of adjustment decreases with the distance from the BMU. This process encourages nearby nodes to adapt their weight vectors to represent similar data patterns.

Iterative Learning: Repeat steps 3 and 4 for each input data point in your dataset. This process may involve multiple iterations or epochs.

Visualization and Clustering: After training, SOMs can be used for visualization, clustering, or dimensionality reduction. Nodes with similar weight vectors are more likely to be close to each other in the SOM grid, making them candidates for forming clusters.

Key Considerations:

- SOMs are capable of revealing underlying structures and relationships in the data, making them suitable for exploratory data analysis and visualization.
- The size and topology of the SOM grid are crucial parameters that influence the quality of the clustering and visualization. Smaller grids may oversimplify the data, while larger grids can overfit.
- SOMs can be used in a wide range of applications, including image processing, data mining, and pattern recognition.
- SOMs can be sensitive to the initialization of weight vectors, which may affect the quality of the learned representation.
- SOMs can be adapted for online learning or batch learning, depending on the specific problem.

Applications: Self-Organizing Maps find applications in a variety of fields, including data visualization, image analysis, speech recognition, and feature extraction. They are particularly valuable when dealing with high-dimensional data and exploring complex patterns within it.

A simple example of creating and training a Self-Organizing Map (SOM) using the kohonen package in R. In this example, we'll use a synthetic dataset to demonstrate how a SOM can help identify clusters and visualize the data. First, make sure you have the kohonen package installed. You can install it using the following command if you haven't already:

Now, let's create a synthetic dataset, train a SOM, and visualize the results:

```
# Load the kohonen library
library(kohonen)
# Create a synthetic dataset
set.seed(123)
data <- matrix(rnorm(200, mean = c(0, 0), sd = 1), ncol = 2)
data <- rbind(data, matrix(rnorm(200, mean = c(4, 4), sd = 1), ncol =
2))
# Normalize the data
data <- scale(data)
# Define the SOM grid dimensions
grid_rows <- 10 grid_cols <- 10
# Create and initialize the SOM
som_grid <- somgrid(xdim = grid_rows, ydim = grid_cols, topo =
"rectangular")
som_model <- som(data, grid = som_grid, rlen = 100, alpha = c(0.05,
0.01))
# Plot the SOM results
plot(som_model, type = "property", property = 1, main = "SOM
Clustering Example")
plot(som_model, type = "mapping", pchs = 20, main = "SOM Clustering
Example")
```

In this R code, we: Load the kohonen library, which provides functions for working with SOMs. Create a synthetic dataset with two clusters. The dataset consists of 400 data points (200 in each cluster) with different means and standard deviations. Normalize the data to have a mean of 0 and a standard deviation of 1. Define the dimensions of the SOM grid. In this example, we use a 10x10 grid. Create and

initialize the SOM using the somgrid function to specify the grid dimensions. We then train the SOM using the som function. The rlen parameter controls the number of training iterations, and alpha controls the learning rate decay. Finally, we visualize the results of the SOM. The first plot displays cluster properties, while the second plot shows data points mapped onto the SOM grid.

When you run this code, you will see two plots. The first plot represents the SOM's cluster properties, showing how the SOM has grouped data points into clusters. The second plot displays data points as they are mapped onto the SOM grid, helping you visualize the relationships between data points.

You can experiment with different datasets, grid dimensions, and training parameters to see how the SOM adapts to different data distributions.

Resources:

1. https://www.geeksforgeeks.org/dbscan-clustering-in-r-programming/#
2. http://www.sthda.com/english/wiki/wiki.php?id_contents=7940
3. https://rpubs.com/datalowe/dbscan-simple-example
4. https://www.datanovia.com/en/lessons/dbscan-density-based-clustering-essentials/
5. https://www.kaggle.com/code/pmcgovern/dbscan-example-in-r
6. https://rdrr.io/cran/stream/man/DSC_BIRCH.html
7. https://medium.com/@noel.cs21/balanced-iterative-reducing-and-clustering-using-heirachies-birch-5680adffaa58
8. https://www.geeksforgeeks.org/ml-birch-clustering/
9. https://www.polarmicrobes.org/tutorial-self-organizing-maps-in-r/
10. https://www.r-bloggers.com/2014/02/self-organising-maps-for-customer-segmentation-using-r/
11. https://rpubs.com/AlgoritmaAcademy/som
12. https://raraasnawi.medium.com/self-organizing-map-som-with-rstudio-81b5c5713f54
13. https://en.proft.me/2016/11/29/modeling-self-organising-maps-r/