Università Di Pisa
Dipartimento Di Ingegneria Dell'Informazione

Corso Di Laurea Triennale In
Ingegneria Informatica

Federated Dbscan

Relatore:
Prof. Francesco Marcelloni

Laureando:
Gabriele Marino

Anno Accademico 2020-2021

CONTENTS

1. DBSCAN

1.1 Introduction

Clustering algorithms are used to solve identification problems in spatial databases. DBSCAN (Density Based Spatial Clustering of Applications with Noise) is a clustering algorithm which deals with good efficiency with the main problems that rise when applying clustering algorithms to large spatial databases: minimal requirements of domain knowledge to determine the input parameters and discovery of clusters with arbitrary shape.

For a gentle introduction to clustering algorithms refer to [1].
For a detailed description of DBSCAN refer to [2].

1.2 A Density Based Notion of Clusters

DBSCAN relies on a density-based notion of clusters. The algorithm is based on the key idea is that for each point of a cluster the neighborhood of a given radius has to contain at least a minimum number of points, where both the radius of the neighbourhood and the minimum number of points depend on the specific cluster.

Let D be a database of points of some n-dimensional space E, and let $dist(p, q)$ be a distance function between two points of D. The following definitions hold.

Definition 1: (Eps-neighborhood of a point) The *Eps-neighborhood* of a point p, denoted by $N_{Eps}(p)$, is defined by $N_{Eps}(p) = \{q \in D \,|\, dist(p, q) \leq Eps\}$.

Definition 2: (directly density-reachable) A point p is *directly density-reachable* from a point q wrt. Eps and $MinPts$ if:
- $p \in N_{Eps}(q)$, and
- $|N_{Eps}(q)| \geq MinPts$ (core point condition).

Definition 3: (density-reachable) A point p is *density-reachable* from a point q wrt. Eps and $MinPts$ if there is a chain of points $p_1, \ldots, p_n$, such that $p_1 = q$, $p_n = p$ and p_{i+1} is directly reachable from p_i.

Definition 4: (density-connected) A point p is *density-connected* from a point q wrt. Eps and $MinPts$ if there is a point o such that both, p and q are density reachable from o wrt. Eps and $MinPts$.

Definition 5: (cluster) Let D be a database of points. A *cluster* C wrt. Eps and $MinPts$ is a non-empty subset of D satisfying the following conditions:

- $\forall\, p, q$: if $p \in C$ and q is density-reachable fro p wrt. Eps and $MinPts$, then $q \in C$. (Maximality)
- $\forall\, p, q \in C$: p is density-connected to q wrt. Eps and $MinPts$. (Connectivity)

Definition 6: (noise) Let $C_1, \ldots, C_k$ be the clusters of the database D wrt. Eps_i, $MinPts_i$, $i = 1, \ldots, k$. Then the *noise* is the set of points in the database D not belonging to any cluster C_i.

DBSCAN algorithm is designed to discover the clusters and the noise in a spatial database according to definitions 5 and 6, given Eps and $MinPts$.

1.3 The Algorithm

DBSCAN uses global values for Eps and $MinPts$ for all clusters. Good candidates for these parameter values are those specifying the density of the "thinnest" cluster of the database, i.e. the lowest density which is not considered to be noise.

Initially, all points in the database D are marked as "**unvisited**". DBSCAN randomly selects an unvisited point p, marks it as "**visited**" and checks the core point condition. If not, p is marked as a "**noise**" point. Otherwise, a new cluster C is created for p, and all the points in its neighborhood are added to a candidate set, N. Then, the points in N that do not belong to any cluster are iteratively added to C. Furthermore, for a point p' in N that carries the label "**unvisited**", DBSCAN marks it as "**visited**" and check its core point condition. If the condition holds, all points in the Eps-neighborhood of p' are added to N. The loop continues adding points to C until N reaches emptiness. At this time, cluster C is completed. To compute the next cluster, DBSCAN randomly selects an "**unvisited**" point from the remaining ones, until all points are visited.

The following pseudocode describes DBSCAN algorithm.

```
DBSCAN

Input
D: database containing n points
Eps: the radius parameter
MinPts: the neighborhood density threshold

Output
A set of density-based clusters
```

```
Method
• Mark each point as unvisited
• do:
   • Randomly select an unvisited point p
   • Mark p as visited
   • If the Eps-neighborhood of p has at least MinPts points:
      • Create a new cluster C
      • Add p to C
      • Let N be the set of points in the Eps-neighborhood of p
      • For each point p′ in N:
         • If p′ is unvisited:
            • Mark p′ as visited
            • If the Eps-neighborhood of p′ has at least MinPts
              points: add those points to N
         • If p′ is not yet a member of any cluster: add p′ to C
      • Output C
   • Else: mark p as noise
• Until no point is unvisited
```

2. Federated Learning

2.1 Introduction

Federated learning deals with the possibility to fuse together data from different organizations. This is crucial in real-world situations, where with the exception of a few industries, most fields have only limited data or poor-quality data. Still, in many situations, it is very difficult to break the barriers between data sources. This is due to industry competition, privacy security and complicated administrative procedures. In fact, as a result of new data regulations and privacy laws, we are forbidden to collect, fuse and process data from different places. Federated learning is a possible solution for this challenge.

Federated learning is widely overviewed in [3].

2.2 A Categorization of Federated Learning

To define what federated learning is, consider N data owners $\{P_1, \ldots, P_N\}$ with their respective data $\{D_1, \ldots, D_N\}$, and let M_{SUM} be the model trained by $D = D_1 \cup \ldots \cup D_N$. A federated-learning system is a learning process in which the data owners collaboratively train a model M_{FED} without exposing their own data to others. The accuracy of M_{FED}, V_{FED}, should be as close as possible to that of M_{SUM}, V_{SUM}. Formally, let δ be a non-negative real number; if $|V_{FED} - V_{SUM}| < \delta$, the federated learning algorithm is said to have δ-accuracy loss.

Federated learning systems can be coarsely categorized based on the data partitioning scheme, i.e. how data are distributed across the various data owners. To introduce this categorization, let F_i be the feature space and I_i the sample ID space of the data D_i. Then, we can distinguish *horizontal federated learning* from *vertical federated learning* as follows.

In horizontal federated learning the dataset is said to be *sample-partitioned*, and the following relations hold:

$$F_i = F_j, \qquad I_i \neq I_j, \qquad \forall\, D_i, D_j \colon i \neq j,$$

whilst in vertical federated learning the dataset is *feature-partitioned*:

$$F_i \neq F_j, \qquad I_i = I_j, \qquad \forall\, D_i, D_j \colon i \neq j.$$

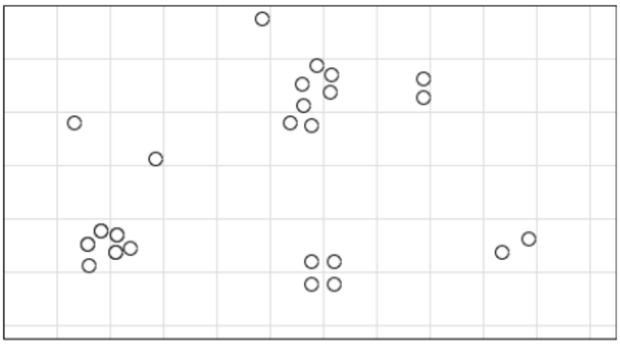
3. Federated DBSCAN

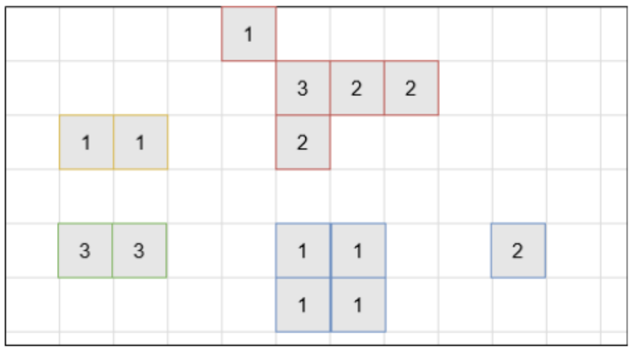
3.1 Horizontal Federated DBSCAN

In presenting the algorithm we refer to a set of clients willing to train a federated learning model interacting with a trustworthy server.

The key idea of the algorithm is the partitioning of each local feature space with a fixed granularity. The domain of definition of the features and the granularity of the grid are known and are the same for each client. This approach allows each client to share with the server only the number of points within the non-empty cells of the grid, preventing raw data sharing and thus preserving privacy.

The picture below describes the idea: the left picture represents a local dataset for a given client with the superimposed grid partitioning, supposing a two dimensional feature space, while the right one represents the information shared with the server, that is the number of points in each cell.





The following pseudocode describes horizontal federated DBSCAN.

<u>**Horizontal Federated DBSCAN**</u>

Input
L: the granularity of the cells
$MinPts$: the cell density threshold

Method
<u>*Each client m*</u>
- Compute grid for its local dataset, based on L
- Evaluate the number of points in each non-empty cell and transmits this information to the server

```
Server
• For each cell c:
   • Compute N_c, the overall number of points in the cell c,
     obtained as the sum of the contributions from all clients
   • If N_C ≥ MinPts: Mark c as a dense cell
• Evaluate clustering by expanding a cluster along adjacent dense
  cells
• Transmit to each client information about cluster membership of
  each dense cell
Each client m
• For each cell c:
   • If c is dense: assign all the points in c to the cluster the
     cell belongs to, as assigned by the server
   • Else:
      • If at least one of the cells adjacent to c is dense: assign
        each point in c to the cluster of the nearest adjacent
        dense cell
      • Else: Mark the points in the cell as outliers
```

3.2 Vertical Federated DBSCAN

As for the horizontal federated version, we still refer to a set of clients willing to train a federated learning model interacting with a trustworthy server.

The key idea of the algorithm is to locally compute a neighborhood matrix and share this matrix with the server, thus protecting raw data. The server then aggregates such local neighborhood matrices in a global neighborhood matrix, considering two points as neighbors if and only if they are neighbors for each client. Finally, the server executes an adapted version of the classical DBSCAN algorithm and share the results with the clients.

The following pseudocode describes vertical federated DBSCAN.

```
Vertical Federated DBSCAN

Input
Eps: neighborhood radius
MinPts: the neighborhood density threshold

Method (Vertical DBSCAN)
Server
• Share with client Eps parameter
Each client m
// Let N be the number of points of the database
• For i ∈ {1,…,N}:
   • For j ∈ {1,…,N}:
• If distance(x_i, x_j) < Eps: Sim_m[i,j] = Sim_m[j,i] = 1
```

- Send Sim_m matrix to server

Server

- **For** $i \in \{1,\ldots,N\}$:
  - **For** $j \in \{1,\ldots,N\}$:
    - **If** $Sim_m[i,j] = Sim_m[j,i] = 1$ for each client m: $Sim[i,j] = Sim[j,i] = 1$
- $Q =$ Server_DBSCAN(Sim, $MinPts$)
- Send the Q vector to the clients

Server_DBSCAN

Input

Sim: global neighborhood matrix
$MinPts$: the neighborhood density threshold

Method

- **For** each row i in Sim:
  - **If** i is *visited*: **continue**
  - **Else:**
    - Mark i as *visited*
    - $NumPts =$ sum(i) // Sum of cells on the i-row equal to 1
    - **If** $NumPts < MinPts$: mark i as *noise*
    - **Else:**
      - $c = newCluster$
      - $toVisit = \{j : Sim[i,j] = 1\}$
      - Expand_Cluster(Sim, i, $toVisit$, c, $MinPts$)
- Let $Q = [q_1, \ldots, q_N]$ be the vector of cluster assignment of the points in the database
- **Return** Q

Expand_Cluster

Input

Sim: global neighborhood matrix
i: current row of the matrix, related to the i-th point
c: current cluster
$toVisit$: vector of points to visit
$MinPts$: the neighborhood density threshold

Method

- $q_i = c$ // Add i to cluster c
- **For** j in $toVisit$:
  - **If** j is *unvisited:*
    - Mark j as *visited*
    - $NumPts_j =$ sum(j) // Sum of cells on the j-row equal to 1
    - **If** $NumPts_j \geq MinPts$: $toVisit = toVisit \cup \{k : Sim[j,k] = 1\}$
  - **If** j is not member of any cluster: $q_j = c$ // Add j to cluster c

4. Python Implementation

4.1 Introduction

A Python implementation of the horizontal and vertical federated DBSCAN is here proposed. Flask and Flask-RESTful frameworks and HTTP post method have been used to handle the clients-server network communication.

The following function allow server and clients to communicate with each other.

```
# File: utils.py

import requests
import concurrent.futures
from typing import Dict, List

def send_post(url: str, data: Dict):
  r = requests.post(f'http://{url}', json = data)
  return r.json(), r.status_code

def process_http_posts(clients: List, data: Dict):
  with concurrent.futures.ThreadPoolExecutor() as executor:
    futures = [executor.submit(send_post, c, data) for c in clients]
    concurrent.futures.wait(futures)

  results = []
  failures = []

  for future in futures:
    failure = future.exception()
    if failure is not None:
      failures.append(failure)
    else:
      result = future.result()
      results.append(result[0])

  return results, failures
```

4.2 Horizontal Federated DBSCAN

The file `HF_DBSCAN/fd_dbscan.py` defines the methods of the federated server and client.

```
# File: HF_DBSCAN/fd_dbscan.py

import math
import numpy as np
from typing import List, Dict
from scipy.spatial import distance
```

```python
def get_all_neighbors(cell: tuple):
  diag_coord = [(x - 1, x, x + 1) for x in cell]
  cartesian_product = [[]]
  for pool in diag_coord:
    cartesian_product = [(x + [y]) for x in cartesian_product for y in pool]

  neighbors = []
  for prod in cartesian_product:
    differential_coord = 0
    for i in range(len(prod)):
      if prod[i] != cell[i]:
        differential_coord += 1
    if differential_coord == 1:
      neighbors.append(tuple(prod))
  return neighbors

class FDBSCAN_Client():

  def initialize(self, params: Dict):
    self.__dataset = params['dataset']
    self.__L = params['L']
    self.__labels = []
    self.__true_labels = params['true_labels']
    self.__passive = True

  def get_dataset(self):
    return self.__dataset

  def get_labels(self):
    return self.__labels, self.__true_labels

  def is_passive(self):
    return self.__passive

  def __get_points(self, floor: bool = False):
    dimension = len(self.__dataset[0])
    points = []
    for row in self.__dataset:
      if floor:
        points.append(tuple(math.floor(row[i] / self.__L) for i in
range(dimension)))
      else:
        points.append(tuple(row[i] for i in range(dimension)))
    return points

  def compute_local_update(self):

    self.__passive = False

    cells = np.array(self.__get_points(floor = True))
    dimensions = len(cells[0])

    max_cell_coords = []
    min_cell_coords = []
    for i in range(dimensions):
      max_cell_coords.append(np.amax(cells[:, i]))
      min_cell_coords.append(np.amin(cells[:, i]))

    shifts = np.zeros(dimensions)
    for i in range(dimensions):
      if min_cell_coords[i] < 0:
        shifts[i] = -1 * min_cell_coords[i]
```

```python
    shifted_dimensions = ()
    for i in range(dimensions):
      shifted_dimensions += (int(max_cell_coords[i] + 1 + shifts[i]), )

    count_matrix = np.zeros(shifted_dimensions)
    for cell in cells:
      shifted_cell_coords = ()
      for i in range(dimensions):
        shifted_cell_coords += (int(cell[i] + shifts[i]),)
      count_matrix[shifted_cell_coords] += 1

    non_zero = np.where(count_matrix > 0)
    non_zero_indexes = []
    for i in range(len(non_zero)):
      for j in range(len(non_zero[i])):
        if i == 0:
          non_zero_indexes.append((int(non_zero[i][j]), ))
        else:
          non_zero_indexes[j] += (int(non_zero[i][j]), )

    dict_to_return = {}
    for index in non_zero_indexes:
      shifted_index = ()
      for i in range(len(index)):
        shifted_index += (int(index[i] - shifts[i]), )
      dict_to_return[shifted_index] = count_matrix[index]

    return dict_to_return

  def assign_points_to_cluster(self, cells: List, labels: List):

    points = self.__get_points()

    dense_cells = []
    for row in cells:
      dense_cells.append(tuple(row))

    while len(points) > 0:
      actual_point = points.pop(0)
      actual_cell = tuple(math.floor(actual_point[i] / self.__L) for i in
range(len(actual_point)))
      outlier = True
      if actual_cell in dense_cells:
        self.__labels.append(labels[dense_cells.index(actual_cell)])
      else:
        min_dist = float('inf')
        cluster_to_assign = -1
        check_list = get_all_neighbors(actual_cell)
        for check_cell in check_list:
          if check_cell in dense_cells:
            cell_mid_point = tuple(cell_coord * self.__L + self.__L/2 for
cell_coord in check_cell)
            actual_dist = distance.euclidean(actual_point, cell_mid_point)
            if actual_dist < min_dist:
              min_dist = actual_dist
              cluster_to_assign = labels[dense_cells.index(check_cell)]
            outlier = False
        self.__labels.append(cluster_to_assign)
```

```
class FDBSCAN_Server():

  def initialize(self, params: Dict):
    self.__MIN_POINTS = params['MIN_POINTS']
    self.__running = False

  def get_running(self):
    return self.__running

  def run(self, value: bool = True):
    self.__running = value

  def compute_clusters(self, contribution_map: Dict):

    key_list = list(contribution_map.keys())
    value_list = list(contribution_map.values())

    n_cells = len(key_list)
    visited = np.zeros(n_cells)
    clustered = np.zeros(n_cells)
    cells = []
    labels = []
    cluster_ID = 0

    while 0 in visited:
      curr_index = np.random.choice(np.where(np.array(visited) == 0)[0])
      curr_cell = key_list[curr_index]
      visited[curr_index] = 1

      num_points = value_list[curr_index]
      if num_points >= self.__MIN_POINTS:
        cells.append(curr_cell)
        labels.append(cluster_ID)
        clustered[curr_index] = 1

        list_of_cells_to_check = get_all_neighbors(curr_cell)
        while len(list_of_cells_to_check) > 0:
          neighbor = list_of_cells_to_check.pop(0)
          neighbor_index = key_list.index(neighbor) if neighbor in key_list
else ""
          if neighbor in key_list and visited[neighbor_index] == 0:
            visited[neighbor_index] = 1
            if value_list[neighbor_index] >= self.__MIN_POINTS:
              list_of_cells_to_check += get_all_neighbors(neighbor)
            if clustered[neighbor_index] == 0:
              cells.append(neighbor)
              labels.append(cluster_ID)
              clustered[neighbor_index] = 1
        cluster_ID += 1

    return cells, labels
```

The server and client interfaces are defined in file `HF_DBSCAN/fd_server.py` and `HF_DBSCAN/fd_client.py`.

```
# File: HF_DBSCAN/fd_server.py

import random
from flask import Flask, request
from flask_restful import Resource, Api
from typing import Dict, List
```

```
from utils import process_http_posts
from fd_dbscan import FDBSCAN_Server

def training(clients: List, server: FDBSCAN_Server, clients_selection_seed:
int, missing_clients_percentage: int):
  import random
  random.seed(clients_selection_seed)
  n_clients = len(clients)
  n_clients_to_select = int(n_clients * (100 - missing_clients_percentage) /
100)
  selected_clients_idx = random.sample(range(n_clients), n_clients_to_select)
  selected_clients = [c for cli, c in enumerate(clients) if cli in
selected_clients_idx]
  print('Training started')
  print(f'Selected clients: {selected_clients_idx}')

  data = {'action': 'compute_local_update'}
  local_updates, failures = process_http_posts(selected_clients, data)
  contribution_map = {}
  for i in range(len(local_updates)):
    local_update = local_updates[i]
    for string_key, value in local_update.items():
      tuple_key = eval(string_key)
      if tuple_key in contribution_map:
        contribution_map[tuple_key] += value
      else:
        contribution_map[tuple_key] = value

  cells, labels = server.compute_clusters(contribution_map)

  data = {'action': 'assign_points_to_cluster', 'cells': cells, 'labels':
labels}
  process_http_posts(clients, data)

def connect(json_data: Dict, clients: List, client_ids: List, running: bool):
  if (not running):
    client_id = json_data.get('client_id')
    address = json_data.get('address')
    client_ids.append(client_id)
    clients.append(address)
    code = 200
    message = {'message': f'Client {client_id} Connected'}
  else:
    code = 500
    message = {'message': 'Unable to connect: a training process is runnning'}
  return message, code

def run_server(host: str, port: int, MIN_POINTS: int, clients_selection_seed:
int, missing_clients_percentage: int):

  clients = []
  client_ids = []

  params = {
    'MIN_POINTS': MIN_POINTS
  }
  server = FDBSCAN_Server()
  server.initialize(params)
```

```
class FederatedServer(Resource):

    def get(self):
      action = request.args.get('action')
      if (action == 'start'):
        server.run();
        training(clients, server, clients_selection_seed,
missing_clients_percentage)
        server.run(False);
        return {'message': 'Training completed'}
      else:
        return {'message': 'Hello world from server', 'clients': clients}

    def post(self):
      json_data = request.get_json()
      action = json_data.get('action')
      if (action == 'connect'):
        return connect(json_data, clients, client_ids, server.get_running())
      else:
        return {'message': 'Action not Supported'}, 201

  app = Flask(__name__)
  api = Api(app)
  api.add_resource(FederatedServer, '/')
  app.run(host = host, port = port)
```

```
# File: HF_DBSCAN/fd_client.py

from typing import List
from flask import Flask, request
from flask_restful import Resource, Api

from utils import send_post
from fd_dbscan import FDBSCAN_Client

def run_client(client_id: str, server_url: str, host: str, port: int, dataset:
List, L: float, true_labels: List):

  address = host + f':{port}'
  send_post(server_url, {'action': 'connect','client_id': client_id,
'address': address})

  params = {
    'dataset': dataset,
    'L': L,
    'true_labels': true_labels
  }
  client = FDBSCAN_Client()
  client.initialize(params)

  class FederatedClient(Resource):

    def get(self):
      action = request.args.get('action')
      if (action == 'results'):
        labels, true_labels = client.get_labels();
        return {'passive': client.is_passive(), 'dataset':
client.get_dataset().tolist(), 'labels': labels, 'true_labels':
true_labels.tolist()}
      else:
        return {'message': 'Hello World from Client', 'client_id': client_id}
```

```
    def post(self):
      json_data = request.json
      action = json_data.get('action')
      if (action == 'compute_local_update'):
        result = client.compute_local_update()
        to_return = {}
        for tuple_key in result:
          string_key = ','.join([str(coord) for coord in tuple_key])
          to_return[string_key] = result[tuple_key]
        return to_return
      elif (action == 'assign_points_to_cluster'):
        cells = json_data.get('cells')
        labels = json_data.get('labels')
        client.assign_points_to_cluster(cells, labels)
      else:
        return {'message': 'Action not Supported'}, 201

  app = Flask(client_id)
  api = Api(app)
  api.add_resource(FederatedClient, '/')
  app.run(host = host, port = port)
```

Finally, `HF_DBSCAN/main_server.py` runs the server and `HF_DBSCAN/main_client.py` runs the client.

```
# File: HF_DBSCAN/main_server.py

from fd_server import run_server

host = "127.0.0.1"
port = 8080
# MIN_POINTS = 4 # banana
MIN_POINTS = 15 # s-set1

clients_selection_seed = 1
missing_client_percentage = 10

run_server(host, port, MIN_POINTS, clients_selection_seed,
missing_client_percentage)
```

```
# File: HF_DBSCAN/main_client.py

import numpy as np
import pandas as pd
import uuid
from typing import List
from threading import Thread
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import MinMaxScaler
from scipy.io import arff

from fd_client import run_client
```

```python
def generate_dataset_chunks(X: np.array, Y: List, n_splits: int):
  if (n_splits == 1):
    return [X]
  skf = StratifiedKFold(n_splits = n_splits)
  dataset_chunks = []
  true_labels = []
  for train_index, test_index in skf.split(X, Y):
    dataset_chunks.append(X[test_index])
    true_labels.append(Y[test_index])
  return dataset_chunks, true_labels

def prepare_dataset(num_clients: int):
  dataset_dir = '../datasets'
  # dataset_file = 'banana.arff'
  dataset_file  = 's-set1.arff'
  dataset_path = f'{dataset_dir}/{dataset_file}'
  dataset = arff.loadarff(dataset_path)
  df = pd.DataFrame(dataset[0])

  Y = df['class'].tolist()
  Y = np.array([-1 if y == b'noise' else int(y) for y in Y])
  del df['class']
  X_original = np.array(df.values)
  min_max_scaler = MinMaxScaler()
  X = min_max_scaler.fit_transform(X_original)
  dataset_chunks, true_labels = generate_dataset_chunks(X, Y, num_clients)
  return dataset_chunks, true_labels

server_url = "127.0.0.1:8080"
host = "127.0.0.1"
start_port = 5000
N_clients = 10
dataset_chunks, true_labels = prepare_dataset(N_clients)
# L = 0.03 # banana
L = 0.03 # s-set1

threads = []
for cli in range(N_clients):
  client_id = uuid.uuid4().hex
  port = start_port + cli
  thread_obj = Thread(target = run_client, args = (client_id, server_url,
host, port, dataset_chunks[cli], L, true_labels[cli]))
  threads.append(thread_obj)
  thread_obj.start()

for t in threads:
  t.join()
```

4.3 Vertical Federated DBSCAN

As for the horizontal federated DBSCAN implementation, the file `VF_DBSCAN/fd_dbscan.py` defines the methods of the federated server and client, their interfaces are defined in the files `VF_DBSCAN/fd_server.py` and `VF_DBSCAN/fd_client.py`, and the `VF_DBSCAN/main_server.py` and `VF_DBSCAN/main_client.py` files run them.

```python
# File: VF_DBSCAN/fd_dbscan.py

import numpy as np
from typing import List, Dict
from scipy.spatial import distance
from numba import jit

@jit(nopython=True)
def numba_euclidean_distance(u:np.ndarray, v:np.ndarray):
  return np.linalg.norm(u - v)

class FDBSCAN_Client():

  def initialize(self, params: Dict):
    self.__dataset = params['dataset']
    self.__labels = []

  def get_results(self):
    return self.__labels

  def __get_points(self):
    dimension = len(self.__dataset[0])
    points = []
    for row in self.__dataset:
      points.append(tuple(row[i] for i in range(dimension)))
    return points

  def compute_neighborhood_matrix(self, epsilon: float):
    points = self.__get_points()
    n_points = len(points)
    matrix = [[0] * n_points for i in range(n_points)]
    for i in range(n_points):
      for j in range(n_points):
        if (numba_euclidean_distance(points[i], points[j]) < epsilon):
          matrix[i][j] = 1
    return matrix

  def update_labels(self, labels: List):
    self.__labels = labels

class FDBSCAN_Server():

  def initialize(self, params: Dict):
    self.__MIN_POINTS = params['MIN_POINTS']
    self.__EPSILON = params['EPSILON']
    self.__running = False

  def get_running(self):
    return self.__running

  def run(self, value: bool = True):
    self.__running = value

  def get_epsilon(self):
    return self.__EPSILON
```

```
  def DBSCAN(self, global_matrix: np.ndarray):
    N = len(global_matrix)
    visited = np.zeros(N)
    labels = np.zeros(N)
    labels -= 1
    cluster_ID = 0
    for i in range(N):
      if visited[i]:
        continue
      else:
        visited[i] = 1
        num_points = np.sum(global_matrix[i])
        if num_points >= self.__MIN_POINTS:
          labels[i] = cluster_ID
          to_visit = np.where(global_matrix[i] == 1)[0].tolist()
          cluster_ID = self.__expand_cluster(to_visit, visited, global_matrix,
labels, cluster_ID)
    return labels

  def __expand_cluster(self, to_visit: List, visited: np.array,
global_matrix:np.ndarray, labels: np.array, cluster_ID: int):
    for j in to_visit:
      if visited[j] == 0:
        visited[j] = 1
        num_neighbors = np.sum(global_matrix[j])
        if num_neighbors >= self.__MIN_POINTS:
          to_visit += np.where(global_matrix[j] == 1)[0].tolist()
      if labels[j] == -1:
        labels[j] = cluster_ID
    cluster_ID += 1
    return cluster_ID
```

```
# File: VF_DBSCAN/fd_server.py

import numpy as np
from flask import Flask, request
from flask_restful import Resource, Api
from typing import Dict, List

from utils import process_http_posts
from fd_dbscan import FDBSCAN_Server

def training(clients: List, server: FDBSCAN_Server):
  data = {'action': 'compute_neighborhood_matrix', 'epsilon':
server.get_epsilon()}
  results, failures = process_http_posts(clients, data)

  N = len(results[0]['matrix'])
  global_matrix = np.zeros((N, N))
  for i in range(len(results)):
    matrix = np.array(results[i]['matrix'])
    global_matrix += matrix
  global_matrix = np.where(global_matrix < len(results), 0, 1)

  Q = server.DBSCAN(global_matrix)

  data = {'action': 'update_labels', 'labels': Q.tolist()}
  process_http_posts(clients, data)
```

```python
def connect(json_data: Dict, clients: List, client_ids: List, running: bool):
  if (not running):
    client_id = json_data.get('client_id')
    address = json_data.get('address')
    client_ids.append(client_id)
    clients.append(address)
    code = 200
    message = {'message': f'Client {client_id} Connected'}
  else:
    code = 500
    message = {'message': 'Unable to connect: a training process is runnning'}
  return message, code

def run_server(host: str, port: int, MIN_POINTS: int, EPSILON: float):

  clients = []
  client_ids = []

  params = {
    'MIN_POINTS': MIN_POINTS,
    'EPSILON': EPSILON
  }
  server = FDBSCAN_Server()
  server.initialize(params)

  class FederatedServer(Resource):

    def get(self):
      action = request.args.get('action')
      if (action == 'start'):
        server.run();
        training(clients, server)
        server.run(False);
        return {'message': 'Training completed'}
      else:
        return {'message': 'Hello world from server', 'clients': clients}

    def post(self):
      json_data = request.get_json()
      action = json_data.get('action')
      if (action == 'connect'):
        return connect(json_data, clients, client_ids, server.get_running())
      else:
        return {'message': 'Action not Supported'}, 201

  app = Flask(__name__)
  api = Api(app)
  api.add_resource(FederatedServer, '/')
  app.run(host = host, port = port)
```

```python
# File: VF_DBSCAN/fd_client.py

from typing import List
from flask import Flask, request
from flask_restful import Resource, Api

from utils import send_post
from fd_dbscan import FDBSCAN_Client
```

```python
def run_client(client_id: str, server_url: str, host: str, port: int, dataset: List):

  address = host + f':{port}'
  send_post(server_url, {'action': 'connect','client_id': client_id, 'address': address})

  params = {
    'dataset': dataset,
  }
  client = FDBSCAN_Client()
  client.initialize(params)

  class FederatedClient(Resource):

    def get(self):
      action = request.args.get('action')
      if (action == 'results'):
        labels = client.get_results();
        return {'labels': labels}
      else:
        return {'message': 'Hello World from Client', 'client_id': client_id}

    def post(self):
      json_data = request.json
      action = json_data.get('action')
      if (action == 'compute_neighborhood_matrix'):
        epsilon = json_data.get('epsilon')
        neighborhood_matrix = client.compute_neighborhood_matrix(epsilon)
        return {'matrix': neighborhood_matrix}
      elif (action == 'update_labels'):
        labels = json_data.get('labels')
        client.update_labels(labels)
      else:
        return {'message': 'Action not Supported'}, 201

  app = Flask(client_id)
  api = Api(app)
  api.add_resource(FederatedClient, '/')
  app.run(host = host, port = port)
```

```python
# File: VF_DBSCAN/main_server.py

from fd_server import run_server

host = "127.0.0.1"
port = 8080

# MIN_POINTS = 4 # 3MC
MIN_POINTS = 6 # aggregation
# EPSILON = 0.1 # 3MC
EPSILON = 0.04 # aggregation

run_server(host, port, MIN_POINTS, EPSILON)
```

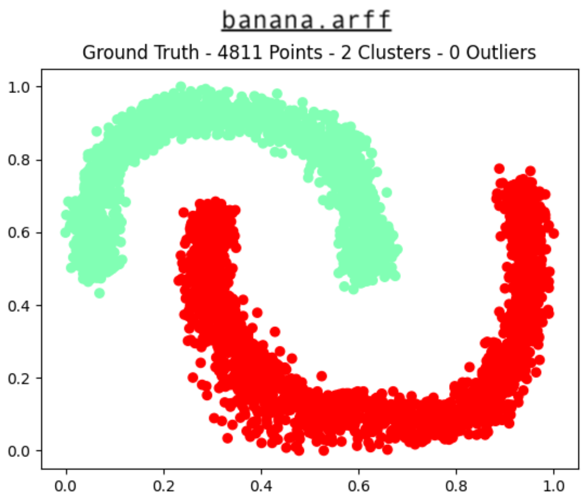
```python
# File: VF_DBSCAN/main_client.py

import numpy as np
import pandas as pd
import uuid
from threading import Thread
from sklearn.preprocessing import MinMaxScaler
from scipy.io import arff

from fd_client import run_client

def generate_dataset_chunks(X: np.array, num_clients: int):
  dataset_chunks = []
  features_list = [i for i in range(len(X[0]))]
  for i in range(num_clients):
    dataset_chunks.append(X[:, features_list[i::num_clients]])
  return dataset_chunks

def prepare_dataset(num_clients: int):
  dataset_dir = '../datasets'
  # dataset_file = '3MC.arff'
  dataset_file = 'aggregation.arff'
  dataset_path = f'{dataset_dir}/{dataset_file}'
  dataset = arff.loadarff(dataset_path)
  df = pd.DataFrame(dataset[0])

  Y = df['class'].tolist()
  Y = np.array([-1 if y == b'noise' else int(y) for y in Y])
  del df['class']
  X_original = np.array(df.values)
  min_max_scaler = MinMaxScaler()
  X = min_max_scaler.fit_transform(X_original)
  dataset_chunks = generate_dataset_chunks(X, num_clients)
  return dataset_chunks

server_url = "127.0.0.1:8080"
host = "127.0.0.1"
start_port = 5000
N_clients = 2
dataset_chunks = prepare_dataset(N_clients)

threads = []
for cli in range(N_clients):
  client_id = uuid.uuid4().hex
  port = start_port + cli
  thread_obj = Thread(target = run_client, args = (client_id, server_url,
host, port, dataset_chunks[cli]))
  threads.append(thread_obj)
  thread_obj.start()

for t in threads:
  t.join()
```

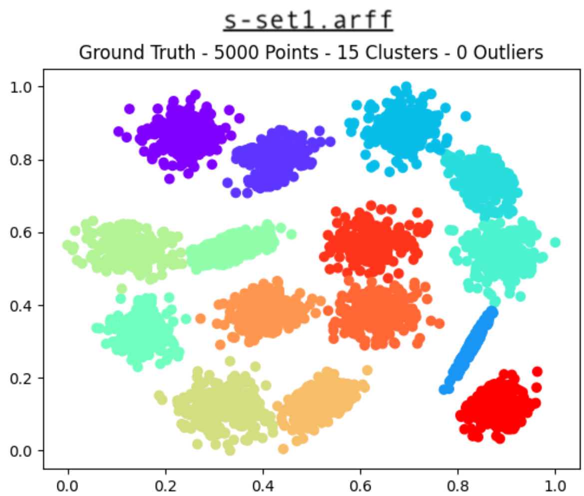
5. Experimental Analysis

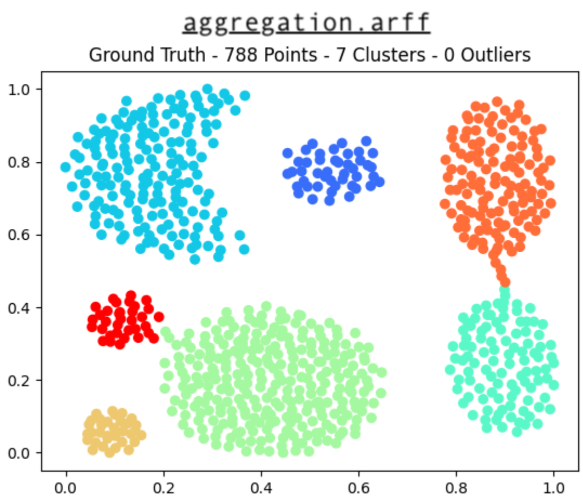
5.1 Experimental Setup

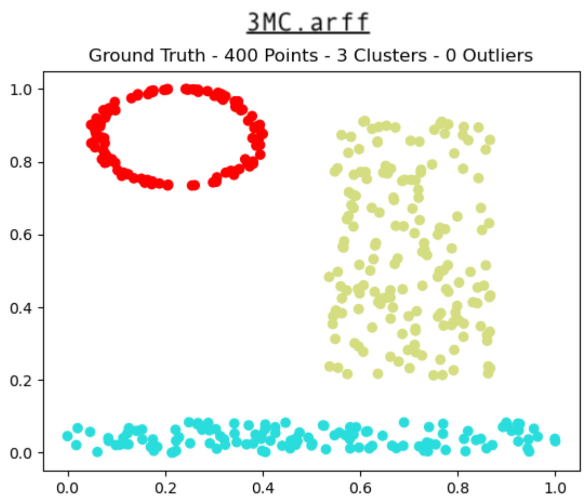
The horizontal and vertical federated versions of DBSCAN presented have been tested using the datasets described in the following table and represented below.

	HORIZONTAL FEDERATED DBSCAN		VERTICAL FEDERATED DBSCAN	
Dataset	**banana.arff**	**s-set1.arff**	**aggregation.arff**	**3MC.arff**
Attributes	2	2	2	2
Points	4811	5000	788	400
Clusters	2	15	7	3
Outliers	0	0	0	0









Five different metrics have been used to evaluate the efficiency of the algorithms: ARI and AMI scores, purity score, BCubed precision score and BCubed recall score, defined in the script `results/metrics.py`.

```
# File: results/metrics.py

import sklearn.metrics as mtr
import numpy as np
import bcubed

def ARI_score(true_labels, predicted_labels):
  return mtr.adjusted_rand_score(true_labels, predicted_labels)

def AMI_score(true_labels, predicted_labels):
  return mtr.adjusted_mutual_info_score(true_labels, predicted_labels)

def PURITY_score(true_labels, predicted_labels):
  contingency_matrix = mtr.cluster.contingency_matrix(true_labels,
predicted_labels)
  return np.sum(np.amax(contingency_matrix, axis=0)) /
np.sum(contingency_matrix)

def BCubed_Precision_score(true_labels, predicted_labels):
  ldict = {}
  cdict = {}
  for i in range(len(true_labels)):
    ldict[i] = set([true_labels[i]])
    cdict[i] = set([predicted_labels[i]])
  return bcubed.precision(cdict, ldict)

def BCubed_Recall_score(true_labels, predicted_labels):
  ldict = {}
  cdict = {}
  for i in range(len(true_labels)):
    ldict[i] = set([true_labels[i]])
    cdict[i] = set([predicted_labels[i]])
  return bcubed.recall(cdict, ldict)

def print_metrics(true_labels, elaborated_labels, message):
  print(f'{message}:')
  print(f'Purity: {PURITY_score(true_labels, elaborated_labels):.4f}')
  print(f'ARI: {ARI_score(true_labels, elaborated_labels):.4f}')
  print(f'AMI: {AMI_score(true_labels, elaborated_labels):.4f}')
  print(f'BCubed Precision: {BCubed_Precision_score(true_labels,
elaborated_labels):.4f}')
  print(f'BCubed Recall: {BCubed_Recall_score(true_labels,
elaborated_labels):.4f}\n')
```

The following scripts have been used to retrieve the results.

```
# File: results/HF_results.py

from scipy.io import arff
from sklearn.preprocessing import MinMaxScaler
from sklearn.cluster import DBSCAN
from urllib.request import urlopen
import numpy as np
import pandas as pd
import json
```

```python
from plot import plot2D
import metrics as mtr

# results_folder = 'banana'
results_folder = 's-set1'

start_port = 5000
N_clients = 10
url = 'http://localhost:8080/?action=start'
urlopen(url)

dataset = []
active_dataset = []
passive_dataset = []
labels = []
active_labels = []
passive_labels = []
true_labels = []
active_true_labels = []
passive_true_labels = []
passive_clients = False
for port in range(start_port, start_port + N_clients):
  url = f'http://localhost:{port}/?action=results'
  response = urlopen(url)
  data_json = json.loads(response.read())
  res_dataset = data_json['dataset']
  res_labels = data_json['labels']
  res_true_labels = data_json['true_labels']
  passive = data_json['passive']
  dataset += res_dataset
  labels += res_labels
  true_labels += res_true_labels
  if passive:
    passive_clients = True
    passive_dataset += res_dataset
    passive_labels += res_labels
    passive_true_labels += res_true_labels
  else:
    active_dataset += res_dataset
    active_labels += res_labels
    active_true_labels += res_true_labels

# MIN_POINTS = 4 # banana
MIN_POINTS = 15 # s-set1
# EPSILON = 0.03 # banana
EPSILON = 0.03 # s-set1
clustering = DBSCAN(eps = EPSILON, min_samples = MIN_POINTS)
dbscan_labels = clustering.fit_predict(np.array(dataset))

plot2D(points = np.array(dataset), labels = np.array(true_labels), folder =
results_folder, message = 'Ground Truth')
plot2D(points = np.array(dataset), labels = dbscan_labels, folder =
results_folder, message = 'DBSCAN')
plot2D(points = np.array(dataset), labels = np.array(labels), folder =
results_folder, message = 'Federated DBSCAN')
plot2D(points = np.array(active_dataset), labels = np.array(active_labels),
folder = results_folder, message = 'Federated DBSCAN - Active')
if passive_clients:
  plot2D(points = np.array(passive_dataset), labels =
np.array(passive_labels), folder = results_folder, message = 'Federated DBSCAN
- Passive')
```

```
mtr.print_metrics(true_labels, dbscan_labels, 'DBSCAN')
mtr.print_metrics(true_labels, labels, 'Federated DBSCAN')
mtr.print_metrics(active_true_labels, active_labels, 'Federated DBSCAN –
Active')
if passive_clients:
  mtr.print_metrics(passive_true_labels, passive_labels, 'Federated DBSCAN –
Passive')
```

```
# File: results/VF_results.py

from scipy.io import arff
from sklearn.preprocessing import MinMaxScaler
from sklearn.cluster import DBSCAN
from urllib.request import urlopen
import numpy as np
import pandas as pd
import json

from plot import plot2D
import metrics as mtr

# results_folder = '3MC'
results_folder = 'aggregation'

dataset_dir = '../datasets'
# dataset_file = '3MC.arff'
dataset_file = 'aggregation.arff'
dataset_path = f'{dataset_dir}/{dataset_file}'
dataset = arff.loadarff(dataset_path)
df = pd.DataFrame(dataset[0])

true_labels = df['class'].tolist()
true_labels = np.array([-1 if label == b'noise' else int(label) for label in
true_labels])
del df['class']
X_original = np.array(df.values)
min_max_scaler = MinMaxScaler()
X = min_max_scaler.fit_transform(X_original)

# MIN_POINTS = 4 # 3MC
MIN_POINTS = 6 # aggregation
# EPSILON = 0.1 # 3MC
EPSILON = 0.04 # aggregation
clustering = DBSCAN(eps = EPSILON, min_samples = MIN_POINTS)
dbscan_labels = clustering.fit_predict(X)

url = 'http://localhost:8080/?action=start'
urlopen(url)
url = 'http://localhost:5000/?action=results'
response = urlopen(url)
data_json = json.loads(response.read())
labels = np.array(data_json['labels'])

plot2D(points = X, labels = true_labels, folder = results_folder, message =
"Ground Truth")
plot2D(points = X, labels = dbscan_labels, folder = results_folder, message =
"DBSCAN")
plot2D(points = X, labels = labels, folder = results_folder, message =
f'Federated DBSCAN')

mtr.print_metrics(true_labels, dbscan_labels, 'DBSCAN')
mtr.print_metrics(true_labels, labels, 'Federated DBSCAN')
```

Finally, the script `results/plot.py` was used to plot the results.

```
# File: results/plot.py

from matplotlib import pyplot as plt
import matplotlib.cm as cm
import numpy as np

def plot2D(points: np.ndarray, labels: np.array, folder: str, message: str):
  int_labels = [int(label) for label in labels]
  color_range = cm.rainbow(np.linspace(0, 1, np.max(np.unique(int_labels))+1))
  colors = []
  count_outliers = 0

  for label in int_labels:
    if label == -1:
      count_outliers += 1
      colors.append([0, 0, 0, 1])
    else:
      colors.append(color_range[label])

  plt.scatter(points[:, 0], points[:, 1], color = colors, marker = "o")
  plt.title(f'{message} - '
        f'{len(int_labels)} Points - '
        f'{len(np.unique(int_labels)) - (1 if count_outliers > 0 else 0)}
Clusters - '
        f'{count_outliers} Outliers')
  plt.savefig(f'{folder}/{message}.png')
  plt.clf()
```

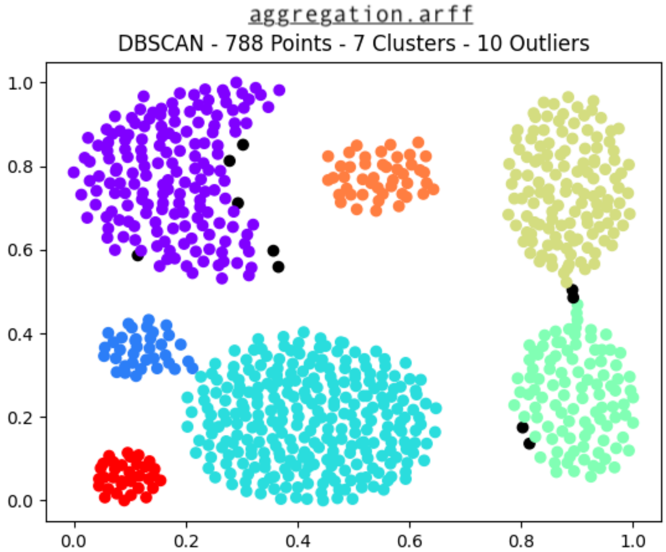
In a further experiment some of the clients were excluded from the learning process when testing horizontal federated DBSCAN over `banana.arff` dataset. Thus, active clients, whose datasets were used in training the model, were distinguished by passive clients. Passive clients, who did not share any knowledge with the server, were notified by the server at the conclusion of the learning process with the resulting clustering. Statistics have been tracked both for the active and the passive components of the learning system. The algorithm was tested with increasing passive clients percentages (10%, 20%, 30%). For each percentage, five tests have been run, each one with a different random client selection, such that it was possible to compute the mean value (μ) and the standard deviation (σ) of the metrics specified above in each of the cases.

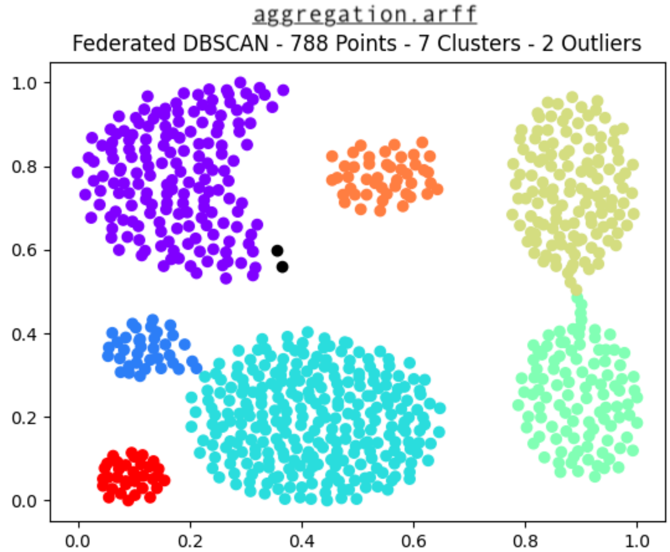
5.2 Results

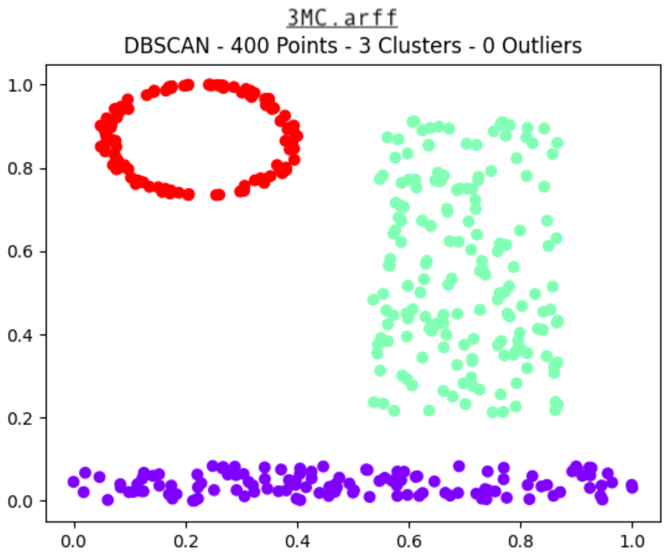
The following table shows the results obtained when testing vertical federated DBSCAN, compared to those obtained by the application of the standard version of the algorithm. The plots graphically represents the resulting clustering.

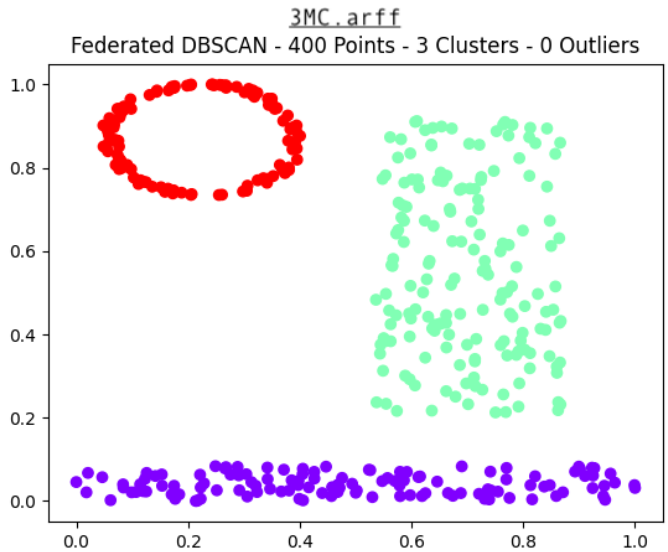
When training the model for `aggregation.arff` dataset, $MinPts$ was set to 6, and Eps to 0.04; when training the model for `3MC.arff` dataset, $MinPts$ was set to 4, and Eps to 0.1.

	VERTICAL FEDERATED DBSCAN		STANDARD DBSCAN	
Dataset	**aggregation.arff**	**3MC.arff**	**aggregation.arff**	**3MC.arff**
AMI	0,9808	1,0000	0,9675	1,0000
ARI	0,9866	1,0000	0,9779	1,0000
Purity	0,9949	1,0000	0,9911	1,0000
BCubed Precision	0,9902	1,0000	0,9856	1,0000
BCubed Recall	0,9849	1,0000	0,9678	1,0000





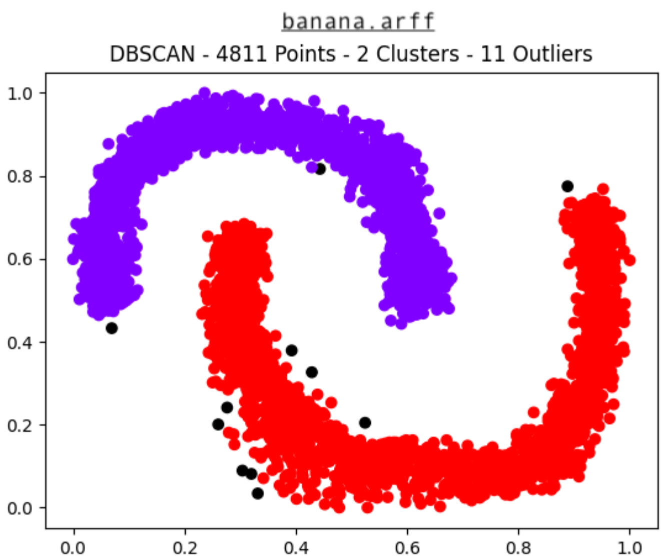


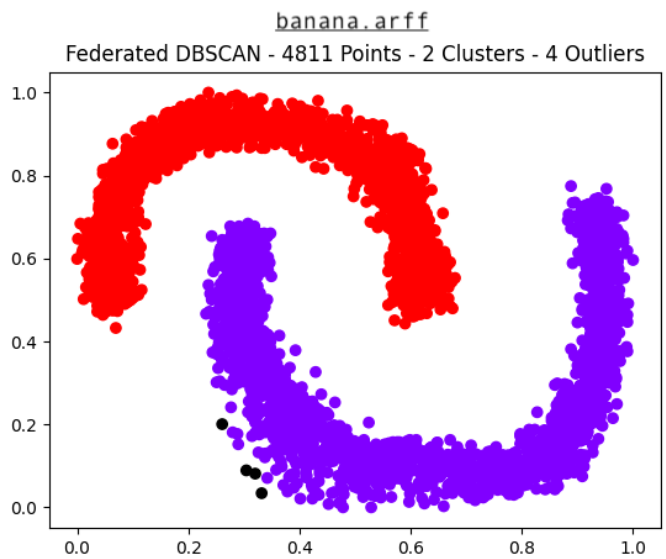


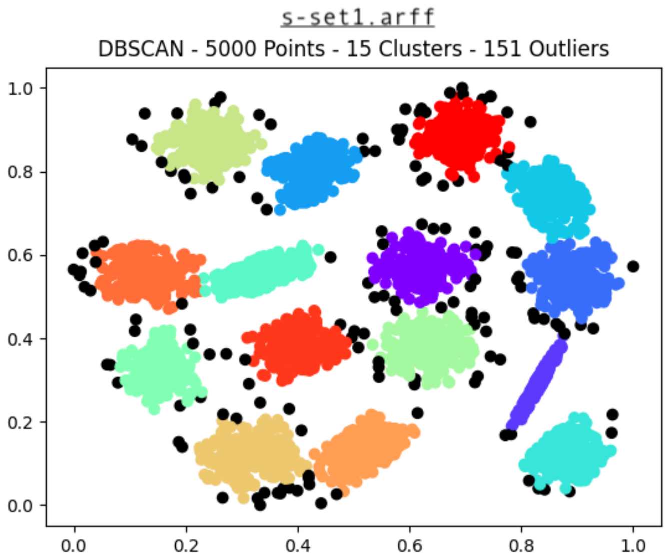
The following table shows the results obtained when testing horizontal federated DBSCAN, compared to those obtained by the application of the standard version of the algorithm. The plots graphically represents the resulting clustering. No passive clients are introduced.

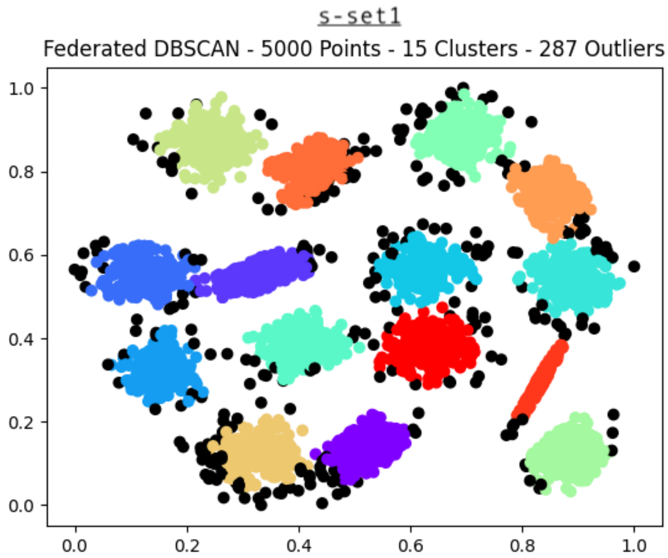
When training the model for `banana.arff` dataset, $MinPts$ was set to 4, and Eps to 0.03; when training the model for `s-set1.arff` dataset, $MinPts$ was set to 15, and Eps to 0.03.

	HORIZONTAL FEDERATED DBSCAN		STANDARD DBSCAN	
Dataset	**banana.arff**	**s-set1.arff**	**banana.arff**	**s-set1.arff**
AMI	0,9956	0,9316	0,9881	0,9615
ARI	0,9984	0,9136	0,9956	0,9600
Purity	1,0000	0,9522	0,9996	0,9740
BCubed Precision	1,0000	0,9451	0,9993	0,9716
BCubed Recall	0,9983	0,8916	0,9954	0,9411







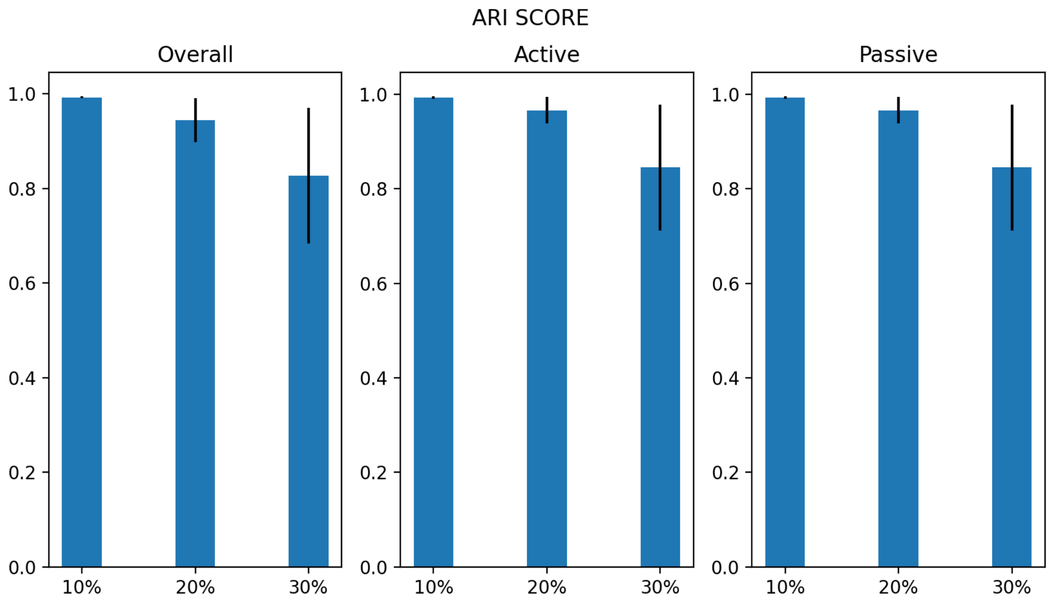


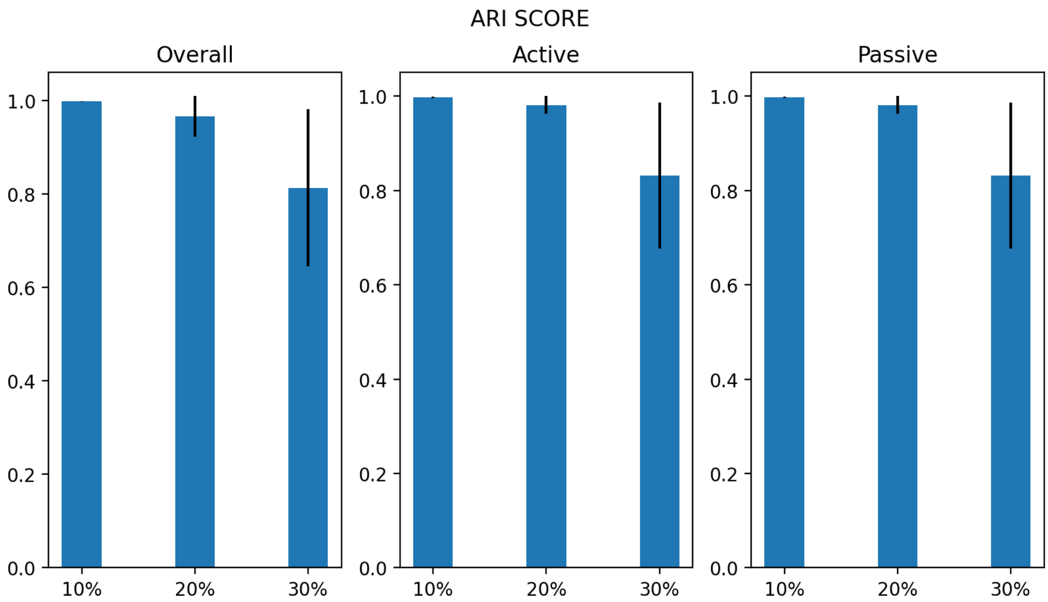
Finally, the following tables and charts show the results obtained when testing horizontal federated DBSCAN over `banana.arff` dataset with an increasing percentage of passive clients.

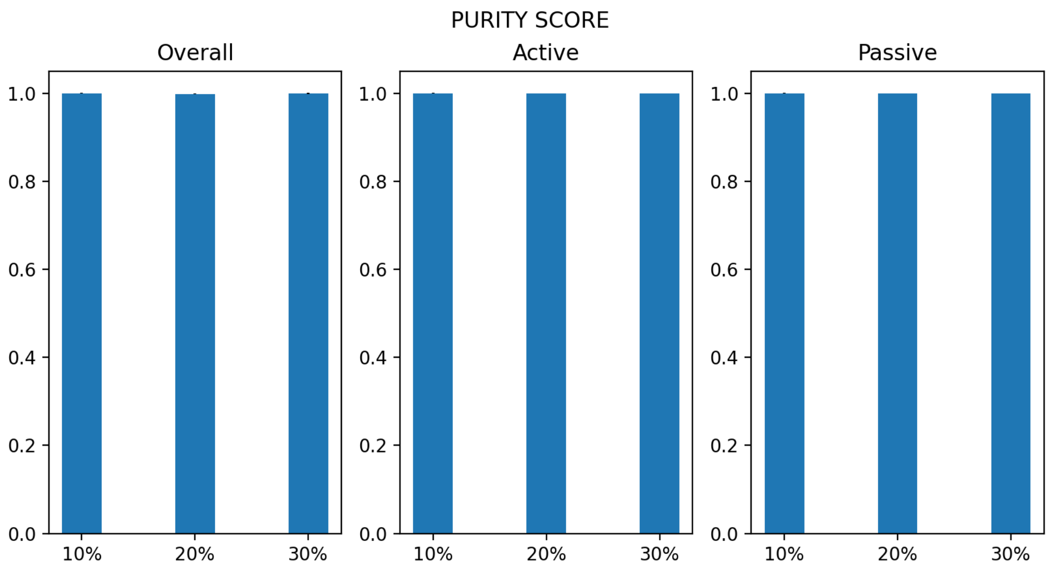
			I	II	III	IV	V	μ	σ
10%	**AMI**	**Overall**	0,9947	0,9947	0,9894	0,9912	0,9947	0,9929	0,0025
		Active	0,9952	0,9952	0,9884	0,9923	0,9952	0,9933	0,0030
		Passive	0,9904	0,9904	1,0000	0,9830	0,9904	0,9908	0,0060
	ARI	**Overall**	0,9980	0,9980	0,9961	0,9968	0,9980	0,9974	0,0009
		Active	0,9982	0,9982	0,9956	0,9973	0,9982	0,9975	0,0011
		Passive	0,9960	0,9960	1,0000	0,9921	0,9960	0,9960	0,0028
	Purity	**Overall**	1,0000	1,0000	1,0000	0,9998	1,0000	1,0000	0,0001
		Active	1,0000	1,0000	1,0000	0,9998	1,0000	1,0000	0,0001
		Passive	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	0,0000
	BCubed Precision	**Overall**	1,000	1,000	1,0000	0,9996	1,000	0,9999	0,0002
		Active	1,0000	1,0000	1,0000	0,9996	1,0000	0,9999	0,0002
		Passive	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	0,0000
	BCubed Recall	**Overall**	0,9979	0,9979	0,9959	0,9967	0,9979	0,9973	0,0009
		Active	0,9982	0,9982	0,9954	0,9972	0,9982	0,9974	0,0012
		Passive	0,9959	0,9959	1,0000	0,9918	0,9959	0,9959	0,0029

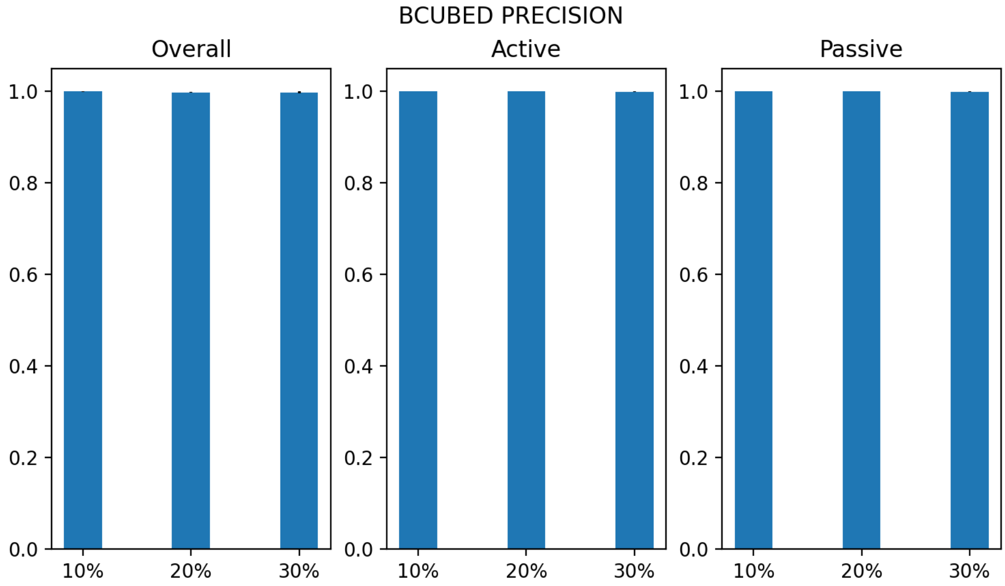
			I	II	III	IV	V	μ	σ
20%	AMI	Overall	0,8665	0,9473	0,9508	0,9779	0,9826	0,9450	0,0466
		Active	0,9289	0,9651	0,9492	0,9936	0,9936	0,9661	0,0282
		Passive	0,7721	0,8984	0,9683	0,9345	0,9569	0,9060	0,0795
	ARI	Overall	0,8886	0,9769	0,9765	0,9913	0,9933	0,9653	0,0436
		Active	0,9527	0,9844	0,9745	0,9975	0,9975	0,9813	0,0187
		Passive	0,7504	0,9473	0,9851	0,9666	0,9766	0,9252	0,0987
	Purity	Overall	0,9985	0,9979	0,9996	0,9979	0,9988	0,9985	0,0007
		Active	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	0,0000
		Passive	0,9969	0,9917	1,0000	0,9938	0,9990	0,9963	0,0035
	BCubed Precision	Overall	0,9983	0,9970	0,9994	0,9978	0,9984	0,9982	0,0009
		Active	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	0,0000
		Passive	0,9952	0,9908	1,0000	0,9922	0,9981	0,9953	0,0039
	BCubed Recall	Overall	0,8834	0,9762	0,9758	0,9913	0,9934	0,9640	0,0458
		Active	0,9503	0,9836	0,9735	0,9974	0,9974	0,9804	0,0196
		Passive	0,7402	0,9472	0,9856	0,9673	0,9775	0,9236	0,1035

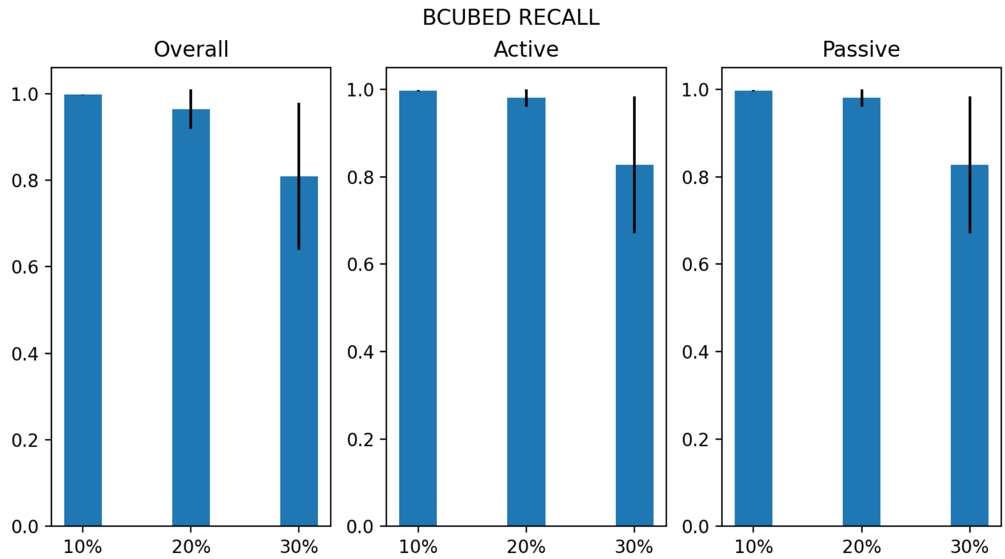
			I	II	III	IV	V	μ	σ
30%	AMI	Overall	0,9797	0,7093	0,9783	0,7830	0,6854	0,8271	0,1432
		Active	0,9886	0,7310	0,9899	0,7764	0,7390	0,8450	0,1328
		Passive	0,9651	0,6880	0,9599	0,8049	0,6254	0,8087	0,1545
	ARI	Overall	0,9913	0,6435	0,9905	0,7643	0,6747	0,8129	0,1685
		Active	0,9959	0,6673	0,9965	0,7618	0,7341	0,8311	0,1546
		Passive	0,9806	0,6171	0,9768	0,7723	0,5868	0,7867	0,1889
	Purity	Overall	0,9996	0,9985	0,9996	0,9998	0,9967	0,9988	0,0013
		Active	0,9994	0,9997	0,9994	0,9997	1,0000	0,9996	0,0003
		Passive	1,0000	0,9986	1,0000	1,0000	0,9896	0,9976	0,0045
	BCubed Precision	Overall	0,9992	0,9977	0,9992	0,9996	0,9942	0,9980	0,0022
		Active	0,9992	0,9995	0,9992	0,9995	1,0000	0,9995	0,0003
		Passive	1,0000	0,9974	1,0000	1,0000	0,9817	0,9958	0,0080
	BCubed Recall	Overall	0,9909	0,6492	0,9901	0,7525	0,6630	0,8091	0,1702
		Active	0,9959	0,6749	0,9964	0,7499	0,7207	0,8276	0,1562
		Passive	0,9797	0,6185	0,9757	0,7612	0,5827	0,7836	0,1894


ARI SCORE
Overall
Active
Passive
0.0
0.2
0.4
0.6
0.8
1.0
10%
20%
30%


ARI SCORE
Overall
Active
Passive
0.0
0.2
0.4
0.6
0.8
1.0
10%
20%
30%


PURITY SCORE
Overall
Active
Passive
0.0
0.2
0.4
0.6
0.8
1.0
10%
20%
30%





5.3 Conclusions

At first, a brief overview of DBSCAN algorithm and federated learning has been given. Then, horizontal and vertical federated learning versions of DBSCAN have been introduced. These versions of the well-known algorithm have the great advantage of preserving privacy , avoiding raw data sharing, when dealing with distributed databases. An implementation of both algorithms has then been presented, using Python programming language.

Each federated version was finally tested using two different datasets, and results were compared to those given by the application of standard DBSCAN. The federated versions proved to be equally efficient, giving slightly different but consistent results. Furthermore, the horizontal federated version was tested in the hypothesis of an increasing number of passive clients, and demonstrated to be resistant to raw data loss, up to 20% of the whole dataset.

6. References

[1] An introduction to clustering algorithms:
Jiawei Han, Micheline Kamber, and Jian Pei. 2011. Data Mining: Concepts and Techniques (3rd. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] A detailed description of DBSCAN:
Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96). AAAI Press, 226–231.

[3] A comprehensive overview of Federated Learning:
Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. ACM Trans. Intell. Syst. Technol. 10, 2, Article 12 (February 2019), 19 pages.