Task1

August 15, 2024

0.1 TOC:

1 Distributed Classification via Logistic Regression

Suppose to have N agents that want to cooperatively determine a nonlinear classifier for a set of points in a given feature space.

1.1 Task 1.1 – Distributed Optimization

As a preliminary task, implement the Gradient Tracking algorithm to solve a consensus optimization problem in the form

$$min_z \sum_{i=1}^{N} l_i(z)$$

with $z \in \mathbb{R}^d$ while $\ell_i : \mathbb{R}^d \mapsto \mathbb{R}$ is a quadratic function, for all $i \in \{1, \dots, N\}$. You can extend the code provided during lectures.

Run a set of simulations to test the effectiveness of the implementation. Moreover, provide a set of solutions that includes different weighted graph patterns (e.g., cycle, path, star) whose weights are determined by the Metropolis-Hastings method. Finally, for each simulation, plot the evolution of the cost function and of the norm of the gradient of the cost function across the iterations.

[ ]:
```
### Libraries
import numpy as np
import networkx as nx
```

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from sympy import symbols, Eq, solve
from scipy.optimize import minimize
import sympy as sp
import sys
import math
from sklearn.model_selection import train_test_split
import matplotlib
import random as rand
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score
```

Quadratic function and it's derivative

[ ]:
```
def quadratic(z, a, b):
    return 0.5*a*z*z+b*z

def grad_quadratic(z, a, b):
    return a*z+b
```

Metropolis hastings methods to define graph weights

[ ]:
```
def metropolis_hastings(N, Adj):

    I = np.eye(N)
    A = np.zeros(shape=(N,N))
    for i in range(N):
        neighbors = np.nonzero(Adj[i])[0]
        degree_i = len(neighbors)

        for j in neighbors:
            degree_j = len(np.nonzero(Adj[j])[0])
            A[i,j] = 1/(1+max([degree_i, degree_j]))

    A += I-np.diag(np.sum(A, axis=0))
    return A
```

Generate strongly connected graph

[ ]:
```
def generate_graph(N, seed):

    I_N = np.eye(N)
    np.random.seed(seed)
    while 1:
        Adj = np.random.binomial(n=1, p=0.3, size=(N, N))
        Adj = np.logical_or(Adj, Adj.T)
        Adj = np.logical_and(Adj, np.logical_not(I_N)).astype(int)
```

```
        test = np.linalg.matrix_power(I_N + Adj, N)
        if np.all(test > 0):
            break
        # else:
        # continue
    return Adj
```

Gradient Tracking

$$z_i^{k+1} = \sum_{j \in N_i} a_{ij} z_j^k - \alpha s_i^k \qquad z_i^0 \in \mathbb{R}$$

$$s_i^{k+1} = \sum_{j \in N_i} a_{ij} s_j^k + \nabla \ell_i \left(z_i^{k+1}\right) - \nabla \ell_i \left(z_i^k\right) \quad s_i^0 = \nabla \ell_i \left(z_i^0\right)$$

[ ]:
```
def gradient_tracking(iterations, N, A,Adj, alpha, a,b):

    Z = np.zeros((iterations, N))
    S = np.zeros((iterations, N))
    for i in range(N):
        S[0,i] = grad_quadratic(Z[0,i], a[i], b[i])

    cost = np.zeros((iterations))

    for k in range(iterations-1):
        for i in range(N):
            neighbors = np.nonzero(Adj[i])[0]

            Z[k+1,i] += Z[k,i] * A[i,i]
            S[k+1,i] += S[k,i] * A[i,i]

            for j in neighbors:
                Z[k+1,i] += Z[k,j] * A[i,j]
                S[k+1,i] += S[k,j] * A[i,j]

            Z[k+1, i] -= alpha * S[k,i]

            nabla_new = grad_quadratic(Z[k+1,i], a[i], b[i])
            nabla_old = grad_quadratic(Z[k,i], a[i], b[i])
            S[k+1,i] +=  nabla_new - nabla_old

            cost[k] += quadratic(Z[k,i], a[i], b[i])

    return Z,S, cost
```

[ ]:
```
def plot_results(iterations, Z,S, cost):
```

```
    fig, ax = plt.subplots()
    ax.plot(np.arange(iterations), Z)
    ax.title.set_text('Agents\' States')
    ax.grid()

    fig, ax = plt.subplots()
    ax.semilogy(np.arange(iterations), S)
    ax.title.set_text('Agents\' Gradient')
    ax.grid()

    fig, ax = plt.subplots()
    average_gradient = np.mean(S, axis=1)
    ax.semilogy(np.arange(iterations), average_gradient)
    ax.title.set_text('Average Gradient')
    ax.grid()


    Z_opt = -np.sum(b) / np.sum(a)
    opt_cost = 0.5 * np.sum(a) * Z_opt**2 + np.sum(b) * Z_opt

    fig, ax = plt.subplots()
    ax.title.set_text('Cost')
    ax.semilogy(np.arange(iterations-1), np.abs(cost[:-1] - opt_cost))
    ax.grid()

    plt.show()
```

Parameters

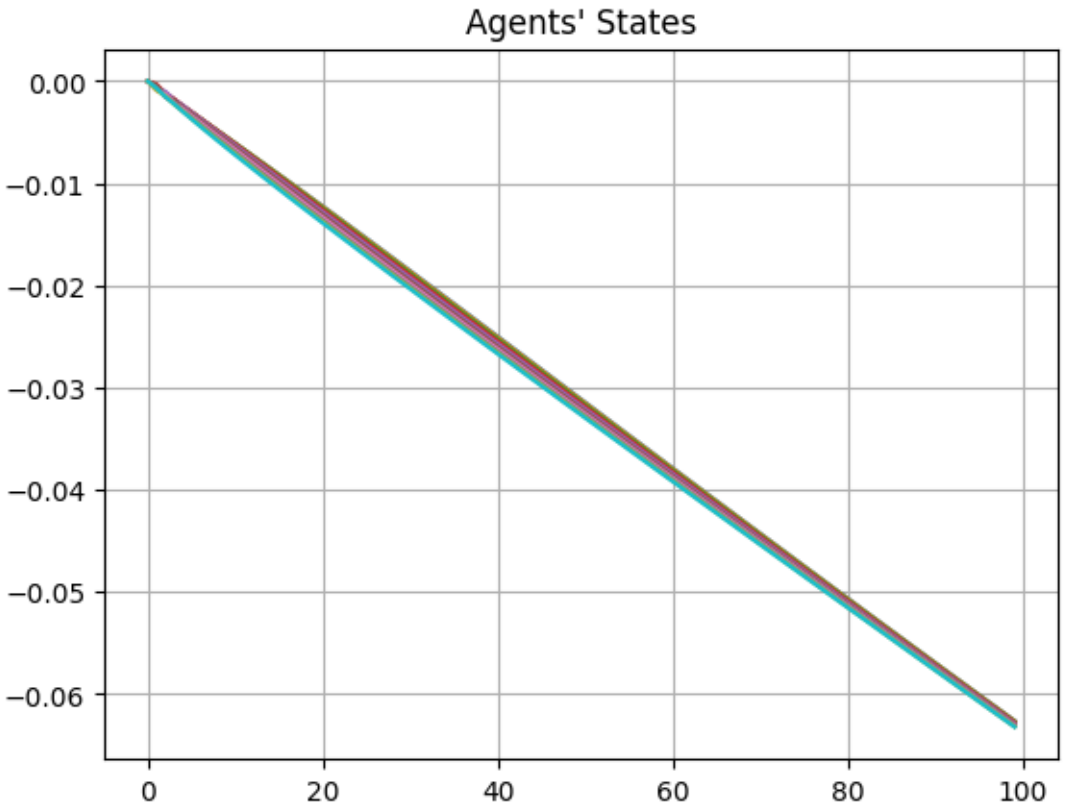
[ ]:
```
seed = 0 #random seed
iterations = 100 #number of iterations
N = 10 #number of agents
alpha = 1e-3 #step-size

a = np.random.uniform(size=N)
b = np.random.uniform(size=N)

Adj = generate_graph(N,seed)
A = metropolis_hastings(N, Adj)
```

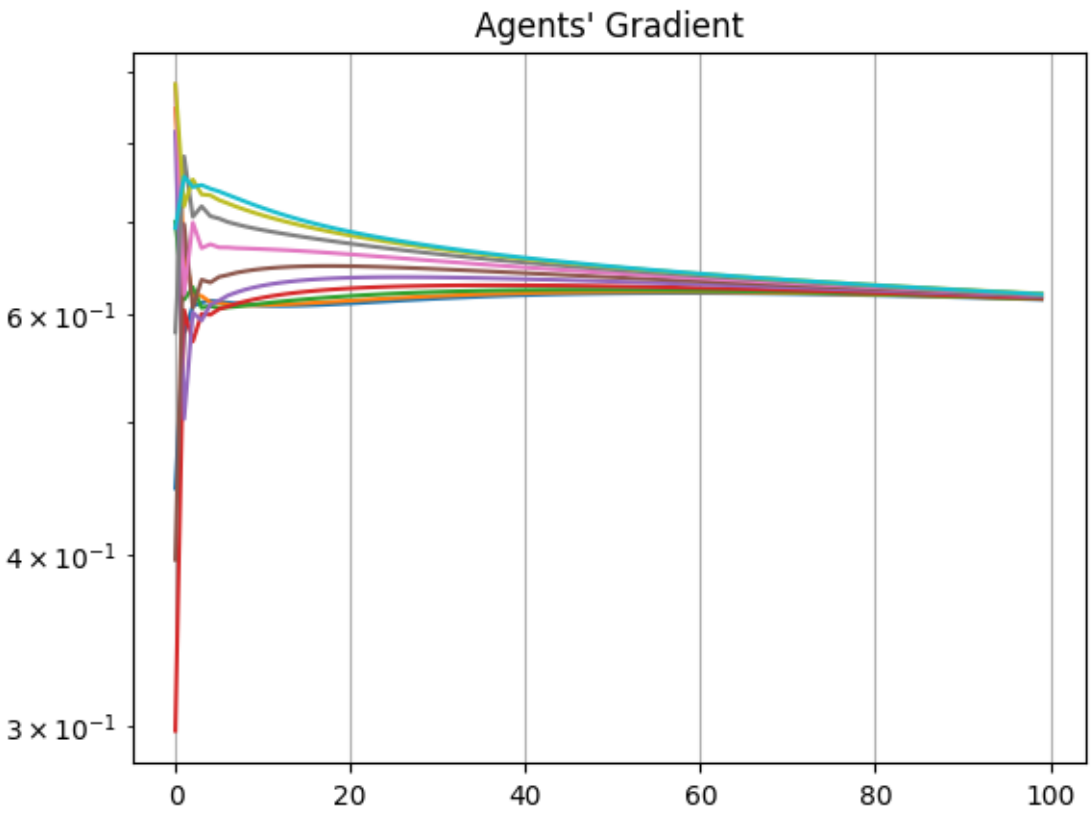
[ ]:
```
#generate graph
G = nx.path_graph(N)
Adj = nx.adjacency_matrix(G).toarray()
A = metropolis_hastings(N, Adj)
```

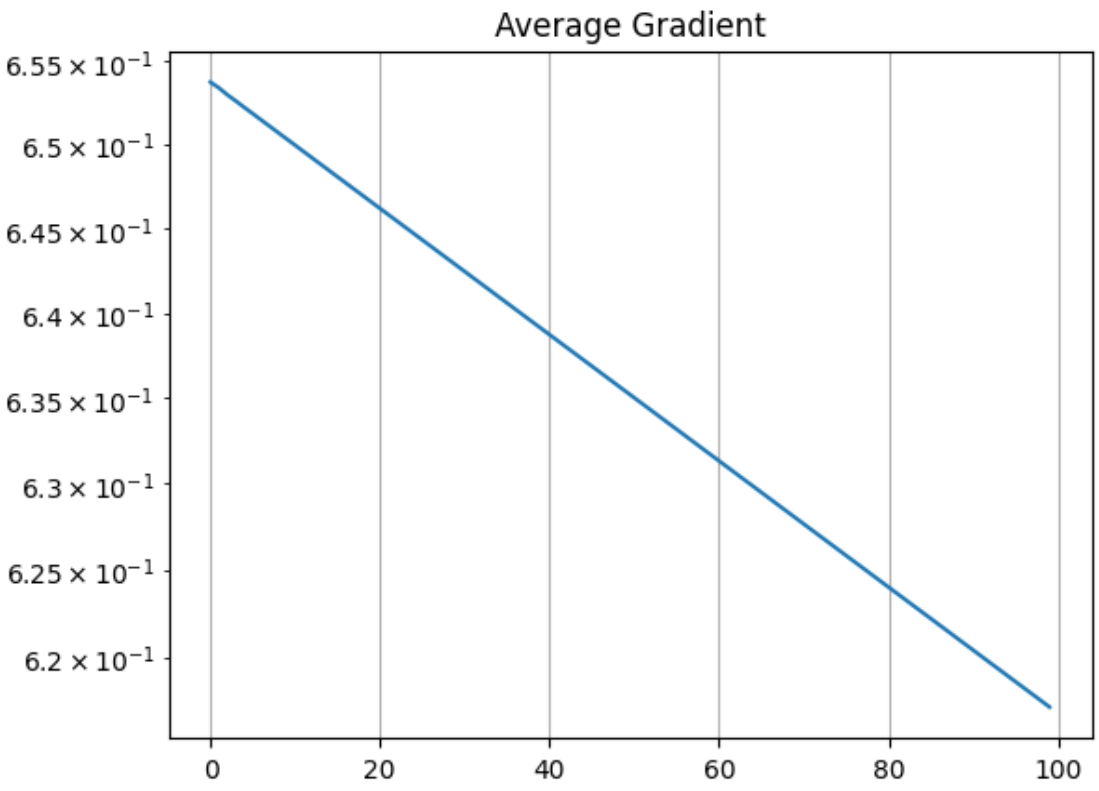
Simulations

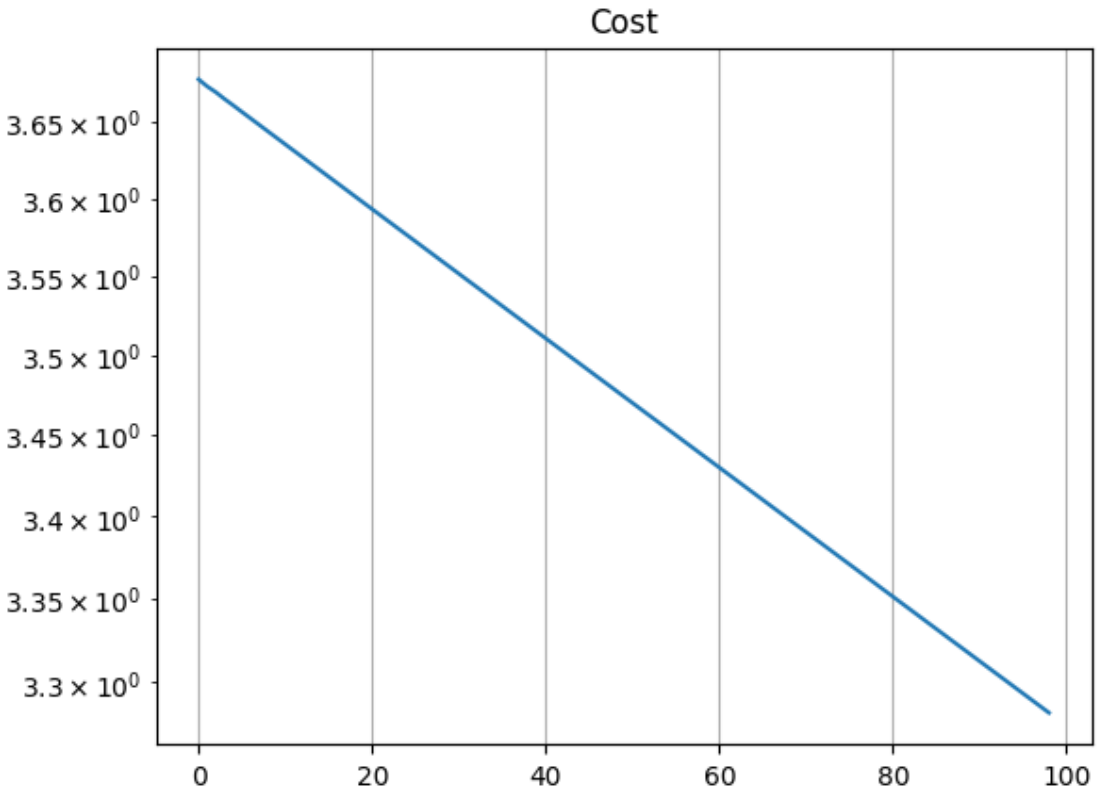
[ ]:
```
#launch gradient tracking algorithm
Z,S, cost = gradient_tracking(iterations, N, A, Adj, alpha, a, b)
```

[ ]:
```
plot_results(iterations, Z,S, cost)
```




Agents' Gradient
6×10^{-1}
4×10^{-1}
3×10^{-1}
0
20
40
60
80
100


Average Gradient
6.55×10^{-1}
6.5×10^{-1}
6.45×10^{-1}
6.4×10^{-1}
6.35×10^{-1}
6.3×10^{-1}
6.25×10^{-1}
6.2×10^{-1}
0
20
40
60
80
100



1.2 Task 1.2 - Centralized Classification

1. Generate a dataset of $\mathcal{M} \in \mathbb{N}$ points $\mathcal{D}^m \in \mathbb{R}^d, m = 1, \ldots, \mathcal{M}$.
2. Label the points with a binary label $p^m \in \{-1, 1\}$ for all $m = 1, \ldots, \mathcal{M}$. The separating function is given in the form

$$\left\{x \in \mathbb{R}^d \mid w^\top \varphi(x) + b = 0\right\},$$

with $\varphi : \mathbb{R}^d \to \mathbb{R}^q$ a nonlinear function and $w \in \mathbb{R}^q, b \in \mathbb{R}$ parameters. Hence,

$$\begin{aligned} w^\top \varphi\left(\mathcal{D}^m\right) + b \geq 0, &\quad \text{if } p^m = 1, \text{ and} \\ w^\top \varphi\left(\mathcal{D}^m\right) + b < 0, &\quad \text{if } p^m = -1. \end{aligned}$$

Hint. In a two-dimensional example, assuming $\mathcal{D} = ([\mathcal{D}]_1, [\mathcal{D}]_2) \in \mathbb{R}^2$, the separating function can be an ellipse. To this end, define

$$\varphi(\mathcal{D}) := \begin{bmatrix} [\mathcal{D}]_1 \\ [\mathcal{D}]_2 \\ ([\mathcal{D}]_1)^2 \\ ([\mathcal{D}]_2)^2 \end{bmatrix}$$

3. Implement a (centralized) Gradient Method to minimize a Logistic Regression Function in order to classify the points. The resulting optimization problem is

$$\min_{w,b} \sum_{m=1}^{\mathcal{M}} \log\left(1+\exp\left(-p^m\left(w^\top \varphi\left(\mathcal{D}^m\right)+b\right)\right)\right)$$

where $(w,b) \in \mathbb{R}^q \times \mathbb{R}$ is the optimization variable.

4. Run a set of simulations to test the effectiveness of the implementation. Moreover, provide a set of solutions that includes, e.g., different dataset patterns. Finally, for each simulation, plot the evolution of the cost function and of the norm of the gradient of the cost function across the iterations.

Auxilliary functions

[ ]:
```
def compute_semiaxes(a, b, c, d, g, center):

    x, y = symbols('x y')
    x_c, y_c = center

    # Initialize intersection points
    x_ints = [0, 0]
    y_ints = [0, 0]

    # Solve for x when y = yc to find x-intersections
    eq_x = a*x + b*y_c + c*x**2 + d*y_c**2 + g
    x_ints = sp.solve(eq_x, x)

    # Solve for y when x = xc to find y-intersections
    eq_y = a*x_c + b*y + c*x_c**2 + d*y**2 + g #put a plus to avoid complex␣
↪solutions
    y_ints = sp.solve(eq_y, y)

    #add the center
    x_ints.extend([x_c])
    y_ints.extend([y_c])

    #find the distances from the origin
    min_ox = min(x_ints)
    max_ox = max(x_ints)
    min_oy = min(y_ints)
    max_oy = max(y_ints)
    dist_o = [min_ox, max_ox, min_oy, max_oy]

    return dist_o, x_ints, y_ints

def compute_center(a, b, c, d, g):
```

```
    #define symbolic letters
    x, y = symbols('x y')

    #define the equation
    eq = a*x + b * y + c* (x**2) + d * (y**2) - g

    #solve system of equations to find center's coordinates
    sol = solve((eq.diff(x), eq.diff(y)), (x, y))

    if sol: #there is a solution (non-empty list)
        center = [0,0]
        center[0] = float(sol[x])
        center[1] = float(sol[y])

    else: #there is no center (empty list)

        print("\nThe conic function has no predefined center.")
        center = None

    return center
```

[ ]:
```
def generate_two_dimensional_function(random, shape, desired):

    if not random:
        if desired is not None:
            return desired, 1
        shapes = {
            'circle': ([0, 0, 1, 1], -1),
            'vert_ellipse': ([0, 0, 5 ,1], -1),
            'hor_ellipse': ([0, 0, 1, 5], -1),
            'parabola': ([1, 1, 0, 1], -1)
        }
        if shape in shapes:
            return shapes[shape]
        else:
            print("\nConic type not allowed!\n")
            sys.exit(0)
    else:
        return np.random.rand(4), -1 * np.random.rand()


def generate_multi_dimensional_function(q):
    return np.random.rand(q), -1 * np.random.rand()
```

[ ]:
```
def find_conic_parameters(random, q, shape=None, desired=None):

    if q == 4:
```

```
        w, b = generate_two_dimensional_function(random, shape, desired)
    else:
        w, b = generate_multi_dimensional_function(q)

    center = compute_center(w[0], w[1], w[2], w[3], b)

    if center != None:
        dist_o, x_ints, y_ints = compute_semiaxes(w[0], w[1], w[2], w[3], b,␣
↪center)

    else:
        dist_o = x_ints = y_ints = None

    return w , b, center, dist_o, x_ints, y_ints
```

[ ]:
```
def dataset_plot(dataset, conic_parameters, datasets=None, regression=None,␣
↪w=None, b=None, x_vals=None, y_vals=None, set_type=None, centered_data=None,␣
↪margin=None, spread=None, random=None, markers=None):

    '''This function plots the dataset, either centered in the
    conic's center or centered in the axes' origin, according
    to the parameters passed as arguments'''
    center = conic_parameters["center"]
    x_ints= conic_parameters["x_ints"]
    y_ints = conic_parameters["y_ints"]
    dist_o = conic_parameters["dist_o"]
    #set the labels
    plt.figure()

    if not regression:
        plt.title('Labeled Dataset')
        plt.xlabel('Features')
        plt.ylabel('Labels')

    elif set_type == "train":
        plt.title('Classified Training Set')
        plt.xlabel('Features')
        plt.ylabel('Labels')

    elif set_type == 'test':
        plt.title('Classified Test Set')
        plt.xlabel('Features')
        plt.ylabel('Labels')

    #plt.title(f'Plot of equation {w_opt[3]:.2f}*y^2 + {w_opt[1]:.2f}*y +␣
↪{w_opt[2]:.2f}*x^2 + {w_opt[0]:.2f}*x {b_opt:.2f} = 0')
```

```
    #centralized data visualization
    if datasets is None:

        #label the points
        one_labeled = [dataset[m][0] for m in range(len(dataset)) if␣
↪dataset[m][1] == 1] #before it was D[m] for m in range(M)
        minus_one_labeled = [dataset[m][0] for m in range(len(dataset)) if␣
↪dataset[m][1] == -1]

        #plot the labelled points
        plt.plot([x[0] for x in one_labeled], [x[1] for x in one_labeled], 'bo')
        plt.plot([x[0] for x in minus_one_labeled], [x[1] for x in␣
↪minus_one_labeled], 'ro')

    #distributed data visualization
    else:

        for agent_idx, dataset_agent in enumerate(datasets):

            #label the points
            one_labeled = [point[0] for point in dataset_agent if point[1] == 1]
            minus_one_labeled = [point[0] for point in dataset_agent if␣
↪point[1] == -1]

            # plot the labelled points
            plt.plot([x[0] for x in one_labeled], [x[1] for x in one_labeled],␣
↪'bo', marker=markers[agent_idx % len(markers)], linestyle='', label=f'Class␣
↪1 - Agent {agent_idx+1}')
            plt.plot([x[0] for x in minus_one_labeled], [x[1] for x in␣
↪minus_one_labeled], 'ro', marker=markers[agent_idx % len(markers)],␣
↪linestyle='', label=f'Class -1 - Agent {agent_idx+1}')

    #plot the regression function
    if regression: plt.plot(x_vals, y_vals, color='blue')

    #set the plot's dimensions
    if not random or not centered_data:
        plt.xlim(-spread, spread)
        plt.ylim(-spread, spread)
    else:
        plt.xlim(float(dist_o[0])-margin, float(dist_o[1])+margin)
        plt.ylim(float(dist_o[2])-margin, float(dist_o[3])+margin)

    #plot the center and intersection's coordinates
    if random and centered_data:
```

```
        #extract the center's coordinates
        center_x = center[0]
        center_y = center[1]

        #plot the center
        plt.scatter(center[0], center[1], color='green', label = 'Center')
        plt.axhline(y=center[1], color='gray', linestyle='--', label='Asse y')
        plt.axvline(x=center[0], color='gray', linestyle='--', label='Asse x')

        #plot the intersections between conic and center's axes
        plt.scatter(x_ints[0], center_y, color='green', label = 'P1')
        plt.scatter(x_ints[1], center_y, color='green', label = 'P2')
        plt.scatter(center_x, y_ints[0], color='green', label = 'P3')
        plt.scatter(center_x, y_ints[1], color='green', label = 'P4')

        plt.gca().set_aspect('auto', adjustable='box')
        plt.grid(True)

    plt.show()
```

[ ]:
```
def phi(D): #non-linear function

    return np.array([D[0], D[1], D[0]**2, D[1]**2])
```

[ ]:
```
def P(x, w, b):  #labels' computation

    if np.dot(w, phi(x)) + b >= 0:
        return 1
    else:
        return -1
```

[ ]:
```
def generate_dataset(M, n, spread, train_size, conic_parameters, centered_data,␣
 ↪margin, random):

    """
    Parameters:
        M (int): Number of points in the dataset.
        n (int): Dimension of the datasets
        w (numpy array): Weight vector of shape (q,) for the separating␣
 ↪function.
        b (float): Bias term for the separating function.
        spread (int): Spreading factor for plot axes
        plot (bool): boolean variable to decide whether plot the dataset
        center: center of the conic
        fires: fires of the conic

    Returns:
```

```
        X (numpy array): Array of shape (M, 2) containing the coordinates of␣
↪the points.
        y (numpy array): Array of shape (M,) containing the labels (-1 or 1)␣
↪for each point.
    """

    #generate a dataset fixed in dimension, either random or deterministic
    if not random or not centered_data: D = np.random.uniform(-spread,spread,␣
↪size=(M, n))

    #generate a random dataset centered in the conic's center if random is True
    else: D = np.random.uniform(low=[conic_parameters["dist_o"][0]-margin,␣
↪conic_parameters["dist_o"][2]-margin],␣
↪high=[conic_parameters["dist_o"][1]+margin,conic_parameters["dist_o"][3]+margin],␣
↪size=(M, n))

    #assign labels (prob.) to each point
    dataset = [(D[m], P(D[m],conic_parameters["w"],conic_parameters["b"])) for␣
↪m in range(M)]

    #split the dataset into training and test sets (hold-out method)
    D_train, D_test = train_test_split(dataset, train_size=train_size,␣
↪random_state=42) #think of using X and y

    return dataset, D_train, D_test
```

1.2.1 Dataset Generation

Select Conic Shape Available shapes: 'circle'-'vert_ellipse'-'hor_ellipse'-'parabola'

[ ]:
```
random = False
q = 4
shape = 'vert_ellipse'
```

[ ]:
```
conic_parameters = {

    'w': [], 'b': [],
    'center': [],
    'dist_o': [],
    'x_ints': [], 'y_ints': []}

conic_parameters['w'], conic_parameters['b'],\
conic_parameters['center'], conic_parameters['dist_o'],\
conic_parameters['x_ints'], conic_parameters['y_ints'] =␣
 ↪find_conic_parameters(random, q, shape)
```

[ ]:
```
if conic_parameters['center'] is not None:

    print("\n\n-----------------------------------------------------------------")
    print("CONIC GENERATION")
    print("Center coordinates: ", conic_parameters["center"])
    print("X intersection points: ", conic_parameters["x_ints"])
    print("Y intersection points: ",conic_parameters["y_ints"] )
    print("Minimum distance x from the origin: ", conic_parameters["dist_o"][0])
    print("Maximum distance x from the origin: ", conic_parameters["dist_o"][1])
    print("Minimum distance y from the origin: ", conic_parameters["dist_o"][2])
    print("Maximum distance y from the origin: ", conic_parameters["dist_o"][3])
    print("-----------------------------------------------------------------")
    print(f"\nw: {conic_parameters['w']}\nb: {conic_parameters['b']}\n")
```

```
-----------------------------------------------------------------
CONIC GENERATION
Center coordinates:  [0.0, 0.0]
X intersection points:  [-sqrt(5)/5, sqrt(5)/5, 0.0]
Y intersection points:  [-1, 1, 0.0]
Minimum distance x from the origin:  -sqrt(5)/5
Maximum distance x from the origin:  sqrt(5)/5
Minimum distance y from the origin:  -1
Maximum distance y from the origin:  1
-----------------------------------------------------------------

w: [0, 0, 5, 1]
b: -1
```

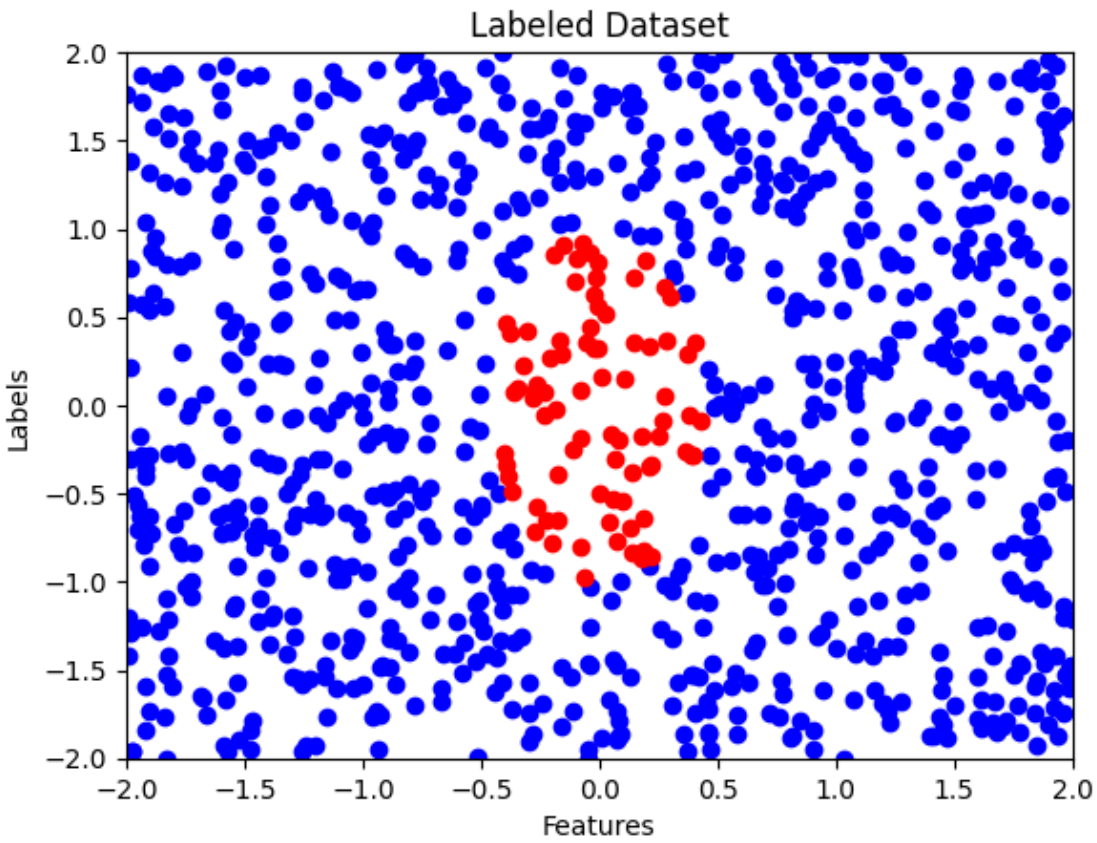
Parameters Also the spread of the points, the train size, if centering the data

[ ]:
```
M = 1000
n = 2
spread = 2
train_size = 0.7
centered_data = True
margin = 0.5
random = False
```

Points Generation with Labels

[ ]:
```
dataset, D_train, D_test = generate_dataset(M, n, spread, train_size,␣
 ↪conic_parameters, centered_data, margin, random)
```

[ ]:
```
dataset_plot(dataset, conic_parameters, centered_data=centered_data,␣
 ↪margin=margin, spread=spread, random=random)
```



1.2.2 Classification

[ ]:
```python
def logistic_regression_loss(w, b, D, p):

    #initialize the loss function
    loss = 0

    #compute the loss function for all the data-points
    for m in range(len(D)):
        z = -p[m] * (np.dot(w, phi(D[m])) + b)
        z = np.clip(z, -np.inf, 500)  # Clip values to prevent overflow
        loss += np.log(1 + np.exp(z))

    #initialize the gradients
    grad_w = np.zeros(len(w))
    grad_b = 0.0

    #compute the gradients of the loss function
    for m in range(len(D)):
        z = -p[m] * (np.dot(w, phi(D[m])) + b)
```

```
        z = np.clip(z, -np.inf, 500)  # Clip values to prevent overflow
        exp_term = np.exp(z)
        grad_w += -p[m] * exp_term / (1 + exp_term) * phi(D[m])
        grad_b += -p[m] * exp_term / (1 + exp_term)

    return loss, grad_w, grad_b
```

[ ]:
```
def centralized_gradient_method(D, p, step_size, max_iterations, min_grad_norm):

    #initialization
    q = len(phi(D[0]))
    w = np.random.rand(q)
    b = np.random.rand()
    costs = []
    grad_norms = []
    effective_iterations = 0

    for k in range(max_iterations - 1):
        effective_iterations += 1
        #compute the gradient of the cost function
        loss, grad_w, grad_b = logistic_regression_loss(w, b, D, p)
        grad_norm = np.sqrt(np.sum(grad_w ** 2) + grad_b ** 2)

        #update w and b using the gradient descent
        w -= step_size * grad_w
        b -= step_size * grad_b

        #store cost and gradient norm
        costs.append(loss)
        grad_norms.append(grad_norm)

        #print progress
        if k % 200 == 0: print(f"Iteration [{k}]: Cost = {loss}, Gradient Norm␣
↪= {grad_norm}")

        #stop the algorithm
        if grad_norm < min_grad_norm:

            print(f"Gradient norm is below threshold ({min_grad_norm}).␣
↪Stopping optimization.")
            break

    return w, b, costs, grad_norms, effective_iterations
```

[ ]:
```
print(D_train[0][1])
```

```
1
```

[ ]:
```
points = np.array([data[0] for data in D_train]) #P
labels = np.array([data[1] for data in D_train]) #D
```

1.2.3 Centralized Gradient Method

Parameters

[ ]:
```
step_size = 0.05
max_iters = int(1e6) #maximum number of iterations
min_grad_norm = 0.02 #gradient method's stopping criterion
```

Algorithm

[ ]:
```
w_opt, b_opt, costs, grad_norms,effective_iterations =␣
↪centralized_gradient_method(points, labels, step_size, max_iters,␣
↪min_grad_norm)
```

```
Iteration [0]: Cost = 146.11085272043263, Gradient Norm = 79.3940485683614
Iteration [200]: Cost = 22.69605564790747, Gradient Norm = 0.8992116903811985
Iteration [400]: Cost = 17.574387314492665, Gradient Norm = 0.5777959490981142
Iteration [600]: Cost = 15.02367503665948, Gradient Norm = 0.4438145868314652
Iteration [800]: Cost = 13.395023862531673, Gradient Norm = 0.36795068005313286
Iteration [1000]: Cost = 12.226010407260434, Gradient Norm = 0.3181982030263275
Iteration [1200]: Cost = 11.327663133493632, Gradient Norm = 0.28259289642560037
Iteration [1400]: Cost = 10.60582343792548, Gradient Norm = 0.25559393153654186
Iteration [1600]: Cost = 10.007315068925434, Gradient Norm = 0.2342626194067311
Iteration [1800]: Cost = 9.499381189863493, Gradient Norm = 0.2168857429402624
Iteration [2000]: Cost = 9.060511580627773, Gradient Norm = 0.20239203237301864
Iteration [2200]: Cost = 8.675871612899025, Gradient Norm = 0.19007439553669775
Iteration [2400]: Cost = 8.33482598530766, Gradient Norm = 0.17944558227604196
Iteration [2600]: Cost = 8.029506547767497, Gradient Norm = 0.170157841365983
Iteration [2800]: Cost = 7.753939140467465, Gradient Norm = 0.16195564114771122
Iteration [3000]: Cost = 7.503487619423381, Gradient Norm = 0.1546465394165999
Iteration [3200]: Cost = 7.274486718779088, Gradient Norm = 0.14808252891818607
Iteration [3400]: Cost = 7.063991984973787, Gradient Norm = 0.14214769120584006
Iteration [3600]: Cost = 6.869604847844546, Gradient Norm = 0.13674978983958055
Iteration [3800]: Cost = 6.689347380044743, Gradient Norm = 0.13181440218118676
Iteration [4000]: Cost = 6.521570785864463, Gradient Norm = 0.12728073278173035
Iteration [4200]: Cost = 6.3648873196585285, Gradient Norm = 0.12309856807651723
Iteration [4400]: Cost = 6.21811881542365, Gradient Norm = 0.1192260226182327
Iteration [4600]: Cost = 6.080257210483214, Gradient Norm = 0.11562784499755809
Iteration [4800]: Cost = 5.950433872926113, Gradient Norm = 0.11227412646976982
Iteration [5000]: Cost = 5.827895487671455, Gradient Norm = 0.1091393039439229
Iteration [5200]: Cost = 5.711984894872231, Gradient Norm = 0.10620138124986953
Iteration [5400]: Cost = 5.602125714013101, Gradient Norm = 0.10344131439998133
Iteration [5600]: Cost = 5.497809894640755, Gradient Norm = 0.10084252155211439
Iteration [5800]: Cost = 5.398587553127734, Gradient Norm = 0.0983904888499624
Iteration [6000]: Cost = 5.304058612198788, Gradient Norm = 0.09607245073617063
Iteration [6200]: Cost = 5.213865874709597, Gradient Norm = 0.0938771286618929
```

```
Iteration [6400]: Cost = 5.127689247870036, Gradient Norm = 0.09179451599078893
Iteration [6600]: Cost = 5.045240897313949, Gradient Norm = 0.08981569974510184
Iteration [6800]: Cost = 4.966261158068406, Gradient Norm = 0.08793271195997283
Iteration [7000]: Cost = 4.890515065741938, Gradient Norm = 0.08613840500291914
Iteration [7200]: Cost = 4.817789399099408, Gradient Norm = 0.08442634642110068
Iteration [7400]: Cost = 4.747890146753339, Gradient Norm = 0.08279072980089337
Iteration [7600]: Cost = 4.6806403275280095, Gradient Norm = 0.08122629883497616
Iteration [7800]: Cost = 4.615878107279316, Gradient Norm = 0.0797282823442771
Iteration [8000]: Cost = 4.553455165424262, Gradient Norm = 0.07829233843423458
Iteration [8200]: Cost = 4.493235272775679, Gradient Norm = 0.07691450630532722
Iteration [8400]: Cost = 4.435093048965389, Gradient Norm = 0.07559116450789295
Iteration [8600]: Cost = 4.378912873132167, Gradient Norm = 0.07431899464679677
Iteration [8800]: Cost = 4.324587925923333, Gradient Norm = 0.07309494971452954
Iteration [9000]: Cost = 4.272019344423123, Gradient Norm = 0.07191622637097973
Iteration [9200]: Cost = 4.221115474541259, Gradient Norm = 0.07078024060147632
Iteration [9400]: Cost = 4.171791207798048, Gradient Norm = 0.06968460627713083
Iteration [9600]: Cost = 4.1239673914300505, Gradient Norm = 0.06862711621727591
Iteration [9800]: Cost = 4.0775703023903525, Gradient Norm = 0.06760572541616258
Iteration [10000]: Cost = 4.0325311771937455, Gradient Norm =
0.06661853614766258
Iteration [10200]: Cost = 3.9887857907089583, Gradient Norm =
0.06566378470454022
Iteration [10400]: Cost = 3.9462740779683685, Gradient Norm =
0.06473982956456988
Iteration [10600]: Cost = 3.9049397938817005, Gradient Norm =
0.06384514080565112
Iteration [10800]: Cost = 3.864730206431592, Gradient Norm = 0.06297829061717793
Iteration [11000]: Cost = 3.8255958195151307, Gradient Norm =
0.06213794477607049
Iteration [11200]: Cost = 3.7874901220955755, Gradient Norm =
0.06132285497377959
Iteration [11400]: Cost = 3.7503693607551547, Gradient Norm =
0.060531851895753726
Iteration [11600]: Cost = 3.7141923331060838, Gradient Norm =
0.05976383896778822
Iteration [11800]: Cost = 3.6789201998310994, Gradient Norm =
0.059017786694711936
Iteration [12000]: Cost = 3.644516313395999, Gradient Norm = 0.058292727526317
Iteration [12200]: Cost = 3.610946061710811, Gradient Norm = 0.05758775119355
Iteration [12400]: Cost = 3.5781767252189556, Gradient Norm =
0.05690200046497459
Iteration [12600]: Cost = 3.546177346069969, Gradient Norm =
0.056234667279541045
Iteration [12800]: Cost = 3.5149186081845873, Gradient Norm =
0.055584989216921614
Iteration [13000]: Cost = 3.4843727271548586, Gradient Norm =
0.05495224627119991
Iteration [13200]: Cost = 3.4545133490387117, Gradient Norm =
```

```
0.05433575789763358
Iteration [13400]: Cost = 3.425315457211019, Gradient Norm =
0.053734880305649074
Iteration [13600]: Cost = 3.3967552865231108, Gradient Norm =
0.05314900397422107
Iteration [13800]: Cost = 3.3688102441018057, Gradient Norm =
0.05257755136840944
Iteration [14000]: Cost = 3.3414588361888504, Gradient Norm =
0.05201997483813706
Iteration [14200]: Cost = 3.3146806004832894, Gradient Norm =
0.05147575468230311
Iteration [14400]: Cost = 3.2884560435037473, Gradient Norm =
0.05094439736311724
Iteration [14600]: Cost = 3.2627665825360146, Gradient Norm =
0.05042543385710579
Iteration [14800]: Cost = 3.2375944917741823, Gradient Norm =
0.049918418130631276
Iteration [15000]: Cost = 3.2129228523017543, Gradient Norm =
0.04942292572899744
Iteration [15200]: Cost = 3.188735505593113, Gradient Norm = 0.04893855246929943
Iteration [15400]: Cost = 3.1650170102462254, Gradient Norm =
0.04846491322815427
Iteration [15600]: Cost = 3.1417526016843045, Gradient Norm = 0.0480016408162992
Iteration [15800]: Cost = 3.1189281545886343, Gradient Norm =
0.04754838493282429
Iteration [16000]: Cost = 3.096530147846423, Gradient Norm = 0.04710481119248413
Iteration [16200]: Cost = 3.074545631817053, Gradient Norm =
0.046670600220150925
Iteration [16400]: Cost = 3.052962197737588, Gradient Norm =
0.046245446807018135
Iteration [16600]: Cost = 3.031767949104399, Gradient Norm = 0.04582905912365817
Iteration [16800]: Cost = 3.010951474881666, Gradient Norm = 0.04542115798547773
Iteration [17000]: Cost = 2.9905018244006407, Gradient Norm =
0.04502147616651247
Iteration [17200]: Cost = 2.9704084838249454, Gradient Norm =
0.04462975775785971
Iteration [17400]: Cost = 2.9506613540678366, Gradient Norm =
0.044245757567374214
Iteration [17600]: Cost = 2.9312507300567163, Gradient Norm =
0.04386924055753517
Iteration [17800]: Cost = 2.9121672812488804, Gradient Norm =
0.04349998131866298
Iteration [18000]: Cost = 2.8934020333102826, Gradient Norm =
0.04313776357490068
Iteration [18200]: Cost = 2.8749463508761552, Gradient Norm =
0.042782379720584546
Iteration [18400]: Cost = 2.8567919213188953, Gradient Norm =
0.04243363038483311
```

```
Iteration [18600]: Cost = 2.838930739454344, Gradient Norm = 0.04209132402235316
Iteration [18800]: Cost = 2.821355093123144, Gradient Norm = 0.04175527652862631
Iteration [19000]: Cost = 2.8040575495885802, Gradient Norm =
0.04142531087778671
Iteration [19200]: Cost = 2.7870309426968936, Gradient Norm =
0.041101256781631276
Iteration [19400]: Cost = 2.7702683607501015, Gradient Norm =
0.04078295036832725
Iteration [19600]: Cost = 2.75376313504505, Gradient Norm = 0.040470233879493674
Iteration [19800]: Cost = 2.7375088290360083, Gradient Norm =
0.040162955384432805
Iteration [20000]: Cost = 2.7214992280810217, Gradient Norm =
0.039860968510381796
Iteration [20200]: Cost = 2.7057283297353742, Gradient Norm =
0.03956413218773946
Iteration [20400]: Cost = 2.6901903345578906, Gradient Norm =
0.039272310409298766
Iteration [20600]: Cost = 2.674879637398523, Gradient Norm = 0.03898537200259133
Iteration [20800]: Cost = 2.6597908191376636, Gradient Norm = 0.0387031904145099
Iteration [21000]: Cost = 2.6449186388498385, Gradient Norm =
0.03842564350744046
Iteration [21200]: Cost = 2.6302580263662634, Gradient Norm =
0.03815261336618528
Iteration [21400]: Cost = 2.615804075212532, Gradient Norm = 0.03788398611501471
Iteration [21600]: Cost = 2.60155203589926, Gradient Norm = 0.03761965174422612
Iteration [21800]: Cost = 2.587497309545113, Gradient Norm =
0.037359503945637064
Iteration [22000]: Cost = 2.57363544181292, Gradient Norm = 0.03710343995647602
Iteration [22200]: Cost = 2.559962117140869, Gradient Norm = 0.0368513604111706
Iteration [22400]: Cost = 2.546473153252028, Gradient Norm =
0.036603169200570135
Iteration [22600]: Cost = 2.533164495926456, Gradient Norm = 0.03635877333816544
Iteration [22800]: Cost = 2.520032214021195, Gradient Norm =
0.036118082832903936
Iteration [23000]: Cost = 2.5070724947244223, Gradient Norm =
0.03588101056821868
Iteration [23200]: Cost = 2.4942816390308713, Gradient Norm =
0.035647472186920603
Iteration [23400]: Cost = 2.481656057426397, Gradient Norm = 0.03541738598161876
Iteration [23600]: Cost = 2.469192265770472, Gradient Norm = 0.03519067279036528
Iteration [23800]: Cost = 2.456886881365883, Gradient Norm =
0.034967255897228613
Iteration [24000]: Cost = 2.4447366192057247, Gradient Norm =
0.03474706093752911
Iteration [24200]: Cost = 2.432738288388329, Gradient Norm =
0.034530015807479005
Iteration [24400]: Cost = 2.420888788691311, Gradient Norm = 0.03431605057798993
Iteration [24600]: Cost = 2.409185107296482, Gradient Norm = 0.03410509741242384
```

```
Iteration [24800]: Cost = 2.39762431565781, Gradient Norm = 0.033897090488076605
Iteration [25000]: Cost = 2.386203566505216, Gradient Norm = 0.03369196592119737
Iteration [25200]: Cost = 2.374920090977169, Gradient Norm =
0.033489661695358774
Iteration [25400]: Cost = 2.3637711958757293, Gradient Norm =
0.03329011759300269
Iteration [25600]: Cost = 2.3527542610377883, Gradient Norm =
0.033093275129997164
Iteration [25800]: Cost = 2.341866736816892, Gradient Norm = 0.03289907749305207
Iteration [26000]: Cost = 2.3311061416700602, Gradient Norm =
0.03270746947984557
Iteration [26200]: Cost = 2.32047005984456, Gradient Norm = 0.03251839744172711
Iteration [26400]: Cost = 2.3099561391597825, Gradient Norm =
0.03233180922886511
Iteration [26600]: Cost = 2.299562088879565, Gradient Norm = 0.03214765413772029
Iteration [26800]: Cost = 2.289285677670776, Gradient Norm =
0.031965882860728226
Iteration [27000]: Cost = 2.279124731643914, Gradient Norm = 0.03178644743808305
Iteration [27200]: Cost = 2.2690771324719985, Gradient Norm = 0.031609301211521
Iteration [27400]: Cost = 2.2591408155839523, Gradient Norm =
0.03143439878000549
Iteration [27600]: Cost = 2.2493137684291833, Gradient Norm =
0.031261695957224826
Iteration [27800]: Cost = 2.2395940288098926, Gradient Norm =
0.031091149730813176
Iteration [28000]: Cost = 2.229979683278189, Gradient Norm =
0.030922718223217134
Iteration [28200]: Cost = 2.2204688655949143, Gradient Norm =
0.030756360654126444
Iteration [28400]: Cost = 2.2110597552475224, Gradient Norm =
0.03059203730439954
Iteration [28600]: Cost = 2.201750576024243, Gradient Norm =
0.030429709481412277
Iteration [28800]: Cost = 2.192539594642078, Gradient Norm =
0.030269339485764962
Iteration [29000]: Cost = 2.183425119426349, Gradient Norm =
0.030110890579287435
Iteration [29200]: Cost = 2.1744054990392767, Gradient Norm =
0.02995432695427924
Iteration [29400]: Cost = 2.165479121255747, Gradient Norm =
0.029799613703934368
Iteration [29600]: Cost = 2.156644411783985, Gradient Norm =
0.029646716793892615
Iteration [29800]: Cost = 2.1478998331293515, Gradient Norm =
0.029495603034870876
Iteration [30000]: Cost = 2.1392438834993324, Gradient Norm =
0.02934624005632516
Iteration [30200]: Cost = 2.130675095748009, Gradient Norm =
```

```
0.029198596281098894
Iteration [30400]: Cost = 2.122192036358332, Gradient Norm =
0.029052640901012675
Iteration [30600]: Cost = 2.11379330446059, Gradient Norm = 0.028908343853357145
Iteration [30800]: Cost = 2.105477530885589, Gradient Norm =
0.028765675798248066
Iteration [31000]: Cost = 2.0972433772511447, Gradient Norm =
0.028624608096808477
Iteration [31200]: Cost = 2.0890895350803977, Gradient Norm =
0.028485112790140796
Iteration [31400]: Cost = 2.081014724950768, Gradient Norm =
0.028347162579057106
Iteration [31600]: Cost = 2.073017695672208, Gradient Norm = 0.02821073080453464
Iteration [31800]: Cost = 2.0650972234937077, Gradient Norm =
0.028075791428867576
Iteration [32000]: Cost = 2.0572521113367244, Gradient Norm =
0.027942319017484896
Iteration [32200]: Cost = 2.0494811880546098, Gradient Norm =
0.027810288721406753
Iteration [32400]: Cost = 2.041783307716961, Gradient Norm = 0.02767967626031514
Iteration [32600]: Cost = 2.034157348917916, Gradient Norm =
0.027550457906211617
Iteration [32800]: Cost = 2.026602214107368, Gradient Norm =
0.027422610467638806
Iteration [33000]: Cost = 2.019116828944414, Gradient Norm =
0.027296111274444673
Iteration [33200]: Cost = 2.01170014167192, Gradient Norm = 0.027170938163065234
Iteration [33400]: Cost = 2.0043511225116064, Gradient Norm =
0.027047069462307408
Iteration [33600]: Cost = 1.9970687630787112, Gradient Norm =
0.02692448397961036
Iteration [33800]: Cost = 1.9898520758155638, Gradient Norm =
0.02680316098776798
Iteration [34000]: Cost = 1.9827000934433792, Gradient Norm =
0.026683080212093695
Iteration [34200]: Cost = 1.9756118684315038, Gradient Norm =
0.026564221818011012
Iteration [34400]: Cost = 1.9685864724835054, Gradient Norm =
0.026446566399051703
Iteration [34600]: Cost = 1.961622996039482, Gradient Norm = 0.02633009496524883
Iteration [34800]: Cost = 1.9547205477940206, Gradient Norm =
0.026214788931906926
Iteration [35000]: Cost = 1.9478782542291322, Gradient Norm =
0.02610063010873609
Iteration [35200]: Cost = 1.941095259161721, Gradient Norm =
0.025987600689336587
Iteration [35400]: Cost = 1.934370723304991, Gradient Norm =
0.025875683241019503
```

```
Iteration [35600]: Cost = 1.9277038238433306, Gradient Norm =
0.02576486069495271
Iteration [35800]: Cost = 1.9210937540201527, Gradient Norm =
0.02565511633661852
Iteration [36000]: Cost = 1.914539722738232, Gradient Norm = 0.02554643379657193
Iteration [36200]: Cost = 1.908040954172154, Gradient Norm =
0.025438797041489176
Iteration [36400]: Cost = 1.9015966873923043, Gradient Norm =
0.025332190365494514
Iteration [36600]: Cost = 1.8952061760002141, Gradient Norm =
0.025226598381756796
Iteration [36800]: Cost = 1.8888686877746217, Gradient Norm =
0.025122006014344896
Iteration [37000]: Cost = 1.882583504328051, Gradient Norm =
0.025018398490332747
Iteration [37200]: Cost = 1.8763499207734338, Gradient Norm =
0.024915761332145656
Iteration [37400]: Cost = 1.8701672454005598, Gradient Norm =
0.024814080350138833
Iteration [37600]: Cost = 1.8640347993618176, Gradient Norm =
0.024713341635400194
Iteration [37800]: Cost = 1.8579519163671576, Gradient Norm =
0.024613531552769706
Iteration [38000]: Cost = 1.8519179423877339, Gradient Norm =
0.024514636734066574
Iteration [38200]: Cost = 1.8459322353680359, Gradient Norm = 0.0244166440715181
Iteration [38400]: Cost = 1.8399941649463025, Gradient Norm =
0.02431954071138381
Iteration [38600]: Cost = 1.834103112182747, Gradient Norm =
0.024223314047765804
Iteration [38800]: Cost = 1.8282584692955597, Gradient Norm =
0.024127951716602015
Iteration [39000]: Cost = 1.8224596394042119, Gradient Norm =
0.02403344158983305
Iteration [39200]: Cost = 1.8167060362799885, Gradient Norm =
0.023939771769738768
Iteration [39400]: Cost = 1.8109970841034255, Gradient Norm =
0.023846930583438525
Iteration [39600]: Cost = 1.8053322172284507, Gradient Norm =
0.023754906577548647
Iteration [39800]: Cost = 1.7997108799530208, Gradient Norm =
0.023663688512993216
Iteration [40000]: Cost = 1.7941325262960408, Gradient Norm =
0.02357326535996183
Iteration [40200]: Cost = 1.7885966197803225, Gradient Norm =
0.023483626293010193
Iteration [40400]: Cost = 1.7831026332214526, Gradient Norm =
0.023394760686298567
```

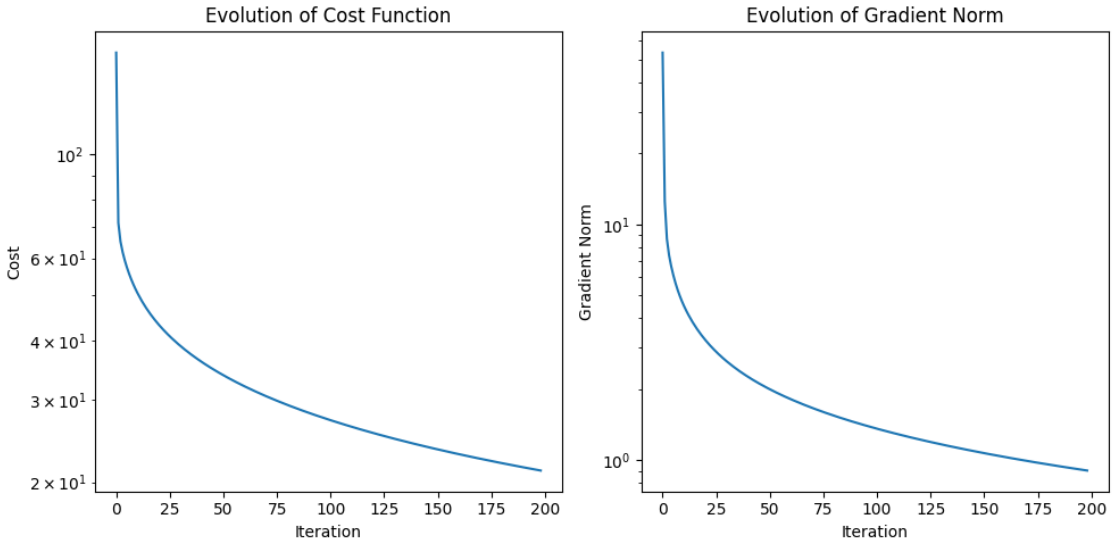
```
Iteration [40600]: Cost = 1.7776500485223636, Gradient Norm =
0.02330665810896395
Iteration [40800]: Cost = 1.772238356473367, Gradient Norm = 0.0232193083206204
Iteration [41000]: Cost = 1.7668670565576026, Gradient Norm =
0.02313270126698538
Iteration [41200]: Cost = 1.761535656761599, Gradient Norm =
0.023046827075626056
Iteration [41400]: Cost = 1.756243673390848, Gradient Norm = 0.02296167605182259
Iteration [41600]: Cost = 1.7509906308902727, Gradient Norm =
0.022877238674544983
Iteration [41800]: Cost = 1.7457760616693645, Gradient Norm =
0.022793505592538937
Iteration [42000]: Cost = 1.7405995059318844, Gradient Norm =
0.02271046762051808
Iteration [42200]: Cost = 1.7354605115100297, Gradient Norm =
0.02262811573545923
Iteration [42400]: Cost = 1.7303586337027983, Gradient Norm =
0.02254644107299612
Iteration [42600]: Cost = 1.7252934351185936, Gradient Norm =
0.022465434923910697
Iteration [42800]: Cost = 1.7202644855218192, Gradient Norm =
0.022385088730716724
Iteration [43000]: Cost = 1.7152713616833852, Gradient Norm =
0.02230539408433466
Iteration [43200]: Cost = 1.710313647235002, Gradient Norm =
0.022226342720853585
Iteration [43400]: Cost = 1.7053909325271899, Gradient Norm =
0.022147926518378682
Iteration [43600]: Cost = 1.7005028144907952, Gradient Norm =
0.02207013749396019
Iteration [43800]: Cost = 1.6956488965020202, Gradient Norm =
0.021992967800603187
Iteration [44000]: Cost = 1.690828788250814, Gradient Norm =
0.021916409724354432
Iteration [44200]: Cost = 1.6860421056124835, Gradient Norm =
0.02184045568146374
Iteration [44400]: Cost = 1.6812884705225484, Gradient Norm = 0.0217650982156193
Iteration [44600]: Cost = 1.6765675108545839, Gradient Norm =
0.021690329995252586
Iteration [44800]: Cost = 1.671878860301123, Gradient Norm =
0.021616143810912133
Iteration [45000]: Cost = 1.667222158257436, Gradient Norm =
0.021542532572704547
Iteration [45200]: Cost = 1.6625970497081342, Gradient Norm =
0.021469489307798868
Iteration [45400]: Cost = 1.658003185116472, Gradient Norm = 0.02139700715799432
Iteration [45600]: Cost = 1.653440220316405, Gradient Norm = 0.02132507937734871
Iteration [45800]: Cost = 1.6489078164071214, Gradient Norm =
```

```
0.021253699329865858
Iteration [46000]: Cost = 1.6444056396501612, Gradient Norm =
0.02118286048724002
Iteration [46200]: Cost = 1.639933361368978, Gradient Norm =
0.021112556426656992
Iteration [46400]: Cost = 1.635490657850829, Gradient Norm =
0.021042780828647783
Iteration [46600]: Cost = 1.631077210251002, Gradient Norm =
0.020973527474995892
Iteration [46800]: Cost = 1.6266927044992974, Gradient Norm =
0.020904790246695282
Iteration [47000]: Cost = 1.622336831208678, Gradient Norm = 0.02083656312195785
Iteration [47200]: Cost = 1.6180092855860373, Gradient Norm =
0.02076884017426888
Iteration [47400]: Cost = 1.6137097673450658, Gradient Norm =
0.02070161557048991
Iteration [47600]: Cost = 1.6094379806210677, Gradient Norm =
0.020634883569006238
Iteration [47800]: Cost = 1.6051936338877986, Gradient Norm =
0.020568638517919215
Iteration [48000]: Cost = 1.600976439876147, Gradient Norm =
0.020502874853281255
Iteration [48200]: Cost = 1.5967861154947207, Gradient Norm =
0.02043758709737301
Iteration [48400]: Cost = 1.5926223817521807, Gradient Norm =
0.020372769857020447
Iteration [48600]: Cost = 1.5884849636813787, Gradient Norm =
0.020308417821952137
Iteration [48800]: Cost = 1.5843735902651548, Gradient Norm =
0.02024452576319466
Iteration [49000]: Cost = 1.5802879943638422, Gradient Norm =
0.020181088531505426
Iteration [49200]: Cost = 1.5762279126443397, Gradient Norm =
0.020118101055841487
Iteration [49400]: Cost = 1.5721930855108128, Gradient Norm =
0.02005555834186469
Gradient norm is below threshold (0.02). Stopping optimization.
```

Cost & Gradient Plots

[ ]:
```
stop_plot = 200
#plot evolution of cost function
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.semilogy(np.arange(stop_plot-1), costs[1:stop_plot] - min(costs))
#semilogy(np.arange(max_iters-1), np.abs(cost[:-1] - min(cost))
plt.title('Evolution of Cost Function')
plt.xlabel('Iteration')
```

```
plt.ylabel('Cost')

#plot evolution of gradient norm
plt.subplot(1, 2, 2)
plt.semilogy(grad_norms[1:stop_plot])
plt.title('Evolution of Gradient Norm')
plt.xlabel('Iteration')
plt.ylabel('Gradient Norm')
plt.tight_layout()
plt.show()
```



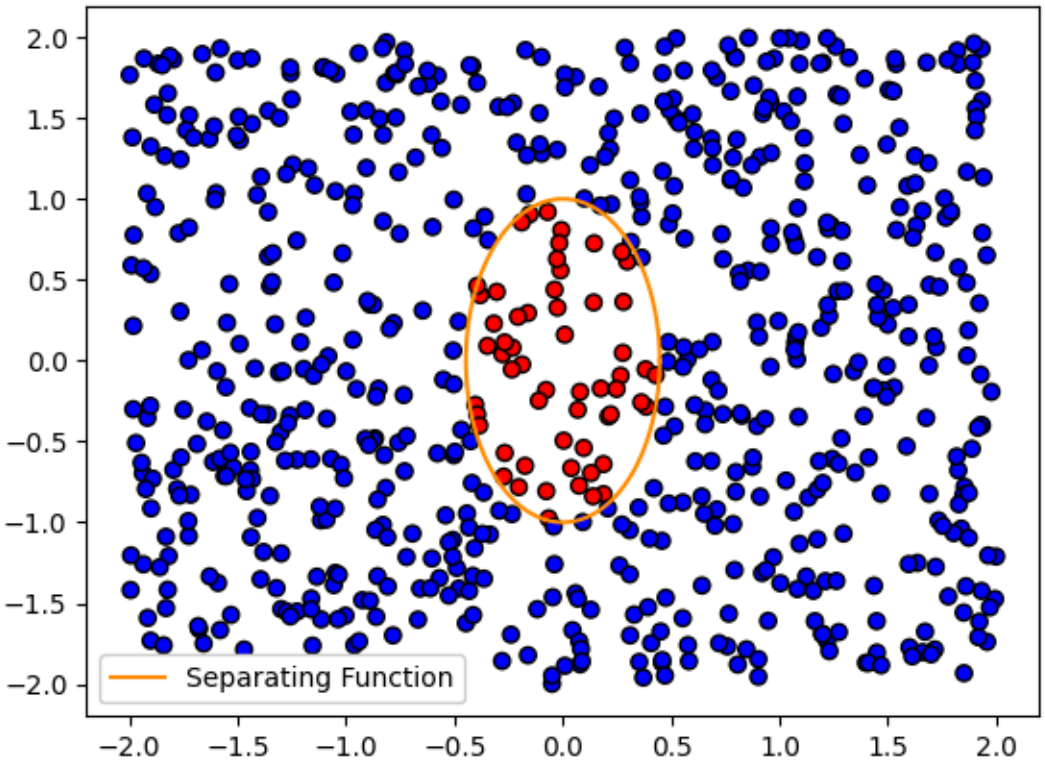
1.2.4 Classification Test

[ ]:
```
def plot_classification_and_true_function(points, labels, w, b,␣
↪conic_parameters):

    # Classify each point in the grid
    Z = np.array([w.dot(phi(x)) + b for x in points])
    Z = 1 / (1 + np.exp(-Z))  # Apply sigmoid function
    Z = (Z > 0.5).astype(int)  # Threshold at 0.5

    # Define the colormap
    cmap = matplotlib.colors.ListedColormap(['red', 'blue'])

    # Plot the points with different colors depending on their classification
    plt.scatter(points[:, 0], points[:, 1], c=Z, cmap=cmap, edgecolors='k')
```

```
    if conic_parameters['center'] is not None:

        # Plot the true ellipse
        h, k = conic_parameters['center']
        a = (conic_parameters['x_ints'][1] - conic_parameters['x_ints'][0]) / 2
        b = (conic_parameters['y_ints'][1] - conic_parameters['y_ints'][0]) / 2
        theta = np.linspace(0, 2*np.pi, 100)
        x = h + a * np.cos(theta)
        y = k + b * np.sin(theta)
        plt.plot(x, y, color="darkorange", label='Separating Function')


    else:
        #plot the true parabola

        # Generate a range of y values around the vertex
        y_vals = np.linspace(-spread, spread, 200)

        # Calculate the corresponding x values
        x_vals = [-(conic_parameters['w'][0]*y**2 + conic_parameters['w'][1]*y␣
↪+ conic_parameters['b']) / conic_parameters['w'][1] for y in y_vals]

        # Plot the parabola
        plt.plot(x_vals,y_vals,color="darkorange", linestyle='--', linewidth=3,␣
↪label='Separating Function')
        plt.xlim(-spread, spread)
        plt.ylim(-spread, spread)

    plt.legend(framealpha=1)
    plt.show()
```

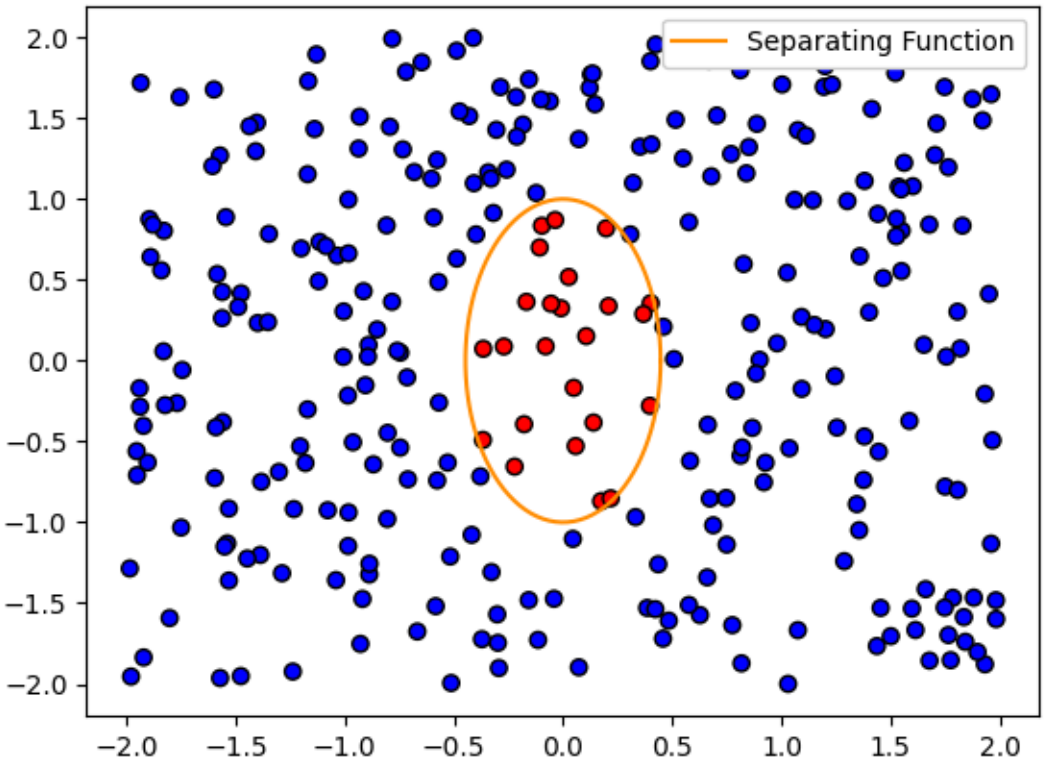
[ ]:
```
points = np.array([data[0] for data in D_train]) #P
labels = np.array([data[1] for data in D_train]) #D
```

[ ]:
```
plot_classification_and_true_function(points, labels, w_opt, b_opt,␣
↪conic_parameters) #plot of the test set
```



[ ]:
```
points = np.array([data[0] for data in D_test]) #P
labels = np.array([data[1] for data in D_test]) #D
```

[ ]:
```
plot_classification_and_true_function(points, labels, w_opt, b_opt,␣
↪conic_parameters) #plot of the test set
```



[ ]:
```
#show the optimal decision variables
print("\nOPTIMAL VALUES:\n")
print(f"b_opt: {b_opt}, b: {conic_parameters['b']}")
print(f"w0_opt: {w_opt[0]}, w: {conic_parameters['w'][0]}")
print(f"w1_opt: {w_opt[1]}, w: {conic_parameters['w'][1]}")
print(f"w2_opt: {w_opt[2]}, w: {conic_parameters['w'][2]}")
print(f"w3_opt: {w_opt[3]}, w: {conic_parameters['w'][3]}")
```

```
OPTIMAL VALUES:

b_opt: -32.547628104780394, b: -1
w0_opt: 4.052794284799239, w: 0
w1_opt: 0.2670796145228478, w: 0
w2_opt: 155.31248064057417, w: 5
w3_opt: 33.24778973211476, w: 1
```

1.2.5 Evaluation Metrics

[ ]:
```
def evaluation_metrics(y_train, y_train_pred, y_test, y_test_pred):

    # Compute evaluation metrics
```

```
    metrics_dict = {
        'Classification Error': [1 - accuracy_score(y_train, y_train_pred), 1 -␣
↪accuracy_score(y_test, y_test_pred)],
        'Accuracy': [accuracy_score(y_train, y_train_pred),␣
↪accuracy_score(y_test, y_test_pred)],
        'Precision': [precision_score(y_train, y_train_pred),␣
↪precision_score(y_test, y_test_pred)],
        'Recall': [recall_score(y_train, y_train_pred), recall_score(y_test,␣
↪y_test_pred)],
        'F1 Score': [f1_score(y_train, y_train_pred), f1_score(y_test,␣
↪y_test_pred)]
    }
    return metrics_dict
```

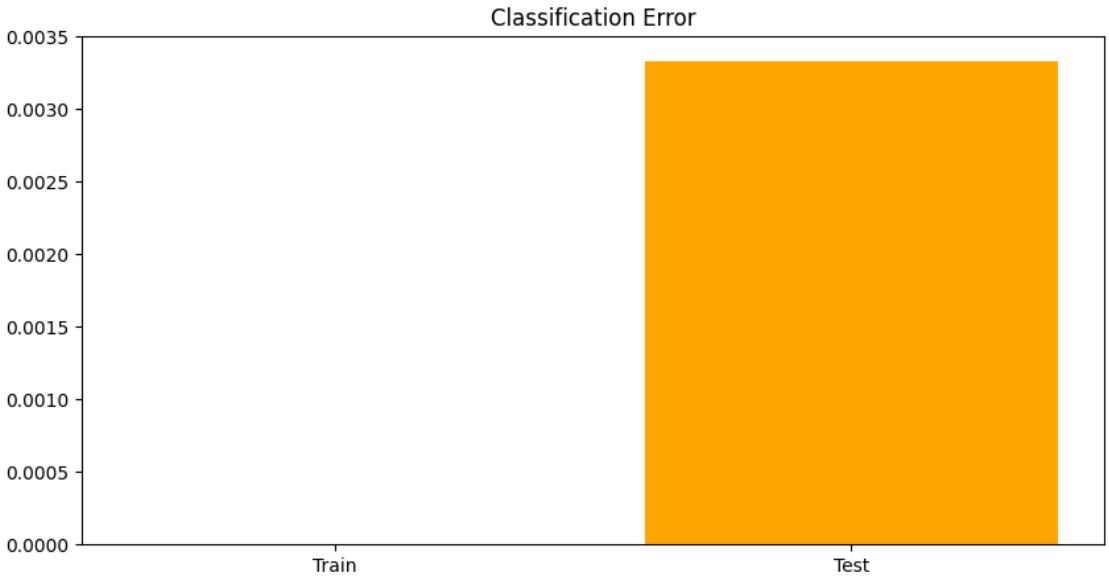
[ ]:
```
y_train_pred = np.array([P(data[0], w_opt, b_opt) for data in D_train])
y_test_pred = np.array([P(data[0], w_opt, b_opt) for data in D_test])

y_train = np.array([data[1] for data in D_train])
y_test = np.array([data[1] for data in D_test])

# Compute evaluation metrics
metrics_dict = evaluation_metrics(y_train, y_train_pred, y_test, y_test_pred)
```

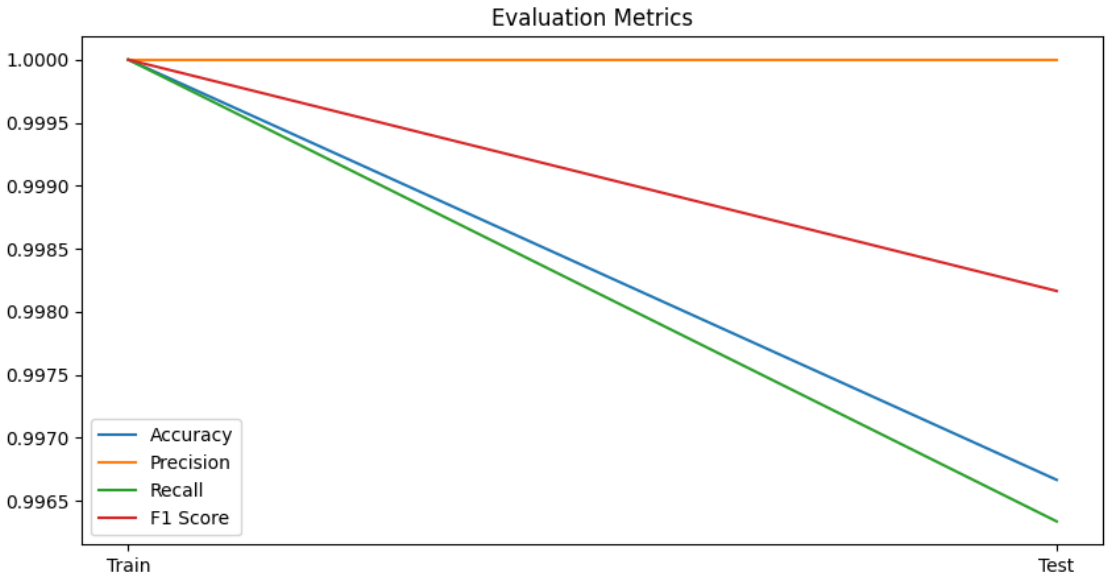
[ ]:
```
def print_evaluation_metrics(metrics_dict):

    print("\n\n-----------------------------------------------------------------")
    print("TRAIN-SET METRICS")
    print(f"Classification Error: {metrics_dict['Classification Error'][0]}")
    print(f"Accuracy: {metrics_dict['Accuracy'][0]}")
    print(f"Precision: {metrics_dict['Precision'][0]}")
    print(f"Recall: {metrics_dict['Recall'][0]}")
    print(f"F1 Score: {metrics_dict['F1 Score'][0]}")
    print("-----------------------------------------------------------------")
    print("\n\n-----------------------------------------------------------------")
    print("TEST-SET METRICS")
    print(f"Classification Error: {metrics_dict['Classification Error'][1]}")
    print(f"Accuracy: {metrics_dict['Accuracy'][1]}")
    print(f"Precision: {metrics_dict['Precision'][1]}")
    print(f"Recall: {metrics_dict['Recall'][1]}")
    print(f"F1 Score: {metrics_dict['F1 Score'][1]}")
    print("-----------------------------------------------------------------")

    plt.figure(figsize=(10, 5))
    plt.bar(['Train', 'Test'], metrics_dict['Classification Error'],␣
↪color=['blue', 'orange'])
    plt.title('Classification Error')
```

```
    plt.show()

    # Plot other metrics
    plt.figure(figsize=(10, 5))
    for metric, values in metrics_dict.items():
        if metric != 'Classification Error':
            plt.plot(['Train', 'Test'], values, label=metric)
    plt.title('Evaluation Metrics')
    plt.legend()
    plt.show()
```

[ ]:
```
print_evaluation_metrics(metrics_dict)
```

```
------------------------------------------------------------------
TRAIN-SET METRICS
Classification Error: 0.0
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0
------------------------------------------------------------------


------------------------------------------------------------------
TEST-SET METRICS
Classification Error: 0.0033333333333332993
Accuracy: 0.9966666666666667
Precision: 1.0
Recall: 0.9963369963369964
F1 Score: 0.998165137614679
------------------------------------------------------------------
```





1.3 Task 1.3 - Distributed Classification

1. Split (randomly) the dataset in N subsets, one for each agent $i \in \{1, \dots, N\}$.
2. Implement the Gradient Tracking algorithm to classify the dataset in a distributed fashion (you can extend the code designed in Task 1.1).
3. Generate a set of simulations testing different dataset sizes and patterns, showing the convergence of the distributed algorithm to a stationary point of the optimization problem. Moreover, plot the evolution of the cost function and of the norm of the gradient of the cost

function across the iterations.
4. Evaluate the quality of the obtained solution by computing the percentage of misclassified points.

1.3.1 Distributed Dataset Generation

[ ]:
```python
def distribute_data(data, num_agents, pattern):

    if pattern == 'sequential':
        # Sort the data by the first feature
        data.sort(key=lambda x: x[0][0])

        # Calculate the size of each slice
        slice_size = len(data) // num_agents

        # Split the data into num_agents slices
        return [data[i*slice_size:(i+1)*slice_size] for i in range(num_agents)]
    elif pattern == 'random':
        rand.shuffle(data)
        return [data[i::num_agents] for i in range(num_agents)]
    else:
        raise ValueError(f'Unknown distribution pattern: {pattern}')
```

[ ]:
```python
#Add noise to D_train
def add_noise(data, noise_level):

    noisy_data = []
    for point, label in data:
        if rand.random() < noise_level:
            noisy_data.append((point, -label))
        else:
            noisy_data.append((point, label))
    return noisy_data
```

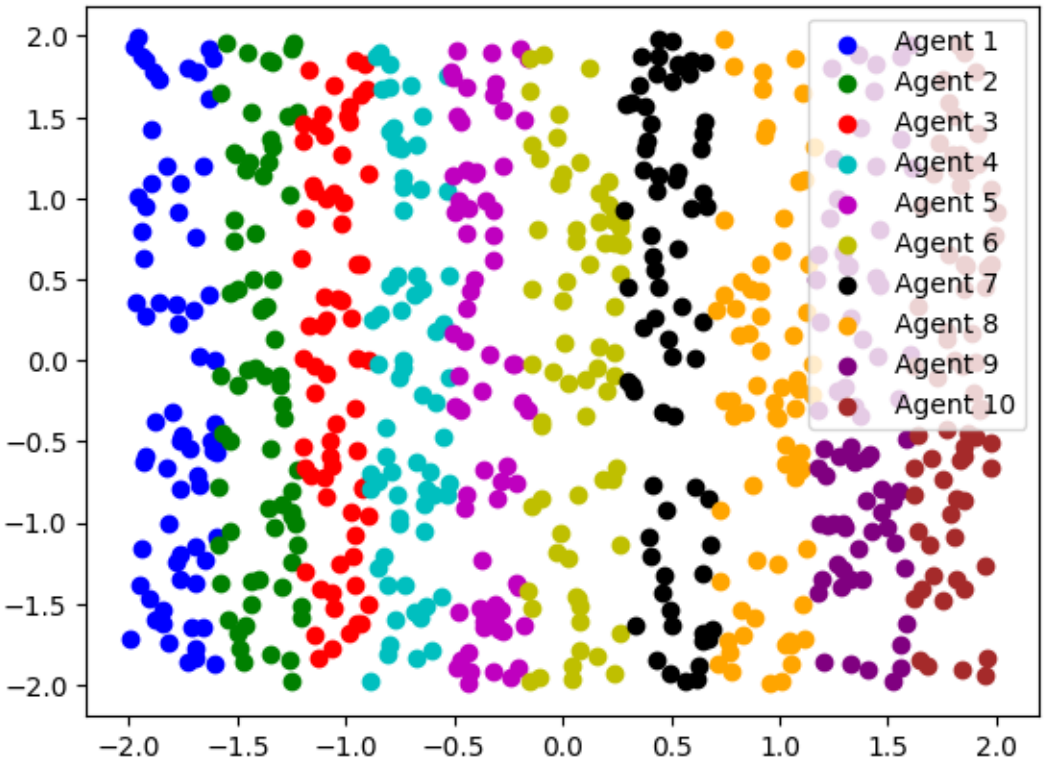
[ ]:
```python
noise = False
split = 'sequential'
```

[ ]:
```python
if noise:
    D_train = add_noise(D_train, 0.2)
D_train_distributed = distribute_data(D_train, N, split)
```

[ ]:
```python
def plot_distributed_data(agent_datasets):

    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k','orange', 'purple', 'brown' ] ␣
 ↪# Add more colors if you have more than 10 agents

    for i, data in enumerate(agent_datasets):
```

```
        # Extract the features and labels from the data
        features = [item[0] for item in data]
        labels = [item[1] for item in data]

        # Separate the features into x and y values
        x_values = [feature[0] for feature in features]
        y_values = [feature[1] for feature in features]

        # Plot the data, coloring it based on the label
        plt.scatter(x_values, y_values, color=colors[i % len(colors)],␣
↪label=f'Agent {i+1}')

    plt.legend()
    plt.show()
```

[ ]:
```
plot_distributed_data(D_train_distributed)
```



1.3.2 Gradient Tracking Algorithm

[ ]:
```
def distributed_gradient_tracking(iterations, N, A, Adj, alpha, agent_datasets):

    Z = np.zeros((iterations, N, 5))
    S = np.zeros((iterations, N, 5))
    W_norm = np.zeros((iterations,4))
    B_norm = np.zeros((iterations,1))
    cost = np.zeros((iterations))

    D = {}  # dictionary to store agents' attributes
    p = {}  # dictionary to store agents' labels

    for i in range(N):
        D[i] = np.array([data[0] for data in agent_datasets[i]])  # Attributes
        p[i] = np.array([data[1] for data in agent_datasets[i]])  # Labels
        Z[0, i, :4] = np.random.uniform(-1,1, size=4)
        Z[0, i, 4] = np.random.uniform(-1,1)
        _, S[0, i, :4], S[0, i, 4] = logistic_regression_loss(Z[0, i, :4], Z[0,␣
↪i, 4], D[i], p[i])

    for k in range(iterations-1):

        for i in range(N):

            neighbors = np.nonzero(Adj[i])[0]
            # Update the values of Z and S for each agent
            Z[k+1,i,:4] += Z[k,i,:4] * A[i,i]
            Z[k+1,i,4] += Z[k,i,4] * A[i,i]
            S[k+1,i,:4] += S[k,i,:4] * A[i,i]
            S[k+1,i,4] += S[k,i,4] * A[i,i]

            for j in neighbors:
                # Update the values of Z and S for each agent
                Z[k+1,i,:4] += Z[k,j,:4] * A[i,j]
                Z[k+1,i,4] += Z[k,j,4] * A[i,j]
                S[k+1,i,:4] += S[k,j,:4] * A[i,j]
                S[k+1,i,4] += S[k,j,4] * A[i,j]

            Z[k+1, i, :4] -= alpha * S[k, i, :4]
            Z[k+1, i, 4] -= alpha * S[k, i, 4]

            # update the gradients using the logistic regression loss function
            _, grad_new_w, grad_new_b = logistic_regression_loss(Z[k+1,i,:4],␣
↪Z[k+1,i,4], D[i], p[i])
            _, grad_old_w, grad_old_b = logistic_regression_loss(Z[k,i,:4],␣
↪Z[k,i,4], D[i], p[i])
```

```
            S[k+1,i,:4] +=  grad_new_w - grad_old_w
            S[k+1,i,4] +=  grad_new_b - grad_old_b

            #W_norm[k+1] += np.linalg.norm(S[k+1,i,:4]) + np.linalg.
 ↪norm(S[k+1,i,4])
            #B_norm[k+1] += np.linalg.norm(Z[k+1,i,:4]) + np.linalg.
 ↪norm(Z[k+1,i,4])

            cost[k] += logistic_regression_loss(Z[k,i,:4], Z[k,i,4], D[i],␣
 ↪p[i])[0]

    return Z,S, cost
```

1.3.3 Parameters

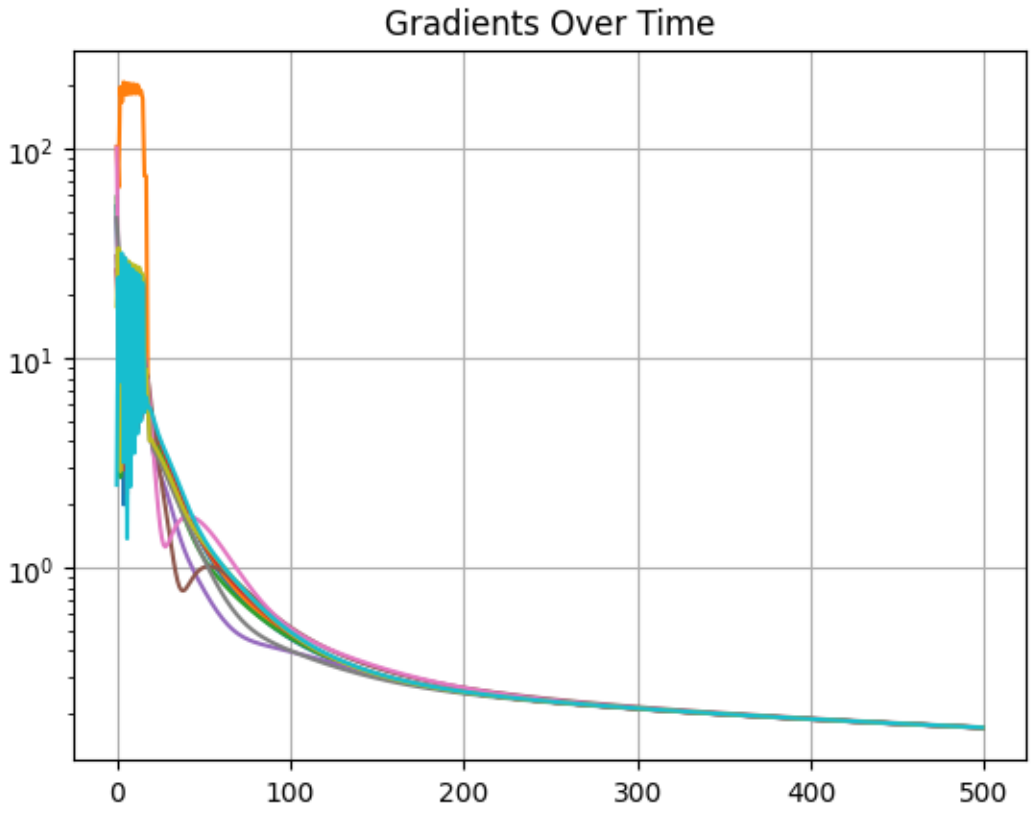
[ ]:
```
Adj = generate_graph(N,seed)
A = metropolis_hastings(N, Adj)
alpha = 0.05
iterations = 2000
```

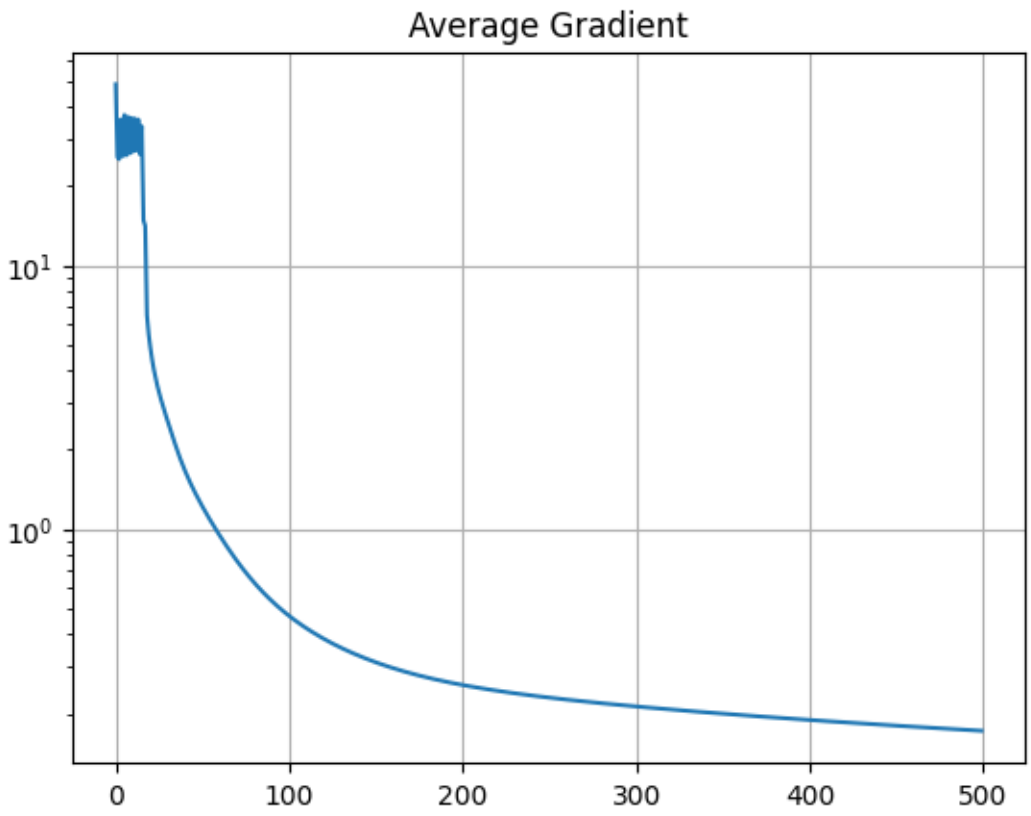
1.3.4 Simulations

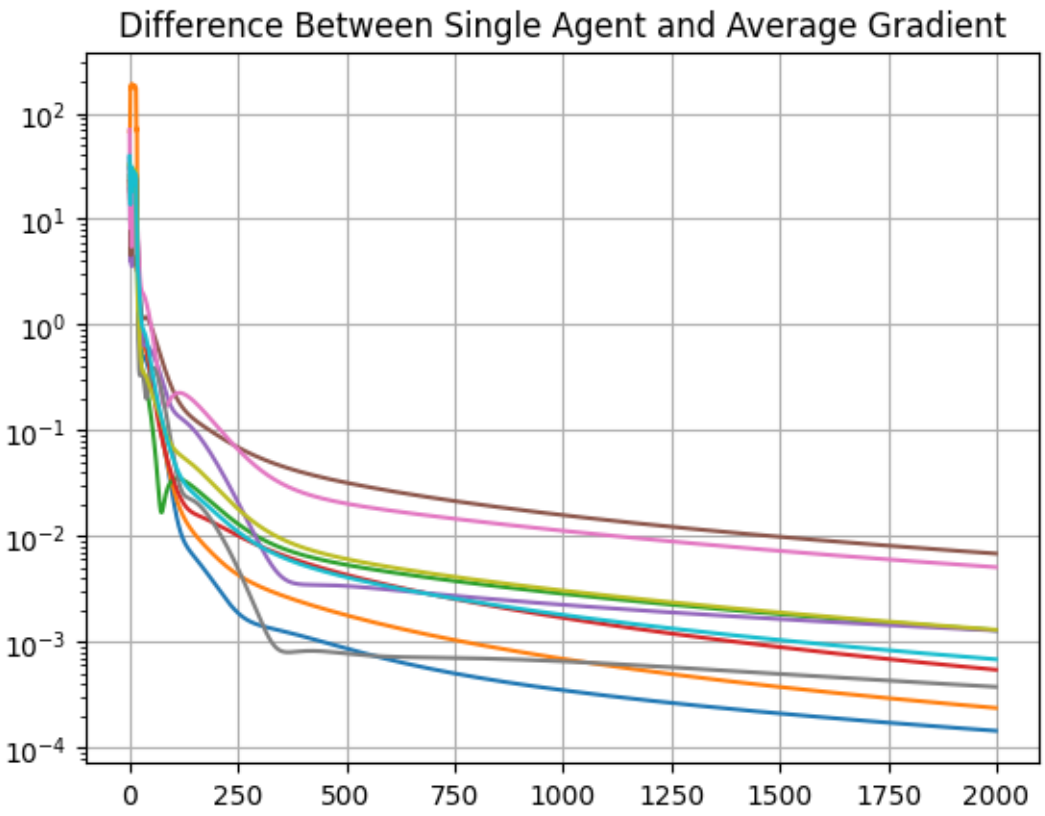
[ ]:
```
Z,S,cost = distributed_gradient_tracking(iterations, N, A, Adj, alpha,␣
 ↪D_train_distributed)
```

[ ]:
```
#convert S to it's norm
stop_plot = 500 #number of iterations to plot
S_norm = np.linalg.norm(S[:stop_plot], axis=2)
#plot S
fig, ax = plt.subplots()
for i in range(N):
    ax.semilogy(np.arange(stop_plot), S_norm[:,i])
ax.title.set_text('Gradients Over Time')
ax.grid()

#plot the average gradient
fig, ax = plt.subplots()
average_gradient = np.mean(S_norm, axis=1)
ax.semilogy(np.arange(stop_plot), average_gradient)
ax.title.set_text('Average Gradient')
ax.grid()

#plot difference between single agent and average gradient
stop_plot = 2000
fig, ax = plt.subplots()
for i in range(N):
```

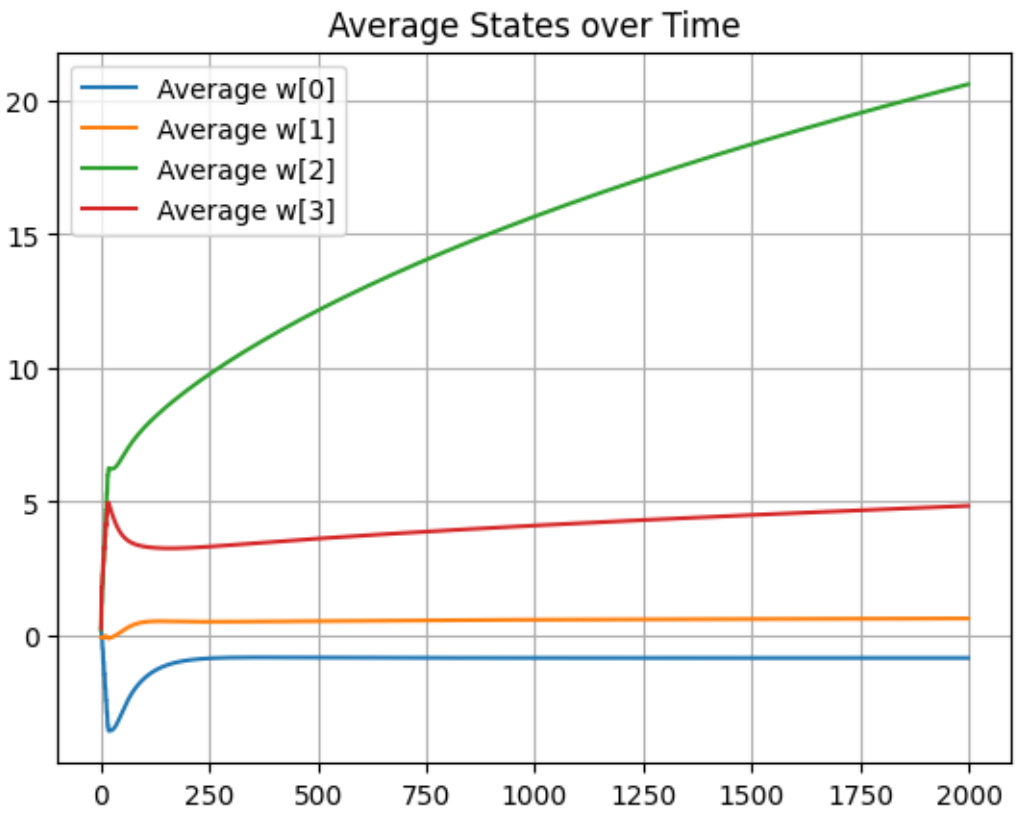
```
    difference = S[:stop_plot,i,:] - np.mean(S[:stop_plot,:,:], axis=1)
    ax.semilogy(np.arange(stop_plot), np.linalg.norm(difference, axis=1))
ax.title.set_text('Difference Between Single Agent and Average Gradient')
ax.grid()
```



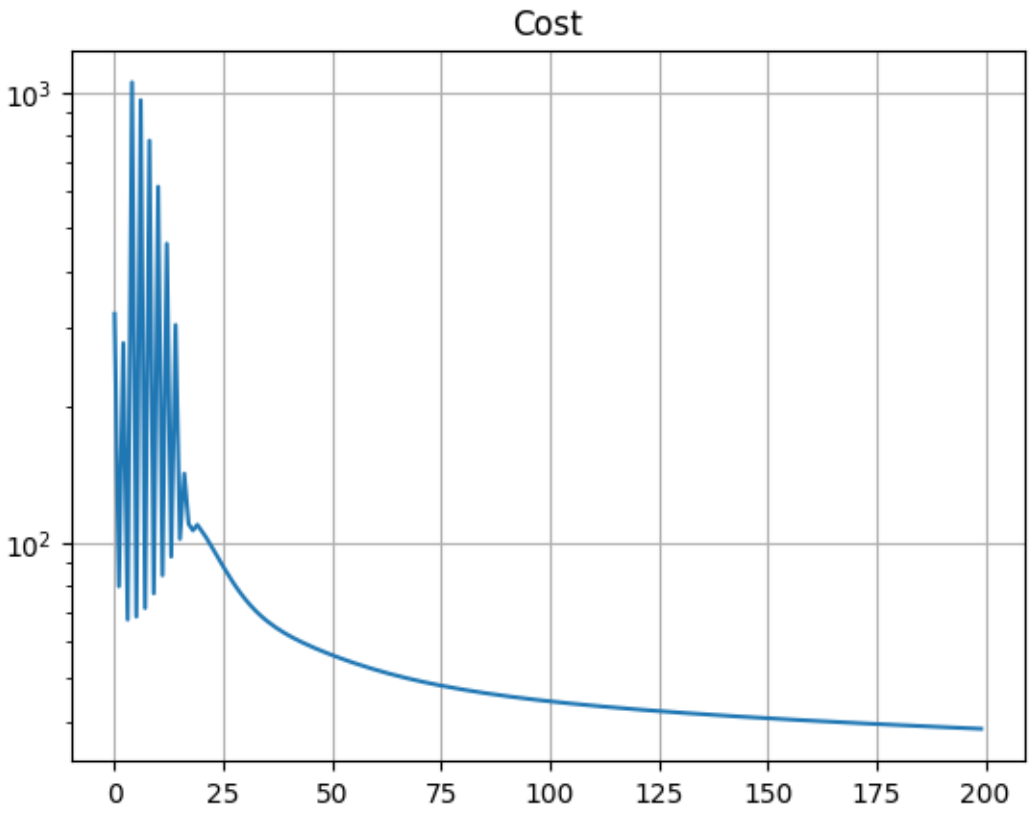

Average Gradient
10^1
10^0
0
100
200
300
400
500



[ ]:
```
#plot average Z
plt.figure()
for k in range(4):
    avg_Z = np.mean(Z[:,:,k], axis=1)  # calculate average over agents
    plt.plot(np.arange(iterations), avg_Z, label=f'Average w[{k}]')
plt.title('Average States over Time')
plt.grid()
plt.legend()
plt.show()
```



[ ]:
```
#plot the cost
stop_plot = 200
fig, ax = plt.subplots()
ax.title.set_text('Cost')
ax.semilogy(np.arange(stop_plot), (cost[:stop_plot]))
ax.grid()
```



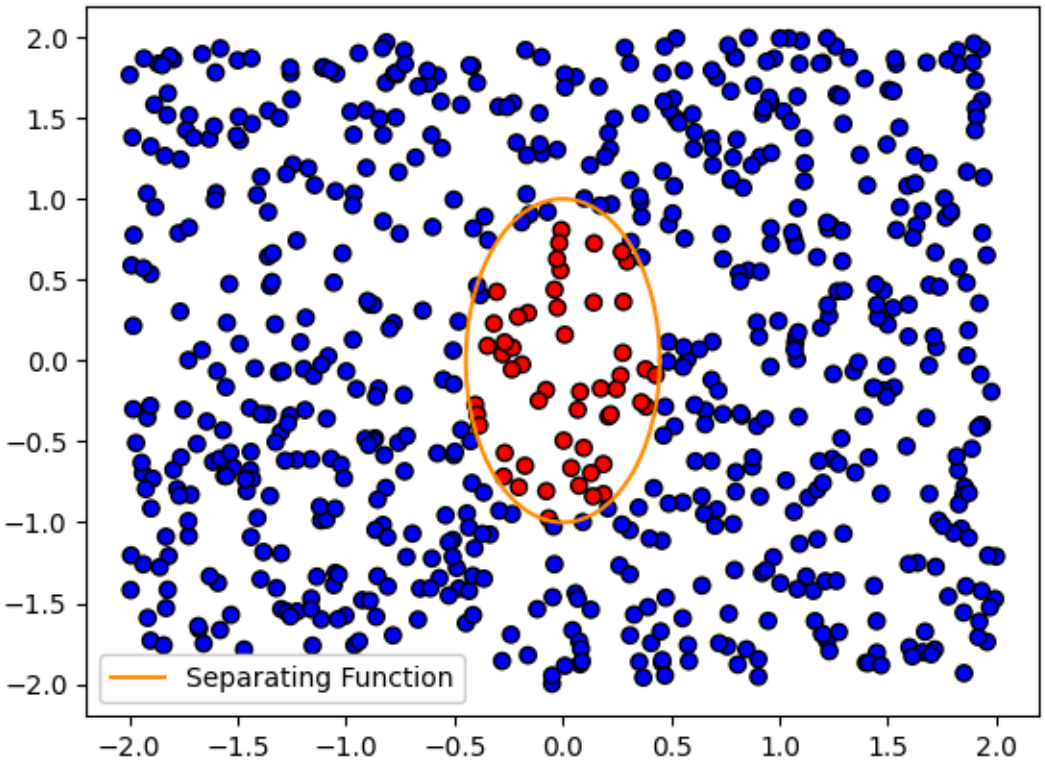
1.3.5 Classification Test

[ ]:
```
w_opt_distributed = np.mean(Z[-1,:,:4], axis=0)
b_opt_distributed = np.mean(Z[-1,:,4], axis=0)
```

```
[-0.04122734  0.03043118  1.          0.23483036]
1.0
[0, 0, 5, 1]
```

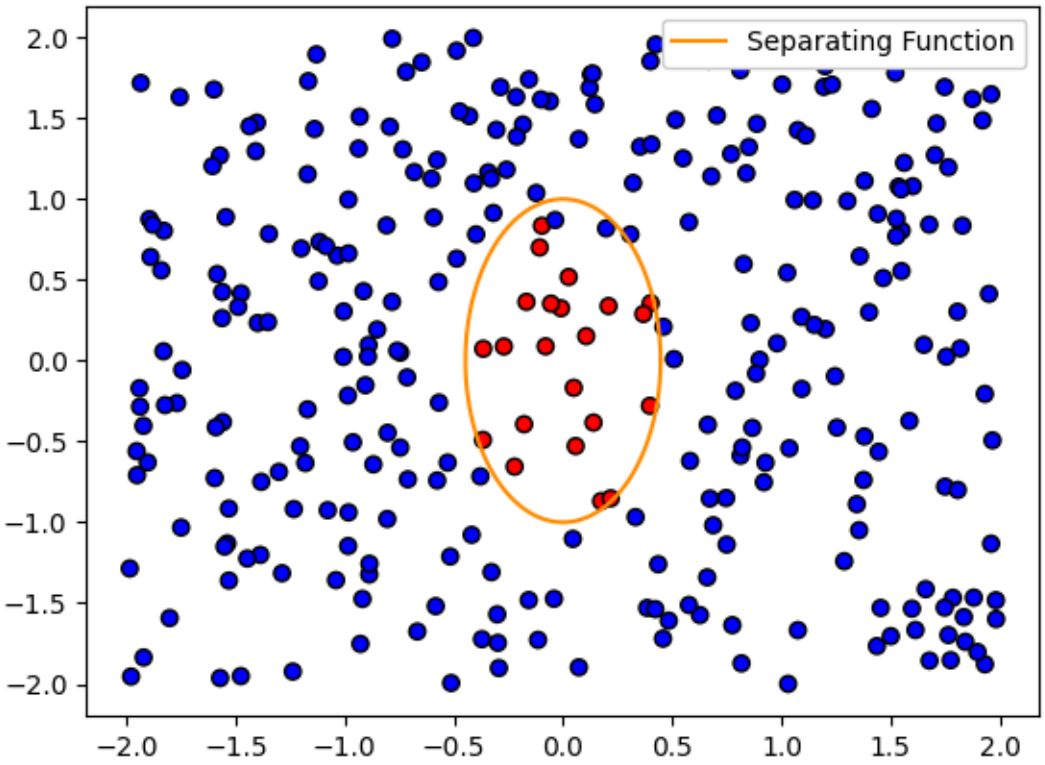
[ ]:
```
points = np.array([data[0] for data in D_train]) #P
labels = np.array([data[1] for data in D_train]) #D
```

[ ]:
```
plot_classification_and_true_function(points, labels, w_opt_distributed,␣
↪b_opt_distributed, conic_parameters)
```



[ ]:
```
points = np.array([data[0] for data in D_test]) #P
labels = np.array([data[1] for data in D_test]) #D
```

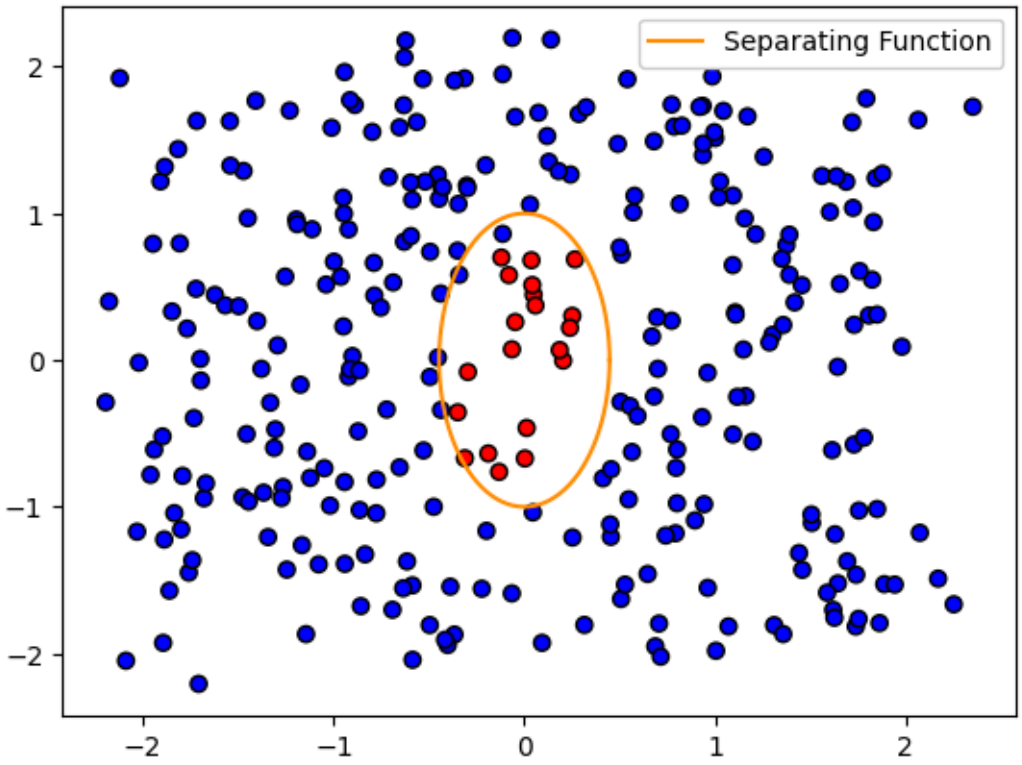
[ ]:
```
plot_classification_and_true_function(points, labels, w_opt_distributed,␣
↪b_opt_distributed, conic_parameters)
```



1.3.6 Classification Test with Noise

[ ]:
```
points = np.array([data[0] for data in D_test]) #P
labels = np.array([data[1] for data in D_test]) #D

noise = np.random.normal(scale=0.2, size=points.shape)
points_noisy = points + noise
```

[ ]:
```
plot_classification_and_true_function(points_noisy, labels, w_opt_distributed,␣
↪b_opt_distributed, conic_parameters)
```



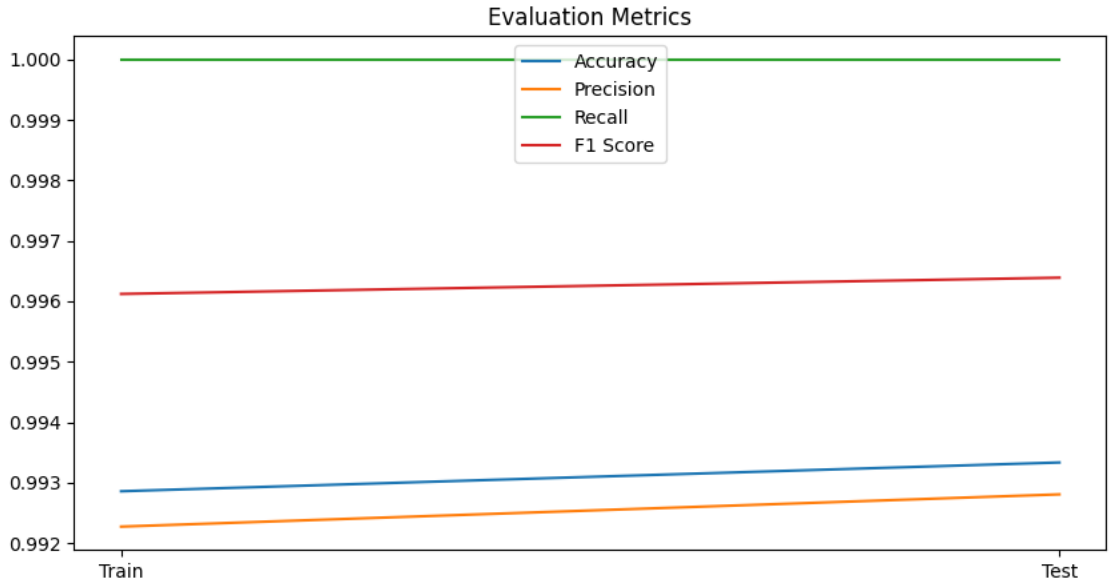
1.3.7 Evaluation Metrics

[ ]:
```
y_train_pred = np.array([P(data[0], w_opt_distributed, b_opt_distributed) for␣
 ↪data in D_train])
y_test_pred = np.array([P(data[0], w_opt_distributed, b_opt_distributed) for␣
 ↪data in D_test])

y_train = np.array([data[1] for data in D_train])
y_test = np.array([data[1] for data in D_test])

# Compute evaluation metrics
metrics_dict = evaluation_metrics(y_train, y_train_pred, y_test, y_test_pred)
```

[ ]:
```
print_evaluation_metrics(metrics_dict)
```

```
------------------------------------------------------------
TRAIN-SET METRICS
Classification Error: 0.0071428571428571175
Accuracy: 0.9928571428571429
Precision: 0.9922720247295209
```

```
Recall: 1.0
F1 Score: 0.996121024049651
------------------------------------------------------------


------------------------------------------------------------
TEST-SET METRICS
Classification Error: 0.00666666666666671
Accuracy: 0.9933333333333333
Precision: 0.9928057553956835
Recall: 1.0
F1 Score: 0.996389891696751
------------------------------------------------------------
```



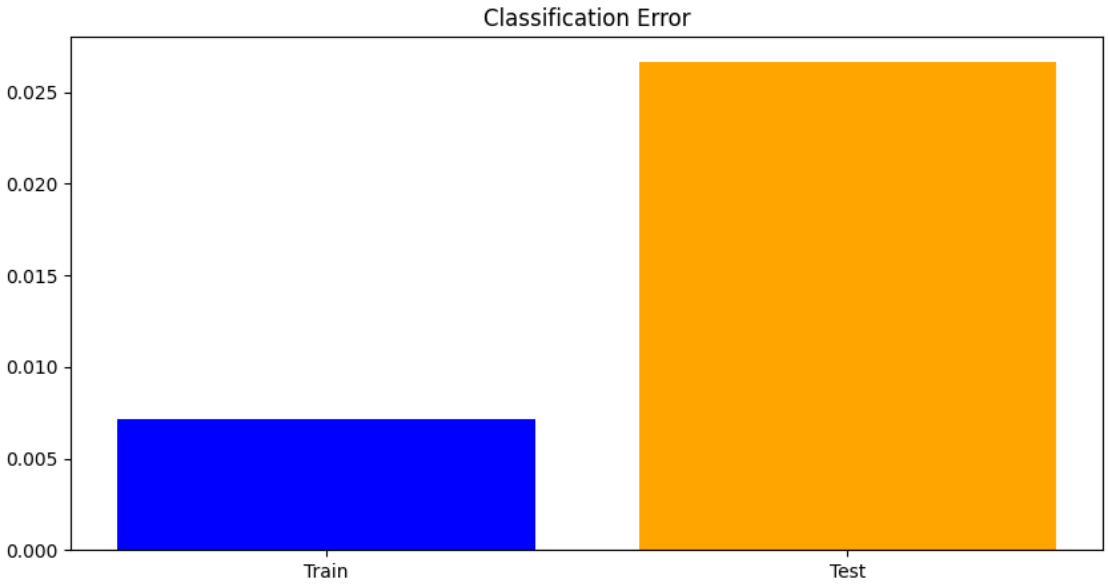
1.3.8 Evaluation with Noise

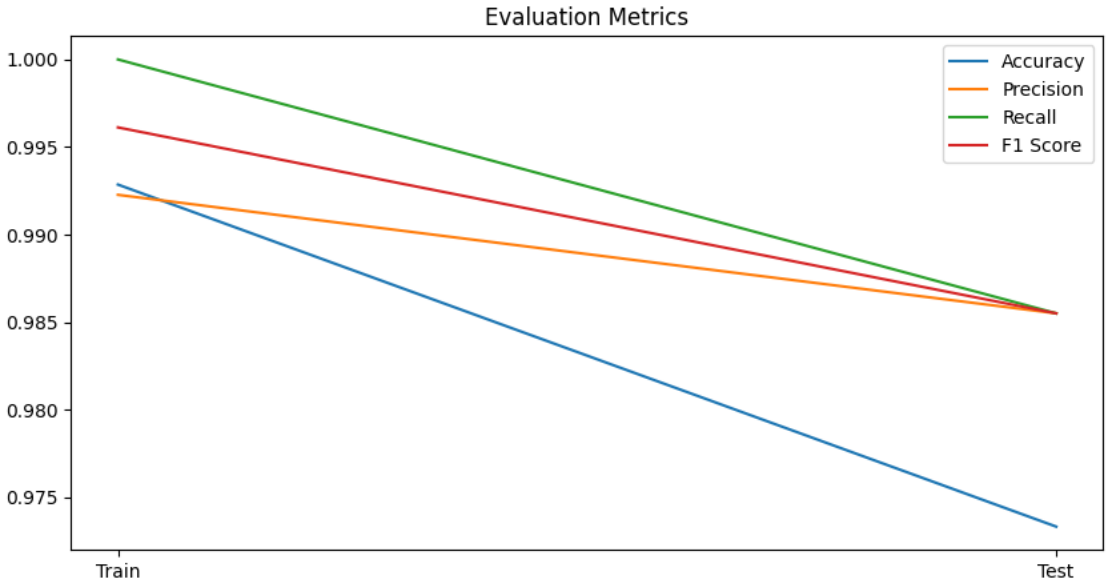
[ ]:
```
points_test = np.array([data[0] for data in D_test])
labels_test = np.array([data[1] for data in D_test])

# Add noise
noise_test = np.random.normal(scale=0.2, size=points_test.shape)
points_test_noisy = points_test + noise_test

# Make predictions
y_train_pred = np.array([P(data[0], w_opt_distributed, b_opt_distributed) for␣
 ↪data in D_train])
y_test_pred = np.array([P(point, w_opt_distributed, b_opt_distributed) for␣
 ↪point in points_test_noisy])

# Compute evaluation metrics
metrics_dict = evaluation_metrics(y_train, y_train_pred, labels_test,␣
 ↪y_test_pred)
```

[ ]:
```
print_evaluation_metrics(metrics_dict)
```

```
----------------------------------------------------------------
TRAIN-SET METRICS
Classification Error: 0.0071428571428571175
Accuracy: 0.9928571428571429
```

```
Precision: 0.9922720247295209
Recall: 1.0
F1 Score: 0.996121024049651
-----------------------------------------------------------------


-----------------------------------------------------------------
TEST-SET METRICS
Classification Error: 0.026666666666666616
Accuracy: 0.9733333333333334
Precision: 0.9855072463768116
Recall: 0.9855072463768116
F1 Score: 0.9855072463768116
-----------------------------------------------------------------
```




Evaluation Metrics
1.000
0.995
0.990
0.985
0.980
0.975
Train
Test
Accuracy
Precision
Recall
F1 Score