Assignment 3
Introdução a Aprendizagem automática
Unsupervised Learning

João Carlos Cambaia Gomes de Almeida
Nº 87583

Geração de Pontos

Para todos os exercícios decidi encapsular a função que nos deram no enunciado para mais facilmente a poder utilizar:

```python
def generate_Points(plot: bool, alpha: float, pointN: int) -> (np.ndarray, np.ndarray, np.ndarray):
    mean = [3, 3]
    cov = [[1, 0], [0, 1]]
    a = np.random.multivariate_normal(mean, cov, int(pointN / 2)).T
    mean = [-3, -3]
    cov = [[2, 0], [0, 5]]
    b = np.random.multivariate_normal(mean, cov, int(pointN / 2)).T

    #mean = [5,-5]
    #cov = [[1, 0], [0, 1]]
    #b1 = np.random.multivariate_normal(mean, cov, int(pointN / 2)).T
    #c = np.concatenate((a, b, b1), axis=1)

    c = np.concatenate((a, b), axis=1)
    c = c.T
    np.random.shuffle(c)
    c = c.T
    if plot:
        # x = c[0]
        # y = c[1]
        # plt.plot(x, y, 'x')
        plt.scatter(a[0], a[1], marker='+', label="a", alpha=alpha, color=COLORS[0])
        plt.scatter(b[0], b[1], marker='+', label="b", alpha=alpha, color=COLORS[2])
        #plt.scatter(b1[0], b1[1], marker='+', label="b1", alpha=alpha, color=COLORS[4])
        # plt.axis('equal')
        # plt.show()
    return a, b, c
```

Podemos observar que deixei uma opção para poder fazer a visualização dos pontos gerados baseado no parâmetro plot que passo quando a função é chamada. Em relação ao código de geração em sim, alterei muito pouca coisa, foi principalmente a forma de fazer plot e deixei comentado um código adicional para gerar mais um cluster, para efetuar alguns testes.

1º Exercício

Olhando agora para a logica especifica do primeiro exercicio,

```python
def assign3_exercise1(seed:int):
    figure, axes = plt.subplots()

    a, b, c = generate_Points(plot=True, alpha=0.2, pointN=1000)
    r1ListBeggining: list = []
    r2ListBeggining: list = []
    r3ListBeggining: list = []
    r1ListEndOfPassage: list = []
    r2ListEndOfPassage: list = []
    r3ListEndOfPassage: list = []

    r1 = random.choice(c.T)
    r2 = random.choice(c.T)
```

```
    r3 = random.choice(c.T)
    iterations = 10
    for i in range(iterations):
        for point in c.T:
            r1Closeness: float = ((r1[0] - point[0]) ** 2 + (r1[1] - point[1]) ** 2)**0.5
            r2Closeness: float = ((r2[0] - point[0]) ** 2 + (r2[1] - point[1]) ** 2)**0.5
            r3Closeness: float = ((r3[0] - point[0]) ** 2 + (r3[1] - point[1]) ** 2)**0.5
            # print(point_index,r1Closeness,r2Closeness)
            if min(r1Closeness,r2Closeness,r3Closeness) == r1Closeness:
                r1 = (1 - alpha) * r1 + alpha * point
            elif min(r1Closeness,r2Closeness,r3Closeness) == r2Closeness:
                r2 = (1 - alpha) * r2 + alpha * point
            else:
                r3 = (1 - alpha) * r3 + alpha * point

            if i == 0:
                r1ListBeggining.append((r1[0], r1[1]))
                r2ListBeggining.append((r2[0], r2[1]))
                r3ListBeggining.append((r3[0], r3[1]))
        r1ListEndOfPassage.append((r1[0], r1[1]))
        r2ListEndOfPassage.append((r2[0], r2[1]))
        r3ListEndOfPassage.append((r3[0], r3[1]))

    exercise1_plot(
        [r1ListBeggining, r2ListBeggining, r3ListBeggining],
        [r1ListEndOfPassage, r2ListEndOfPassage, r3ListEndOfPassage])

    axes.set_aspect(1)
    plt.title("seed=" + str(seed)+" alpha="+str(alpha))
    plt.tight_layout()
    plt.legend()
    plt.show()
```

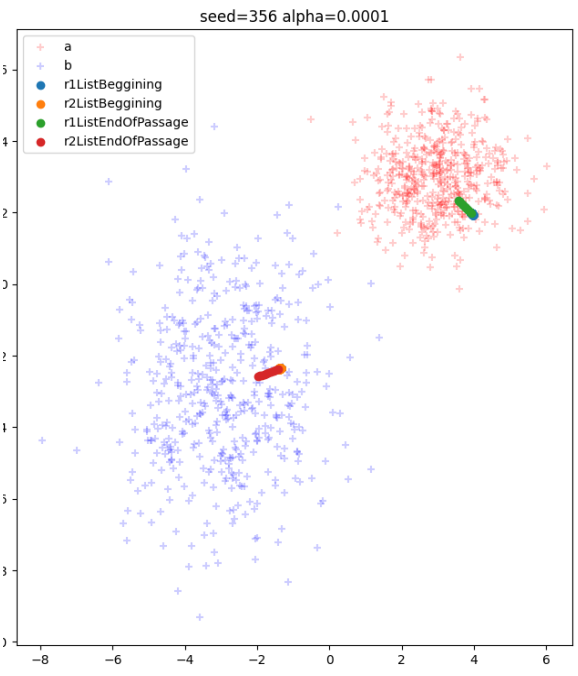
A lógica básica deste exercício consiste em escolher dois pontos aleatórios, diferentes, que serão os centros dos clusters. Depois dos pontos escolhidos começamos com as iterações, dentro de cada iteração percorro todos os pontos do dataset e verifico a sua proximidade a ambos os centros (r1 e r2) dependendo de qual é o centro mais próximo movo esse centro de acordo com a seguinte função:

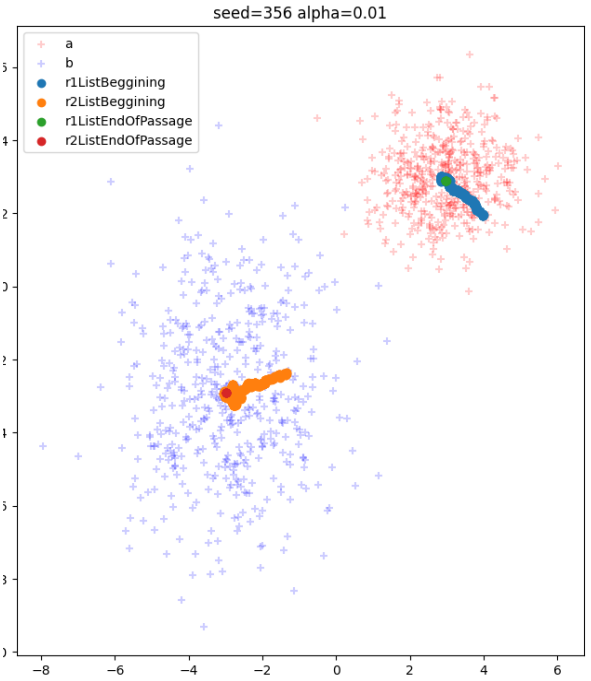
```
r1 = (1 - alpha) * r1 + alpha * point
```

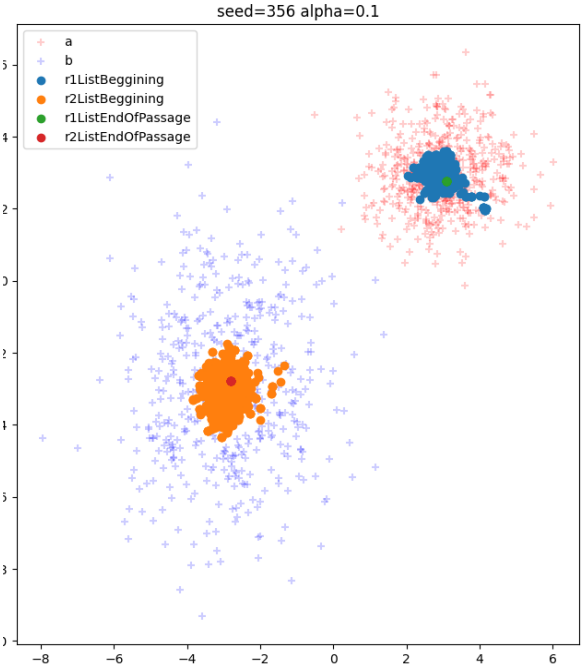
Nos gráficos seguintes podemos ver os resultados ao fim das iterações, onde os labels significam:
rXListBeggining = Posições consecutivas de r1 durante a primeira passagem/iteração
rXListEndOfPassage = Posição de r1 ao final de cada passagem/iteração

Onde X é qualquer número inteiro

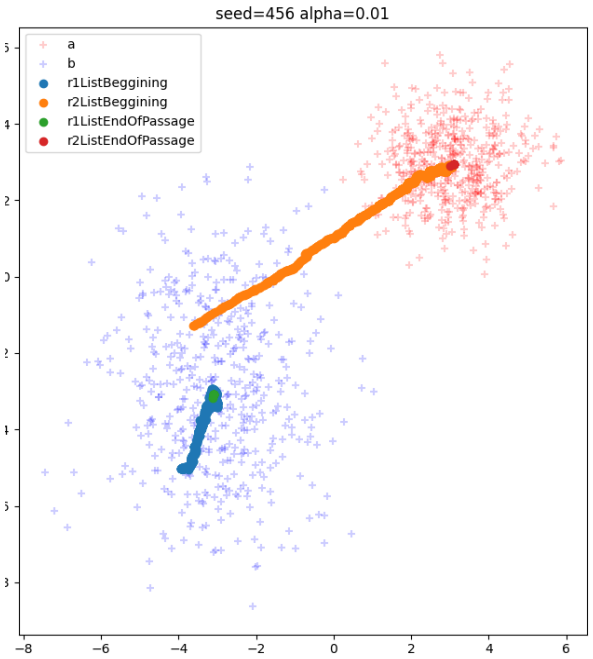
Os primeiros dois gráficos são relativos a um alpha igual a 0.0001 e 0,01 respetivamente



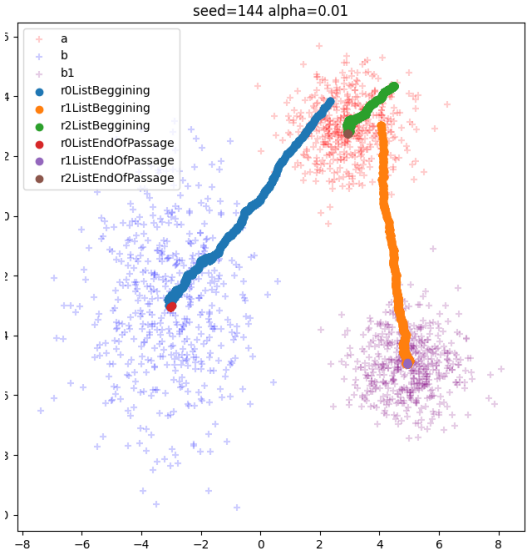




Claramente podemos observar que á medida que o alpha aumenta, mais rapidamente conseguimos chegar ao centro do cluster mas que a partir de um certo ponto, entre 0.01 e 0.1 os movimentos dos centros ficam muito caóticos e “impacientes”



Aqui podemos ver um exemplo, com uma seed diferente, onde os dois pontos começam no mesmo cluster e mesmo assim cada um chega ao centro do seu cluster respetivo.



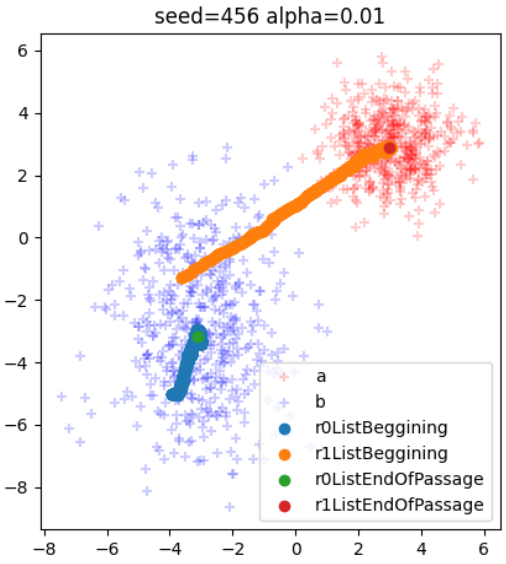
O código da função que estava comentado, foi utilizado para conduzir este teste, onde tenho 3 clusters e 3 pontos “r”, tal como no exemplo anterior, todos começaram no mesmo cluster mas conseguiram acabar no seu cluster respetivo.

What do you conclude about the evolution of the two points in the different situations?

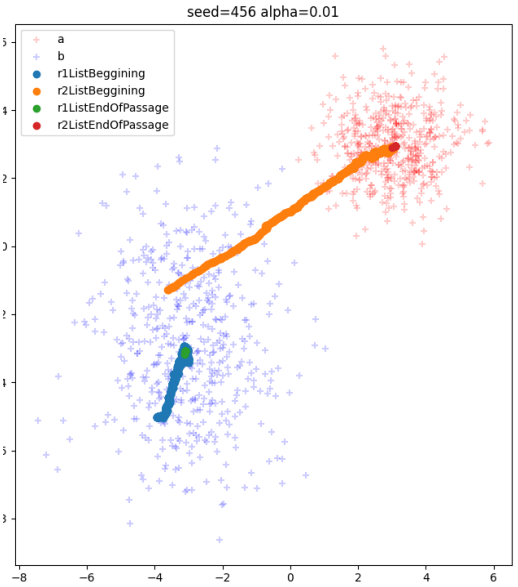
Com uma análise superficial, poderíamos concluir que nas diferentes situações a “accuracy” do nosso algoritmo conseguir determinar o centro correto do cluster depende muito mais do alpha que se usa do que do número de iterações que efetuou.

No seguinte exemplo utilizo a mesma seed para comparar justamente.

1 iteração:



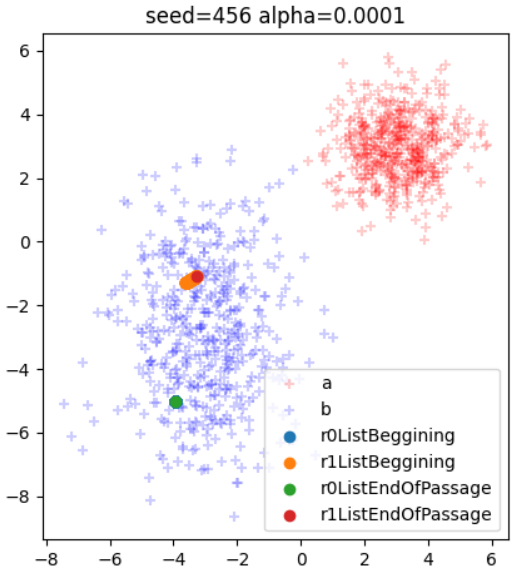
10 iterações:



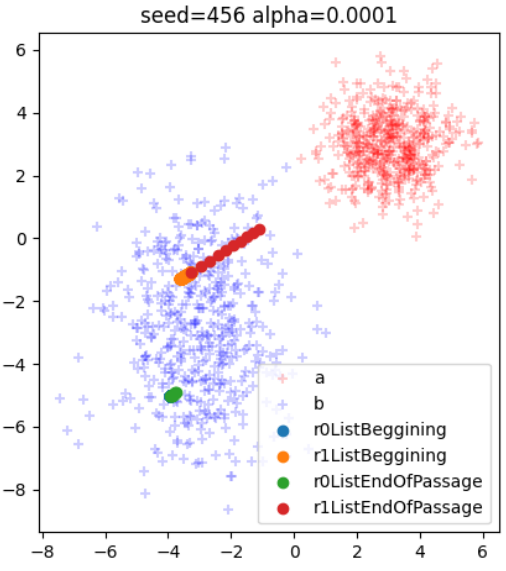
Como podemos observar, os gráficos gerados são quase indistinguíveis, as unicas diferenças notam-se nos pontos de EndOfPassage porque como houve mais passagens nas 10 iterações temos mais pontos com esse label, mas mesmo assim esses pontos caiem quase todos no mesmo sitio, confirmando que o resultado da primeira passagem já estava correto.

Mas agora analisando um exemplo com um alpha muito mais pequeno (passou de 0.01 para 0.0001)

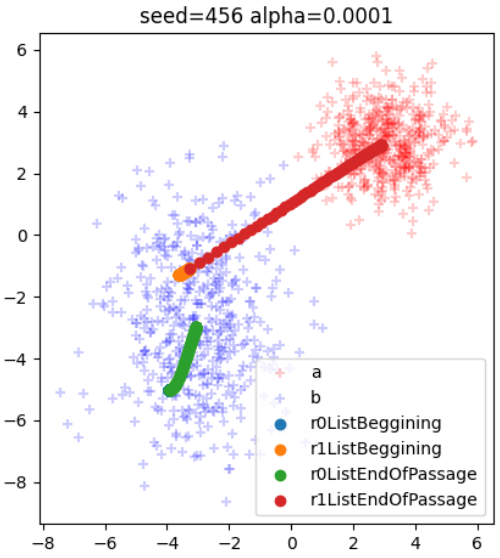
1 iteração:



10 Iterações:



100 iterações



Agora podemos ver que a maioria do “caminho” foi percorrido pelos List EndOfPassage, ou seja, a cada passagem os centros mudam pouco a sua posição e é ao longo das várias iterações que eles vão chegando ao centro correto.

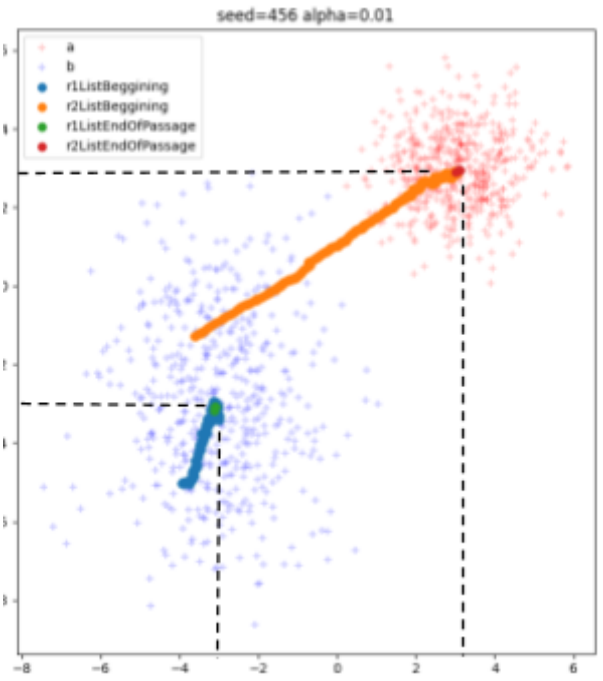
Com isto podemos concluir que, em ambos os casos, é apenas quando as posições dos centros estagnam que podemos considerar o algoritmo como terminado, numa situação com um alpha relativamente grande (alpha=1) pode ser que cheguemos a esse ponto com 2 ou 3 iterações mas quanto menor o alpha mais iterações demoramos para chegar ao ponto de estagnação.

Is there any relation between the (most common) final values of r1 and r2 and the parameters used to generate the dataset?

Os parâmetros usados para gerar o dataset são os seguintes:

Para o Cluster a:
mean = [-3, -3]
cov = [[2, 0], [0, 5]]

Para o cluster b:
mean = [3, 3]
cov = [[1, 0], [0, 1]]



Apesar da imagem ter ficado um pouco desfocada, podemos observar que ambos os centros gerados pelo algoritmo tem coordenadas muito próximas das usadas para os parâmetros mean dos dois clusters.

2º Exercício

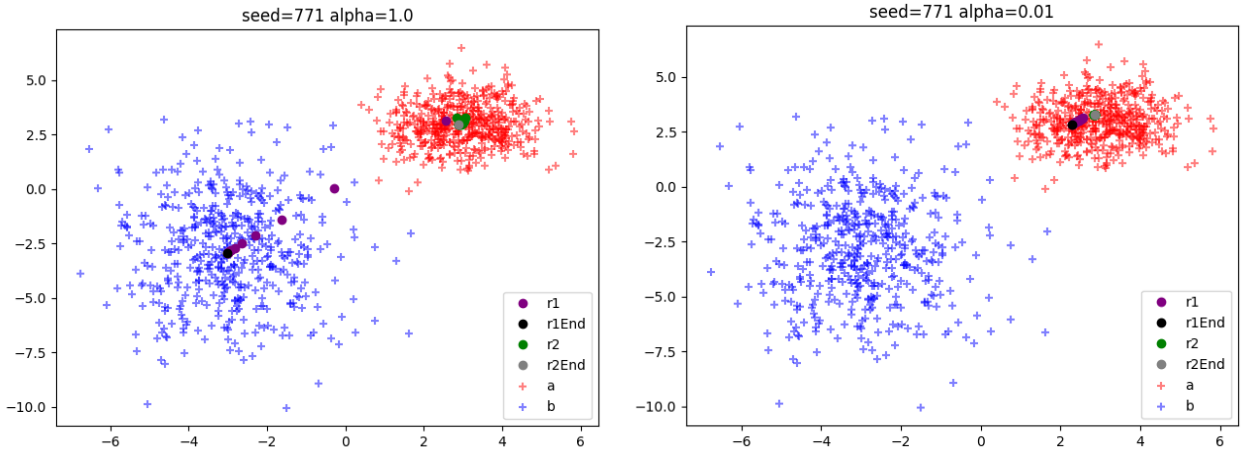
A grande diferença entre o exercício 1 e 2 é que no segundo em vez de mudarmos a posição dos centros dos clusters a cada ponto que analisamos, agora armazenamos a "diferença" num vetor e mudamos a posição do centro apenas depois de analizar todos os pontos do dataset. Ou seja, num dataset com 1000 pontos em vez de mudarmos a posição do centro 1000 vezes, mudamos apenas 1 tendo em conta todas as variações que cada ponto causa.

Na seguinte screenshot podemos ver o mecanismo principal usado para todo o 2º Exercício. Basicamente o que se está a fazer aqui são 10 (ou quantas eu quiser) iterações e em cada uma delas percorro todos os pontos no dataset e calculo se o ponto atual está mais próximo de r1 ou de r2 e baseado nisso atualizo as variáveis d1 e d2 com a diferença entre o ponto e a posição atual de r1 ou r2, se estiver na última iteração ainda executo a função de decode_positives_matrix que (ver abaixo) simplesmente armazena e organiza cada ponto numa das 4 listas diferentes baseando-se sempre na distância ao centro mais próximo e se esse ponto originalmente pertencia ao cluster A ou B.

```
    iterations = 10
    d1: np.ndarray = np.zeros(2)
    d2: np.ndarray = np.zeros(2)
    n_examples = len(c)
    for i in range(iterations):
        for point in c:
            r1Closeness: float = ((r1[0] - point[0]) ** 2 + (r1[1] - point[1]) ** 2) ** 0.5
            r2Closeness: float = ((r2[0] - point[0]) ** 2 + (r2[1] - point[1]) ** 2) ** 0.5
            if r1Closeness < r2Closeness:
                d1 = d1 + (point - r1)
            else:
                d2 = d2 + (point - r2)
            if i == (iterations - 1):
                decode_positives_matrix(a, b, point, points_closer_r1_label1, points_closer_r1_label2,
                                        points_closer_r2_label1, points_closer_r2_label2, r1Closeness, r2Closeness)

        r1 = r1 + (alpha / n_examples) * d1
        r2 = r2 + (alpha / n_examples) * d2
        d1 = np.zeros(2)
        d2 = np.zeros(2)
```

```
def decode_positives_matrix(a, b, point, points_closer_r1_label1, points_closer_r1_label2, points_closer_r2_label1,
                            points_closer_r2_label2, r1Closeness, r2Closeness):
    if r1Closeness < r2Closeness:
        # closer to r1 from a
        if point in a:
            points_closer_r1_label1.append(point)
        # closer to r1 from b
        elif point in b:
            points_closer_r1_label2.append(point)
    else:
        # closer to r2 from a
        if point in a:
            points_closer_r2_label1.append(point)
        # closer to r2 from b
        elif point in b:
            points_closer_r2_label2.append(point)
```

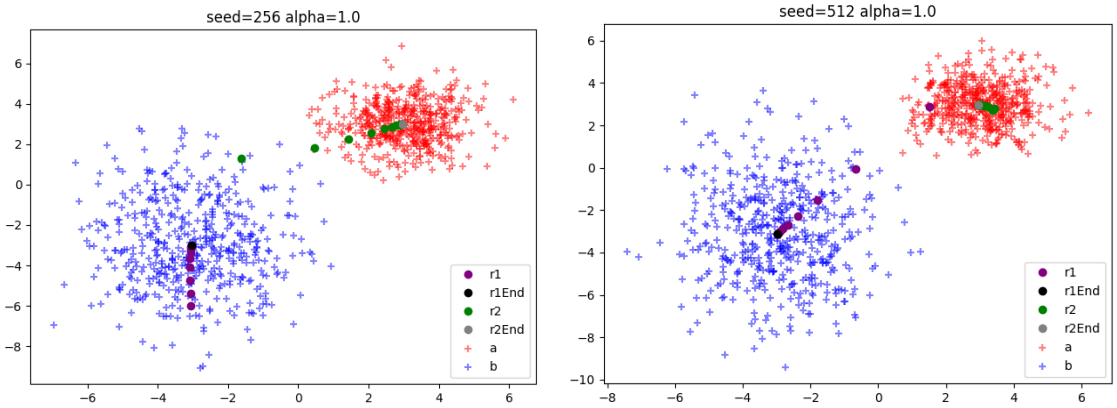
Alínea A



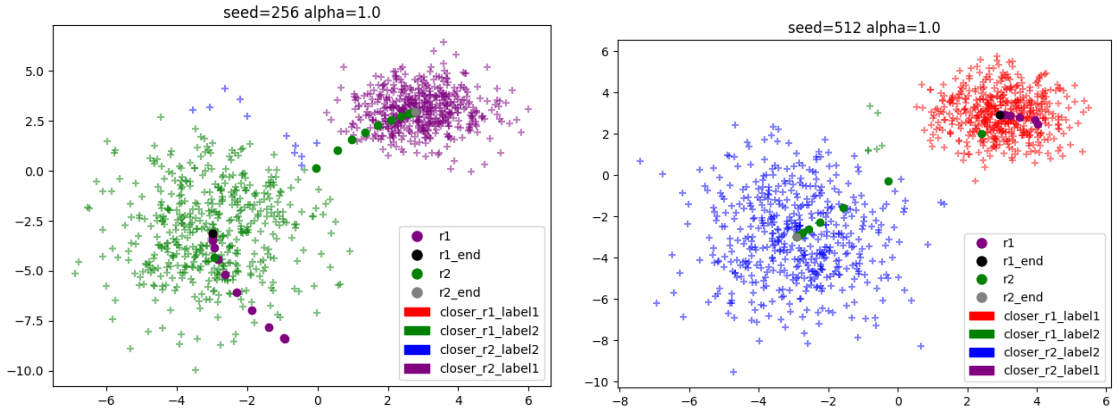
Como vimos no exercício 1, quanto menor o alpha menores serão os movimentos que os centros farão.

What do you observe?

Nos seguintes gráficos podemos ver que são bastante idênticos aos gráficos gerados pelo primeiro exercício. Com este algoritmo temos a vantagem de não efetuar tantos movimentos como anteriormente e também eliminámos a aleatoriedade que tínhamos, quando o alpha era grande (devido a movemo-nos a cada ponto analisado).



Alínea B

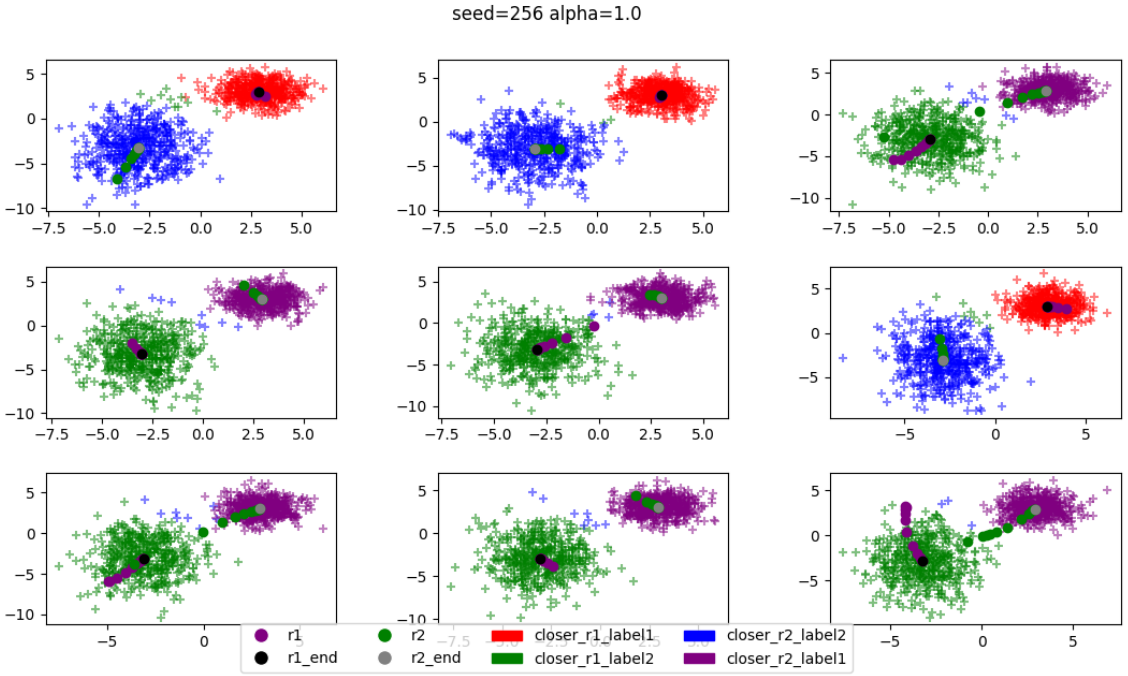


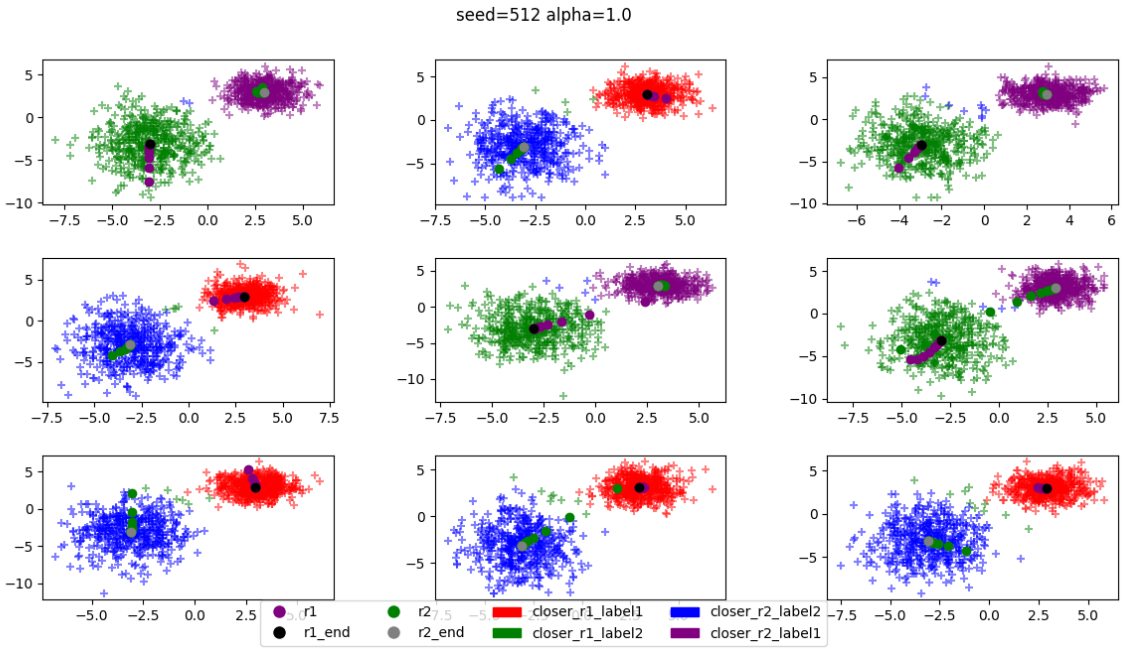
Ao fazer plot das 4 cores como sugerido pelo enunciado, no meu caso utilizei:

Vermelho: pontos perto de r1 e labeled 1
Verde: pontos perto de r1 mas labeled 2
Azul: pontos perto de r1 e labeled 1
Roxo: pontos perto de r1 mas labeled 2

Podemos ver um fenômeno interessante que é que á esquerda temos as cores trocadas, comparando com a direita, isso é porque o r1 e o r2 “decidiram” trocar de cluster, então as cores mudam, o r2 em vez de ir para o cluster de baixo foi para o de cima e o mesmo com r1, em vez de ficar em cima foi para baixo.

Alínea C





Aqui podemos ver o que acontece em muitos cenários diferentes, com posições iniciais diferentes.

3º Exercício

Class Point

Para o 3º e o 4º exercicio decidi deixar de usar os npArray do numpy para começar a usar a minha própria classe de pontos, tomei a decisão para aumentar a performance do algoritmo e também para poder adicionar os meus próprios atributos aos pontos.

A classe é relativamente pequena e tem o seguinte aspeto:

```python
class Point:
    x: float
    y: float
    label: str
    visited: bool

    def __init__(self, x: float, y: float, visited: bool = False, label: str = "", ):
        self.x = x
        self.y = y
        self.visited = visited
        self.label = label

    def __str__(self):
        round_digits = 5
        return self.label + "[" + str(round(self.x, round_digits)) + "," + str(round(self.y, round_digits)) + "]" + str(
            self.visited)

    def __repr__(self):
        return self.__str__()

    @classmethod
    def generate_Points(cls, alpha: float, plot: bool, pointN: int):
        _a, _b, _c = generate_Points_as_arrays(alpha=alpha, plot=plot, pointN=pointN)
        _points_list: [Point] = []
        for point in _a.T:
            _points_list.append(Point(x=point[0], y=point[1], label="a"))
        for point in _b.T:
            _points_list.append(Point(x=point[0], y=point[1], label="b"))
        return _points_list

```

Adicionei os parâmetros label e visited para poder fazer track a se o ponto já foi visitado ou não e o label para poder logo armazenar se o ponto pertence ao cluster A ou B, utilizei este

parâmetro em baixo no método da classe generate_Points que utiliza a função generate_Points_as_arrays que tem o seguinte aspeto:

```python
def generate_Points_as_arrays(plot: bool, alpha: float, pointN: int) -> (np.ndarray, np.ndarray, np.ndarray):
    mean = [3, 3]
    cov = [[1, 0], [0, 1]]
    a = np.random.multivariate_normal(mean, cov, int(pointN / 2)).T
    mean = [-3, -3]
    cov = [[2, 0], [0, 5]]
    b = np.random.multivariate_normal(mean, cov, int(pointN / 2)).T

    #mean = [5,-5]
    #cov = [[1, 0], [0, 1]]
    #b1 = np.random.multivariate_normal(mean, cov, int(pointN / 2)).T
    #c = np.concatenate((a, b, b1), axis=1)

    c = np.concatenate((a, b), axis=1)
    c = c.T
    np.random.shuffle(c)
    c = c.T
    if plot:
        # x = c[0]
        # y = c[1]
        # plt.plot(x, y, 'x')
        plt.scatter(a[0], a[1], marker='+', label="a", alpha=alpha, color=COLORS[0])
        plt.scatter(b[0], b[1], marker='+', label="b", alpha=alpha, color=COLORS[2])
        #plt.scatter(b1[0], b1[1], marker='+', label="b1", alpha=alpha, color=COLORS[4])
        # plt.axis('equal')
        # plt.show()
    return a, b, c
```

Ou seja, é basicamente o código dado no enunciado (apenas alterei a forma de fazer plot para poder ter legendas diferentes).

BruteForce

Ao início, a primeira implementação que fiz deste exercício foi bastante simples, percorria todos os pontos do dataset, calculando as distâncias entre eles, ou seja percorria todos os pontos e por cada ponto voltava a percorrer todos os pontos e calculava a distancia entre eles. Isto resultava em 1000*1000 cálculos de distâncias.
Após algum brainstorming com colegas da turma, chegámos a uma ideia de implementar uma matriz que guarda as distâncias entre todos os pontos e desse modo evita recalcular distâncias previamente calculadas.

Distance Matrix

Essa distance Matrix foi implementada com a sua própria classe e tem o seguinte aspeto:

[0.0, 0.91, 0.82, 1.17, 0.88, 0.62, 0.99, 0.99, 1.19, 0.41,]
[0.91, 0.0, 0.56, 0.27, 0.34, 0.52, 0.09, 0.12, 0.29, 0.68,]
[0.82, 0.56, 0.0, 0.75, 0.84, 0.2, 0.58, 0.67, 0.78, 0.42,]
[1.17, 0.27, 0.75, 0.0, 0.45, 0.75, 0.19, 0.19, 0.03, 0.94,]
[0.88, 0.34, 0.84, 0.45, 0.0, 0.73, 0.39, 0.29, 0.45, 0.81,]
[0.62, 0.52, 0.2, 0.75, 0.73, 0.0, 0.57, 0.63, 0.78, 0.23,]
[0.99, 0.09, 0.58, 0.19, 0.39, 0.57, 0.0, 0.11, 0.21, 0.75,]
[0.99, 0.12, 0.67, 0.19, 0.29, 0.63, 0.11, 0.0, 0.2, 0.78,]
[1.19, 0.29, 0.78, 0.03, 0.45, 0.78, 0.21, 0.2, 0.0, 0.96,]
[0.41, 0.68, 0.42, 0.94, 0.81, 0.23, 0.75, 0.78, 0.96, 0.0,]

Cada row e cada coluna representa um ponto no dataset, o exemplo acima é só com um dataset bem pequeno que usei para testar a classe.

Portanto a row0 e a coluna0 representam o ponto 0 do dataset enquanto a row1 e coluna1 representam o ponto1 e sucessivamente. Daí a diagonal da matriz é sempre zero porque é onde está calculada a distância entre um ponto e ele próprio.

Na realidade a matriz contém a mesma distância calculada duas vezes porque calcular a distância entre o Ponto0 e Ponto1 é igual a calcular Ponto1 com Ponto0 mas decidi manter esta redundância para facilitar a operação de determinar qual o ponto mais próximo de outro ponto, podendo apenas analisar a row do ponto que estou interessado em obter o mínimo valor dentro da row para saber qual o ponto mais próximo.

Na classe da DistanceMatrix implementei operações de adicionar um novo ponto á matriz, remover um ponto existente, get_points_inside_epsilon, que retorna todos os pontos dentro de uma certa distância (epsilon) do ponto que definimos como centro, get_dist_between que retorna a distância entre dois pontos dados, get_closest_pair que retorna o par de pontos mais próximos dentro da matriz e por fim a representação em string da classe.

A função get_points_inside_epsilon é apenas usada para o 4º exercicio.

Node

Para poder fazer “track” aos pontos que dão origem aos pontos finais tive de implementar uma binary tree onde cada node tem três atributos, os seus dados, a node da direita e a da esquerda que são os seus “pais”

```
6   class Node:
7
8       # Construct to create a newNode
9       def __init__(self, key):
10          self.data = key
11          self.left = None
12          self.right = None
13
14      # Function to print binary tree in 2D
```

Depois também copiei da internet os seguintes métodos para poder facilmente fazer print da node e todos os seus pais em forma de texto:

```
        self.right = None

    # Function to print binary tree in 2D
    # It does reverse inorder traversal
    @classmethod
    def print2DUtil(cls, root, space, maxdepth):
        currdepth = space / COUNT

        # Base case
        if root is None or currdepth > maxdepth:
            return

        # Increase distance between levels
        space += COUNT

        # Process right child first
        Node.print2DUtil(root.right, space, maxdepth)

        # Print current node after space
        for i in range(COUNT, space):
            print(end=" ")
        print(root.data)

        # Process left child
        Node.print2DUtil(root.left, space, maxdepth)

    # Wrapper over print2DUtil()
    @classmethod
    def print2D(cls, root, depth: int):
        # space=[0]
        # Pass initial space count as 0
        Node.print2DUtil(root, 0, depth)
```

TreeManager

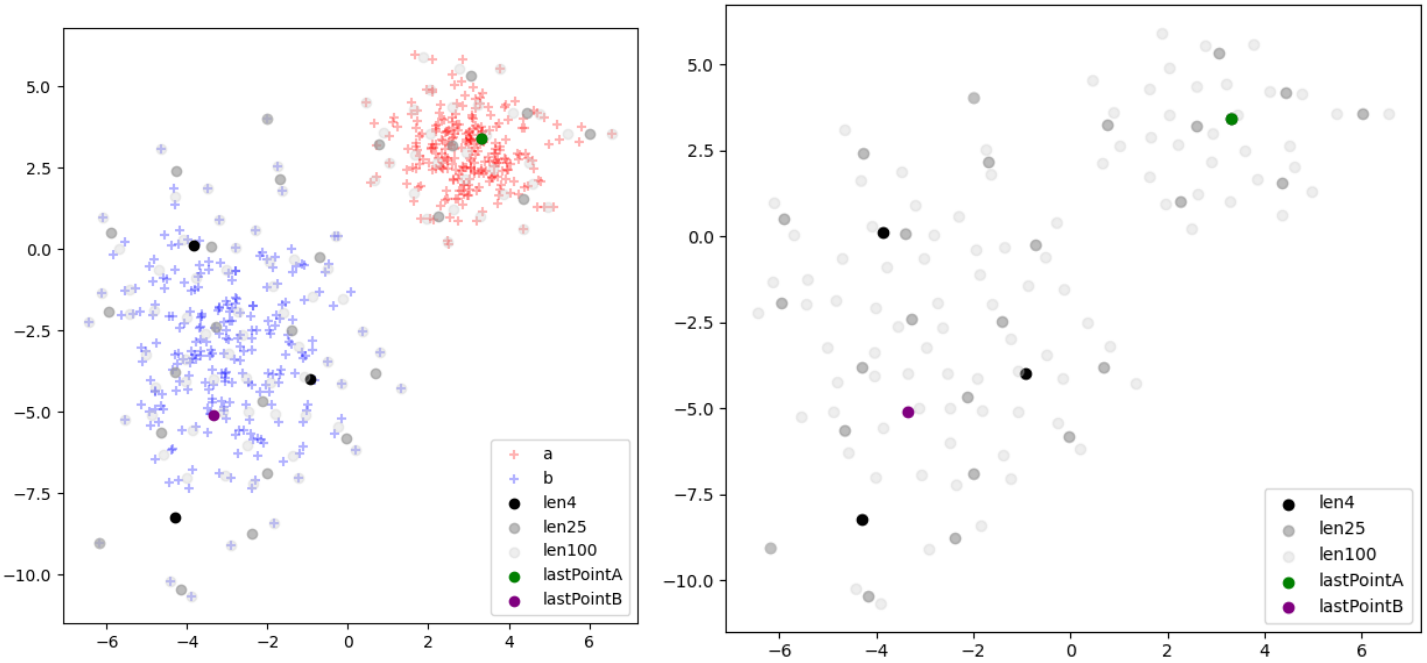
Para armazenar todas as nodes e poder obter uma node que não estaria atualmente no loop de execução do algoritmo criei a classe TreeManager para poder ter uma lista com todas as Nodes, podendo assim criar nodes á “vontade” e fazer o get delas posteriormente.

```
class TreeManager:
    Assign3_Nodes_List: [Node]

    def __init__(self):
        self.Assign3_Nodes_List = []

    def get(self, content: [float, float]):
        for node in self.Assign3_Nodes_List:
            if node.data == content:
                return node
        return Node(content)

    def exists(self, node: Node) -> bool:
        return node in self.Assign3_Nodes_List

    def add(self, node: Node):
        if not self.exists(node):
            self.Assign3_Nodes_List.append(node)
        else:
            raise Exception(node, "Node already exists in Tree")

    def build(self):
        depth = 3
        print("----------FirstNode----------")
        print(Node.print2D(self.Assign3_Nodes_List[-1], depth))
        print("----------SecondNode----------")
        print(Node.print2D(self.Assign3_Nodes_List[-2], depth))

    @classmethod
    def get_all_parents(cls, root_node: Node):
```

```
        print(Node.print2D(self.Assign3_Nodes_List[-2], depth))

    @classmethod
    def get_all_parents(cls, root_node: Node):
        def aux(children_list: list, node: Node):
            children_list.append(node)
            if node.right is not None:
                aux(children_list=children_list, node=node.right)
            if node.left is not None:
                aux(children_list=children_list, node=node.left)

        children: [Node] = []
        aux(node=root_node, children_list=children)
        return children
```

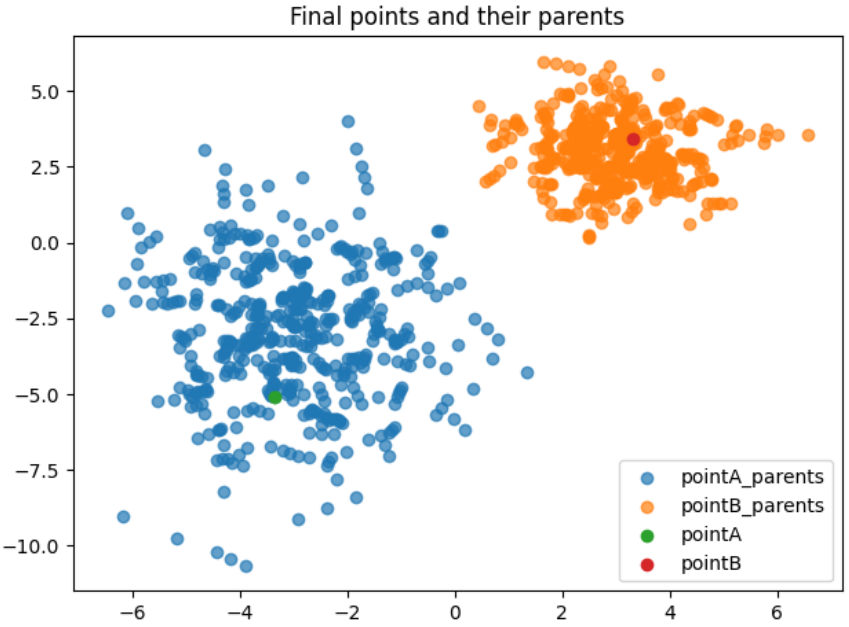
Resultados

Os resultados usando a seed 1206 com um dataset com 500 pontos são os seguintes



À esquerda podemos ver a azul e vermelho os dois clusters iniciais e com os labels len100, len 25 e len 4 podemos ver os pontos que pertenciam ao nosso dataset quando o mesmo tinha 100, 25 e 4 pontos apenas nele, apesar de ser um pouco confuso mas é como se estivéssemos a ver os pontos finais realçados sobre os pontos iniciais, à direita podemos ver esse efeito melhor, devido a não estar presente o plot dos clusters iniciais.

Em baixo podemos ver o que resultou de ter implementado a classe Node e TreeManager, com elas consegui obter todos os parentes de cada ponto final e fazer o seu plot, que resultou no seguinte gráfico:



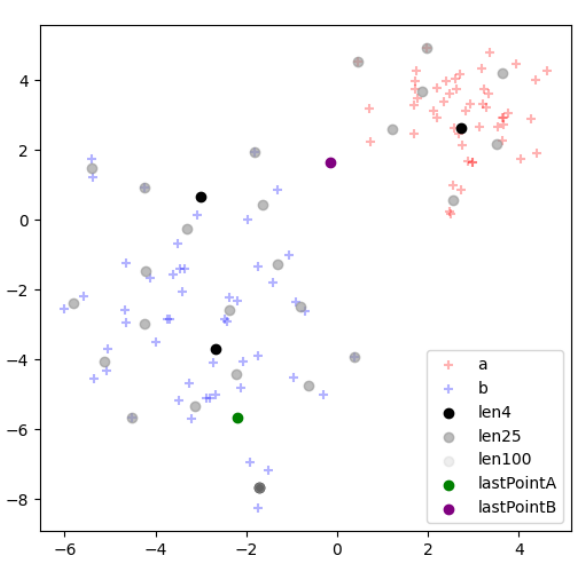
Também obtive esta tree de pontos no final onde podemos ver exatamente as coordenadas dos pontos finais e os seus parentes:

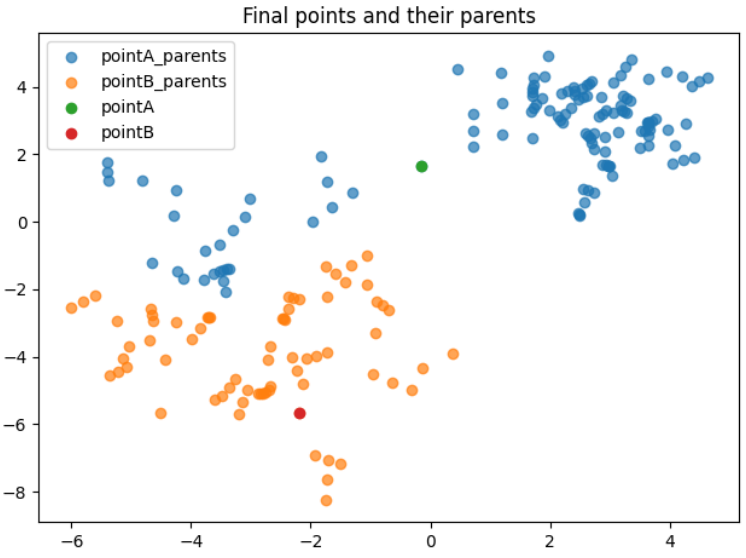
```
----------FirstNode----------
                              b[-6.17707,-9.03783]False
                    avg[-5.17137,-9.74047]False
                              avg[-4.16566,-10.4431]False
          avg[-4.29676,-8.23637]False
                              avg[-4.65022,-5.64412]False
                    avg[-3.42215,-6.73226]False
                              avg[-2.19409,-7.82041]False
avg[-3.34393,-5.08523]False
                              avg[-5.92659,-0.7106]False
                    avg[-3.85303,0.12805]False
                              avg[-1.77948,0.96669]False
          avg[-2.39111,-1.93409]False
                              avg[0.33012,-4.81315]False
                    avg[-0.92918,-3.99623]False
                              avg[-2.18848,-3.17931]False
None
----------SecondNode----------
                              avg[-5.90161,0.49885]False
                    avg[-5.92659,-0.7106]False
                              avg[-5.95158,-1.92005]False
          avg[-3.85303,0.12805]False
                              avg[-0.71736,-0.2413]False
                    avg[-1.77948,0.96669]False
                              avg[-2.84159,2.17469]False
avg[-2.39111,-1.93409]False
                              avg[-0.03311,-5.8068]False
                    avg[0.33012,-4.81315]False
                              avg[0.69335,-3.81951]False
          avg[-0.92918,-3.99623]False
                              avg[-1.4146,-2.48018]False
                    avg[-2.18848,-3.17931]False
                              avg[-2.96237,-3.87843]False
None
```

A tree está organizada da esquerda para a direita, na esquerda os dois pontos finais, representados no gráfico e á direita os parentes que os originaram. A geração do gráfico funciona á base de um spacing pré definido (neste caso são 4 espaços) logo os dois pontos que se encontram na mesma coluna de texto originaram o ponto entre eles mais á esquerda.

Poucos pontos

Na minha opinião os resultados ficam mais interessantes quando reduzimos o numero de pontos no dataset.
Ficam interessantes porque não são os resultados que estariamos á espera, nem os corretos a maioria das vezes mas como temos menos pontos, significa que cada ponto influencia muito mais a posição final dos centros o que resulta no seguinte:
(com a mesma seed de 1206)





Como podemos ver, nada o que estávamos à espera. Ambos os centros estão totalmente fora do que são os clusters A e B iniciais.
Podemos ver na imagem da direita todos os pontos que geraram o pontoA a azul e os do pontoB a laranja.

4º Exercício

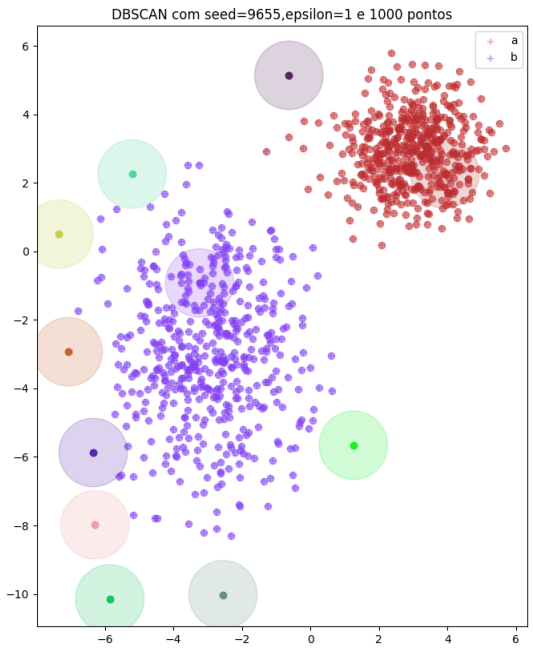
Cluster

Para a implementação da classe Cluster não são necessárias muitas variáveis, apenas uma lista de pontos e o ponto inicial (na realidade o ponto inicial poderia ser o primeiro ponto da lista mas decidi separar apenas para facilidade e clareza)
Aqui a “magia” está mais nas funções:

```python
class Cluster:
    _points_list: [Point]
    _initial_point: Point

    def __init__(self, epsilon: float, distance_matrix: DistanceMatrix):
        self._points_list = []
        initial_point = random.choice(distance_matrix.points_list)
        while initial_point.visited:
            initial_point: Point = random.choice(distance_matrix.points_list)
        self._initial_point = initial_point
        self._initial_point.visited = True
        self._points_list.append(self._initial_point)
        initial_point_index = distance_matrix.points_list.index(self._initial_point)


        def aux( distance_matrix, epsilon, initial_point):
            initial_point.visited = True
            self.addPoint(point=initial_point)

            first_layer = distance_matrix.get_points_inside_epsilon(center_point=initial_point, epsilon=epsilon)
            for point in first_layer:
                point.visited = True
                self.addPoint(point=point)
                # distance_matrix.remove_point(point=point)
            for point in first_layer:
                aux(distance_matrix, epsilon, point)
                # distance_matrix.remove_point(point=point2)
        aux(distance_matrix, epsilon, initial_point)

    def getInitial_Point(self) -> Point:
        return self._initial_point

    def addPoint(self, point: Point):
        if point not in self._points_list:
            self._points_list.append(point)

    def addPointList(self, points: [Point]):
        for point in points:
            self.addPoint(point)
```

Para além de uns métodos simples de getters e adders, implementei um construtor bastante complexo, que na realidade obtém logo todos os pontos que pertencem ao cluster e altera os tais pontos para indicar que estes já foram visitados.

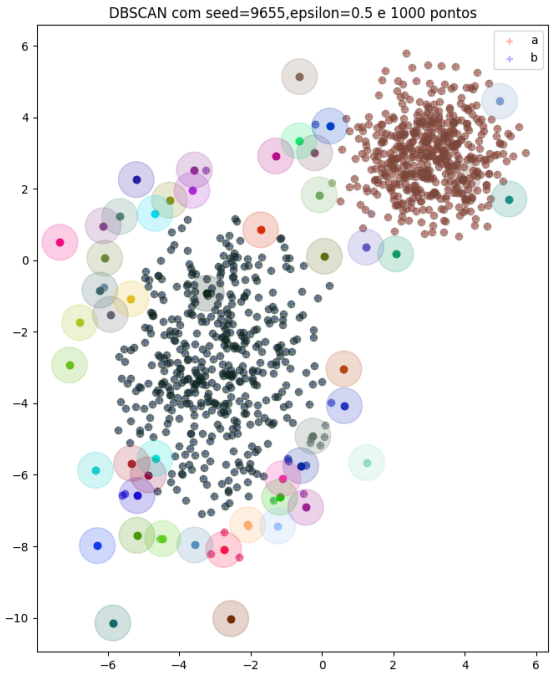
Resultados

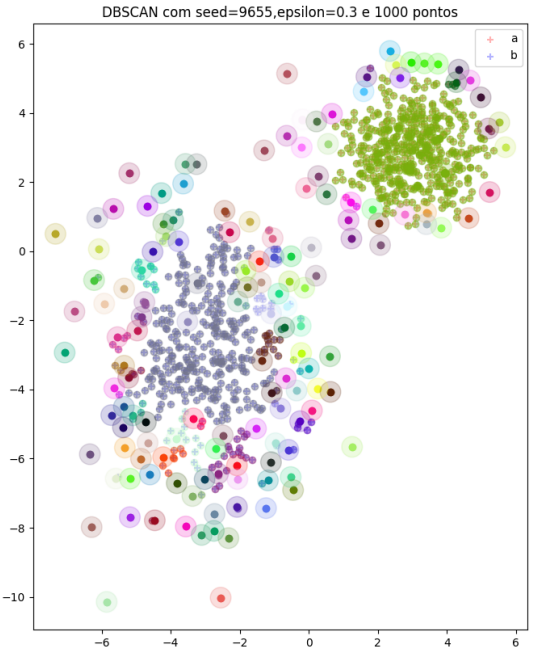


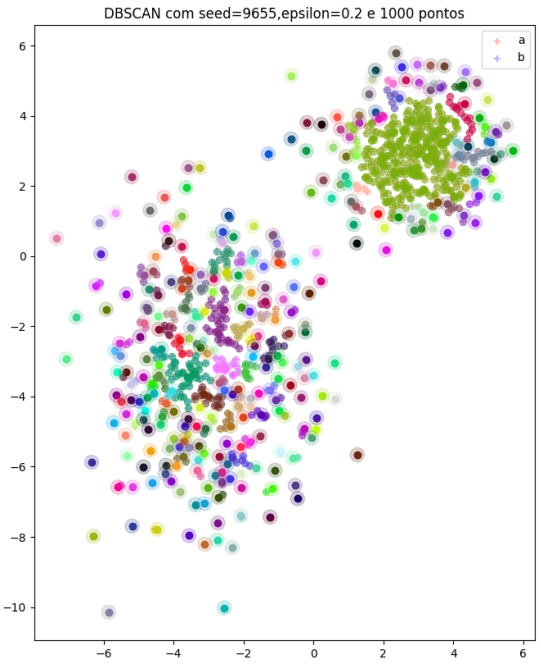
Podemos ver aqui o resultado da execução do algoritmo com um epsilon = 1, usando a seed 9655 e um dataset com 1000 pontos.

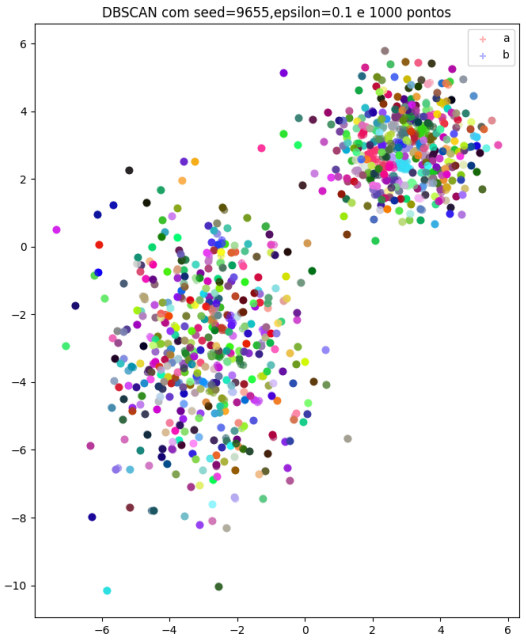
Cada cor corresponde a um cluster e os pontos iniciais de cada cluster têm um circulo com raio = epsilon para demonstrar os pontos

Conseguimos obter resultados bastante mais interessantes se formos reduzindo o epsilon, podemos ver qual o threshold dos nossos clusters









É interessante que podemos ver os clusters a diminuírem consoante o epsilon até chegar a um ponto (epsilon=0.1) onde já nem conseguimos distinguir clusters de pontos individuais, na última figura.