Universidade Federal do Rio de Janeiro
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

Programa de Engenharia de Sistemas e Computação

CPS844 - Inteligência Computacional I

Prof. Dr. Carlos Eduardo Pedreira

Trabalho prático

Luiz Henrique Souza Caldas

email: lhscaldas@cos.ufrj.br

3 de junho de 2024

1 Perceptron

Neste problema, você criará a sua própria função target f e uma base de dados D para que possa ver como o Algoritmo de Aprendizagem Perceptron funciona. Escolha $d = 2$ pra que você possa visualizar o problema, e assuma $\chi = [-1, 1] \times [-1, 1]$ com probabilidade uniforme de escolher cada $x \in \mathcal{X}$.

Em cada execução, escolha uma reta aleatória no plano como sua função target f (faça isso selecionando dois pontos aleatórios, uniformemente distribuídos em $\chi = [-1, 1] \times [-1, 1]$, e pegando a reta que passa entre eles), de modo que um lado da reta mapeia pra $+1$ e o outro pra -1. Escolha os inputs x_n da base de dados como um conjunto de pontos aleatórios (uniformemente em $\mathcal{X}$), e avalie a função target em cada x_n para pegar o output correspondente y_n.

Agora, pra cada execução, use o Algoritmo de Aprendizagem Perceptron (PLA) para encontrar g. Inicie o PLA com um vetor de pesos w zerado (considere que $sign(0) = 0$, de modo que todos os pontos estejam classificados erroneamente ao início), e a cada iteração faça com que o algoritmo escolha um ponto aleatório dentre os classificados erroneamente. Estamos interessados em duas quantidades: o número de iterações que o PLA demora para convergir pra g, e a divergência entre f e g que é $\mathbb{P}[f(x) \neq g(x)]$ (a probabilidade de que f e g vão divergir na classificação de um ponto aleatório). Você pode calcular essa probabilidade de maneira exata, ou então aproximá-la ao gerar uma quantidade suficientemente grande de novos pontos para estimá-la (por exemplo, 10.000).

A fim de obter uma estimativa confiável para essas duas quantias, você deverá realizar 1000 execuções do experimento (cada execução do jeito descrito acima), tomando a média destas execuções como seu resultado final.

Para ilustrar os resultados obtidos nos seus experimentos, acrescente ao seu relatório gráficos scatterplot com os pontos utilizados para calcular E_{out}, assim como as retas correspondentes à função target e à hipótese g encontrada.

Implementação:

Para responder os itens referentes a este problema, foi implementado em Python um Perceptron 2D utilizando as fórmulas e procedimentos apresentados na primeira aula do professor Yaser Abu-Mostafa. Foram criadas três classes: uma para gerar a função target (código 1), outra para gerar base de dados(código 2) usando a função target, e a terceira pra criar e treinar o perceptron e classificar utilizando o mesmo (código 3). Os seus códigos estão listados abaixo.

Código 1: Geração da função target f

```
1    # Classe para criar a função target
2    class Target:
3        def __init__(self):
4            self.a = 0 # coeficiente angular
5            self.b = 0 # coeficiente linear
6
7        # Método para gerar a linha da função target
8        def generate_random_line(self):
9            point1 = np.random.uniform(-1, 1, 2) # ponto aleatorio no domínio
10           point2 = np.random.uniform(-1, 1, 2) # ponto aleatorio no domínio
11           a = (point2[1] - point1[1]) / (point1[0] - point2[0]) # cálculo do
                 coeficiente angular
12           b = point1[1] - a*point1[0] # cálculo do coeficiente linear
13           self.a = a
```

```
            self.b = b
            return a, b

        # Método para classificar pontos de acordo a função target
        def classify_point(self, point):
            a = self.a
            b = self.b
            y_reta = a*point[0] + b
            return np.sign(point[1] - y_reta) # verifica se a coordenada y do
                ponto está acima ou abaixo da reta
```

Código 2: Geração do da base de dados D

```
    # Classe para criar o dataset
    class Dataset:
        def __init__(self, N):
            self.N = N # tamanho do dataset

        # Método para gerar a base de dados D
        def generate_dataset(self, target):
            N = self.N
            data = np.random.uniform(-1, 1, (N, 2)) # gera N pontos no R2 com
                coordenadas entre [-1, 1]
            labels = np.array([target.classify_point(point) for point in data])
            return data, labels
```

Código 3: Perceptron

```
    # Classe para criar e treinar o perceptron 2D
    class Perceptron2D:
        def __init__(self, weights = np.zeros(3)):
            self.w = weights # inicializa os pesos (incluindo o w_0)

            # Método para treinar o perceptron usando o PLA
            def pla(self, data, labels):
                n_samples = len(data)
                X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona
                    uma coluna de 1s para o X_0 (coordenada artificial)
                iterations = 0
                while True:
                    predictions = np.sign(np.dot(X_bias, self.w))
                    errors = labels != predictions
                    if not np.any(errors):
                        break
                    misclassified_indices = np.where(errors)[0]
                    random_choice = np.random.choice(misclassified_indices)
                    self.w += labels[random_choice] * X_bias[random_choice] #
                        atualiza os pesos
                    iterations += 1
                return iterations, self.w

        # Método para classificar um dataset com base nos pesos aprendidos.
        def classificar(self, data):
            n_samples = len(data)
```

```
            X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma
                coluna de 1s para o bias X_0
            return np.sign(np.dot(X_bias, self.w)) # verifica o sinal do
                produto escalar entre x e w
```

Para testar as classes, foram feitas duas funções: uma para plotar uma base de dados junto com uma função target f e uma hipótese g (código 4) e outra para gerar uma função target f, uma base de dados, e uma hipótese g (um perceptron com pesos calculados pelo PLA) para serem plotadas pela primeira função (código 5). O resultado pode ser observado na figura 1.

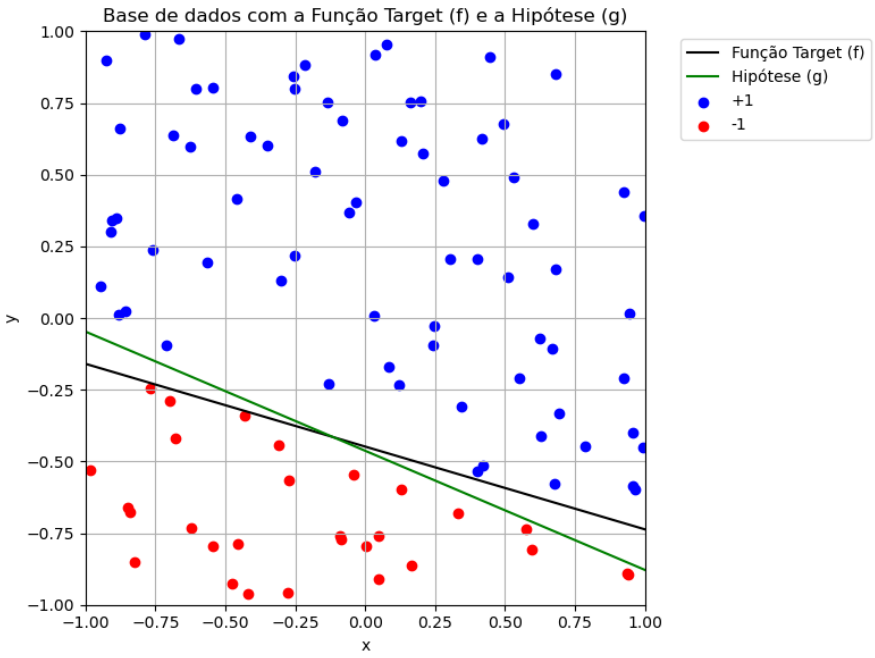
Código 4: Plotagem de dataset com função target (f) e hipótese (g)

```
def scatterplot(data, labels, target, hipotese):
a, b = target.a, target.b
w = hipotese.w
# Plotar resultados
plt.figure(figsize=(8, 6))
x_pos = [data[i][0] for i in range(len(data)) if labels[i] == 1]
y_pos = [data[i][1] for i in range(len(data)) if labels[i] == 1]
x_neg = [data[i][0] for i in range(len(data)) if labels[i] == -1]
y_neg = [data[i][1] for i in range(len(data)) if labels[i] == -1]
plt.scatter(x_pos, y_pos, c='blue', label='+1')
plt.scatter(x_neg, y_neg, c='red', label='-1')
x = np.linspace(-1, 1, 100)
y_target = a*x+b
y_g = -(w[1] * x + w[0]) / w[2]
plt.plot(x, y_g, 'g-', label='Hipótese (g)')
plt.plot(x, y_target, 'k-', label='Função Target (f)')
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Base de dados com a Função Target (f) e a Hipótese (g)')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout(rect=[0, 0, 1, 1])
plt.grid(True)
plt.show()
```

Código 5: Teste das classes

```
def teste():
    # Criar a função target
    target = Target()
    a, b = target.generate_random_line()
    # Criar o dataset
    num_points = 100
    dataset = Dataset(num_points)
    data, labels = dataset.generate_dataset(target)
    # Criar e treinar o perceptron
    perceptron = Perceptron2D()
    _, w = perceptron.pla(data,labels)
    # Plotar resultados
    scatterplot(data, labels, target, perceptron)
```

Figura 1: Base de dados com a Função Target (f) e a Hipótese (g)



Pela figura 1, é possível observar que a Hipótese (g) gera uma linha que, apesar de não sobrepor completamente a linha da Função Target (f), separa os pontos perfeitamente, resultado já esperado pelo treinamento utilizando o PLA, uma vez que esse algoritmo só converge quando o erro dentro da amosta E_{in} é nulo.

1. Considere $N = 10$. Quantas iterações demora, em média, para que o PLA convirja com $N = 10$ pontos de treinamento? Escolha o valor mais próximo do seu resultado.

   (a) 1

   (b) 15

   (c) 300

   (d) 5000

   (e) 10000

   Justificativa:

   Para responder a esse item foi implementada a seguinte função:

Código 6: Cálculo do número de iterações

```
1        def calc_num_iter(num_points, verbose = True):
2        lista_iter = list()
3        for _ in range(1000):
4            target = Target()
5            target.generate_random_line()
6            dataset = Dataset(num_points)
7            data, labels = dataset.generate_dataset(target)
```

```
            perceptron = Perceptron2D()
            iter, _ = perceptron.pla(data,labels)
            lista_iter.append(iter)
        if verbose: print(f"{np.mean(lista_iter)} iterações com desvio
           padrão {np.std(lista_iter):.4f} (min:{np.min(lista_iter)}, máx
           :{np.max(lista_iter)})")
        return lista_iter
```

O resultado após 1000 execuções do experimento, com a variável $num\_points = 10$, foi uma média de $8.656 (\approx 9)$ iterações, com desvio padrão de $29.5039 (\approx 30)$ iterações, mínimo de 0 iterações e máximo de 719 iterações. Nota-se que o número de iterações pode variar bastante entre uma execução e outra, o que se deve ao alto grau de aleatoriedade presente no problema, uma vez que tanto os pontos quanto a reta da Função Target são gerados de forma aleatória, além do ponto selecionado para se atualizar os erros também ser aleatório, podendo levar a configurações de diferentes níveis de dificuldade para o PLA convergir. Como 9 está mais próximo de 15 do que de 1, o **item b** foi selecionado.

2. Qual das alternativas seguintes é mais próxima de $\mathbb{P}[f(x) \neq g(x)]$ para $N = 10$?

   (a) 0.001

   (b) 0.01

   (c) 0.1

   (d) 0.5

   (e) 1

Justificativa:

$\mathbb{P}[f(x) \neq g(x)]$ pode ser estimada computacionalmente gerando-se uma quiantidade suficientemente grande de pontos novos e calculando o percentual de erro na classificação desses pontos. Para responder esse item foram realizadas 1000 execuções, nas quais foram gerados 10010 pontos, sendo 10 utilizados para o treinamento do perceptron e 10 mil utilizados para teste. A cada execução foi calculado o perceltual de erro, isto é, a quantidade de vezes que a classificação do perceptron foi diferente da função target divido pela quantidade de pontos. No final das execuções, $\mathbb{P}[f(x) \neq g(x)]$ foi estimada fazendo a média do percentual de erro em cada execução. A Implementação pode ser vista abaixo:

Código 7: Cálculo da probabilidade de erro

```
def calc_p_erro(num_points, verbose = True):
lista_erro = list()
for _ in range(1000):
    target = Target()
    target.generate_random_line()
    dataset_train = Dataset(num_points)
    x_train, y_train = dataset_train.generate_dataset(target)
    dataset_test = Dataset(10000) # mais 10mil pontos
    x_test, y_test = dataset_test.generate_dataset(target)
    perceptron = Perceptron2D()
    perceptron.pla(x_train,y_train)
```

```
            y_predicted = perceptron.classificar(x_test)
            erro = np.mean(y_test != y_predicted)
            lista_erro.append(erro)
        if verbose: print(f"P[f(x)\u2260g(x)] = {np.mean(lista_erro):.4f}"
            )
        return lista_erro
```

O resultado após 1000 execuções, com $num\_points = 10010$ e $train\_size = 10$, foi $\mathbb{P}[f(x) \neq g(x)] = 0.0959 = 9.59\%$. Como 0.0959 está mais próximo de 0.1 do que de 0.01, o **item c** foi selecionado.

3. Agora considere $N = 100$. Quantas iterações demora, em média, para que o PLA convirja com N = 100 pontos de treinamento? Escolha o valor mais próximo do seu resultado.

   (a) 50
   (b) 100
   (c) 500
   (d) 1000
   (e) 5000

   Justificativa:

   Para responder a este item foi utilizada a mesma função do item 1 (código 6), com $num\_points = 100$.

   O resultado após 1000 execuções do experimento foi uma média de $76.006 (\approx 76)$ iterações, com desvio padrão de $181.3750 (\approx 181)$ iterações, mínimo de 1 iteração e máximo de 2598 iterações. Nota-se que novamente o número de iterações pode variar bastante entre uma execução e outra, conforme já observado e explicado no item 1. Como 33 está abaixo de 50 e não existe alternativa menor, o **item b** foi selecionado.

4. Qual das alternativas seguintes é mais próxima de $\mathbb{P}[f(x) \neq g(x)]$ para $N = 100$?

   (a) 0.001
   (b) 0.01
   (c) 0.1
   (d) 0.5
   (e) 1

   Justificativa:

   Para responder a este item foi utilizada a mesma função do item 2 (código 7), com $num\_points = 100$.

   O resultado após 1000 execuções foi $\mathbb{P}[f(x) \neq g(x)] = 0.0126 = 1.26\%$. Como 0.0126 está mais próximo de 0.01 do que de 0.1, o **item b** foi selecionado.

5. É possível estabelecer alguma regra para a relação entre N, o número de iterações até a convergência, e $\mathbb{P}[f(x) \neq g(x)]$?

Resposta:

Para responder a este item foram implementadas duas funções: uma para calcular o número de iterações e $\mathbb{P}[f(x) \neq g(x)]$ para uma faixa de diferentes números de pontos (código 8) e outra para plotar os resultados (código 9). O código das duas pode ser observado abaixo.

Código 8: Cálculo da probabilidade de erro e do número de iterações

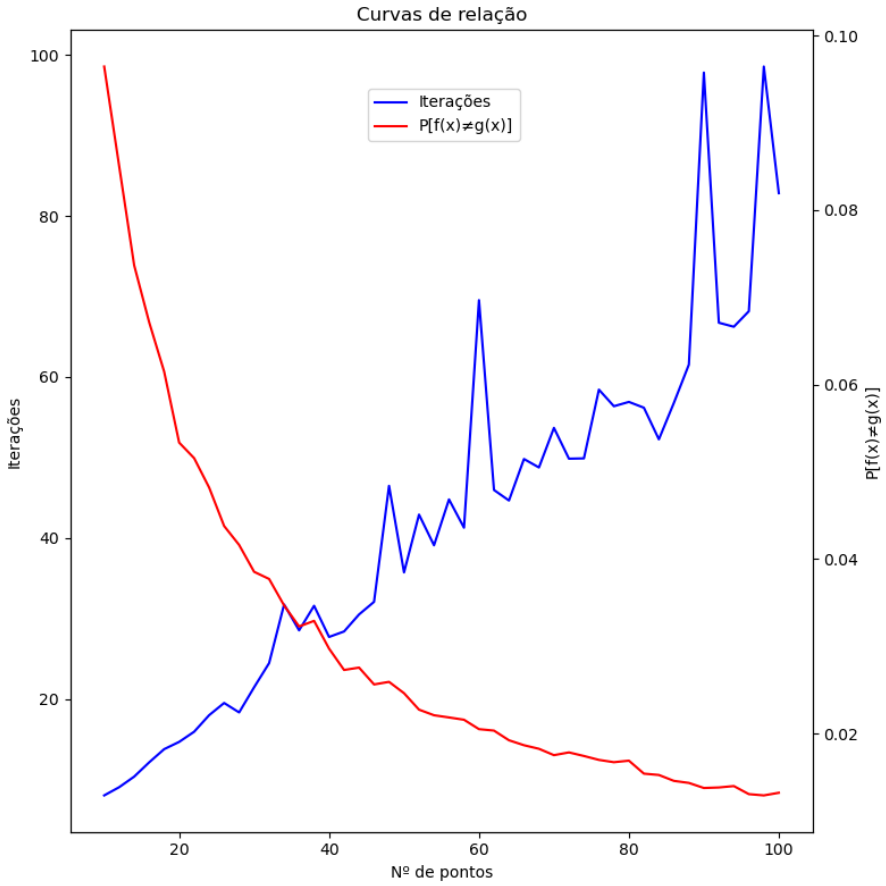
```
def relationship(lista_num_points):
    lista_iter_medio = list()
    lista_erro_medio = list()
    for num_points in lista_num_points:
        lista_iter = calc_num_iter(num_points, verbose=False)
        lista_erro = calc_p_erro(num_points, verbose=False)
        lista_iter_medio.append(np.mean(lista_iter))
        lista_erro_medio.append(np.mean(lista_erro))
    return lista_iter_medio, lista_erro_medio
```

Código 9: Plot da probabilidade de erro e do número de iterações

```
def plot_relationship(num_points_list, lista_iter_medio,
    lista_erro_medio):
    fig, ax = plt.subplots(1, 1, figsize=(8, 8))
    ax.plot(num_points_list,lista_iter_medio,c="blue",label="Itera
        ções")
    ax.set_title("Curvas de relação")
    ax.set_xlabel('Nº de pontos')
    ax.set_ylabel('Iterações')
    ax2=ax.twinx()
    ax2.plot(num_points_list,lista_erro_medio,c="red", label='P[f(
        x)\u2260g(x)]')
    ax2.set_ylabel('P[f(x)\u2260g(x)]')
    fig.legend(loc='upper center', bbox_to_anchor=(0.5, 0.9))
    fig.tight_layout()
    plt.show()
```

O resultado para o tamanho N da amostra variando de 10 a 100 com passo 2 pode ser observado na figura 2.

Figura 2: Resultado para N variando de 10 a 100 com passo 2



A quantidade de iterações para o PLA convergir, como constatado nos itens 1 e 3, oscila bastante, porém, pela figura, é possível observar que ela tem uma tendencia de alta conforme N aumenta. Já a probabilidade de erro $\mathbb{P}[f(x) \neq g(x)]$ visivelmente apresenta uma queda exponencial com o aumento de N. Conclui-se que, com o aumento de N, o algoritmo se torna mais confiável, porém demora mais para ser treinado.

2 Regressão Linear

Nestes problemas, nós vamos explorar como Regressão Linear pode ser usada em tarefas de classificação. Você usará o mesmo esquema de produção de pontos visto na parte acima do Perceptron, com $d = 2$, $\mathcal{X} = [-1, 1] \times [-1, 1]$, e assim por diante.

newline

Implementação:

Código 10: Classificador por Regressão Linear

```
# Classe para criar e treinar o classificador linear
class Linear():
    def __init__(self):
        self.w = np.zeros(3)  # inicializa os pesos (incluindo o w_0)

    # Método para calcular a matriz X
    def calc_matriz_X(self, data):
        n_samples = len(data)
        X = np.hstack([np.ones((n_samples, 1)), data]) # adiciona coluna de
            1s
        return X

    # Método para treinar o classificador linear
    def fit(self, data, labels):
        X = self.calc_matriz_X(data)
        y = labels
        X_T = np.transpose(X)
        X_pseudo_inv = np.dot(np.linalg.inv(np.dot(X_T, X)), X_T)
        self.w = np.dot(X_pseudo_inv, y)
        return self.w

    def classificar(self, data):
        X = self.calc_matriz_X(data)
        w_T = np.transpose(self.w)
        y_predicted = np.array([np.sign(np.dot(w_T, xn)) for xn in X])
        return y_predicted
```

1. Considere $N = 100$. Use Regressão Linear para encontrar g e calcule E_{in}, a fração de pontos dentro da amostra que foram classificados incorretamente (armazene os g's pois eles serão usados no item seguinte). Repita o experimento 1000 vezes. Qual dos valores abaixo é mais próximo do E_{in} médio?

   (a) 0
   (b) 0.001
   (c) 0.01
   (d) 0.1
   (e) 0.5

   Justificativa:

   Para responder a esse item foi implementada a seguinte função:

Código 11: Cálculo do E_in

```
        def calc_E_in(num_points, verbose = True):
            lista_E_in = list()
            lista_target = list()
            lista_linear = list()
            for _ in range(1000):
                # Criar a função target
                target = Target()
                a, b = target.generate_random_line()
                # Criar o dataset
                dataset = Dataset(num_points)
                data, labels = dataset.generate_dataset(target)
                # Criar e treinar o classificador linear
                linear = Linear()
                w = linear.fit(data,labels)
                # Classificar os pontos
                y_predicted = linear.classificar(data)
                # Calcular E_in para essa execução
                lista_E_in.append(np.mean(labels != y_predicted))
                # Guardar para saida
                lista_target.append(target)
                lista_linear.append(linear)
            E_in = np.mean(lista_E_in)
            if verbose: print(f"E_in = {E_in:.4f}")
            return E_in, lista_target, lista_linear
```

Foram realizadas 1000 execuções e em cada uma foi gerada uma nova função target, um novo dataset e foi treinado um novo regressor linear. O E_{in} em cada iteração foi calculado pelo número de erros dividido pela quantidade de pontos dentro da amostra. O valor final $E_{in} = 0.0420 = 4.20\%$ foi dado pela média dos E_{in} em cada iteração. Como 0.0420 está mais próximo de 0.01 do que de 0.1, o item (c) foi o escolhido.

2. Agora, gere 1000 pontos novos e use eles para estimar o *Eout* dos g's que você encontrou no item anterior. Novamente, realize 1000 execuções. Qual dos valores abaixo é mais próximo do E_{out} médio?

   (a) 0

   (b) 0.001

   (c) 0.01

   (d) 0.1

   (e) 0.5

   Justificativa:

   Para responder a esse item foi implementada a seguinte função:

Código 12: Cálculo do E_out

```
        def calc_E_out(num_points, lista_target, lista_linear, verbose =
            True):
```

```
        lista_E_out = list()
        for target, linear in zip(lista_target, lista_linear):
            # Criar o dataset com a mesma função target do E_in
            dataset = Dataset(num_points)
            data, labels = dataset.generate_dataset(target)
            # Classificar os pontos com a mesma hipótese do E_in
            y_predicted = linear.classificar(data)
            # Calcular E_out para essa execução
            lista_E_out.append(np.mean(labels != y_predicted))
        E_out = np.mean(lista_E_out)
        if verbose: print(f"E_out = {E_out:.4f}")
        return E_out
```

Como o enunciado pede explicitamente que sejam utilizados os mesmos g's do item anterior, a função no código 12 recebe como entrada os targets e as hipóteses calculadas no item anterior (e que por isso foram colocadas como saída na função apresentada no código 11). Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset com 1000 pontos utilizando um dos targets gerados no item anterior e classificados pela hipótese treinada no item anterior para o mesmo target. O E_{out} em cada iteração foi calculado pelo número de erros dividido pela quantidade de pontos no dataset. O valor final $E_{out} = 0.0489 = 4.89\%$ foi dado pela média dos E_{out} em cada iteração. Como 0.0489 está mais próximo de 0.01 do que de 0.1, o item (c) foi o escolhido.

3. Agora, considere $N = 10$. Depois de encontrar os pesos usando Regressão Linear, use-os como um vetor de pesos iniciais para o Algoritmo de Aprendizagem Perceptron (PLA). Execute o PLA até que ele convirja num vetor final de pesos que separa perfeitamente os pontos dentro-de-amostra. Dentre as opções abaixo, qual é mais próxima do número médio de iterações (sobre 1000 execuções) que o PLA demora para convergir?

   (a) 1
   (b) 15
   (c) 300
   (d) 5000
   (e) 10000

   Justificativa:

   Para responder a esse item foi implementada a seguinte função:

Código 13: Cálculo do número de iterações do PLA

```
    def calc_PLA_iter(num_points):
        lista_iter = list()
        for _ in range(1000):
            # Criar a função target
            target = Target()
            target.generate_random_line()
            # Criar o dataset
            dataset = Dataset(num_points)
```

```
                data, labels = dataset.generate_dataset(target)
                # Criar e treinar o classificador linear
                linear = Linear()
                w = linear.fit(data,labels)
                # Criar e treinar o perceptron
                perceptron = Perceptron2D(weights=w)
                iter, _ = perceptron.pla(data,labels)
                lista_iter.append(iter)
            print(f"{np.mean(lista_iter)} iterações com desvio padrão {np.
                std(lista_iter):.4f} (min:{np.min(lista_iter)}, máx:{np.max
                (lista_iter)})")
```

Foram realizadas 1000 execuções e em cada uma foi gerada uma nova função target e um novo dataset com 10 pontos. Em cada iteração é treinado um Classificador Linear e seus pesos são utilizados como pesos iniciais para treinar um Perceptron 2D. O número de iterações internas realizadas no método $perceptron.pla(data, labels)$, que treina o Perceptron utilizando o PLA, é armazenado em uma lista e no final das execuções é calculada a média e o desvio padrão desses valores. O resultado após 1000 execuções do experimento foi uma média de $3.1460(\approx 3)$ iterações, com desvio padrão de $7.8247(\approx 8)$ iterações, mínimo de 1 iteração e máximo de 104 iterações. Ainda é possível observar uma grande variação do número de iterações entre cada execução, porém a média é menor do que a calculada no item 1 do problema 1 (≈ 5). É possível constatar então que o PLA converge mais rápido quando seus pesos são inicializados por uma Regressão Linear. Como 3 está mais próximo de 1 do que de 15, o **item a** foi selecionado.

4. Vamos agora avaliar o desempenho da versão pocket do PLA em um conjunto de dados que não é linearmente separável. Para criar este conjunto, gere uma base de treinamento com N_1 pontos como foi feito até agora, mas selecione aleatoriamente 10% dos pontos e inverta seus rótulos. Em seguida, implemente a versão pocket do PLA, treine-a neste conjunto não-linearmente separável, e avalie seu E_{out} numa nova base de N_2 pontos na qual você não aplicará nenhuma inversão de rótulos. Repita para 1000 execuções, e mostre o E_{in} e E_{out} médios para as seguintes configurações (não esqueça dos gráficos scatterplot, como anteriormente):

   Implementação:

   Para responder os subitens abaixo, foi adicionado à classe Perceptron2D (código 3), utilizada no problema 1, um método para fazer o treinamento utilizando o algoritmo Pocket (código 14) e foi implementada também uma função para calcular o E_{in} e E_{out} médio (código 15) em 1000 execuções de treinamento do Perceptron utilizando o algoritmo Pocket. A função permite escolher se irá ou não utilizar os pesos de um classificador linear como pesos iniciais do perceptron e também plota o dataset com a Função Target e a Hipótese para a última execução.

Código 14: Algoritmo Pocket

```
    # Método para treinar o perceptron usando o algoritmo Pocket
    def pocket(self, data, labels, max_iter = 1000):
        n_samples = len(data)
        X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona
            uma coluna de 1s para o X_0 (coordenada artificial)
```

```
            iterations = 0
            E_in = 1
            w = self.w
            while E_in > 0 and iterations < max_iter:
                E_in_atual = 0
                w_atual = w
                errors = 0
                for i in range(n_samples):
                    if labels[i] * np.dot(self.w, X_bias[i]) <= 0:
                        w_atual += labels[i] * X_bias[i] # atualiza os
                            pesos
                        errors += 1
                iterations += 1
                E_in_atual = errors/n_samples
                if E_in_atual < E_in: # atualiza os pesos e o E_in apenas
                    se o E_in for menor que o anterior
                    E_in = E_in_atual
                    w = w_atual
            self.w = w
            return iterations, self.w, E_in
        setattr(Perceptron2D, "pocket", pocket)
```

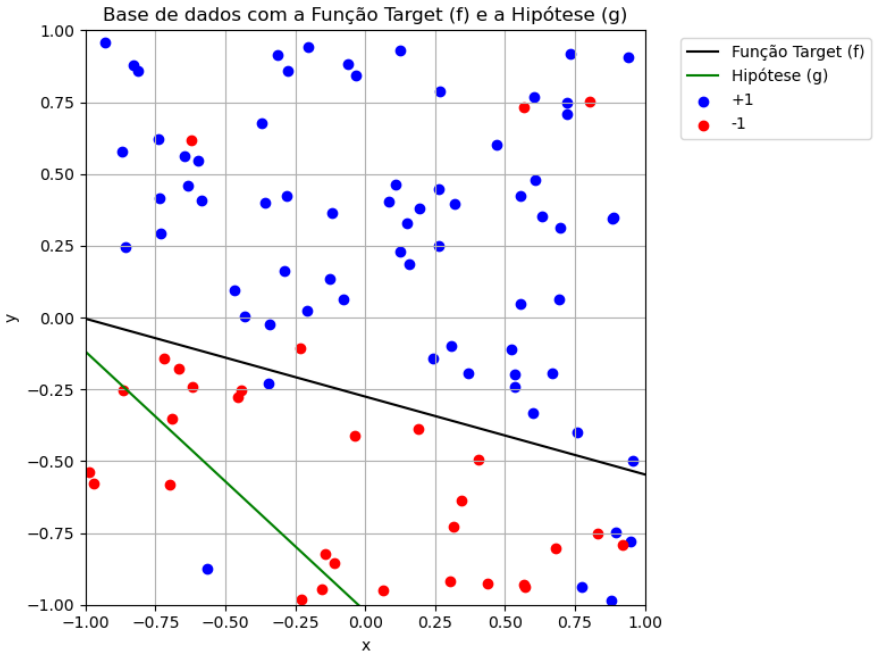
Código 15: Cálculo do E_in e do E_out

```
        def calc_pocket_E(N1, N2, i, LinReg = False):
            lista_E_in = list()
            lista_E_out = list()
            for _ in range(1000):
                # Criar a função target
                target = Target()
                a, b = target.generate_random_line()
                # Criar o dataset de treinamento
                dataset_train = Dataset(N1)
                x_train, y_train = dataset_train.generate_dataset(target)
                selected_indices = np.random.choice(len(y_train), int(len(
                    y_train) * 0.1), replace=False) # seleciona 10%
                y_train[selected_indices] *= -1 # inverte o valor de 10%
                if LinReg: # pesos inicializados com regressão linear
                    # Criar e treinar o classificador linear
                    linear = Linear()
                    w = linear.fit(x_train,y_train)
                    # Criar e treinar o perceptron com pocket
                    perceptron = Perceptron2D(weights=w)
                    _, _, E_in = perceptron.pocket(x_train,y_train,
                        max_iter = i)
                    lista_E_in.append(E_in)
                else: # pesos inicializados com 0
                    # Criar e treinar o perceptron com pocket
                    perceptron = Perceptron2D()
                    _, _, E_in = perceptron.pocket(x_train,y_train,
                        max_iter = i)
                    lista_E_in.append(E_in)
                # Criar o dataset de teste
                dataset_test = Dataset(N2)
```

```
            x_test, y_test = dataset_test.generate_dataset(target)
            # Classificar os pontos com a mesma hipotese do E_in
            y_predicted = perceptron.classificar(x_test)
            # Calcular E_out para essa execução
            E_out = np.mean(y_test != y_predicted)
            lista_E_out.append(E_out)
        # Printar E_in e E_out médios
        print(f"E_in = {np.mean(E_in):.4f}")
        print(f"E_out = {np.mean(E_out):.4f}")
        # Plotar o dataset, a função target e a hipótese g da última
            execução
        scatterplot(x_train, y_train, target, perceptron)
```

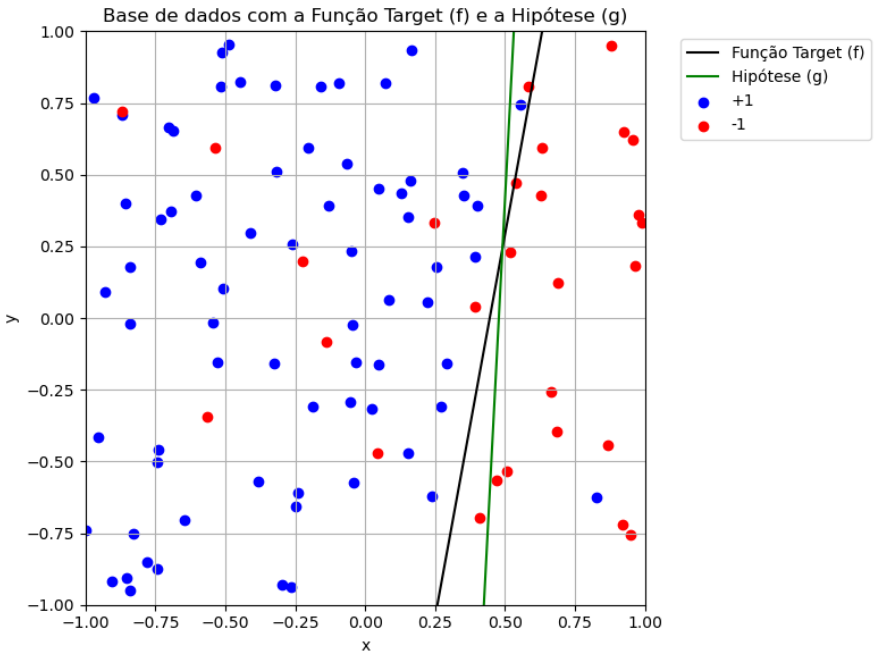
(a) Inicializando os pesos com 0; $i = 10$; $N_1 = 100$; $N_2 = 1000$.
Resposta: $E_{in} = 0.2200$ e $E_{out} = 0.2540$.

Figura 3: Scatterplot inicializando os pesos com 0 e $i = 10$



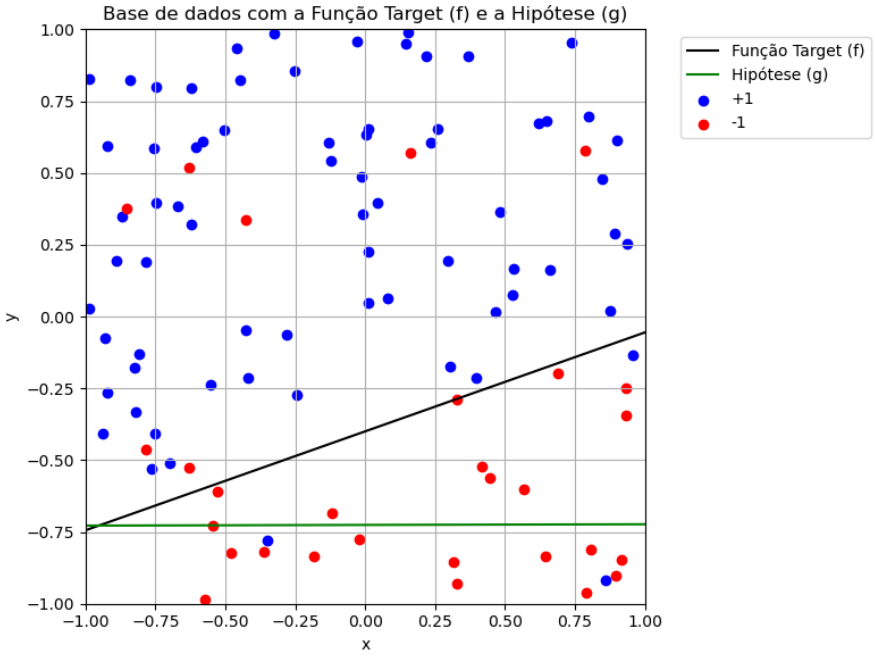
(b) Inicializando os pesos com 0; $i = 50$; $N_1 = 100$; $N_2 = 1000$.
Resposta: $E_{in} = 0.2400$ e $E_{out} = 0.0350$.

Figura 4: Scatterplot inicializando os pesos com 0 e $i = 50$



(c) Inicializando os pesos com Regressão Linear; $i = 10$; $N_1 = 100$; $N_2 = 1000$.
Resposta: $E_{in} = 0.2200$ e $E_{out} = 0.01400$.

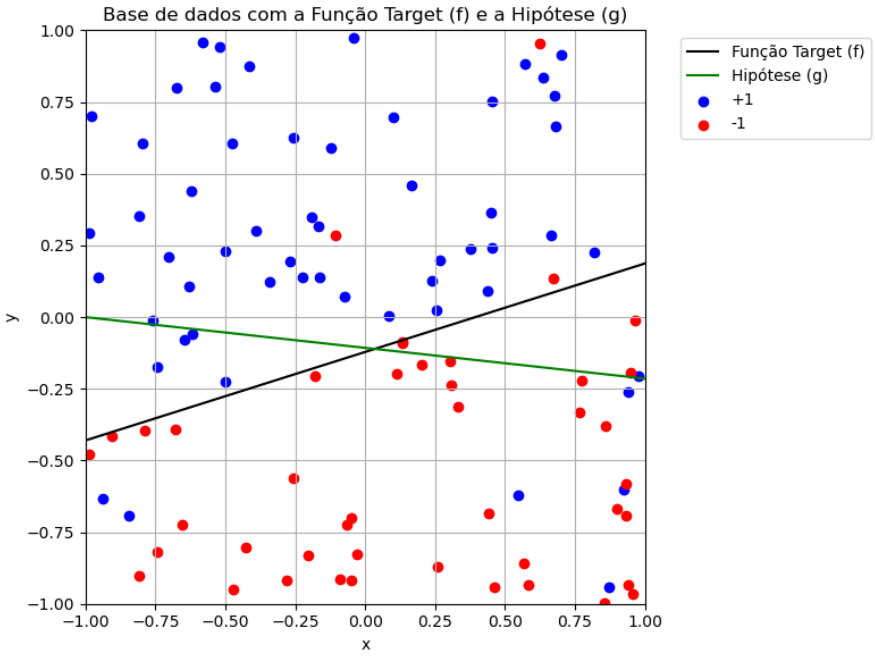
Figura 5: Scatterplot inicializando os pesos com Regressão Linear e $i = 10$



(d) Inicializando os pesos com Regressão Linear; $i = 50$; $N_1 = 100$; $N_2 = 1000$.

Resposta: $E_{in} = 0.1800$ e $E_{out} = 0.0880$.

Figura 6: Scatterplot inicializando os pesos com Regressão Linear e $i = 50$



3 Regressão Não-Linear

Nestes problemas, nós vamos novamente aplicar Regressão Linear para classificação. Considere a função target

$$f(x_1, x_2) = sign(x_1^2 + x_2^2 - 0.6)$$

Gere um conjunto de treinamento de $N = 1000$ pontos em $\mathcal{X} = [-1, 1] \times [-1, 1]$ com probabilidade uniforme escolhendo cada $x \in \mathcal{X}$. Gere um ruído simulado selecionando aleatoriamente 10% do conjunto de treinamento e invertendo o rótulo dos pontos selecionados.

newline

Implementação:

Para responder os itens referentes a este problema, foram implementado em Python duas classes: uma para gerar a função target não-linear (código 16) e outra para gerar o Classificador Não-Linear (código 17).

Código 16: Target Não-Linear

```
# Classe para criar a função target não linear
class TargetNaoLinear:

    # Método para classificar pontos de acordo a função target
    def classify_point(self, point):
        return np.sign(point[0]**2 + point[1]**2 - 0.6)

    # Método para criar uma elipse para plots
    def criar_elipse(self):
        r = np.sqrt(0.6)
        elipse =  Ellipse(xy=(0,0), width=2*r, height=2*r, angle=np.degrees
            (0),
                          edgecolor='k', fc='None', lw=2, label='Função
                             Target (f)')
        return elipse
```

A classe para gerar a função target não-linear possui dois métodos: um para classificar os pontos de acordo com $f(x_1, x_2) = sign(x_1^2 + x_2^2 - 0.6)$ e outro que gera uma elipse a partir $f(x_1, x_2)$ para ser plotada. Esta classe poderia ter sido quebrada em duas funções, porém foi decidido criar uma classe para agrupar as duas funções e também para que ela continuasse funcionando como entrada para o método $generate\_dataset(target)$ da classe $Dataset$ (código 2), reaproveitada neste problema para gerar a base de dados $\mathcal{X}$.

Código 17: Classificador por Regressão Não-Linear

```
# Classe para criar e treinar o classificador linear
class NaoLinear(Linear, w = np.zeros(6)):
    def __init__(self):
        self.w = w  # inicializa os pesos (incluindo o w_0)

    # Modifica o método para calcular a matriz X
    def calc_matriz_X(self, data):
        X = list()
        for point in data:
            x1 = point[0]
```

```
                x2 = point[1]
                X.append([1, x1, x2, x1*x2, x1**2, x2**2])
            return np.array(X)

        # Método novo para criar uma elipse para plots
        def criar_elipse(self):
            # Coeficientes da equação geral da cônica (Ax1^2 + Bx1x2 + Cx2^2 +
               Dx1 + Ex2 + F=0)
            # relacionados com os pesos (1, x1, x2, x1x2, x1^2, x2^2)
            A = self.w[4] # coef de x1^2
            B = self.w[3] # coef de x1x2
            C = self.w[5] # coef de x2^2
            D = self.w[1] # coef de x1
            E = self.w[2] # coef de x2
            F = self.w[0] # termo independente
            # Matrizes associadas à equação geral da cônica
            M = np.array([[A, B / 2], [B / 2, C]])
            offset = np.array([D, E])
            # Calcular o centro da elipse
            center = np.linalg.solve(-2 * M, offset)
            # Calcular os semi-eixos e o ângulo de rotação
            eigenvalues, eigenvectors = np.linalg.eigh(M)
            order = np.argsort(eigenvalues)
            eigenvalues = eigenvalues[order]
            eigenvectors = eigenvectors[:, order]
            semi_major = np.sqrt(-F / (eigenvalues[0] * eigenvalues[1])) / np.
               sqrt(eigenvalues[0])
            semi_minor = np.sqrt(-F / (eigenvalues[0] * eigenvalues[1])) / np.
               sqrt(eigenvalues[1])
            angle = np.arctan2(eigenvectors[1, 0], eigenvectors[0, 0])
            # Criar a elipse
            elipse =  Ellipse(xy=center, width=2 * semi_major, height=2 *
               semi_minor, angle=np.degrees(angle),
                              edgecolor='g', fc='None', lw=2, label='Hipótese (g)
                                 ')
            return elipse
```

A classe do Classificador Não-Linear herda a classe do Classificador Linear (código 10), modifica a inicialização dos pesos e o método para gerar a matriz de entradas X seguindo o vetor de atributos não-linear $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$ e insere um novo método que gera uma elipse a partir pesos calculados (relacionando eles com a fórmula geral das cônicas) para ser plotada.

Para testar as classes foram criadas duas funções: uma para plotar o dataset não-linear juntamente com a função target e a hipotese (código 18), e outra parar gerar os dados utilizando a função target não-linear e ainda gerando ruido em 10% deles e depois treinar um Classificador Não-Linear (código 19).

Código 18: Scatterplot Não-Linear

```
    def scatterplot_nao_linear(data, labels, target, hipotese):
        fig, ax = plt.subplots(subplot_kw={'aspect': 'equal'}, figsize=(8, 6))
        # plotar a função target
        elipse_target = target.criar_elipse()
        ax.add_patch(elipse_target)
        # plotar a hipótese
```

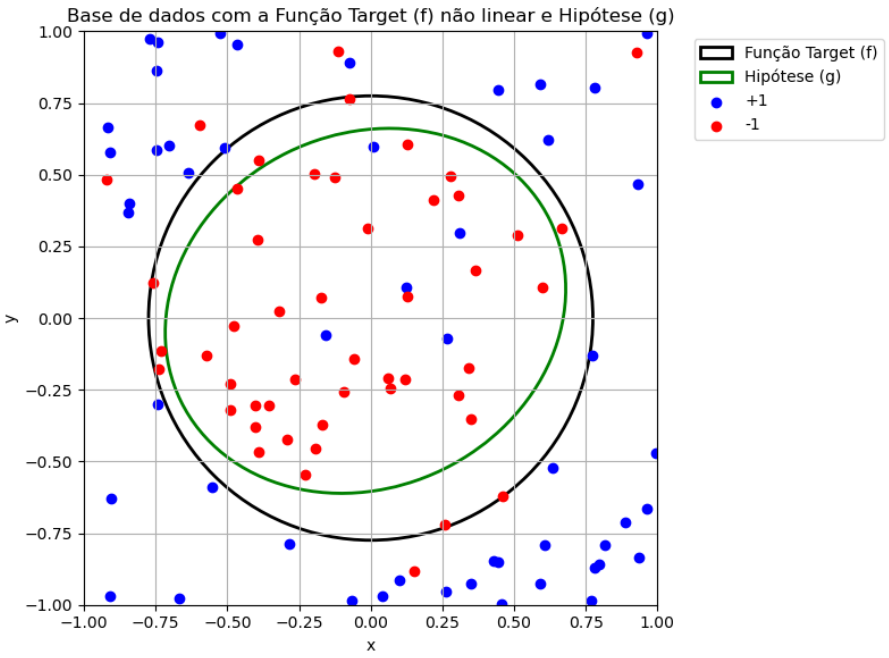
```
        if type(hipotese) is NaoLinear:
            elipse_hipotese = hipotese.criar_elipse()
            ax.add_patch(elipse_hipotese)
        elif type(hipotese) is Linear:
            w = hipotese.w
            x = np.linspace(-1, 1, 100)
            y_g = -(w[1] * x + w[0]) / w[2]
            plt.plot(x, y_g, 'g-', label='Hipótese (g)')
        else:
            return print("Hipotese não suportada")
        # plotar os pontos
        x_pos = [data[i][0] for i in range(len(data)) if labels[i] == 1]
        y_pos = [data[i][1] for i in range(len(data)) if labels[i] == 1]
        x_neg = [data[i][0] for i in range(len(data)) if labels[i] == -1]
        y_neg = [data[i][1] for i in range(len(data)) if labels[i] == -1]
        plt.scatter(x_pos, y_pos, c='blue', label='+1')
        plt.scatter(x_neg, y_neg, c='red', label='-1')
        # ajustar a figura
        plt.xlim(-1, 1)
        plt.ylim(-1, 1)
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title('Base de dados com a Função Target (f) não linear e Hipótese
            (g)')
        plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
        plt.tight_layout(rect=[0, 0, 1, 1])
        plt.grid(True)
        plt.show()
```

Código 19: Teste para a Regressão Não-Linear

```
    def teste(num_points, hipotese):
        # Criar a target não-linear
        target = TargetNaoLinear()
        # Criar o dataset
        dataset = Dataset(num_points)
        data, labels = dataset.generate_dataset(target)
        # Adicionar ruido
        selected_indices = np.random.choice(len(labels), int(len(labels) * 0.1)
            , replace=False) # seleciona 10%
        labels[selected_indices] *= -1 # inverte o valor de 10%
        # Criar o classificador
        hipotese.fit(data, labels)
        # Plotar
        scatterplot_nao_linear(data, labels, target, hipotese)
```

O resultado da função $teste(num\_points, hipotese)$ para uma hipótese gerada pelo treinamento de uma Regressão Não-Linear pode ser observado na figura 7.

Figura 7: Base de dados com a Função Target (f) Não-Linear e a Hipótese (g) Não-Linear



1. Execute a Regressão Linear sem nenhuma transformação, usando o vetor de atributos $(1, x_1, x_2)$ para encontrar o peso $\boldsymbol{w}$. Qual é o valor aproximado de classificação do erro médio dentro da amostra E_{in} (medido ao longo de 1000 execuções)?

   (a) 0
   (b) 0.1
   (c) 0.3
   (d) 0.5
   (e) 0.8

Justificativa:

Para responder a esse item foi implementada a seguinte função:

Código 20: Cálculo do E_in para a Regressão Linear

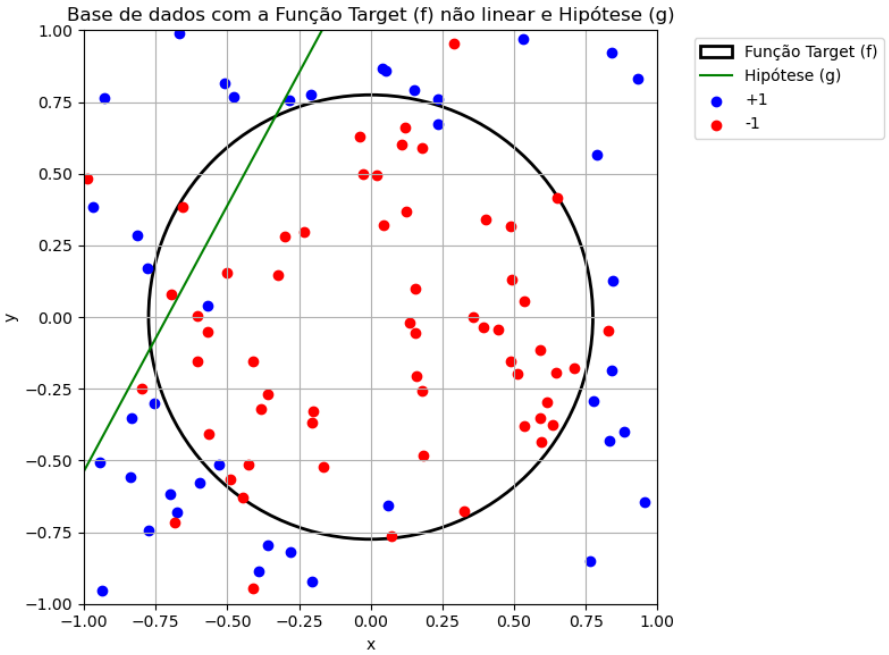
```
        def calc_Ein_linear(num_points, verbose = True):
            lista_E_in = list()
            for _ in range(1000):
                # Criar a target não-linear
                target = TargetNaoLinear()
                # Criar o dataset
                dataset = Dataset(num_points)
                data, labels = dataset.generate_dataset(target)
                # Adicionar ruido
                selected_indices = np.random.choice(len(labels), int(len(
                    labels) * 0.1), replace=False) # seleciona 10%
```

```
                labels[selected_indices] *= -1 # inverte o valor de 10%
                # Criar o classificador não-linear
                linear = Linear()
                linear.fit(data, labels)
                # Classificar os pontos
                y_predicted = linear.classificar(data)
                # Calcular E_in para essa execução
                lista_E_in.append(np.mean(labels != y_predicted))
            E_in = np.mean(lista_E_in)
            # Plotar a última execução
            if verbose: print(f"E_in = {E_in:.4f}")
            return E_in
```

Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset para a função target $f(x_1, x_2) = sign(x_1^2 + x_2^2 - 0.6)$ e foi treinado um novo Classificador Linear. O E_{in} em cada iteração foi calculado pelo número de erros dividido pela quantidade de pontos dentro da amostra. O valor final $E_{in} = 0.5044 = 50.44\%$ foi dado pela média dos E_{in} em cada iteração. O valor de E_{in} foi consideravelmente alto, como esperado de um Classificador Linear tentando classificar um dataset que não é linear. Como 0.5044 está mais próximo de 0.5 do que de 0.8, o item (d) foi o escolhido.

A figura 8 (gerada para apenas 100 pontos) ilustra por que o resultado do Regressor Linear foi tão ruim.

Figura 8: Base de dados com a Função Target (f) Não-Linear e a Hipótese (g) Linear



2. Agora, transforme os $N = 1000$ dados de treinamento seguindo o vetor de atributos não-linear $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$. Encontre o vetor $\tilde{\boldsymbol{w}}$ que corresponda à solução da Regressão

Linear. Quais das hipóteses a seguir é a mais próxima à que você encontrou? Avalie o resultado médio obtido após 1000 execuções.

(a) $g(x_1, x_2) = sign(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 1.5x_2^2)$

(b) $g(x_1, x_2) = sign(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 15x_2^2)$

(c) $g(x_1, x_2) = sign(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 15x_1^2 + 1.5x_2^2)$

(d) $g(x_1, x_2) = sign(-1 - 1.5x_1 + 0.08x_2 + 0.13x_1x_2 + 0.05x_1^2 + 0.05x_2^2)$

(e) $g(x_1, x_2) = sign(-1 - 0.05x_1 + 0.08x_2 + 1.5x_1x_2 + 0.15x_1^2 + 0.15x_2^2)$

Justificativa:

Para responder a esse item foi implementada a seguinte função:

Código 21: Cálculo dos pesos para a Regressão Não-Linear

```
        def calc_w_naolinear(num_points, verbose = True):
            lista_w = list()
            for _ in range(1000):
                # Criar a target não-linear
                target = TargetNaoLinear()
                # Criar o dataset
                dataset = Dataset(num_points)
                data, labels = dataset.generate_dataset(target)
                # Adicionar ruido
                selected_indices = np.random.choice(len(labels), int(len(
                    labels) * 0.1), replace=False) # seleciona 10%
                labels[selected_indices] *= -1 # inverte o valor de 10%
                # Criar o classificador não-linear
                naolinear = NaoLinear()
                w = naolinear.fit(data, labels)
                # Armazenar os pesos
                lista_w.append(w)
            w = np.mean(lista_w, axis=0)
            if verbose:
                np.set_printoptions(suppress=True)
                print(f"w = {w}")
            return w
```

Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset para a função target $f(x_1, x_2) = sign(x_1^2 + x_2^2 - 0.6)$ e foi treinado um novo Classificador Não-Linear. O vetor $\boldsymbol{w}$ em cada iteração foi calculado pelo método fit que multiplica a pseudo-inversa da matriz X pelo vetor de labels. O valor final do vetor

$$\tilde{\boldsymbol{w}} = [-0.9894, \quad -0.00164, \quad 0.0009, \quad 0.0031, \quad 1.5505, \quad 1.556]$$

foi dado pela média em cada coluna da matriz $lista\_w$, a qual cada linha é composta pelo $\boldsymbol{w}$ calculado pelo treinamento naquela execução. O segundo, terceiro e quarto elementos divergem bastante das alternativas apresentadas, mas como o primeiro, o quinto e o sexto elementos se aproximam dos coeficientes da primeira alternativa, o item (a) foi o escolhido.

3. Qual o valor mais próximo do erro de classificação fora da amostra E_{out} de sua hipótese na questão anterior? (Estime-o gerando um novo conjunto de 1000 pontos e usando 1000 execuções diferentes, como antes).

   (a) 0

   (b) 0.1 *

   (c) 0.3

   (d) 0.5

   (e) 0.8

   Para responder a esse item foi implementada a seguinte função:

Código 22: Cálculo do E_out para a Regressão Não-Linear

```
        def calc_Eout_naolinear(num_points, w, verbose = True):
            lista_E_out = list()
            for _ in range(1000):
                # Criar a target não-linear
                target = TargetNaoLinear()
                # Criar o dataset de teste
                dataset = Dataset(num_points)
                data, labels = dataset.generate_dataset(target)
                # Criar o classificador não-linear
                naolinear = NaoLinear(w = w) # força a usar os pesos
                    anteriores
                # Classificar os pontos com a mesma hipotese do E_in
                y_predicted = naolinear.classificar(data)
                # Calcular E_out para essa execução
                lista_E_out.append(np.mean(labels != y_predicted))
            E_out = np.mean(lista_E_out)
            if verbose: print(f"E_out = {E_out:.4f}")
            return E_out
```

Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset para a função target $f(x_1, x_2) = sign(x_1^2 + x_2^2 - 0.6)$ e Classificador Linear foi gerado reutilizando o $\tilde{\boldsymbol{w}}$ calculado no item anterior. O E_{out} em cada iteração foi calculado pelo número de erros dividido pela quantidade de pontos dentro da amostra. O valor final $E_{out} = 0.0291 = 2.91\%$ foi dado pela média dos E_{out} em cada iteração. Como 0.0291 está mais próximo de 0 do que de 0.1, o item (a) foi o escolhido.