

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

21 de maio 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  $\text{sign}(0) = 0$ , 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
```

```

14         self.b = b
15         return a, b
16
17     # Método para classificar pontos de acordo a função target
18     def classify_point(self, point):
19         a = self.a
20         b = self.b
21         y_reta = a*point[0] + b
22         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 $D$

```

1     # Classe para criar o dataset
2     class Dataset:
3         def __init__(self, N):
4             self.N = N # tamanho do dataset
5
6         # Método para gerar a base de dados D
7         def generate_dataset(self, target):
8             N = self.N
9             data = np.random.uniform(-1, 1, (N, 2)) # gera N pontos no R2 com
                coordenadas entre [-1, 1]
10            labels = np.array([target.classify_point(point) for point in data])
11            return data, labels

```

### Código 3: Perceptron

```

1     # Classe para criar e treinar o perceptron 2D
2     class Perceptron2D:
3         def __init__(self, weights = np.zeros(3)):
4             self.w = weights # inicializa os pesos (incluindo o w_0)
5
6         # Método para treinar o perceptron usando o algoritmo de aprendizagem
            perceptron (PLA)
7         def pla(self, data, labels):
8             n_samples = len(data)
9             X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma
                coluna de 1s para o X_0 (coordenada artificial)
10            iterations = 0
11            errors = 1
12            while errors > 0:
13                errors = 0
14                for i in range(n_samples):
15                    if labels[i] * np.dot(self.w, X_bias[i]) <= 0:
16                        self.w += labels[i] * X_bias[i] # atualiza os pesos
17                        errors += 1
18                iterations += 1
19            return iterations, self.w
20
21        # Método para classificar um dataset com base nos pesos aprendidos.
22        def classificar(self, data):
23            n_samples = len(data)
24            X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma
                coluna de 1s para o bias X_0

```

```

25         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)

```

1  def scatterplot(data, labels, target, hipotese):
2      a, b = target.a, target.b
3      w = hipotese.w
4      # Plotar resultados
5      plt.figure(figsize=(8, 6))
6      x_pos = [data[i][0] for i in range(len(data)) if labels[i] == 1]
7      y_pos = [data[i][1] for i in range(len(data)) if labels[i] == 1]
8      x_neg = [data[i][0] for i in range(len(data)) if labels[i] == -1]
9      y_neg = [data[i][1] for i in range(len(data)) if labels[i] == -1]
10     plt.scatter(x_pos, y_pos, c='blue', label='+1')
11     plt.scatter(x_neg, y_neg, c='red', label='-1')
12     x = np.linspace(-1, 1, 100)
13     y_target = a*x+b
14     y_g = -(w[1] * x + w[0]) / w[2]
15     plt.plot(x, y_g, 'g-', label='Hipótese (g)')
16     plt.plot(x, y_target, 'k-', label='Função Target (f)')
17     plt.xlim(-1, 1)
18     plt.ylim(-1, 1)
19     plt.xlabel('x')
20     plt.ylabel('y')
21     plt.title('Base de dados com a Função Target (f) e a Hipótese (g)')
22     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
23     plt.tight_layout(rect=[0, 0, 1, 1])
24     plt.grid(True)
25     plt.show()

```

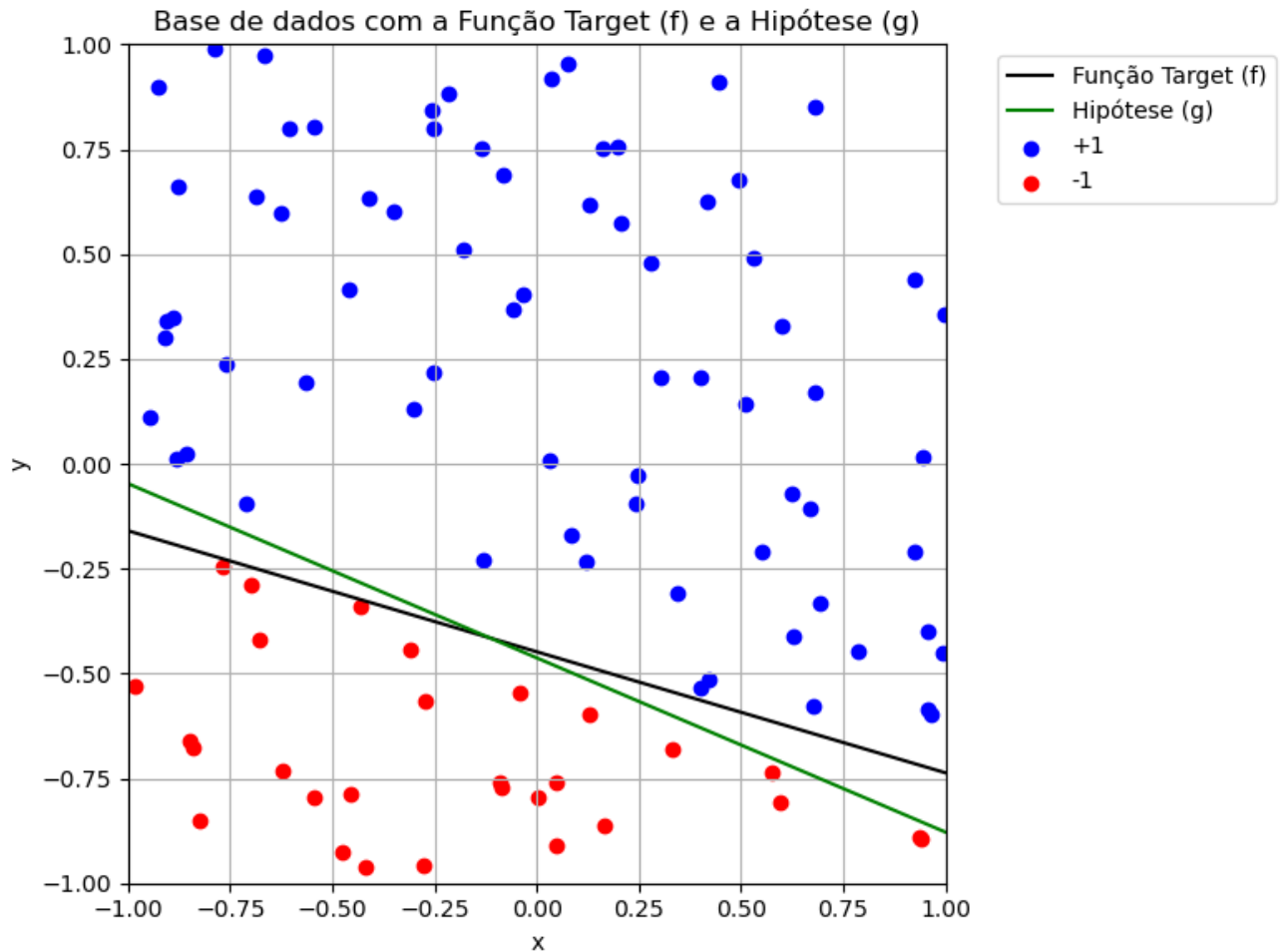
**Código 5:** Teste das classes

```

1  def teste():
2      # Criar a função target
3      target = Target()
4      a, b = target.generate_random_line()
5      # Criar o dataset e a função target
6      num_points = 100
7      dataset = Dataset(num_points)
8      data, labels = dataset.generate_dataset(target)
9      # Criar e treinar o perceptron
10     perceptron = Perceptron2D()
11     _, w = perceptron.pla(data, labels)
12     # Plotar resultados
13     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 amostra  $E_{in}$  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)
8         perceptron = Perceptron2D()
9         iter, _ = perceptron.pla(data, labels)
10        lista_iter.append(iter)
11    if verbose: print(f"{np.mean(lista_iter)} iterações com desvio
12                  padrão {np.std(lista_iter):.4f} (min:{np.min(lista_iter)}, máx
                  :{np.max(lista_iter)})")
13    return lista_iter
```

O resultado após 1000 execuções do experimento, com a variável  $num\_points = 10$ , foi uma média de 5.0490 ( $\approx 5$ ) iterações, com desvio padrão de 7.7770 ( $\approx 8$ ) iterações, mínimo de 2 iterações e máximo de 99 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, podendo levar a configurações de diferentes níveis de dificuldade para o PLA convergir. Como 5 está mais próximo de 1 do que de 15, o **item a** 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 quantidade 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 percentual de erro, isto é, a quantidade de vezes que a classificação do perceptron foi diferente da função target dividido 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

```

1  def calc_p_erro(num_points, verbose = True):
2      lista_erro = list()
3      for _ in range(1000):
4          target = Target()
5          target.generate_random_line()
6          dataset_train = Dataset(num_points)
7          x_train, y_train = dataset_train.generate_dataset(target)
8          dataset_test = Dataset(10000) # mais 10mil pontos
9          x_test, y_test = dataset_test.generate_dataset(target)
10         perceptron = Perceptron2D()
11         perceptron.pla(x_train, y_train)
12         y_predicted = perceptron.classificar(x_test)
13         erro = np.mean(y_test != y_predicted)
14         lista_erro.append(erro)
15         if verbose: print(f"P[f(x) \u2260 g(x)] = {np.mean(lista_erro):.4f}")
16     return lista_erro

```

O resultado após 1000 execuções, com  $num\_points = 10010$  e  $train\_size = 10$ , foi  $\mathbb{P}[f(x) \neq g(x)] = 0.0671 = 6.71\%$ . Como 0.0671 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 32.981 ( $\approx 33$ ) iterações, com desvio padrão de 164.5924 ( $\approx 165$ ) iterações, mínimo de 2 iterações e máximo de 4149 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 a** foi selecionado.

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

- (a) 0.001
- (c) 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  $\mathbb{P}[f(x) \neq g(x)] = 0.0069 = 0.69\%$ . Como 0.0069 está mais próximo de 0.01 do que de 0.001, 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

```
1 def relationship(lista_num_points):
2     lista_iter_medio = list()
3     lista_erro_medio = list()
4     for num_points in lista_num_points:
5         lista_iter = calc_num_iter(num_points, verbose=False)
6         lista_erro = calc_p_erro(num_points, verbose=False)
7         lista_iter_medio.append(np.mean(lista_iter))
8         lista_erro_medio.append(np.mean(lista_erro))
9     return lista_iter_medio, lista_erro_medio
```

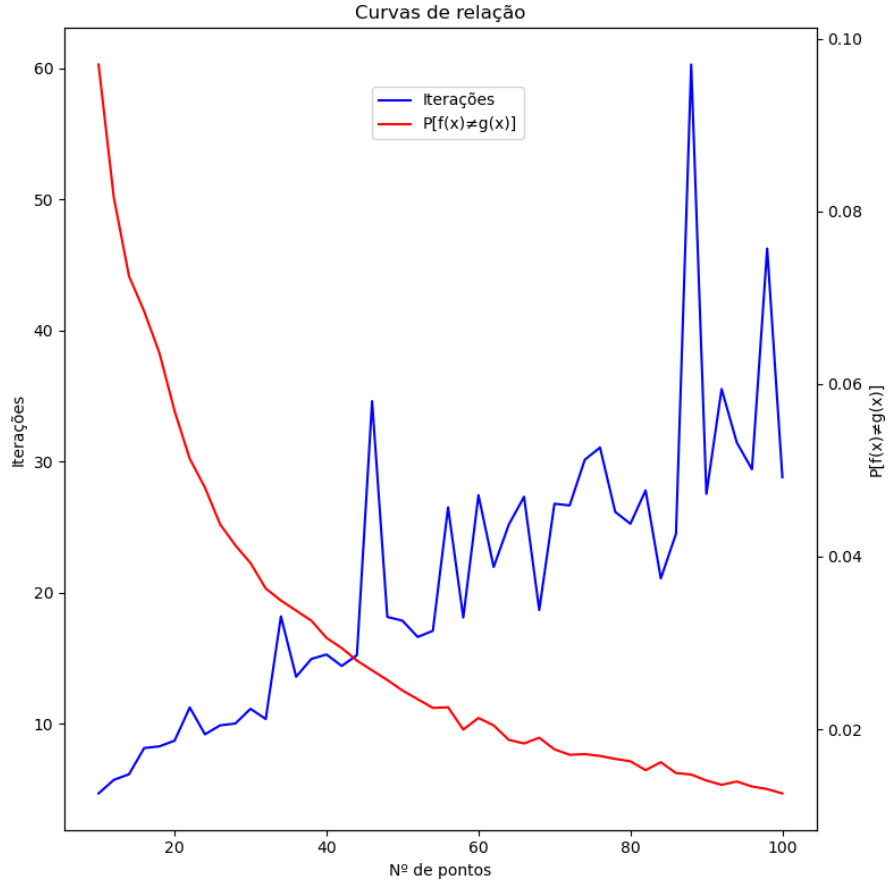
**Código 9:** Plot da probabilidade de erro e do número de iterações

```
1 def plot_relationship(num_points_list, lista_iter_medio,
2     lista_erro_medio):
3     fig, ax = plt.subplots(1, 1, figsize=(8, 8))
4     ax.plot(num_points_list, lista_iter_medio, c="blue", label="Itera
5     ções")
6     ax.set_title("Curvas de relação")
7     ax.set_xlabel('Nº de pontos')
8     ax.set_ylabel('Iterações')
9     ax2=ax.twinx()
10    ax2.plot(num_points_list, lista_erro_medio, c="red", label='P[f(
11    x)\u2260g(x)]')
12    ax2.set_ylabel('P[f(x)\u2260g(x)]')
13    fig.legend(loc='upper center', bbox_to_anchor=(0.5, 0.9))
14    fig.tight_layout()
15    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:

Para responder os itens referentes a este problema, foi implementado em Python uma classe para gerar um Classificador Linear (código 10), utilizando as fórmulas e procedimentos apresentados na terceira aula do professor Yaser Abu-Mostafa. A classe possui três métodos: um para calcular a matriz de entradas  $X$ , outro para treinar o classificador e a terceira para classificar pontos utilizando o mesmo. Para gerar a função target e gerar o dataset foram reaproveitadas as mesmas classes apresentadas nos códigos 1 e 2.

**Código 10:** Classificador por Regressão Linear

```
1  # Classe para criar e treinar o classificador linear
2  class Linear():
3      def __init__(self):
4          self.w = np.zeros(3)  # inicializa os pesos (incluindo o w_0)
5
6      # Método para calcular a matriz X
7      def calc_matriz_X(self, data):
8          N = 5
9          n_samples = len(data)
10         X = np.hstack([np.ones((n_samples, 1)), data]) # adiciona coluna de
11         1s
12         return X
13
14     # Método para treinar o classificador linear
15     def fit(self, data, labels):
16         X = self.calc_matriz_X(data)
17         y = labels
18         X_T = np.transpose(X)
19         X_pseudo_inv = np.dot(np.linalg.inv(np.dot(X_T, X)), X_T)
20         self.w = np.dot(X_pseudo_inv, y)
21         return self.w
22
23     def classificar(self, data):
24         X = self.calc_matriz_X(data)
25         w_T = np.transpose(self.w)
26         y_predicted = np.array([np.sign(np.dot(w_T, xn)) for xn in X])
27         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  $E_{in}$

```
1  def calc_E_in(num_points, verbose = True):
2      lista_E_in = list()
3      lista_target = list()
4      lista_linear = list()
5      for _ in range(1000):
6          # Criar a função target
7          target = Target()
8          a, b = target.generate_random_line()
9          # Criar o dataset
10         dataset = Dataset(num_points)
11         data, labels = dataset.generate_dataset(target)
12         # Criar e treinar o classificador linear
13         linear = Linear()
14         w = linear.fit(data, labels)
15         # Classificar os pontos
16         y_predicted = linear.classificar(data)
17         # Calcular E_in para essa execução
18         lista_E_in.append(np.mean(labels != y_predicted))
19         # Guardar para saída
20         lista_target.append(target)
21         lista_linear.append(linear)
22     E_in = np.mean(lista_E_in)
23     if verbose: print(f"E_in = {E_in:.4f}")
24     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  $E_{out}$  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  $E_{out}$

```
1 def calc_E_out(num_points, lista_target, lista_linear, verbose =
2     True):
3     lista_E_out = list()
4     for target, linear in zip(lista_target, lista_linear):
5         # Criar o dataset com a mesma função target do E_in
6         dataset = Dataset(num_points)
7         data, labels = dataset.generate_dataset(target)
8         # Classificar os pontos com a mesma hipótese do E_in
9         y_predicted = linear.classificar(data)
10        # Calcular E_out para essa execução
11        lista_E_out.append(np.mean(labels != y_predicted))
12    E_out = np.mean(lista_E_out)
13    if verbose: print(f"E_out = {E_out:.4f}")
14    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

```
1 def calc_PLA_iter(num_points):
2     lista_iter = list()
3     for _ in range(1000):
4         # Criar a função target
5         target = Target()
6         target.generate_random_line()
7         # Criar o dataset
8         dataset = Dataset(num_points)
9         data, labels = dataset.generate_dataset(target)
10        # Criar e treinar o classificador linear
11        linear = Linear()
12        w = linear.fit(data, labels)
13        # Criar e treinar o perceptron
14        perceptron = Perceptron2D(weights=w)
15        iter, _ = perceptron.pla(data, labels)
16        lista_iter.append(iter)
17    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 treinada 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 ( $\approx 5$ ). 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

```
1      # Método para treinar o perceptron usando o algoritmo Pocket
2      def pocket(self, data, labels, max_iter = 1000):
3          n_samples = len(data)
4          X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona
                    uma coluna de 1s para o X_0 (coordenada artificial)
5          iterations = 0
6          E_in = 1
7          w = self.w
8          while E_in > 0 and iterations < max_iter:
9              E_in_atual = 0
10             w_atual = w
11             errors = 0
12             for i in range(n_samples):
13                 if labels[i] * np.dot(self.w, X_bias[i]) <= 0:
14                     w_atual += labels[i] * X_bias[i] # atualiza os
                            pesos
15                     errors += 1
16             iterations += 1
17             E_in_atual = errors/n_samples
18             if E_in_atual < E_in: # atualiza os pesos e o E_in apenas
                    se o E_in for menor que o anterior
19                 E_in = E_in_atual
20                 w = w_atual
21             self.w = w
22             return iterations, self.w, E_in
23         setattr(Perceptron2D, "pocket", pocket)
```

**Código 15:** Cálculo do E<sub>in</sub> e do E<sub>out</sub>

```
1      def calc_pocket_E(N1, N2, i, LinReg = False):
2          lista_E_in = list()
3          lista_E_out = list()
4          for _ in range(1000):
5              # Criar a função target
6              target = Target()
7              a, b = target.generate_random_line()
8              # Criar o dataset de treinamento
9              dataset_train = Dataset(N1)
10             x_train, y_train = dataset_train.generate_dataset(target)
11             selected_indices = np.random.choice(len(y_train), int(len(
                    y_train) * 0.1), replace=False) # seleciona 10%
12             y_train[selected_indices] *= -1 # inverte o valor de 10%
13             if LinReg: # pesos inicializados com regressão linear
14                 # Criar e treinar o classificador linear
15                 linear = Linear()
16                 w = linear.fit(x_train, y_train)
17                 # Criar e treinar o perceptron com pocket
18                 perceptron = Perceptron2D(weights=w)
19                 _, _, E_in = perceptron.pocket(x_train, y_train,
                    max_iter = i)
```

```

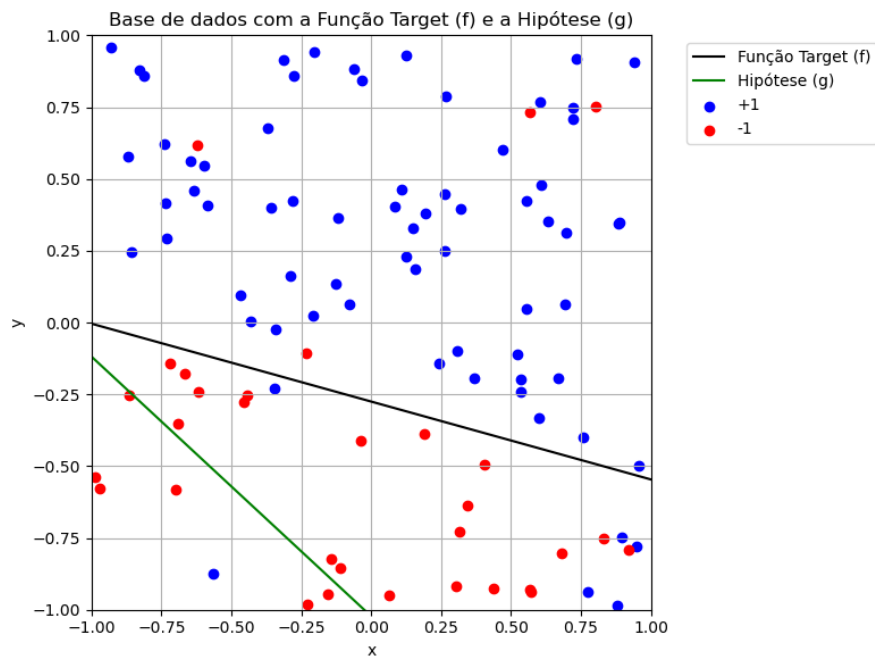
20         lista_E_in.append(E_in)
21     else: # pesos inicializados com 0
22         # Criar e treinar o perceptron com pocket
23         perceptron = Perceptron2D()
24         _, _, E_in = perceptron.pocket(x_train, y_train,
25                                         max_iter = i)
26         lista_E_in.append(E_in)
27         # Criar o dataset de teste
28         dataset_test = Dataset(N2)
29         x_test, y_test = dataset_test.generate_dataset(target)
30         # Classificar os pontos com a mesma hipotese do E_in
31         y_predicted = perceptron.classificar(x_test)
32         # Calcular E_out para essa execucao
33         E_out = np.mean(y_test != y_predicted)
34         lista_E_out.append(E_out)
35     # Printar E_in e E_out medios
36     print(f"E_in = {np.mean(E_in):.4f}")
37     print(f"E_out = {np.mean(E_out):.4f}")
38     # Plotar o dataset, a funcao target e a hipotese g da ultima
39     execucao
40     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  $E_{out} = 0.2540$ .

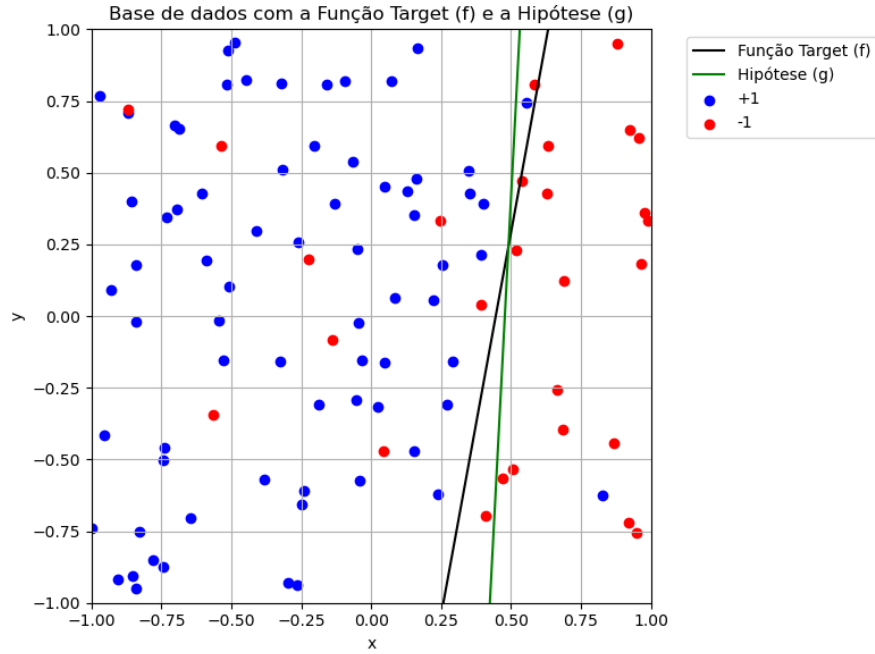
**Figura 3:** Scatterplot inicializando os pesos com 0 e  $i = 10$



(b) Inicializando os pesos com 0;  $i = 50$ ;  $N_1 = 100$ ;  $N_2 = 1000$ .

**Resposta:**  $E_{in} = 0.2400$  e  $E_{out} = 0.0350$ .

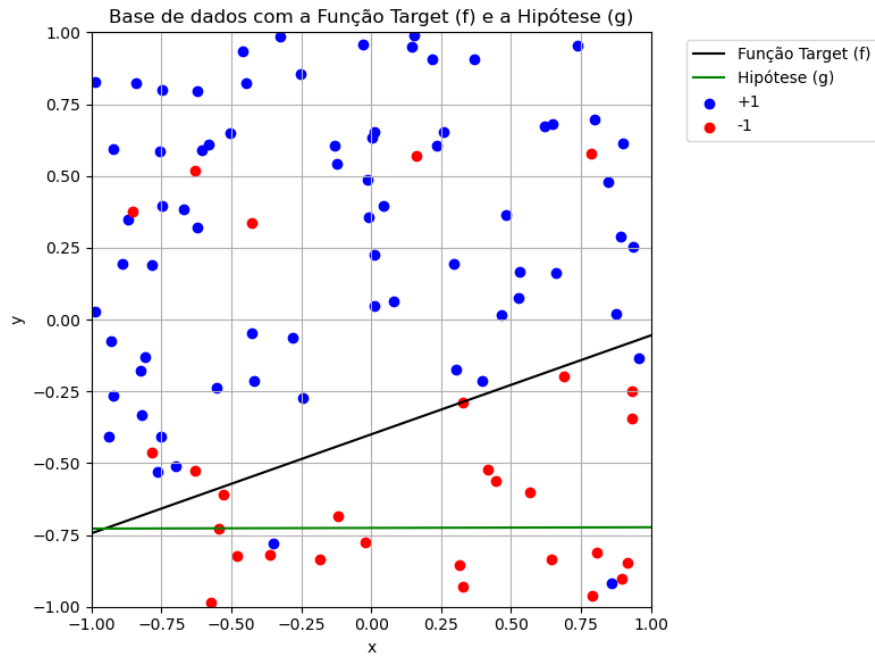
**Figura 4:** Scatterplot inicializando os pesos com 0 e  $i = 50$



(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  $i = 10$

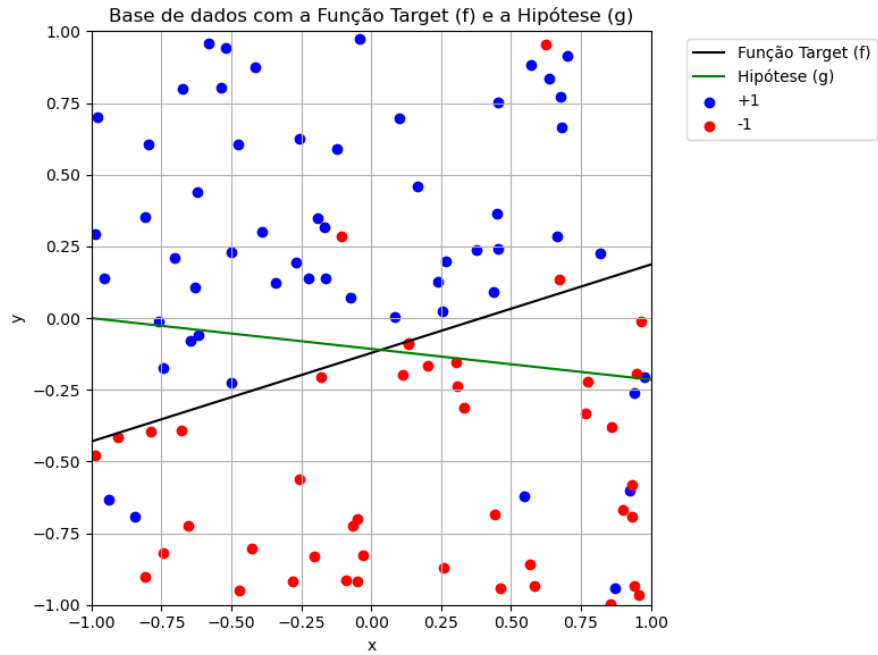


(d) Inicializando os pesos com Regressão Linear;  $i = 50$ ;  $N_1 = 100$ ;  $N_2 = 1000$ .



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

$$f(x_1, x_2) = \text{sign}(x_1^2 + x_2^2 - 0.6)$$

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.

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