

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  $\text{sign}(0) = 0$ , 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 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            while True:
12                predictions = np.sign(np.dot(X_bias, self.w))
13                errors = labels != predictions
14                if not np.any(errors):
15                    break
16                misclassified_indices = np.where(errors)[0]
17                random_choice = np.random.choice(misclassified_indices)
18                self.w += labels[random_choice] * X_bias[random_choice] #
                    atualiza os pesos
19                iterations += 1
20            return iterations, self.w
21
22        # Método para classificar um dataset com base nos pesos aprendidos.
23        def classificar(self, data):
24            n_samples = len(data)

```

```

25         X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma
                        coluna de 1s para o bias X_0
26         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()

```

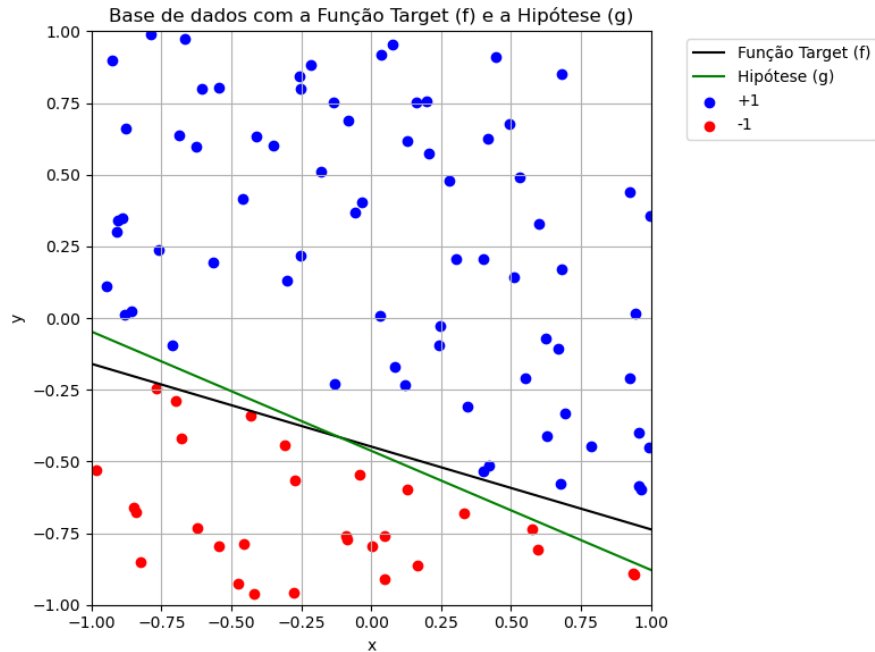
**Código 5:** Teste das classes

```

1  def teste():
2      # Criar a função target
3      target = Target()
4      a, b = target.generate_random_line()
5      # Criar o dataset
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)})")
    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 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)\u2260g(x)] = {np.mean(lista_erro):.4f}")
16     )
17     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

```
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
```

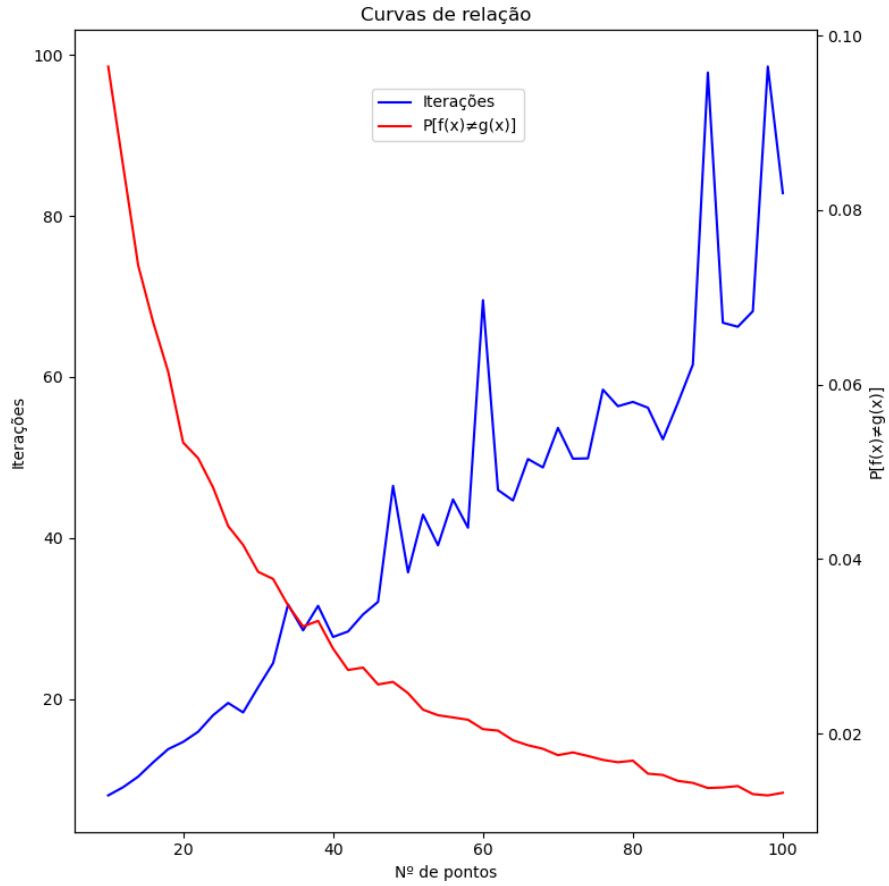
**Código 9:** Plot da probabilidade de erro e do número de iterações

```
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:**

**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_samples = len(data)
9          X = np.hstack([np.ones((n_samples, 1)), data]) # adiciona coluna de
10             1s
11             return X
12
13     # Método para treinar o classificador linear
14     def fit(self, data, labels):
15         X = self.calc_matriz_X(data)
16         y = labels
17         X_T = np.transpose(X)
18         X_pseudo_inv = np.dot(np.linalg.inv(np.dot(X_T, X)), X_T)
19         self.w = np.dot(X_pseudo_inv, y)
20         return self.w
21
22     def classificar(self, data):
23         X = self.calc_matriz_X(data)
24         w_T = np.transpose(self.w)
25         y_predicted = np.array([np.sign(np.dot(w_T, xn)) for xn in X])
26         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 =
    True):
```

```

2      lista_E_out = list()
3      for target, linear in zip(lista_target, lista_linear):
4          # Criar o dataset com a mesma função target do E_in
5          dataset = Dataset(num_points)
6          data, labels = dataset.generate_dataset(target)
7          # Classificar os pontos com a mesma hipótese do E_in
8          y_predicted = linear.classificar(data)
9          # Calcular E_out para essa execução
10         lista_E_out.append(np.mean(labels != y_predicted))
11     E_out = np.mean(lista_E_out)
12     if verbose: print(f"E_out = {E_out:.4f}")
13     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 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 ( $\approx 5$ ). 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
15                        pesos
16                    errors += 1
17            iterations += 1
18            E_in_atual = errors/n_samples
19            if E_in_atual < E_in: # atualiza os pesos e o E_in apenas
20                se o E_in for menor que o anterior
21                E_in = E_in_atual
22                w = w_atual
23            self.w = w
24        return iterations, self.w, E_in
25        setattr(Perceptron2D, "pocket", pocket)

```

**Código 15:** Cálculo do  $E_{in}$  e do  $E_{out}$

```

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(
12             y_train) * 0.1), replace=False) # seleciona 10%
13         y_train[selected_indices] *= -1 # inverte o valor de 10%
14         if LinReg: # pesos inicializados com regressão linear
15             # Criar e treinar o classificador linear
16             linear = Linear()
17             w = linear.fit(x_train, y_train)
18             # Criar e treinar o perceptron com pocket
19             perceptron = Perceptron2D(weights=w)
20             _, _, E_in = perceptron.pocket(x_train, y_train,
21                 max_iter = i)
22             lista_E_in.append(E_in)
23         else: # pesos inicializados com 0
24             # Criar e treinar o perceptron com pocket
25             perceptron = Perceptron2D()
26             _, _, E_in = perceptron.pocket(x_train, y_train,
27                 max_iter = i)
28             lista_E_in.append(E_in)
29         # Criar o dataset de teste
30         dataset_test = Dataset(N2)

```

```

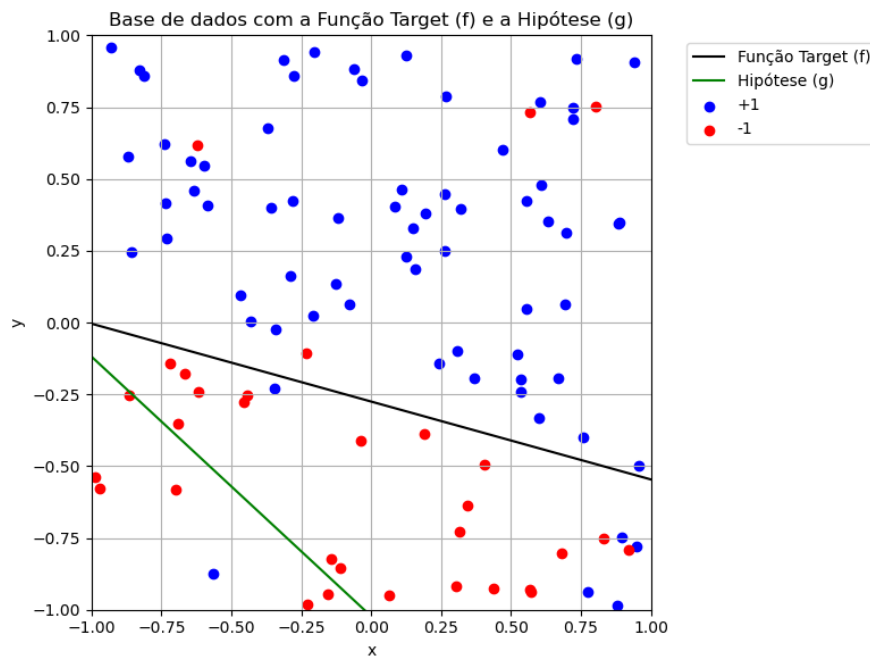
28     x_test, y_test = dataset_test.generate_dataset(target)
29     # Classificar os pontos com a mesma hipotese do E_in
30     y_predicted = perceptron.classificar(x_test)
31     # Calcular E_out para essa execucao
32     E_out = np.mean(y_test != y_predicted)
33     lista_E_out.append(E_out)
34     # Printar E_in e E_out medios
35     print(f"E_in = {np.mean(E_in):.4f}")
36     print(f"E_out = {np.mean(E_out):.4f}")
37     # Plotar o dataset, a funcao target e a hipotese g da ultima
38     execucao
    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  $E_{out} = 0.2540$ .

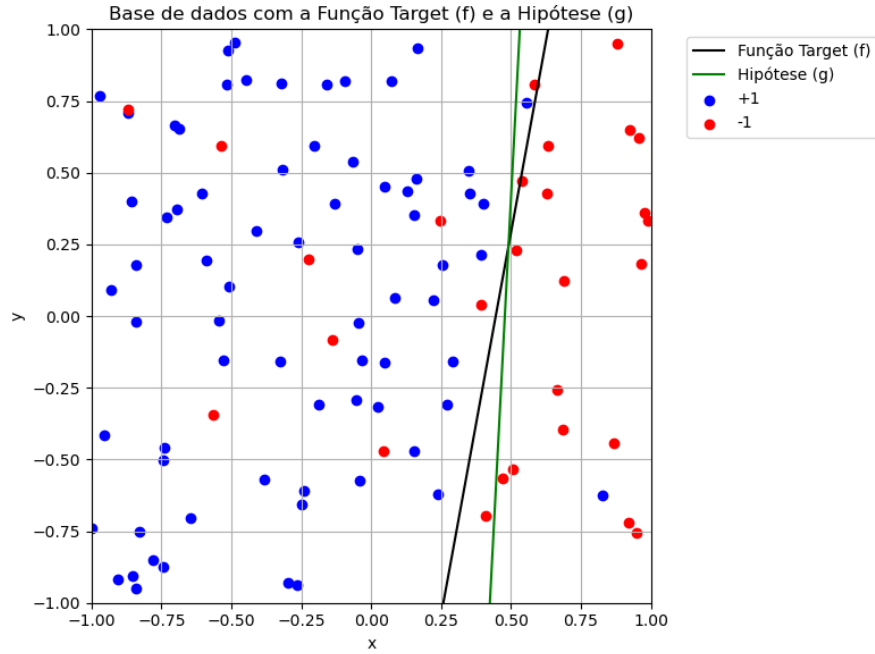
**Figura 3:** Base de dados de treinamento inicializando os pesos com 0 e  $i = 10$



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

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

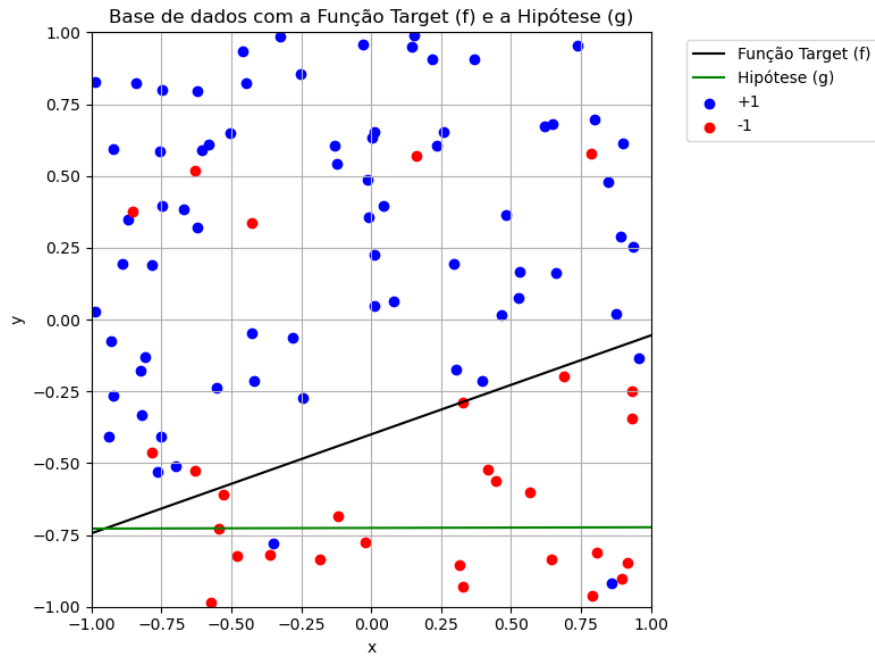
**Figura 4:** Base de dados de treinamento inicializando os pesos com 0 e  $i = 50$



(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:** Base de dados de treinamento inicializando os pesos com Regressão Linear e  $i = 10$

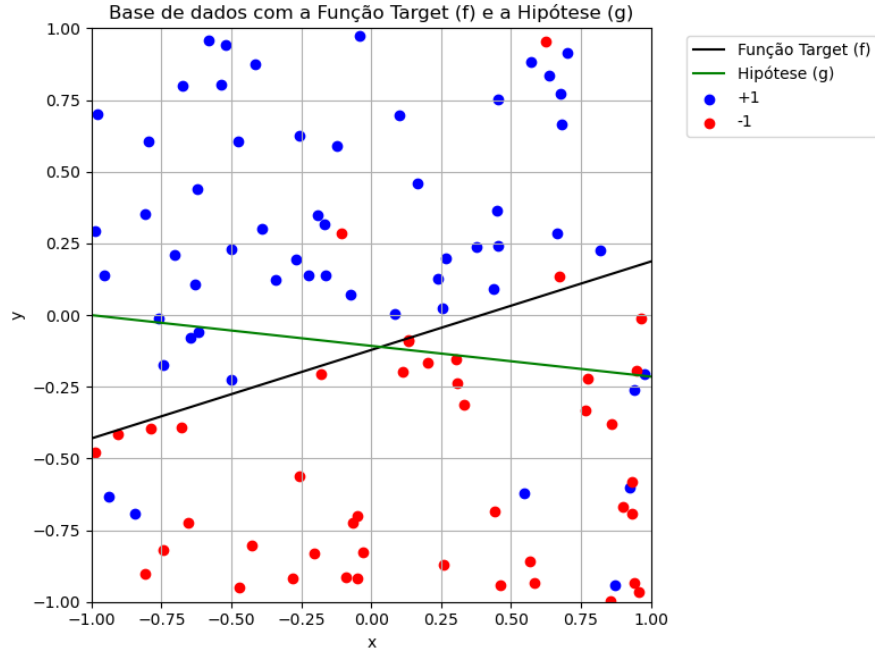


(d) Inicializando os pesos com Regressão Linear;  $i = 50$ ;  $N_1 = 100$ ;  $N_2 = 1000$ .



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

**Figura 6:** Base de dados de treinamento inicializando os pesos com Regressão Linear e  $i = 50$



**Comentários sobre os 4 subitens:** Para todos os subitens o  $E_{out}$  foi menor do que o  $E_{in}$ , o que normalmente não ocorre, porém neste experimento se Justifica pelo fato do dataset de treinamento (dentro da amostra) ter sofrido inversões de labels e o dataset de teste (fora da amostra) não.

### 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.

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

```
1  # Classe para criar a função target não linear
2  class TargetNaoLinear:
3
4      # Método para classificar pontos de acordo a função target
5      def classify_point(self, point):
6          return np.sign(point[0]**2 + point[1]**2 - 0.6)
7
8      # Método para criar uma ellipse para plots
9      def criar_ellipse(self):
10         r = np.sqrt(0.6)
11         ellipse = Ellipse(xy=(0,0), width=2*r, height=2*r, angle=np.degrees
12                        (0),
13                        edgecolor='k', fc='None', lw=2, label='Função
14                        Target (f)')
15
16         return ellipse
```

A classe para gerar a função target não-linear possui dois métodos: um para classificar os pontos de acordo com  $f(x_1, x_2) = \text{sign}(x_1^2 + x_2^2 - 0.6)$  e outro que gera uma ellipse 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

```
1  # Classe para criar e treinar o classificador linear
2  class NaoLinear(Linear):
3      def __init__(self, w = np.zeros(6)):
4          self.w = w # inicializa os pesos (incluindo o w_0)
5
6      # Modifica o método para calcular a matriz X
7      def calc_matriz_X(self, data):
8          X = list()
9          for point in data:
10             x1 = point[0]
```

```

11         x2 = point[1]
12         X.append([1, x1, x2, x1*x2, x1**2, x2**2])
13     return np.array(X)
14
15     # Método novo para criar uma elipse para plots
16     def criar_elipse(self):
17         # Coeficientes da equação geral da cônica ( $Ax_1^2 + Bx_1x_2 + Cx_2^2 +$ 
18              $Dx_1 + Ex_2 + F=0$ )
19         # relacionados com os pesos (1,  $x_1$ ,  $x_2$ ,  $x_1x_2$ ,  $x_1^2$ ,  $x_2^2$ )
20         A = self.w[4] # coef de  $x_1^2$ 
21         B = self.w[3] # coef de  $x_1x_2$ 
22         C = self.w[5] # coef de  $x_2^2$ 
23         D = self.w[1] # coef de  $x_1$ 
24         E = self.w[2] # coef de  $x_2$ 
25         F = self.w[0] # termo independente
26         # Matrizes associadas à equação geral da cônica
27         M = np.array([[A, B / 2], [B / 2, C]])
28         offset = np.array([D, E])
29         # Calcular o centro da elipse
30         center = np.linalg.solve(-2 * M, offset)
31         # Calcular os semi-eixos e o ângulo de rotação
32         eigenvalues, eigenvectors = np.linalg.eigh(M)
33         order = np.argsort(eigenvalues)
34         eigenvalues = eigenvalues[order]
35         eigenvectors = eigenvectors[:, order]
36         semi_major = np.sqrt(-F / (eigenvalues[0] * eigenvalues[1])) / np.
37             sqrt(eigenvalues[0])
38         semi_minor = np.sqrt(-F / (eigenvalues[0] * eigenvalues[1])) / np.
39             sqrt(eigenvalues[1])
40         angle = np.arctan2(eigenvectors[1, 0], eigenvectors[0, 0])
41         # Criar a elipse
42         ellipse = Ellipse(xy=center, width=2 * semi_major, height=2 *
43             semi_minor, angle=np.degrees(angle),
44             edgecolor='g', fc='None', lw=2, label='Hipótese (g)')
45     return ellipse

```

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 para gerar os dados utilizando a função target não-linear e ainda gerando ruído em 10% deles e depois treinar um Classificador Não-Linear (código 19).

#### Código 18: Scatterplot Não-Linear

```

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

```

```

7     if type(hipotese) is NaoLinear:
8         ellipse_hipotese = hipotese.criar_elipse()
9         ax.add_patch(ellipse_hipotese)
10    elif type(hipotese) is Linear:
11        w = hipotese.w
12        x = np.linspace(-1, 1, 100)
13        y_g = -(w[1] * x + w[0]) / w[2]
14        plt.plot(x, y_g, 'g-', label='Hipótese (g)')
15    else:
16        return print("Hipotese não suportada")
17    # plotar os pontos
18    x_pos = [data[i][0] for i in range(len(data)) if labels[i] == 1]
19    y_pos = [data[i][1] for i in range(len(data)) if labels[i] == 1]
20    x_neg = [data[i][0] for i in range(len(data)) if labels[i] == -1]
21    y_neg = [data[i][1] for i in range(len(data)) if labels[i] == -1]
22    plt.scatter(x_pos, y_pos, c='blue', label='+1')
23    plt.scatter(x_neg, y_neg, c='red', label='-1')
24    # ajustar a figura
25    plt.xlim(-1, 1)
26    plt.ylim(-1, 1)
27    plt.xlabel('x')
28    plt.ylabel('y')
29    plt.title('Base de dados com a Função Target (f) não linear e Hipótese (g)')
30    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
31    plt.tight_layout(rect=[0, 0, 1, 1])
32    plt.grid(True)
33    plt.show()

```

**Código 19:** Teste para a Regressão Não-Linear

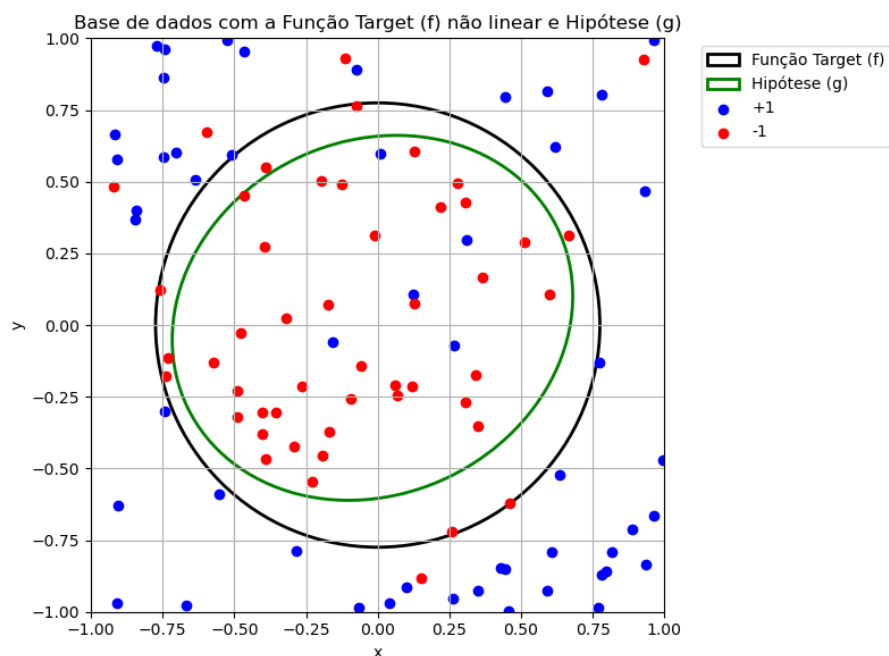
```

1    def teste(num_points, hipotese):
2        # Criar a target não-linear
3        target = TargetNaoLinear()
4        # Criar o dataset
5        dataset = Dataset(num_points)
6        data, labels = dataset.generate_dataset(target)
7        # Adicionar ruído
8        selected_indices = np.random.choice(len(labels), int(len(labels) * 0.1)
9            , replace=False) # seleciona 10%
10       labels[selected_indices] *= -1 # inverte o valor de 10%
11       # Criar o classificador
12       hipotese.fit(data, labels)
13       # Plotar
14       scatterplot_ao_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  $\mathbf{w}$ . 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

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

```

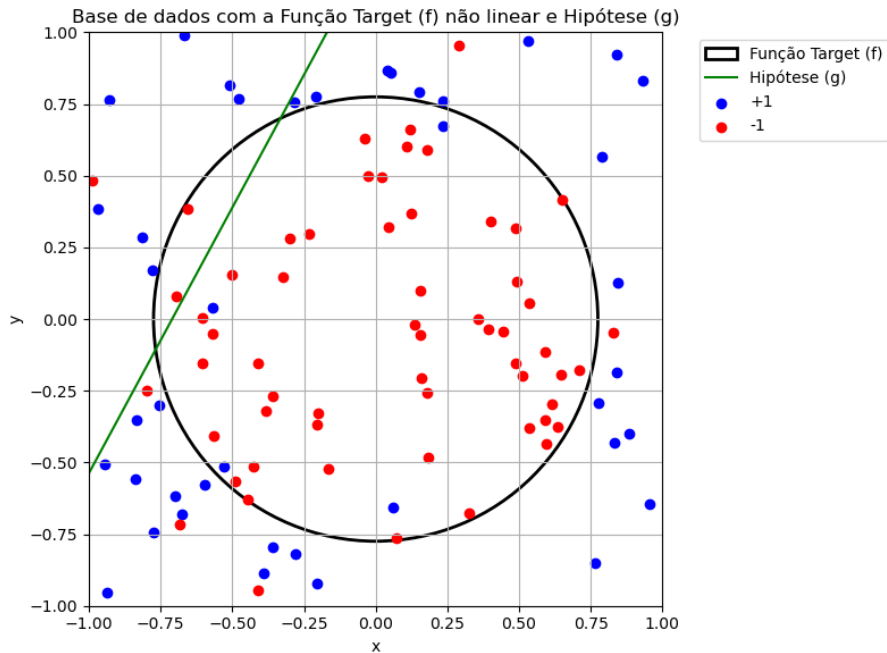
11     labels[selected_indices] *= -1 # inverte o valor de 10%
12     # Criar o classificador não-linear
13     linear = Linear()
14     linear.fit(data, labels)
15     # Classificar os pontos
16     y_predicted = linear.classificar(data)
17     # Calcular E_in para essa execução
18     lista_E_in.append(np.mean(labels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

```

Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset para a função target  $f(x_1, x_2) = \text{sign}(x_1^2 + x_2^2 - 0.6)$  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  $\tilde{\mathbf{w}}$  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)  $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 1.5x_2^2)$
- (b)  $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 15x_2^2)$
- (c)  $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 15x_1^2 + 1.5x_2^2)$
- (d)  $g(x_1, x_2) = \text{sign}(-1 - 1.5x_1 + 0.08x_2 + 0.13x_1x_2 + 0.05x_1^2 + 0.05x_2^2)$
- (e)  $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 1.5x_1x_2 + 0.15x_1^2 + 0.15x_2^2)$

### Justificativa:

Para responder a esse item foi implementada a seguinte função:

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

```

1  def calc_w_naolinear(num_points, verbose = True):
2      lista_w = list()
3      for _ in range(1000):
4          # Criar a target não-linear
5          target = TargetNaoLinear()
6          # Criar o dataset
7          dataset = Dataset(num_points)
8          data, labels = dataset.generate_dataset(target)
9          # Adicionar ruído
10         selected_indices = np.random.choice(len(labels), int(len(
11             labels) * 0.1), replace=False) # seleciona 10%
12         labels[selected_indices] *= -1 # inverte o valor de 10%
13         # Criar o classificador não-linear
14         naolinear = NaoLinear()
15         w = naolinear.fit(data, labels)
16         # Armazenar os pesos
17         lista_w.append(w)
18     w = np.mean(lista_w, axis=0)
19     if verbose:
20         np.set_printoptions(suppress=True)
21         print(f"w = {w}")
22     return w

```

Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset para a função target  $f(x_1, x_2) = \text{sign}(x_1^2 + x_2^2 - 0.6)$  e foi treinado um novo Classificador Não-Linear. O vetor  $\mathbf{w}$  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

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

foi dado pela média em cada coluna da matriz *lista\_w*, a qual cada linha é composta pelo  $\mathbf{w}$  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

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

Foram realizadas 1000 execuções e em cada uma foi gerado um novo dataset para a função target  $f(x_1, x_2) = \text{sign}(x_1^2 + x_2^2 - 0.6)$  e Classificador Linear foi gerado reutilizando o  $\tilde{\mathbf{w}}$  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.



## A Códigos

Código 23: Problema do Perceptron

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # START_TARGET
5 # Classe para criar a função target
6 class Target:
7     def __init__(self):
8         self.a = 0 # coeficiente angular
9         self.b = 0 # coeficiente linear
10
11     # Método para gerar a linha da função target
12     def generate_random_line(self):
13         point1 = np.random.uniform(-1, 1, 2) # ponto aleatorio no domínio
14         point2 = np.random.uniform(-1, 1, 2) # ponto aleatorio no domínio
15         a = (point2[1] - point1[1]) / (point1[0] - point2[0]) # cálculo do
            coeficiente angular
16         b = point1[1] - a*point1[0] # cálculo do coeficiente linear
17         self.a = a
18         self.b = b
19         return a, b
20 # END_TARGET
21
22     # Método para classificar pontos de acordo a função target
23     def classify_point(self, point):
24         a = self.a
25         b = self.b
26         y_reta = a*point[0] + b
27         return np.sign(point[1] - y_reta) # verifica se a coordenada y do ponto
            está acima ou abaixo da reta
28
29 # Classe para criar o dataset
30 class Dataset:
31     def __init__(self, N):
32         self.N = N # tamanho do dataset
33
34     # Método para gerar a base de dados D
35     def generate_dataset(self, target):
36         N = self.N
37         data = np.random.uniform(-1, 1, (N, 2)) # gera N pontos no R2 com
            coordenadas entre [-1, 1]
38         labels = np.array([target.classify_point(point) for point in data])
39         return data, labels
40
41 # Classe para criar e treinar o perceptron 2D
42 class Perceptron2D:
43     def __init__(self, weights = np.zeros(3)):
44         self.w = weights # inicializa os pesos (incluindo o w_0)
45
46     # Método para treinar o perceptron usando o PLA
47     def pla(self, data, labels):
```

```

48     n_samples = len(data)
49     X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma
        coluna de 1s para o X_0 (coordenada artificial)
50     iterations = 0
51     while True:
52         predictions = np.sign(np.dot(X_bias, self.w))
53         errors = labels != predictions
54         if not np.any(errors):
55             break
56         misclassified_indices = np.where(errors)[0]
57         random_choice = np.random.choice(misclassified_indices)
58         self.w += labels[random_choice] * X_bias[random_choice] # atualiza
        os pesos
59         iterations += 1
60     return iterations, self.w
61
62 # Método para classificar um dataset com base nos pesos aprendidos.
63 def classificar(self, data):
64     n_samples = len(data)
65     X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma
        coluna de 1s para o bias X_0
66     return np.sign(np.dot(X_bias, self.w)) # verifica o sinal do produto
        escalar entre x e w
67
68 def scatterplot(data, labels, target, hipotese):
69     a, b = target.a, target.b
70     w = hipotese.w
71     plt.figure(figsize=(8, 6))
72     # plotar a função target
73     x = np.linspace(-1, 1, 100)
74     y_target = a*x+b
75     plt.plot(x, y_target, 'k-', label='Função Target (f)')
76     # plotar a hipótese
77     y_g = -(w[1] * x + w[0]) / w[2]
78     plt.plot(x, y_g, 'g-', label='Hipótese (g)')
79     # plotar os pontos
80     x_pos = [data[i][0] for i in range(len(data)) if labels[i] == 1]
81     y_pos = [data[i][1] for i in range(len(data)) if labels[i] == 1]
82     x_neg = [data[i][0] for i in range(len(data)) if labels[i] == -1]
83     y_neg = [data[i][1] for i in range(len(data)) if labels[i] == -1]
84     plt.scatter(x_pos, y_pos, c='blue', label='+1')
85     plt.scatter(x_neg, y_neg, c='red', label='-1')
86     # ajustar a figura
87     plt.xlim(-1, 1)
88     plt.ylim(-1, 1)
89     plt.xlabel('x')
90     plt.ylabel('y')
91     plt.title('Base de dados com a Função Target (f) e a Hipótese (g)')
92     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
93     plt.tight_layout(rect=[0, 0, 1, 1])
94     plt.grid(True)
95     plt.show()
96
97 def teste():

```

```

98     # Criar a função target
99     target = Target()
100     a, b = target.generate_random_line()
101     # Criar o dataset
102     num_points = 100
103     dataset = Dataset(num_points)
104     data, labels = dataset.generate_dataset(target)
105     # Criar e treinar o perceptron
106     perceptron = Perceptron2D()
107     _, w = perceptron.pla(data, labels)
108     # Plotar resultados
109     scatterplot(data, labels, target, perceptron)
110
111 def calc_num_iter(num_points, verbose = True):
112     lista_iter = list()
113     for _ in range(1000):
114         target = Target()
115         target.generate_random_line()
116         dataset = Dataset(num_points)
117         data, labels = dataset.generate_dataset(target)
118         perceptron = Perceptron2D()
119         iter, _ = perceptron.pla(data, labels)
120         lista_iter.append(iter)
121     if verbose: print(f"{np.mean(lista_iter)} iterações com desvio padrão {np.
122                     std(lista_iter):.4f} (min:{np.min(lista_iter)}, máx:{np.max(lista_iter)
123                     })")
124     return lista_iter
125
126 def calc_p_erro(num_points, verbose = True):
127     lista_erro = list()
128     for _ in range(1000):
129         target = Target()
130         target.generate_random_line()
131         dataset_train = Dataset(num_points)
132         x_train, y_train = dataset_train.generate_dataset(target)
133         dataset_test = Dataset(10000) # mais 10mil pontos
134         x_test, y_test = dataset_test.generate_dataset(target)
135         perceptron = Perceptron2D()
136         perceptron.pla(x_train, y_train)
137         y_predicted = perceptron.classificar(x_test)
138         erro = np.mean(y_test != y_predicted)
139         lista_erro.append(erro)
140     if verbose: print(f"P[f(x)\u2260g(x)] = {np.mean(lista_erro):.4f}")
141     return lista_erro
142
143 def relationship(lista_num_points):
144     lista_iter_medio = list()
145     lista_erro_medio = list()
146     for num_points in lista_num_points:
147         lista_iter = calc_num_iter(num_points, verbose=False)
148         lista_erro = calc_p_erro(num_points, verbose=False)
149         lista_iter_medio.append(np.mean(lista_iter))
150         lista_erro_medio.append(np.mean(lista_erro))
151     return lista_iter_medio, lista_erro_medio

```

```

150
151 def plot_relationship(num_points_list, lista_iter_medio, lista_erro_medio):
152     fig, ax = plt.subplots(1, 1, figsize=(8, 8))
153     ax.plot(num_points_list, lista_iter_medio, c="blue", label="Iterações")
154     ax.set_title("Curvas de relação")
155     ax.set_xlabel('Nº de pontos')
156     ax.set_ylabel('Iterações')
157     ax2=ax.twinx()
158     ax2.plot(num_points_list, lista_erro_medio, c="red", label='P[f(x)\u2260g(x)]
159             ')
159     ax2.set_ylabel('P[f(x)\u2260g(x)]')
160     fig.legend(loc='upper center', bbox_to_anchor=(0.5, 0.9))
161     fig.tight_layout()
162     plt.show()
163
164
165 if __name__ == "__main__":
166     # teste()
167     # calc_num_iter(num_points = 100)
168     # calc_p_erro(num_points = 100)
169     num_points_list = np.arange(10, 101, 2, dtype=int)
170     lista_iter_medio, lista_erro_medio = relationship(num_points_list)
171     plot_relationship(num_points_list, lista_iter_medio, lista_erro_medio)

```

#### Código 24: Problema da Regressão Linear

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from perceptron import Target, Dataset, Perceptron2D, scatterplot
4
5 # Classe para criar e treinar o classificador linear
6 class Linear():
7     def __init__(self):
8         self.w = np.zeros(3) # inicializa os pesos (incluindo o w_0)
9
10    # Método para calcular a matriz X
11    def calc_matriz_X(self, data):
12        n_samples = len(data)
13        X = np.hstack([np.ones((n_samples, 1)), data]) # adiciona coluna de 1s
14        return X
15
16    # Método para treinar o classificador linear
17    def fit(self, data, labels):
18        X = self.calc_matriz_X(data)
19        y = labels
20        X_T = np.transpose(X)
21        X_pseudo_inv = np.dot(np.linalg.inv(np.dot(X_T, X)), X_T)
22        self.w = np.dot(X_pseudo_inv, y)
23        return self.w
24
25    def classificar(self, data):
26        X = self.calc_matriz_X(data)
27        w_T = np.transpose(self.w)
28        y_predicted = np.array([np.sign(np.dot(w_T, xn)) for xn in X])

```

```

29         return y_predicted
30
31 # Método para treinar o perceptron usando o algoritmo Pocket
32 def pocket(self, data, labels, max_iter = 1000):
33     n_samples = len(data)
34     X_bias = np.hstack([np.ones((n_samples, 1)), data]) # adiciona uma coluna
35     # de 1s para o X_0 (coordenada artificial)
36     iterations = 0
37     E_in = 1
38     w = self.w
39     while E_in > 0 and iterations < max_iter:
40         E_in_atual = 0
41         w_atual = w
42         errors = 0
43         for i in range(n_samples):
44             if labels[i] * np.dot(self.w, X_bias[i]) <= 0:
45                 w_atual += labels[i] * X_bias[i] # atualiza os pesos
46                 errors += 1
47             iterations += 1
48         E_in_atual = errors/n_samples
49         if E_in_atual < E_in: # atualiza os pesos e o E_in apenas se o E_in for
50             # menor que o anterior
51             E_in = E_in_atual
52             w = w_atual
53     self.w = w
54     return iterations, self.w, E_in
55 setattr(Perceptron2D, "pocket", pocket)
56
57 def teste():
58     # Criar a função target
59     target = Target()
60     a, b = target.generate_random_line()
61     # Criar o dataset e a função target
62     num_points = 100
63     dataset = Dataset(num_points)
64     data, labels = dataset.generate_dataset(target)
65     # Criar e treinar o classificador linear
66     linear = Linear()
67     w = linear.fit(data, labels)
68     # Plotar resultados
69     scatterplot(data, labels, target, linear)
70
71 def calc_E_in(num_points, verbose = True):
72     lista_E_in = list()
73     lista_target = list()
74     lista_linear = list()
75     for _ in range(1000):
76         # Criar a função target
77         target = Target()
78         a, b = target.generate_random_line()
79         # Criar o dataset
80         dataset = Dataset(num_points)
81         data, labels = dataset.generate_dataset(target)
82         # Criar e treinar o classificador linear

```

```

81     linear = Linear()
82     w = linear.fit(data, labels)
83     # Classificar os pontos
84     y_predicted = linear.classificar(data)
85     # Calcular E_in para essa execução
86     lista_E_in.append(np.mean(labels != y_predicted))
87     # Guardar para saída
88     lista_target.append(target)
89     lista_linear.append(linear)
90 E_in = np.mean(lista_E_in)
91 if verbose: print(f"E_in = {E_in:.4f}")
92 return E_in, lista_target, lista_linear
93
94 def calc_E_out(num_points, lista_target, lista_linear, verbose = True):
95     lista_E_out = list()
96     for target, linear in zip(lista_target, lista_linear):
97         # Criar o dataset com a mesma função target do E_in
98         dataset = Dataset(num_points)
99         data, labels = dataset.generate_dataset(target)
100        # Classificar os pontos com a mesma hipótese do E_in
101        y_predicted = linear.classificar(data)
102        # Calcular E_out para essa execução
103        lista_E_out.append(np.mean(labels != y_predicted))
104    E_out = np.mean(lista_E_out)
105    if verbose: print(f"E_out = {E_out:.4f}")
106    return E_out
107
108 def calc_PLA_iter(num_points):
109     lista_iter = list()
110     for _ in range(1000):
111         # Criar a função target
112         target = Target()
113         target.generate_random_line()
114         # Criar o dataset
115         dataset = Dataset(num_points)
116         data, labels = dataset.generate_dataset(target)
117         # Criar e treinar o classificador linear
118         linear = Linear()
119         w = linear.fit(data, labels)
120         # Criar e treinar o perceptron
121         perceptron = Perceptron2D(weights=w)
122         iter, _ = perceptron.pla(data, labels)
123         lista_iter.append(iter)
124     print(f"{np.mean(lista_iter)} iterações com desvio padrão {np.std(
125         lista_iter):.4f} (min:{np.min(lista_iter)}, máx:{np.max(lista_iter)})")
126
127 def calc_pocket_E(N1, N2, i, LinReg = False):
128     lista_E_in = list()
129     lista_E_out = list()
130     for _ in range(1000):
131         # Criar a função target
132         target = Target()
133         a, b = target.generate_random_line()
134         # Criar o dataset de treinamento

```

```

134     dataset_train = Dataset(N1)
135     x_train, y_train = dataset_train.generate_dataset(target)
136     selected_indices = np.random.choice(len(y_train), int(len(y_train) *
137         0.1), replace=False) # seleciona 10%
138     y_train[selected_indices] *= -1 # inverte o valor de 10%
139     if LinReg: # pesos inicializados com regressão linear
140         # Criar e treinar o classificador linear
141         linear = Linear()
142         w = linear.fit(x_train, y_train)
143         # Criar e treinar o perceptron com pocket
144         perceptron = Perceptron2D(weights=w)
145         _, _, E_in = perceptron.pocket(x_train, y_train, max_iter = i)
146         lista_E_in.append(E_in)
147     else: # pesos inicializados com 0
148         # Criar e treinar o perceptron com pocket
149         perceptron = Perceptron2D()
150         _, _, E_in = perceptron.pocket(x_train, y_train, max_iter = i)
151         lista_E_in.append(E_in)
152     # Criar o dataset de teste
153     dataset_test = Dataset(N2)
154     x_test, y_test = dataset_test.generate_dataset(target)
155     # Classificar os pontos com a mesma hipótese do E_in
156     y_predicted = perceptron.classificar(x_test)
157     # Calcular E_out para essa execução
158     E_out = np.mean(y_test != y_predicted)
159     lista_E_out.append(E_out)
160     # Printar E_in e E_out médios
161     print(f"E_in = {np.mean(E_in):.4f}")
162     print(f"E_out = {np.mean(E_out):.4f}")
163     # Plotar o dataset, a função target e a hipótese g da última execução
164     scatterplot(x_train, y_train, target, perceptron)
165
166 if __name__ == "__main__":
167     teste()
168     _, target, linear = calc_E_in(num_points=100)
169     calc_E_out(num_points=1000, lista_target=target, lista_linear=linear)
170     calc_PLA_iter(num_points=10)
171     calc_pocket_E(N1 = 100, N2 = 1000, i = 50, LinReg = True)

```

### Código 25: Problema da Regressão Não-Linear

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.patches import Ellipse
4  from perceptron import Dataset
5  from regressao_linear import Linear
6
7  # Classe para criar a função target não linear
8  class TargetNaoLinear:
9
10     # Método para classificar pontos de acordo a função target
11     def classify_point(self, point):
12         return np.sign(point[0]**2 + point[1]**2 - 0.6)

```

```

13
14 # Método para criar uma ellipse para plots
15 def criar_ellipse(self):
16     r = np.sqrt(0.6)
17     ellipse = Ellipse(xy=(0,0), width=2*r, height=2*r, angle=np.degrees(0),
18                       edgecolor='k', fc='None', lw=2, label='Função Target
19                               (f)')
20
21     return ellipse
22
23 # Classe para criar e treinar o classificador linear
24 class NaoLinear(Linear):
25
26     def __init__(self, w = np.zeros(6)):
27         self.w = w # inicializa os pesos (incluindo o w_0)
28
29 # Modifica o método para calcular a matriz X
30 def calc_matriz_X(self, data):
31     X = list()
32     for point in data:
33         x1 = point[0]
34         x2 = point[1]
35         X.append([1, x1, x2, x1*x2, x1**2, x2**2])
36     return np.array(X)
37
38 # Método novo para criar uma ellipse para plots
39 def criar_ellipse(self):
40     # Coeficientes da equação geral da cônica ( $Ax_1^2 + Bx_1x_2 + Cx_2^2 + Dx_1$ 
41     #  $+ Ex_2 + F=0$ )
42     # relacionados com os pesos (1, x1, x2, x1x2, x1^2, x2^2)
43     A = self.w[4] # coef de x1^2
44     B = self.w[3] # coef de x1x2
45     C = self.w[5] # coef de x2^2
46     D = self.w[1] # coef de x1
47     E = self.w[2] # coef de x2
48     F = self.w[0] # termo independente
49     # Matrizes associadas à equação geral da cônica
50     M = np.array([[A, B / 2], [B / 2, C]])
51     offset = np.array([D, E])
52     # Calcular o centro da ellipse
53     center = np.linalg.solve(-2 * M, offset)
54     # Calcular os semi-eixos e o ângulo de rotação
55     eigenvalues, eigenvectors = np.linalg.eigh(M)
56     order = np.argsort(eigenvalues)
57     eigenvalues = eigenvalues[order]
58     eigenvectors = eigenvectors[:, order]
59     semi_major = np.sqrt(-F / (eigenvalues[0] * eigenvalues[1])) / np.sqrt(
60         eigenvalues[0])
61     semi_minor = np.sqrt(-F / (eigenvalues[0] * eigenvalues[1])) / np.sqrt(
62         eigenvalues[1])
63     angle = np.arctan2(eigenvectors[1, 0], eigenvectors[0, 0])
64     # Criar a ellipse
65     ellipse = Ellipse(xy=center, width=2 * semi_major, height=2 *
66                       semi_minor, angle=np.degrees(angle),
67                       edgecolor='g', fc='None', lw=2, label='Hipótese (g)'
68                       )

```



```

61         return ellipse
62
63
64 def scatterplot_nao_linear(data, labels, target, hipotese):
65     fig, ax = plt.subplots(subplot_kw={'aspect': 'equal'}, figsize=(8, 6))
66     # plotar a função target
67     ellipse_target = target.criar_elipse()
68     ax.add_patch(ellipse_target)
69     # plotar a hipótese
70     if type(hipotese) is NaoLinear:
71         ellipse_hipotese = hipotese.criar_elipse()
72         ax.add_patch(ellipse_hipotese)
73     elif type(hipotese) is Linear:
74         w = hipotese.w
75         x = np.linspace(-1, 1, 100)
76         y_g = -(w[1] * x + w[0]) / w[2]
77         plt.plot(x, y_g, 'g-', label='Hipótese (g)')
78     else:
79         return print("Hipotese não suportada")
80     # plotar os pontos
81     x_pos = [data[i][0] for i in range(len(data)) if labels[i] == 1]
82     y_pos = [data[i][1] for i in range(len(data)) if labels[i] == 1]
83     x_neg = [data[i][0] for i in range(len(data)) if labels[i] == -1]
84     y_neg = [data[i][1] for i in range(len(data)) if labels[i] == -1]
85     plt.scatter(x_pos, y_pos, c='blue', label='+1')
86     plt.scatter(x_neg, y_neg, c='red', label='-1')
87     # ajustar a figura
88     plt.xlim(-1, 1)
89     plt.ylim(-1, 1)
90     plt.xlabel('x')
91     plt.ylabel('y')
92     plt.title('Base de dados com a Função Target (f) não linear e Hipótese (g)')
93
94     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
95     plt.tight_layout(rect=[0, 0, 1, 1])
96     plt.grid(True)
97     plt.show()
98
99 def teste(num_points, hipotese):
100     # Criar a target não-linear
101     target = TargetNaoLinear()
102     # Criar o dataset
103     dataset = Dataset(num_points)
104     data, labels = dataset.generate_dataset(target)
105     # Adicionar ruído
106     selected_indices = np.random.choice(len(labels), int(len(labels) * 0.1),
107                                         replace=False) # seleciona 10%
108     labels[selected_indices] *= -1 # inverte o valor de 10%
109     # Criar o classificador
110     hipotese.fit(data, labels)
111     # Plotar
112     scatterplot_nao_linear(data, labels, target, hipotese)
113
114 def calc_Ein_linear(num_points, verbose = True):

```

```

113 lista_E_in = list()
114 for _ in range(1000):
115     # Criar a target não-linear
116     target = TargetNaoLinear()
117     # Criar o dataset
118     dataset = Dataset(num_points)
119     data, labels = dataset.generate_dataset(target)
120     # Adicionar ruído
121     selected_indices = np.random.choice(len(labels), int(len(labels) * 0.1)
122         , replace=False) # seleciona 10%
123     labels[selected_indices] *= -1 # inverte o valor de 10%
124     # Criar o classificador não-linear
125     linear = Linear()
126     linear.fit(data, labels)
127     # Classificar os pontos
128     y_predicted = linear.classificar(data)
129     # Calcular E_in para essa execução
130     lista_E_in.append(np.mean(labels != y_predicted))
131 E_in = np.mean(lista_E_in)
132 # Plotar a última execução
133 if verbose: print(f"E_in = {E_in:.4f}")
134 return E_in
135
136 def calc_w_naolinear(num_points, verbose = True):
137     lista_w = list()
138     for _ in range(1000):
139         # Criar a target não-linear
140         target = TargetNaoLinear()
141         # Criar o dataset
142         dataset = Dataset(num_points)
143         data, labels = dataset.generate_dataset(target)
144         # Adicionar ruído
145         selected_indices = np.random.choice(len(labels), int(len(labels) * 0.1)
146             , replace=False) # seleciona 10%
147         labels[selected_indices] *= -1 # inverte o valor de 10%
148         # Criar o classificador não-linear
149         naolinear = NaoLinear()
150         w = naolinear.fit(data, labels)
151         # Armazenar os pesos
152         lista_w.append(w)
153     w = np.mean(lista_w, axis=0)
154     if verbose:
155         np.set_printoptions(suppress=True)
156         print(f"w = {w}")
157     return w
158
159 def calc_Eout_naolinear(num_points, w, verbose = True):
160     lista_E_out = list()
161     for _ in range(1000):
162         # Criar a target não-linear
163         target = TargetNaoLinear()
164         # Criar o dataset de teste
165         dataset = Dataset(num_points)
166         data, labels = dataset.generate_dataset(target)

```

```

165     # Criar o classificador não-linear
166     nao_linear = NaoLinear(w = w) # força a usar os pesos anteriores
167     # Classificar os pontos com a mesma hipótese do E_in
168     y_predicted = nao_linear.classificar(data)
169     # Calcular E_out para essa execução
170     lista_E_out.append(np.mean(labels != y_predicted))
171     E_out = np.mean(lista_E_out)
172     if verbose: print(f"E_out = {E_out:.4f}")
173     return E_out
174
175 if __name__ == "__main__":
176     teste(num_points = 100, hipótese = Linear())
177     teste(num_points = 100, hipótese = NaoLinear())
178     calc_Ein_linear(num_points=1000)
179     w = calc_w_nao_linear(num_points=1000)
180     calc_Eout_nao_linear(num_points=1000, w = w)

```