Isabel Sales de Castro Almeida

DRE:118016977

Programa de Engenharia de Sistemas e Computação

Trabalho Prático III (Parte 1) do curso de Inteligência Computacional II

COPPE-UFRJ

Rio de Janeiro - RJ

2018

Programa de Engenharia de Sistemas e Computação

Isabel Sales de Castro Almeida

Coppe – UFRJ

Este documento corresponde a parte 1 do Trabalho Prático III do curso de Inteligência Computacional II (CPS849) do Programa de Engenharia de Sistemas e Computação da Coppe - UFRJ.

Rio de Janeiro - RJ
2018

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Este documento corresponde a parte 1 do Trabalho Prático III do curso de Inteligência Computacional II (CPS849) do Programa de Engenharia de Sistemas e Computação da Coppe - UFRJ. O mesmo visa a implementação dos exercícios computacionais do *Homework 5*, disponibilizado em: https://work.caltech.edu/homework/hw5.pdf.

O documento é composto por duas seções: Gradiente Descendente e Regressão Logística. Em ambas é apresentado uma breve introdução, seguida pelos resultados obtidos e uma interpretação dos mesmos. Os códigos comentados se encontram no anexo.

2. GRADIENTE DESCENDENTE

Gradiente Descendente é um método numérico usado em otimização. O objetivo é encontrar um mínimo (local) de uma função e usa-se um esquema iterativo, onde em cada passo se toma a direção (negativa) do gradiente, que corresponde à direção de declive máximo.

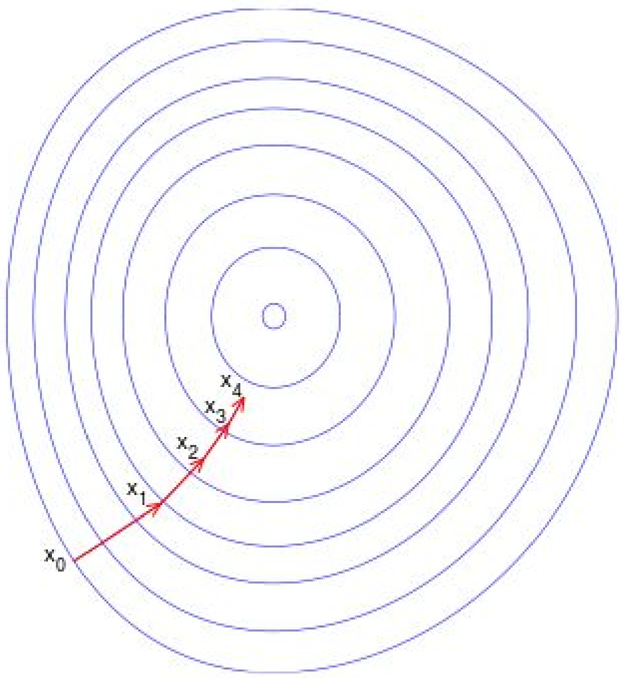

Figura 1: Gradiente Descendente

A figura acima ilustra o método do Gradiente Descendente (as linhas azuis correspondem a curva de nível, e as setas em vermelho correspondem a 4 iterações do método.

Explicação Matemática

Um exemplo de regressão linear será utilizado, com apenas uma variável dependente e uma independente. A relação entre elas pode ser expressa na equação:

$$\boldsymbol{y} = b + \boldsymbol{x}w + \epsilon$$

Figura 2: Equação 1

A ideia é achar os valores $\hat{b}$ e $\hat{w}$ que minimizam o quadrado da norma do vetor $\boldsymbol{\epsilon}$.

A concepção por trás dos métodos iterativos de otimização é bastante simples: começa com algum chute razoável para os valores de $\hat{b}$ e $\hat{w}$, em seguida estes são atualizados na direção certa até que chegar no valor mínimo da função custo, nesse caso, $||\hat{\boldsymbol{\epsilon}}||^2$. Matematicamente, nota-se que a função custo $||\hat{\boldsymbol{\epsilon}}||^2$ é uma função de $\hat{b}$ e $\hat{w}$.

$$
\begin{aligned}
L(\hat{b}, \hat{w}) &= ||\hat{\boldsymbol{\epsilon}}||^2 \\
&= \sum \hat{\epsilon}^2 \\
&= \sum (\hat{y} - y)^2 \\
&= \sum (\hat{b} + x\hat{w} - y)^2 \\
&= (\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y})^T(\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y})
\end{aligned}
$$

Figura 3: Equação 2

Em que $\hat{w}$ é o vetor com os parâmetros, incluindo $\hat{b}$. É possível minimizar essa função custo em seus parâmetros usando cálculo multivariado. Essa função custo, específica de regressão linear, é uma função convexa, o que quer dizer que o único ponto de mínimo que ela tem é um mínimo global. Em outras palavras, a função custo pode ser vista como uma tigela, e o gradiente desta função apontará a direção de descida mais íngreme, de forma que possamos chegar ao fundo da tigela, onde está o ponto de menor custo.

No exemplo com apenas dois parâmetros, essas direções são nos espaços de $\hat{b}$ e $\hat{w}$. Para implementar o gradiente descendente, basta atualizar simultaneamente os valores de $\hat{b}$ e $\hat{w}$, subtraindo deles as respectivas derivadas parciais da função custo, vezes uma taxa de aprendizado α (o sinal ":=" abaixo significa atualizar):

$$\hat{b} := \hat{b} - \alpha \frac{\partial}{\partial \hat{b}} L(\hat{b}, \hat{w})$$
$$\hat{w} := \hat{w} - \alpha \frac{\partial}{\partial \hat{w}} L(\hat{b}, \hat{w})$$

Figura 4: Equação 3

Ou, no caso específico da função custo de soma dos erros quadrados:

$$\hat{b} := \hat{b} - \alpha 2 \sum (\hat{b} + \hat{w}x - y)$$
$$\hat{w} := \hat{w} - \alpha 2 \sum ((\hat{b} + \hat{w}x - y)x)$$

Figura 5: Equação 4

Para simplificar, é possível retirar da fórmula o 2 que não fará diferença, uma vez que as derivadas já estão sendo multiplicadas por uma constante α. Simplificando mais ainda, é possível utilizar a notação de vetores:

$$\hat{\boldsymbol{w}} := \hat{\boldsymbol{w}} - \alpha \nabla(L)$$

Figura 6: Equação 5

No caso, $L(\hat{\boldsymbol{w}}) = (\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y})^T(\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y})$ é a função custo e é possível achar o gradiente dela com cálculo de vetores:

$$
\begin{aligned}
\nabla(L) &= \frac{\partial}{\partial \hat{\boldsymbol{w}}}(\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y})^T(\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y}) \\
&= \frac{\partial}{\partial \hat{\boldsymbol{w}}}(\hat{\boldsymbol{w}}^T\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} - 2\hat{\boldsymbol{w}}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{y}) \\
&= 2\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} - 2\boldsymbol{X}^T\boldsymbol{y}
\end{aligned}
$$

Figura 7: Equação 6

Em que $\hat{w}$ é o vetor dos parâmetros da regressão linear, incluindo o intercepto $\hat{b}$. Note que essa última regra de atualização é geral para qualquer número de dimensões que nossos dados possam ter.

Visualizando o Gradiente Descendente

Para melhor entendimento do algoritmo, é uma boa visualizar como é a função custo quando plotada nas duas dimensões dos parâmetros $\hat{b}$ e $\hat{w}$:

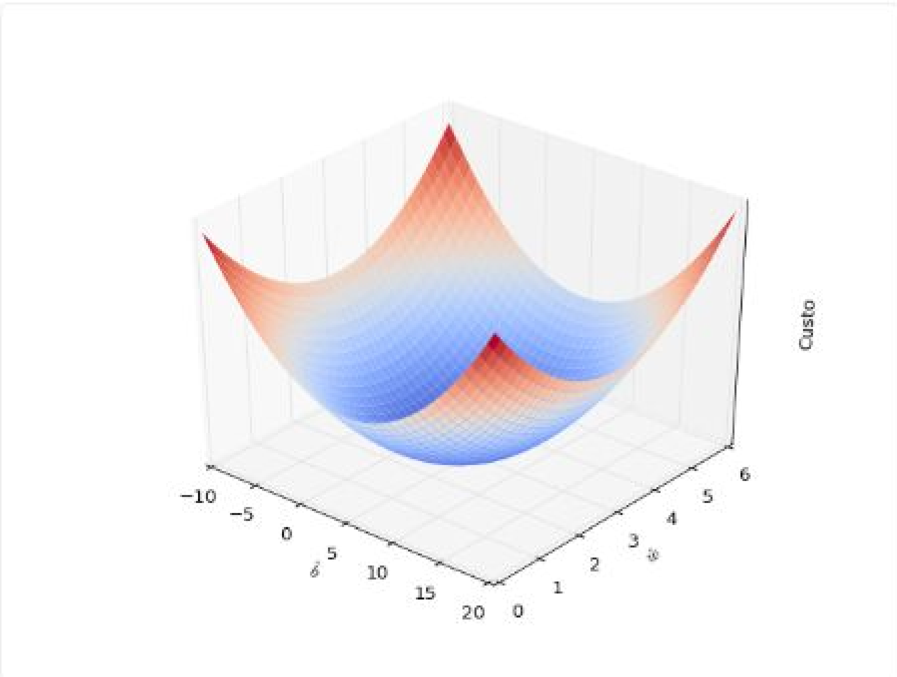

Figura 8 : Visualização da função de custo

Nota-se que a função de custo parece uma tigela. O gradiente desta função é simplesmente um vetor de derivadas parciais, que dão a inclinação dessa tigela em cada ponto e em cada direção:

$$\nabla(L) = \left[\frac{\partial L}{\partial \hat{b}}, \frac{\partial L}{\partial \hat{w}}\right]$$

Figura 9: Equação 7

Seguindo na direção oposta do gradiente, chegaremos no ponto de mínimo. Podemos traçar uma analogia com uma bolinha de gude sendo solta em uma tigela: a bolinha irá descer na direção mais inclinada e eventualmente irá parar no ponto mais baixo da tigela. Há uma importante diferença, no entanto, quando se fala de uma bolinha de gude deslizando para o fundo de uma tigela, podemos visualizar a bolinha começando com uma pequena velocidade e acelerando ao longo do trajeto.

Com gradiente descendente ocorre o oposto: inicialmente, os parâmetros $\hat{b}$ e $\hat{w}$ caminham rapidamente em direção ao ponto de mínimo e, quanto mais se aproximam dele, passam a caminhar cada vez mais devagar.

Vejamos como a cada iteração os parâmetros $\hat{b}$ e $\hat{w}$ dão um passo em direção ao mínimo. O tamanho desse passo será o valor do gradiente naquele ponto multiplicado pela constante α. Note que quanto mais próximos estamos do ponto de mínimo, menor a inclinação da função custo, ou seja, menor o gradiente, logo, menor o passo dado em direção ao mínimo.

Essa característica do método de gradiente descendente é ruim, em vista que atrasa o processo de aprendizado quando chegamos próximo do mínimo, mas ao mesmo tempo é boa porque nos permite uma exploração mais minuciosa da superfície de custo em torno do ponto de mínimo. Dessa forma, é possível localizá-lo com mais precisão. Isso é muito importante quando estamos lidando com aprendizado de redes neurais com milhares de parâmetros e uma função custo não convexa é importante essa exploração minuciosa do espaço da função custo.

Implementação

As soluções foram desenvolvidas em Python, por ser uma linguagem de alto nível e de desenvolvimento rápido (RAD - Rapid Application Development). Foram apresentados três problemas sobre Gradiente Descendente e para cada, era necessário apresentar qual das soluções informadas é a correta ou a mais próxima do resultado obtido pelo algoritmo.

Problemas 1 e 2:

Considere o erro não linear de superfície $E(u,v) = (ue^{v} - 2ve^{-u})^2$. Vamos iniciar o ponto (u, v) = (1, 1) e minimizar este erro usando gradiente descendente no espaço (uv). Use η = 0.1 (taxa de aprendizado, não tamanho de parada).

1) Quantas iterações (das opções dadas) que encontra o erro E(u, v) abaixo de 10^{-14} pela primeira vez? Na sua implementação, use precisão dupla para obter a exatidão necessária.

a) 1

b) 3

c) 5

d) 10

e) 17

2) Após executar iterações suficientes para que o erro seja menor que 10^{-14}, quais são os valores próximos (na distância euclidiana) entre as seguintes opções de valores (u, v) que você obteve no Problema 1?

a) (1.000, 1.000)

b) (0.713, 0.045)

c) (0.016, 0.112)

d) (-0.083, 0.029)

e) (0.045, 0.024)

Resultados obtidos pelo algoritmo

```
---------------------------------- Resultado Problema 1 -------------------------------------------------
Quantidade de iterações que encontra o erro E(u, v) abaixo de (10)^14 pela primeira vez:  10
Final(u,v) =  [0.04473629 0.02395871]

---------------------------------- Resultado Problema 2 -------------------------------------------------
Distância dos pontos dados até a final (u, v):
Ponto x =  [1. 1.]  => distância =  1.365717886924672
Ponto x =  [0.713 0.045]  => distância =  0.6685948857743971
Ponto x =  [0.016 0.112]  => distância =  0.0926123232021653
Ponto x =  [-0.083  0.029]  => distância =  0.12783573228217807
Ponto x =  [0.045 0.024]  => distância =  0.0002669218610597792

O ponto com distância mínima até a final (u, v) é:  [0.045 0.024]
```

Figura 10: Resultados problemas 1 e 2

O algoritmo retorna 10, como a quantidade de iterações que encontra o erro E(u, v) abaixo de 10^{-14} pela primeira vez. Deste modo, é possível concluir que a letra **d**, é a alternativa correta para o problema **1**. O ponto com distância mínima até a final (u, v) obtida pelo algoritmo é **[0.045, 0.024]**, sendo a alternativa **e**, a alternativa correta para o problema **2**.

Análise e conclusões

No primeiro problema, o algoritmo tem como objetivo encontrar a quantidade de iterações que encontra o erro E(u,v) abaixo do número (10)^-14 pela primeira vez. É feito um laço de repetição e quando E(u,v) < (10)^-14 é realizado um break e o número de iterações é retornado. Para o valor informado no enunciado (10)^-14, o algoritmo retorna 10 como o número de iterações necessárias. Testando com outros valores, notamos que quanto menor o número de comparação com o E(u,v), maior será a quantidade de iterações. Um exemplo é o número (10)^-20, onde o algoritmo retorna 13 como a quantidade de iterações. Para comparação com valores positivos maiores que zero, sempre será necessário apenas 1 iteração.

No segundo problema, o objetivo é encontrar o ponto com menor distância euclidiana de (u,v), encontrada no problema 1. Com os dados informados no enunciado, o ponto mais próximo é o [0.045 0.024], percebemos também que este é o ponto mais próximo sempre que o erro E(u,v) é um número negativo. Para valores maiores que zero a distância mais próximas passa a ser, dentre as opções apresentadas [-0.083 0.029].

Problema 3

3) Agora, nós vamos comparar a performance de “coordenada descendente”. Em cada iteração, nós vamos ter dois passos ao longo de duas coordenadas. O passo 1 é mover apenas a coordenada u para reduzir o erro (assuma a aproximação de primeira ordem assim como no gradiente descendente), e o passo 2 é reavaliar e mover apenas a coordenada v para reduzir o erro (novamente assuma a aproximação de primeira ordem). Continue usando a taxa de aprendizagem η = 0.1 como feito no gradiente descendente. Qual vai ser o valor de erro E(u, v) mais próximo após 15 iterações completas (30 passos)?

a) 10^{-1}

b) 10^{-7}

c) 10^{-14}

d) 10^{-17}

e) 10^{-20}

Resultados obtidos pelo algoritmo

```
---------------------------------- Resultado Problema 3 ----------------------------------------------
Coordenada descendente:
Valor de erro E(u, v) mais próximo após 15 iterações completas: 0.13981379199615324
```

Figura 11: Resultado problema 3

Para o cálculo do valor de erro E(u,v) mais próximo após 15 iterações completas, o algoritmo apresentou o valor **0.13981379199615324**, sendo a alternativa **a**, a mais próxima dentre as opções apresentadas pela para a questão **3**.

Análise e conclusões

No problema 3, é feito uma comparação de performance de “coordenada descendente”. A cada iteração temos dois passos ao longo de duas coordenadas. O passo 1 é mover apenas a coordenada u para reduzir o erro, e o passo 2 é reavaliar e mover apenas a coordenada v para reduzir o erro. O algoritmo retorna (10)^-1 como o valor de erro E(u, v) mais próximo de 15 iterações completas (30 passos). A partir de outros valores, notamos que quanto maior o número de iterações menor será o valor de erro E(u,v). Como exemplo temos o valor 0.019352066243402324 retornado pelo algoritmo a partir de 150 iterações.

3. Regressão Logística

É uma técnica recomendada para situações em que a variável dependente é de natureza dicotômica ou binária. Quanto às independentes, tanto podem ser categóricas ou não. No modelo logístico a variável resposta Y é binária. Uma variável binária assume dois valores, Y = 0 e Y = 1 denominados "fracasso" e "sucesso", respectivamente. Neste caso, "sucesso" é o evento de interesse.

A regressão logística é um recurso que nos permite estimar a probabilidade associada à ocorrência de determinado evento em face de um conjunto de variáveis explanatórias.

Vantagens do Modelo Logístico

- Facilidade para lidar com variáveis independentes categóricas.
- Fornece resultados em termos de probabilidade.
- Facilidade de classificação de indivíduos em categorias.
- Requer pequeno número de suposições.

Função Logística

Na regressão logística, a probabilidade de ocorrência de um evento pode ser estimada diretamente. Pelo fato da variável dependente Y assumir apenas dois possíveis estados (1 ou 0) e haver um conjunto de p variáveis independentes X1 , X2 , ... , Xp , o modelo de regressão logística pode ser escrito da seguinte forma:

$$P(Y=1) = \frac{1}{1+e^{-g(x)}}$$

Figura 12: Equação 8

onde, g(x)= B0 + B1X1 + $\cdots$ + Bp Xp

Os coeficientes B0 , B1 , ... , Bp são estimados a partir do conjunto dados, pelo método da máxima verossimilhança, em que encontra uma combinação de coeficientes que maximiza a probabilidade da amostra ter sido observada.

Considerando uma certa combinação de coeficientes B0 , B1 , ... , Bp e variando os valores de X. Observa-se que a curva logística tem um comportamento probabilístico no formato da letra S, o que é uma característica da regressão logística.

- g (x) → +∞ , então P(Y =1) → 1
- g (x) → −∞ , então P(Y =1) → 0

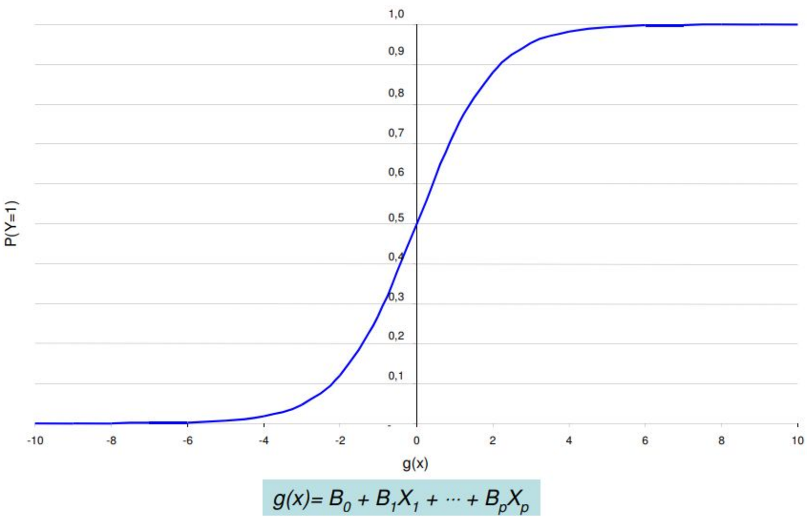

Figura 13: Curva Logística

Para utilizar o modelo de regressão logística para discriminação de dois grupos, a regra de classificação é a seguinte:

- se P(Y=1) > 0,5 então classifica-se Y=1
- se P(Y=1) < 0,5 então classifica-se Y=0

Para obter-se uma boa estimativa da eficiência classificatória do modelo, recomenda-se separar a amostra em duas partes:

- uma parte para estimação do modelo
- outra parte para testar a eficiência da classificação (holdout sample)

Implementação

As soluções foram desenvolvidas em Python, por ser uma linguagem de alto nível e de desenvolvimento rápido (RAD - Rapid Application Development). Foram apresentados dois problemas sobre Regressão Logística e para cada, era necessário apresentar qual das soluções informadas é a correta ou a mais próxima do resultado obtido pelo algoritmo.

Problemas 5 e 6

Neste problema nós vamos criar uma função target própria, f (probabilidade neste caso) e o conjunto de dados D para verificar como a Regressão Logística funciona. Para simplicidade, nós vamos usar f para ser uma probabilidade 0/1, então apenas y é uma função determinística de x. Use d = 2 apenas para você visualizar o problema, e seja $X = [-1, 1] \times [-1, 1]$ com probabilidade uniforme de seleção para cada x pertencente a X. Escolha uma reta no plano com limite entre f(x) = 1 (onde y deve ser +1) e f(x) = 0 (onde y deve ser -1), a partir de dois pontos aleatórios de X e tomando a reta passando por eles como limite entre y = ±1. Use N = 100 pontos aleatórios para treinamento a partir de X , e avalie as saídas yn para cada um desses pontos xn. Execute a Regressão Logística com o Gradiente descendente para encontrar g, e estimar Eout (o erro de entropia cruzada) gerando um conjunto de pontos suficientemente grande e separado para avaliar o erro. Repita o experimento para 100 execuções com diferentes alvos e use a m´edia. Inicialize o vetor de pesos para a Regressão Logística zerado (todos os valores iguais a zero) em cada execução. Pare o algoritmo quando $w(t-1) - w(t) < 0.01$, no qual w(t) representa o vetor de pesos final da época t. Uma época é uma passagem completa pelos N pontos de dados (use uma permutação aleatória de 1, 2, ..., N para representar os pontos de dados do algoritmo dentro de cada época, e uso diferentes permutações para diferentes épocas). Use uma taxa de aprendizagem de 0.01.

4) Qual das seguintes opções é a mais próxima de Eout para N = 100?

a) 0.025

b) 0.050

c) 0.075

d) 0.100

e) 0.125

5) Quantas épocas, em média, são necessárias para a Regressão Logística convergir para N = 100 usando as regras de inicialização e finalização especificadas na taxa de aprendizado? Escolha o valor que mais se aproxima dos seus resultados.

a) 350

b) 550

c) 750

d) 950

e) 1750

Resultados obtidos pelo algoritmo

```
---------------------------------- Resultado Problema 4 ------------------------------------------------
Erro médio de entropia cruzada E_out para 100 execuções:  0.10280774645309988

---------------------------------- Resultado Problema 5 ------------------------------------------------
Número médio de épocas:  341.58
```

Figura 14: Resultados problemas 4 e 5

Para N = 100, o algoritmo retorna **0.10280774645309988**, como o erro médio de entropia cruzada para 100 execuções. Deste modo, é possível concluir que a letra **b**, cuja alternativa apresenta o número **0.100**, é a opção mais próxima do valor alcançado pelo algoritmo, sendo esta, a alternativa correta dentre as opções apresentadas para a questão **4**. Para o cálculo do número médio de épocas, o algoritmo apresentou o valor **341.58**, sendo a alternativa **a**, a mais próxima dentre as opções apresentadas pela para a questão **5**.

Análise e conclusões

Para N = 100, no problema 4 é solicitado qual das opções é a mais próxima de Eout e o algoritmo retorna um valor próximo à 0.100.

No problema 5, com o mesmo valor de N, o algoritmo retorna aproximadamente 350, como número médio de épocas necessárias para a Regressão Logística convergir.

Testando com números distintos de execuções, notamos que seja N menor ou maior que 100, temos um Eout menor um maior número médio de épocas. Abaixo segue uma tabela comparativa para os diferentes valores de N.

N	**E_out**	**Número médio de épocas**
10000	0.1025422370965339	341.175
1000	0.10266356071976776	337.741
100	0.10445268833084967	336.36
10	0.09748318477878554	356.5
1	0.09227565164101396	368.0

4. REFERÊNCIAS

[1] Gradiente Descendente: um método poderoso e flexível para otimização iterativa, Disponível em: <https://matheusfacure.github.io/2017/02/20/MQO-Gradiente-Descendente/>. Acesso em 26 de agosto de 2018.

[2] Método do Gradiente, Disponível em: <https://pt.wikipedia.org/wiki/M%C3%A9todo_do_gradiente>. Acesso em 26 de agosto de 2018.

[3] Lecture 09 - The Linear Model II, Disponível em: <https://www.youtube.com/watch?v=qSTHZvN8hzs>. Acesso em 26 de agosto de 2018.

[4] Lecture 10 - Neural Networks, Disponível em: <https://www.youtube.com/watch?v=qSTHZvN8hzs>. Acesso em 26 de agosto de 2018.

5. ANEXO

Nesta seção são apresentados as implementações devidamente comentadas dos problemas apresentados no trabalho. Os códigos completos estão disponíveis em: https://github.com/isabelsales/Trabalho-pratico-III

Implementação problemas 1 e 2

```python
import math
import numpy as np

e = math.e

def E(u, v):
    return (u * e ** v - 2 * v * e ** (-u)) ** 2

def problem_1_2_trab3():
    #Iterações exigidas
    iteracoes = -1

    #Inicia (u, v) = (1,1)
    x = [1, 1]
    eta = 0.1

    #Criação de um laço de repetição
    for t in range(1, 10 ** 5):

        #Descompactar os valores na lista x
        u, v = x

        #Cálculo de gradiente
        dE_du = 2 * (u * e ** v - 2 * v * e ** (-u)) * (e ** v + 2 * v * e ** (-u))
        dE_dv = 2 * (u * e ** v - 2 * v * e ** (-u)) * (u * e ** v - 2 * e ** (-u))
        grad = np.array([dE_du, dE_dv])

        #Atualiza posições
        x = x - eta * grad

        #Iterações necessárias
        iteracoes = t

        #Armazenar posição atual x como final uv
        final_uv = x

        #Parar se E cair abaixo de 10 ^ (- 14)
        if E(x[0], x[1]) < 10 ** (-14):
            break

    return iteracoes, final_uv

iteracoes, final_uv = problem_1_2_trab3()
print("\n---------------------------------- Resultado Problema 1 ------------------------------------------------")
print("Quantidade de iterações que encontra o erro E(u, v) abaixo de (10)^14 pela primeira vez: ", iteracoes)
print("Final(u,v) = ", final_uv)

# ---------------------------------------------------------------------------------------------------------

#Calcular qual dos seguintes pontos é o mais próximo do final (u, v)
L = [(1.000, 1.000), (0.713, 0.045), (0.016, 0.112), (-0.083, 0.029), (0.045, 0.024)]

min_dist = 2 ** 64
min_ponto = None

print("\n---------------------------------- Resultado Problema 2 ------------------------------------------------")
print("Distância dos pontos dados até a final (u, v):")

for ponto in L:
    x = np.array(ponto)
    distance = np.linalg.norm(final_uv - x)
    print("Ponto x = ", x, " => distância = ", distance)
    if distance < min_dist:
        min_dist = distance
        min_ponto = x

print("\nO ponto com distância mínima até a final (u, v) é: ", min_ponto)
```

Figura 15: Implementação problemas 1 e 2

Código completo da implementação dos problemas 1 e 2, está disponível em: https://github.com/isabelsales/Trabalho-pratico-III/blob/master/solucoes_1%20e_2_trab3.py

Implementação problema 3

```python
import math
import numpy as np

e = math.e

def E(u, v):
    return (u * e ** v - 2 * v * e ** (-u)) ** 2

def problem_3_trab3():
    #Iterações exigidas
    num_iteracoes = 15

    #Inicia (u,v) = (1,1)
    x = [1, 1]
    eta = 0.1

    #Criação de laço de repetição
    for t in range(num_iteracoes):
        #Primeiro passo: Calcular o gradiente na direção u
        u, v = x
        dE_du = 2 * (u * e ** v - 2 * v * e ** (-u)) * (e ** v + 2 * v * e ** (-u))
        grad = np.array([dE_du, 0])

        #Atualiza a posição somente na direção u
        x = x - eta * grad

        #Segundo passo: Calcular o gradiente na direção v
        u, v = x
        dE_dv = 2 * (u * e ** v - 2 * v * e ** (-u)) * (u * e ** v - 2 * e ** (-u))
        grad = np.array([0, dE_dv])

        #Atualiza a posição somente na direção u
        x = x - eta * grad

        #Iterações necessárias
        iteracoes = t

    #Erro final
    erro_final = E(x[0], x[1])
    return erro_final

print("\n---------------------------------- Resultado Problema 3 ------------------------------------------------")
print("Coordenada descendente:")
print("Valor de erro E(u, v) mais próximo após 15 iterações completas:", problem_3_trab3())

# ------------------------------------------------------------------------------------------------------------------
```

Figura 16: Implementação problema 3

Código completo da implementação do problemas 3, está disponível em: https://github.com/isabelsales/Trabalho-pratico-III/blob/master/solucao_3_trab3.py

Implementação problemas 4 e 5

```python
import random
import numpy as np
import math

def problem_4_e_5():
    execucoes = 100
    E_out_total = 0
    epoca_total = 0

    for n in range(execucoes):
        #Conjunto de treino com N = 100 pontos através da linha de separação

        #Escolher dois pontos aleatórios A, B em [-1,1] x [-1,1]
        A = np.random.uniform(-1, 1, 2)
        B = np.random.uniform(-1, 1, 2)

        #A linha descrita por y = m * x + b onde m é a inclinação
        m = (B[1] - A[1]) / (B[0] - A[0])
        b = B[1] - m * B[0]
        w_f = np.array([b, m, -1])

        # ------------------------------------------------------------------------------------------

        #Escolher N pontos de dados (x, y) uniformemente de [-1,1] x [-1,1]
        N = 100
        x1 = np.random.uniform(-1, 1, N)
        x2 = np.random.uniform(-1, 1, N)

        X = np.transpose(np.array([np.ones(N), x1, x2]))  # input

        #Classificação dos pontos
        y_f = np.sign(np.dot(X, w_f))

        # ------------------------------------------------------------------------------------------

        #Executa a Regressão Logistica
        #Inicializar pesos para hipóteses com zeros
        eta = 0.01
        w_g = np.zeros(3)  #Vetor de peso para a hipótese g

        #Inicializa as iterações
        for t in range(10 ** 5):

            #Criação de permutação de pontos de dados
            indices = list(range(N))
            random.shuffle(indices)
            w_old = w_g

            #Laço de repetição: para cada época
            for i in indices:
                xn = X[i, :]  #escolha um ponto
                yn = y_f[i]
                delta_w = -yn * xn / (1 + math.exp(yn * np.dot(w_g.T, xn)))

                #Atualizar w
                w_g = w_g - eta * delta_w

            #Verificar quanto w_g mudou
            #print("t = ", t, "    Diferença de w = ", np.linalg.norm(w_g - w_old))
            if np.linalg.norm(w_g - w_old) < 0.01:
                break

        epoca_total += t


        #1000 pontos de teste para calcular o E_out
        N_test = 1000
        x1_test = np.random.uniform(-1, 1, N_test)  #1000 pontos
        x2_test = np.random.uniform(-1, 1, N_test)
        X_test = np.array([np.ones(N_test), x1_test, x2_test]).T

        y_f_test = np.sign(np.dot(X_test, w_f))  #classificação

        #Calculo do E_out (entropia cruzada)
        E_out = 0
        for i in range(N_test):
            E_out += math.log(1 + math.exp(-y_f_test[i] * np.dot(X_test[i, :], w_g)))

        E_out_total += (E_out / N_test)

    E_out_avg = E_out_total / execucoes
    epoca_avg = epoca_total / execucoes

    return (E_out_avg, epoca_avg)

E_out_avg, epoca_avg = problem_4_e_5()

print("\n--------------------------------- Resultado Problema 4 ---------------------------------------------")
print("Erro médio de entropia cruzada E_out para 100 execuções: ", E_out_avg)

print("\n--------------------------------- Resultado Problema 5 ---------------------------------------------")
print("Número médio de épocas: ", epoca_avg)
```

Figura 17: Implementação problemas 4 e 5

Código completo da implementação dos problemas 4 e 5, está disponível em: https://github.com/isabelsales/Trabalho-pratico-III/blob/master/solucoes_4_e_5.py