Lab 2 - BCC406/PCC177

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Regressão logística

Prof. Eduardo e Prof. Pedro

Aluna: Daniela Costa Terra

Data da entrega : 08/04

- Complete o código (marcado com ToDo) e quando requisitado, escreva textos diretamente nos notebooks. Onde tiver *None*, substitua pelo seu código.
- Execute todo notebook e salve tudo em um PDF nomeado como "NomeSobrenome-Lab2.pdf"
- Envie o PDF via FORM

Classificador Binário com Regressão Logística

Você criará um classificador baseado em regressão logística para reconhecer gatos em imagens.

Dica:

- Evite loops (for / while) em seu código. Isso o tornará mais eficiente.

Notebook para:

- Construir a arquitetura geral de um algoritmo regressão logística, incluindo:
  - Inicializando parâmetros
  - Cálculo da função de custo e seu gradiente
  - Algoritmo de otimização - gradiente descendente

1 - Pacotes

Primeiro, vamos executar a célula abaixo para importar todos os pacotes que precisaremos.

- numpy é o pacote fundamental para a computação científica com Python.

arquivo H5.

- matplotlib é uma biblioteca famosa para plotar gráficos em Python.
- PIL e scipy são usados aqui para carregar as imagens e testar seu modelo final.

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
```

```
%matplotlib inline
```

```
# Você vai precisar fazer o upload dos arquivos no seu drive (faer na pasta raiz) e montá-l
# não se esqueça de ajustar o path para o seu drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

2 - Visão geral do problema

Classificar as imagens com gato ou sem-gato.

Conjunto de dados ("data.h5") contem:

- um conjunto de imagens para treinamento, rotuladas como gato (y = 1) ou sem-gato (y = 0)
- um conjunto de imagens de testes, rotuladas como gato ou sem-gato
- cada imagem tem a forma (num_px, num_px, num_ch), em que num_ch é relativos aos canais de cores (RGB) e deve ser fixado em 3. Assim, cada imagem é quadrada (altura = num_px) e (largura = num_px).

Carregue os dados executando o seguinte código.

```
# Lendo os dados (gato/não-gato)
def load_dataset():

  train_dataset = h5py.File('/content/drive/MyDrive/disciplinasDoutorado/PCC177-2022-1(Rede

  train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
  train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels
```

```
  test_dataset = h5py.File('/content/drive/MyDrive/disciplinasDoutorado/PCC177-2022-1(Redes
  test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
  test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

  classes = np.array(test_dataset["list_classes"][:]) # the list of classes
  train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
  test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

  return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes
```

```
treino_x_orig, treino_y, teste_x_orig, teste_y, classes = load_dataset()
print(classes)
print(treino_x_orig.shape, treino_y.shape, teste_x_orig.shape, teste_y.shape)

linhas = treino_x_orig.shape[0]
treino_x_vet_orig = treino_x_orig.reshape(linhas,(64*64*3))
print(treino_x_vet_orig.shape, treino_x_vet_orig.T.shape)
```

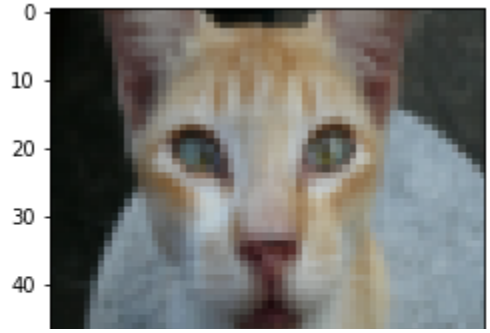
```
[b'non-cat' b'cat']
(209, 64, 64, 3) (1, 209) (50, 64, 64, 3) (1, 50)
(209, 12288) (12288, 209)
```

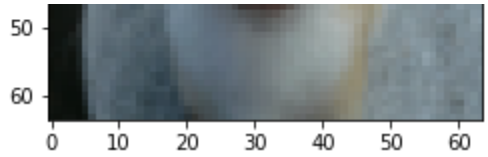
O termo `_orig` no final dos conjuntos de dados (treino e teste) significa que estamos tratando com os dados lidos originalmente. Após o pré-processamento, atribuiremos a outros objetos (treino_x e teste_x).

Cada linha de treino_x_orig e teste_x_orig é uma matriz que representa uma imagem. Você pode visualizar um exemplo executando o seguinte código.

```
# Exemplo
index = 11
plt.imshow(treino_x_orig[index])
print(np.squeeze(treino_y[:, index]), classes[0], classes[1])
print ("y = " + str(treino_y[:, index]) + ", it's a '" + classes[np.squeeze(treino_y[:, inc
```

```
1 b'non-cat' b'cat'
y = [1], it's a 'cat' picture.
```





Exercício: Encontre os valores para: (0,5pt)

- m_treino (número de exemplos de treinamento)
- m_teste (número de exemplos de teste)
- num_px (altura = largura de uma imagem de treinamento)

dica: você tem estes valores nas dimensões dos tensores treino_x_orig e treino_y_orig

```
### Início do código ### (≈ 3 linhas)
#ToDo : implemente o bloco
m_treino = treino_x_orig.shape[0]  # ToDo
m_teste =teste_x_orig.shape[0] # ToDo
num_px = teste_x_orig.shape[1] # ToDo
### Fim do código ###

print ("Número de exemplos de treinamento: m_treino = " + str(m_treino))
print ("Número de exemplos de teste: m_teste = " + str(m_teste))
print ("Altura/largura de cada imagem: num_px = " + str(num_px))
print ("Tamanho de cada imagem: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("formato de treino_x: " + str(treino_x_orig.shape))
print ("formato de treiro_y: " + str(treino_y.shape))
print ("formato de teste_x: " + str(teste_x_orig.shape))
print ("formato de teste_y: " + str(teste_y.shape))
```

```
Número de exemplos de treinamento: m_treino = 209
Número de exemplos de teste: m_teste = 50
Altura/largura de cada imagem: num_px = 64
Tamanho de cada imagem: (64, 64, 3)
formato de treino_x: (209, 64, 64, 3)
formato de treiro_y: (1, 209)
formato de teste_x: (50, 64, 64, 3)
formato de teste_y: (1, 50)
```

Valores esperados para m_treino, m_teste and num_px:

m_treino	209
m_teste	50
num_px	64

3 - Pré-processamento

3.1 - Formatação (0,5pt)

Por conveniência, vamos "**vetorizar**" as imagens para que elas fiquem nas dimensões: (num_px $*$ num_px $*$ 3, 1). Depois disso, nosso conjunto de dados de treinamento (e teste) será uma matriz ndarray(numpy) em que cada coluna representa uma imagem vetorizada. Deve haver m_treino colunas. O mesmo para o conjunto de teste (m_teste colunas)

Exercício: Formate os conjuntos de dados de treinamento e teste para que as imagens de tamanho (num_px, num_px, 3) sejam vetores de forma (num_px $*$ num_px $*$ 3, 1).

dica: ver documentação da função reshape(..)

```
# Formate o conjunto de treinamento e teste

### Início do código ### (≈ 2 linhas)
#ToDo : implemente o bloco
treino_x_vet = treino_x_orig.reshape(m_treino,(64*64*3))# ToDo
treino_x_vet = treino_x_vet.T

teste_x_vet = teste_x_orig.reshape(m_teste,(64*64*3)) # ToDo
teste_x_vet = teste_x_vet.T
### Fim do código ###

print ("Formato de treino_x_vet: " + str(treino_x_vet.shape))
print ("Formato de treino_y: " + str(treino_y.shape))
print ("Formato de teste_x_vet: " + str(teste_x_vet.shape))
print ("Formato de teste_y: " + str(teste_y.shape))
```

```
Formato de treino_x_vet: (12288, 209)
Formato de treino_y: (1, 209)
Formato de teste_x_vet: (12288, 50)
Formato de teste_y: (1, 50)
```

3.2 - Normalização (0,5pt)

As imagens do conjunto de dados são repreesentadas por canais (RGB). Os canais vermelho, verde e azul devem ser especificados para cada pixel e, portanto, o valor do pixel é na verdade um vetor de três números que podem variar de 0 a 255.

Uma etapa comum de pré-processamento no aprendizado de máquina é centralizar e normalizar seu conjunto de dados, que significa subtrair cada exemplo pela média e dividir pelo desvio padrão de toda a matriz (de dados de treino ou de teste). Porém, para conjuntos de dados de imagens, é mais simples e conveniente e funciona muito bem apenas dividir todas as linhas do

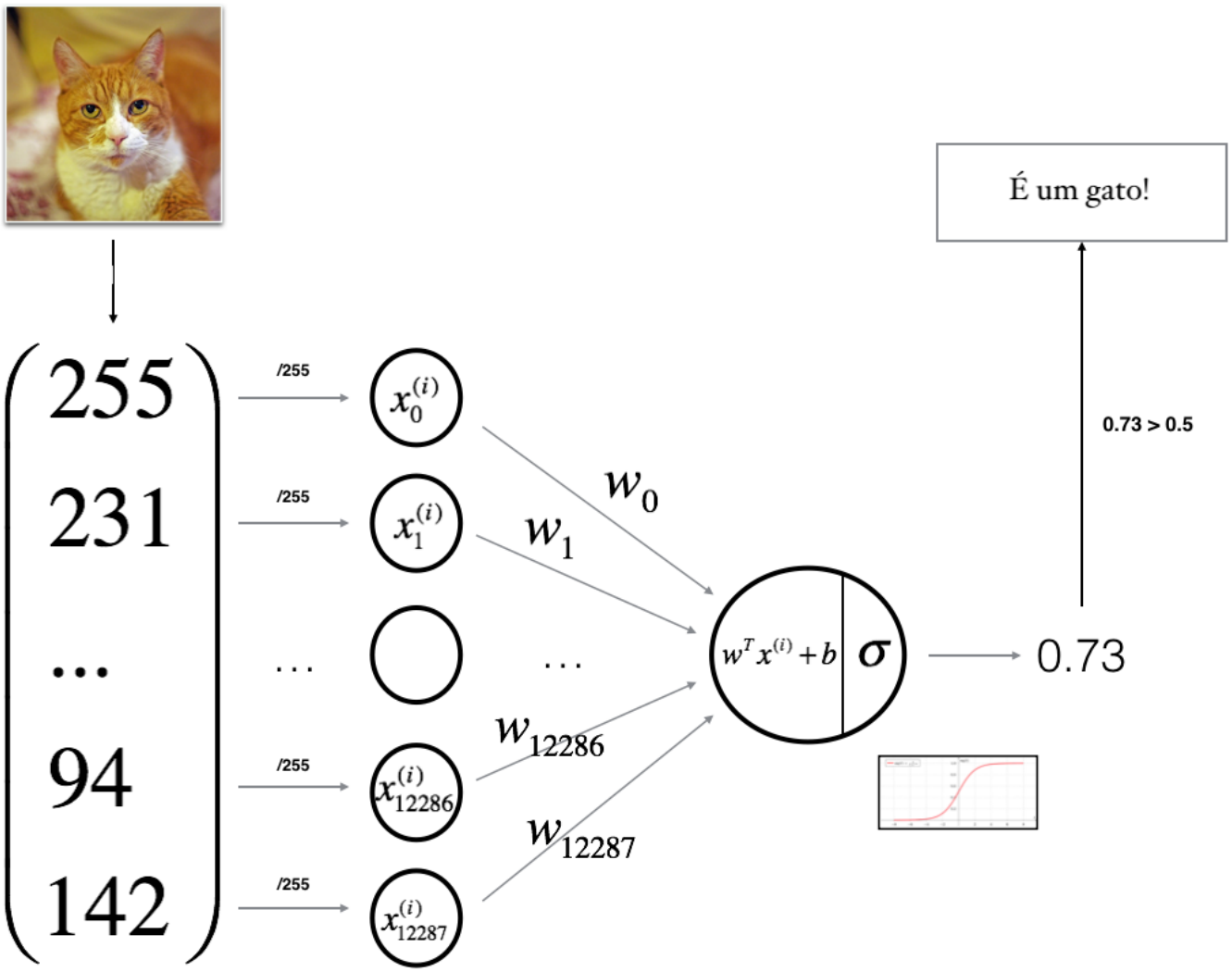
conjunto de dados por 255 (o valor máximo).

Vamos normalizar o conjunto de dados, deixando os valores dos pixels entre 0 e 1.

```
# Normaliza os dados

#ToDo : implemente o bloco
treino_x = treino_x_vet/255 # ToDo
teste_x = teste_x_vet/255 # ToDo
```

4 - Arquitetura geral do algoritmo de aprendizado

A figura a seguir explica por que **a regressão logística é realmente uma rede neural muito simples!**



Expressão matemática do algoritmo:

Para um exemplo $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b \tag{1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$

$$y^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \tag{3}$$

O custo é então calculado somando sobre todos os exemplos do treinamento:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{6}$$

Etapas principais:

Neste exercício, você executará as seguintes etapas:

- Inicializar os parâmetros do modelo
- Aprender os parâmetros do modelo, minimizando o custo
- Use os parâmetros aprendidos para fazer a predição (no conjunto de testes)
- Analisar os resultados.

4.1 - Funções auxiliares (1pt)

Exercício: Implemente a função de ativação `sigmoid()`. Como você viu na figura acima, você precisa calcular $sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$ para fazer previsões.

dica: você pode usar a função exponencial do numpy (np.exp(-z))

```
# Função de ativação sigmoid

def sigmoid(z):
    """
    Calcula a sigmoid de z
    entrada:
    z -- Um escalar ou um numpy array de qualquer tamanho.

    saída:
    s -- sigmoid(z)
    """
    ### Início do código ### (≈ 1 linha)
    #ToDo : implemente o bloco
    s = 1/(1+(np.exp(-1*z))) # ToDo
    ### Fim do código ###
    return s


# Teste
print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
```

```
sigmoid([0, 2]) = [0.5        0.88079708]
```

Valor esperado:

sigmoid([0, 2]) [0.5 0.88079708]

4.2 - Inicializando Parâmetros (1pt)

Exercício: Implemente a inicialização dos parâmetros w e b.

dica: veja função np.zeros(..)

```
# Função: inicializa w e b

def inicialize(dim):
    """
    Inicializa um vetor de tamanho (dim, 1) para w and  b = 0.

    Entrada:
    dim -- tamanho de w (número de parâmetros)

    Saída:
    w -- tamanho (dim, 1)
    b -- um escalar (correspondente ao bias)
    """

    ### Início do código ### (≈ 1 line of code)
    #ToDo : implemente o bloco
    w = np.zeros((dim, 1)) # ToDo
    b = 0 # ToDo

    ### Fim do código ###

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b


# Teste

dim = 2
w, b = inicialize(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

```
     w = [[0.]
      [0.]]
     b = 0
```

Valores esperados:

Valores esperados:

** w **	[[0.] [0.]]
** b **	0

4.3 - Forward and Backward propagation (2pt)

Exercício: Implemente a função `propagacao ()` que calcula a função de custo e seu gradiente.

Forward-Propagation:

- Entrada X
- Calcule a ativação $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$
- Calcule a função de custo: $J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Backward-propagation:

Fórmulas do gradiente:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T \tag{7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{8}$$

```
# Forward and Backward propagation

def propagacao(w, b, X, Y):
    """
    Implementa a função custo e seus gradientes

    Entrada:
    w -- pesos, de tamanho (num_px * num_px * 3, 1)
    b -- bias, um escalar
    X -- dados de treinamentos de tamanho (num_px * num_px * 3, número de exemplos)
    Y -- ( 0 se não-gato, 1 se gato) de tamanho (1, número de exemplos)

    Saída:
    custo -- custo para regressão logística
    dw -- gradiente da função loss em relação a w
    db -- gradiente da função loss em relação a b


    """

    #ToDo : implemente a função
    Y = np.array(Y) # converte para o tipo ndarray para acessar o .shape do objeto
    m = Y.shape[1]  # número de exemplos

    # FORWARD PROPAGATION
```

```
    ### Início do código ### (≈ 4 a 5 linhas)
    #ToDo : implemente o bloco
    w = np.array(w)
    X = np.array(X)
    o = (w.T.dot(X) + b) # calcula o = w.T * X + b
    A = sigmoid(o) # calcula ativação, dica use sua função sigmoid
    custo =  -1/m * np.sum(Y*np.log(A) + (1 - Y)*np.log(1 - A)) # calcula custo. Dica : use
    ### Fim do código ###

    # BACKWARD PROPAGATION
    ### Início do código ### (≈ 2 linhas)
    #ToDo : implemente o bloco
    dw = 1/m * (X.dot((A - Y).T)) # dica: use np.dot(..) e não esqueça que em algumas opera
    db = 1/m * np.sum(A - Y)
    ### Fim do código ###

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    custo = np.squeeze(custo)
    assert(custo.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, custo


# Teste
w, b, X, Y = np.array([[1.],[2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-3.2]]), np.array([[1,
grads, custo = propagacao(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("custo = " + str(custo))
```

```
dw = [[0.99845601]
 [2.39507239]]
db = 0.001455578136784208
custo = 5.801545319394553
```

Valores esperado:

** dw **	[[0.99845601] [2.39507239]]
** db **	0.00145557813678
** custo **	5.801545319394553

4.4 - Otimização (2pt)

- O processo de atualização dos parâmetros é realizado pelo algoritmo da descida do gradiente.

Exercício: Atualize w e b, minimizando a função de custo J. Para um parâmetro θ, a regra de atualização é $\theta = \theta - \alpha\ d\theta$, em que α é a taxa de aprendizado.

```
# Algoritmo da descida do gradiente

def gradiente_descendente(w, b, X, Y, num_iter, learning_rate, print_custo = False):
    """
    Esta função atualiza/otimiza os parâmetros w e b através do algoritmo do graidente

    Entrada:
    w -- pesos, de tamanho (num_px * num_px * 3, 1)
    b -- bias, um escalar
    X -- dados de treinamentos de tamanho (num_px * num_px * 3, número de exemplos)
    Y -- ( 0 se não-gato, 1 se gato) de tamanho (1, número de exemplos)
    num_iter -- número de interações
    learning_rate -- taxa de aprendizegem do algoritmo gradiente descendente
    print_custo -- print flag

    Saída:
    params -- dictionário contendo os pesos w e bias b
    grads -- dictionário contendo os gradientes dos pesos w e bias b com relação a função c
    custos -- lista de todos os custos durante a otimização, será usado para plotar a curva


    """

    custos = []

    for i in range(num_iter):


        # Calcula o custo e os gradientes (≈ 1-4 linhas)
        ### Início do código ###
        #ToDo : implemente o bloco
        grads, custo = propagacao(w, b, X, Y)   # dica: use sua funcao propagacao(..)
        ### Fim do código ###

        # Recupera os gradientes do dicionário grads
        #ToDo : implemente o bloco
        dw = grads["dw"]      # dica: fique atento ao tipo de dados de grads para acessar c
        db = grads["db"]

        # Atualiza w e b (≈ 2 linhas)
        ### Início do código ###
        #ToDo : implemente o bloco, lembrando sempre da taxa de aprendizagem (learning_rate
        w = w - learning_rate*dw
        b = b - learning_rate*db
        ### Fim do código ###
```

```
        # Guarda custos
        if i % 100 == 0:
            custos.append(custo)

        # Imprime o custo a cada 100 interações
        if print_custo and i % 100 == 0:
            print (f"Custo após {i:4d} iterações: {custo:.4f}")

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, custos


# Teste
params, grads, custos = gradiente_descendente(w, b, X, Y, num_iter= 100, learning_rate = 0.

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

```
w = [[0.19033591]
 [0.12259159]]
b = 1.9253598300845747
dw = [[0.67752042]
 [1.41625495]]
db = 0.21919450454067657
```

Valores esperados:

w	[[0.19033591] [0.12259159]]
b	1.92535983008
dw	[[0.67752042][1.41625495]]
db	0.219194504541

4.5 - Predição (2pt)

- Depois do aprendizado dos parâmentros w e b, eles são usados para predizer os rótulos para um conjunto de dados X.

Exercício:. Implemente a função `predicao ()`:

1. Calcule $\hat{Y} = A = \sigma(w^T X + b)$
2. Converta $\hat{Y}$ em 0 (se ativação <= 0,5) ou 1 (se ativação > 0,5).

```
# GRADED FUNCTION: predição

def predicao(w, b, X):
    '''
    Prediz se o rótulo é 0 ou 1 usando os parâmetros de aprendizagem (w,b) da regressão log

    Entrada:
    w -- pesos, de tamanho (num_px * num_px * 3, 1)
    b -- bias, um escalar
    X -- dados de treinamentos de tamanho (num_px * num_px * 3, número de exemplos)

    Saída:
    Y_pred -- um vetor contendo todas as predições (0/1) para os dados X
    '''

    #ToDo : implemente a função

    m = X.shape[1]       # número de exemplos. Dica: acesso o shape de X e veja qual valor
    Y_pred = np.zeros((1,m))  # inicialize o vetor de predições. dica: ver  np.zeros()


    # Calcule o vetor "A" da probabilidade de um gato estar na imagem
    ### Início do código ### (≈ 1 lnha)
    #ToDo : implemente o bloco
    A = sigmoid(w.T.dot(X) + b)  # dica: mesma ideia da função propagacao(..)
    ### Fim do código ###

    # Converta as proobabilidades A[0,i] para predição p[0,i]
    ### Início do código ### (≈ 1 a 4 linhas)
    #ToDo : implemente o bloco
    # dica: considere, no vetor A, valores maiores ou iguais a 0.5 como classe 1 e menores
    # e coloque o resultado no vetor Y_pred

    Y_pred = np.zeros((A.shape))
    positive_pred_index = A > 0.5
    Y_pred[positive_pred_index] =  1
    ### Fim do código ###


    assert(Y_pred.shape == (1, m))

    return Y_pred


# Teste
w = np.array([[0.1124579],[0.23106775]])
b = -0.3
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]])
print ("predições = " + str(predicao(w, b, X)))
```

```
predições = [[1. 1. 0.]]
```

Valor esperado:

predições [[1. 1. 0.]]

5 - Construa o modelo com as funções anteriores

Exercício: Construa um modelo. Use a seguinte notação:

```
- Y_pred_teste para suas previsões no conjunto de testes
- Y_pred_treino para suas previsões no treino
- w, custos, grads para as saídas do algoritmo `gradiente()`.
```

```
# Modelo

def modelo(X_treino, Y_treino, X_teste, Y_teste, num_iter = 5000, learning_rate = 0.5, prin
    """
    Cria o modelo de regressão logística chamando as funções auxiliares

    Entradas:
    X_treino -- conjunto de treinamento representado por uma matriz numpy da forma (num_px
    Y_treino -- rótulos de treinamento representados por uma matriz numpy (vetor) da forma
    X_teste -- conjunto de teste representado por uma matriz numpy da forma (num_px * num_p
    Y_teste -- rótulos de teste representados por uma matriz numpy (vetor) da forma (1, m_t
    num_iter -- hiperparâmetro que representa o número de iterações para otimizar os parâme
    learning_rate -- hiperparâmetro que representa a taxa de aprendizado usada na regra de
    print_custo -- Defina como true para imprimir o custo a cada 100 iterações

    Saída:
    d -- dicionário contendo informações sobre o modelo.
    """

    ### Início do código ###
    #ToDo : implemente a função e complete os blocos abaixo

    # iniciliza os parâmetros (≈ 1 linha). Use sua funcao de inicialização e coloque o reto
    w, b = inicialize(X_treino.shape[0])

    # Gradiente descendente (≈ 1 linha). Use sua função  gradiente_descendente e preencha a
    parametros, grads, custos = gradiente_descendente(w, b, X_treino, Y_treino, num_iter, l

    # Recupere os parâmetros w e b do dicionário "parametros"
    w = parametros["w"]
    b = parametros["b"]
```

```
    # Compute predicoes para os conjuntos treino e teste (≈ 2 linhas). Use sua função predi
    Y_pred_teste = predicao(w, b, X_teste)
    Y_pred_treino = predicao(w, b, X_treino)

    ### Fim do código ###

    # Imprime erros do treino/teste
    print("treino acurácia:{} %".format(100 - np.mean(np.abs(Y_pred_treino - Y_treino)) * 1
    print("teste acurácia: {} %".format(100 - np.mean(np.abs(Y_pred_teste - Y_teste)) * 100


    d = {"custos": custos,
         "Y_pred_teste": Y_pred_teste,
         "Y_pred_treino" : Y_pred_treino,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iter": num_iter}

    return d
```

6 - Execute o modelo

Execute a célula a seguir para treinar seu modelo.

```
d = modelo(treino_x, treino_y, teste_x, teste_y, num_iter = 3000, learning_rate = 0.01, pri
```

```
 Custo após    0 iterações: 0.6931
 Custo após  100 iterações: 0.8239
 Custo após  200 iterações: 0.4189
 Custo após  300 iterações: 0.6173
 Custo após  400 iterações: 0.5221
 Custo após  500 iterações: 0.3877
 Custo após  600 iterações: 0.2363
 Custo após  700 iterações: 0.1542
 Custo após  800 iterações: 0.1353
 Custo após  900 iterações: 0.1250
 Custo após 1000 iterações: 0.1165
 Custo após 1100 iterações: 0.1092
 Custo após 1200 iterações: 0.1028
 Custo após 1300 iterações: 0.0971
 Custo após 1400 iterações: 0.0920
 Custo após 1500 iterações: 0.0875
 Custo após 1600 iterações: 0.0833
 Custo após 1700 iterações: 0.0795
 Custo após 1800 iterações: 0.0760
 Custo após 1900 iterações: 0.0728
 Custo após 2000 iterações: 0.0699
 Custo após 2100 iterações: 0.0671
```

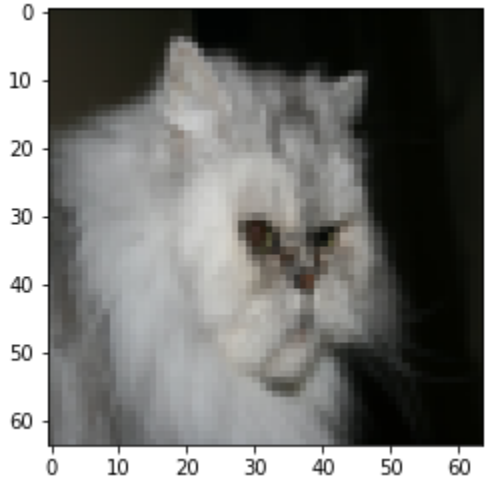
```
Custo após 2200 iterações: 0.0646
Custo após 2300 iterações: 0.0622
Custo após 2400 iterações: 0.0601
Custo após 2500 iterações: 0.0580
Custo após 2600 iterações: 0.0561
Custo após 2700 iterações: 0.0543
Custo após 2800 iterações: 0.0526
Custo após 2900 iterações: 0.0510
treino acurácia:100.0 %
teste acurácia: 68.0 %
```

Valores esperados:

**Custo depois da iteração 0 **	0.693147
$\vdots$	$\vdots$
Acurácia no treino	100 %
Acurácia no teste	68.0 %

```
# Exemplos das predições
index = 10

plt.imshow(teste_x[:,index].reshape((num_px, num_px, 3)))
print (f'y = {classes[teste_y[0,index]].decode("utf-8")}({teste_y[0,index]}), o modelo prec
```

```
y = cat(1), o modelo predizeu que é um "non-cat" picture.
```



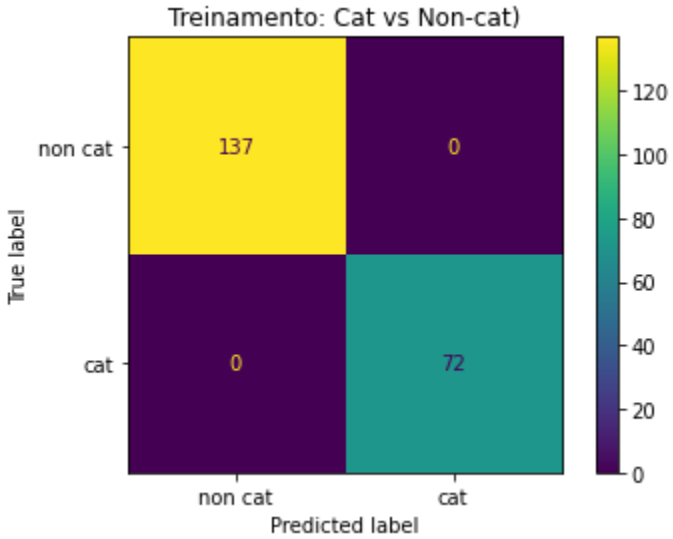
Análisando resultados do modelo:

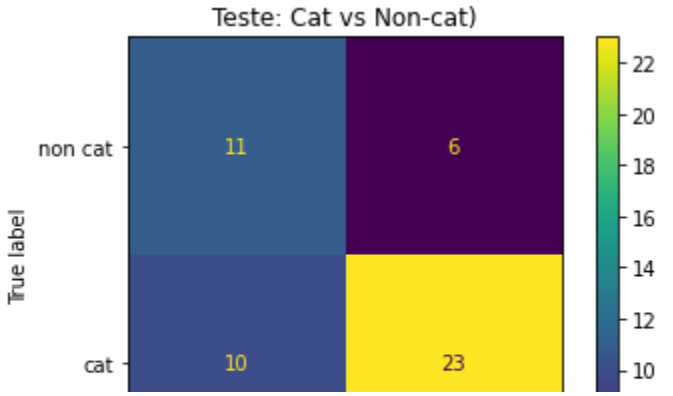
```
# (Daniela) Análise dos dados de treinamento:
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

n_cats_treino = np.sum(treino_y)
print(f" - Total de exemplos para treinamento da classe cat {n_cats_treino}")
print(f" - Total de exemplos para treinamento da classe not_cat {m_treino - n_cats_treino}'
print(f' - Pesos da rede treinada: {np.squeeze(d["w"])}')
```

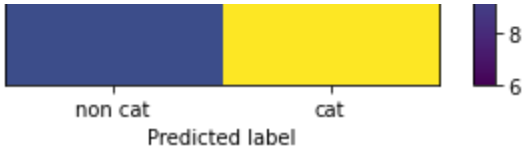
```
print(f' - Bias da rede treinada: {d["b"]}\n')

# Exibe matrizes de confusão para dados de treino e teste:
cm = confusion_matrix(treino_y[0][:], d["Y_pred_treino"][0][:], labels=[0, 1])
disp_treino = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["non cat", "cat"]

cm = confusion_matrix(teste_y[0][:], d["Y_pred_teste"][0][:], labels=[0, 1])
disp_teste = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["non cat", "cat"]

plt.figure(figsize=(4, 2))
disp_treino.plot()
plt.title("Treinamento: Cat vs Non-cat)")
disp_teste.plot()
plt.title("Teste: Cat vs Non-cat)")
plt.show()
plt.tight_layout()
```

```
 - Total de exemplos para treinamento da classe cat 72
 - Total de exemplos para treinamento da classe not_cat 137
 - Pesos da rede treinada: [ 0.0161983  -0.04201861 -0.01844923 ... -0.01787753 -0.04
  0.04123042]
 - Bias da rede treinada: -0.0036565102180048897
```

```
<Figure size 288x144 with 0 Axes>
```








```
<Figure size 432x288 with 0 Axes>
```

Visualizando imagens classe Cat do conjunto de treinamento:

```
# (Daniela) Alguns exemplos da classe cat do conjunto de treinamento:
print("\nExemplos da classe cat do conjunto de treino\n")
is_cat = (treino_y == 1)
is_cat_indexes = np.where(is_cat)
# captura indexes de linha e coluna na tupla:
lin_index = is_cat_indexes[0]
col_index = is_cat_indexes[1]

rows = 3
cols = 6
fig = plt.figure(figsize=(19, 12))
"""for i in range (0, rows*cols):
      fig.add_subplot(rows, cols, i+1)
      plt.imshow(treino_x_orig[col_index[i]])
plt.tight_layout()
plt.show()
"""
```

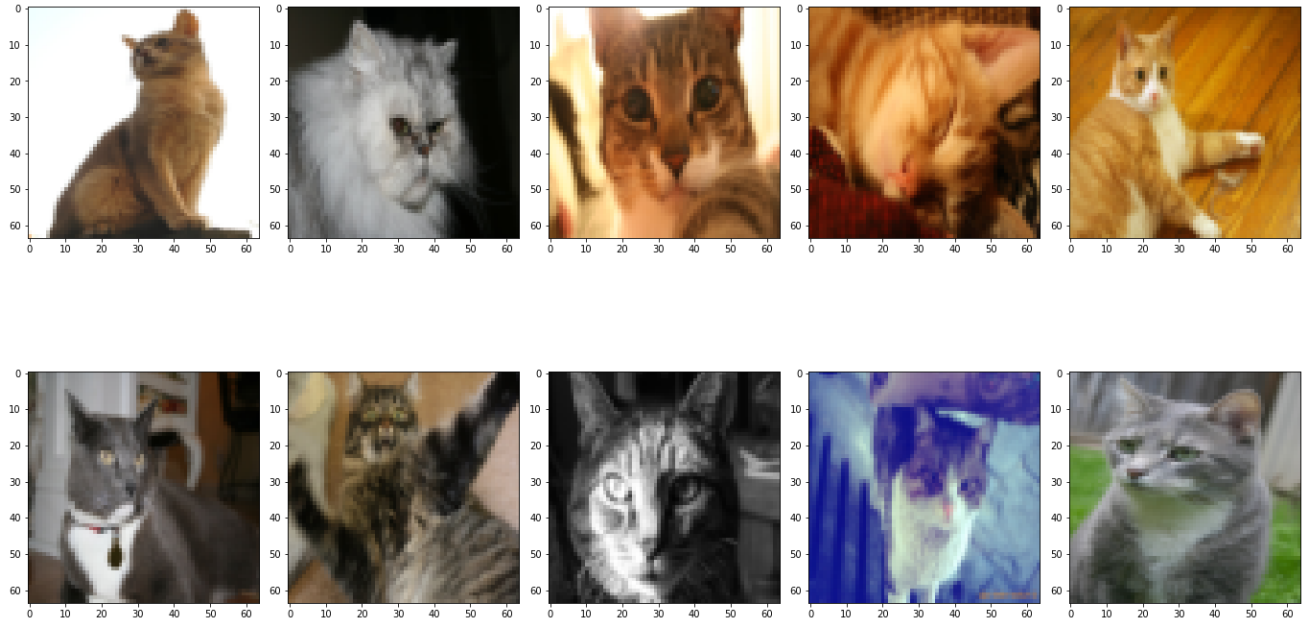
```
Exemplos da classe cat do conjunto de treino

'for i in range (0, rows*cols):\n      fig.add_subplot(rows, cols, i+1)\n      plt.i
mshow(treino_x_orig[col_index[i]])\nplt.tight_layout()  \nplt.show()\n'
```

Casos de `Falso Negativo` no teste (gatos classificados como não-gatos):

```
# (Daniela) Visualizando exemplos da classe cat do conjunto de testes (classificados incorr
print("\nResultados do teste: falsos negativos\n")
erros = teste_y[0][:] - d["Y_pred_teste"][0][:]
false_negative_cat = (erros == 1)
FN_cats_indexes = np.where(false_negative_cat)
# captura indexes de linha e coluna na tupla:
lin_index = FN_cats_indexes[0]

rows = 2
cols = 5
fig = plt.figure(figsize=(19, 12))
for i in range (0, rows*cols):
```

```
for i in range (0, rows*cols):
    fig.add_subplot(rows, cols, i+1)
    plt.imshow(teste_x_orig[lin_index[i]])
fig.tight_layout()
plt.show()
```

```
Resultados do teste: falsos negativos
```

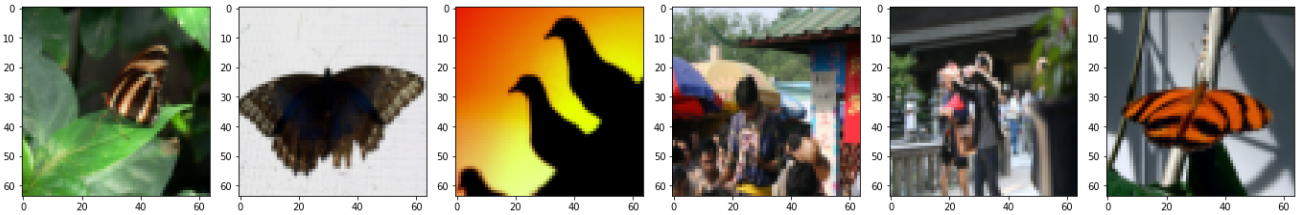


Casos `Falso Positivos` no teste (classificados como gatos):

```
# (Daniela) Visualizando exemplos da classe cat do conjunto de testes (classificados incorr
print("\nResultados do teste: classe non cat classificados incorretamento como cat\n")
erros = teste_y[0][:] - d["Y_pred_teste"][0][:]
false_positive_cat = (erros == -1)
FP_cats_indexes = np.where(false_positive_cat)
# captura indexes de linha e coluna na tupla:
```

```
# captura indexes de linha e coluna na tupla:
lin_index = FP_cats_indexes[0]

rows = 1
cols = 6
fig = plt.figure(figsize=(19, 12))
for i in range (0, rows*cols):
      fig.add_subplot(rows, cols, i+1)
      plt.imshow(teste_x_orig[lin_index[i]])
fig.tight_layout()
plt.show()
```

```
Resultados do teste: classe non cat classificados incorretamento como cat
```



Responda (0,5pt): A acurácia no treinamento é próxima de 100%. Seu modelo está funcionando e tem capacidade alta o suficiente para ajustar os dados de treinamento. A acurácia no teste é de 68%. Porque tanta diferença?

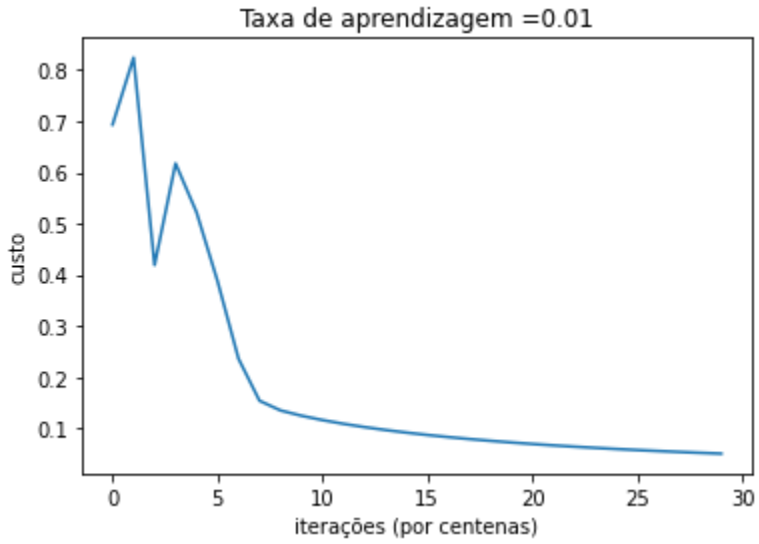
- A acurácia alta no treino indica que a sigmoide encontrada se adaptou muito bem aos dados de treinamento (overfiting) como se o modelo 'decorasse' a resposta. O mesmo modelo não generalizou bem conforme a matriz de confusão exibida acima e a acurácia baixa de 68% para o conjunto de imagens de teste.

O baixo rendimento levanta a dúvida sobre se a quantidade e mesmo a representatividade dos dados usados para treinamento foi suficiente. Outro aspecto diz respeito à arquitetura da rede. Com apenas 1 neurônio é surpreendente o sobreajuste para um modelo simplificado dada a complexidade do problema. Pois, pelas imagens (e como o modelo não aplica filtros a mesma) a mera visualização de respostas incorretas (casos de falsos positivos e falsos negativos) e de algumas imagens de treinamento não ajudou evidenciar uma explicação para o baixo rendimento nos testes. Deve-se considerar também que o treinamento usou uma base contendo 72 exemplares cat e 137 non-cats. Seria necessário balancear a base de treino, para ter o mesmo número de casos de cada classe, e melhorar a arquitetura da rede tendo em vista a complexidade do problema (variabilidade das imagens).

complexidade do problema (variabilidade das imagens).

Plota a função custo e os gradientes

```
# Plot learning curve (with costs)
custos = np.squeeze(d['custos'])
plt.plot(custos)
plt.ylabel('custo')
plt.xlabel('iterações (por centenas)')
plt.title("Taxa de aprendizagem =" + str(d["learning_rate"]))

plt.show()
custos[-1]
```




```
0.05098963863548528
```

Interpretação: Você pode ver o custo diminuindo. Isso mostra que os parâmetros estão sendo aprendidos. No entanto, você pode treinar o modelo ainda mais no conjunto de treinamento. Tente aumentar o número de iterações e execute novamente. O que acontece?

ToDo : discorra sobre a pergunta.

Para mais iterações no treinamento o modelo continua aprendendo. O teste abaixo mostra que a função de custo para 6000 ou mais iterações cai de 0.05 para 0.0077, ao final de 22.000 execuções da função gradiente de otimização.

A acurácia do teste aumentou para 72% ao reduzir de 6 para 4 os casos de falsos positivos. A melhoria na aprendizagem foi como esperado na classificação de não-gatos uma vez que o número de exemplares desta classe é maior no treinamento.

```
# Daniela (execuções com número maior de épocas)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

d = modelo(treino_x, treino_y, teste_x, teste_y, num_iter = 22000, learning_rate = 0.01, pr
# Plot learning curve (with costs)
custos = np.squeeze(d['custos'])
plt.plot(custos)
plt.ylabel('custo')
plt.xlabel('iterações (por centenas)')
plt.title("Taxa de aprendizagem =" + str(d["learning_rate"]))
plt.show()

print(f' - Pesos da rede treinada: {np.squeeze(d["w"])}')
print(f' - Bias da rede treinada: {d["b"]}')

# Exibe matrizes de confusão para dados de teste:
cm = confusion_matrix(teste_y[0][:], d["Y_pred_teste"][0][:], labels=[0, 1])
disp_teste = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["non cat", "cat"])

plt.figure(figsize=(4, 2))
disp_treino.plot()
plt.title("Treinamento - classes: Cat vs Non-cat)")
disp_teste.plot()
plt.title("Teste - classes: Cat vs Non-cat)")
plt.show()
plt.tight_layout()
```

```
Custo após    0 iterações: 0.6931
Custo após  100 iterações: 0.8239
Custo após  200 iterações: 0.4189
Custo após  300 iterações: 0.6173
Custo após  400 iterações: 0.5221
Custo após  500 iterações: 0.3877
Custo após  600 iterações: 0.2363
Custo após  700 iterações: 0.1542
Custo após  800 iterações: 0.1353
Custo após  900 iterações: 0.1250
Custo após 1000 iterações: 0.1165
Custo após 1100 iterações: 0.1092
Custo após 1200 iterações: 0.1028
Custo após 1300 iterações: 0.0971
Custo após 1400 iterações: 0.0920
Custo após 1500 iterações: 0.0875
Custo após 1600 iterações: 0.0833
Custo após 1700 iterações: 0.0795
Custo após 1800 iterações: 0.0760
Custo após 1900 iterações: 0.0728
Custo após 2000 iterações: 0.0699
Custo após 2100 iterações: 0.0671
Custo após 2200 iterações: 0.0646
Custo após 2300 iterações: 0.0622
Custo após 2400 iterações: 0.0601
```

```
Custo após 2500 iteraçoes: 0.0580
Custo após 2600 iterações: 0.0561
Custo após 2700 iterações: 0.0543
Custo após 2800 iterações: 0.0526
Custo após 2900 iterações: 0.0510
Custo após 3000 iterações: 0.0495
Custo após 3100 iterações: 0.0481
Custo após 3200 iterações: 0.0467
Custo após 3300 iterações: 0.0454
Custo após 3400 iterações: 0.0442
Custo após 3500 iterações: 0.0431
Custo após 3600 iterações: 0.0420
Custo após 3700 iterações: 0.0410
Custo após 3800 iterações: 0.0400
Custo após 3900 iterações: 0.0390
Custo após 4000 iterações: 0.0381
Custo após 4100 iterações: 0.0372
Custo após 4200 iterações: 0.0364
Custo após 4300 iterações: 0.0356
Custo após 4400 iterações: 0.0349
Custo após 4500 iterações: 0.0341
Custo após 4600 iterações: 0.0334
Custo após 4700 iterações: 0.0328
Custo após 4800 iterações: 0.0321
Custo após 4900 iterações: 0.0315
Custo após 5000 iterações: 0.0309
Custo após 5100 iterações: 0.0303
Custo após 5200 iterações: 0.0298
Custo após 5300 iterações: 0.0292
Custo após 5400 iterações: 0.0287
Custo após 5500 iterações: 0.0282
```

6 - Mais análises

Escolha da Taxa de aprendizagem

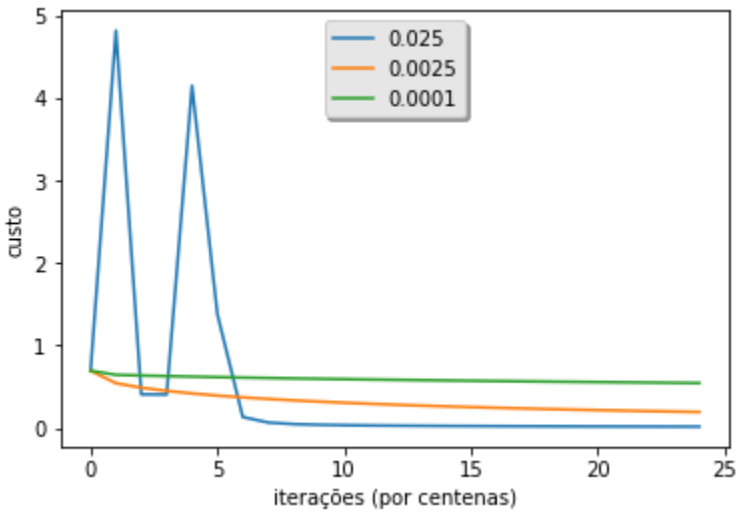
Lembrete: O algoritmo da descida do gradiente, depende da escolha da taxa de aprendizado. A taxa de aprendizado α determina a rapidez com que atualizamos os parâmetros. Se a taxa de aprendizado for muito alta, podemos "ultrapassar" o valor ideal. Da mesma forma, se for muito pequeno, precisaremos de muitas iterações para convergir para os melhores valores. É por isso que é crucial usar uma taxa de aprendizado bem ajustada.

Vamos comparar a curva de aprendizado do modelo com várias opções de taxas de aprendizado. Execute a célula abaixo.

```
learning_rates = [0.025, 0.0025, 0.0001]
modelos = {}
for i in learning_rates:
```

```
    print ("learning rate is: " + str(i))
    modelos[str(i)] = modelo(treino_x, treino_y, teste_x, teste_y, num_iter = 2500, learnir
    print ('\n' + "-------------------------------------------------------" + '\n')

for i in learning_rates:
    plt.plot(np.squeeze(modelos[str(i)]["custos"]), label= str(modelos[str(i)]["learning_ra

plt.ylabel('custo')
plt.xlabel('iterações (por centenas)')

legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()
```

```
learning rate is: 0.025
treino acurácia:100.0 %
teste acurácia: 68.0 %

-------------------------------------------------------

learning rate is: 0.0025
treino acurácia:97.12918660287082 %
teste acurácia: 70.0 %

-------------------------------------------------------

learning rate is: 0.0001
treino acurácia:72.2488038277512 %
teste acurácia: 42.00000000000001 %

-------------------------------------------------------
```



Interpretação:

- Diferentes taxas de aprendizado fornecem custos diferentes e, portanto, resultados de

previsões diferentes.

- Se a taxa de aprendizado for muito alta (0,025), o custo poderá oscilar para cima e para baixo. Pode até divergir (embora, neste exemplo, o uso de 0,025 ainda termine com um bom valor para o custo).
- Um custo menor não significa um modelo melhor. Pode ocorrer o `overfitting`. Isso acontece quando a precisão do treinamento é muito maior que a precisão do teste.

Atividade Complementar 1

Use seu modelo de regressão logistica para o seguinte dataset.

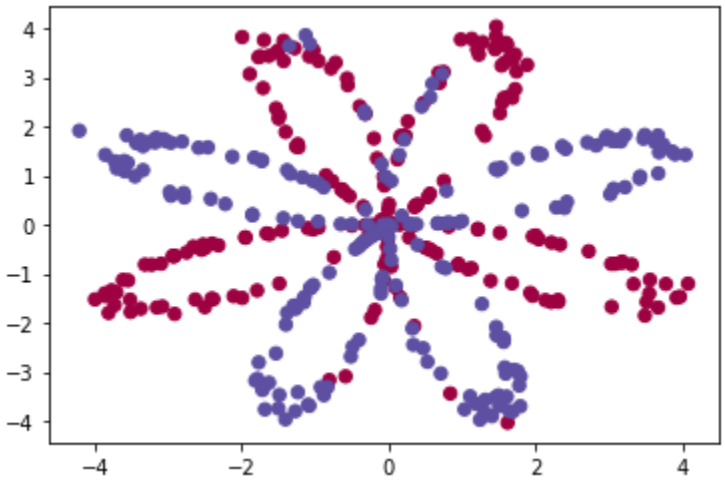
Conjunto de dados de 2 classes com entradas X e rótulos (Y=0, vermelho) e (Y=1, azul). Seu objetivo é verificar se seu classificador se ajusta a esses dados. Em outras palavras, o classificador defina as regiões de vermelha ou azul.

```
def load_planar_dataset():
    np.random.seed(1)
    m = 400 # number of examples
    N = int(m/2) # number of points per class
    D = 2 # dimensionality
    X = np.zeros((m,D)) # data matrix where each row is a single example
    Y = np.zeros((m,1), dtype='uint8') # labels vector (0 for red, 1 for blue)
    a = 4 # maximum ray of the flower

    for j in range(2):
        ix = range(N*j, N*(j+1))
        t = np.linspace(j*3.12,(j+1)*3.12,N) + np.random.randn(N)*0.2 # theta
        r = a*np.sin(4*t) + np.random.randn(N)*0.2 # radius
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        Y[ix] = j

    X = X.T
    Y = Y.T

    return X, Y


X, Y = load_planar_dataset()
Y.shape, X.shape
```

```
((1, 400), (2, 400))
```

```
# Visualize os dados
plt.scatter(X[0,:], X[1,:], c=Y[0,:], s=40, cmap=plt.cm.Spectral);
np.max(X[0,:]), np.max(X[1,:]), np.min(X[0,:]), np.min(X[1,:])
```

```
(4.078165875644617, 4.037643470481511, -4.211898112302497, -4.035124003739587)
```

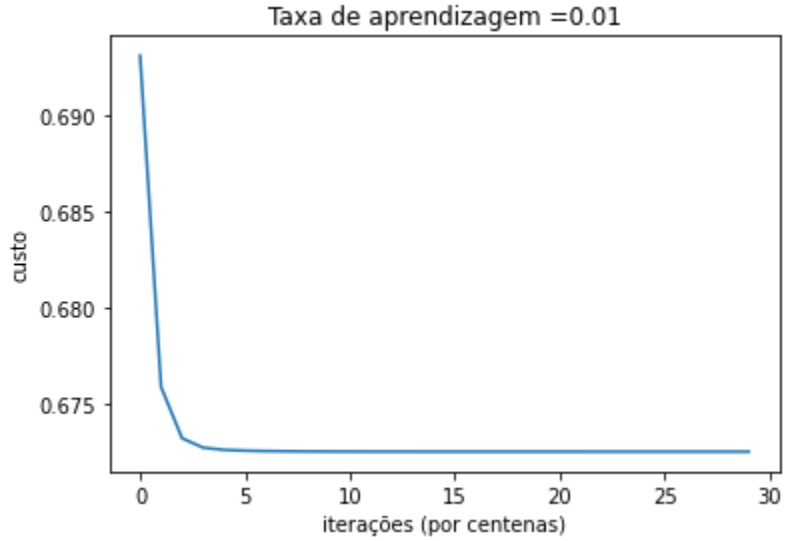


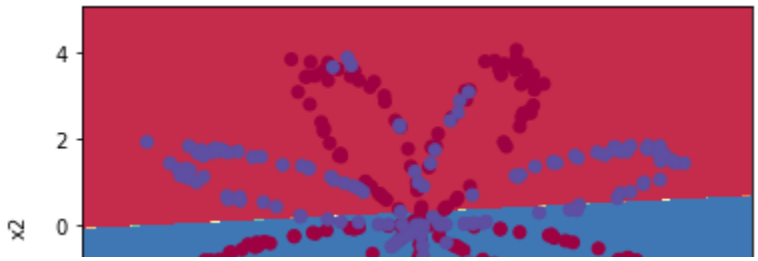
```
# ToDo: plote a fronteira de decisão do seu modelo!
from sklearn.model_selection import train_test_split

# Adaptação baseado no Código: https://baozoulin.gitbook.io/neural-networks-and-deep-learni
def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole grid
    Z = predicao(model["w"], model["b"], np.c_[xx.ravel(), yy.ravel()].T)
    Z = Z.reshape(xx.shape)

    # Plot the contour and training examples
    #plt.contourf(xx, yy, Z,  cmap=plt.cm.gray_r)
    plt.contourf(xx, yy, Z,  cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y[0,:],cmap=plt.cm.Spectral)

# Normaliza entre 0 e 1:
def normalize(X):
   """
     Entrada: X de formato (nº features x nº exemplos)
     Saida: X normalizado
   """
   max_sin = np.max(X[0,:])
   min_sin = np.min(X[0,:])
   X[0,:] = X[0,:]/(max_sin - min_sin)
   max_cos = np.max(X[1,:])
   min_cos = np.min(X[1,:])
   X[1,:] = X[1,:]/(max_cos - min_cos)
```

```
    return X

X_train, X_test, y_train, y_test = train_test_split(X.T, Y[0,:], train_size=0.8, random_sta

X_train = X_train.T
X_test = X_test.T
y_train = y_train[np.newaxis, :]
y_test = y_test[np.newaxis, :]
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)


# Execução do modelo de regressão logística para verificar adequação ao problema (ou inadeq
#d = modelo(X_train_norm, y_train, X_test_norm, y_test, num_iter = 3000, learning_rate = 0.
d = modelo(X_train, y_train, X_test, y_test, num_iter = 3000, learning_rate = 0.01, print_c

# Plot learning curve (with costs)
custos = np.squeeze(d['custos'])
plt.plot(custos)
plt.ylabel('custo')
plt.xlabel('iterações (por centenas)')
plt.title("Taxa de aprendizagem =" + str(d["learning_rate"]))
plt.show()

# Exibindo a fronteira de decisão:
plot_decision_boundary(d, X, Y)
```

```
(2, 320) (2, 80) (1, 320) (1, 80)
Custo após    0 iterações: 0.6931
Custo após  100 iterações: 0.6759
Custo após  200 iterações: 0.6732
Custo após  300 iterações: 0.6727
Custo após  400 iterações: 0.6726
Custo após  500 iterações: 0.6725
Custo após  600 iterações: 0.6725
Custo após  700 iterações: 0.6725
Custo após  800 iterações: 0.6725
Custo após  900 iterações: 0.6725
Custo após 1000 iterações: 0.6725
Custo após 1100 iterações: 0.6725
Custo após 1200 iterações: 0.6725
Custo após 1300 iterações: 0.6725
Custo após 1400 iterações: 0.6725
Custo após 1500 iterações: 0.6725
Custo após 1600 iterações: 0.6725
Custo após 1700 iterações: 0.6725
Custo após 1800 iterações: 0.6725
Custo após 1900 iterações: 0.6725
Custo após 2000 iterações: 0.6725
Custo após 2100 iterações: 0.6725
Custo após 2200 iterações: 0.6725
Custo após 2300 iterações: 0.6725
Custo após 2400 iterações: 0.6725
```

```
Custo após 2400 iterações: 0.6725
Custo após 2500 iterações: 0.6725
Custo após 2600 iterações: 0.6725
Custo após 2700 iterações: 0.6725
Custo após 2800 iterações: 0.6725
Custo após 2900 iterações: 0.6725
treino acurácia:46.875 %
teste acurácia: 51.25 %
```





Atividade Complementar 2

Repita o problema de classificação de gatos usando um objeto da biblioteca scikit-learn que implementa o classificador de vizinhos mais próximos (Nearest Neighbors Classification - KNN). O KNN é um classificador baseado em instâncias e o parâmetro K define o número de vizinhos a se considerar durante a classificação. Ajuste este parâmetro de forma empírica e compare com os resultados obtidos com a regressão logística.

Comparando resultados:

Os resultados para o KNN e a Regressão logística são próximos para o conjunto de testes:

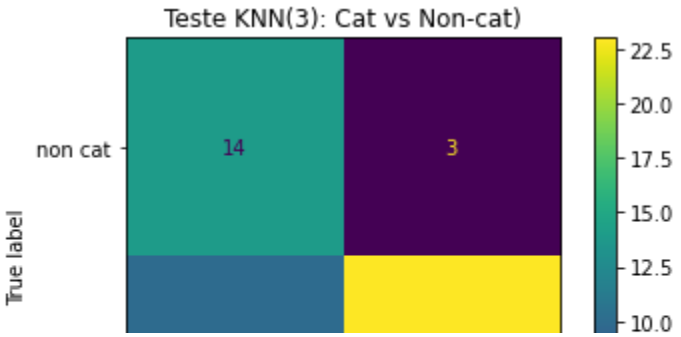
- KNN com 3 vizinhos tem acurácia de 64%, com número maior de falsos negativos. O kNN executado para 1 vizinho teve acurácia superior, de 174%
- Como visto, a regressão logística para treino com 3000 iterações apresentou 68% de acurácia. Uma acurácia de 72% foi atingida após em 22000 iterações com a mesma taxe

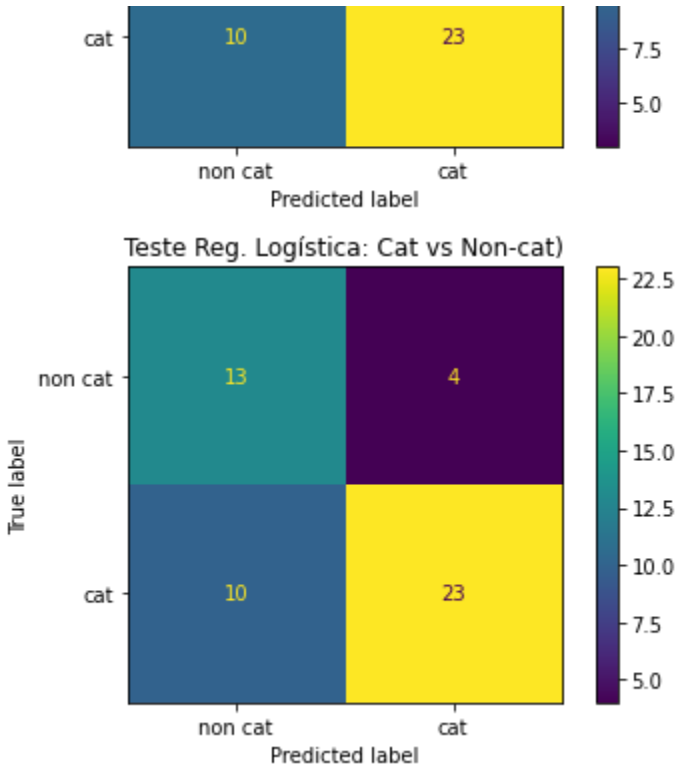
de aprendizagem (0.01).

A vantagem aqui é que o KNN não exige tempo para treinamento, assim como a rede neural aqui implementada.

```
from sklearn.neighbors import KNeighborsClassifier
#ToDo : repita usando-se o KNN do sklearn

nbors = 1
model_knn = KNeighborsClassifier(n_neighbors= nbors)                    ## instancia
# Transpondo treino_x e teste_x
treino_x_knn = treino_x.T
teste_x_knn = teste_x.T
# Extraindo rótulos de treino_y
treino_y_knn = treino_y[0,:]
teste_y_knn = teste_y[0,:]
model_knn.fit(treino_x_knn, treino_y_knn)  ## formato da entrada (n_samples, n_features)
target_prediction = model_knn.predict(teste_x_knn)                    ## teste

# Calculo da acurácia KNN:
print("Acurácia do teste usando KNN (1 vizinho):{} %".format(100 - np.mean(np.abs(target_pr
print("Acurácia do teste usando Regr. Logística: \n- 3.000 iterações e learning rate de 0.(


### Imprimento as matrizes de confusão: Regressão Logística e KNN (1 vizinhos))
cm_knn = confusion_matrix(teste_y_knn, target_prediction, labels=[0, 1])
disp_teste_knn = ConfusionMatrixDisplay(confusion_matrix=cm_knn, display_labels=["non cat",

plt.figure(figsize=(3, 2))
disp_teste_knn.plot()
plt.title("Teste KNN(3): Cat vs Non-cat)")
disp_teste.plot()
plt.title("Teste Reg. Logística: Cat vs Non-cat)")
plt.show()
plt.tight_layout()
```

```
Acurácia do teste usando KNN (1 vizinho):74.0 %
Acurácia do teste usando Regr. Logística para 3.000 iterações e learning rate de 0.01

<Figure size 216x144 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```