Machine Learning: Lista de Exercícios 2

Paulo Orenstein

Monitores: Antônio Catão, Otávio Moreira, Melvin Poveda

Verão, 2025

Exercício 1 (Um pouco de teoria de boosting)**.** Seria possível combinar algoritmos de aprendizado um pouco melhores do que chutes aleatórios (*i.e.*, algoritmos fracos) num algoritmo de performance arbitrariamente boa (*i.e.*, um algoritmo forte)? Essa foi a motivação para o desenvolvimento de boosting; nessa questão, vamos ver como o AdaBoost fornece uma resposta afirmativa no caso de classificação. Para isso, precisamos qualificar o que é um algoritmo fraco. Dada uma amostra de treino $\{(x_i, \tilde{y}_i)\}_{i=1}^n$ e uma distribuição de probabilidade $p = \{p_i\}_{i=1}^n$ sobre as amostras de treino, com $p_i \geq 0$ e $\sum_{i=1}^n p_i = 1$, um algoritmo de aprendizado $F(\cdot, \theta)$ é γ-fraco se para cada p existe $\hat{\theta}$ tal que

$$\sum_{i=1}^{n} p_i \mathbb{I}_{[F(x_i;\hat{\theta}) \neq \tilde{y}_i]} \leq \frac{1}{2} - \gamma.$$

Ou seja, para toda distribuição, existe sempre um parâmetro $\hat{\theta}$ tal que $F(\cdot, \hat{\theta})$ é ao menos um pouco melhor do que um chute aleatório.

Nessa questão, vamos assumir que F é um algoritmo fraco e mostrar que, mesmo assim, a acurácia de treino do AdaBoost decai exponencialmente rápido e quantificar quantas iterações são necessárias para atingir acurácia de treino igual a um.

(a) Seja $\hat{f}_{t-1}(x_i) = \sum_{s=1}^{t-1} \hat{\alpha}_s F(x_i; \hat{\theta}_s)$, com $\hat{\alpha}_s \geq 0$, a função de previsão no período $t-1$, e, no período t, os parâmetros são $(\hat{\alpha}_t, \hat{\theta}_t) = \operatorname{argmin}_{\tilde{\alpha}, \tilde{\theta}} L_t(\tilde{\alpha}, \tilde{\theta})$, onde

$$L_t(\tilde{\alpha}, \tilde{\theta}) = \frac{1}{n} \sum_{i=1}^{n} e^{-\tilde{y}_i(\hat{f}_{t-1}(x_i) + \tilde{\alpha} F(x_i; \tilde{\theta}))} = \sum_{i=1}^{n} w_{it} e^{-\tilde{y}_i \tilde{\alpha} F(x_i; \tilde{\theta})},$$

e $w_{it} = e^{-\tilde{y}_i \hat{f}_{t-1}(x_i)}/n$. Defina os termos

$$W_t^{=} = \sum_{i:\tilde{y}_i = F(x_i;\hat{\theta}_t)} w_{it}, \qquad W_t^{\neq} = \sum_{i:\tilde{y}_i \neq F(x_i;\hat{\theta}_t)} w_{it}.$$

Mostre que, nesse caso, vale que

$$L_t(\hat{\alpha}_t, \hat{\theta}_t) = \min_{\tilde{\alpha}} L_t(\tilde{\alpha}, \hat{\theta}_t) = 2\sqrt{W_t^{=} \cdot W_t^{\neq}}$$
$$L_{t-1}(\hat{\alpha}_{t-1}, \hat{\theta}_{t-1}) = W_t^{=} + W_t^{\neq}.$$

(b) Prove que o fato de $F(\cdot, \theta)$ ser um algoritmo γ-fraco implica que existe um $\hat{\theta}$ tal que,

$$\frac{1}{W_t^{=} + W_t^{\neq}} \sum_{i:\tilde{y}_i \neq F(x_i;\hat{\theta})} w_i \leq \frac{1}{2} - \gamma.$$

(c) Conclua que o item (b) acima junto com a definição de $\hat{\theta}_t$ implica que,

$$\frac{1}{W_t^{=} + W_t^{\neq}} \sum_{i:\tilde{y}_i \neq F(x_i;\hat{\theta}_t)} w_i = \frac{W_t^{\neq}}{W_t^{=} + W_t^{\neq}} \leq \frac{1}{2} - \gamma.$$

(d) Use o item (c) para mostrar que

$$\left(\frac{1+2\gamma}{1-2\gamma}\right) W_t^{\neq} \leq W_t^{=}$$

(e) Usando o fato de que $L_t(\hat{\alpha}_t, \hat{\theta}_t) \leq L_t(\tilde{\alpha}, \hat{\theta}_t)$ para $\tilde{\alpha} = \frac{1}{2}\log\frac{1+2\gamma}{1-2\gamma}$, prove que

$$L_t(\hat{\alpha}_t, \hat{\theta}_t) \leq \sqrt{1-4\gamma^2}\, L_{t-1}(\hat{\alpha}_{t-1}, \hat{\theta}_{t-1}).$$

(f) Supondo que inicializemos $\hat{\alpha}_0 = 0$, de modo que $L_0(\hat{\alpha}_0, \hat{\theta}_0) = 1$, e usando a desigualdade $1 + x \leq e^x$ para $x \in \mathbb{R}$, mostre que

$$L_t(\hat{\alpha}_t, \hat{\theta}_t) \leq e^{-2t\gamma^2},$$

i.e., o erro de treino com a perda exponencial do AdaBoost cai exponencialmente rápido.

(g) Argumente que $L_t(\hat{\alpha}_t, \hat{\theta}_t) < 1/n$ implica em acurácia de treino igual a um, e ache o T^* a partir do qual isso acontece.

(h) Agora vamos verificar esse fenômeno na prática. Considere o seguinte script:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Cria um problema de classificacao artificial
X, y = make_classification(n_samples=2000, n_features=10,
                           n_informative=8, n_redundant=0,
                           random_state=0, shuffle=False)

# Separa dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0)

train_errors = []
test_errors = []

interval = np.arange(1, 250, 10)

for i in tqdm(interval):

    clf = AdaBoostClassifier(n_estimators=i, random_state=0)
    ...  #  Completar
```

```
plt.plot(interval, train_errors, label="train")
plt.plot(interval, test_errors, label="test")

plt.legend()
```

Complete o código acima de forma que você tenha acesso aos valores de erro de treino e teste em cada iteração do algoritmo AdaBoost. Então exiba a curva de treino e teste que você encontrou, e comente sobre o decaimento de cada uma das curvas.

Exercício 2 (MNIST)**.** Vamos comparar métodos de classificação para um problema clássico em machine learning: classificação de imagem de dígitos no dataset MNIST.

Rode o seguinte trecho de código para carregar os dados, os modelos, realizar a divisão dos dados em treino e teste e visualizar algumas amostras contendo os dígitos.

```
from sklearn.datasets import fetch_openml
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import BernoulliNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix
from sklearn import svm
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

def plot_digits(images,
                n_rows = 2,
                n_cols = 5,
                fig_shape = (20,8),
                indexes = [0,1,2,3,4,5,6,7,8,9],
                img_shape = (28,28),
                labels = [0,1,2,3,4,5,6,7,8,9]):

    fig, axs = plt.subplots(n_rows, n_cols, figsize = fig_shape)

    ind = np.array(indexes).reshape(n_rows, n_cols)

    if labels:
        plt_labels = np.array(labels).reshape(n_rows, n_cols)

    for i in range(0,n_rows):
        for j in range(0,n_cols):
            if labels:
                axs[i,j].set_title(plt_labels[i,j], fontsize = 20)

            axs[i,j].imshow((images[ind[i,j]].reshape(img_shape)), cmap = "plasma")
            axs[i,j].axis('off')

mnist = fetch_openml("mnist_784")  # Baixar os dados

X, y = mnist.data.to_numpy(), mnist.target.to_numpy()

X = X/255  # Colocar as features em [0, 1]

# Divisão em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X,
```

```
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)

# Plot dos digítos
plot_digits(X,
            n_rows = 2,
            n_cols = 5,
            indexes = [21, 24, 16, 27, 26, 35, 13, 15, 17, 19])
```

(a) Defina os seguintes modelos:

```
nb = BernoulliNB(force_alpha=True)  # Naive Bayes com features bernoulli
lda = LinearDiscriminantAnalysis()  # LDA
qda = QuadraticDiscriminantAnalysis()  # QDA
lr = LogisticRegression(random_state=42)  # Regressão Logística
knn = KNeighborsClassifier(n_neighbors=6)  # kNN com k = 6
svc = svm.SVC(gamma="scale",class_weight="balanced",C=100)  # SVM
rf = RandomForestClassifier(max_depth=30, random_state=0,
n_estimators = 100)  # Random forest
nn = MLPClassifier(random_state=42, hidden_layer_sizes = [100],
max_iter=300)  # Rede neural
```

Treine cada um destes modelos nos dados de treino e calcule a acurácia no conjunto de treino e de teste. Compare a acurácia e o tempo (em segundos) para treinar e o tempo para gerar previsões obtido nos modelos vistos na primeira parte do curso (NB, LDA, QDA, Regressão Logística e KNN) com os vistos na segunda parte (SVM, random forests e redes neurais). Quais têm melhor acurácia? Quais rodam mais rápido?

(b) Para cada um dos gráficos gerados pelos códigos abaixo, explique o que está sendo mostrado, como isso se relaciona com as previsões do modelo em questão e se é possível interpretar com facilidade o que está acontecendo.

(i)

```
nb_params = {}
for i in range(0,10):
    nb_params[i] = np.exp(nb.feature_log_prob_[i])

plot_digits(nb_params)
```

(ii)

```
lda_params = {}
for i in range(0,10):
    lda_params[i] = lda.means_[i]

plot_digits(lda_params)
```

(iii)

```
log_reg_params = {}
for i in range(0,10):
    log_reg_params[i] = lr.coef_[i]
```

```
plot_digits(log_reg_params)
```

(iv)

```
rf_params = rf.feature_importances_.reshape(28,28)

plt.axis("off")
plt.imshow(rf_params, cmap = "plasma")
```

(v)

```
nn_params_1 = {}
for i in range(0,100):
    nn_params_1[i] = nn.coefs_[0][:,i]

indexes = [i for i in range(0,100)]

plot_digits(nn_params_1,
            n_rows = 10,
            n_cols = 10,
            fig_shape = (100,100),
            indexes = indexes,
            img_shape = (28,28),
            labels = None)
```

e

```
nn_params_2 = {}
for i in range(0,10):
    nn_params_2[i] = nn.coefs_[1][:,i]

plot_digits(nn_params_2,
            n_rows = 2,
            n_cols = 5,
            fig_shape = (20,8),
            img_shape = (10,10))
```

(c) Gere a matriz de confusão para as previsões do Naive Bayes no conjunto de teste. Veja para cada classe qual é o erro mais comum e dê uma possível explicação.

Exercício 3 (Atenção, por favor)**.** Uma das peças fundamentais que compõem transformers é a camada de atenção, em que a rede aprende como relacionar unidades de texto, ou tokens. Apesar de atenção ser uma ideia recente, usar *keys* e *queries* para aprender a relação entre objetos não é algo novo. Considere o seguinte trecho de código, que lê um dataset de notícias.

Sugestão: Para essa questão, é útil usar o Google Colab, para ter acesso a uma GPU. Clique em "Ambiente de Execução"(ou "Runtime"), "Alterar o tipo de ambiente de execução" ("Change runtime type"), e, finalmente, "T4 GPU".

```
from sklearn.datasets import fetch_20newsgroups

categories = [
    "alt.atheism",
    "rec.autos",
    "comp.graphics",
    "sci.space",
]

data_train = fetch_20newsgroups(
    subset="train",
    categories=categories,
    shuffle=True,
    random_state=42,
    remove=(),
)
data_test = fetch_20newsgroups(
    subset="test",
    categories=categories,
    shuffle=True,
    random_state=42,
    remove=(),
)
```

No dataset, cada notícia tem um texto (x) e um assunto (y) associados. Imagine que tenhamos um exemplo de notícia e queiramos encontrar outras notícias similares olhando apenas para o seu texto x. Para isso, vamos primeiro transformar o texto num vetor. Seja $V = \{w_1, ..., w_N\}$ o vocabulário com todas as palavras w_i que aparecem em alguma notícia, com $|V| = D$. Um exemplo de vetorização $v \in \mathbb{R}^D$ de um texto é tomar $v = (v_i)_{i=1}^D$ tal que v_i é o número de vezes em que a palavra w_i aparece nesse texto.

Uma vetorização um pouco mais sofisticada é conhecida como tf-idf, e pondera o peso da contagem associada a uma palavra w_i proporcionalmente ao inverso da frequência dessa palavra no acervo de todas as notícias, com o objetivo de dar menos peso a palavras comuns mas com pouco valor semântico (como "a", "e", "ou", etc.) e mais peso a palavras relativamente raras mas informativas (como "inflação").

Vamos usar a vetorização tf-idf para calcular similaridade par a par para 16 notícias. Rode o seguinte trecho de código:

```
import torch
from sklearn.feature_extraction.text import TfidfVectorizer
!pip install bertviz
from bertviz import head_view, model_view

# order of labels in `target_names` can be different from `categories`
```

```
target_names = data_train.target_names

# split target in a training set and a test set
y_train, y_test = data_train.target, data_test.target

# Extracting features from the training data using a sparse vectorizer
vectorizer = TfidfVectorizer(
    sublinear_tf=True, max_df=0.5, min_df=5, stop_words="english"
)
X_train = vectorizer.fit_transform(data_train.data)

# Extracting features from the test data using the same vectorizer
X_test = vectorizer.transform(data_test.data)

feature_names = vectorizer.get_feature_names_out()

seq = X_test[:32,:]
b = seq.toarray()
att = b@b.T
att = att.reshape(1,1,32,32)
att = torch.tensor(att)
counts = [0,0,0,0]
tokens = []
for i in y_test[:32]:
  token = target_names[i] + ' ' + str(counts[i])
  tokens.append(token)
  counts[i]+=1
head_view((att,), tokens)
```

(a) Explique o que essa visualização representa e interprete os resultados obtidos.

(b) Vamos tornar concreto o motivo de usar produto interno para quantificar similaridade.

(b-i) Sejam $u, v \in \mathbb{R}^d$, P_u a projeção ortogonal sobre o espaço gerado por u. Mostre que

$$P_u(v) = \frac{\langle u, v \rangle}{\|u\|^2} u$$

(b-ii) Considere a definição de cosseno entre dois vetores

$$\cos(u, v) := \frac{\langle u, v \rangle}{\|u\| \, \|v\|}.$$

Use a fórmula do item (b-i) para relacionar essa definição de cosseno com o cosseno de um ângulo (construa um triângulo retângulo com v e $P_u(v)$).

(b-iii) Mostre que, se $u_1, v_1, u_2, v_2 \in \mathbb{R}^d$ com $\|u_1\| = \|v_1\| = \|u_2\| = \|v_2\| = 1$, então

$$\cos(u_1, v_1) \geq \cos(u_2, v_2) \iff \|u_1 - v_1\| \leq \|u_2 - v_2\| .$$

Argumente que $\cos$ pode ser usado como uma medida de similaridade quando os vetores em consideração têm norma 1.

(b-iv) Em vista do item anterior, aponte diferenças entre a noção de similaridade capturada por $\|u - v\|$ e $\cos(u, v)$. Em que casos seria mais vantajoso usar cada uma dessas noções de similaridade?

Agora imagine que queiramos usar essa vetorização para aprender a similaridade entre notícias, em que notícias de mesmo assunto (y) têm similaridade 1 e notícias de assuntos diferentes têm similaridade 0. Vamos treinar uma rede neural que aprenda a decidir a similaridade par a par em sequências de 16 notícias usando apenas a vetorização das palavras de cada uma. Primeiro, precisamos converter as nossas matrizes para tensores do Torch (e, se disponível, enviá-las para a GPU com `.cuda()`):

```
X_train = torch.tensor(X_train.toarray(), dtype=torch.float32).cuda()
X_train = (X_train-X_train.mean())/X_train.std()
y_train = torch.tensor(y_train, dtype=torch.float32).cuda()
X_test = torch.tensor(X_test.toarray(), dtype=torch.float32).cuda()
X_test = (X_test-X_test.mean())/X_test.std()
y_test = torch.tensor(y_test, dtype=torch.float32).cuda()
```

Agora podemos começar a treinar nossa rede! Execute o seguinte trecho.

```
torch.manual_seed(0)

n_epochs = 5
lr = 1e-5
sequence_length = 32

def init_scale(fan_in):
    return (2/fan_in)**.5

D = X_train.shape[1]

A_K = torch.randn((D,D)).cuda()#*init_scale(D)
A_Q = torch.randn((D,D)).cuda()#*init_scale(D)

A_K.requires_grad = True
A_Q.requires_grad = True

optimizer = torch.optim.Adam([A_K, A_Q], lr = lr)
softmax = torch.nn.Softmax(dim=0)
cos = torch.nn.CosineSimilarity(dim=0)
losses = []
for epoch in range(n_epochs):
  permutation = torch.randperm(X_train.shape[0])
  for i in range(0,X_train.shape[0], sequence_length):
    optimizer.zero_grad()
    indices = permutation[i:i+sequence_length]
    if len(indices)!=sequence_length:
      continue
    seq_X, seq_Y = X_train[indices], y_train[indices]

    K = A_K@(seq_X.T)
    Q = A_Q@(seq_X.T)
```

```
att = softmax((K.T@Q)/(D**(1/2)))
truth = ((seq_Y==seq_Y.reshape(-1,1))==0).type(torch.float32)
loss = -cos(att,truth).mean()
loss.backward()
optimizer.step()
losses.append(loss.detach().cpu())
```

(c) Note que a rede não está aprendendo. Explique qual é o problema e exponha evidências de que ele realmente está acontecendo.

(d) Descomente a parte do código comentada acima. Explique por que essa modificação resolve o problema.

(e) Outro detalhe da implementação que ajuda a evitar que incorramos no mesmo problema é a divisão por $\sqrt{D}$ antes de tomar o Softmax. Vamos justificar essa mudança de escala.

(e-i) A camada de auto-atenção (*self-attention*) pode ser representada matematicamente como

$$\text{Sa}[X] = V[X] \cdot \text{Softmax}\left[\frac{K[X]^T Q[X]}{\sqrt{D}}\right],$$

em que $V[X], K[X], Q[X] \in \mathbb{R}^{D \times n}$ são as matriz de *values*, *keys* e *queries* associados a X em que n é o tamanho da sequência (no nosso caso $n = 32$).
Mostre que, se $K[X]$ e $Q[X]$ têm entradas gaussianas com média zero e variância σ^2, todas independentes entre si, cada entrada de $K[X]^T Q[X]$ tem variância $\sigma^4 D$. Use esse fato para argumentar a favor da divisão por $\sqrt{D}$.

(e-ii) É necessário assumir que as entradas são gaussianas para chegar a essa conclusão? Caso contrário, é necessário que sejam identicamente distribuídas?

(f) Usando o comando `head_view`, visualize a similaridade par a par que a rede associa a sequência de notícias correspondente a `X_test[:32]`, `y_test[:32]`. Compare com a visualização realizada no item (a) e interprete o resultado.

(g) Observe que nessa implementação não foi empregado o uso da matriz $V[X]$ e tampouco foi usada a ideia de multi-head attention, muito comum em Transformers. Aponte um exemplo de problema em que seria útil implementar esses detalhes.

Caso tenha interesse em visualizar o mecanismo de atenção funcionando em um LLM de verdade, sugerimos acessar o notebook do Google Colab disponibilizado em github.com/jessevig/bertviz.

Exercício 4 (Compressão com k)**.** Em uma imagem colorida com P pixeis, cada pixel pode ser representado por um vetor em três dimensões $p_i = (r_i, g_i, b_i) \in (0,1)^3$, onde r_i, g_i, b_i representam a intensidade da cor vermelha, verde e azul, respectivamente.

(a) Faça o download da imagem `araras.png` e considere o seguinte código:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from sklearn.cluster import KMeans
from tqdm import tqdm
import os

def function_1(img, parameter):
    temp = np.reshape(img, (img.shape[0]*img.shape[1],3))
    model = KMeans(n_clusters=parameter, random_state=0, n_init=1).fit(temp)

    output = model.cluster_centers_[model.labels_]
    output = np.reshape(output,(img.shape[0],img.shape[1],3))
    return output

imagem = mpimg.imread('araras.png')
output = function_1(imagem, 3)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15,15))

ax1.imshow(imagem)
ax1.set_title("Figura 1")

ax2.imshow(output)
ax2.set_title(f"Figura 2")

ax3.imshow(np.abs(imagem - output))
ax3.set_title("Figura 3")

plt.show()
```

Explique o que as Figuras 1, 2 e 3 representam no código acima e o impacto da variável `parameter` na Figura 2. Exiba o output do código acima para `parameter=3, 10, 30`.

(b) Em geral, um arquivo `.png` usa valores $j/255$, com $j = 0, 1, \ldots, 255$ para a intensidade de cor (vermelho, verde ou azul) em cada pixel. Ou seja, cada intensidade é representada com no máximo $\log_2(256) = 8$ bits por pixel e, portanto, cada pixel pode ser representado por até $3 \text{ cores} \times 8 \text{ bits} = 24$ bits. Por outro lado, a `function_1` retorna imagens comprimidas, isto é, em que o número de bits por pixel é potencialmente muito menor. Determine qual é o número de bits por pixel necessário em função de `parameter`.

(c) O código a seguir roda a função `function_1` em um conjunto pré-definido de valores `parameter` e salva em uma lista a sua taxa de compressão, isto é, o tamanho em bits de cada output salvo dividido pelo tamanho em bits da imagem original.

```
parameter_interval = np.array(range(2, 50, 5))

for i in parameter_interval:
    image = mpimg.imread('araras.png')
    tmp = function_1(image, i)
    plt.imsave(f"imagem_{i}.png", tmp)

sizes = []
original_size = os.path.getsize('araras.png')

for i in parameter_interval:
    file_size = os.path.getsize(f'imagem_{i}.png') / original_size
    sizes.append(file_size)
```

Faça um gráfico onde o eixo x é dado por `parameter_interval` e o eixo y é dado pela taxa de compressão da imagem. Desenhe duas linhas, uma representando a taxa de compressão verificada na lista `sizes` e a outra representando a taxa de compressão teórica encontrada no item anterior. Comente o resultado.

(d) Considere o código abaixo:

```
def function_2(img, parameter):
    temp = np.reshape(img, (img.shape[0]*img.shape[1],3))
    model = KMeans(n_clusters=parameter, random_state=0, n_init=1).fit(temp)

    return model.inertia_

within_cluster_variation = []

for i in parameter_interval:
    image = mpimg.imread('araras.png')
    within_cluster_variation.append(function_2(image, i))
```

O valor `model.inertia_` representa a soma dos quadrados das distâncias dos pontos até o centro do cluster em que se encontram. Faça um gráfico desenhando para cada valor em `parameter_interval` seu respectivo valor na lista `within_cluster_variation`. Intuitivamente, como você usaria esse gráfico para escolher uma quantidade de clusters k apropriada para o problema? Justifique sua resposta e use o código abaixo para apresentar o resultado da imagem utilizando o valor de k que você encontrou.

```
k = ...  # Preencher

output = function_1(imagem, k)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15,15))

ax1.imshow(imagem)
ax1.set_title("Figura 1")

ax2.imshow(output)
```

```
ax2.set_title(f"Figura 2")

ax3.imshow(np.abs(imagem - output))
ax3.set_title("Figura 3")

plt.show()
```

(e) A técnica apresentada nos itens anteriores também pode ser usada para criação de um algoritmo de detecção de bordas. Para isso, podemos pintar de preto os pontos da imagem em que um de seus vizinhos (acima, abaixo, à esquerda e à direita) pertence a um cluster diferente do seu. Lembrando que a cor preta é indicada pelo vetor $(0, 0, 0)$, preencha o código a seguir para obter o efeito desejado. Ao final, exiba a imagem encontrada.

```
def function_3(img):
    res = img.copy()

    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            up = (img[i,j] != img[i+1,j]).any()
            down = ...  # Preencher
            left = ...  # Preencher
            right = ...  # Preencher

            if up or down or left or right:
                res[i,j] = np.array([0,0,0])
    return res

imagem = mpimg.imread('araras.png')
output = function_1(imagem, 4)
edge = function_3(output)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(50,50))

ax1.imshow(imagem)
ax1.set_title("Figura 1")

ax2.imshow(output)
ax2.set_title(f"Figura 2")

ax3.imshow(edge)
ax3.set_title("Figura 3")

plt.show()
```

(f) Agora, suponha que além do vetor com as intensidades de vermelho, verde e azul, adicionemos também a coordenada x e y de cada pixel, com um certo peso w. Isso é implementado na função `function_4` no código abaixo, que imprime o resultado para diferentes valores de w.

```
def function_4(img, parameter, weight):
    tmp = img.copy()
```

```
    tmp = np.dstack((imagem,  np.zeros((imagem.shape[0], imagem.shape[1]))))
    tmp = np.dstack((tmp,  np.zeros((imagem.shape[0], imagem.shape[1]))))

    for i in range(tmp.shape[0]):
        for j in range(tmp.shape[1]):
            tmp[i,j, 3] = weight*i
            tmp[i,j, 4] = weight*j

    temp = np.reshape(tmp, (img.shape[0]*img.shape[1], 5))

    model = KMeans(n_clusters=parameter, random_state=0, n_init="auto").fit(temp)

    output = model.cluster_centers_[model.labels_]
    output = np.reshape(output,(img.shape[0],img.shape[1],5))

    return output[:,:,:3]

imagem = mpimg.imread('araras.png')
k = 100
weights = [0, 0.001, 0.005, 0.0075, 0.01, 0.02, 0.05, 0.1, 1]

fig, axes = plt.subplots(nrows = 3, ncols = 3, figsize=(25,25))

for i, axis in tqdm(enumerate(axes.flatten()), total=len(weights)):
    axis.imshow(function_3(function_4(imagem, k, weight=weights[i])))
    axis.set_title(f"Weight: {weights[i]}")

plt.show()
```

A partir das figuras resultantes e do seu conhecimento do algoritmo de k-means, explique o efeito da inclusão das coordenadas x e y de cada pixel no vetor com as intensidades das cores. Discuta o impacto do peso w e, em particular, o formato regular dos contornos na figura quando w é grande.