INTELIGÊNCIA ARTIFICIAL

Responsável: Prof. Rodrigo Carvalho Souza Costa

Resumo

Inteligência Artificial (IA) é um ramo da ciência da computação é capaz de simular a capacidade humana de reconhecer um problema, ou uma tarefa a ser realizada, analisar dados e tomar decisões através de algoritmos definidos por especialistas.

A IA tem ganhado cada vez mais importância nos últimos anos devido ao crescente volume de dados disponíveis e à necessidade de automatizar tarefas repetitivas em diferentes áreas.

Este trabalho aborda uma introdução à Inteligência Artificial e ao Reconhecimento de Padrões. Em seguida, ele se aprofunda em Classificadores, abordando tanto Classificadores Supervisionados quanto não Supervisionados. Em seguida, são discutidas Redes Neurais e Deep Learning, incluindo uma visão geral das bibliotecas mais populares, como TensorFlow, Keras e Pytorch.

Ao longo do trabalho, são apresentados exemplos práticos e aplicações de cada um dos tópicos discutidos. O objetivo é fornecer uma visão geral desses tópicos e uma compreensão básica de como eles funcionam, permitindo ao leitor ter uma base sólida para explorar ainda mais esses conceitos.

Palavras chave: Inteligência Artificial, Reconhecimento de Padrões, Classificadores, Redes Neurais, Deep Learning.

Sumário

UNIDADE 1: INTRODUÇÃO À INTELIGÊNCIA ARTIFICIAL

Inteligência Artificial (IA) é um ramo da ciência da computação é capaz de simular a capacidade humana de reconhecer um problema, ou uma tarefa a ser realizada, analisar dados e tomar decisões através de algoritmos definidos por especialistas [1].

A IA tem ganhado cada vez mais importância nos últimos anos devido ao crescente volume de dados disponíveis e à necessidade de automatizar tarefas repetitivas em diferentes áreas. De acordo com um estudo de Haddadi et al., a IA tem o potencial de ajudar em áreas como medicina, finanças, transporte e segurança, entre outras[1] [2].

1.1 ÁREAS E APLICAÇÕES DE IA

Existem várias subáreas dentro da IA, incluindo aprendizado de máquina, visão computacional, processamento de linguagem natural e robótica. O aprendizado de máquina, por exemplo, é uma subárea da IA que se concentra em desenvolver algoritmos capazes de aprender a partir de dados e melhorar seu desempenho com o tempo. Segundo um estudo de Goodfellow et al., o aprendizado de máquina tem sido amplamente aplicado em áreas como reconhecimento de fala, processamento de imagens e recomendação de conteúdo [2].

Outra subárea da IA é a visão computacional, que se concentra em desenvolver algoritmos capazes de interpretar e analisar imagens e vídeos. A visão computacional tem várias aplicações, incluindo detecção de objetos em imagens, reconhecimento facial e condução autônoma. De acordo com um estudo de Sermanet et al., a visão computacional tem o potencial de revolucionar a indústria automotiva, permitindo a criação de veículos autônomos mais seguros e eficientes [3].

O processamento de linguagem natural é outra subárea da IA que se concentra em desenvolver algoritmos capazes de interpretar e gerar linguagem natural. Essa subárea tem várias aplicações, incluindo assistentes virtuais, tradução automática e análise de sentimento. De acordo com um estudo de Jurafsky e Martin, o processamento de linguagem natural tem sido amplamente utilizado em serviços de atendimento ao cliente, permitindo a automatização de tarefas como triagem de chamadas e resolução de problemas [4].

Por fim, a robótica é outra subárea da IA que se concentra em desenvolver sistemas robóticos capazes de interagir com o ambiente e realizar tarefas complexas. A robótica tem várias aplicações, incluindo manufatura, saúde e serviços. De acordo com um estudo de Khatib, a robótica tem o potencial de transformar a indústria manufatureira, permitindo a criação de sistemas de produção mais flexíveis e eficientes [5].

1.2 TIPOSDE INTELIGENCIA ARTIFICIAL

A Inteligência Artificial (IA) é dividida em dois tipos principais: IA fraca (Weak AI) e IA forte (Strong AI). A IA fraca é aquela que é projetada para realizar tarefas específicas e limitadas,

sendo capaz de imitar a inteligência humana em tarefas específicas, mas sem ter a capacidade de pensar ou raciocinar como um ser humano. De acordo com a definição de Russell e Norvig, a IA fraca é uma "inteligência simulada em uma máquina projetada para realizar uma tarefa específica" [1].

Já a IA forte é aquela que busca criar sistemas inteligentes que possam pensar e raciocinar como seres humanos, ou até mesmo superá-los em inteligência. Essa forma de IA é baseada na ideia de que a inteligência é uma propriedade geral da matéria organizada, e não apenas uma característica específica dos seres humanos. Segundo Russell e Norvig, a IA forte é uma "inteligência simulada em uma máquina capaz de realizar qualquer tarefa intelectual que um ser humano pode realizar" [1].

A IA fraca é amplamente utilizada em diferentes áreas, como finanças, medicina e manufatura. Um exemplo de IA fraca é o sistema de recomendação de filmes da Netflix, que é projetado para recomendar filmes com base no histórico de visualizações do usuário.

Já a IA forte, apesar de ainda ser objeto de estudo e pesquisa, tem o potencial de revolucionar a forma como interagimos com a tecnologia e de transformar completamente diversos setores da sociedade.

Portanto, a IA pode ser dividida em dois tipos principais, sendo eles a IA fraca e a IA forte. A IA fraca é projetada para realizar tarefas específicas e limitadas, enquanto a IA forte busca criar sistemas inteligentes que possam pensar e raciocinar como seres humanos. Ambas as formas de IA possuem aplicações em diferentes áreas e têm o potencial de transformar a forma como vivemos e trabalhamos.

UNIDADE 2: INTRODUÇÃO AO RECONHECIMENTO DE PADRÕES

O Reconhecimento de Padrões trata da classificação e da descrição de uma estrutura de dados através de um conjunto de propriedades ou características. Esta área de pesquisa tem despertado nas últimas décadas um grande interesse na comunidade científica. Com o advento dos computadores, este assunto tem sido um dos mais desafiadores na tentativa de atribuir habilidades humanas aos sistemas artificiais [7].

Um padrão é uma descrição de um objeto que pode ser classificado como: concretos espacial: caracteres, imagens; temporal: formas de onda, séries etc) e abstrato: raciocínio, soluções a problemas etc [7].

O Reconhecimento de Padrões através de máquinas envolve técnicas para a atribuição dos padrões a suas respectivas classes, de forma automática ou com a menor intervenção humana possível [7].

O reconhecimento de padrões é composto pelas seguintes tarefas [6]:

- **extração de características:** apresenta os dados de entrada em termos de medidas ou informações que possam ser utilizados facilmente na etapa de classificação;
- **classificação de padrões**: os padrões são agrupados em função das características comuns entre eles.

Neste capítulo são descritos os fundamentos do reconhecimento de padrões utilizados nesta apostila.

2.1 VETOR E ESPAÇO DE CARACTERÍSTICAS

Através da extração de características o espaço de dados é transformado em um espaço de características que possui a mesma dimensão do espaço de dados original, porém, representado por um número reduzido de características efetiva.

Cada padrão $\boldsymbol{P_x}$ pode ser representado através de um vetor composto por um conjunto de $\boldsymbol{n}$ características $\boldsymbol{x_i}$:

$$P_x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Uma forma de expressar este vetor é através da notação $P_x = [x_1, x_n, \dots x_n]^T$, em que $\boldsymbol{T}$ indica a operação de transposição de matrizes.

2.2 INTRODUÇÃO À CLASSIFICAÇÃO DE PADRÕES

A classificação pode ser definida como um problema relacionado com a determinação de uma fronteira de decisão que consegue distinguir diferentes padrões em classes dentro de um espaço de características n-dimensional [7]. A classificação, então, pode ser realizada, e pode ser entendida, de maneira geral, pela partição do espaço de atributos em um número finito de regiões de tal forma que objetos de uma mesma classe recaiam, pelo menos em tese, sempre dentro de uma mesma região.

Os problemas de classificação podem ser subdivididos em três tipos baseados em [7]:

- **critério imposto**: os critérios são impostos de acordo com cada problema prático. Por exemplo, alguém querendo classificar galinhas maduras como todas aquelas cujo peso exceder um limiar específico. Como os critérios são claramente atribuídos pela saída, tudo que resta a ser feito é implementar um meio eficaz para medir as características, por consequência, esta é a situação mais simples;
- **através de exemplos (ou classificação supervisionada)**: um ou mais padrões de exemplos, conhecidos como conjunto de treinamento, de classes de objetos conhecidos a priori são fornecidos como referência para classificação de padrões desconhecidos. Esse tipo de classificação é dividido em dois estágios:
    - (i) aprendizado, correspondendo à etapa em que os parâmetros dos classificadores e os critérios de seleção são formulados a partir dos protótipos e
    - (ii) reconhecimento, onde o sistema treinado é usado para classificar novos padrões desconhecidos;
- **critério aberto (ou classificação não-supervisionada)**: não há necessidade de um conhecimento prévio das amostras para identificação de um padrão, geralmente utilizam-se técnicas de agregação de dados. Essas técnicas baseiam-se frequentemente na minimização de um critério derivado de uma medida de similaridade entre as amostras .

Apesar da intensa pesquisa na área de Reconhecimento de Padrões nas últimas décadas, não há nenhuma solução definitiva e geral para selecionar as características ótimas e obter um algoritmo otimizado para classificação [8].

Muitos classificadores assumem que os dados podem ser modelados por distribuição estatística. Os classificadores paramétricos estimam os parâmetros das distribuições envolvidas, enquanto que os classificadores não paramétricos realizam a classificação sem modelar os dados [7].

2.3 REVISÃO DE PYTHON

Neste curso pressupõe-se que os alunos tenham uma base de programação. Desta forma, espera-se que os alunos tenham o conhecimento básico sobre os comandos do python mostrados na Tabela 1.

A Python é uma linguagens de programação de extrema popularidade, sendo uma mais utilizadas em todo o mundo. O Python possui uma grande comunidade de desenvolvedores e usuários que continuamente criam bibliotecas e funcionalidades para a linguagem.

Tabela 1 – principais comandos em python.

Comando	Descrição e modo de uso
`input()`	utilizado para receber dados de entrada do usuário. Por exemplo, para receber um valor inteiro do usuário e armazená-lo em uma variável "idade", `idade = input("Digite sua idade: ")`
`print()`	utilizado para imprimir informações na tela do usuário. Por exemplo, para imprimir a mensagem "Olá, mundo!" na tela, utilizamos o seguinte código: `print ("Olá, mundo!")`
`variáveis=`	tilizadas para armazenar valores. Para criar uma variável, utilizamos o sinal de igual "=" para atribuir um valor a ela. Por exemplo, para criar uma variável "nome" e atribuir o valor "João" a ela, utilizamos o seguinte código: `nome = "Iracema"`

`+ - * / **`	são operadores operações matemáticas básicos soma (+), subtração (-), multiplicação (*), divisão (/) e potência (**). Exemplo:
	`a = 5`
	`b = 3`
	`resultado = a + b`
	`print(resultado)`

`if (cond):`
 	
`else:`
	utilizado para executar um bloco de código apenas se uma determinada condição for verdadeira. Por exemplo, para verificar se um número é positivo e imprimir uma mensagem na tela, utilizamos o seguinte código:
	
	`numero = 5`
	`if numero > 0:`
	`  print("O número é positivo")`
	`else`
	`  print("O número é negativo")`

`["a", "b", "c"]`	usamos colchetes [ ] e separamos os elementos com vírgulas para criar listas. A lista é um tipo de objeto que pode conter um conjunto ordenado de elementos. Os elementos em uma lista podem ser de diferentes tipos, incluindo números, strings, booleanos, outras listas e objetos complexos. As listas são muito úteis para armazenar um conjunto de valores que precisam ser processados em conjunto. Por exemplo, podemos criar uma lista de frutas e armazená-la em uma variável chamada frutas utilizando o seguinte código: `frutas = ["maçã", "banana", "laranja"]`

`for seq:`	usado para iterar sobre uma sequência (como uma lista, tupla, dicionário, etc.). Por exemplo:
	`for fruta in frutas:`
	`  print(fruta)`

`def fun(a,b):`	usado para definir funções em Python. As funções são blocos de código que podem ser chamados várias vezes. Por exemplo:
	`def soma(a, b):`
	`  return a + b`

Assim, a cada dia que passa surgem novas bibliotecas e ferramentas disponíveis para Python, tornando-a uma linguagem poderosa e versátil para muitos fins, incluindo inteligência artificial, ciência de dados, desenvolvimento web e muito mais. Antes de importar uma biblioteca, deve-se instalá-la através do comando pip.

O comando pip é um gerenciador de pacotes para o Python que permite instalar, atualizar e desinstalar bibliotecas e módulos de terceiros. Essas bibliotecas e módulos podem ser encontrados no repositório oficial do Python Package Index (PyPI). Uma forma de instalar o pip no sistema operacional é realizar o seguinte comando:

```
python -m easy_install python
```

Uma vez instalado, você pode usar o comando pip para instalar bibliotecas e módulos de terceiros para o Python. Para importar uma biblioteca python, o comando **`import`** é usado em Python para importar módulos, que são arquivos Python que contêm definições e declarações de funções, variáveis e classes.

As bibliotecas ou pacotes em Python, muitas vezes contêm vários módulos que fornecem funções e ferramentas especializadas para resolver tarefas específicas. Ao importar esses pacotes e módulos em nossos scripts Python, podemos acessar suas funções e recursos para tornar nosso trabalho mais eficiente e produtivo.

Na tabela 2 é mostrada algumas das bibliotecas mais comuns usadas em inteligência artificial com Python:

Tabela 2 – exemplos de bibliotecas importantes para a área de inteligência artificial.

Biblioteca	Área de utilização	
NumPy	computação científica	fornece suporte para matrizes e operações matemáticas de alto nível. É frequentemente usado para manipulação e processamento de dados numéricos, e é uma das bibliotecas mais populares para aprendizado de máquina.
Pandas	análise e manipulação de dados	comumente usado para trabalhar com dados estruturados, como tabelas e planilhas.
Matplotlib	criação de gráficos e visualizações	útil para criar visualizações de dados para explorar e entender melhor os padrões e relacionamentos nos dados.
Scikit-learn	aprendizado de máquina em Python	fornece uma variedade de algoritmos de aprendizado de máquina para tarefas de classificação, regressão e agrupamento, além de ferramentas para avaliação de modelos e pré-processamento de dados.
TensorFlow	aprendizado de máquina de código aberto da Google	usado principalmente para criação e treinamento de modelos de aprendizado profundo, incluindo redes neurais profundas.
Keras	aprendizado profundo de alto nível	pode ser executado em cima do TensorFlow é comumente usada para criar modelos de aprendizado profundo com poucas linhas de código.
PyTorch	aprendizado profundo de código aberto	É usado principalmente para criação e treinamento de modelos de aprendizado profundo, incluindo redes neurais profundas.
NLTK	processamento de linguagem natural (NLP)	é usado para tarefas como tokenização de texto, análise sintática e semântica, extração de entidades e geração de texto.

Ao importar essas bibliotecas em nossos scripts Python, podemos usar as funções e recursos que elas fornecem para construir modelos de aprendizado de máquina e processar dados de maneira eficiente e eficaz.

O comando import é utilizado em Python para incluir módulos (arquivos com código Python) em um programa. Isso permite que o programa tenha acesso a todas as funções e variáveis definidas no módulo.

Para importar um módulo, basta usar o comando **`import`** seguido do nome do módulo. Por exemplo, para importar o módulo **`math`** (que contém funções matemáticas), podemos usar o seguinte comando:

```
import math
```

Uma vez que o módulo é importado, podemos utilizar qualquer função ou variável definida nele através da sintaxe nome_do_modulo.nome_da_funcao. Por exemplo, para calcular a raiz quadrada de um número, podemos usar a função sqrt() definida no módulo math, da seguinte forma:

```
import math

raiz_quadrada = math.sqrt(16)
print(raiz_quadrada) # Imprime 4.0
```

Podemos também importar funções específicas de um módulo, em vez de importar o módulo inteiro. Para fazer isso, usamos a sintaxe from nome_do_modulo import nome_da_funcao. Por exemplo, para importar apenas a função sqrt() do módulo math, podemos usar o seguinte comando:

```
from math import sqrt

raiz_quadrada = sqrt(16)
print(raiz_quadrada) # Imprime 4.0
```

Isso tem a vantagem de evitar a necessidade de digitar o nome do módulo ao usar a função, o que pode tornar o código mais conciso e legível. No entanto, se quisermos usar várias funções de um mesmo módulo, é geralmente mais conveniente importar o módulo inteiro.

2.4 REVISÃO DE PYTHON PYTHON PARA MANIPULAÇÃO DE DADOS

Para trabalharmos com a manipulação de dados neste curso, primeiro, importamos as bibliotecas necessárias: numpy, pandas e matplotlib.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Em seguida, lemos o arquivo caracteristicas.txt[1] com o Pandas e o armazenamos na variável dados.

```
dados = pd.read_csv('caracteristicas.txt', header=None)
```

O parâmetro header=None indica que o arquivo não tem cabeçalho, e os nomes das colunas são especificados em seguida com o comando dados.columns = ['X', 'Y', 'Classe'].

```
dados.columns = ['X', 'Y', 'Classe']
```

Em seguida, criamos um array para cada classe, utilizando a função dados[dados['Classe'] == i] para selecionar as linhas do dos dados que correspondem a cada classe.

```
# Criando um array para cada classe
classes = []
for i in range(1, 4):
    classes.append(dados[dados['Classe'] == i])
```

Finalmente, plotamos as classes com o scatter do Matplotlib, utilizando um marcador diferente para cada classe (definidos na lista marcadores).

```
marcadores = ['*', '-', '+']
for i, classe in enumerate(classes):
    plt.scatter(classe['X'], classe['Y'], marker=marcadores[i])
```

[1] O arquivo caracteristicas.txt é um arquivo de texto simples que contém uma tabela de dados. Cada linha representa uma observação ou exemplo, com três valores separados por vírgula: o primeiro valor é a característica X, o segundo valor é a característica Y e o terceiro valor é a classe. As classes são representadas pelos números 1, 2 ou 3.

```
plt.show()
```

Desta forma, ao realizar os comandos na ordem indicada, teremos o seguinte código fonte que carrega o arquivo caracteristicas.txt presente no mesmo diretório do script Python:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Lendo o arquivo de características
dados = pd.read_csv('caracteristicas.txt', header=None)
dados.columns = ['X', 'Y', 'Classe']

# Criando um array para cada classe
classes = []
for i in range(1, 4):
    classes.append(dados[dados['Classe'] == i])

# Plotando as classes
marcadores = ['*', '-', '+']
for i, classe in enumerate(classes):
    plt.scatter(classe['X'], classe['Y'], marker=marcadores[i])

plt.show()
```

É importante ressaltar que se o arquivo estiver em outro local, será necessário especificar o caminho completo para o arquivo.

UNIDADE 3:CLASSIFICADORES

Classificadores são algoritmos que fazem parte do conjunto de técnicas de aprendizado supervisionado em Inteligência Artificial. Eles são utilizados para classificar uma instância em uma ou mais classes pré-definidas, com base em um conjunto de características extraídas dessa instância [1].

Segundo Goodfellow [2], os classificadores têm como objetivo encontrar uma função que mapeia as características de entrada para as classes de saída. Essa função é treinada a partir de um conjunto de dados rotulados, ou seja, um conjunto de dados em que as classes de cada instância já são conhecidas.

Os classificadores são algoritmos que têm como objetivo classificar uma instância em uma ou mais classes pré-definidas, com base em um conjunto de características extraídas dessa instância. Existem diversos tipos de classificadores, cada um com suas particularidades e aplicações.

Um exemplo dos classificadores mais simples é o classificador linear, que procura dividir o espaço de características em duas ou mais regiões de decisão [9]. Essas regiões são definidas por um hiperplano, que é uma generalização da reta para espaços com mais de duas dimensões. O classificador linear é adequado para problemas em que as classes são linearmente separáveis.

Um conhecido classificador são as redes neurais [11] que é inspirado no funcionamento do cérebro humano e é composto por um conjunto de neurônios artificiais interconectados. Cada neurônio recebe um conjunto de características da instância de entrada e produz uma saída que é combinada com as saídas dos demais neurônios para produzir a classe de saída. Na próxima seção são descritos os diferentes tipos de classificadores.

3.1 TIPOS DE CLASSIFICADORES

Os classificadores são algoritmos usados em aprendizado de máquina para prever uma categoria ou classe de um conjunto de dados de entrada. Eles podem ser divididos em duas categorias principais: classificadores supervisionados e não supervisionados [1, 9,10].

O primeiro grupo é usado quando os dados de treinamento já possuem rótulos de classe, ou seja, cada exemplo do conjunto de treinamento já possui uma categoria atribuída. Esse tipo de classificador é mais comumente usado em tarefas de classificação de imagens, detecção de fraudes em cartões de crédito e diagnóstico médico.

Já os classificadores não supervisionados são usados quando os dados de treinamento não possuem rótulos de classe pré-definidos. Neste caso, o algoritmo agrupa os dados em clusters ou categorias, com base em suas características e atributos. Esse tipo de classificador é usado em tarefas como análise de sentimentos em textos, detecção de anomalias em dados financeiros e segmentação de imagens [1,6,12].

Os algoritmos de aprendizado de máquina supervisionados mais populares são as árvores de decisão, as redes neurais, as máquinas de vetores de suporte e os métodos de regressão logística. Já os classificadores não supervisionados mais comuns são o K-means, o DBSCAN, o Mean-Shift e o agrupamento hierárquico. [1,11].

Em geral, o desempenho dos classificadores supervisionados depende da qualidade dos dados de treinamento e do número de exemplos de treinamento. Já os classificadores não supervisionados são úteis quando a estrutura dos dados não é conhecida previamente e o objetivo é descobrir padrões e estruturas

ocultas nos dados. Ambos os tipos de classificadores são muito importantes na aplicação de inteligência artificial em indústrias e em muitas outras áreas, como robótica e processamento de linguagem natural. [1,5,9]

3.2 EXEMPLOS DE CLASSIFICADORES

Um tipo de classificador bastante utilizado é o classificador baseado em árvores de decisão. Segundo Hastie et al. [10], esse tipo de classificador constrói uma árvore em que cada nó interno representa uma decisão baseada em uma característica do conjunto de dados, e cada folha representa uma classe. A partir de uma instância de teste, o classificador percorre a árvore seguindo os caminhos correspondentes às características da instância até chegar a uma folha, que determina a classe da instância.

Outro tipo de classificador é o classificador baseado em regras. Segundo Witten et al. [12], esse tipo de classificador é composto por um conjunto de regras do tipo "se-então", em que cada regra representa uma decisão baseada em um conjunto de características do conjunto de dados. A partir de uma instância de teste, o classificador verifica quais regras são satisfeitas e combina as classes correspondentes para produzir a classe de saída.

Por fim, um último tipo de classificador é o classificador baseado em instâncias. Segundo Aha et al. [13], esse tipo de classificador armazena todo o conjunto de dados de treinamento e classifica uma instância de teste com base nas classes das instâncias mais próximas a ela no espaço de características. Esse tipo de classificador é adequado para problemas em que as fronteiras entre as classes são complexas e não podem ser representadas por uma função simples.

3.3 PRINCIPIOS DA CLASSIFICAÇÃO DE PADRÕES EM PYTHON

Para utilizar o python em classificação de padrão é comum separar os dados em dois conjuntos de dados, um para realizar o treinamento e um para avaliar o poder de aprendizado do algoritmo chamado de conjunto de testes.

Desta forma, a primeira etapa é ler o arquivo de dados 'caracteristicas.txt' usando a biblioteca pandas e separar o conjunto de dados em conjuntos de treinamento e teste.

```
import pandas as pd

# Lê o arquivo de dados
df = pd.read_csv('caracteristicas.txt', header=None, names=['X', 'Y', 'Classe'])
# Divide o conjunto de dados em treinamento e teste
mask = np.random.rand(len(df)) < 0.8
train = df[mask]
test = df[~mask]
```

Em seguida, o conjunto de treinamento é dividido em diferentes conjuntos de dados, um para cada classe. Considerando o mesmo arquivo utilizado no exemplo anterior com três classes, utiliza-se o seguinte através do comando:

```
# Separa as classes em diferentes dataframes
class1 = train[train['Classe'] == 1]
class2 = train[train['Classe'] == 2]
class3 = train[train['Classe'] == 3]
```

Uma vez separado os conjuntos de treinamento de cada uma das classes, o programador poderá apresentar os conjuntos de treino de cada classe para treinar o classificador e em seguida poderá avaliar a taxa de acerto do classificador. Desta forma, o código completo que realiza o carregamento de dados e cria variáveis separando os conjuntos de treinamento para cada uma das classes e o conjunto de testes é apresentado na seguinte sequencia de comandos:

```
import pandas as pd

# Lê o arquivo de dados
df = pd.read_csv('caracteristicas.txt', header=None, names=['X', 'Y', 'Classe'])

# Divide o conjunto de dados em treinamento e teste
mask = np.random.rand(len(df)) < 0.8
train = df[mask]
test = df[~mask]

# Separa as classes em diferentes dataframes
class1 = train[train['Classe'] == 1]
class2 = train[train['Classe'] == 2]
class3 = train[train['Classe'] == 3]
```

UNIDADE 4: CLASSIFICADORES SUPERVISIONADOS

Os classificadores supervisionados são uma classe de algoritmos de aprendizado de máquina que são usados para prever a classe de saída (ou rótulo) de um conjunto de dados de entrada [1].

Esses algoritmos são chamados de supervisionados porque o processo de treinamento envolve o uso de um conjunto de dados de entrada rotulado com classes conhecidas. O objetivo é aprender um modelo que possa generalizar para prever as classes de novos dados que nunca foram vistos antes[9].

Os classificadores supervisionados são amplamente usados em áreas como reconhecimento de fala, visão computacional, processamento de linguagem natural, detecção de fraudes, diagnóstico médico e muitas outras. Eles operam por meio de um processo iterativo de treinamento, onde o modelo é ajustado para minimizar a diferença entre a classe prevista e a classe real dos dados de entrada.

A precisão do modelo é avaliada em um conjunto de dados de teste, que não foi usado durante o processo de treinamento, para garantir que o modelo possa generalizar para dados que nunca foram vistos antes, podendo ser divididos em classificadores paramétricos e não paramétricos [1] [9]

Os classificadores paramétricos assumem que os dados seguem uma distribuição de probabilidade específica e usam esses parâmetros para fazer previsões. Esses modelos podem ser simples e eficientes, mas geralmente são limitados pela sua suposição de distribuição [1,10].

Por outro lado, os classificadores não paramétricos não fazem suposições sobre a distribuição dos dados e usam abordagens mais flexíveis para fazer previsões. Eles são mais úteis em situações em que os dados possuem distribuições complexas ou desconhecidas. Esses modelos tendem a ser mais complexos e requerem mais dados de treinamento, mas oferecem maior flexibilidade e precisão em situações complexas [2,11].

Entre os exemplos de classificadores paramétricos mais populares estão as regressões logísticas, as redes neurais e as máquinas de vetores de suporte. Por outro lado, exemplos de classificadores não paramétricos incluem o K-NN (k-vizinhos mais próximos), o kernel de densidade estimada e as árvores de decisão. A escolha entre um classificador paramétrico e não paramétrico depende do conjunto de dados e da natureza do problema em questão [10,11].

4.1 KNN

O cálculo da distância é uma técnica amplamente utilizada em algoritmos de classificação, podendo ser utilizado para classificar padrões em função da proximidade entre as amostras medidas a partir da distância que entre um padrão conhecido e um padrão desconhecido [7]

O mais simples de um classificador não paramétrico que utiliza este princípio é o classificador do vizinho mais próximo (*Nearest Neighborhood* - NN). Este é um método intuitivo em que cada padrão desconhecido a ser classificado é atribuído à classe do vizinho mais próximo dentro do espaço de treinamento [7].

Essa distância pode ser calculada de diversas formas, entre as quais, podem-se destacar as distâncias de Euclidiana, Quarteirão, Hamming, dentre outras.

A distância Euclidiana é uma medida de dissimilaridade que avalia a distância entre dois pontos em um espaço Euclidiano, sendo é uma das medidas mais populares e simples para avaliar a distância entre padrões [7].

Ela é calculada como a raiz quadrada da soma dos quadrados das diferenças entre as coordenadas dos pontos. Formalmente, a distância Euclidiana entre dois padrões P_x e P_y de dimensão n é definida como [7] :

$$D(P_x, P_y) = \sqrt{\sum_{c=1}^{n} (x_i - y_i)^2}$$

A distância quarteirão (também conhecida como distância de Manhattan) recebe esse nome por causa da forma como as ruas são organizadas em Manhattan, Nova York. As ruas são organizadas em um sistema de grade, com as avenidas correndo na vertical e as ruas correndo na horizontal [7].

A distância entre duas interseções ao longo desse sistema de grade é dada pela soma das diferenças em cada coordenada (horizontal e vertical), o que resulta em um percurso que lembra a trajetória feita ao se andar pelas quadras da cidade (quarteirões). Desta forma, esta distância é calculada como a soma dos módulos das diferenças entre os valores de cada característica nos dois vetores.

$$(P_x, P_y) = \sum_{c=1}^{n} |x_i - y_i|$$

Segundo Duda et al [7] o desempenho deste classificador costuma ser muito bom. Contudo, este método é altamente sensível ao ruído, gerando erros de classificação para regiões do espaço característico em que há uma mistura de várias classes.

A abordagem do vizinho mais próximo pode ser estendida para o método do k-ésimo vizinho mais próximo (k-NN).

Conforme seu nome indica, para cada padrão desconhecido determinam-se os k-vizinhos mais próximos. Neste caso, ao invés de assumir a classe do vizinho mais próximo, k vizinhos próximos são determinados, e a classe assumida é aquela que possui a maioria desses vizinhos.

Desta forma a técnica k-NN é mais robusta ao ruído do que o classificador NN. O parâmetro k é determinado para cada tipo de aplicação. Geralmente é estabelecido um valor pequeno, em que k = 3 ou 5 são os valores mais comuns para este parâmetro), porém na realidade o valor de k depende da quantidade de dados disponíveis.

Para criar um código fonte em python que realize a classificação utilizando o algoritmo k-NN, deve-se realizar o seguinte procedimento. Utilizando o código fonte do capítulo anterior, deve-se carregar o arquivo e separar os conjuntos de treinamento e de testes.

Em seguida, deve-se criar uma função para o cálculo de distância. Neste exemplo, criamos uma função para o cálculo da distância euclidiana:

```
# Função para calcular a distância euclidiana entre dois pontos
def euclidean_distance(x, y):
    return np.sqrt(np.sum((x - y)**2))
```

Em seguida, podemos criar uma função para realizar a classificação com k classes. Na definição da função, podemos até definir que caso o usuário não informe a quantidade de vizinhos, o classificador utilizará a quantidade igual a 3 como valor padrão.

```
    # Função para classificar um exemplo usando K-NN
    def knn_classify(observation, k=3):
        distances = []
        for i in range(len(train)):
            dist = euclidean_distance(observation, train.iloc[i][['X', 'Y']])
            distances.append((dist, train.iloc[i]['Classe']))
        # Ordena as distâncias em ordem crescente e pega as k classes mais
    próximas
        sorted_distances = sorted(distances)[:k]
        classes = [x[1] for x in sorted_distances]
        # Retorna a classe mais comum
        return max(set(classes), key=classes.count)
```

Uma vez definido a função knn_classify, podemos realizar a classificação do conjunto de testes utilizando função 'apply' da biblioteca pandas :

```
# Classifica as observações de teste usando o algoritmo k-NN com k=3
test['Predicted'] = test[['X', 'Y']].apply(lambda x: knn_classify(x, 3), axis=1)
```

Com o vetor classificado, ele pode ser usado para realizar o cálculo da taxa de acerto do classificador no conjunto de testes:

```
    # Calcula a acurácia da classificação
    accuracy = len(test[test['Classe'] == test['Predicted']]) / len(test)
    print('Acurácia:', accuracy)
```

Desta forma, o código completo deste classificador é descrito a seguir:

```
import numpy as np
import pandas as pd
```

```python
# Função para calcular a distância euclidiana entre dois pontos
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))

# Função para classificar uma observação usando o algoritmo k-NN
def knn_classify(observation, k=3):
    distances = []
    for i in range(len(train)):
        dist = euclidean_distance(observation, train.iloc[i][['X', 'Y']])
        distances.append((dist, train.iloc[i]['Classe']))
    # Ordena as distâncias em ordem crescente e pega as k classes mais próximas
    sorted_distances = sorted(distances)[:k]
    classes = [x[1] for x in sorted_distances]
    # Retorna a classe mais comum
    return max(set(classes), key=classes.count)

# Lê o arquivo de dados
df = pd.read_csv('caracteristicas.txt', header=None, names=['X', 'Y', 'Classe'])

# Divide o conjunto de dados em treinamento e teste
mask = np.random.rand(len(df)) < 0.8
train = df[mask]
test = df[~mask]

# Classifica as observações de teste usando o algoritmo k-NN com k=3
test['Predicted'] = test[['X', 'Y']].apply(lambda x: knn_classify(x, 3), axis=1)

# Calcula a acurácia da classificação
accuracy = len(test[test['Classe'] == test['Predicted']]) / len(test)
print('Acurácia:', accuracy)
```

4.2 NATIVE BAYES

A classificação bayesiana é um exemplo de abordagem supervisionada que utiliza os conceitos da teoria de decisão estatística para estabelecer fronteiras de decisões entre classes de padrões[7,14].

Seja $\Omega = \{\omega_1, \ldots, \omega_c\}$ um conjunto de c classes. Seja também x um vetor de atributos com n características, em que cada uma delas é uma variável aleatória, $p(x \mid \omega_j)$ a função de densidade de probabilidade condicional do padrão (função de verossimilhança da classe ωk) x dado a classe ω_j e P(ω_j) a probabilidade de ocorrência a priori da classe ω_j.

A regra de Bayes permite estabelecer a probabilidade a posteriori $P(\omega_j \mid x)$ de ocorrer a classificação na classe ωj dado o vetor x em função da probabilidade a priori P(ω_j) [7]:

$$P(\omega j \mid x) = \frac{(p(x \mid \omega_j) \cdot P(\omega_j))}{p(x)},$$

em que a função de densidade de probabilidade da mistura de classes é:

$$p(x) = \sum_{j=1}^{c} c\ (p(x \mid \omega_j) \cdot P(\omega_j)).$$

A regra de decisão de Bayes afirma que dado x, devemos atribuí-lo à classe ωj, de tal forma que $P(\omega_j \mid x)$ seja máxima .

Sendo p(x) um fator de normalização na Equação 1, e denotando ĉ a regra de decisão, para o caso particular de duas classes apenas, podemos escrever a regra de decisão de Bayes como [14]:

$$\hat{c}(x) = \begin{cases} \omega_1 \text{ se } p(x \mid \omega_1)\, P(\omega_1) > p(x \mid \omega_2)\, P(\omega_2) \\ \omega_2 \text{ se } p(x \mid \omega_1)\, P(\omega_1) < p(x \mid \omega_2)\, P(\omega_2) \end{cases}$$

Uma variante do classificador bayesiano é o naive bayes que assume que as características (ou atributos) dos dados de entrada são independentes entre si, dado a classe a que pertencem. Esta suposição é chamada de independência condicional e permite que a probabilidade conjunta da entrada seja calculada como o produto das probabilidades condicionais de cada característica, dada a classe a que pertencem [7].

O classificador naive Bayes é uma variante do classificador bayesiano que assume que as características (ou atributos) dos dados de entrada são independentes entre si, dado a classe a que pertencem. Esta suposição é chamada de independência condicional e permite que a probabilidade conjunta da entrada seja calculada como o produto das probabilidades condicionais de cada característica, dada a classe a que pertencem.

$$p(x_i \mid \omega_j) = \prod_{j=1}^{N} p(x_i \mid \omega_j)$$

O classificador naive Bayes estima as densidades de probabilidade condicionais $p(x_i \mid \omega_j)$ de cada característica x_i para cada classe ω_j . Em seguida, ele utiliza a regra de Bayes para calcular as probabilidades a posteriori $P(\omega_j \mid x)$ e atribuir uma classe para cada entrada.

O naive Bayes é um algoritmo computacionalmente eficiente e fácil de implementar, sendo bastante utilizado em aplicações práticas. Por outro lado, o classificador bayesiano pode ser mais preciso do que o naive Bayes em algumas situações, mas requer uma modelagem mais cuidadosa das densidades de probabilidade condicionais e das probabilidades *a priori*.

Para criar um código fonte em python que realize a classificação utilizando o algoritmo naive Bayes, deve-se realizar o seguinte procedimento. Utilizando o código fonte do capítulo anterior, deve-se carregar o arquivo e separar os conjuntos de treinamento e de testes.

Inicialmente são calculados as médias e os desvios padrão das características 'X' e 'Y' para cada classe:

```
# Calcula as médias e os desvios padrão
de cada característica para cada classe
mean1 = class1[['X', 'Y']].mean().values
mean2 = class2[['X', 'Y']].mean().values
mean3 = class3[['X', 'Y']].mean().values

std1 = class1[['X', 'Y']].std().values
```

```
std2 = class2[['X', 'Y']].std().values
std3 = class3[['X', 'Y']].std().values
```

A função 'calculate_probability' calcula a probabilidade de uma observação pertencer a uma classe usando a fórmula da distribuição normal.

```
# Função que calcula a probabilidade de uma observação pertencer
a uma classe
def calculate_probability(x, mean, stdev):
    exponent = math.exp(-((x - mean) ** 2 / (2 * stdev ** 2)))
    return (1 / (math.sqrt(2 * math.pi) * stdev)) * exponent
```

A função 'classify' utiliza o algoritmo Naive Bayes para classificar uma observação de teste. Ela calcula a probabilidade de a observação pertencer a cada classe e retorna a classe com maior probabilidade.

```
Função que realiza a classificação usando o algoritmo Naive Bayes
def classify(observation):
    probabilities = []
    for i in range(1, 4):
        p_class = len(train[train['Classe'] == i]) / len(train)
        p_x_given_class = calculate_probability(observation[0], eval('mean' +
str(i))[0], eval('std' + str(i))[0]) * calculate_probability(observation[1],
eval('mean' + str(i))[1], eval('std' + str(i))[1])
        probabilities.append(p_x_given_class * p_class)
    return np.argmax(probabilities) + 1
```

Finalmente, a função 'apply' da biblioteca pandas é usada para realizar a classificação do conjunto de testes que em seguida pode ser usado para realizar o cálculo da taxa de acerto do classificador no conjunto de testes.

```
# Classifica as observações de teste
test['Predicted'] = test.apply(classify, axis=1)

# Calcula a acurácia da classificação
accuracy = len(test[test['Classe'] == test['Predicted']]) /
len(test)
print('Acurácia:', accuracy)
```

O código completo deste classificador é listado a seguir:

```
import numpy as np
import math
import pandas as pd
# Função que calcula a probabilidade de uma observação pertencer a uma classe
def calculate_probability(x, mean, stdev):
    exponent = math.exp(-((x - mean) ** 2 / (2 * stdev ** 2)))
    return (1 / (math.sqrt(2 * math.pi) * stdev)) * exponent
# Função que realiza a classificação usando o algoritmo Naive Bayes
def classify(observation):
    probabilities = []
    for i in range(1, 4):
        p_class = len(train[train['Classe'] == i]) / len(train)
        p_x_given_class = calculate_probability(observation[0], eval('mean' +
str(i))[0], eval('std' + str(i))[0]) * calculate_probability(observation[1],
eval('mean' + str(i))[1], eval('std' + str(i))[1])
```

```
            probabilities.append(p_x_given_class * p_class)
        return np.argmax(probabilities) + 1

# Lê o arquivo de dados
df = pd.read_csv('caracteristicas.txt', header=None, names=['X', 'Y', 'Classe'])
# Divide o conjunto de dados em treinamento e teste
mask = np.random.rand(len(df)) < 0.8
train = df[mask]
test = df[~mask]
# Separa as classes em diferentes dataframes
class1 = train[train['Classe'] == 1]
class2 = train[train['Classe'] == 2]
class3 = train[train['Classe'] == 3]
# Calcula as médias e os desvios padrão de cada característica para cada classe
mean1 = class1[['X', 'Y']].mean().values
mean2 = class2[['X', 'Y']].mean().values
mean3 = class3[['X', 'Y']].mean().values
std1 = class1[['X', 'Y']].std().values
std2 = class2[['X', 'Y']].std().values
std3 = class3[['X', 'Y']].std().values
# Classifica as observações de teste
test['Predicted'] = test.apply(classify, axis=1)
# Calcula a acurácia da classificação
accuracy = len(test[test['Classe'] == test['Predicted']]) / len(test)
print('Acurácia:', accuracy)
```

4.3 ÁRVORE DE DECISÃO

O algoritmo de árvore de decisão é um método de aprendizado supervisionado que é utilizado para classificação e regressão. Ele é uma técnica simples e intuitiva que pode ser usada para tomar decisões com base em um conjunto de regras lógicas.

De acordo com Schölkopf & Smola [16], uma árvore de decisão é uma estrutura hierárquica que divide os dados em subconjuntos com base em características ou atributos relevantes para a tarefa de classificação. Cada nó interno da árvore corresponde a um atributo e cada ramo que sai desse nó representa um possível valor desse atributo. Os nós folha correspondem às classes ou rótulos de saída.

O objetivo do algoritmo de árvore de decisão é encontrar a melhor divisão dos dados em subconjuntos usando um critério de impureza, como a entropia ou o índice de Gini. De acordo com Hastie et al [10], a impureza é uma medida de quão misturadas são as classes em um subconjunto de dados. O critério de impureza escolhido é usado para selecionar o melhor atributo para dividir os dados em cada nó interno da árvore.

O processo de construção da árvore começa com um único nó, que contém todos os dados de treinamento. Em seguida, o algoritmo seleciona o melhor atributo para dividir os dados e cria um novo nó interno correspondente a esse atributo. Os dados são divididos em subconjuntos com base no valor do atributo e o processo é repetido para cada subconjunto em cada nó interno da árvore. Esse processo é repetido até que todos os dados sejam classificados em um nó folha.

De acordo com Breiman [22], uma vantagem do algoritmo de árvore de decisão é que ele pode ser facilmente interpretado e visualizado. Além disso, ele é robusto a dados faltantes e pode lidar com conjuntos de dados de alta dimensionalidade.

No entanto, o algoritmo de árvore de decisão também tem algumas limitações, como a tendência de sobreajustar os dados de treinamento e a sensibilidade a pequenas variações nos dados de entrada. Para lidar com esses problemas, várias técnicas foram desenvolvidas, como a poda da árvore e o uso de ensemble de árvores, como o Random Forest.

Os algoritmos de decision tree são um pouco mais complexos do que os apresentados nos algoritmos anteriores.

O Scikit-Learn é uma biblioteca de aprendizado de máquina de código aberto que fornece algoritmos para análise de dados e modelagem estatística.

De acordo com Pedregosa et al. [25], o objetivo do Scikit-Learn é fornecer uma biblioteca acessível e eficiente para usuários e desenvolvedores que desejam criar aplicativos de aprendizado de máquina em Python.

O Scikit-Learn inclui uma ampla gama de algoritmos de aprendizado de máquina, desde regressão linear até aprendizado de máquina profundo, e é projetado para ser fácil de usar e flexível. O Scikit-Learn facilita muito o processo de classificação de padrões, pois possui funções desde a separação dos dados em conjuntos de teste e treino, bem como possui a implementação de vários classificadores, simplificando a vida do desenvolvedor de algoritmos de inteligência artificial.

Para utilizá-lo devemos importar os módulos da biblioteca através do comando import

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

Através deste comando, vamos utilizar a biblioteca não só para realizar a classificação, mas também para realizar a separação dos conjuntos de teste e treino e realizar o cálculo da taxa de acerto deste classificador.

Após o carregamento de dados através do pandas, realiza-se a separação dos conjuntos de treino e teste através dos seguintes comandos

```
# Separação dos dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(df[['X', 'Y']], df['Classe'],
test_size=0.2)
```

Em seguida, pode-se realizar o treinamento do classificador realizando o instanciamento do classificador e realizando a operação fit:

```
# Criação do classificador de árvore de decisão
dtc = DecisionTreeClassifier()
# Treinamento do classificador
dtc.fit(X_train, y_train)
```

E por fim, pode-se realizar a classificação e realizar o cálculo da taxa de acerto:

```
# Criação do classificador de árvore de decisão
dtc = DecisionTreeClassifier()

# Realização da predição dos dados de teste
y_pred = dtc.predict(X_test)

# Cálculo da acurácia da classificação
accuracy = accuracy_score(y_test, y_pred)
print('Acurácia:', accuracy)
```

Assim, o código-fonte completo que realiza a classificação de padrões do arquivo

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Leitura do arquivo de dados
df = pd.read_csv('caracteristicas.txt', header=None, names=['X', 'Y', 'Classe'])

# Separação dos dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(df[['X', 'Y']], df['Classe'],
test_size=0.2)

# Criação do classificador de árvore de decisão
dtc = DecisionTreeClassifier()

# Treinamento do classificador
dtc.fit(X_train, y_train)

# Realização da predição dos dados de teste
y_pred = dtc.predict(X_test)

# Cálculo da acurácia da classificação
accuracy = accuracy_score(y_test, y_pred)
print('Acurácia:', accuracy)
```

UNIDADE 5: CLASSIFICAÇÃO NÃO SUPERVISIONADA

A classificação não supervisionada é um método de aprendizado de máquina que visa agrupar dados em classes ou clusters, sem a necessidade de um conjunto de dados rotulados. De acordo com Hastie et al. [10], a classificação não supervisionada é usada para encontrar estruturas ou padrões desconhecidos nos dados.

A classificação não supervisionada é importante em diversas áreas, como reconhecimento de padrões, processamento de imagens e análise de dados. Ela permite identificar grupos ou padrões em um conjunto de dados sem a necessidade de informações prévias sobre as classes. Segundo Hastie et al[10], "a classificação não supervisionada pode ser utilizada quando não há conhecimento prévio das classes presentes nos dados ou quando se deseja descobrir possíveis classes ou padrões inesperados".

A classificação não supervisionada tem sido amplamente utilizada em diferentes áreas. Em processamento de imagens, por exemplo, ela pode ser utilizada para segmentar imagens em regiões com características semelhantes [26]. Já em análise de dados, a classificação não supervisionada pode ser utilizada para agrupar dados em clusters com características semelhantes, permitindo a identificação de padrões ou anomalias [27].

Os algoritmos de classificação não supervisionada são usados para encontrar grupos de dados semelhantes ou padrões em dados complexos e grandes. Segundo Bishop [17], esses algoritmos operam encontrando padrões nos dados, calculando distâncias entre os pontos de dados e agrupando os pontos em clusters.

Existem vários métodos de classificação não supervisionada, incluindo o k-médias (k-means) e o hierarchical clustering. De acordo com Alpaydin [9], o método k-médias é um algoritmo simples, rápido e eficiente que divide os dados em k clusters e é amplamente utilizado em aplicações de clustering.

O k-means é um algoritmo de clusterização amplamente utilizado em mineração de dados e aprendizado de máquina. Ele é usado para agrupar dados não rotulados em clusters com base em suas características e é amplamente utilizado em aplicações como segmentação de clientes, agrupamento de imagens e análise de dados genéticos.

5.1 K-MÉDIAS

Segundo a referência Hastie et al. [1], o algoritmo k-médias funciona da seguinte maneira: inicialmente, ele seleciona k pontos aleatórios como centróides iniciais para os clusters. Em seguida, cada ponto de dados é atribuído ao centróide mais próximo com base em alguma medida de distância, como a distância euclidiana.

Depois de todos os pontos serem atribuídos a um cluster, o centróide de cada cluster é recalculado como a média de todos os pontos atribuídos a ele. O processo de atribuir pontos

a clusters e recalcular os centróides é repetido até que a convergência seja alcançada, ou seja, até que os centróides deixem de mudar significativamente.

Hastie et al.[1] também afirma que o k-means é um algoritmo simples e eficiente, mas pode ser sensível à inicialização dos centróides e pode produzir resultados diferentes para diferentes execuções com diferentes pontos iniciais. Além disso, é importante definir o valor de k, ou seja, o número de clusters desejados, de maneira apropriada, pois um valor muito alto ou muito baixo pode afetar a qualidade dos resultados.

5.2 PRÁTICA EM PYTHON

Um exemplo de código-fonte em python capaz de utilizar o k-medias é mostrado a seguir:

```python
import pandas as pd
from sklearn.cluster import KMeans
import numpy as np

# Carregando os dados com pandas
df = pd.read_csv('dados.csv')

# Convertendo o DataFrame para um array NumPy
X = df.to_numpy()

# Definindo o número de clusters
k = 3

# Instanciando o modelo K-means
kmeans = KMeans(n_clusters=k)

# Treinando o modelo
kmeans.fit(X)

# Obtendo os centróides dos clusters
centroids = kmeans.cluster_centers_

# Obtendo as labels de cada exemplo
labels = kmeans.labels_

# Imprimindo os resultados
print("Centroides dos clusters:")
print(centroids)
print("\nLabels dos exemplos:")
print(labels)
```

UNIDADE 6: REDES NEURAIS

Uma rede neural artificial é definida como um modelo matemático que consiste em um conjunto de elementos computacionais não lineares, conhecidos como neurônios, que operam em paralelo e são conectados por ligações caracterizadas por diferentes pesos. Conforme descrito por Haykin [11], "a topologia da rede, as características dos neurônios e as regras de aprendizagem ou treinamento são os principais fatores que determinam o desempenho da rede neural".

De acordo com Bishop [17], um neurônio artificial é um bloco básico de uma rede neural, que recebe entradas ponderadas pelos pesos das conexões de entrada. A saída do neurônio é calculada pela aplicação de uma função de ativação não linear ao nível de ativação produzido pela soma ponderada das entradas.

A topologia da rede neural é responsável pela interconexão dos neurônios, especificando como as entradas, saídas e camadas escondidas são conectadas. A topologia da rede é um dos fatores cruciais que determinam o desempenho da rede neural [11].

6.1 PERCEPTRON SIMPLES

O perceptron simples é um modelo de classificador linear binário usado em aprendizado supervisionado. É baseado em um único neurônio artificial que pode classificar objetos em duas classes distintas [7].

O perceptron simples é capaz de aprender a separar dados que são linearmente separáveis em duas classes, atualizando iterativamente os pesos sinápticos do neurônio a partir de exemplos de treinamento rotulados.

De acordo com Duda & Hart [7], o algoritmo perceptron é baseado em um neurônio artificial que recebe múltiplas entradas, cada uma com um peso sináptico correspondente. O neurônio computa uma soma ponderada dessas entradas e passa o resultado por uma função de ativação.

Geralmente usa-se uma função de limiar, para produzir a saída binária final. A função de limiar determina se a saída é 0 ou 1, dependendo se a soma ponderada das entradas está abaixo ou acima de um valor de limiar predefinido.

O perceptron simples é capaz de aprender a classificar exemplos de treinamento linearmente separáveis em duas classes, ajustando iterativamente os pesos sinápticos com base em uma regra de atualização do tipo gradiente descendente. A regra de atualização é projetada para minimizar a diferença entre as saídas do perceptron e os rótulos corretos dos exemplos de treinamento.

Em virtude da sua simplicidade, é possível desenvolver um classificador perceptron simples utilizando apenas o numpy. O código-fonte de um classificador de um espaço de características armazenado no arquivo características.txt é mostrado a seguir:

```
import pandas as pd
import numpy as np
# Função para obter os dados do arquivo de características
def get_data(file_path):
    data = []
    with open(file_path, 'r') as file:
        for line in file:
            features = line.split(',')
            x = [float(feature) for feature in features[:-1]]
            y = int(features[-1])
            data.append((x, y))
    return data
# Função para treinar o modelo de Perceptron Simples
def train_perceptron(data, learning_rate=0.1, epochs=100):
    n_features = len(data[0][0])
    weights = np.zeros(n_features)
    bias = 0
    for epoch in range(epochs):
        for x, y in data:
            y_pred = np.dot(weights, x) + bias
            y_pred = 1 if y_pred > 0 else 0
            error = y - y_pred
            weights += learning_rate * error * np.array(x)
            bias += learning_rate * error
    return weights, bias
# Função para testar o modelo de Perceptron Simples
def test_perceptron(data, weights, bias):
    n_samples = len(data)
    n_correct = 0
    for x, y in data:
        y_pred = np.dot(weights, x) + bias
        y_pred = 1 if y_pred > 0 else 0
        if y_pred == y:
            n_correct += 1
    accuracy = n_correct / n_samples
    return accuracy
# Ler o arquivo de características com Pandas
df = pd.read_csv('caminho/do/arquivo/caracteristicas.txt', header=None)
# Extrair as colunas de características e rótulos
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
# Converter os rótulos para valores binários (-1 ou 1)
y = np.where(y == 0, -1, 1)
# Definir os parâmetros do modelo
lr = 0.1 # taxa de aprendizagem
n_epochs = 100 # número de épocas de treinamento
# Inicializar o modelo e treinar
n_features = X.shape[1]
perceptron = Perceptron(n_features)
perceptron.train(X, y, lr, n_epochs)

# Realizar a classificação de um novo exemplo
x_new = np.array([1.2, 2.3, 0.4])
y_pred = perceptron.predict(x_new)
print("Rótulo previsto:", y_pred)
```

6.2 PERCEPTRON MULTI-CAMADA

O Perceptron Multi-camada (Multi-Layer Perceptron - MLP) é uma rede neural artificial comumente utilizada para separar dados não-linearmente separáveis. Para iniciar o processo de aprendizagem da rede, é necessário selecionar um conjunto de amostras das classes padrões (conjunto de treinamento) a serem reconhecidos pela rede e as saídas desejadas correspondentes [11].

É necessário selecionar amostras representativas de cada classe e um número suficiente dessas amostras para que a rede possa aprender a identificar os padrões [7]

Essa rede apresenta três ou mais camadas de neurônios, incluindo um conjunto de unidades sensoriais (nós de fonte) que constituem a camada de entrada, uma ou mais camadas ocultas de nós computacionais e uma camada de saída.

Sua topologia é completamente interconectada na direção da camada de saída sem retroalimentação. O sinal de entrada se propaga através da rede, camada por camada [7]

A definição do número de neurônios das camadas de entrada e saída é realizada de acordo com o problema em questão. O número de neurônios da camada intermediária, ou mesmo o número de camadas intermediárias, é definido de forma intuitiva, não havendo uma regra que defina seu número. Se a quantidade de neurônios escolhidos for muito pequena, apenas alguns neurônios podem especializar-se em características não úteis, como ruído. Se o número de neurônios for insuficiente, a rede pode não conseguir aprender os padrões desejados [7].

O neurônio individual é o bloco construtivo de cada camada e é caracterizado principalmente por sua função de ativação. As funções de ativação mais comumente utilizadas entre as camadas da MLP são as funções logística e sigmoide.

As redes MLP são projetadas para aproximar uma relação entre entrada e saída não conhecida, através dos pesos de cada conexão via regras de aprendizagem. Uma característica de grande importância desse modelo é o aprendizado supervisionado baseado em duas etapas, a saber, propagação e adaptação.

A propagação ocorre na fase de treinamento da rede e consiste em fornecer à rede um conjunto de estímulos (padrões de entradas) e a saída desejada correspondente ao padrão de entrada apresentado. Nessa fase, o primeiro padrão de entrada é propagado até a saída. Durante esse passo, os pesos sinápticos não mudam de valor.

Na fase de adaptação, o sinal do erro é computado (resultado da diferença entre a saída desejada e saída real da rede) e transmitido de volta para cada neurônio da camada intermediária que contribuiu para a saída obtida. Sendo assim, cada neurônio da camada intermediária recebe somente uma parte do erro total, conforme a contribuição relativa que o neurônio teve na saída gerada [7].

Este processo repete-se camada por camada, até que cada neurônio da rede receba o seu valor correspondente. Tal processo é conhecido como retropropagação do erro, pois, o aprendizado baseia-se na propagação retrógrada do erro, contra a direção das conexões sinápticas da rede.

Os pesos existentes nas conexões entre os neurônios são atualizados de acordo com o erro recebido pelo neurônio associado. Esta atualização é um processo iterativo em que a rede ajusta seus pesos até que a informação do ambiente seja aprendida. O processo de aprendizagem para quando a saída obtida pela rede neural, para cada um dos padrões de entrada, for próxima o suficiente da saída desejada, de forma que a diferença entre ambas seja aceitável. Esta diferença é obtida pelo cálculo do erro quadrático médio [7].

A formulação matemática do algoritmo de retropropagação do erro pode ser encontrada em Hinton et al. [28]. O processo de treinamento é repetido até que a rede neural atinja um nível aceitável de precisão na classificação de novos dados, ou até que um critério pré-definido de parada seja atingido.

As redes MLP são amplamente utilizadas em tarefas de classificação, previsão e aproximação de funções. A sua capacidade de generalização permite que a rede possa ser utilizada para dados não vistos no processo de treinamento. Por causa disto, a MLP é um modelo amplamente utilizado em tarefas de classificação, previsão e aproximação de funções, devido à sua capacidade de generalização e aprendizado não-linear, tornando-a uma ferramenta valiosa em diversas áreas, como finanças, medicina, engenharia e outras.

6.3 PRÁTICA SCILEARN

Uma forma simples para implementar um classificador MLP consiste em utilizar o método MLPClassifier da biblioteca sci-learn.

No código a seguir é criada uma MLP com 2 camadas ocultas de 4 neurônios cada, e parametrizamos o classificador para aprender em no máximo 1000 épocas através do comando:

```
clf = MLPClassifier(hidden_layer_sizes=(4, 4), max_iter=1000)
```

Depois, treinamos o classificador com o conjunto de dados utilizando o comando fit e realizamos a classificação de um novo registro utilizando o método predict:

```
clf.fit(X, y)

# Classifica um novo registro
novo_registro = [[0.7, 0.5, 0.2]]
print(clf.predict(novo_registro))
```

Assim, um exemplo de código fonte utilizado para realizar a classificação utilizando a MLP é mostrado a seguir:

```python
from sklearn.neural_network import MLPClassifier
import pandas as pd

# Carrega o arquivo csv com o pandas
dataset = pd.read_csv('caracteristicas.csv', header=None)

# Divide o conjunto de dados em features (X) e classes (y)
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

# Instancia o classificador MLP com 2 camadas ocultas de 4 neurônios cada
clf = MLPClassifier(hidden_layer_sizes=(4, 4), max_iter=1000)

# Treina o classificador
clf.fit(X, y)

# Classifica um novo registro
novo_registro = [[0.7, 0.5, 0.2]]
print(clf.predict(novo_registro))
```

UNIDADE 7: DEEP LEARNING

Deep learning é uma técnica de aprendizado de máquina que se baseia em redes neurais artificiais com múltiplas camadas ocultas para extrair características e realizar tarefas de alto nível em dados brutos, como imagens, sons e textos [2]. Essa técnica permite que modelos de aprendizado de máquina possam ser treinados em grandes quantidades de dados e, assim, aprender representações hierárquicas de diferentes níveis de abstração dos dados, o que possibilita um melhor desempenho em tarefas complexas [29].

As redes neurais artificiais profundas (DNNs) são compostas por várias camadas ocultas que operam na transformação da entrada em uma representação abstrata. Cada camada recebe como entrada a saída da camada anterior e realiza uma operação não-linear sobre essa entrada para gerar uma saída que é enviada para a próxima camada [2] O número de camadas em uma rede neural profunda pode variar dependendo da complexidade da tarefa que se deseja resolver e da quantidade de dados disponíveis para treinamento.

O processo de treinamento de uma rede neural profunda envolve a definição da arquitetura da rede, a inicialização dos pesos das conexões entre os neurônios e a aplicação de um algoritmo de otimização para ajustar esses pesos durante o processo de treinamento, a fim de minimizar uma função de custo que quantifica a diferença entre as saídas da rede e as saídas desejadas para cada entrada [2].

Uma das técnicas mais utilizadas para treinamento de DNNs é a retropropagação do erro (backpropagation), que consiste em propagar o erro da saída da rede para as camadas ocultas e ajustar os pesos em cada camada de acordo com a contribuição de cada neurônio para o erro total [28]. Além disso, outras técnicas como dropout, normalização de batch e regularização também são frequentemente utilizadas para melhorar o desempenho e prevenir overfitting [2].

As DNNs têm sido aplicadas com sucesso em diversas áreas, incluindo visão computacional, processamento de linguagem natural, reconhecimento de fala e processamento de áudio. Devido à sua capacidade de extrair características complexas de dados brutos e resolver tarefas complexas, as DNNs têm sido consideradas como um dos principais impulsionadores da inteligência artificial nos últimos anos [29].

7.1 BIBLIOTECAS

Existem diversas bibliotecas em Python que permitem trabalhar com aprendizado profundo. Algumas das mais populares são:

- Tensorflow: uma biblioteca de código aberto desenvolvida pelo Google, que se tornou uma das mais populares para o desenvolvimento de modelos de aprendizado profundo [30];
- Keras: uma biblioteca de alto nível para construir redes neurais que roda sobre o TensorFlow e outras bibliotecas de aprendizado profundo [33];
- PyTorch: uma biblioteca de aprendizado de máquina de código aberto baseada no Torch, que é usada tanto para pesquisa quanto para produção [31];
- Theano: uma biblioteca de aprendizado profundo de código aberto, que permite definir, otimizar e avaliar expressões matemáticas envolvendo matrizes multidimensionais de forma eficiente [23];
- Caffe: uma biblioteca de aprendizado profundo de código aberto, desenvolvida pelo Berkeley Vision and Learning Center, que é especialmente útil para aplicações de visão computacional. [33]
- MXNet: uma biblioteca de aprendizado profundo escalável e de código aberto, que permite construir modelos de aprendizado profundo em diversas plataformas e dispositivos. (cite{chen2015mxnet})

Na seção a seguir serão apresentadas as instruções para se instalar as principais bibliotecas de aprendizado profundo: Tensorflow, Keras e Pytorch.

7.2 INSTALAÇÃO DAS BIBLIOTECAS EM PYTHON

As bibliotecas TensorFlow, Keras e PyTorch podem ser instaladas através do pip, que é o gerenciador de pacotes padrão do Python. Você pode usar os seguintes comandos no terminal (ou no prompt de comando do Windows) para instalar cada uma das bibliotecas:

- TensorFlow: pip install tensorflow
- Keras: pip install keras
- PyTorch: pip install torch

No entanto, é sempre recomendado verificar se as versões instaladas são compatíveis com o seu ambiente de desenvolvimento e se estão funcionando corretamente. Para verificar as versões instaladas, você pode usar o seguinte comando:

- TensorFlow: pip show tensorflow
- Keras: pip show keras
- PyTorch: pip show torch

UNIDADE 8: NOVAS BIBLIOTECAS UTLIZADAS EM INTELIGÊNCIA ARTIFICIAL

Neste capítulo serão apresentadas algumas bibliotecas de aprendizado profundo disponíveis para utilização na Internet. Iniciaremos este capítulo apresentando o TENSORFLOW, em seguida o KERAS e PYTORCH e para concluir iremos demonstrar a utilização destas bibliotecas em python e como utilizá-las em aplicações de IA em dispositivos móveis.

8.1 TENSORFLOW

O TensorFlow é uma biblioteca de código aberto para computação numérica, que utiliza grafos computacionais para representar e executar cálculos. Ele foi desenvolvido pelo Google Brain Team e é amplamente utilizado para criar modelos de aprendizado de máquina e deep learning.

Segundo Abadi et al. [30], o TensorFlow permite a criação de modelos de aprendizado de máquina em várias plataformas, incluindo CPUs, GPUs e dispositivos móveis. Ele suporta diferentes tipos de redes neurais, como redes neurais convolucionais, redes neurais recorrentes e redes neurais de alimentação direta.

Além disso, o TensorFlow oferece várias ferramentas para otimização de modelos, como otimizadores de gradiente descendente estocástico, normalização de lotes e regularização L1 e L2 [31] Ele também suporta a criação de modelos distribuídos, permitindo o treinamento de grandes conjuntos de dados em várias máquinas.

Em resumo, o TensorFlow é uma ferramenta essencial para a criação de modelos de aprendizado de máquina e deep learning, permitindo que os usuários criem e treinem modelos eficientes e escaláveis em várias plataformas.

8.2 KERAS

Claro! O Keras é uma biblioteca de aprendizado profundo de alto nível, escrita em Python e capaz de ser executada em cima de diferentes backends, incluindo o TensorFlow, CNTK e Theano. Ele foi projetado para permitir a rápida prototipagem de redes neurais profundas, com foco em ser fácil de usar, modular e extensível [31].

De acordo com Chollet [31], o Keras permite que os usuários definam e treinem redes neurais profundas em poucas linhas de código, graças a sua API intuitiva e consistente. Ele fornece uma ampla gama de modelos de rede neural pré-definidos, bem como camadas personalizáveis para criar modelos personalizados. Além disso, o Keras possui recursos para treinamento em GPU, gerenciamento de dados, regularização e visualização de resultados.

Uma das vantagens do Keras é que ele é compatível com a maioria das bibliotecas Python existentes, incluindo NumPy, SciPy e Pandas, tornando-o uma ferramenta flexível para lidar

com problemas de aprendizado de máquina complexos. Além disso, o Keras é amplamente utilizado pela comunidade de aprendizado de máquina e tem uma grande base de usuários, o que significa que há muita documentação e suporte disponíveis.

8.3 PYTORCH

PyTorch é uma biblioteca de aprendizado de máquina de código aberto que oferece uma estrutura flexível para desenvolver modelos de aprendizado profundo. PyTorch permite a definição e treinamento de modelos em uma variedade de domínios de aplicação, incluindo processamento de linguagem natural, visão computacional, processamento de áudio, entre outros. É desenvolvido pelo Facebook AI Research e tem sido amplamente adotado pela comunidade de pesquisa e indústria.

PyTorch se destaca pela sua facilidade de uso e clareza de código, bem como pela sua flexibilidade em permitir a construção de modelos complexos e personalizados. Ele utiliza uma abordagem de programação dinâmica, o que significa que as operações do modelo são executadas em tempo de execução, em vez de serem definidas em um grafo estático antes da execução. Essa abordagem permite que os modelos PyTorch sejam mais fáceis de depurar e personalizar em comparação com outras bibliotecas de aprendizado de máquina.

Segundo Paszke et al. [32], a estrutura do PyTorch é composta por dois principais componentes: a biblioteca PyTorch Tensor e o módulo PyTorch autograd. O PyTorch Tensor é uma matriz multidimensional com suporte a operações de alto nível, como cálculo diferencial, indexação e operações de redução.

O PyTorch autograd é responsável pelo cálculo automático de gradientes para backpropagation, que é uma técnica comum usada no treinamento de modelos de aprendizado profundo.

8.4 APLICAÇÃO DO TENSORFLOW NO ANDROID

Existem várias formas de utilizar o TensorFlow em aplicações Android. Uma das mais comuns é a utilização da biblioteca TensorFlow Lite. O TensorFlow Lite é uma versão otimizada do TensorFlow para dispositivos móveis e embarcados, que permite executar modelos de aprendizado de máquina de forma eficiente em dispositivos com recursos limitados.

Para utilizar o TensorFlow Lite em aplicações Android, é necessário adicionar a dependência do TensorFlow Lite no arquivo de configuração do projeto Gradle e, em seguida, copiar o modelo treinado para o diretório de recursos do aplicativo. Depois disso, é possível carregar o modelo usando a API do TensorFlow Lite e realizar inferências nos dados de entrada.

Além disso, o TensorFlow também oferece uma API Java para Android, que permite integrar o TensorFlow diretamente no aplicativo Android. Isso permite a criação de modelos personalizados e a utilização do hardware do dispositivo para acelerar a execução dos modelos.

Segue abaixo um exemplo de como importar o TensorFlow Lite no Flutter:

Adicione a dependência do TensorFlow Lite, incluindo a linha tflite: ^1.0.0 no arquivo pubspec.yaml:

```
dependencies:
  flutter:
    sdk: flutter
  tflite: ^1.0.0 # Adicione esta linha
```

Execute o comando flutter pub get para instalar a dependência e importe o pacote tflite no arquivo que deseja utilizar:

```
import 'package:tflite/tflite.dart';
```

Utilize as funções do TensorFlow Lite disponíveis no pacote tflite, como por exemplo:

```
// Carrega um modelo TensorFlow Lite
await Tflite.loadModel(
  model: 'assets/model.tflite',
  labels: 'assets/labels.txt',
);

// Executa uma inferência no modelo com uma imagem
var output = await Tflite.runModelOnImage(
  path: 'path/to/image.jpg',
);

// Libera o modelo da memória
Tflite.close();
```

O código acima é apenas um exemplo básico e que pode variar dependendo da sua aplicação e das funcionalidades do TensorFlow Lite que você deseja utilizar. Além disso, certifique-se de ter os arquivos do modelo e do arquivo de rótulos em seu projeto e que os caminhos para eles estejam corretos.

REFERÊNCIAS

BIBLIOGRAFIA BÁSICA

{BIBLIOGRAPHY}

[1] Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach. Prentice Hall Press.

[1] A. Haddadi, P. Hui, and T. Henderson, "Artificial intelligence in industry: state of the art and future directions," IEEE Transactions on Industrial Informatics, vol. 15, no. 4, pp. 2224-2234, 2019.

[2] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.

[3] P. Sermanet, C. Lynch, Y. Chebotar, J. Hsu, E. Jang, S. Schaal, and S. Levine, "Time-contrastive networks: self-supervised learning from video," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2914-2923, 2018.

[4] D. Jurafsky and J. H. Martin, Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 3rd ed.

[5] O. Khatib, "Robots in the loop: challenges and opportunities for robot-to-robot interaction," Science Robotics, vol. 3, no. 18, pp. eaat5976, 2018.

[6] K. N. Plataniotis and A. N. Venetsanopoulos, "Color Image Processing and Applications", Springer Science & Business Media, 2000.

[7] R. O. Duda, P. E. Hart and D. G. Stork, "Pattern Classification," John Wiley & Sons, Inc., 2012.

[8] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[9] Alpaydin, E. (2010). Introduction to Machine Learning. MIT Press.

[10] Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer.

[11] Haykin, S. (1999). Neural Networks: A Comprehensive Foundation. Prentice Hall.

[12] Witten, I. H., Frank, E., & Hall, M. A. (2016). Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann.

[13] Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-based learning algorithms. Machine learning, 6(1), 37-66.

[14] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2016.
[15] B. Schölkopf and A. J. Smola, Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, 2002.

[16] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, no. 7553, pp. 436-444, 2015.
[17] C. M. Bishop, Pattern Recognition and Machine Learning. Springer, 2006.
[18] A. Ng, "Machine learning yearning," 2019. [Online]. Available: https://www.deeplearning.ai/machine-learning-yearning/.
[19] K. Murphy, Machine Learning: A Probabilistic Perspective. MIT Press, 2012.
[20] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning," Springer, 2001.
[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, et al., "Mastering the game of Go with deep neural networks and tree search," Nature, vol. 529, no. 7587, pp. 484-489, 2016.
[22] L. Breiman, "Random forests," Machine Learning, vol. 45, no. 1, pp. 5-32, 2001.
[23] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, et al., "Theano: A CPU and GPU Math Expression Compiler," in Proceedings of the Python for Scientific Computing Conference (SciPy), 2010.
[24] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, et al., "Large scale distributed deep networks," in Proceedings of the 25th International Conference on Neural Information Processing Systems, pp. 1232-1240, 2012.
[25]Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.
[26] JAIN, A. K. Data clustering: 50 years beyond K-means. Pattern Recognition Letters, v. 31, n. 8, p. 651-666, 2010.
[27] NG, A. Y.; JORDAN, M. I.; WEISS, Y. On spectral clustering: Analysis and an algorithm. In: Advances in Neural Information Processing Systems. v. 14. Cambridge: MIT Press, 2002. p. 849-856.
[28] G. E. Hinton, D. E. Rumelhart, and R. J. Williams, "Learning representations by back-propagating errors," Nature, vol. 323, no. 6088, pp. 533–536, 1986.
[29] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.
[30] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). TensorFlow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.
[31] Chollet, F., Allaire, J. J., & others. (2018). Deep learning with R. Manning Publications Co.
[32] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems (pp. 8024-8035).
[33] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... & Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia (pp. 675-678).
[34] F. Chollet et al., "Keras," 2015. [Online]. Available: https://keras.io/. [Accessed: Mar. 12, 2023].

[4] HAN, J.; KAMBER, M.; PEI, J. Data Mining: Concepts and Techniques. 3rd ed. San Francisco: Morgan Kaufmann, 2012.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed

Systems," arXiv preprint arXiv:1603.04467, 2016. [Online]. Available: https://arxiv.org/abs/1603.04467. [Accessed: Mar. 12, 2023].

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024-8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/9015ca1c0e4b5f3e4e7934e9e4744eba-Paper.pdf. [Accessed: Mar. 12, 2023].

[13] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, Oct. 2011. [Online]. Available: https://jmlr.csail.mit.edu/papers/volume12/pedregosa11a/pedregosa11a.pdf. [Accessed: Mar. 12, 2023].