
Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais - Campus Ouro Preto
Especialização em Inteligência Artificial
Disciplina: Redes Neurais e Aprendizado Profundo
Profs.: Dr. Agnaldo José da Rocha Reis
Alunos: Fernando dos Santos Alves Fernandes, Ivanete Fátima de Azevedo e Marco Antônio do Nascimento

---

1. O que é inteligência para você(s)?

Entendemos a inteligência como uma combinação de conceitos como a sabedoria, o conhecimento, a transformação da mente e a virtude. Para nós, a sabedoria seria a capacidade de compreensão do indivíduo, dado uma informação transmitida a ele. O conhecimento pode ser visto como a consciência adquirida por meio da experiência, estudo ou introspecção, capacitando uma pessoa a interpretar e agir no mundo. A transformação da mente é a capacidade de uma pessoa de mudar seus pensamentos e perspectivas ao longo do tempo, sendo assim uma busca constante por um entendimento de evolução intelectual. Por fim temos a virtude que é a capacidade do indivíduo em buscar a excelência moral e ética, não apenas a excelência intelectual. Isso posto, a inteligência, para nós, não é apenas uma capacidade intelectual do indivíduo, mas também envolve a busca por virtude e a capacidade de transformação da mente. O conceito de inteligência também pode ser associado à capacidade humana de tomar decisões e resolver problemas; à capacidade de se adaptar a diferentes situações; à capacidade de aprender algo novo, a partir da detecção de padrões.

2. Em sua opinião (ou na do grupo), o que aconteceria se alguém descobrisse como implementar uma IA mais abrangente (e.g., AGI) em um robô?

Em algumas tarefas, as máquinas já são capazes de ter um desempenho semelhante ou melhor do que o ser humano, por exemplo, no processamento e análise de dados e imagens. De qualquer forma, os modelos de inteligência artificial atuais são capazes de resolver tarefas específicas. No momento em que for possível o desenvolvimento e implantação de uma IA mais abrangente em um robô, principalmente, se (ou quando) alcançarmos uma AGI (Artificial General Intelligence), para alguns pesquisadores, estaremos diante de um risco à raça humana; para outros, esse é o propósito das pesquisas em Inteligência Artificial. Acreditamos que a implementação de uma IA mais abrangente, como a AGI em robôs, representa uma potencial revolução para a sociedade, podendo causar avanços em diversas esferas, desde a automação industrial até aplicações médicas e pesquisa científica. Dentro da automação industrial, uma AGI poderia otimizar processos em setores variados, elevando eficiência na produção, logística e serviços. Na medicina, os benefícios seriam notáveis, com robôs utilizando IAG para diagnósticos mais precisos, execução de cirurgias complexas e até mesmo fornecimento de suporte emocional a pacientes. Contudo, diversos dilemas éticos podem surgir e demandam reflexões profundas sobre temas como responsabilidade, privacidade e acesso à tecnologia. Além disso, a regulamentação torna-se um ponto que merece atenção, dado que a sociedade deveria estabelecer diretrizes claras para o desenvolvimento e uso responsável da AGI, crucial para assegurar a segurança, transparência e conformidade ética. De todo modo, acreditamos que uma AGI deve ser vista como uma ferramenta para aprimorar capacidades humanas e não uma força substitutiva. Isso posto, precisamos interpretar esse dilema de forma cautelosa e ética, alinhada aos valores fundamentais da sociedade, para garantir que os benefícios da AGI sejam amplamente distribuídos, preservando o bem-estar humano numa era de transformação tecnológica.

3. A partir da análise de um processo de destilação fracionada de petróleo observou-se que determinado óleo poderia ser classificado em duas classes de pureza {C1 e C2}, mediante a medição de três grandezas {x1, x2 e x3} que representam algumas das propriedades físico-químicas do óleo. Para tanto, pretende-se utilizar um perceptron para executar a classificação automática dessas duas classes. Assim, baseadas nas informações coletadas do processo, formou-se o conjunto de treinamento em anexo1, tomando por convenção o valor –1 para óleo pertencente à classe C1 e o valor +1 para óleo pertencente à classe C2.

Daí, pede-se:

a. Execute dois treinamentos para a rede perceptron, inicializando-se o vetor de pesos em cada treinamento com valores aleatórios entre zero e um de tal forma que os elementos do vetor de pesos iniciais não sejam os mesmos.

In [ ]:
```python
import numpy as np

class Perceptron:
    def __init__(self, num_features, learning_rate=0.01, epochs=100):
        self.num_features = num_features
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.random.rand(num_features + 1)  # initial random weights +1 for the bias term
        self.initialWeights = self.weights
        print(f'[INFO] \tRandom initial weights: {self.weights}')

    def predict(self, inputs): # activation function
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0] # activation potential: u
        return 1 if summation >= 0 else -1 # Use of the bipolar step function

    def train(self, training_data, labels):
        hasError = True
        for epoch in range(self.epochs):
            #print(f'[INFO] Epoch: {epoch}')
            hasError = False
            for inputs, label in zip(training_data, labels):
                prediction = self.predict(inputs) # return of activation function: y
                if prediction != label:
                    hasError = True
                    update = self.learning_rate * (label - prediction) # eta * (dk - y)
                    self.weights[1:] += update * inputs # update weights: w <- w + eta * (dk - y) * xk
                    self.weights[0] += update # update activation limiar: tetha <- tetha + eta * (dk - y)
                    #print(f'[INFO] Weights: {self.weights}')
            if hasError == False:
                print(f'[INFO] \tConverged after: {epoch + 1} epochs.')
```

```
                break
        print(f'[INFO] \tFinal weights: {self.weights}')
        print(f'[INFO] \tTotal of epochs: {epoch + 1}')

    def getInitialWeights(self):
        return self.initialWeights

    def getFinalWeights(self):
        return self.weights
```

In [ ]:

```
print(f'\n[INFO] ###### Perceptron Implementation #######')
print(f'\n[INFO] Loading training dataset and labels...')
file = open('tab_treinamento1.dat', 'r')
results = list()
l = list()

for line in file:
    columns = line.split()
    columns = np.array(columns, dtype=float)
    results.append(columns[:3])
    l.append(columns[-1:])

training_data = np.array(results)
labels = np.array(l)
print(f'\t[INFO] OK!')

#training_data = np.array([[0.6508, 0.1097, 4.0009], [-1.4492, 0.8896, 4.4005], [2.085, 0.6876, 1.2071], [0.2626, 1.1476, 7.7985], [0.6418,
#labels = np.array([-1, -1, -1, 1, 1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, -1, 1, -1, 1])
print(f'\n[INFO] Getting information about training dataset...')
print(f'[INFO] Training dataset: \n{training_data}')
print(f'[INFO] Labels of training dataset: \n{labels}')
```

```
[INFO] ###### Perceptron Implementation #######

[INFO] Loading training dataset and labels...
       [INFO] OK!

[INFO] Getting information about training dataset...
[INFO] Training dataset:
[[-0.6508  0.1097  4.0009]
 [-1.4492  0.8896  4.4005]
 [ 2.085   0.6876 12.071 ]
 [ 0.2626  1.1476  7.7985]
 [ 0.6418  1.0234  7.0427]
 [ 0.2569  0.673   8.3265]
 [ 1.1155  0.6043  7.4446]
 [ 0.0914  0.3399  7.0677]
 [ 0.0121  0.5256  4.6316]
 [-0.0429  0.466   5.4323]
 [ 0.434   0.687   8.2287]
 [ 0.2735  1.0287  7.1934]
 [ 0.4839  0.4851  7.485 ]
 [ 0.4089 -0.1267  5.5019]
 [ 1.4391  0.1614  8.5843]
 [-0.9115 -0.1973  2.1962]
 [ 0.3654  1.0475  7.4858]
 [ 0.2144  0.7515  7.1699]
 [ 0.2013  1.0014  6.5489]
 [ 0.6483  0.2183  5.8991]
 [-0.1147  0.2242  7.2435]
 [-0.797   0.8795  3.8762]
 [-1.0625  0.6366  2.4707]
 [ 0.5307  0.1285  5.6883]
 [-1.22    0.7777  1.7252]
 [ 0.3957  0.1076  5.6623]
 [-0.1013  0.5989  7.1812]
 [ 2.4482  0.9455 11.2095]
 [ 2.0149  0.6192 10.9263]
 [ 0.2012  0.2611  5.4631]]
[INFO] Labels of training dataset:
[[-1.]
 [-1.]
 [-1.]
 [ 1.]
 [ 1.]
 [-1.]
 [ 1.]
 [-1.]
 [ 1.]
 [ 1.]
 [-1.]
 [ 1.]
 [-1.]
 [-1.]
 [-1.]
 [-1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [-1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [-1.]
 [-1.]
 [ 1.]
 [-1.]
 [ 1.]]
```

In [ ]:
```python
# Creating a Perceptron
print(f'\n[INFO] Creating a Perceptron...')
number_of_epochs = 10000
perceptron1 = Perceptron(num_features=3, learning_rate=0.01, epochs=number_of_epochs)
print(f'[INFO] \tOK!')
```

```
[INFO] Creating a Perceptron...
[INFO]  Random initial weights: [0.17736342 0.15638614 0.47744044 0.1209934 ]
[INFO]  OK!
```

In [ ]:
```python
# Training the perceptron 1
print(f'\n[INFO] Getting information about training dataset...')
print(f'[INFO] \tTraining dataset size = {training_data.shape[0]}')
print(f'[INFO] \tLabels size = {labels.shape[0]}')
print(f'[INFO] \tLimit of epochs: {number_of_epochs}')
print(f'\n[INFO] Training the Perceptron 1...')
perceptron1.train(training_data, labels)

initialWeights = perceptron1.getInitialWeights()
finalWeights = perceptron1.getFinalWeights()
print(f'\t[INFO] OK!')
```

```
[INFO] Getting information about training dataset...
[INFO]  Training dataset size = 30
[INFO]  Labels size = 30
[INFO]  Limit of epochs: 10000

[INFO] Training the Perceptron 1...
[INFO]  Converged after: 333 epochs.
[INFO]  Final weights: [ 2.93736342  1.41135414  2.43657844 -0.7023866 ]
[INFO]  Total of epochs: 333
        [INFO] OK!
```

In [ ]:
```python
# Creating a new Perceptron
print(f'\n[INFO] Creating a new Perceptron...')
number_of_epochs = 10000
perceptron2 = Perceptron(num_features=3, learning_rate=0.01, epochs=number_of_epochs)
print(f'[INFO] \tOK!')
```

```
[INFO] Creating a new Perceptron...
[INFO]  Random initial weights: [0.44467875 0.60329962 0.69505087 0.33930767]
[INFO]  OK!
```

In [ ]:
```python
# Training the perceptron 2
print(f'\n[INFO] Getting information about training dataset...')
print(f'[INFO] \tTraining dataset size = {training_data.shape[0]}')
print(f'[INFO] \tLabels size = {labels.shape[0]}')
print(f'[INFO] \tLimit of epochs: {number_of_epochs}')
print(f'\n[INFO] Training the Perceptron 2...')
perceptron2.train(training_data, labels)

print(f'\t[INFO] OK!')
```

```
[INFO] Getting information about training dataset...
[INFO]  Training dataset size = 30
[INFO]  Labels size = 30
[INFO]  Limit of epochs: 10000

[INFO] Training the Perceptron 2...
[INFO]  Converged after: 386 epochs.
[INFO]  Final weights: [ 3.06467875  1.55732162  2.47131887 -0.73088233]
[INFO]  Total of epochs: 386
        [INFO] OK!
```

b. Registre os resultados dos dois treinamentos na tabela a seguir:

Tabela 1 - Resultados dos treinamentos (**tab_treinamento1.dat**):

Treinamento	Vetor de Pesos Inicial				Vetor de Pesos Final				Número de Épocas
	b	w1	w2	w3	b	w1	w2	w3	
1º (T1)	0.1773634	0.15638614	0.47744044	0.1209934	2.93736342	1.41135414	2.43657844	-0.7023866	333
2º (T1)	0.44467875	0.15638614	0.69505087	0.33930767	3.06467875	1.55732162	2.47131887	-0.73088233	386

Nota: os valores atuais dessa tabela se referem ao últimos treinamentos realizados. Caso os trechos de código acima sejam executados novamente, necessita-se atualizar a tabela.

c. Após o treinamento do perceptron, aplique-o na classificação automática de novas amostras de óleo (ver arquivo tab_teste1.dat), indicando-se na tabela seguinte os resultados das saídas (Classes) referentes aos dois processos de treinamento realizados no item a.

In [ ]:
```python
# Testing dataset
# test_data = np.array([[0.6508, 0.1097, 4.0009], [-1.4492, 0.8896, 4.4005], [2.085, 0.6876, 1.2071], [0.2626, 1.1476, 7.7985], [0.6418, 1.


# Testing the perceptron
print(f'\n[INFO] Loading testing dataset...')
file = open('tab_teste1.dat', 'r')
results = list()

for line in file:
    columns = line.split()
    columns = np.array(columns, dtype=float)
    results.append(columns[:])

testing_data = np.array(results)
print(f'\t[INFO] OK!')

print(f'\n[INFO] Getting information about testing dataset...')
print(f'[INFO] Testing dataset: \n{testing_data}')
# print(f'[INFO] \tTesting dataset size = {training_data.shape[0]}')
print(f'[INFO] \tTesting dataset size = {testing_data.shape[0]}')
```

```
[INFO] Loading testing dataset...
        [INFO] OK!

[INFO] Getting information about testing dataset...
[INFO] Testing dataset:
[[-0.3565  0.062   5.9891]
 [-0.7842  1.1267  5.5912]
 [ 0.3012  0.5611  5.8234]
 [ 0.7757  1.0648  8.0677]
 [ 0.157   0.8028  6.304 ]
 [-0.7014  1.0316  3.6005]
 [ 0.3748  0.1536  6.1537]
 [-0.692   0.9404  4.4058]
 [-1.397   0.7141  4.9263]
 [-1.8842 -0.2805  1.2548]]
[INFO]  Testing dataset size = 10
```

In [ ]:
```
print(f'\n[INFO] Running testing data with Perceptron 1...')
# for inputs in training_data:
for inputs in testing_data:
    result = perceptron1.predict(inputs)
    print(f"[INFO] \tInput: {inputs} -> Output: {result}")
```

```
[INFO] Running testing data with Perceptron 1...
[INFO]  Input: [-0.3565  0.062   5.9891] -> Output: -1
[INFO]  Input: [-0.7842  1.1267  5.5912] -> Output: 1
[INFO]  Input: [0.3012 0.5611 5.8234] -> Output: 1
[INFO]  Input: [0.7757 1.0648 8.0677] -> Output: 1
[INFO]  Input: [0.157  0.8028 6.304 ] -> Output: 1
[INFO]  Input: [-0.7014  1.0316  3.6005] -> Output: 1
[INFO]  Input: [0.3748 0.1536 6.1537] -> Output: -1
[INFO]  Input: [-0.692   0.9404  4.4058] -> Output: 1
[INFO]  Input: [-1.397   0.7141  4.9263] -> Output: -1
[INFO]  Input: [-1.8842 -0.2805  1.2548] -> Output: -1
```

In [ ]:
```
print(f'\n[INFO] Running testing data with Perceptron 2...')
# for inputs in training_data:
for inputs in testing_data:
    result = perceptron2.predict(inputs)
    print(f"[INFO] \tInput: {inputs} -> Output: {result}")
```

```
[INFO] Running testing data with Perceptron 2...
[INFO]  Input: [-0.3565  0.062   5.9891] -> Output: -1
[INFO]  Input: [-0.7842  1.1267  5.5912] -> Output: 1
[INFO]  Input: [0.3012 0.5611 5.8234] -> Output: 1
[INFO]  Input: [0.7757 1.0648 8.0677] -> Output: 1
[INFO]  Input: [0.157  0.8028 6.304 ] -> Output: 1
[INFO]  Input: [-0.7014  1.0316  3.6005] -> Output: 1
[INFO]  Input: [0.3748 0.1536 6.1537] -> Output: -1
[INFO]  Input: [-0.692   0.9404  4.4058] -> Output: 1
[INFO]  Input: [-1.397   0.7141  4.9263] -> Output: -1
[INFO]  Input: [-1.8842 -0.2805  1.2548] -> Output: -1
```

Tabela 2 - Resultados com as saídas das saída (classes) para os dados de teste (**tab_teste1.dat**):

Amostra	x1	x2	x3	y (T1)	y (T2)
1	-0.3565	0.0620	5.9891	-1	-1
2	-0.7842	1.1267	5.5912	1	1
3	0.3012	0.5611	5.8234	1	1
4	0.7757	1.0648	8.0677	1	1
5	0.157	0.8028	6.304	1	1
6	-0.7014	1.0316	3.6005	1	1
7	0.3748	0.1536	6.1537	-1	-1
8	-0.692	0.9404	4.4058	1	1
9	-1.397	0.7141	4.9263	-1	-1
10	-1.8842	-0.2805	1.2548	-1	-1

Nota: os valores atuais dessa tabela se referem ao últimos testes realizados. Caso os trechos de código acima sejam executados novamente, necessita-se atualizar a tabela.

d. Explique por que o número de épocas de treinamento varia a cada vez que se executa o treinamento do perceptron.

O número de épocas necessárias para a convergência do processo de treinamento varia conforme os valores iniciais dos pesos e do limiar de ativação. Esses valores iniciais definem quão longe da fronteira de seperação o treinamento inicia o ajuste do modelo. Se os valores iniciais atribuídos aos pesos e ao limiar de ativação são definidos aleatoriamente, a cada execução do treinamento, o número de épocas para a convergência normalmente é diferente.

e. Qual é a principal limitação do perceptron quando aplicado em problemas de classificação de padrões?

Os perceptrons são um tipo de rede neurais apropriada para tarefa de classificação de padrões em que as classes são linearmente separáveis. Por isso, para problemas não linearmente separáveis utilizando perceptrons, é necessário especificar um número máximo de épocas de treinamento.

4. Um sistema de gerenciamento automático de controle de duas válvulas, situado a 500 metros de um processo industrial, envia um sinal codificado constituído de quatro grandezas {x1, x2, x3 e x4} que são necessárias para o ajuste de cada uma das válvulas. Conforme mostra a figura abaixo, a mesma via de comunicação é utilizada para acionamento de ambas as válvulas, sendo que o comutador localizado próximo das válvulas deve decidir se o sinal é para a válvula A ou B. Porém, durante a transmissão, os sinais sofrem interferências que alteram o conteúdo das informações transmitidas. Para resolver este problema, treinar-se-á uma rede ADALINE para classificar os sinais ruidosos, que informará ao sistema comutador se os dados devem ser encaminhados para o comando de ajuste da válvula A ou B. Assim, baseado nas medições dos sinais já com ruídos, formou-se o conjunto de treinamento em anexo2, tomando por convenção o valor –1 para os sinais que devem ser encaminhados para o ajuste da válvula A e o valor +1 se os mesmos devem ser enviados para a válvula B.

Daí, pede-se:

a. Execute 2 treinamentos para a rede ADALINE inicializando o vetor de pesos em cada treinamento com valores aleatórios entre zero e um de tal forma que os elementos do vetor de pesos iniciais não sejam os mesmos.

In [ ]:
```python
import numpy as np

class Adaline:
    def __init__(self, num_features, learning_rate=0.01, epochs=100, epsilon=1e-5):
        self.num_features = num_features
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.epsilon = epsilon
        self.weights = np.random.rand(num_features + 1)  # initial random weights +1 for the bias term
        print(f'[INFO] \tRandom initial weights: {self.weights}')

    def predict(self, inputs): # activation function
        activation = np.dot(inputs, self.weights[1:]) + self.weights[0] # activation potential: u
        return 1 if activation >= 0 else -1  # Use of the bipolar step function

    def train(self, training_data, targets):
        mse = 0
        for epoch in range(self.epochs):
            total_error = 0
            #print(f'[INFO] Epoch: {epoch}')
            for inputs, target in zip(training_data, targets):
                activation = self.predict(inputs) # return of activation function: y
                error = target - activation
                self.weights[1:] += self.learning_rate * error * inputs
                self.weights[0] += self.learning_rate * error
                total_error += error ** 2
            mse = total_error / len(targets)
            if mse < self.epsilon:
                print(f'[INFO] \tConverged after: {epoch + 1} epochs.')
                break
        print(f'[INFO] \tFinal weights: {self.weights}')
        print(f'[INFO] \tTotal of epochs: {epoch + 1}')
        print(f'[INFO] \tMean square error: {mse}')
```

In [ ]:
```python
print(f'\n[INFO] ###### Adaline Implementation #######')
print(f'\n[INFO] Loading training dataset and targets...')
file = open('tab_treinamento2.dat', 'r')
results = list()
t = list()

for line in file:
    columns = line.split()
    columns = np.array(columns, dtype=float)
    results.append(columns[:4])
    t.append(columns[-1:])

training_data = np.array(results)
targets = np.array(t)
print(f'[INFO] \tOK!')

#training_data = np.array([[4.3290000e-01, -1.3719000e+00, 7.0220000e-01, -8.5350000e-01],[3.0240000e-01, 2.2860000e-01, 8.6300000e-01, 2.7
# targets = np.array([1.0000000e+00, -1.0000000e+00, -1.0000000e+00, -1.0000000e+00, 1.0000000e+00, 1.0000000e+00, 1.0000000e+00, 1.0000000
print(f'[INFO] \tTraining dataset: \n{training_data}')
print(f'[INFO] \tTargets of training dataset: \n{targets}')
```

```
[INFO] ###### Adaline Implementation #######

[INFO] Loading training dataset and targets...
[INFO]  OK!
[INFO]  Training dataset:
[[ 4.3290e-01 -1.3719e+00  7.0220e-01 -8.5350e-01]
 [ 3.0240e-01  2.2860e-01  8.6300e-01  2.7909e+00]
 [ 1.3490e-01 -6.4450e-01  1.0530e+00  5.6870e-01]
 [ 3.3740e-01 -1.7163e+00  3.6700e-01 -6.2830e-01]
 [ 1.1434e+00 -4.8500e-02  6.6370e-01  1.2606e+00]
 [ 1.3749e+00 -5.0710e-01  4.4640e-01  1.3009e+00]
 [ 7.2210e-01 -7.5870e-01  7.6810e-01 -5.5920e-01]
 [ 4.4030e-01 -8.0720e-01  5.1540e-01 -3.1290e-01]
 [-5.2310e-01  3.5480e-01  2.5380e-01  1.5776e+00]
 [ 3.2550e-01 -2.0000e+00  7.1120e-01 -1.1209e+00]
 [ 5.8240e-01  1.3915e+00 -2.2910e-01  4.1735e+00]
 [ 1.3400e-01  6.0810e-01  4.4500e-01  3.2230e+00]
 [ 1.4800e-01 -2.9880e-01  4.7780e-01  8.6490e-01]
 [ 7.3590e-01  1.8690e-01 -8.7200e-02  2.3584e+00]
 [ 7.1150e-01 -1.1469e+00  3.3940e-01  9.5730e-01]
 [ 8.2510e-01 -1.2840e+00  8.4520e-01  1.2382e+00]
 [ 1.5690e-01  3.7120e-01  8.8250e-01  1.7633e+00]
 [ 3.3000e-03  6.8350e-01  5.3890e-01  2.8249e+00]
 [ 4.2430e-01  8.3130e-01  2.6340e-01  3.5855e+00]
 [ 1.0490e+00  1.3260e-01  9.1380e-01  1.9792e+00]
 [ 1.4276e+00  5.3310e-01 -1.4500e-02  3.7286e+00]
 [ 5.9710e-01  1.4865e+00  2.9040e-01  4.6069e+00]
 [ 8.4750e-01  2.1479e+00  3.1790e-01  5.8235e+00]
 [ 1.3967e+00 -4.1710e-01  6.4430e-01  1.3927e+00]
 [ 4.4000e-03  1.5378e+00  6.0990e-01  4.7755e+00]
 [ 2.2010e-01 -5.6680e-01  5.1500e-02  7.8290e-01]
 [ 6.3000e-01 -1.2480e+00  8.5910e-01  8.0930e-01]
 [-2.4790e-01  8.9600e-01  5.4700e-02  1.7381e+00]
 [-3.0880e-01 -9.2900e-02  8.6590e-01  1.5483e+00]
 [-5.1800e-01  1.4974e+00  5.4530e-01  2.3993e+00]
 [ 6.8330e-01  8.2660e-01  8.2900e-02  2.8864e+00]
 [ 4.3530e-01 -1.4066e+00  4.2070e-01 -4.8790e-01]
 [-1.0690e-01 -3.2329e+00  1.8560e-01 -2.4572e+00]
 [ 4.6620e-01  6.2610e-01  7.3040e-01  3.4370e+00]
 [ 8.2980e-01 -1.4089e+00  3.1190e-01  1.3235e+00]]
[INFO]  Targets of training dataset:
[[ 1.]
 [-1.]
 [-1.]
 [-1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [-1.]
 [ 1.]
 [-1.]
 [-1.]
 [ 1.]
 [ 1.]
 [-1.]
 [-1.]
 [ 1.]
 [-1.]
 [-1.]
 [ 1.]
 [ 1.]
 [-1.]
 [ 1.]
 [-1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [-1.]
 [-1.]
 [-1.]]
```

In [ ]:
```
# Creating an Adaline
print(f'\n[INFO] Creating an Adaline...')
number_of_epochs = 10000
epsilon = 1e-5
adaline1 = Adaline(num_features=4, learning_rate=0.01, epochs=number_of_epochs, epsilon=epsilon)
print(f'[INFO] \tOK!')
```

```
[INFO] Creating an Adaline...
[INFO]  Random initial weights: [0.85687093 0.02633478 0.96817149 0.24457783 0.34848885]
[INFO]  OK!
```

In [ ]:
```
# Training the Adaline
print(f'\n[INFO] Getting information about training dataset...')
print(f'[INFO] \tTraining dataset size = {training_data.shape[0]} {training_data.shape[1]}')
print(f'[INFO] \tLabels size = {targets.shape[0]} {targets.shape[1]}')
print(f'[INFO] \tLimit of epochs: {number_of_epochs}')
```

```
print(f'[INFO] \tEpsilon: {epsilon}')
print(f'\n[INFO] Training the Adaline 1...')
adaline1.train(training_data, targets)
print(f'[INFO] \tOK!')
```

```
[INFO] Getting information about training dataset...
[INFO]  Training dataset size = 35 4
[INFO]  Labels size = 35 1
[INFO]  Limit of epochs: 10000
[INFO]  Epsilon: 1e-05

[INFO] Training the Adaline 1...
[INFO]  Converged after: 39 epochs.
[INFO]  Final weights: [ 0.61687093  0.54973078  0.61733949 -0.08346417 -0.44539315]
[INFO]  Total of epochs: 39
[INFO]  Mean square error: [0.]
[INFO]  OK!
```

In [ ]:
```
# Creating an Adaline
print(f'\n[INFO] Creating an new Adaline...')
number_of_epochs = 10000
epsilon = 1e-5
adaline2 = Adaline(num_features=4, learning_rate=0.01, epochs=number_of_epochs, epsilon=epsilon)
print(f'[INFO] \tOK!')
```

```
[INFO] Creating an new Adaline...
[INFO]  Random initial weights: [0.73993445 0.71783308 0.88046206 0.71642348 0.55246439]
[INFO]  OK!
```

In [ ]:
```
# Training the Adaline
print(f'\n[INFO] Getting information about training dataset...')
print(f'[INFO] \tTraining dataset size = {training_data.shape[0]} {training_data.shape[1]}')
print(f'[INFO] \tLabels size = {targets.shape[0]} {targets.shape[1]}')
print(f'[INFO] \tLimit of epochs: {number_of_epochs}')
print(f'[INFO] \tEpsilon: {epsilon}')
print(f'\n[INFO] Training the Adaline 2...')
adaline2.train(training_data, targets)
print(f'[INFO] \tOK!')
```

```
[INFO] Getting information about training dataset...
[INFO]  Training dataset size = 35 4
[INFO]  Labels size = 35 1
[INFO]  Limit of epochs: 10000
[INFO]  Epsilon: 1e-05

[INFO] Training the Adaline 2...
[INFO]  Converged after: 62 epochs.
[INFO]  Final weights: [ 0.63993445  0.64771308  0.67042606 -0.07245452 -0.49953761]
[INFO]  Total of epochs: 62
[INFO]  Mean square error: [0.]
[INFO]  OK!
```

b. Registre os resultados dos dois treinamentos na tabela a seguir:

Tabela 3 - Resultados dos treinamentos (**tab_treinamento2.dat**):

Treinamento	Vetor de Pesos Inicial					Vetor de Pesos Final					Número de Épocas
	b	w1	w2	w3	w4	b	w1	w2	w3	w4	
1º (T1)	0.85687093	0.02633478	0.96817149	0.24457783	0.34848885	0.61687093	0.54973078	0.61733949	-0.08346417	-0.44539315	39
2º (T1)	0.73993445	0.71783308	0.88046206	0.71642348	0.55246439	0.63993445	0.64771308	0.67042606	-0.07245452	-0.49953761	62

Nota: os valores atuais dessa tabela se referem ao últimos treinamentos realizados. Caso os trechos de código acima sejam executados novamente, necessita-se atualizar a tabela.

c. Para os treinamentos realizados, aplique então a rede ADALINE para classificar e informar ao comutador se os sinais seguintes devem ser encaminhados para a válvula A ou B (ver tab_teste2.dat).

In [ ]:
```
# # Testing the adaline
print(f'\n[INFO] Loading testing dataset...')
file = open('tab_teste2.dat', 'r')
results = list()

for line in file:
    columns = line.split()
    columns = np.array(columns, dtype=float)
    results.append(columns[:])

testing_data = np.array(results)
print(f'\t[INFO] OK!')

print(f'\n[INFO] Getting information about testing dataset...')
# print(f'[INFO] Testing dataset size = {training_data.shape[0]}')
print(f'[INFO] \tTesting dataset size = {testing_data.shape[0]}')
```

```
[INFO] Loading testing dataset...
        [INFO] OK!

[INFO] Getting information about testing dataset...
[INFO]  Testing dataset size = 15
```

In [ ]:
```
print(f'\n[INFO] Running testing data with Adaline 1...')
# for inputs in training_data:
for inputs in testing_data:
    result = adaline1.predict(inputs)
    print(f"[INFO] \tInput: {inputs} -> Output: {result}")
```

```
[INFO] Running testing data with Adaline 1...
[INFO]  Input: [0.9694 0.6909 0.4334 3.4965] -> Output: -1
[INFO]  Input: [0.5427 1.3832 0.639  4.0352] -> Output: -1
[INFO]  Input: [ 0.6081 -0.9196  0.5925  0.1016] -> Output: 1
[INFO]  Input: [-0.1618  0.4694  0.203   3.0117] -> Output: -1
[INFO]  Input: [ 0.187  -0.2578  0.6124  1.7749] -> Output: -1
[INFO]  Input: [ 0.4891 -0.5276  0.4378  0.6439] -> Output: 1
[INFO]  Input: [0.3777 2.0149 0.7423 3.3932] -> Output: 1
[INFO]  Input: [ 1.1498 -0.4067  0.2469  1.5866] -> Output: 1
[INFO]  Input: [0.9325 1.095  1.0359 3.3591] -> Output: 1
[INFO]  Input: [0.506  1.3317 0.9222 3.7174] -> Output: -1
[INFO]  Input: [ 0.0497 -2.0656  0.6124 -0.6585] -> Output: -1
[INFO]  Input: [0.4004 3.5369 0.9766 5.3532] -> Output: 1
[INFO]  Input: [-0.1874  1.3343  0.5374  3.2189] -> Output: -1
[INFO]  Input: [0.506  1.3317 0.9222 3.7174] -> Output: -1
[INFO]  Input: [ 1.6375 -0.7911  0.7537  0.5515] -> Output: 1
```

In [ ]:
```
print(f'\n[INFO] Running testing data with Adaline 2...')
# for inputs in training_data:
for inputs in testing_data:
    result = adaline2.predict(inputs)
    print(f"[INFO] \tInput: {inputs} -> Output: {result}")
```

```
[INFO] Running testing data with Adaline 2...
[INFO]  Input: [0.9694 0.6909 0.4334 3.4965] -> Output: -1
[INFO]  Input: [0.5427 1.3832 0.639  4.0352] -> Output: -1
[INFO]  Input: [ 0.6081 -0.9196  0.5925  0.1016] -> Output: 1
[INFO]  Input: [-0.1618  0.4694  0.203   3.0117] -> Output: -1
[INFO]  Input: [ 0.187  -0.2578  0.6124  1.7749] -> Output: -1
[INFO]  Input: [ 0.4891 -0.5276  0.4378  0.6439] -> Output: 1
[INFO]  Input: [0.3777 2.0149 0.7423 3.3932] -> Output: 1
[INFO]  Input: [ 1.1498 -0.4067  0.2469  1.5866] -> Output: 1
[INFO]  Input: [0.9325 1.095  1.0359 3.3591] -> Output: 1
[INFO]  Input: [0.506  1.3317 0.9222 3.7174] -> Output: -1
[INFO]  Input: [ 0.0497 -2.0656  0.6124 -0.6585] -> Output: -1
[INFO]  Input: [0.4004 3.5369 0.9766 5.3532] -> Output: 1
[INFO]  Input: [-0.1874  1.3343  0.5374  3.2189] -> Output: -1
[INFO]  Input: [0.506  1.3317 0.9222 3.7174] -> Output: -1
[INFO]  Input: [ 1.6375 -0.7911  0.7537  0.5515] -> Output: 1
```

Tabela 4 - Resultados com as saídas das saída (classes) para os dados de teste (**tab_teste2.dat**):

Amostra	x1	x2	x3	x4	y (T1)	y (T2)
1	0.9694	0.6909	0.4334	3.4965	-1	-1
2	0.5427	1.3832	0.639	4.0352	-1	-1
3	0.6081	-0.9196	0.5925	0.1016	1	1
4	-0.1618	0.4694	0.203	3.0117	-1	-1
5	0.187	-0.2578	0.6124	1.7749	-1	-1
6	0.4891	-0.5276	0.4378	0.6439	1	1
7	0.3777	2.0149	0.7423	3.3932	1	1
8	1.1498	-0.4067	0.2469	1.5866	1	1
9	0.9325	1.095	1.0359	3.3591	1	1
10	0.506	1.3317	0.9222	3.7174	-1	-1
11	0.0497	-2.0656	0.6124	-0.6585	-1	-1
12	0.4004	3.5369	0.9766	5.3532	1	1
13	-0.1874	1.3343	0.5374	3.2189	-1	-1
14	0.506	1.3317	0.9222	3.7174	-1	-1
15	1.6375	-0.7911	0.7537	0.5515	1	1

Nota: os valores atuais dessa tabela se referem ao últimos testes realizados. Caso os trechos de código acima sejam executados novamente, necessita-se atualizar a tabela.

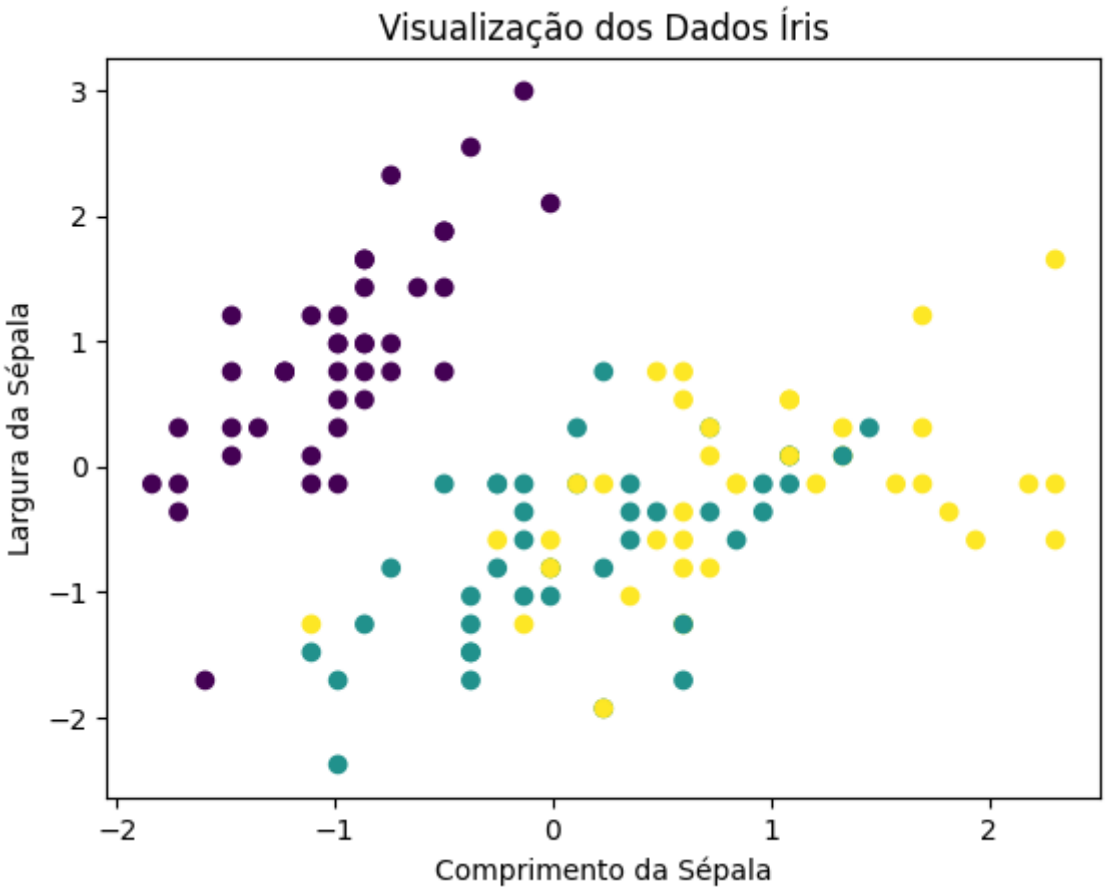
5. Um(a) estudante da disciplina de Redes Neurais e Aprendizado Profundo ficou empolgado(a) com o trabalho do Fisher sobre as flores Íris e resolveu propor uma versão automatizada para ele. Essa nova versão deveria ter dois módulos principais: um módulo de visão computacional e um módulo do tipo classificador neural. Caso você(s) fosse(m) esse(a) estudante, como você(s) desenvolveria(m) esse sistema? Descreva-o em detalhes. Use ilustração(ões) para valorizar o seu pré-projeto. Lembre-se que são três tipos de Íris (Virginica, Versicolor e Setosa) e que 4 parâmetros foram medidos pelo Fisher para cada uma das flores (comprimento e largura da Pétala, Comprimento e largura da Sépala).

Exemplo de código Python para implementar o sistema automatizado de classificação de flores Íris usando visão computacional e um classificador neural. Neste exemplo, usaremos a biblioteca TensorFlow para criar a rede neural. Certifique-se de ter o TensorFlow instalado em seu ambiente antes de executar o código. Você também precisará de outras bibliotecas como NumPy e scikit-learn. Você pode instalá-los usando o pip, se ainda não estiverem instalados:

In [ ]:
```
!pip install numpy scikit-learn tensorflow
```

In [ ]:
```
import numpy as np
import tensorflow as tf
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
import matplotlib.pyplot as plt

# Passo 1: Carregar e preparar o conjunto de dados
iris = datasets.load_iris()
X = iris.data  # Parâmetros de entrada
y = iris.target  # Rótulos das classes

# Dividir o conjunto de dados em treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Padronizar os dados (importante para redes neurais)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Visualização dos dados - Gráfico de dispersão 2D
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.xlabel('Comprimento da Sépala')
plt.ylabel('Largura da Sépala')
plt.title('Visualização dos Dados Íris')
plt.show()

# Passo 2: Criar o modelo de rede neural
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(4,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])

# Compilar o modelo
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Passo 3: Treinar o modelo
history = model.fit(X_train, y_train, epochs=100, batch_size=16, verbose=2)

# Visualizar a curva de aprendizado
plt.plot(history.history['accuracy'])
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.title('Curva de Aprendizado')
plt.show()

# Avaliar o modelo no conjunto de teste
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=2)
print(f'Acurácia no conjunto de teste: {test_accuracy * 100:.2f}%')

# Realizar uma previsão
sample = np.array([[5.1, 3.5, 1.4, 0.2]])  # Exemplo de parâmetros de uma flor Íris
sample = scaler.transform(sample)  # Padronizar os dados
predicted_class = np.argmax(model.predict(sample))
classes = ['Setosa', 'Versicolor', 'Virginica']
print(f'A flor pertence à classe: {classes[predicted_class]}')

# Matriz de Confusão
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
confusion_mtx = confusion_matrix(y_test, y_pred_classes)

# Visualizar a matriz de confusão
plt.figure(figsize=(8, 6))
plt.imshow(confusion_mtx, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Matriz de Confusão')
plt.colorbar()
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation=45)
plt.yticks(tick_marks, classes)
plt.xlabel('Predito')
plt.ylabel('Real')
plt.show()

# Relatório de Classificação
print(classification_report(y_test, y_pred_classes, target_names=classes))

# Curvas ROC
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(classes)):
```

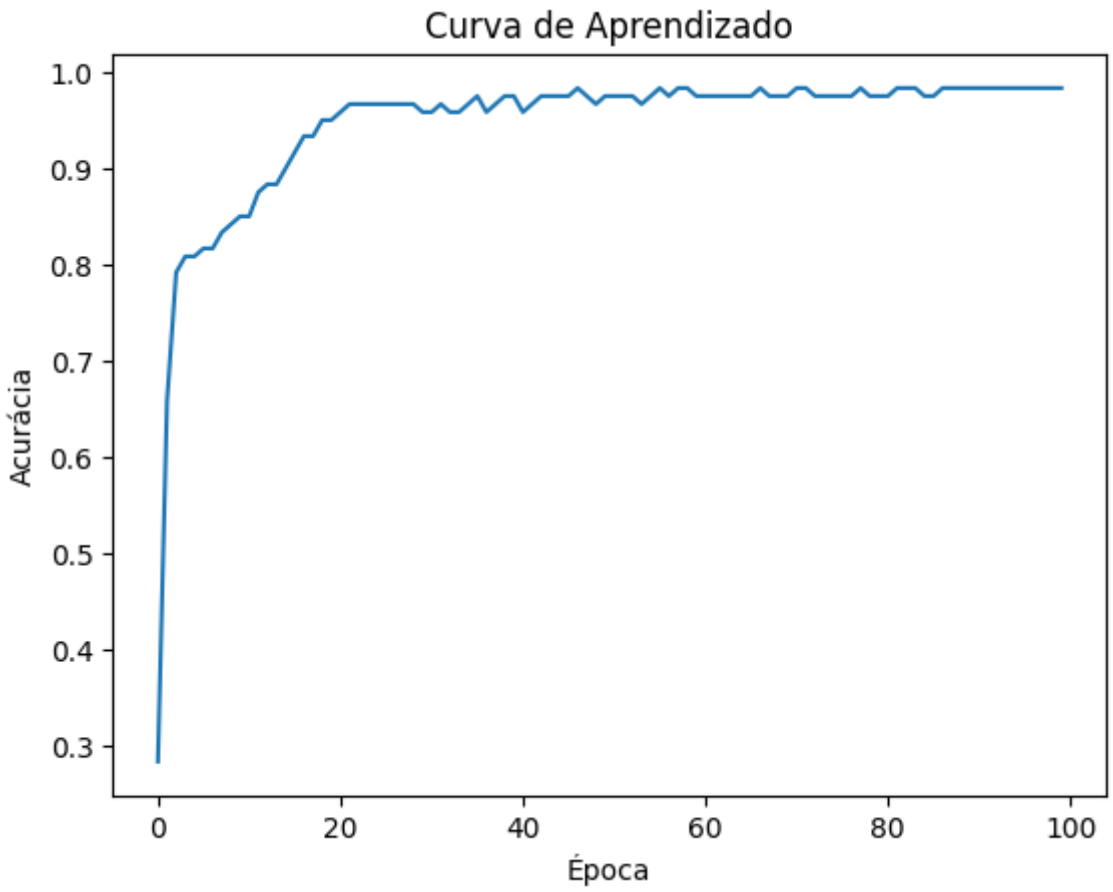
```
    fpr[i], tpr[i], _ = roc_curve(y_test, y_pred[:, i], pos_label=i)
    roc_auc[i] = auc(fpr[i], tpr[i])

# Visualizar as curvas ROC
plt.figure(figsize=(8, 6))
for i in range(len(classes)):
    plt.plot(fpr[i], tpr[i], lw=2, label=f'Classe {i} (AUC = {roc_auc[i]:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Taxa de Falso Positivo')
plt.ylabel('Taxa de Verdadeiro Positivo')
plt.title('Curvas ROC para as Classes')
plt.legend(loc='lower right')
plt.show()
```



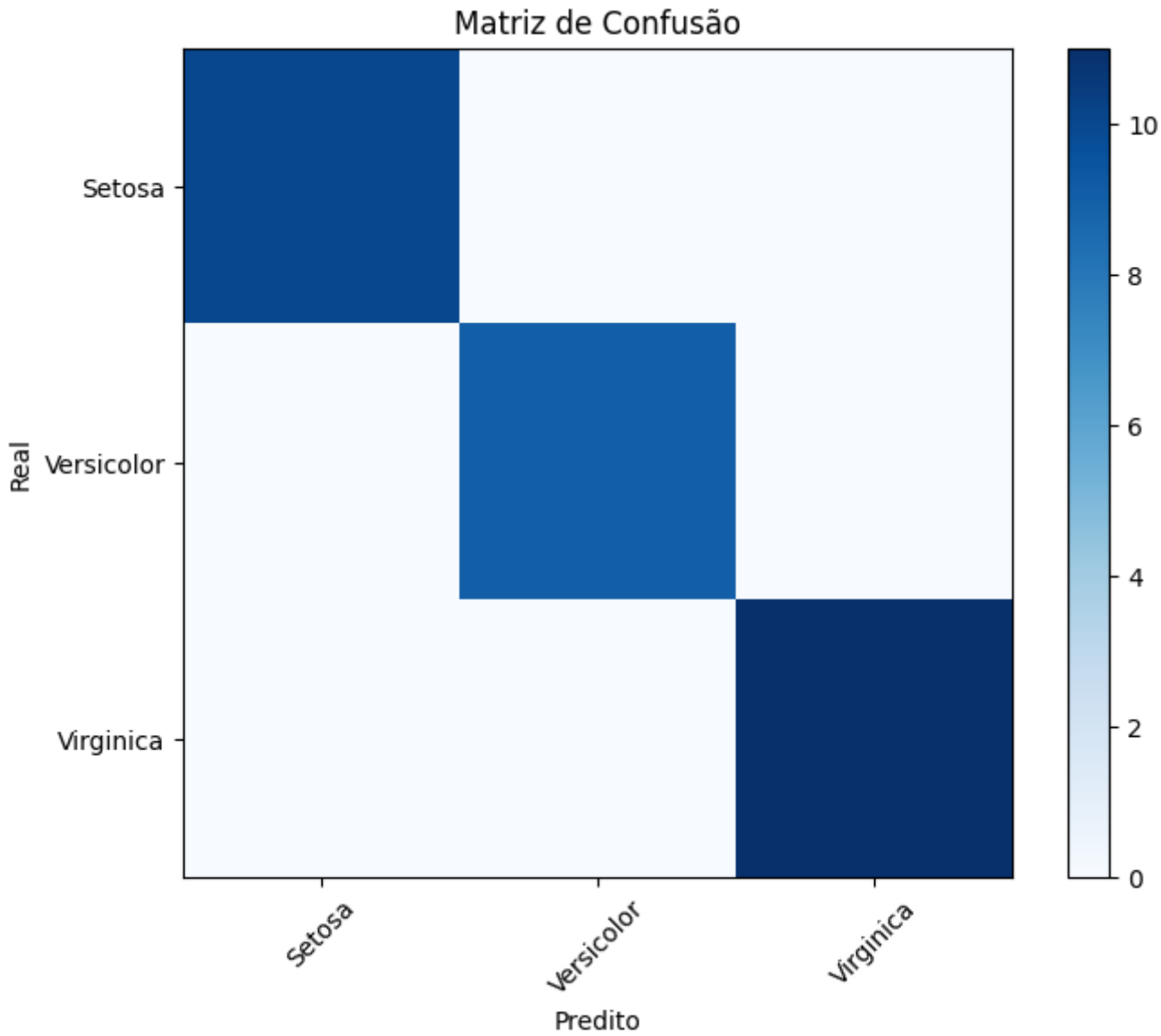
```
Epoch 1/100
8/8 - 2s - loss: 1.0850 - accuracy: 0.2833 - 2s/epoch - 196ms/step
Epoch 2/100
8/8 - 0s - loss: 0.9450 - accuracy: 0.6583 - 15ms/epoch - 2ms/step
Epoch 3/100
8/8 - 0s - loss: 0.8243 - accuracy: 0.7917 - 16ms/epoch - 2ms/step
Epoch 4/100
8/8 - 0s - loss: 0.7219 - accuracy: 0.8083 - 13ms/epoch - 2ms/step
Epoch 5/100
8/8 - 0s - loss: 0.6329 - accuracy: 0.8083 - 14ms/epoch - 2ms/step
Epoch 6/100
8/8 - 0s - loss: 0.5588 - accuracy: 0.8167 - 12ms/epoch - 2ms/step
Epoch 7/100
8/8 - 0s - loss: 0.4987 - accuracy: 0.8167 - 22ms/epoch - 3ms/step
Epoch 8/100
8/8 - 0s - loss: 0.4504 - accuracy: 0.8333 - 21ms/epoch - 3ms/step
Epoch 9/100
8/8 - 0s - loss: 0.4110 - accuracy: 0.8417 - 15ms/epoch - 2ms/step
Epoch 10/100
8/8 - 0s - loss: 0.3779 - accuracy: 0.8500 - 16ms/epoch - 2ms/step
Epoch 11/100
8/8 - 0s - loss: 0.3523 - accuracy: 0.8500 - 18ms/epoch - 2ms/step
Epoch 12/100
8/8 - 0s - loss: 0.3287 - accuracy: 0.8750 - 18ms/epoch - 2ms/step
Epoch 13/100
8/8 - 0s - loss: 0.3086 - accuracy: 0.8833 - 17ms/epoch - 2ms/step
Epoch 14/100
8/8 - 0s - loss: 0.2901 - accuracy: 0.8833 - 19ms/epoch - 2ms/step
Epoch 15/100
8/8 - 0s - loss: 0.2740 - accuracy: 0.9000 - 14ms/epoch - 2ms/step
Epoch 16/100
8/8 - 0s - loss: 0.2564 - accuracy: 0.9167 - 17ms/epoch - 2ms/step
Epoch 17/100
8/8 - 0s - loss: 0.2449 - accuracy: 0.9333 - 24ms/epoch - 3ms/step
Epoch 18/100
8/8 - 0s - loss: 0.2311 - accuracy: 0.9333 - 17ms/epoch - 2ms/step
Epoch 19/100
8/8 - 0s - loss: 0.2190 - accuracy: 0.9500 - 14ms/epoch - 2ms/step
Epoch 20/100
8/8 - 0s - loss: 0.2087 - accuracy: 0.9500 - 20ms/epoch - 3ms/step
Epoch 21/100
8/8 - 0s - loss: 0.1969 - accuracy: 0.9583 - 19ms/epoch - 2ms/step
Epoch 22/100
8/8 - 0s - loss: 0.1879 - accuracy: 0.9667 - 16ms/epoch - 2ms/step
Epoch 23/100
8/8 - 0s - loss: 0.1767 - accuracy: 0.9667 - 14ms/epoch - 2ms/step
Epoch 24/100
8/8 - 0s - loss: 0.1678 - accuracy: 0.9667 - 20ms/epoch - 3ms/step
Epoch 25/100
8/8 - 0s - loss: 0.1593 - accuracy: 0.9667 - 36ms/epoch - 5ms/step
Epoch 26/100
8/8 - 0s - loss: 0.1515 - accuracy: 0.9667 - 22ms/epoch - 3ms/step
Epoch 27/100
8/8 - 0s - loss: 0.1442 - accuracy: 0.9667 - 17ms/epoch - 2ms/step
Epoch 28/100
8/8 - 0s - loss: 0.1371 - accuracy: 0.9667 - 16ms/epoch - 2ms/step
Epoch 29/100
8/8 - 0s - loss: 0.1314 - accuracy: 0.9667 - 27ms/epoch - 3ms/step
Epoch 30/100
8/8 - 0s - loss: 0.1289 - accuracy: 0.9583 - 17ms/epoch - 2ms/step
Epoch 31/100
8/8 - 0s - loss: 0.1201 - accuracy: 0.9583 - 15ms/epoch - 2ms/step
Epoch 32/100
8/8 - 0s - loss: 0.1153 - accuracy: 0.9667 - 39ms/epoch - 5ms/step
Epoch 33/100
8/8 - 0s - loss: 0.1114 - accuracy: 0.9583 - 14ms/epoch - 2ms/step
Epoch 34/100
8/8 - 0s - loss: 0.1064 - accuracy: 0.9583 - 14ms/epoch - 2ms/step
Epoch 35/100
8/8 - 0s - loss: 0.1031 - accuracy: 0.9667 - 16ms/epoch - 2ms/step
Epoch 36/100
8/8 - 0s - loss: 0.0991 - accuracy: 0.9750 - 21ms/epoch - 3ms/step
Epoch 37/100
8/8 - 0s - loss: 0.0964 - accuracy: 0.9583 - 17ms/epoch - 2ms/step
Epoch 38/100
8/8 - 0s - loss: 0.0934 - accuracy: 0.9667 - 16ms/epoch - 2ms/step
Epoch 39/100
8/8 - 0s - loss: 0.0910 - accuracy: 0.9750 - 24ms/epoch - 3ms/step
Epoch 40/100
8/8 - 0s - loss: 0.0885 - accuracy: 0.9750 - 23ms/epoch - 3ms/step
Epoch 41/100
8/8 - 0s - loss: 0.0868 - accuracy: 0.9583 - 19ms/epoch - 2ms/step
Epoch 42/100
8/8 - 0s - loss: 0.0852 - accuracy: 0.9667 - 18ms/epoch - 2ms/step
Epoch 43/100
8/8 - 0s - loss: 0.0814 - accuracy: 0.9750 - 20ms/epoch - 3ms/step
Epoch 44/100
8/8 - 0s - loss: 0.0792 - accuracy: 0.9750 - 23ms/epoch - 3ms/step
Epoch 45/100
8/8 - 0s - loss: 0.0776 - accuracy: 0.9750 - 14ms/epoch - 2ms/step
Epoch 46/100
8/8 - 0s - loss: 0.0759 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 47/100
```

```
8/8 - 0s - loss: 0.0736 - accuracy: 0.9833 - 15ms/epoch - 2ms/step
Epoch 48/100
8/8 - 0s - loss: 0.0744 - accuracy: 0.9750 - 19ms/epoch - 2ms/step
Epoch 49/100
8/8 - 0s - loss: 0.0751 - accuracy: 0.9667 - 19ms/epoch - 2ms/step
Epoch 50/100
8/8 - 0s - loss: 0.0700 - accuracy: 0.9750 - 18ms/epoch - 2ms/step
Epoch 51/100
8/8 - 0s - loss: 0.0690 - accuracy: 0.9750 - 15ms/epoch - 2ms/step
Epoch 52/100
8/8 - 0s - loss: 0.0677 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 53/100
8/8 - 0s - loss: 0.0691 - accuracy: 0.9750 - 18ms/epoch - 2ms/step
Epoch 54/100
8/8 - 0s - loss: 0.0679 - accuracy: 0.9667 - 16ms/epoch - 2ms/step
Epoch 55/100
8/8 - 0s - loss: 0.0648 - accuracy: 0.9750 - 23ms/epoch - 3ms/step
Epoch 56/100
8/8 - 0s - loss: 0.0651 - accuracy: 0.9833 - 20ms/epoch - 3ms/step
Epoch 57/100
8/8 - 0s - loss: 0.0624 - accuracy: 0.9750 - 19ms/epoch - 2ms/step
Epoch 58/100
8/8 - 0s - loss: 0.0617 - accuracy: 0.9833 - 17ms/epoch - 2ms/step
Epoch 59/100
8/8 - 0s - loss: 0.0622 - accuracy: 0.9833 - 19ms/epoch - 2ms/step
Epoch 60/100
8/8 - 0s - loss: 0.0605 - accuracy: 0.9750 - 14ms/epoch - 2ms/step
Epoch 61/100
8/8 - 0s - loss: 0.0611 - accuracy: 0.9750 - 18ms/epoch - 2ms/step
Epoch 62/100
8/8 - 0s - loss: 0.0586 - accuracy: 0.9750 - 14ms/epoch - 2ms/step
Epoch 63/100
8/8 - 0s - loss: 0.0585 - accuracy: 0.9750 - 20ms/epoch - 3ms/step
Epoch 64/100
8/8 - 0s - loss: 0.0586 - accuracy: 0.9750 - 19ms/epoch - 2ms/step
Epoch 65/100
8/8 - 0s - loss: 0.0576 - accuracy: 0.9750 - 22ms/epoch - 3ms/step
Epoch 66/100
8/8 - 0s - loss: 0.0588 - accuracy: 0.9750 - 16ms/epoch - 2ms/step
Epoch 67/100
8/8 - 0s - loss: 0.0562 - accuracy: 0.9833 - 19ms/epoch - 2ms/step
Epoch 68/100
8/8 - 0s - loss: 0.0557 - accuracy: 0.9750 - 12ms/epoch - 2ms/step
Epoch 69/100
8/8 - 0s - loss: 0.0553 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 70/100
8/8 - 0s - loss: 0.0554 - accuracy: 0.9750 - 15ms/epoch - 2ms/step
Epoch 71/100
8/8 - 0s - loss: 0.0546 - accuracy: 0.9833 - 17ms/epoch - 2ms/step
Epoch 72/100
8/8 - 0s - loss: 0.0537 - accuracy: 0.9833 - 17ms/epoch - 2ms/step
Epoch 73/100
8/8 - 0s - loss: 0.0557 - accuracy: 0.9750 - 14ms/epoch - 2ms/step
Epoch 74/100
8/8 - 0s - loss: 0.0531 - accuracy: 0.9750 - 24ms/epoch - 3ms/step
Epoch 75/100
8/8 - 0s - loss: 0.0539 - accuracy: 0.9750 - 19ms/epoch - 2ms/step
Epoch 76/100
8/8 - 0s - loss: 0.0527 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 77/100
8/8 - 0s - loss: 0.0530 - accuracy: 0.9750 - 15ms/epoch - 2ms/step
Epoch 78/100
8/8 - 0s - loss: 0.0520 - accuracy: 0.9833 - 16ms/epoch - 2ms/step
Epoch 79/100
8/8 - 0s - loss: 0.0516 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 80/100
8/8 - 0s - loss: 0.0518 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 81/100
8/8 - 0s - loss: 0.0513 - accuracy: 0.9750 - 20ms/epoch - 3ms/step
Epoch 82/100
8/8 - 0s - loss: 0.0499 - accuracy: 0.9833 - 23ms/epoch - 3ms/step
Epoch 83/100
8/8 - 0s - loss: 0.0551 - accuracy: 0.9833 - 12ms/epoch - 2ms/step
Epoch 84/100
8/8 - 0s - loss: 0.0499 - accuracy: 0.9833 - 20ms/epoch - 3ms/step
Epoch 85/100
8/8 - 0s - loss: 0.0513 - accuracy: 0.9750 - 21ms/epoch - 3ms/step
Epoch 86/100
8/8 - 0s - loss: 0.0489 - accuracy: 0.9750 - 17ms/epoch - 2ms/step
Epoch 87/100
8/8 - 0s - loss: 0.0480 - accuracy: 0.9833 - 15ms/epoch - 2ms/step
Epoch 88/100
8/8 - 0s - loss: 0.0510 - accuracy: 0.9833 - 16ms/epoch - 2ms/step
Epoch 89/100
8/8 - 0s - loss: 0.0520 - accuracy: 0.9833 - 17ms/epoch - 2ms/step
Epoch 90/100
8/8 - 0s - loss: 0.0496 - accuracy: 0.9833 - 17ms/epoch - 2ms/step
Epoch 91/100
8/8 - 0s - loss: 0.0471 - accuracy: 0.9833 - 14ms/epoch - 2ms/step
Epoch 92/100
8/8 - 0s - loss: 0.0469 - accuracy: 0.9833 - 22ms/epoch - 3ms/step
Epoch 93/100
8/8 - 0s - loss: 0.0476 - accuracy: 0.9833 - 14ms/epoch - 2ms/step
```

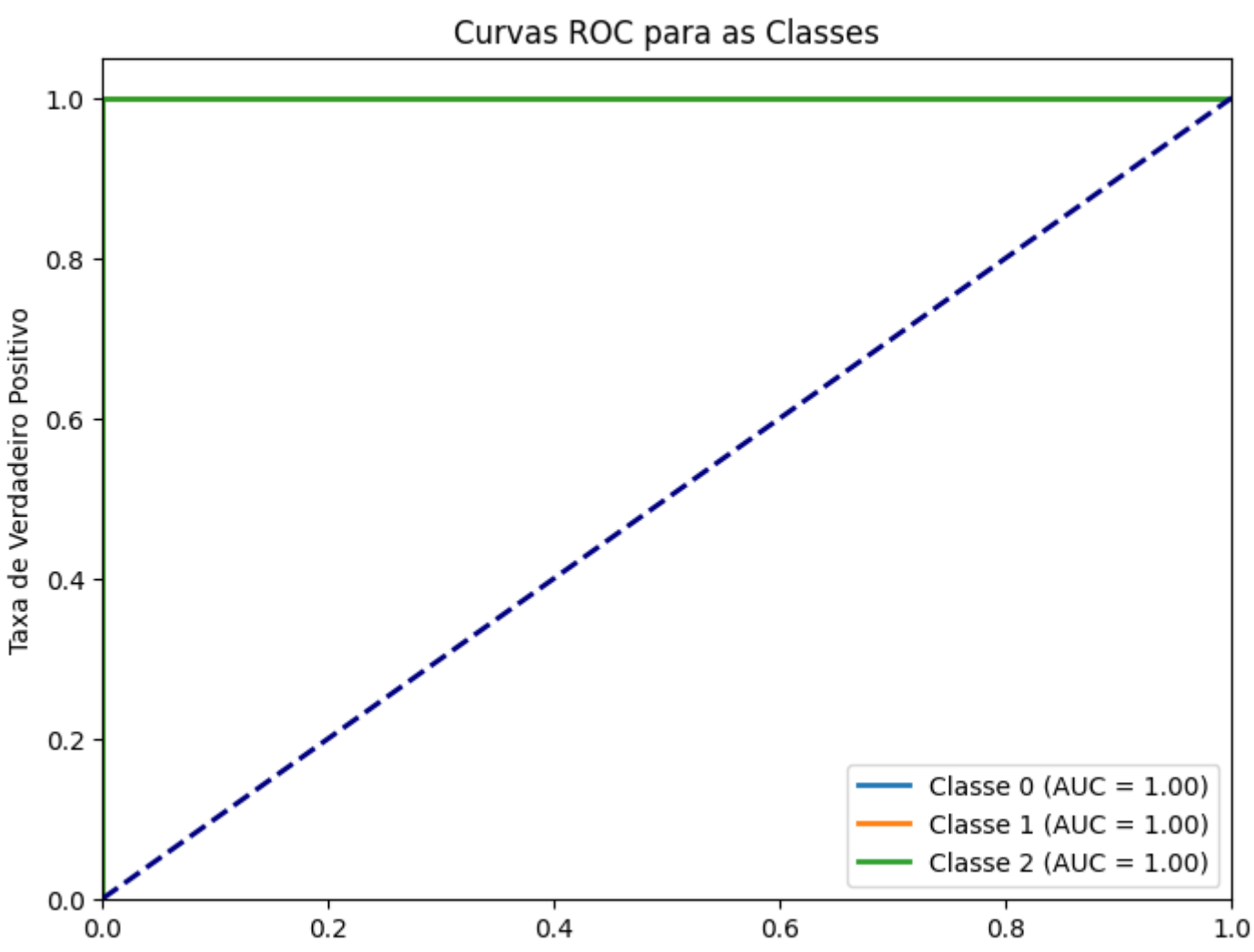
```
Epoch 94/100
8/8 - 0s - loss: 0.0472 - accuracy: 0.9833 - 18ms/epoch - 2ms/step
Epoch 95/100
8/8 - 0s - loss: 0.0461 - accuracy: 0.9833 - 22ms/epoch - 3ms/step
Epoch 96/100
8/8 - 0s - loss: 0.0457 - accuracy: 0.9833 - 22ms/epoch - 3ms/step
Epoch 97/100
8/8 - 0s - loss: 0.0458 - accuracy: 0.9833 - 15ms/epoch - 2ms/step
Epoch 98/100
8/8 - 0s - loss: 0.0467 - accuracy: 0.9833 - 18ms/epoch - 2ms/step
Epoch 99/100
8/8 - 0s - loss: 0.0468 - accuracy: 0.9833 - 15ms/epoch - 2ms/step
Epoch 100/100
8/8 - 0s - loss: 0.0462 - accuracy: 0.9833 - 15ms/epoch - 2ms/step
```




```
1/1 - 0s - loss: 0.0216 - accuracy: 1.0000 - 178ms/epoch - 178ms/step
Acurácia no conjunto de teste: 100.00%
1/1 [==============================] - 0s 101ms/step
A flor pertence à classe: Setosa
1/1 [==============================] - 0s 33ms/step
```



	precision	recall	f1-score	support
Setosa	1.00	1.00	1.00	10
Versicolor	1.00	1.00	1.00	9
Virginica	1.00	1.00	1.00	11
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30



6. Considere a base de dados encontrada em Irisdat.xlsx.

   Daí, pede-se:

   a) Treinar um PMC que classifique observações de flores íris em 3 espécies (Setosa, Versicolor e Virginica) usando como entradas as características SEPALLENGTH (SL), SEPALWIDTH (SW), PETALLENGTH (PL) e PETALWIDTH (PW).

In [ ]:
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report
from sklearn.linear_model import LinearRegression
import os

#print(os.getcwd())

# current_directory = os.path.dirname(os.path.abspath(__file__))
# os.path.abspath('')
# file_path = os.path.join(current_directory, "Irisdat.xlsx")
file_path = os.path.abspath("Irisdat.xlsx")
data = pd.read_excel(file_path)

# # Carregar os dados do arquivo Excel
# data = pd.read_excel("Irisdat.xls")

# # Exibir as primeiras linhas do conjunto de dados para compreendê-lo
print(data.head())


# Separar as características (entradas) e os rótulos (espécies)
X = data[["SL", "SW", "PL", "PW"]]
y = data["TYPE"]

# Dividir os dados em conjuntos de treinamento e teste (80% para treinamento, 20% para teste)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Padronizar os dados (média 0, desvio padrão 1) usando o StandardScaler original
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Treinar um Perceptron Multicamadas (PMC) para classificar as espécies
mlp = MLPClassifier(hidden_layer_sizes=(8, 8), max_iter=1000, random_state=42)
mlp.fit(X_train, y_train)

# Avaliar o modelo PMC
y_pred = mlp.predict(X_test)
print("Relatório de Classificação:\n", classification_report(y_test, y_pred))
```

```
    SL   SW   PL   PW      TYPE
0  5.0  3.3  1.4  0.2    SETOSA
1  6.4  2.8  5.6  2.2  VIRGINIC
2  6.5  2.8  4.6  1.5  VERSICOL
3  6.7  3.1  5.6  2.4  VIRGINIC
4  6.3  2.8  5.1  1.5  VIRGINIC
Relatório de Classificação:
              precision    recall  f1-score   support

      SETOSA       1.00      1.00      1.00         9
    VERSICOL       0.82      0.90      0.86        10
    VIRGINIC       0.89      0.80      0.84        10

    accuracy                           0.90        29
   macro avg       0.90      0.90      0.90        29
weighted avg       0.90      0.90      0.90        29
```

b) Estime SL a partir de SW, PL, PW.

In [ ]:
```python
from sklearn.metrics import mean_squared_error, r2_score

# Treinar um modelo de regressão linear para estimar SL com base em SW, PL e PW
X_reg = data[["SW", "PL", "PW"]]
y_reg = data["SL"]

# Padronizar os dados para o modelo de regressão linear com base em SW, PL e PW
scaler_reg = StandardScaler()
X_reg = scaler_reg.fit_transform(X_reg)

# Divide os dados em conjunto de treino e teste (80% para treino, 20% para teste)
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X_reg, y_reg, test_size=0.2, random_state=42)

reg = LinearRegression()
reg.fit(X_train_reg, y_train_reg)

# Fazer previsões usando o conjunto de teste
previsoes_reg = reg.predict(X_test_reg)

# Calcula métricas de desempenho do modelo de regressão
mse = mean_squared_error(y_test_reg, previsoes_reg)
r2 = r2_score(y_test_reg, previsoes_reg)

print('\nResultado das métricas de desepenho do modelo de regressão:')
print('\tErro quadrático médio (MSE):', mse)
print('\tCoeficiente de determinação (R^2):', r2)

print('\nExemplo de estimação de SL, a partir de SW, PL e PW:')
# Estimar SL a partir de SW, PL e PW
input_data = np.array([[3.5, 1.5, 0.2]])  # Substitua com os valores de SW, PL e PW que deseja estimar SL
scaled_input_data = scaler_reg.transform(input_data)
estimated_SL = reg.predict(scaled_input_data)
print(f"\tEstimativa de SL: {estimated_SL[0]}.")
```

```
Resultado das métricas de desepenho do modelo de regressão:
        Erro quadrático médio (MSE): 0.09470979693558303
        Coeficiente de determinação (R^2): 0.8753272301170403

Exemplo de estimação de SL, a partir de SW, PL e PW:
        Estimativa de SL: 5.090455067235911.
c:\Users\tecnoind\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\base.py:439: UserWarning: X does not have valid feature names, but StandardScaler was fitted with feature names
  warnings.warn(
```

O erro quadrático médio (MSE) de aproximadamente 0.0947 indica que o modelo tem uma precisão razoável na estimativa de SL com base nas outras características.

Além disso, o coeficiente de determinação (R^2) de aproximadamente 0.8753 sugere que o modelo explica cerca de 87,53% da variabilidade em SL usando as características SW, PL e PW. Quanto mais próximo de 1 for o valor de R^2, melhor o modelo se ajusta aos dados.

Para SW = 3.5, PL = 1.5 e PW = 0.2, o valor estimado para SL foi igual a 5.090455067235911.

7. Considere a base de dados encontrada em engines.xlsx, em que 'Fuel rate' e 'Speed' são variáveis de entrada e 'Torque' e 'Nitrous Oxide Emissions (NOE)' são as variáveis de saída, respectivamente. Desenvolva três regressores. Um deles deve estimar conjuntamente o 'Torque' e o NOE. Já os outros dois devem estimar essas saídas separadamente (i.e. um estimará o Torque e o outro o NOE). Compare o desempenho das duas estratégias apontando qual delas apresenta uma maior capacidade de generalização.

In [ ]:
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import make_scorer
import os

# print(os.getcwd())
```

```python
# Carregar os dados do arquivo Excel
# current_directory = os.path.dirname(os.path.abspath(__file__))
# file_path = os.path.join(current_directory, "engines.xlsx")
file_path = os.path.abspath('engines.xlsx')
data = pd.read_excel(file_path)

# Separar as características de entrada (X) e as variáveis de saída (Y)
X = data[["fuel rate", "speed"]]
Y = data[["torque", "nitrous oxide emissions (NOE)"]]

# Dividir os dados em conjuntos de treinamento e teste
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

# Padronizar os dados (média 0, desvio padrão 1)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

### ------------------------------------------------------------------------------------------
### PRIMEIRO MODELO
### ------------------------------------------------------------------------------------------

# Treinar o primeiro regressor para estimar "torque" e "NOE"
regressor1 = MLPRegressor(hidden_layer_sizes=(8, 8), max_iter=10000, random_state=42)
regressor1.fit(X_train, Y_train)

# Fazer previsões usando o primeiro regressor
predictions1 = regressor1.predict(X_test)

print('\nDesempenho dos modelos:')

# Avaliar o desempenho do primeiro regressor (por exemplo, com MSE)
mse1 = mean_squared_error(Y_test, predictions1)
print("\tMSE para regressor 1:", mse1)

### ------------------------------------------------------------------------------------------
### SEGUNDO MODELO
### ------------------------------------------------------------------------------------------

# Treinar o segundo regressor para estimar somente "torque"
Y_train_torque = Y_train["torque"]
regressor2 = MLPRegressor(hidden_layer_sizes=(8, 8), max_iter=10000, random_state=42)
regressor2.fit(X_train, Y_train_torque)

# Fazer previsões usando o segundo regressor
predictions2 = regressor2.predict(X_test)

# Avaliar o desempenho do segundo regressor (por exemplo, com MSE)
mse2 = mean_squared_error(Y_test["torque"], predictions2)
print("\tMSE para regressor 2 (torque):", mse2)

### ------------------------------------------------------------------------------------------
### TERCEIRO MODELO
### ------------------------------------------------------------------------------------------

# Treinar o terceiro regressor para estimar somente "NOE"
Y_train_noe = Y_train["nitrous oxide emissions (NOE)"]
regressor3 = MLPRegressor(hidden_layer_sizes=(8, 8), max_iter=10000, random_state=42)
regressor3.fit(X_train, Y_train_noe)

# Fazer previsões usando o terceiro regressor
predictions3 = regressor3.predict(X_test)

# Avaliar o desempenho do terceiro regressor (por exemplo, com MSE)
mse3 = mean_squared_error(Y_test["nitrous oxide emissions (NOE)"], predictions3)
print("\tMSE para regressor 3 (NOE):", mse3)
```

```
Desempenho dos modelos:
        MSE para regressor 1: 10152.975156644128
        MSE para regressor 2 (torque): 459.62327720999474
        MSE para regressor 3 (NOE): 19090.223831557134
```

In [ ]:

```python
### ------------------------------------------------------------------------------------------
### AVALIANDO MELHOR A CAPACIDADE DE GENERALIZAÇÃO DOS REGRESSORES
### ------------------------------------------------------------------------------------------

def custom_mse(y_true, y_pred):
    return mean_squared_error(y_true, y_pred)

# Validação cruzada com MSE personalizado
mse_scorer = make_scorer(custom_mse, greater_is_better=False)

# Regressor 1 (torque e NOE)
scores1 = -cross_val_score(regressor1, X, Y, cv=5, scoring=mse_scorer)
mean_score1 = scores1.mean()
std_score1 = scores1.std()

# Regressor 2 (somente torque)
scores2 = -cross_val_score(regressor2, X, Y["torque"], cv=5, scoring=mse_scorer)
mean_score2 = scores2.mean()
std_score2 = scores2.std()

# Regressor 3 (somente NOE)
```

```
scores3 = -cross_val_score(regressor3, X, Y["nitrous oxide emissions (NOE)"], cv=5, scoring=mse_scorer)
mean_score3 = scores3.mean()
std_score3 = scores3.std()

print('\nDesempenho dos modelos com base na validação cruzada:')
print("\tRegressor 1 (torque e NOE) - MSE Médio:", mean_score1, "Desvio Padrão:", std_score1)
print("\tRegressor 2 (somente torque) - MSE Médio:", mean_score2, "Desvio Padrão:", std_score2)
print("\tRegressor 3 (somente NOE) - MSE Médio:", mean_score3, "Desvio Padrão:", std_score3)
```

```
Desempenho dos modelos com base na validação cruzada:
        Regressor 1 (torque e NOE) - MSE Médio: 66417.2675522822 Desvio Padrão: 45149.659829828575
        Regressor 2 (somente torque) - MSE Médio: 2474.250283230132 Desvio Padrão: 1172.2613093811758
        Regressor 3 (somente NOE) - MSE Médio: 143385.97775100663 Desvio Padrão: 80364.00570653014
```

8. Valendo-se da base de dados reais referente ao Volume de Vendas de Passagens (VVP) de uma companhia aérea norte-americana que se encontra no arquivo vvp.xlsx, pede-se:

a. Desenvolver um previsor neural que receba como entradas os VVPs registrados nos instantes k-1 e k-12 (i.e. VVP(k-1) e VVP(k-12)) e que disponibilize na saída o VVP no instante corrente k (i.e. VVP(k)). O previsor deverá realizar previsões recursivas de 1 a 12 passos à frente (i.e., de um a doze meses à frente). Os dados se encontram no arquivo 'vvp.csv'.

In [ ]:

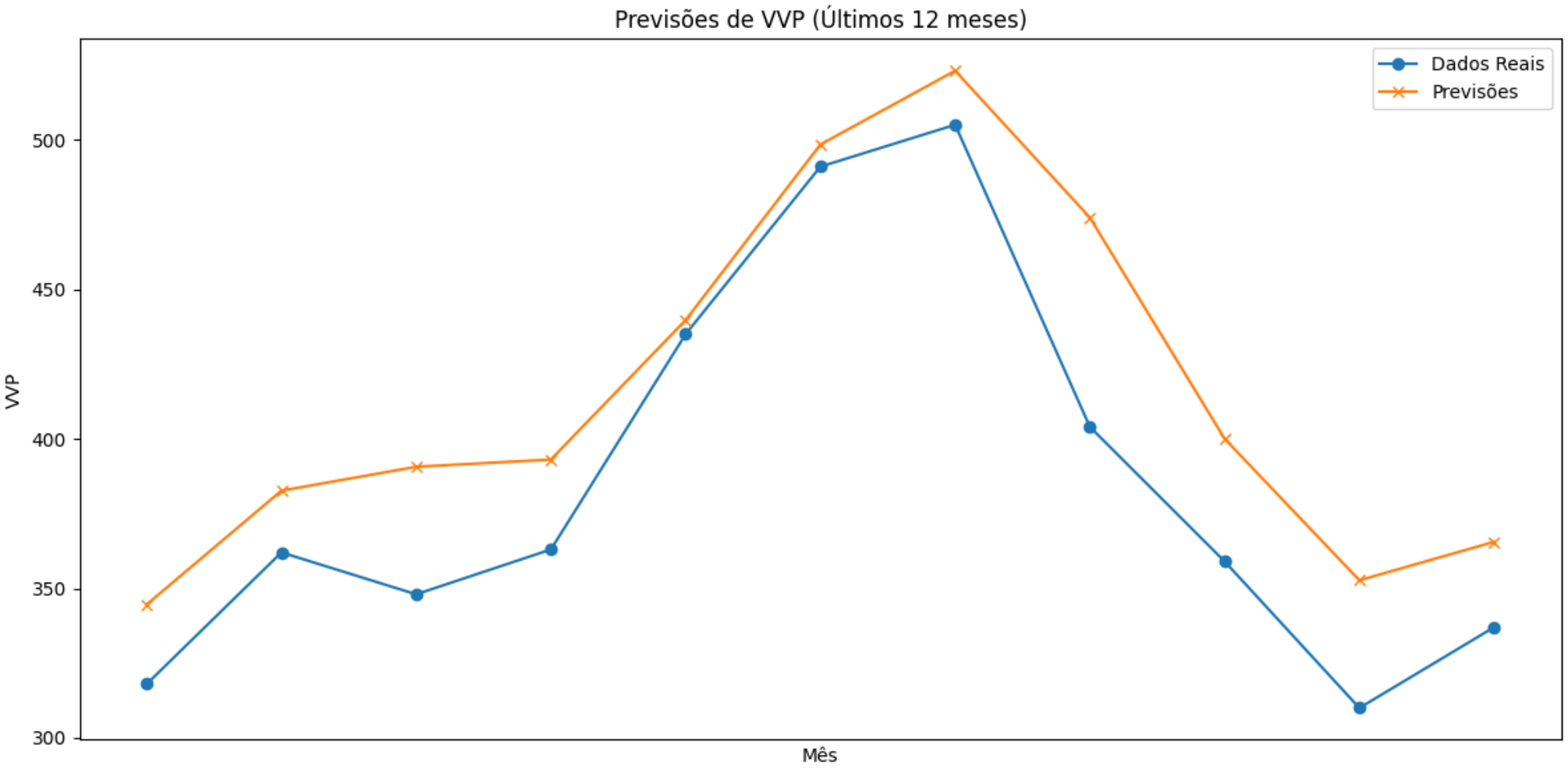
```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# Carregando os dados
data = pd.read_csv('vvp.csv')

# Certifique-se de que os dados estão classificados por data
data['Date'] = pd.to_datetime(data['mes'])
data = data.sort_values('Date')
data.set_index('Date', inplace=True)
#data['Date'] = data['mes'].dt.to_period('M')
#data['Date'] = data.mes.dt.to_period('M')

# Normalizando os dados
scaler = MinMaxScaler()
data['VVP'] = scaler.fit_transform(data['VVP'].values.reshape(-1, 1))

# Adicionando colunas de deslocamento (k-1 e k-12)
data['VVP_lag1'] = data['VVP'].shift(1)
data['VVP_lag12'] = data['VVP'].shift(12)

# Removendo valores ausentes
data = data.dropna()

# Separando recursos (X) e alvo (y)
X = data[['VVP_lag1', 'VVP_lag12']]
y = data['VVP']

# Dividindo os dados em treinamento e teste
split_ratio = 0.9
split_index = int(split_ratio * len(data))

X_train, X_test = X.iloc[:split_index], X.iloc[split_index:]
y_train, y_test = y.iloc[:split_index], y.iloc[split_index:]


# Construindo o modelo neural
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='linear'))

model.compile(optimizer='adam', loss='mean_squared_error')

# Treinando o modelo
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test))

# Previsões
y_pred = model.predict(X_test)

# Invertendo a escala para obter previsões reais
y_pred_inv = scaler.inverse_transform(y_pred)
y_test_inv = scaler.inverse_transform(y_test.values.reshape(-1, 1))
y_real_inv = y_test_inv

# # Visualização dos dados e previsões

plt.figure(figsize=(12, 6))
# plt.plot(data.index[split_index:], y_test_inv, label='Dados Reais', marker='o')
plt.plot(data.index[-len(y_test_inv):], y_test_inv, label='Dados Reais', marker='o')
# plt.plot(data.index[split_index:], y_pred_inv, label='Previsões', marker='x')
plt.plot(data.index[-len(y_pred_inv):], y_pred_inv, label='Previsões', marker='x')
plt.xlabel('Mês')
plt.ylabel('VVP')
plt.title('Previsões de VVP (Últimos 12 meses)')
```

```
plt.legend()
plt.tight_layout()
plt.show()
```

```
Epoch 1/100
4/4 [==============================] - 1s 67ms/step - loss: 0.1515 - val_loss: 0.3914
Epoch 2/100
4/4 [==============================] - 0s 16ms/step - loss: 0.1102 - val_loss: 0.2907
Epoch 3/100
4/4 [==============================] - 0s 23ms/step - loss: 0.0784 - val_loss: 0.2059
Epoch 4/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0528 - val_loss: 0.1377
Epoch 5/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0321 - val_loss: 0.0855
Epoch 6/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0191 - val_loss: 0.0473
Epoch 7/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0106 - val_loss: 0.0232
Epoch 8/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0069 - val_loss: 0.0126
Epoch 9/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0063 - val_loss: 0.0091
Epoch 10/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0063 - val_loss: 0.0083
Epoch 11/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0059 - val_loss: 0.0084
Epoch 12/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0051 - val_loss: 0.0091
Epoch 13/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0046 - val_loss: 0.0081
Epoch 14/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0042 - val_loss: 0.0064
Epoch 15/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0041 - val_loss: 0.0061
Epoch 16/100
4/4 [==============================] - 0s 19ms/step - loss: 0.0037 - val_loss: 0.0059
Epoch 17/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0031 - val_loss: 0.0062
Epoch 18/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0030 - val_loss: 0.0066
Epoch 19/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0030 - val_loss: 0.0063
Epoch 20/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0028 - val_loss: 0.0054
Epoch 21/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0024 - val_loss: 0.0051
Epoch 22/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0023 - val_loss: 0.0054
Epoch 23/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0022 - val_loss: 0.0054
Epoch 24/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0020 - val_loss: 0.0055
Epoch 25/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0020 - val_loss: 0.0068
Epoch 26/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0021 - val_loss: 0.0074
Epoch 27/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0019 - val_loss: 0.0065
Epoch 28/100
4/4 [==============================] - 0s 11ms/step - loss: 0.0016 - val_loss: 0.0056
Epoch 29/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0015 - val_loss: 0.0061
Epoch 30/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0015 - val_loss: 0.0065
Epoch 31/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0015 - val_loss: 0.0085
Epoch 32/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0016 - val_loss: 0.0080
Epoch 33/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0014 - val_loss: 0.0049
Epoch 34/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0015 - val_loss: 0.0043
Epoch 35/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0015 - val_loss: 0.0050
Epoch 36/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0014 - val_loss: 0.0062
Epoch 37/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0014 - val_loss: 0.0069
Epoch 38/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0014 - val_loss: 0.0058
Epoch 39/100
4/4 [==============================] - 0s 16ms/step - loss: 0.0013 - val_loss: 0.0055
Epoch 40/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0013 - val_loss: 0.0055
Epoch 41/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0013 - val_loss: 0.0052
Epoch 42/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0013 - val_loss: 0.0055
Epoch 43/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0012 - val_loss: 0.0064
Epoch 44/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0012 - val_loss: 0.0069
Epoch 45/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0012 - val_loss: 0.0066
Epoch 46/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0012 - val_loss: 0.0065
Epoch 47/100
```

```
4/4 [==============================] - 0s 12ms/step - loss: 0.0012 - val_loss: 0.0068
Epoch 48/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0012 - val_loss: 0.0077
Epoch 49/100
4/4 [==============================] - 0s 46ms/step - loss: 0.0012 - val_loss: 0.0089
Epoch 50/100
4/4 [==============================] - 0s 38ms/step - loss: 0.0013 - val_loss: 0.0086
Epoch 51/100
4/4 [==============================] - 0s 30ms/step - loss: 0.0013 - val_loss: 0.0087
Epoch 52/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0013 - val_loss: 0.0082
Epoch 53/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0012 - val_loss: 0.0068
Epoch 54/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0011 - val_loss: 0.0055
Epoch 55/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0011 - val_loss: 0.0048
Epoch 56/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0012 - val_loss: 0.0051
Epoch 57/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0011 - val_loss: 0.0060
Epoch 58/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0010 - val_loss: 0.0077
Epoch 59/100
4/4 [==============================] - 0s 17ms/step - loss: 0.0011 - val_loss: 0.0079
Epoch 60/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0011 - val_loss: 0.0069
Epoch 61/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0010 - val_loss: 0.0057
Epoch 62/100
4/4 [==============================] - 0s 17ms/step - loss: 0.0010 - val_loss: 0.0056
Epoch 63/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0010 - val_loss: 0.0067
Epoch 64/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0010 - val_loss: 0.0075
Epoch 65/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0010 - val_loss: 0.0069
Epoch 66/100
4/4 [==============================] - 0s 13ms/step - loss: 9.9928e-04 - val_loss: 0.0045
Epoch 67/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0012 - val_loss: 0.0040
Epoch 68/100
4/4 [==============================] - 0s 15ms/step - loss: 0.0012 - val_loss: 0.0052
Epoch 69/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0010 - val_loss: 0.0072
Epoch 70/100
4/4 [==============================] - 0s 20ms/step - loss: 0.0010 - val_loss: 0.0075
Epoch 71/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0010 - val_loss: 0.0049
Epoch 72/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0011 - val_loss: 0.0043
Epoch 73/100
4/4 [==============================] - 0s 11ms/step - loss: 0.0011 - val_loss: 0.0055
Epoch 74/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0010 - val_loss: 0.0076
Epoch 75/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0011 - val_loss: 0.0086
Epoch 76/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0010 - val_loss: 0.0059
Epoch 77/100
4/4 [==============================] - 0s 17ms/step - loss: 9.9641e-04 - val_loss: 0.0045
Epoch 78/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0011 - val_loss: 0.0052
Epoch 79/100
4/4 [==============================] - 0s 12ms/step - loss: 9.9312e-04 - val_loss: 0.0077
Epoch 80/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0011 - val_loss: 0.0089
Epoch 81/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0012 - val_loss: 0.0075
Epoch 82/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0010 - val_loss: 0.0055
Epoch 83/100
4/4 [==============================] - 0s 12ms/step - loss: 9.7496e-04 - val_loss: 0.0048
Epoch 84/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0010 - val_loss: 0.0053
Epoch 85/100
4/4 [==============================] - 0s 14ms/step - loss: 9.6644e-04 - val_loss: 0.0062
Epoch 86/100
4/4 [==============================] - 0s 14ms/step - loss: 9.6950e-04 - val_loss: 0.0062
Epoch 87/100
4/4 [==============================] - 0s 12ms/step - loss: 9.5664e-04 - val_loss: 0.0042
Epoch 88/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0012 - val_loss: 0.0032
Epoch 89/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0012 - val_loss: 0.0046
Epoch 90/100
4/4 [==============================] - 0s 12ms/step - loss: 9.5541e-04 - val_loss: 0.0078
Epoch 91/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0010 - val_loss: 0.0093
Epoch 92/100
4/4 [==============================] - 0s 11ms/step - loss: 0.0012 - val_loss: 0.0110
Epoch 93/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0012 - val_loss: 0.0083
```

```
Epoch 94/100
4/4 [==============================] - 0s 12ms/step - loss: 9.5591e-04 - val_loss: 0.0056
Epoch 95/100
4/4 [==============================] - 0s 14ms/step - loss: 0.0010 - val_loss: 0.0052
Epoch 96/100
4/4 [==============================] - 0s 12ms/step - loss: 9.6873e-04 - val_loss: 0.0079
Epoch 97/100
4/4 [==============================] - 0s 14ms/step - loss: 9.4314e-04 - val_loss: 0.0096
Epoch 98/100
4/4 [==============================] - 0s 12ms/step - loss: 0.0010 - val_loss: 0.0098
Epoch 99/100
4/4 [==============================] - 0s 13ms/step - loss: 0.0010 - val_loss: 0.0091
Epoch 100/100
4/4 [==============================] - 0s 13ms/step - loss: 9.6353e-04 - val_loss: 0.0076
1/1 [==============================] - 0s 61ms/step
```

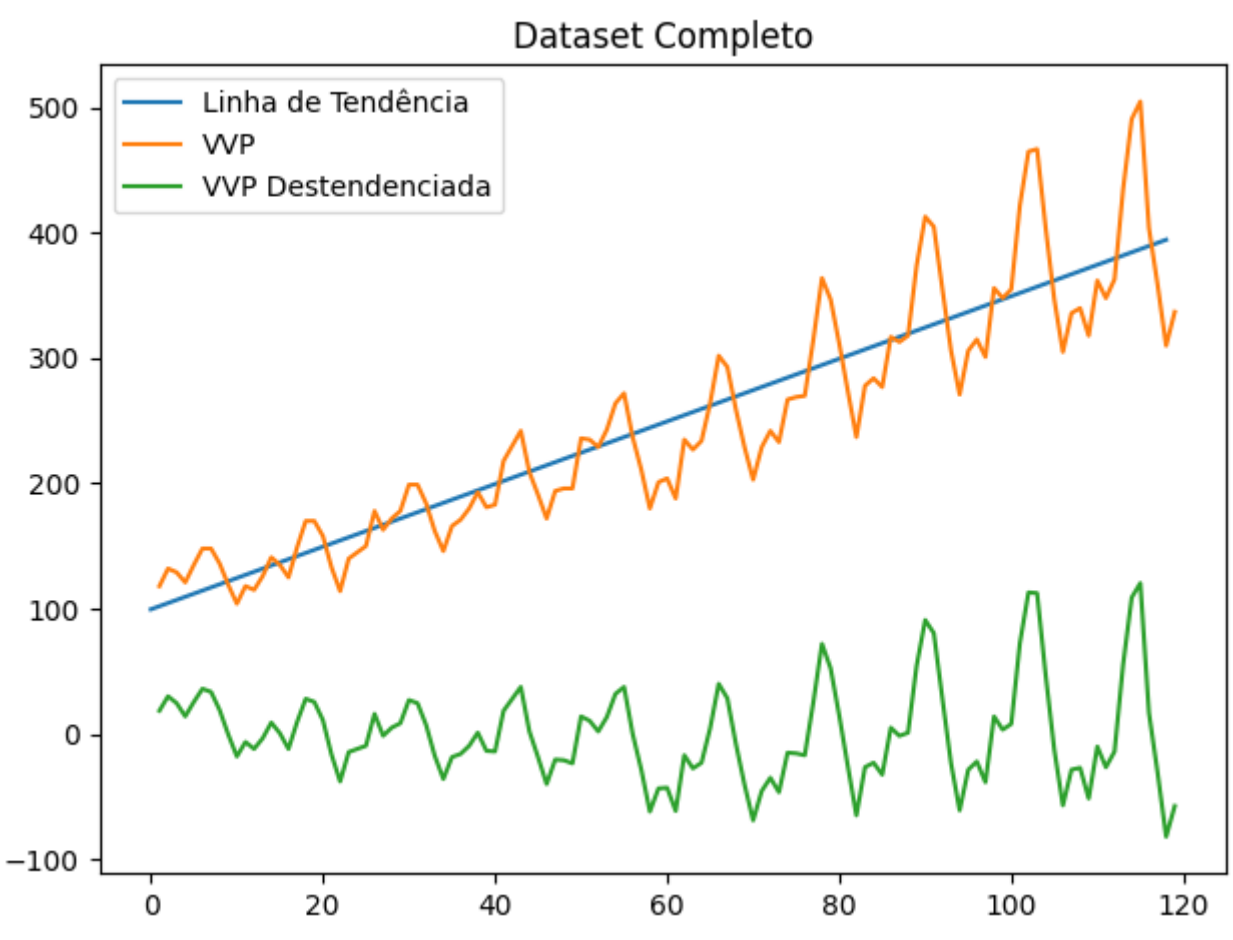


b) De posse da base de dados, remova a tendência linear presente na base de dados original. Desse modo, você conhecerá a série destendenciada e a tendência linear. Para a primeira série, desenvolva um previsor neural que receba como entradas os VVPs registrados nos instantes k-1 e k-12 (i.e. VVP(k-1) e VVP(k-12)) e que disponibilize na saída o VVP no instante corrente k (i.e. VVP(k)). O previsor deverá realizar previsões recursivas de 1 a 12 passos à frente (i.e., de um a doze meses à frente). Para a segunda (i.e., a tendência linear), preveja linearmente os próximos dozes pontos. Em seguida, some ponto a ponto as duas previsões e compare o desempenho dessa abordagem com a anterior apontando qual delas apresenta uma maior capacidade de generalização.

In [ ]:

```python
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Carregue a base de dados do arquivo CSV
data = pd.read_csv('vvp.csv')

# Suponha que 'data' contenha a coluna 'VVP' que representa o Volume de Vendas de Passagens

# Calcule a diferença entre cada ponto e o ponto anterior
data['VVP_diff'] = data['VVP'] - data['VVP'].shift(1)

# A primeira entrada resultará em um NaN, então você pode removê-la
data = data.dropna()

X = [i for i in range(0, len(data['VVP']))]
X = np.reshape(X, (len(X), 1))
y = data['VVP']

# Crie um objeto de modelo de regressão linear para a tendência linear
model_trend = LinearRegression()


model_trend.fit(X,y)
trend = model_trend.predict(X)

data['VVP_destendenciada'] = data['VVP'] - trend

plt.plot(trend, label = 'Linha de Tendência')
plt.plot(y, label = 'VVP')
plt.plot(data['VVP_destendenciada'], label= 'VVP Destendenciada')
plt.title('Dataset Completo')
plt.legend()
plt.tight_layout()
plt.show()

trend_to_add = trend[-11:]
```

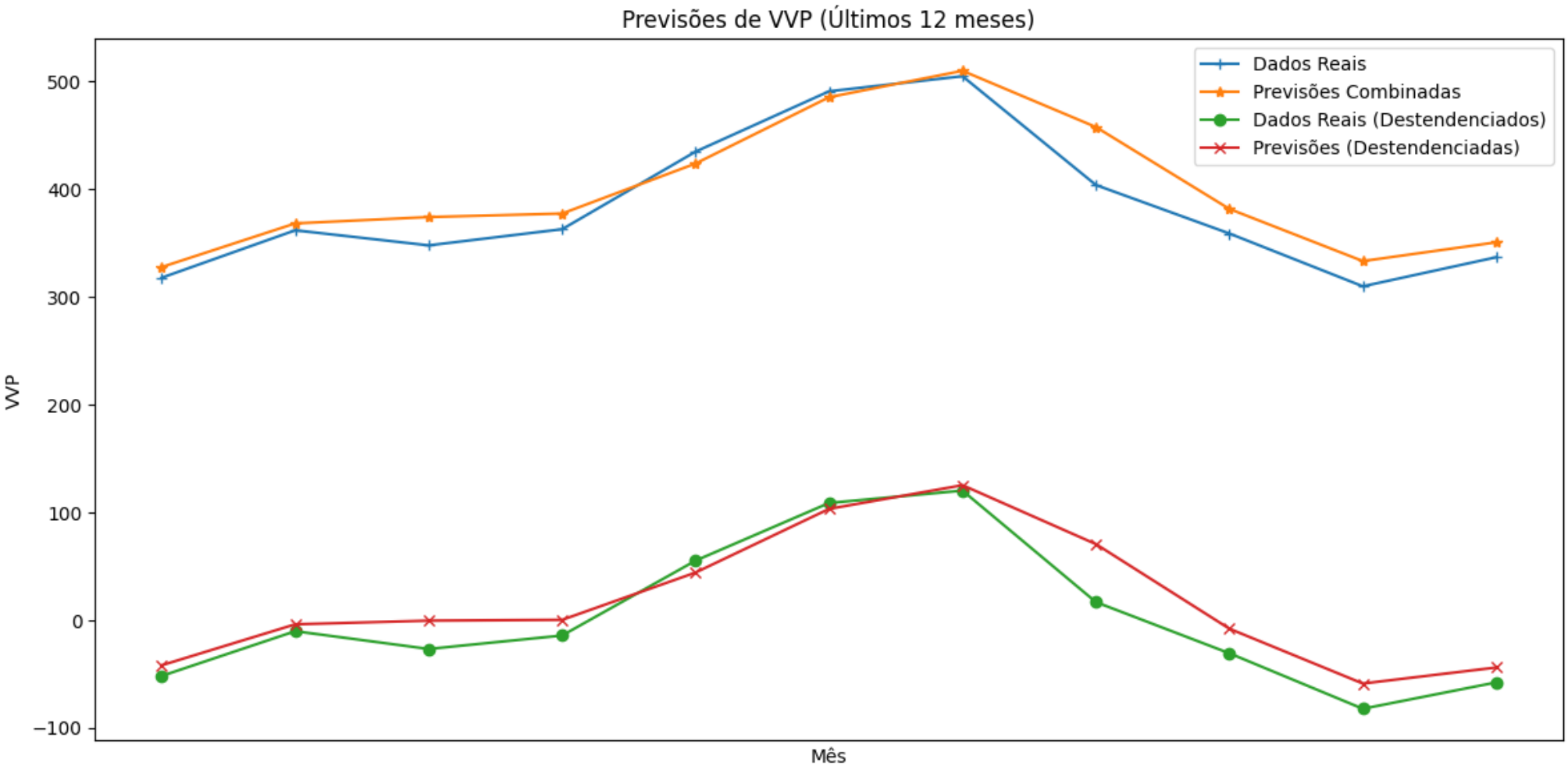
```
trend_to_add = np.array(trend_to_add)

# Certifique-se de que os dados estão classificados por data
data['Date'] = pd.to_datetime(data['mes'])
data = data.sort_values('Date')
data.set_index('Date', inplace=True)

# Normalizando os dados
scaler = MinMaxScaler()
data['VVP_destendenciada'] = scaler.fit_transform(data['VVP_destendenciada'].values.reshape(-1, 1))

# Adicionando colunas de deslocamento (k-1 e k-12)
data['VVP_lag1'] = data['VVP_destendenciada'].shift(1)
data['VVP_lag12'] = data['VVP_destendenciada'].shift(12)

# Removendo valores ausentes
data = data.dropna()

# Separando recursos (X) e alvo (y)
X = data[['VVP_lag1', 'VVP_lag12']]
y = data['VVP_destendenciada']

# Dividindo os dados em treinamento e teste
split_ratio = 0.9
split_index = int(split_ratio * len(data))
# print(f'split_index = {split_index}')
X_train, X_test = X.iloc[:split_index], X.iloc[split_index:]
y_train, y_test = y.iloc[:split_index], y.iloc[split_index:]

# Construindo o modelo neural
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='linear'))

model.compile(optimizer='adam', loss='mean_squared_error')

# Treinando o modelo
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test))

# Previsões
y_pred = model.predict(X_test)

# Invertendo a escala para obter previsões reais
y_pred_inv = scaler.inverse_transform(y_pred)
y_test_inv = scaler.inverse_transform(y_test.values.reshape(-1, 1))

#print(f'previsão = {y_pred_inv} {(y_pred_inv).shape}')
trend_to_add = np.reshape(trend_to_add, (11, 1))
#print(f'tendência = {trend_to_add}, {(trend_to_add).shape}')

y_pred_combined = np.add(y_pred_inv, trend_to_add)
#print(f'combinado = {y_pred_combined}')

# Visualização dos dados e previsões
plt.figure(figsize=(12, 6))
plt.plot(data.index[split_index:], y_real_inv, label='Dados Reais', marker='+')
plt.plot(data.index[split_index:], y_pred_combined, label='Previsões Combinadas', marker='*')
plt.plot(data.index[split_index:], y_test_inv, label='Dados Reais (Destendenciados)', marker='o')
plt.plot(data.index[split_index:], y_pred_inv, label='Previsões (Destendenciadas)', marker='x')
plt.xlabel('Mês')
plt.ylabel('VVP')
plt.title('Previsões de VVP (Últimos 12 meses)')
plt.legend()
plt.tight_layout()
plt.show()

#print(trend_to_add)
#print(y_pred_inv)
#print(y_pred_inv + trend_to_add)

# Calcule o erro médio quadrático (MSE) para ambas as abordagens
mse_destendenciada = mean_squared_error(data['VVP'][split_index:], y_pred_inv)
mse_trend = mean_squared_error(data['VVP'][split_index:], trend_to_add)
mse_combined = mean_squared_error(data['VVP'][split_index:], y_pred_combined)

print("MSE da abordagem da série destendenciada:", mse_destendenciada)
print("MSE da abordagem da tendência linear:", mse_trend)
print("MSE da abordagem combinada:", mse_combined)
```


Dataset Completo
Linha de Tendência
VVP
VVP Destendenciada
500
400
300
200
100
0
−100
0
20
40
60
80
100
120

```
Epoch 1/100
3/3 [==============================] - 1s 77ms/step - loss: 0.0570 - val_loss: 0.0660
Epoch 2/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0331 - val_loss: 0.0440
Epoch 3/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0187 - val_loss: 0.0323
Epoch 4/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0115 - val_loss: 0.0284
Epoch 5/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0088 - val_loss: 0.0295
Epoch 6/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0095 - val_loss: 0.0314
Epoch 7/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0099 - val_loss: 0.0304
Epoch 8/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0092 - val_loss: 0.0270
Epoch 9/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0078 - val_loss: 0.0231
Epoch 10/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0068 - val_loss: 0.0207
Epoch 11/100
3/3 [==============================] - 0s 22ms/step - loss: 0.0067 - val_loss: 0.0192
Epoch 12/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0068 - val_loss: 0.0182
Epoch 13/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0067 - val_loss: 0.0175
Epoch 14/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0062 - val_loss: 0.0173
Epoch 15/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0058 - val_loss: 0.0173
Epoch 16/100
3/3 [==============================] - 0s 22ms/step - loss: 0.0056 - val_loss: 0.0171
Epoch 17/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0054 - val_loss: 0.0164
Epoch 18/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0052 - val_loss: 0.0148
Epoch 19/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0050 - val_loss: 0.0137
Epoch 20/100
3/3 [==============================] - 0s 26ms/step - loss: 0.0049 - val_loss: 0.0129
Epoch 21/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0048 - val_loss: 0.0128
Epoch 22/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0047 - val_loss: 0.0132
Epoch 23/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0047 - val_loss: 0.0136
Epoch 24/100
3/3 [==============================] - 0s 16ms/step - loss: 0.0046 - val_loss: 0.0131
Epoch 25/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0046 - val_loss: 0.0129
Epoch 26/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0125
Epoch 27/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0045 - val_loss: 0.0125
Epoch 28/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0129
Epoch 29/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0045 - val_loss: 0.0130
Epoch 30/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0045 - val_loss: 0.0129
Epoch 31/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0045 - val_loss: 0.0126
Epoch 32/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0045 - val_loss: 0.0127
Epoch 33/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0045 - val_loss: 0.0128
Epoch 34/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0126
Epoch 35/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0045 - val_loss: 0.0124
Epoch 36/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0130
Epoch 37/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0045 - val_loss: 0.0129
Epoch 38/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0045 - val_loss: 0.0127
Epoch 39/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0130
Epoch 40/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0045 - val_loss: 0.0131
Epoch 41/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0045 - val_loss: 0.0122
Epoch 42/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0045 - val_loss: 0.0125
Epoch 43/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0128
Epoch 44/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0130
Epoch 45/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0130
Epoch 46/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0045 - val_loss: 0.0126
Epoch 47/100
```

```
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0127
Epoch 48/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0132
Epoch 49/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0131
Epoch 50/100
3/3 [==============================] - 0s 23ms/step - loss: 0.0044 - val_loss: 0.0126
Epoch 51/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0126
Epoch 52/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0130
Epoch 53/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0128
Epoch 54/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0122
Epoch 55/100
3/3 [==============================] - 0s 54ms/step - loss: 0.0044 - val_loss: 0.0128
Epoch 56/100
3/3 [==============================] - 0s 27ms/step - loss: 0.0044 - val_loss: 0.0135
Epoch 57/100
3/3 [==============================] - 0s 25ms/step - loss: 0.0044 - val_loss: 0.0127
Epoch 58/100
3/3 [==============================] - 0s 22ms/step - loss: 0.0044 - val_loss: 0.0130
Epoch 59/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0124
Epoch 60/100
3/3 [==============================] - 0s 28ms/step - loss: 0.0044 - val_loss: 0.0129
Epoch 61/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0125
Epoch 62/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0131
Epoch 63/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0130
Epoch 64/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0045 - val_loss: 0.0120
Epoch 65/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0125
Epoch 66/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0129
Epoch 67/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0129
Epoch 68/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0044 - val_loss: 0.0136
Epoch 69/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0129
Epoch 70/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0116
Epoch 71/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0124
Epoch 72/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0044 - val_loss: 0.0135
Epoch 73/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0140
Epoch 74/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0124
Epoch 75/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0120
Epoch 76/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0125
Epoch 77/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0132
Epoch 78/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0131
Epoch 79/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0124
Epoch 80/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0044 - val_loss: 0.0126
Epoch 81/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0126
Epoch 82/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0125
Epoch 83/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0127
Epoch 84/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0131
Epoch 85/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0133
Epoch 86/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0134
Epoch 87/100
3/3 [==============================] - 0s 22ms/step - loss: 0.0044 - val_loss: 0.0120
Epoch 88/100
3/3 [==============================] - 0s 28ms/step - loss: 0.0044 - val_loss: 0.0126
Epoch 89/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0121
Epoch 90/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0131
Epoch 91/100
3/3 [==============================] - 0s 21ms/step - loss: 0.0044 - val_loss: 0.0136
Epoch 92/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0124
Epoch 93/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0045 - val_loss: 0.0131
```

```
Epoch 94/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0044 - val_loss: 0.0123
Epoch 95/100
3/3 [==============================] - 0s 18ms/step - loss: 0.0044 - val_loss: 0.0120
Epoch 96/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0132
Epoch 97/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0044 - val_loss: 0.0137
Epoch 98/100
3/3 [==============================] - 0s 17ms/step - loss: 0.0044 - val_loss: 0.0128
Epoch 99/100
3/3 [==============================] - 0s 19ms/step - loss: 0.0044 - val_loss: 0.0126
Epoch 100/100
3/3 [==============================] - 0s 20ms/step - loss: 0.0044 - val_loss: 0.0119
1/1 [==============================] - 0s 59ms/step
```




```
MSE da abordagem da série destendenciada: 135390.94842632103
MSE da abordagem da tendência linear: 4034.2174998677087
MSE da abordagem combinada: 488.375892584191
```

Considerando apenas o erro médio quadrático (MSE), a abordagem combinada parece apresentar o melhor desempenho.

9. Procure na literatura 2 artigos que tratem do tema Sensores Inferenciais (ou Soft Sensors) para uma dada grandeza de seu interesse (e.g. temperatura, pressão, vazão, nível etc.) e que tenham sido publicados nos últimos 5 anos. Explique de forma sucinta o que foi desenvolvido pelos autores, referenciando-os. Sugestão: As principais informações de qualquer artigo geralmente se encontram no título, no resumo e nas conclusões. Ao ler esses três itens, o leitor tem uma boa ideia do que esperar daquele trabalho. A propósito, usualmente o leitor decidirá se lerá todo o artigo ou não com base na sua impressão a respeito desses três itens.

Artigo 1: Data-driven soft sensors targeting heat pump systems Autores: Yang Song, Davide Rolando, Javier Marchante Avellaneda, Gerhard Zucker, Hatef Madani https://www.sciencedirect.com/science/article/pii/S0196890423001152

Neste trabalho, os autores (Yang Song, Davide Rolando, Javier Marchante Avellaneda, Gerhard Zucker e Hatef Madani) desenvolveram Sensores Inferenciais (soft sensors) baseados em dados para compensar informações ausentes em sistemas de bombas de calor. Os soft sensors foram desenvolvidos usando um modelo de rede neural artificial (ANN), um modelo de regressão polinomial multivariado integrado e um modelo empírico, levando em consideração diferentes restrições, como disponibilidade de dados e informações durante o processo de estabelecimento do modelo. Os três modelos foram validados com dados de uma instalação de teste de campo e mostraram bom desempenho para todas as variáveis compensadas (taxa de fluxo de massa de refrigeração, sensores de pressão e potência do compressor), conforme tabela abaixo.

Table 1. Soft sensors and related inputs in referred literature.

Literature	Soft sensor	Inputs
[23]	Refrigerant mass flow rate	$W, T_{dis}, P_{dis}, T_{suc}, P_{suc}, f_{comp}$
[27]	Refrigerant mass flow rate	$P_{dis}, P_{suc}, T_{suc}, T_{amb}$
[28]	Refrigerant mass flow rate	$W, T_c, P_c, T_e, P_e, T_{dis}, T_{suc}$
[24]	Pressure sensors	T_e, T_c
[25]	Compressor power	T_e, T_c, f_{comp}
[28]	Compressor power	$T_e, T_c, T_{suc}, P_{suc}$

Table 2. Available and missing variables for system performance analysis.

Available measurements	Missing information
Compressor speed (rpm)	Evaporation temperature/pressure
Water inlet temperature	Condensation temperature/pressure
Water outlet temperature	Mass flow rate of water/brine/refrigerant
Brine inlet temperature	Heating capacity
Brine outlet temperature	Cooling capacity
Compressor inlet temperature	Compressor power consumption
Compressor outlet temperature	COP
Condenser outlet temperature	

Table 3. Inputs and outputs of ANN models in referred literature.

Literature	Inputs	Outputs
[31]	Evaporator inlet temperature and pressure, Evaporator outlet temperature and pressure, Water inlet temperature in condenser, Discharge pressure	Heating capacity Compressor power consumption
[33]	Driving temperature difference, Vapor superheat, Corrugation enlargement ratio, Equivalent Reynolds number, Liquid Prandtl number	Heat transfer factor
[34]	Reduced pressure, Reduced temperature, Mole mass, Acentric factor	Thermal conductivity Viscosity

Table 5. Inputs and outputs of software and multivariate polynomial model for condenser.

Variables	Software	Model
Mass flow rate of refrigerant	Output	Input(x_1)
Inlet temperature of water	Input	Input(x_2)
Outlet temperature of water	Input	Input(x_3)
Inlet temperature of refrigerant	Input	Input(x_4)
Outlet temperature of refrigerant	Output	Input(x_5)
Subcooling	Input	–
Heating capacity	Input	–
Condensation temperature	Output	Output(T_c)

Table 6. Inputs and outputs of software and multivariate polynomial model for evaporator.

Variables	Software	Model
Mass flow rate of refrigerant	Output	Input(x_1)
Evaporator inlet enthalpy	Output	Input(x_2)
Outlet temperature of refrigerant	Output	Input(x_3)
Inlet temperature of brine	Input	Input(x_4)
Outlet temperature of brine	Input	Input(x_5)
Inlet quality to evaporator	Input	–
Superheating	Input	–
Cooling capacity	Input	–
Evaporation temperature	Output	Output(T_e)

Table 7. Inputs and outputs of software and multivariate polynomial model for compressor.

Variables	Software	Power model	Mass flow rate model
Compressor frequency	Input	Input(x_1)	Input(x_1)
Evaporation temperature	Input	Input(x_2)	Input(x_2)
Condensation temperature	Input	Input(x_3)	–
Superheating	Input	–	Input(x_3)
Power consumption	Output	Output(W)	–
Suction mass flow rate	Output	–	Output(m_{ref})
Subcooling	Input	–	–

O modelo ANN foi o mais preciso, mas requer mais recursos adicionais para coletar dados de treinamento. O modelo de regressão polinomial multivariado integrado mostrou excelente precisão para a maioria dos soft sensors com dados de subcomponentes do fabricante, sem custo adicional. O estudo demonstrou o potencial dos soft sensors para substituir sensores físicos caros e abrir oportunidades para serviços inovadores com dados de monitoramento incompletos.

Table 9. Summary of model regression performance.

Model type	$RMSE$ (K)		$RRMSE$ (%)				
	T_c	T_e	P_c	P_e	$\dot{m}_{ref}$	Q_c	W
ANN model	0.09	0.09	0.25	0.29	1.14	1.13	1.37
Integrated model	0.18	0.53	0.51	1.65	9.29	9.25	6.29
Empirical model	1.83	0.36	4.58	1.13	0.89	3.29	14.25

Isso inclui monitoramento do estado operacional da bomba de calor, previsão de consumo de energia e desenvolvimento de gêmeos digitais para gerenciamento avançado de energia.

Artigo 2: Soft sensors design in a petrochemical process using an Evolutionary Algorithm Autores: Gustavo A.P. de Morais, Bruno H.G. Barbosa, Danton D. Ferreira, Leonardo S. Paiva https://www.sciencedirect.com/science/article/pii/S0263224119307778?casa_token=W1xOW-naRKQAAAAA:ca54pHuAFLQvtGevvFeadfFwkGNeuuUehYL-vBljSBrTC1FC-a4RqkWX6SQKp8PsAjJZPXwCdZ0

Neste trabalho, os autores (Gustavo A.P. de Morais, Bruno H.G. Barbosa, Danton D. Ferreira e Leonardo S. Paiva) desenvolveram soft sensors para prever a pressão em poços de petróleo de águas profundas. Nestes ambientes, é crucial monitorar a pressão no fundo do poço para otimizar a produção de petróleo. O desafio é que os sensores reais localizados no leito do mar têm uma vida útil limitada devido às condições adversas, e a falta de informações precisas sobre a pressão pode afetar a produção de petróleo.
Para abordar esse problema, os autores propuseram um algoritmo chamado "Evolutionary Algorithm with Numerical Differentiation (EAND)" para projetar soft sensors capazes de prever a pressão no fundo do poço. Eles compararam o desempenho do EAND com outros algoritmos de otimização em problemas de otimização simulados para validar sua eficiência. O EAND mostrou convergência rápida e estabilidade. Além disso, os autores usaram o EAND para otimizar os soft sensors propostos para estimar a pressão no fundo do poço em cinco poços reais. O EAND foi comparado com outros métodos de projeto de soft sensors e mostrou a capacidade de encontrar automaticamente os melhores modelos de entrada para os soft sensors, tornando-o uma ferramenta valiosa, dada a complexidade das características dos poços de petróleo. Eles também testaram três modelos diferentes para construir modelos de conjunto, e o Random Forest obteve os melhores resultados em termos de erro percentual médio absoluto (MAPE).
Os resultados obtidos mostraram que os soft sensors desenvolvidos com o EAND têm uma precisão muito alta na previsão da pressão no fundo do poço, com erros muito baixos (de 0,1453% a 0,788%). Isso demonstra a eficácia tanto do algoritmo quanto dos modelos de soft sensor propostos.
O trabalho dos autores é significativo, pois propõe uma solução inovadora para um problema crítico na indústria de petróleo e gás. Ao desenvolver soft sensors eficazes e introduzir um algoritmo adaptativo como o EAND, eles possibilitaram a criação de sistemas de monitoramento mais robustos e confiáveis para a pressão no fundo do poço. Isso pode levar a uma produção mais eficiente de petróleo e uma melhor gestão dos recursos em poços submarinos, reduzindo a necessidade de substituição frequente de sensores físicos.