Módulo 4 – Python Avançado

Bootcamp: Desenvolvedor Python

Túlio Philipe Ferreira e Vieira

2020

Python Avançado

Bootcamp: Desenvolvedor Python

Túlio Philipe Ferreira e Vieira

Sumário

Capítulo 1. Aprendizado de Máquina com Python

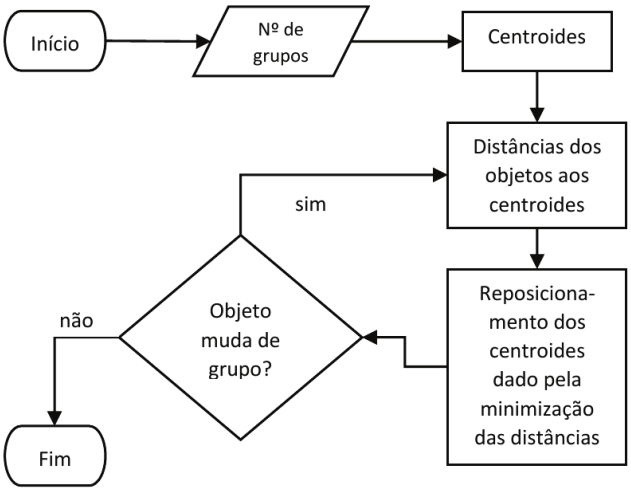
Os algoritmos de aprendizado de máquina são ferramentas essenciais para a identificação de padrões em qualquer *dataset*. É através desses algoritmos que pode ser realizada uma análise de dados mais precisa e gerar o conhecimento necessário para otimizar processos e aumentar o valor de negócio para as organizações.

Neste curso, serão abordados os algoritmos supervisionados K-Nearest Neighbors (KNN), árvore de decisão, SVM (Support Vector Machine) e Redes Neurais Artificiais (RNA), além do algoritmo não supervisionado K-means. Algoritmos supervisionados são aqueles que utilizam um banco de dados que possui valores de entrada e saída. Esses dados são utilizados para realizar o treinamento do modelo. A partir do treinamento é possível que esses algoritmos consigam capturar as características do conjunto de dados e, quando novos dados forem apresentados, é possível realizar a previsão dos valores de saída. Já os algoritmos não supervisionados não possuem a fase de treinamento, pois o banco de dados utilizado não apresenta o mapeamento entre dados de entradas e saídas. Esses algoritmos trabalham com um *dataset* sem conhecer a saída provável. Normalmente, são utilizados em etapas anteriores à aplicação dos algoritmos supervisionados.

K-means

O algoritmo K-means, como apresentado na seção anterior, é utilizado para encontrar grupos em um conjunto de dados. Para isso, o usuário deve definir o número de grupos que devem ser encontrados no *dataset*. O número de grupos é definido através da quantidade de centroides escolhidos. Os centroides correspondem ao elemento responsável por realizar o agrupamento dos dados. Por exemplo, se for definido que existem dois centroides a serem encontrados no conjunto de dados, ao final da execução do K-means, devem existir dois grupos diferentes de dados.

Figura 1 – Fluxograma para a determinaçao dos agrupamentos pelo K-means.



Fonte: Coelho et al. (2013)

Inicialmente são escolhidos, aleatoriamente, pontos que representam os centroides no conjunto de dados. Após esse processo, são calculadas as distâncias desses centroides a cada um dos dados do *dataset*. Essas distâncias são utilizadas para definir os grupos iniciais para os dados. A próxima etapa consiste em reposicionar os centroides para cada um dos grupos encontrados. Depois de reposicionados, ocorre o cálculo das distâncias entre esses novos centroides e cada um dos dados. Essas distâncias são utilizadas para encontrar, novamente, os elementos que compõem os grupos. Esse processo é repetido até que os novos centroides não sofram um reposicionamento.

Para ilustrar esse processo, vamos utilizar um banco de dados contendo duas variáveis, X e Y, que foram criadas aleatoriamente. A Figura 2 apresenta o conjunto de dados desenvolvido.

Figura 2 – Conjunto de dados criado para o K-means.

```
#cria dados aleatórios
dados = {'x': [25,34,22,27,33,33,31,22,35,34,67,54,57,43,50,57,59,52,65,47,49,48,35,33,44,45,38,43,51,46],
         'y': [79,51,53,78,59,74,73,57,69,75,51,32,40,47,53,36,35,58,59,50,25,20,14,12,20,5,29,27,8,7]
        }
```

```
#cria o dataframe
df = DataFrame(dados,columns=['x','y'])
print (df.head())
```

```
    x   y
0  25  79
1  34  51
2  22  53
3  27  78
4  33  59
```

A partir desses dados, é utilizado o algoritmo k-means para identificar dois diferentes agrupamentos nesse conjunto. A Figura 3 apresenta o procedimento para criar o objeto desse algoritmo e aplicá-lo.

Figura 3 – Construção do algoritmo k-means através do sklearn.

```
#adiciona as bibliotecas para construir o algoritmo
from sklearn.cluster import KMeans
```

```
kmeans = KMeans(n_clusters=2)    # cria o objeto de para o algoritmo k-means para encontrar 2 clusters
kmeans.fit(df)  #aplica o algoritmo
centroides = kmeans.cluster_centers_   #encontra as coordenadas dos centroids
print(centroides)
```

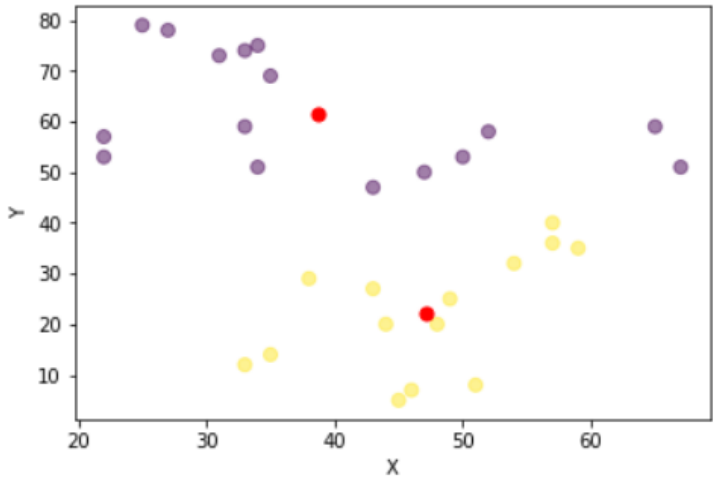
```
[[38.75       61.625     ]
 [47.07142857 22.14285714]]
```

A Figura 4 mostra o resultado final após a aplicação do agrupamento realizado pelo K-means.

Figura 4 – Resultado do agrupamento realizado pelo K-means.

```
#realiza o plot do gráfico da saída
plt.scatter(df['x'], df['y'], c= kmeans.labels_.astype(float), s=50, alpha=0.5)
plt.scatter(centroides[:, 0], centroides[:, 1], c='red', s=50)
plt.xlabel("X")
plt.ylabel("Y")
```

```
Text(0, 0.5, 'Y')
```



KNN

O KNN é um algoritmo supervisionado utilizado, normalmente, para realiza a classificação de instancias em um conjunto de dados. O funcionamento desse algoritmo consiste em encontrar a distância entre os novos pontos (novas instâncias de elementos) adicionados e todo o conjunto de dados. A partir dessa distância é determinada a classificação desse novo elemento. A classificação é realizada através da associação dos K vizinhos mais próximos, encontrados para esse novo ponto. Os vizinhos mais próximos correspondem àqueles que possuem a menor distância para esse novo ponto. A Figura 5 mostra o cálculo das distâncias entre cada um dos dados de um *dataset*.

Figura 5 – Cálculo das distâncias em um conjunto de dados.

#	a1	a2	Classe
1	0.5	1	2
2	2.9	1.9	2
3	1.2	3.1	2
4	0.8	4.7	2
5	2.7	5.4	2
6	8.1	4.7	1
7	8.3	6.6	1
8	6.3	6.7	1
9	8	9.1	1
10	5.4	8.4	1
11	5	7	?

$$d_E(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

$$d_E(\#1,\#11) = \sqrt{(0.5-5)^2 + (1-7)^2}$$

$$d_E(\#1,\#11) = \sqrt{(-4.5)^2 + (-6)^2}$$

$$d_E(\#1,\#11) = \sqrt{20.25 + 36}$$

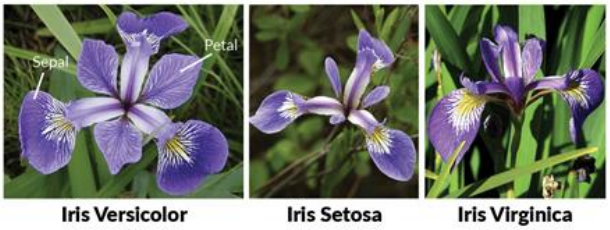
$$d_E(\#1,\#11) = \sqrt{56.25}$$

$$d_E(\#1,\#11) = 7.5$$

Fonte: Coelho et al. (2013).

Para a aplicação desse e dos próximos algoritmos, será utilizado o conjunto de dados conhecido como Iris. Esse *dataset* contém o comprimento e largura das sépalas e pétalas de um conjunto de espécies de Iris (Setosa, Virginica e Versicolor). A Figura 6 exemplifica esse conjunto de dados.

Figura 6 – Exemplo do conjunto de dados Iris.




```
#Converte o banco de dados iris para o dataframe
df_iris = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                       columns= iris['feature_names'] + ['target'])
```

```
print(df_iris.head())
```

```
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  target
0                5.1               3.5  ...               0.2     0.0
1                4.9               3.0  ...               0.2     0.0
2                4.7               3.2  ...               0.2     0.0
3                4.6               3.1  ...               0.2     0.0
4                5.0               3.6  ...               0.2     0.0

[5 rows x 5 columns]
```

Como esse é um algoritmo supervisionado, é necessário dividir o conjunto de dados em instâncias para treinamento e teste. A Figura 7 apresenta esse procedimento.

Figura 7 – Divisão do banco de dados entre treinamento e teste.

```
#transforma os dados em array
X = df_iris.iloc[:, :-1].values  #dados de entrada
y = df_iris.iloc[:, 4].values  # saídas ou target
```

```
#realiza a divisão dos dados entre treinamento e teste
from sklearn.model_selection import train_test_split   # função que realiza a divisão do dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)# divide 20% para teste
```

É necessário realizar um tratamento dos dados de entrada, pois os algoritmos de aprendizado de máquina, normalmente, possuem um comportamento mais natural quando os dados são tratados. Para isso, é realizada a normalização dos dados. Esse procedimento é apresentado na Figura 8.

Figura 8 – Procedimento de normalização dos dados.

```
# realiza o processo de normalização dos dados
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()  #objeto que normaliza os dados
scaler.fit(X_train)  #realiza a normalização dos dados

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Após o processo de normalização, é possível realizar a construção e treinamento do modelo. A Figura 9 apresenta esse processo utilizando a biblioteca ***sklearn***. Para esse exemplo, foram escolhidos cinco vizinhos para realizar a classificação dos dados de teste do modelo.

Figura 9 – Procedimento de construção e treinamento do modelo.

```
#treina o modelo
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5) #utiliza a construção por meio de 5 vizinhos
classifier.fit(X_train, y_train) # aplica a classificação
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

Após o processo de treinamento, pode ser aplicado o processo de previsão, ou seja, encontrar a classificação de um conjunto de dados que não foi utilizado para o treinamento do modelo. Essa previsão é realizada da forma como mostrada na Figura 10.

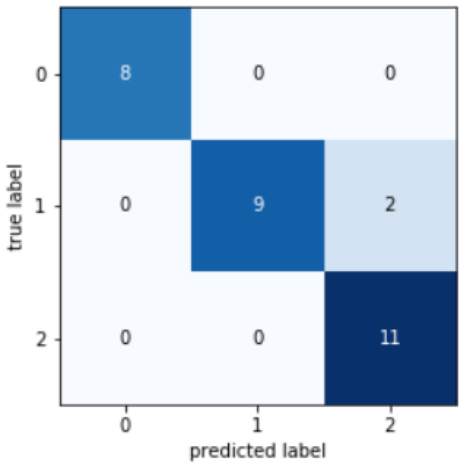
Figura 10 – Previsão realizada pelo modelo KNN treinado.

```
#realiza a previsão
y_pred = classifier.predict(X_test)
```

O resultado da previsão pode ser utilizado para medir o desempenho desse algoritmo. A métrica que será utilizada para comparar o resultado obtido pelo algoritmo de classificação é conhecida como matriz de confusão. A matriz de confusão proporciona uma excelente métrica de desempenho para processos de classificação, pois é possível visualizar os erros e acertos do processo para cada uma das classes e instâncias do modelo. Assim, é possível ter acesso às taxas de classificação de cada uma das diferentes classes. A Figura 11 apresenta a matriz de confusão para o KNN.

Figura 11 – Matriz de confusão para o algoritmo KNN aplicado ao dataset Iris.

```
#realiza o plot da matriz de confusão
matriz_confusao = confusion_matrix(y_test, y_pred)
from mlxtend.plotting import plot_confusion_matrix

fig, ax = plot_confusion_matrix(conf_mat=matriz_confusao)
plt.show()
```

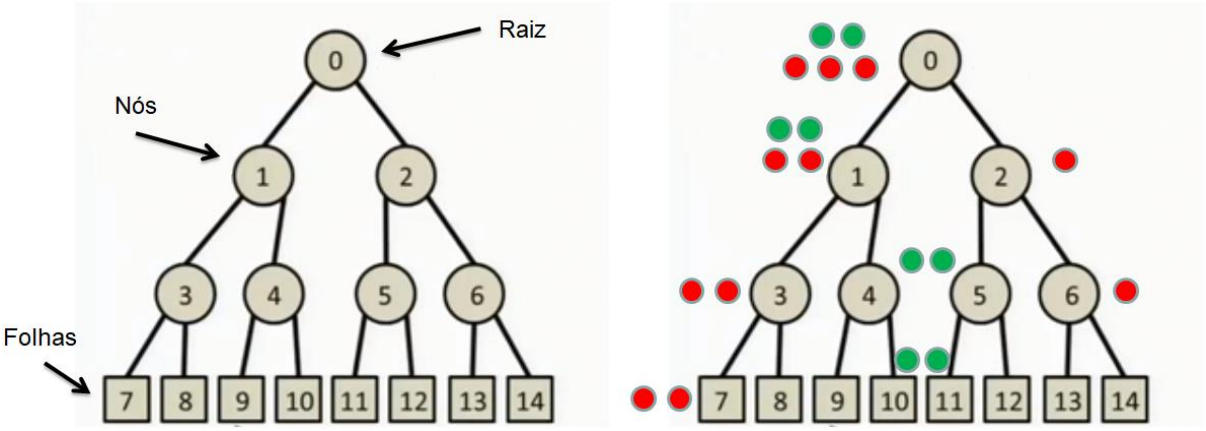


Pela matriz de confusão presente na Figura 11, é possível perceber, por exemplo, que o KNN classificou corretamente 11 instâncias como pertencente a classe 2 e 2 instâncias foram, erroneamente, classificadas como pertencentes à classe 2, quando deveriam ter sido classificadas como pertencentes à classe 1. Portanto, a matriz de confusão apresenta-se como uma excelente métrica para comparar a classificação de modelos.

Árvore de decisão

As árvores de decisão também são algoritmos supervisionados. Elas podem ser utilizadas tanto para resolver problemas de classificação quanto para problemas de regressão. As árvores de decisão são constituídas por um nó raiz, nós intermediários e por folhas. O nó raiz é o primeiro nó da árvore de decisão. É por esse nó que os dados são apresentados para o problema. A partir desse nó as instâncias são repassadas para os nós intermediários até que cheguem às folhas da árvore. A Figura 12 apresenta um exemplo de árvore.

Figura 12 – Árvore de decisão.



Os nós são os elementos responsáveis por realizar a análise dos dados, ou seja, é nos nós que ocorre o processo de decisão e separação do conjunto de dados. À medida que os dados descem pela árvore de decisão, vai ocorrendo uma maior separação dos dados, até que ao final do processo, nas folhas, o conjunto de dados pode ser separado.

Para o exemplo de utilização da árvore de decisão, também será empregado o conjunto de dados Iris. A Figura 13 apresenta como é criado e treinado o modelo de classificação utilizando árvores de decisão.

Figura 13 – Criação do algoritmo de árvore de decisão através do sklearn.

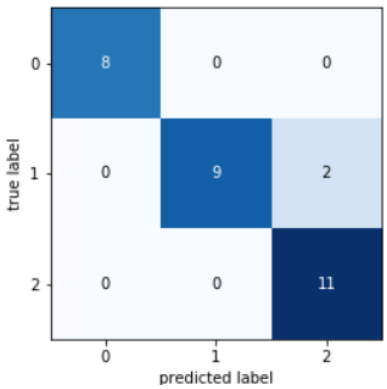
```
from sklearn.tree import DecisionTreeClassifier # importa o classificador árvore de decisão
from sklearn import metrics #importa as métricas para avaliação
```

```
# Cria o objeto de classificação através do
clf = DecisionTreeClassifier()

# Realiza o treinamento do classificador
clf = clf.fit(X_train,y_train)

#Realiza a previsão de classificaçao
y_pred = clf.predict(X_test)
```

A Figura 14 mostra a matriz de confusão para a classificação do banco de dados Iris, utilizando o algoritmo árvore de decisão.

Figura 14 – Matriz de confusão para o algoritmo árvore de decisão.

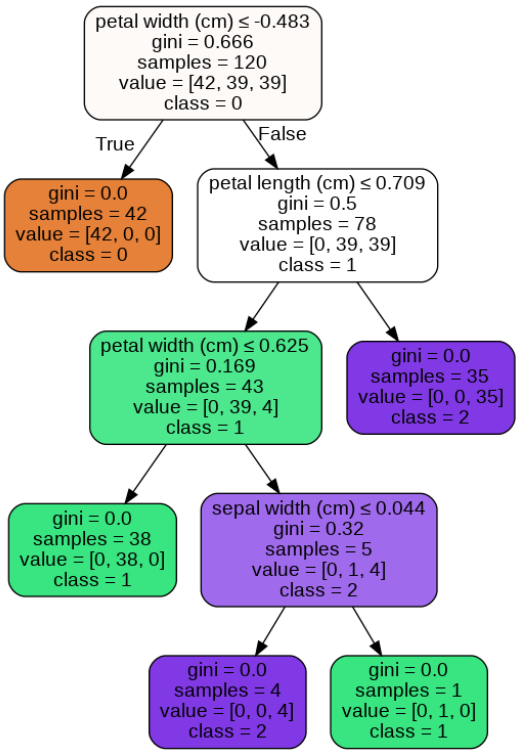
```
#Avaliando o modelo

#realiza o plot da matriz de confusão
matriz_confusao = confusion_matrix(y_test, y_pred)
from mlxtend.plotting import plot_confusion_matrix

fig, ax = plot_confusion_matrix(conf_mat=matriz_confusao)
plt.show()
```



Pela Figura 14 é possível ver que a classificação através da árvore de decisão apresentou um desempenho similar ao encontrado pelo algoritmo KNN. A grande vantagem em se utilizar a árvore de decisão reside no fato de ser possível compreender todo o processo de separação realizado. A Figura 15 apresenta todo o

processo de decisão realizado pelo algoritmo, até encontrar o resultado expresso na Figura 14.

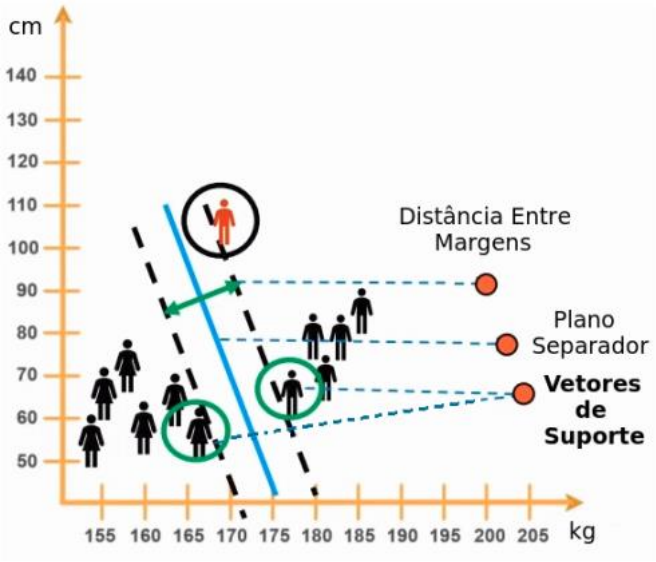
Figura 15 – Processo de decisão realizado pela árvore de decisão.



SVM

Os *Support Vector Machines* (SVM) também pertencem à classe de algoritmos supervisionados. Esse são utilizados, normalmente, para realizar a classificação de um conjunto de elementos. O princípio de funcionamento desse algoritmo consiste em encontrar o hiperplano que garante a maior separação entre um conjunto de dados. Esse hiperplano é encontrado através dos vetores de suporte que, por meio do processo de otimização, encontram o hiperplano que garante a maior distância de separação entre os dados. A Figura 16 mostra, graficamente, o hiperplano (para duas dimensões é uma reta) de separação.

Figura 16 – Separação realizada pelo SVM.



Para a aplicação do SVM, também é utilizado o conjunto de dados Iris. A Figura 17 mostra como deve ser construído, treinado e a previsão de classificação do algoritmo SVM através da biblioteca ***sklearn***.

Figura 17 – Construção, treinamento e previsão do SVM utilizando o Sklearn.

```
#cria o objeto SVM
clf = SVC(gamma='auto')  #escolhe o kernel linear
```

```
#realiza a classificação via SVM
clf.fit(X_train,y_train)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

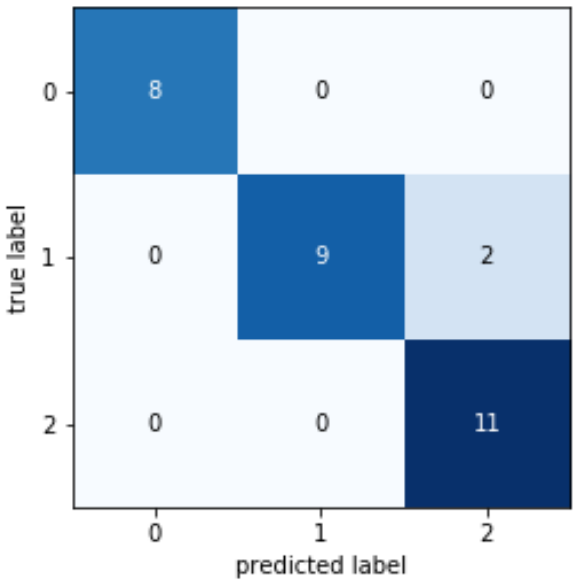
```
#Realiza a previsão de classificaçao
y_pred = clf.predict(X_test)
```

A Figura 18 apresenta a matriz de confusão para a aplicação do algoritmo SVM. Por meio dessa Figura, é possível verificar que os resultados também foram similares aos demais algoritmos de classificação. O SVM, em conjunto de dados mais

complexos, normalmente possui melhor resultado que os outros algoritmos demonstrados nas seções anteriores.

Figura 18 – Matriz de confusão para o SVM.

```
#Avaliando o modelo

#realiza o plot da matriz de confusão
matriz_confusao = confusion_matrix(y_test, y_pred)
from mlxtend.plotting import plot_confusion_matrix

fig, ax = plot_confusion_matrix(conf_mat=matriz_confusao)
plt.show()
```



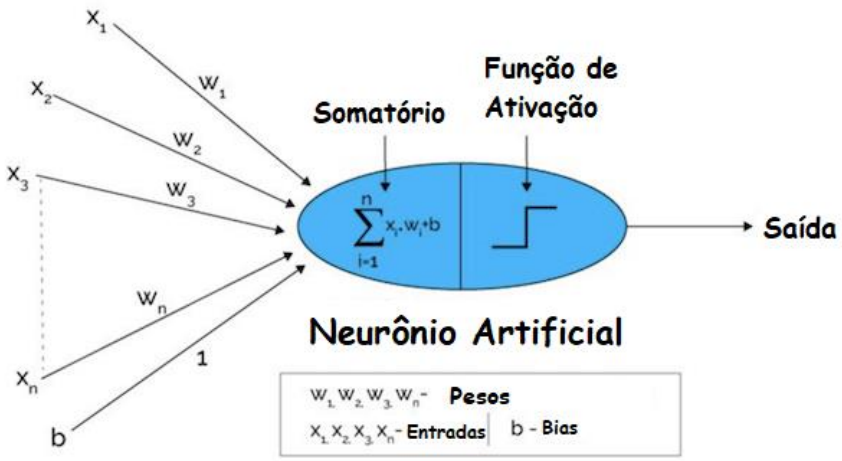
Redes Neurais Artificiais e Deep Learning

Redes neurais artificiais são sistemas de computação com nós interconectados, que funcionam “como os neurônios do cérebro humano”. Usando algoritmos, elas podem reconhecer padrões escondidos e correlações em dados brutos, agrupá-los e classificá-los, e — com o tempo — aprender e melhorar continuamente.

A unidade fundamental de uma rede neural artificial é o *perceptron*. A Figura 19 apresenta um exemplo desse elemento. Os *perceptrons* são os responsáveis por

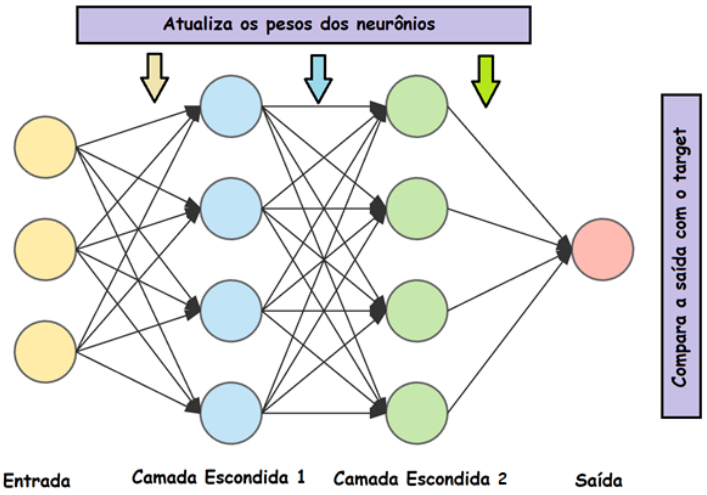
armazenar as características de um conjunto de dados. Eles realizam essa função através do ajuste dos pesos existentes em cada conexão. Esses ajustes ocorrem durante o processo de treinamento da rede. Esse treinamento, normalmente, ocorre por meio de um processo conhecido como *backpropagation*. O *backpropagation* consiste em duas etapas. Na primeira etapa, os dados são apresentados à rede. Desse modo, ela realiza o processo do cálculo de todos os pesos e encontra uma saída estimada. Essa etapa é conhecida como *feedforwad*, ou seja, o processamento ocorre no sentido da esquerda para direita. Na segunda etapa, é calculado o erro entre o valor encontrado no processo de *feedforward* e o valor real. Esse erro é propagado no sentido inverso da rede. À medida que esse erro é propagado, é realizado um processo de otimização para que os erros sejam minimizados e os pesos sejam atualizados. Devido ao sentido inverso de propagação do erro, essa etapa recebe o nome de *backpropagation*.

Figura 19 – Exemplo de um perceptron.



Quando vários *perceptrons* estão conectados, temos a construção de uma rede neural artificial. A Figura 20 apresenta um exemplo de rede *Perceptron Multi-camadas* (MLP). Nessa rede, existem as camadas de entrada, intermediárias e saída. É no processo de treinamento que a rede neural atualiza os pesos e consegue "aprender" as características de um conjunto de dados.

Figura 20 – Exemplo de rede MLP.



Quando uma rede neural possui um grande número de camadas escondidas, temos a construção do chamado *Deep Learning*. A principal vantagem do Deep Learning é que existe um maior número de neurônios e, consequentemente, mais pesos para serem ajustados. Assim, é possível que a rede consiga "aprender" características mais complexas sobre o conjunto de dados. Entretanto, com isso, também a complexidade computacional para o treinamento do modelo será maior.

Para a construção, a aplicação de rede neural artificial será utilizada uma rede MLP. Essa rede será construída utilizando o Sklearn. Ao final desta apostila, no anexo B, existe um tutorial para o uso da API Keras, que permite a construção de modelos utilizando o *Deep Learning*.

O primeiro passo para a construção do modelo consiste em definir a rede, indicando as entradas, a quantidade de camadas escondidas e neurônios em cada uma das camadas. A Figura 21 apresenta a forma como a MLP é criado no ***sklearn***.

Figura 21 – Construção da rede MLP.

```
#definição da biblioteca
from sklearn.neural_network import MLPClassifier
```

```
#define a configuração da rede
clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5, 5), random_state=1) #rede com 2 camadas escondidas com 5 neurônios cada
```

Como pode ser visto, foi construída uma rede com duas camadas escondidas e cinco neurônios em cada uma das camadas. Após essa etapa é necessário realizar o procedimento de treinamento do modelo. A Figura 22 mostra como ele é realizado através do Sklearn.

Figura 22 – Procedimento para realizar o treinamento da rede MLP.

```
#realiza o fit do modelo
clf.fit(X_train,y_train)
```

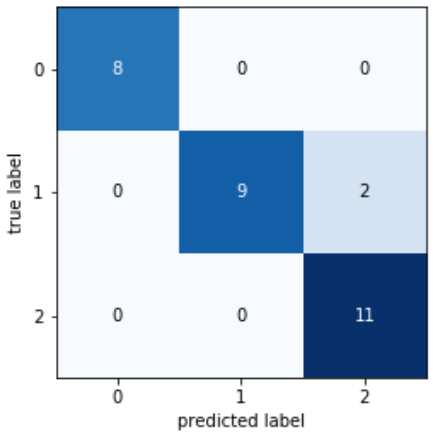
```
MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto', beta_1=0.9,
              beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes=(5, 5), learning_rate='constant',
              learning_rate_init=0.001, max_iter=200, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
              random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
```

Após o treinamento do modelo, podem ser realizadas as previsões sobre a classificação. A previsão de classificação e a matriz de construção podem ser visualizadas através da Figura 23.

Figura 23 – Previsão e matriz de confusão para a rede MLP.

```
#realiza a previsão
y_pred=clf.predict(X_test)
```

```
#Avaliando o modelo

#realiza o plot da matriz de confusão
matriz_confusao = confusion_matrix(y_test, y_pred)
from mlxtend.plotting import plot_confusion_matrix

fig, ax = plot_confusion_matrix(conf_mat=matriz_confusao)
plt.show()
```



Como pode ser visto, o resultado obtido é similar a todos os outros encontrados pelos demais algoritmos. As redes neurais artificiais são utilizadas em sistemas mais complexos, em que as características do conjunto de dados necessitam de uma análise mais apurada.

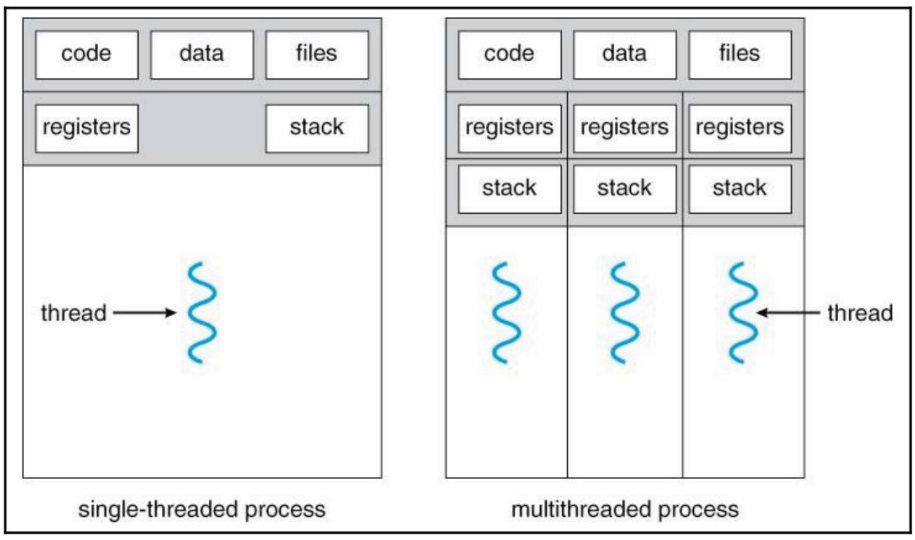
Capítulo 2. Concorrência

Concorrência é uma das mais importantes formas para se aumentar a performance dos algoritmos. Entretanto, quando iniciamos a construção de algoritmos e sistemas que funcionam de maneira concorrente, normalmente, ocorre um aumento da complexidade de construção e manutenção desses sistemas. Desse modo, para construir sistemas com alta performance e que funcione corretamente, é necessário compreender os conceitos que governam o desenvolvimento de sistemas concorrentes.

Processos

Os processos são os programas em execução. Desse modo, para um mesmo programa é possível ter vários processos em execução. Por exemplo, é possível executar o programa *Google Chrome* mais de uma vez. Quando esse programa (Google Chrome) é executado duas vezes, por exemplo, temos que são executados dois processos. Os processos são compostos de threads que executam as ações necessárias para a construção das atividades em um programa. A Figura 24 apresenta um exemplo de processo com apenas uma thread e outro composto de várias threads.

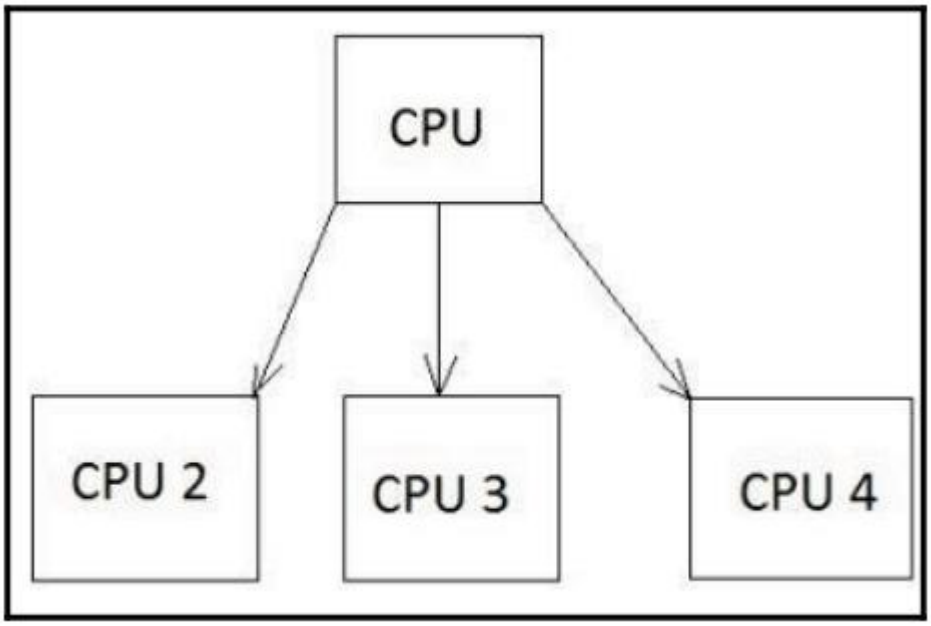
Figura 24 – Processos single-thread e multithread.



Fonte: Forbes (2017).

O multiprocessamento ocorre quando existem mais de uma unidade de processamento presente no sistema e, desse modo, é possível existir vários processos sendo executados em diferentes núcleos. A Figura 25 apresenta um modelo em que um núcleo de processamento delega processos a outros núcleos.

Figura 25 – Multiprocessamento.

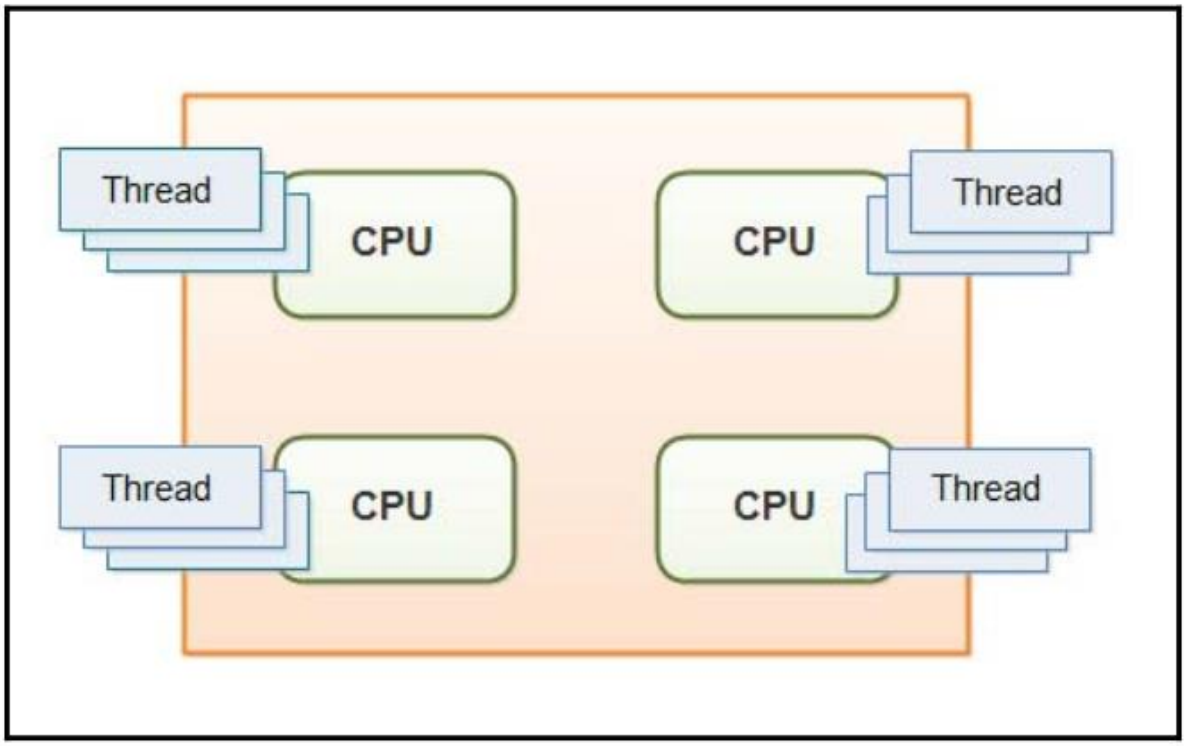


Fonte: Forbes (2017).

Threads

Uma Thread (linha de execução) pode ser vista como um conjunto ordenado de instruções que pode ser executado pelo processador. As threads correspondem à menor unidade de um processo (quando estamos realizando a extração de um arquivo que estava em um formato zip, por exemplo, e observamos a barra de “status” mostrando o progresso da execução, temos ali o exemplo de threads sendo executadas). As threads compartilham recursos em um mesmo processo. A Figura 26 apresenta um conjunto de threads sendo executadas em diferentes unidades de processamento.

Figura 26 – Threads em execução em diferentes CPU.



Fonte: Forbes (2017).

O multithreading ocorre quando um processador possui a capacidade de executar um conjunto de instruções (ou threads) de maneira simultânea. Em um processador de um núcleo, por exemplo, as tarefas não são executadas de maneira paralela, mas a velocidade em que as execuções são “chaveadas” entre as diferentes threads transmitem a sensação de execução em paralelo.

Ciclo de vida de uma thread

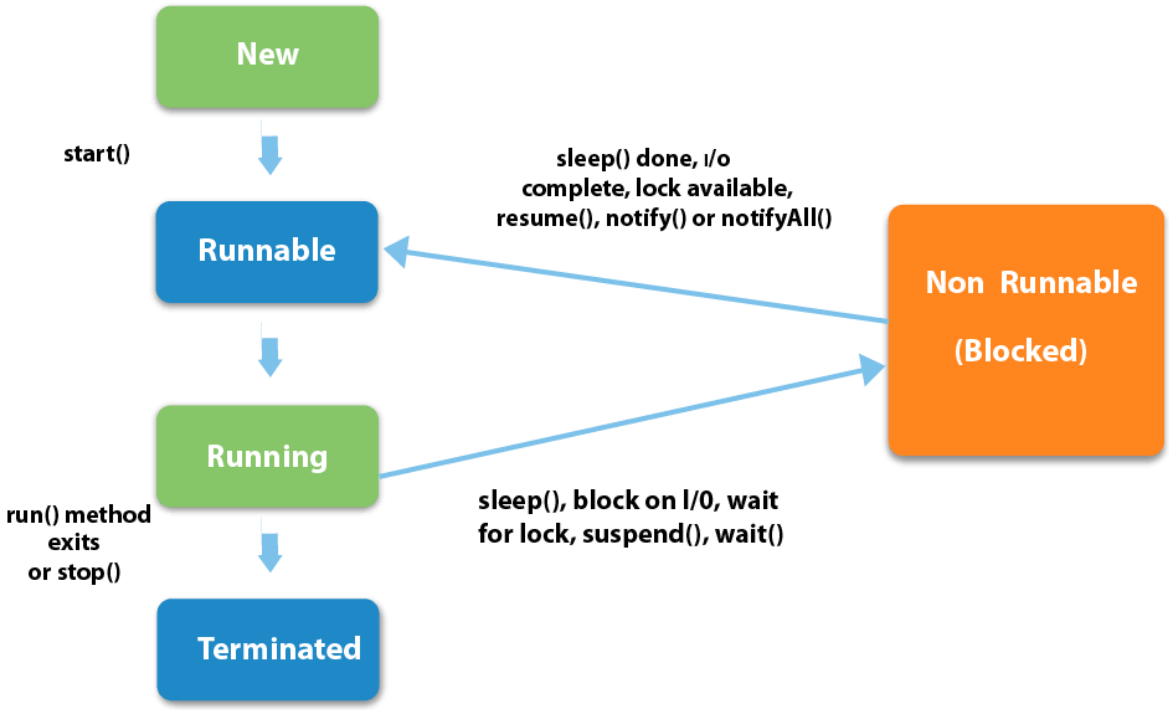
O ciclo de vida de uma thread corresponde aos estados que uma thread pode estar. Esses estados são definidos desde o momento da construção até a finalização dessa thread. O construtor da classe Thread() pode ser visto abaixo:

```
#construtor da classe thread
def __init__ (self, group=None, target=None, name=None,
                                    args=(),kwargs=None,verbose=None)
```

- **group**: parâmetro especial reservado.
- **target**: esse é o objeto que será chamado pelo método ***run***(). Se não for passado um nesse objeto, nada será invocado.
- **name**: esse é o nome da thread.
- **args**: é uma tupla de argumentos para o *target*.
- **kwargs**: corresponde a um dicionário para invocar o construtor da classe base.

A Figura 27, apresenta os estados possíveis para uma thread. Pela Figura 27, é possível ver que existem 5 estados possíveis:

Figura 27 – Ciclo de vida de uma thread.



Fonte: Forbes (2017).

- **new**: esse é o estado em que a thread foi criada, mas ainda não foi iniciada (start).

- **runnable**: esse é o estado em que a thread fica “esperando” para ser executada. Neste estado, a thread já possui alocado todos os recursos necessários para a execução.
- **running**: nesse estado, a thread executa a tarefa designada quando o momento de execução é liberado.
- **not-running**: nesse estado, a thread é pausada por algum motivo, por exemplo, espera a execução de outra thread ou pode ficar bloqueada esperando alguma resposta.
- **terminated**: a thread pode chegar a esse estado de duas formas: a primeira devido a finalização inesperada da execução, e a segunda quando finaliza a tarefa designada.

Herança em threads

Em problemas de maior complexidade, pode ser interessante passar para uma classe os métodos já implementados pela classe thread. Nesses casos, a classe filha de uma classe thread, possibilita trabalhar de maneira mais flexível com as mudanças de execução realizadas por um processo. Utilizar a herança através da classe Thread pode ser interessante em casos em que apenas a definição de uma função pode não ser suficiente para realizar todas as tarefas necessárias através de uma thread. O código presente na Figura 28, mostra como criar uma classe filha da classe Thread.

Figura 28 – Herança em threads.

```
from threading import Thread

#define a classe como filha da classe Thread
class MinhaClasseThread(Thread):
  def __init__(self):
    print("Olá, construtor thread!!")
    Thread.__init__(self)

  #define a função run() que é chamada quando thread.start()
  def run(self):
    print("\nThread em execução.")

#instanciando um objeto da classe criada
minhaThread=MinhaClasseThread()
print("Objeto criado")
minhaThread.start()
print("Thread inicalizada")
minhaThread.join()
print("Thread finalizada")
```

Para uma classe herdar as características da classe Thread(), além de passar como parâmetro a própria classe Thread(), é necessário que seja passada a construção da classe Thread() como construtor para classe filha (***Thread.__init__(self***)) e deve ser implementado o método ***run***().

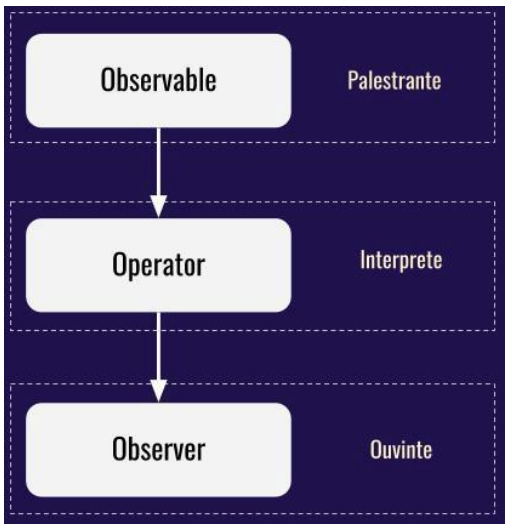
Capítulo 3. Introdução à Programação Reativa

Quando existe a necessidade de desenvolver sistemas que, após receber um conjunto de dados, é disparado um gatilho que executa uma determinada tarefa, existe o conceito de programação reativa. Desse modo, programação reativa pode ser vista como um modelo de programação orientada a evento, em que os eventos monitorados e recebidos corresponde a um conjunto de dados. Para o entendimento desse modelo de programação, é necessário definir três conceitos fundamentais: *observable*, operações e *observer*. Cada um desses conceitos é definido e explicado neste capítulo.

Observable

A Figura 29 apresenta um fluxograma que exemplifica a programação reativa. Nessa Figura é possível identificar o Observable. O Observable corresponde ao elemento que é responsável por enviar os dados para o Observer. Esses dados enviados pelo Observable corresponde a um conjunto de dados obtidos através de um processamento em “streaming”. O processamento em “streaming” corresponde a um conjunto de dados gerados de maneira contínua.

Figura 29 – Exemplo de entidades em programação reativa.



Fonte: Forbes (2017).

Um Observable tem a função de coletar esses dados e transmitir para o Observer que se inscreveu para receber esses dados. Nesse processo de envio de dados, podem ocorrer algumas operações sobre esse conjunto de dados. Essas operações são realizadas através dos operadores (Operator).

Operator

Os Operator (Operadores), correspondem ao conjunto operações que podem ser executadas sobre os dados enviados pelos Observable. Quando um Observable envia um determinado dado para o Observer, algumas operação, como o mapeamento e filtragem, podem ser aplicadas sobre esses dados, antes que eles cheguem aos Observer. Essas operações garantem uma transformação nos dados enviados, assim, o Observer que se inscreveu para receber os dados não receberá os dados “brutos” enviados pelo Observable, mas sim os dados modificados. De posse desses dados modificados, o Observer pode definir a ação a ser tomada.

Observer

Os Observer são os elementos responsáveis por receber os dados enviados pelos Observable. São também os Observer que definem as operações ou funções a serem executadas sobre os dados recebidos. As três principais ações geradas pelos Observer são realizadas pelos parâmetros:

- **on_next**: define a função a ser chamada sempre que um novo dado é recebido pelo observer.
- **on_completed**: define a função a ser chamada sempre que um observable finaliza a tarefa definida.
- **on_error**: define a função a ser executada sempre que um erro ocorrer durante o processamento dos dados enviados.

Essas funções podem ser definidas dentro de uma classe filha de um “observer”, ou através das funções ***lambda***. As funções ***lambda*** são as funções anônimas do Python. Essas funções possuem esse nome pois não existe a necessidade de uma declaração através dos ***def nome_da_funcao( ):***. Assim, ao utilizar a palavra ***lambda***, as ações a serem realizadas através de uma função podem ser definidas em apenas uma linha de código. Os parâmetros da função são passados antes dos “ **:** ” e a execução da ação é realizada após os “ **:** ”. A Figura 30 apresenta a construção de uma modelo de programação reativa, em que as ações realizadas sobre os dados recebidos pelo Observer são implementadas utilizando as funções ***lambda***.

Figura 30 – Exemplo construção de programação reativa utilizando lambda.

```
#Observable factories
from rx import of

#cria o observable
source = of("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

source.subscribe(
on_next = lambda i: print("Recebido {0}".format(i)),
on_error = lambda e: print("Erro identificado: {0}".format(e)),
on_completed = lambda: print("Finalizado!"),
)
```

```
Recebido Alpha
Recebido Beta
Recebido Gamma
Recebido Delta
Recebido Epsilon
Finalizado!
```

Capítulo 4. Introdução ao Pygame

O desenvolvimento de jogos corresponde a uma das tarefas em programação mais recompensadoras, pois é possível criar uma história, desenvolver personagens, construir um enredo e “colocar tudo para funcionar”. Desse modo, é possível tornar realidade várias histórias, e uma das formas mais simples de iniciar no mundo do desenvolvimento de jogos é através da biblioteca Pygame. Isso porque esse módulo agrupa todas as características e funcionalidades da linguagem de programação Python na construção de jogos interativos. Neste capítulo, é apresentada uma introdução à construção de jogos utilizando o Pygame.

Construindo um pequeno jogo através do Pygame

Ao iniciar a construção de um jogo utilizando o Pygame, é necessário importar os módulos desejados. A Figura 31 mostra como importar as bibliotecas utilizadas para a construção deste código.

Figura 31 – Importando os módulos.

```
1  # coding: iso-8859-1 -*-
2
3  #adaptado de https://redhuli.io/game-development/introduction-to-pygame/
4  # importando os módulos necessários
5  import pygame, sys
6  from pygame import *
7
8  #inicalizando o múdlo pygame
9  pygame.init()
10
11 # Definindo as constantes
12 window_width = 500   #comprimento da tela
13 window_height = 400   #altura da tela
14 FPS = 30 #definindo a taxa de atualização da tela
```

Fonte: adaptado de RedHuli (2018).

Nas linhas 5 e 6 são importados os módulos Pygame e sys. Esses dois módulos são utilizados para a construção deste jogo. Na linha 9, é realizada a

inicialização do módulo Pygame. Isso é necessário pois as definições implementadas pelo Pygame precisam ser inicializadas.

Nas linhas 12 e 13 são definidas as constantes utilizadas, respectivamente, para indicar o comprimento e altura da tela a ser criada. Na linha 14, é definida a taxa de atualização da tela. A taxa de atualização representa a velocidade (quadros por segundo) com que ocorre a atualização da tela. A Figura 32 apresenta as definições para as cores utilizadas no código.

Figura 32 – Importando os módulos.

```
16  black_color = (0,0,0)  #definição para a cor preta
17  white_color = (255,255,255)  #definição para a cor branca
```

Fonte: adaptado de RedHuli (2018).

Para a construção deste tutorial de introdução ao Pygame, será criado um personagem. Esse personagem será representado apenas como um quadrado (Sprite). Para isso, é criada uma classe que será filha da classe Sprite. Essa classe “Player” contém todos os movimentos possíveis a serem realizados sobre o personagem do jogo. A Figura 33 apresenta as definições iniciais dessa classe.

Figura 33 – Definindo o construtor da classe jogador.

```
19  #classe que representa um jogador
20  class Player(pygame.sprite.Sprite):
21
22      def __init__(self, width, height):
23          #chama o construtor para a classe Sprite
24          super().__init__()
25
26          # Cria o jogador (quadrado) com os valores passados
27          self.image = pygame.Surface([width,height])
28          self.image.fill(white_color)
29
30          # Desenha o retângulo na tela
31          self.rect = self.image.get_rect()
32
33          # Define as variáveis do jogador
34          self.changex = 0
35          self.changey = 0
```

Fonte: adaptado de RedHuli (2018).

No construtor da classe “Player”, é utilizado o construtor da classe pai “Sprite”. Essa classe recebe o comprimento e a largura do retângulo que será utilizado como o personagem desse código. Para “desenhar” esse personagem, é necessário definir alguns parâmetros como o objeto “Surface”, que é responsável por definir a “tela” do jogo e as modificações das posições.

Como a classe “Player” deve conter os movimentos possíveis a serem realizados pelo jogador, é importante implementar algumas funções que devem capturar as teclas pressionadas e atualizar os quadros na tela do computador. A Figura 34 mostra algumas das funções desenvolvidas.

Figura 34 – Alguns movimentos possíveis.

```
37      def move_left(self, move_x):
38          #movimenta para a esquerda
39          self.changex -= move_x
40
41      def move_right(self, move_x):
42          #movimenta para a direita
43          self.changex += move_x
44
45      def gravity(self):
46          #define a "gravidade"
47          if self.changey == 0:
48              self.changey = 0
49          else:
50              self.changey += .40
51
52          if self.rect.bottom >= 360 and self.changey >= 0:
53              self.changey = 0
54              self.rect.y = 330
```

Fonte: adaptado de RedHuli (2018).

As funções move_left e move_right são responsáveis por realizar a atualização das posições do “personagem” na tela. A função gravity, também é utilizada para atualizar a posição do “personagem”, entretanto, essa função será

chamada quando o jogador liberar a tecla, pois esses movimentos são definidos para simbolizar a ação da gravidade sobre o personagem.

A Figura 35 apresenta os movimentos de “pulo” e a atualização do personagem após a ação de pular. Essa atualização tem o intuito de simular a ação da gravidade.

Figura 35 – Ação da gravidade após realizar o “pulo”.

```
56      def jump(self):
57          #realiza o movimento de "pulo"
58          if self.rect.bottom == 360: #altura máxima do "pu
59              self.changey = -10
60
61      def update(self):
62          #define a "gravidade"
63          self.gravity()
64
65          #atualiza o movimento do jogador
66          self.rect.x += self.changex
67          self.rect.y += self.changey
68
69          #verifica se atingiu as bordas da imagem
70          if self.rect.x < 0:
71              self.rect.x = 0
72
73          if self.rect.x > window_width - self.rect.width:
74              self.rect.x = window_width - self.rect.width
```

Fonte: adaptado de RedHuli (2018).

Após a construção da classe que contém os movimentos do personagem, é necessário definir e construir a tela que será utilizada no jogo. Além disso, é importante definir as posições iniciais dos elementos na tela. A Figura 36 apresenta as definições iniciais para a construção desse jogo.

Figura 36 – Definições iniciais para a construção do jogo.

```
79  # define a tela que será utilizada
80  screen = pygame.display.set_mode((window_width, window_height))  #obtém o objeto Surface
81  pygame.display.set_caption('Pygame - Jogo simples') #coloca o título da tela
82
83  # lista que contém todos os "blocos" do jogo
84  active_sprites_list = pygame.sprite.Group()
85
86  # Desenha o "bloco" nas posições indicadas
87  player = Player(30, 30)  #define o jogador(quadrado) de 30x30 pixels
88  player.rect.x = window_width / 2 - player.rect.centerx  #adiciona o retângulo ao meio da tela (eixo x)
89  player.rect.y = 330  #desenha o "Jogador" na altura desejada
90
91  # Adiciona os blocos para a lista
92  active_sprites_list.add(player)
```

Fonte: adaptado de RedHuli (2018).

Na linha 84, é definida uma lista que contém os objetos do tipo “Sprite”. Essa lista é utilizada para trabalhar com todos os “quadrados” criados e desenhados na tela. Dessa forma, é possível tratar com vários diferentes elementos de forma mais prática, pois ações realizadas sobre a lista podem ser repassadas para vários elementos desse “Group”. A linha 87 define a forma do “quadrado” desenhado e as linhas 88 e 89 a posição inicial (eixo x e y) para esse objeto. A linha 92 adiciona o jogador (quadrado) para a lista criada na linha 84.

Depois de definido todos os elementos necessários para a construção da tela do jogo, deve-se criar o loop infinito para tratar os eventos de interação com o usuário. A Figura 37 apresenta alguns exemplos de eventos realizados neste tutorial. Na linha 97 é apresentado o loop infinito para capturar as ações do jogo. É nesse loop que os eventos são recebidos e as ações são tomadas a partir da interação com o usuário. Os eventos são capturados através do método presente na linha 97. No “if” presente na linha 100, o evento de o usuário clicar em “sair do jogo” é capturado e tratado. Na linha 105, eventos de pressionar as teclas direcionais são capturados e a cada uma das teclas pressionadas é realizado um comando de modificação de posição do personagem na tela. Esses movimentos são tratados através da classe “Player” construída.

Figura 37 – Ilustração do loop infinito.

```
94  #loop principal para o jogo
95  while True:
96      #recebe todos os eventos de interação com o jogo
97      for event in pygame.event.get():
98
99          #se clicar em sair da tela
100         if event.type == QUIT:
101             pygame.quit()
102             sys.exit()
103
104         #recebe as teclas pessionadas
105         if event.type == pygame.KEYDOWN:
106             #tecla para a esquerda
107             if event.key == pygame.K_LEFT:
108                 player.move_left(5)
109             #tecla para a direita
110             if event.key == pygame.K_RIGHT:
111                 player.move_right(5)
112             #tecla para cima
113             if event.key == pygame.K_UP:
114                 player.jump()
```

Fonte: adaptado de RedHuli (2018).

A Figura 38 mostra quais as ações devem ser realizadas após o usuário liberar uma tecla. A linha 118 obtém esse evento de liberação de uma determinada tecla. As ações a serem tomadas são implementadas através da classe “Player”.

Figura 38 – Eventos de liberação de teclas.

```
117         #ao soltar a tecla
118         if event.type == pygame.KEYUP:
119             #atualiza ao soltar a tecla para a esquerda
120             if event.key == pygame.K_LEFT:
121                 player.move_left(-5)
122             #atualiza ao soltar a tecla para a direita
123             if event.key == pygame.K_RIGHT:
124                 player.move_right(-5)
```

Fonte: adaptado de RedHuli (2018).

Os códigos presentes na Figura 39 apresentam as etapas necessárias para realizar as atualizações das ações na tela do jogo. Essas ações correspondem a todos os movimentos de figuras e desenho de imagens exibidas durante um jogo. A linha 127 mostra como cada um dos elementos (blocos) presentes, neste caso, apenas o “personagem”, são atualizados na tela. A linha 130 realiza o preenchimento do plano de fundo da tela através da cor preta. Todos os blocos presentes na lista (neste caso, apenas o quadrado do personagem) são desenhados na tela e todas atualizações para a tela são realizadas através do comando presente na linha 135. Ao final, na linha 136, é definida a taxa com que todas as atualizações devem ocorrer para a tela.

Figura 39 – Atualização das telas.

```
126     # Adiciona a lógica para o jogo
127     active_sprites_list.update()
128
129     # "desenha" o backgroud da tela
130     screen.fill(black_color)
131
132     # desenha os "blocos" em cada momento de atualizaçao
133     active_sprites_list.draw(screen)
134
135     pygame.display.update() # atualiza a tela
136     FPSCLOCK.tick(FPS) # limita a taxa de atualização da tela
```

Fonte: adaptado de RedHuli (2018).

Anexo A – Introdução ao Keras

O Keras é uma Application Programming Interface (API) de desenvolvimento de aplicações em Machine Learning de alto nível que pode utilizar o TensorFlow, Teano ou o Microsoft Cognitive Toolkit (CNTK) pra implementar os algoritmos de Deep Learning. O Keras foi desenvolvido para facilitar o desenvolvimento de aplicações com uma interface mais amigável para o usuário. As principais vantagens em utilizar o Keras são:

- **Facilidade de implementação.**

Com apenas algumas linhas de código é possível criar uma rede neural artificial, definir as entradas e saídas, treinar essa rede e gerar previsões.

- **Open source:**

Como o Keras possui o código aberto, é possível que exista grande interação entre os desenvolvedores e a comunidade que utiliza esse framework.

- **Documentação extensa e completa:**

O Keras conta com uma documentação bastante completa e constantemente atualizada. Assim, é mais fácil compreender o funcionamento de toda a estrutura do framework.

- **Utiliza o TensorFlow como backend.**

Como o Keras pode utilizar o TensorFlow como backend, é possível construir modelos que utilizem o TensorFlow para o treinamento e avaliação da rede neural criada. Portanto, podemos fazer uso da alta performance do TensorFlow utilizando as facilidades de programação do Keras.

Para a construção de redes neurais com o Keras, estão disponíveis dois modos: o modelo Sequencial e o modelo Funcional. No modelo Sequencial, a rede é construída em sequência, ou seja, cada uma das camadas é adicionada uma após a outra em uma espécie de pilha. Na primeira camada da rede é obrigatório definir as dimensões da entrada dessa rede. A Figura A.1 apresenta um exemplo de criação de uma rede neural utilizando o modelo sequencial.

Figura A.1 – Exemplo de criação de uma rede utilizando o modelo Sequencial do Keras.

```
model = Sequential()

model.add(Dense(4, activation='relu', input_dim=3))
model.add(Dense(4, activation='relu',input_dim=4))
model.add(Dense(4, activation='relu',input_dim=4))
model.add(Dense(1, activation='relu', input_dim=4))
```

O modelo Funcional do Keras apresenta algumas diferenças. Nesse modelo a ligação entre as camadas ocorre de forma diferente. É necessário indicar qual deve ser a entrada em cada uma das camadas, ou seja, para cada camada adicionada ao modelo deve-se indicar a saída de qual camada será utilizada como entrada dessa camada criada. Na camada de entrada, também é necessário indicar qual é o formato dos dados utilizados para o treinamento e avaliação do modelo. A Figura A.2 mostra um exemplo de utilização do modelo funcional para a criação de uma rede simples.

Figura A.2 – Exemplo de criação de uma rede utilizando o modelo Funcional do Keras.

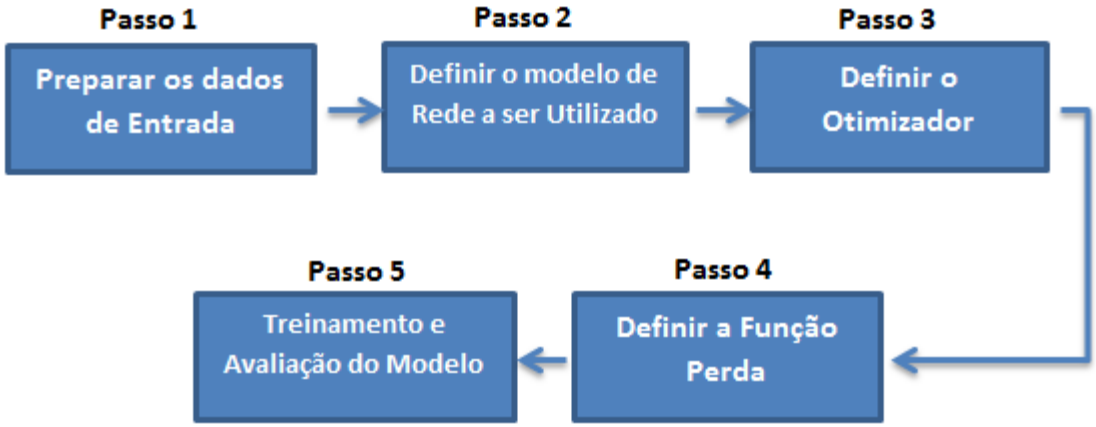
```
#cria a camada de entrada para a rede
entrada=Input(shape=(3,))#especifica o tamanho (formato) da entrada
```

```
x1=Dense(4,activation='relu')(entrada)# cria a primeira camada escondida e o link com a camada de entrada
x2=Dense(4,activation='relu')(x1) # cria a segunda camada escondida
y=Dense(1,activation='softmax')(x2)
```

```
#cria o modelo
model=Model(inputs=entrada,outputs=y)
```

Para a implementação de todas as aplicações presentes neste curso, será utilizado o modelo Sequencial. Além disso, para essas implementações será adotada a sequência de passos presentes na Figura A.3. A adoção desses passos visa tornar sistemática a aplicação dos conceitos apresentados no decorrer deste curso.

Figura A.3 – Passos a serem seguidos para implementar as redes no Keras.



Referências

FORBES, Elliot. *Learning Concurrency in Python*: Build highly efficient, robust, and concurrent applications. 2017.