Python Avançado

Rennan Alves Cardoso

2023

SUMÁRIO

> Capítulo 1

Capítulo 1. Machine Learning com Python

Nos últimos anos, o conceito de Inteligência Artificial (IA) tem elencado um grande hype na comunidade científica e empresarial, destes artigos científicos até produções industriais, como carros autônomos (Tesla), assistentes virtuais (Alexa) entre outras aplicações automatizadas que dispensa, muitas vezes, a iteração humana. Uma nova tendência de mercado e visão do mundo está chegando com a aplicabilidade do aprendizado de máquina e você, aluno IGTI, não ficará de fora da evolução do mercado.

Neste curso serão abordados diversos métodos e algoritmos, como Árvores de Decisão, Regressão Linear e Logística, K-means, SVM e Redes Neurais Artificiais.

Aprendizado de Maquina

Aprendizado Supervisionado

O que é aprendizado de máquina supervisionado e como ele se relaciona com o aprendizado não supervisionado?

A maioria das práticas de aprendizado de máquina utiliza aprendizado supervisionado. Nesta metodologia você fornece um valor *(X)* e uma saída *(Y)* e usa um algoritmo para aprender o mapeamento da função entre a entrada e saída.

$$Y = f(X)$$

O objetivo é aproximar o mapeamento da função tão próximo que quando você insere um novo dado de entrada (X), você poderá predizer qual será a saída desse novo dado. Isso é chamado de aprendizado supervisionado, pois o processo de aprendizado do algoritmo se dá através de um conjunto de dados de treinamento que irá ajustar/ensinar o processo de aprendizado supervisionado. Sabemos as respostas corretas, o algoritmo

iterativamente faz previsões sobre os dados de treinamento e é corrigido pelo professor. O aprendizado para quando o algoritmo atinge um nível aceitável de desempenho. Os problemas de aprendizagem supervisionada podem ser agrupados em problemas de regressão e classificação.

- **Classificação**: um problema de classificação é quando a variável de saída é uma categoria, como preto ou branco, doente ou sadio.

- **Regressão**: uma abordagem de regressão é quando a variável de saída é um valor real, como preço de uma ação na bolsa de valores ou de um aluguel de uma casa.

Algumas abordagens mais comuns que utilizam classificação e regressão incluem sistemas de recomendação e previsões de séries temporais, respectivamente.

Aprendizado Não Supervisionado

O aprendizado não supervisionado é quando você possui apenas o dado de entrada *(X)* e não corresponde a um dado de saída. O objetivo do aprendizado não supervisionado é modelar a estrutura ou distribuição subjacente nos dados para saber mais sobre os dados.

Esta modalidade é chamada de não supervisionada, pois ao contrário do método supervisionado, não há resposta correta para orientar o aprendizado e não há professor para isso. Os algoritmos são deixados por conta própria para descobrir e apresentar uma estrutura sobre os dados. Esta abordagem está mais direcionada para problemas que envolvem agrupamentos e associações.

- **Agrupamento *(Clustering):*** um agrupamento é onde você deseja descobrir os agrupamentos inerentes nos dados, como agrupar clientes por comportamento de compra.

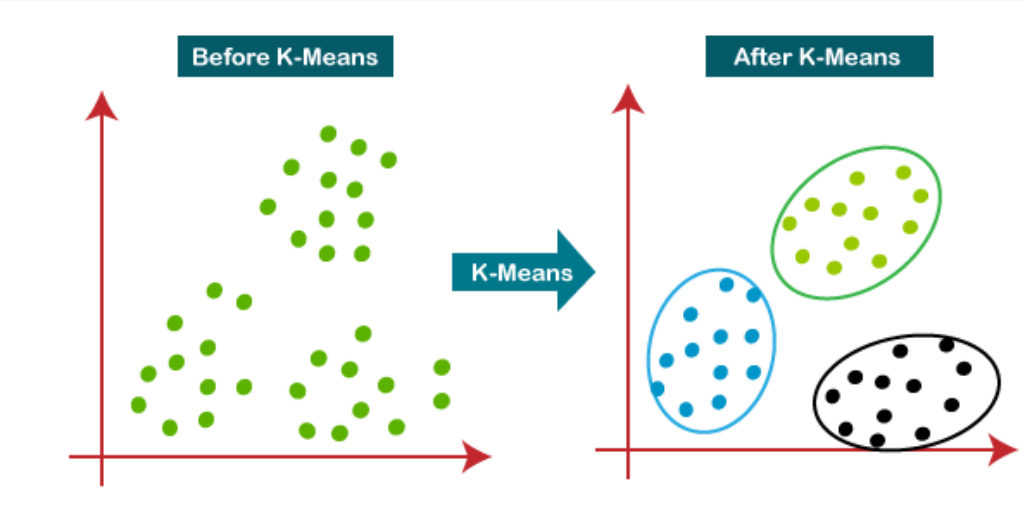
- **Associação:** uma abordagem de aprendizado de regras de associação é onde você deseja descobrir regras que descrevem grandes porções de seus dados, como as pessoas que compram A também tendem a comprar B.

O algoritmo mais utilizado nesta abordagem é o K-*means* para problemas de agrupamento.

K-means

O *K-means* é um método não supervisionado de aprendizado de máquina usada para identificar agrupamentos de dados em um dataset. Existem diferentes métodos de agrupamentos, no entanto o *k-means* é uma das técnicas mais utilizadas por possuir uma fácil usabilidade dos seus parâmetros. Pode-se observar o processo de classificação, conforme a Figura 1.

Figura 1 – Classificação k-means.



K-*means* é um algoritmo interativo que agrupa um conjunto de dados em 'n' subgrupos, onde 'n' é a quantidade de grupos definida pelo operador do algoritmo. A classificação dos dados em cada grupo se dá pela distância média do dado ao centroide do subgrupo em particular. Se n = 3, significa que o número de subgrupos que os dados serão subdivididos será em 3.

O funcionamento do algoritmo k-means é descrito da seguinte forma:

1. Defina a quantidade[1] de clusters que deseja subdividir seus dados.
2. Os centroides são iniciados de forma aleatória.
3. Atribua cada ponto de dado à distância dos centroides mais próximos deste dado.
4. Coloque um novo centroide de cada agrupamento.
5. Repita no passo três, que redefiniu cada ponto de dados ao novo centroide mais próximo de cada grupo.

Podemos acompanhar um exemplo de implementação conforme as imagens a seguir. Na Figura 2, pode-se verificar o conjunto de dados que será classificado.

Figura 2 – Conjunto de dados para o k-means.

```
df_iris = pd.read_csv("Iris.csv")
```

```
df_iris.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

A partir dos dados, é implementado o algoritmo para classificação dos agrupamentos. A Figura 3 apresenta o processo desta classificação.

[1] Um método muito utilizado para definição do número de cluster é o *Elbow Method* (método do cotovelo), no qual testa a variância dos dados em relação ao número de clusters.

Figura 3 – Implementação do k-means.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import seaborn as sns
sns.set(style="darkgrid")
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline
```

```
kmeans = KMeans(n_clusters = 3, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
y_kmeans = kmeans.fit_predict(x)
```

Fonte: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

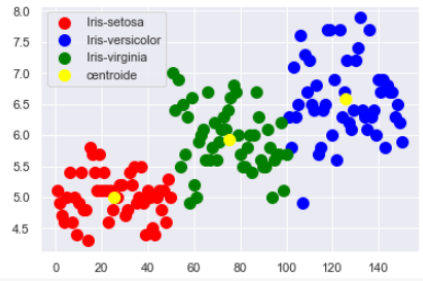
Uma vez implementado, pode-se visualizar os agrupamentos com a biblioteca *matplotlib,* conforme a seguinte implementação demonstrada na figura 4.

Figura 4 – Visualização dos agrupamentos.

```
plt.scatter(x[y_kmeans == 0 ,0], x[y_kmeans == 0,1], s = 100, c = 'red', label="Iris-setosa")
plt.scatter(x[y_kmeans == 1 ,0], x[y_kmeans == 1,1], s = 100, c = 'blue', label="Iris-versicolor")
plt.scatter(x[y_kmeans == 2 ,0], x[y_kmeans == 2,1], s = 100, c = 'green', label="Iris-virginia")

plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1], s = 100, c='yellow', label = 'centroide')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x21e36ac8d00>
```



Regressão Linear

Regressão é um processo de Machine Learning supervisionado. É semelhante à classificação, mas, em vez de prever um rótulo, tentamos prever um valor contínuo. De forma prática, se você estiver tentando prever um valor contínuo, utilize regressão.

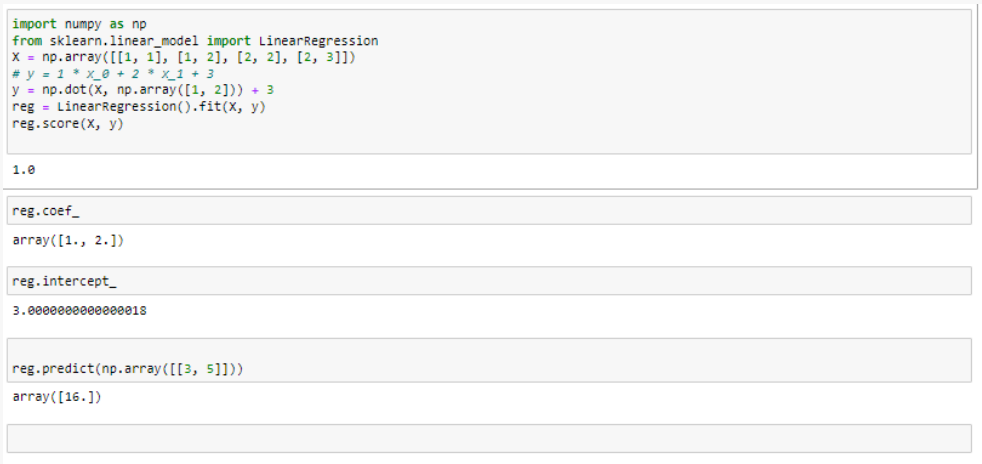
Uma Regressão Linear simples é ensinada em cursos de matemática e em cursos introdutórios de estatística. Ela tenta fazer a adequação da fórmula $y = mx + b$, ao mesmo tempo que minimiza o quadrado dos erros.

Esse modelo pressupõe que a predição é uma combinação linear dos dados de entrada. Para alguns conjuntos de dados, isso não será suficientemente flexível. Mais complexidade poderá ser acrescentada por meio da transformação dos atributos.

Em outras palavras, a análise de regressão linear é usada para prever o valor de uma variável com base no valor de outra. A variável que deseja prever é chamada de variável dependente. A variável que é usada para prever o valor de outra variável é chamada de variável independente.

Para ilustrar, a Figura 5 apresenta a criação de uma regressão linear por meio da biblioteca *sklearn,* nos quais os dados de entrada são observados pelos valores inseridos na variável X, a variável Y é o alvo da aplicação.

Figura 5 – Criando uma regressão linear com a biblioteca Sklearn.



Cabe salientar que o módulo de regressão linear do método sklearn possui alguns métodos dos quais podem ser acompanhados de acordo com a tabela abaixo:

fit(X,y[,sample_weight])	Ajuste do modelo linear.
get_params([deep])	Obtém parâmetros para este estimador.
predict(X)	Prevê os dados da regressão no modelo linear

Score(X,y[,sample_weight])	Retorna o coeficiente de determinação da predição.
Set_params(**param)	Define os parâmetros do estimador.

Árvores de decisão

Entre os métodos mais utilizados, árvores de decisão se destacam no aprendizado de máquina do tipo supervisionado. Esse algoritmo aplica uma segmentação progressiva do conjunto de dados, criando assim partes cada vez mais especificadas destes dados até alcançarem um mapeamento simples o bastante para ser rotulado. Para isso, é necessário treinar o modelo com dados previamente rotulados, de modo a aplicá-lo a dados novos.

Nesse modelo se usa uma estrutura de árvore para representar um número de possíveis **caminhos de decisão,** e um resultado para cada caminho. Diferentemente de outros modelos, as árvores de decisão são muito fáceis de entender e interpretar, e o processo de encontrar a previsão pretendida é muito transparente. Uma outra particularidade das árvores de decisão é a possibilidade de misturar diferentes tipos de dados (numéricos e categóricos).

Conforme Harrison (2020), árvores de decisão é como ir ao médico e ser submetido a uma série de questionamentos com o objetivo obter um diagnóstico. Semelhantemente, pode-se usar um processo de perguntas para prever uma classe alvo do seu modelo.

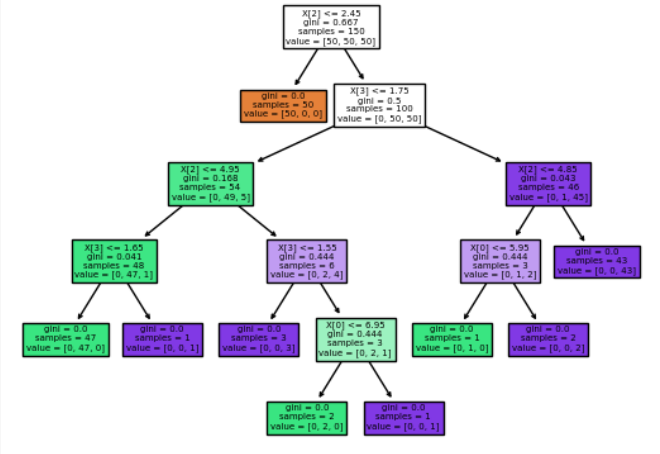
A Figura 6 e 7 demonstram um exemplo utilizando o dataset Iris e a biblioteca Scikit-learn:

Figura 6 – Criando Arvore de decisão com sklearn.

```
from sklearn.datasets import load_iris
from sklearn import tree
iris = load_iris()
X, y = iris.data, iris.target
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)
tree.plot_tree(clf)
```

Dessa forma o processo de classificação dos dados nos gera o seguinte gráfico de árvore de decisão:

Figura 7 – Plot árvore de decisão.



É possível melhorar a visualização dos gráficos com outras bibliotecas, como por exemplo o dtreeviz, graphviz XGBoost entre outros.

Floresta aleatória

Florestas aleatórias é um método de aprendizagem de conjunto para classificação e regressão que abrange um conjunto de árvores de decisão no momento de treinamento. Dessa forma, cada árvore de decisão que compõe a floresta é definida como uma classe (classificação) ou previsão/média (regressão) das árvores individuais.

Conforme Niklas Donges (2018), o algoritmo floresta aleatória adiciona aleatoriedade extra no modelo, quando está criando as árvores. Ao invés de procurar pela melhor característica (features) no ato da partição dos

nós. Esse processo cria uma grande diversidade, o que o geralmente leva a geração de modelos melhores.

Portanto, quando você está criando uma árvore na floresta aleatória, apenas um subconjunto aleatório das características é considerado na partição de um nodo. Você pode até fazer as árvores ficarem mais aleatórias utilizando limiares (*thresholds*) aleatórios para cada característica, ao invés de procurar pelo melhor limiar (como uma árvore de decisão geralmente faz).

Floresta Aleatória é uma coleção de árvores de decisão, mas há algumas diferenças entre esses dois modelos. Se você treinar uma árvore com um dataset de treinamento e rótulos, ela vai elaborar um conjunto de regras que serão utilizadas para realizar predições.

Por exemplo, se você quiser prever se uma pessoa vai clicar em um anúncio on-line, você pode colecionar anúncios que a pessoa clicou no passado e algumas características que descrevam esta decisão. Se você colocar as características e rótulos em uma árvore de decisão, ela vai gerar nós e algumas regras. Você poderá então prever ser o anúncio será clicado ou não.

Quando uma árvore de decisão gera regras e nodos, normalmente utiliza cálculos de ganho de informação. Por outro lado, a Floresta Aleatória faz isso de modo aleatório.

Outra diferença é que árvores de decisão profundas podem sofrer de sobreajuste (overfitting). Florestas Aleatórias evitam o sobreajuste na maioria dos casos, pois trabalham com subconjuntos aleatórios das características e constroem árvores menores a partir de tais subconjuntos. Depois do treinamento, as subárvores são combinadas. Esta abordagem torna a computação mais lenta, dependendo de quantas árvores serão construídas pelo Floresta Aleatória.

Na imagem a seguir pode-se contemplar um exemplo de codificação do algoritmo Random Forest, utilizando a biblioteca sklearn.

Figura 8 – Criando uma Random Forest com sklearn.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=4,
                           n_informative=2, n_redundant=0,
                           random_state=0, shuffle=False)
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X, y)

print(clf.predict([[0, 0, 0, 0]]))
```

```
[1]
```

Com poucas exceções, RandomForestClassifier tem todos os hiperparâmetros de um DecisionTreeClassifier (para controlar como as arvoes se desenvolvem). A biblioteca Sklearn traz consigo os seguintes métodos de aplicação:

apply(X)	Aplique árvores na floresta a X, devolva índices de folhas.
decision_path(X)	Retorne o caminho da decisão na floresta.
fit(X, y[, sample_weight])	Construa uma flores de árvores a partir do conjunto de treinamento (X,y)
get_params([deep])	Obtenha parâmetros para este estimador.
predict(X)	Prever o alvo de regressão para X
score(X, y[, sample_weight])	Devolva o coeficiente de determinação da previsão.
set_params(**params)	Defina os parâmetros deste estimador.

Regressão Logística

A regressão logística é um algoritmo de Machine Learning do tipo supervisionado, que tem sua aplicabilidade para classificação. Apesar de possuir “regressão” em seu nome, que por sua vez está relacionado com a aplicação de uma transformação da função de ativação sigmoide ou função logística, sobre a regressão linear.

A regressão logística é importante no sentido de classificar de maneira discreta a variável de interesse (0 ou 1, verdadeiro ou falso etc.). Sua

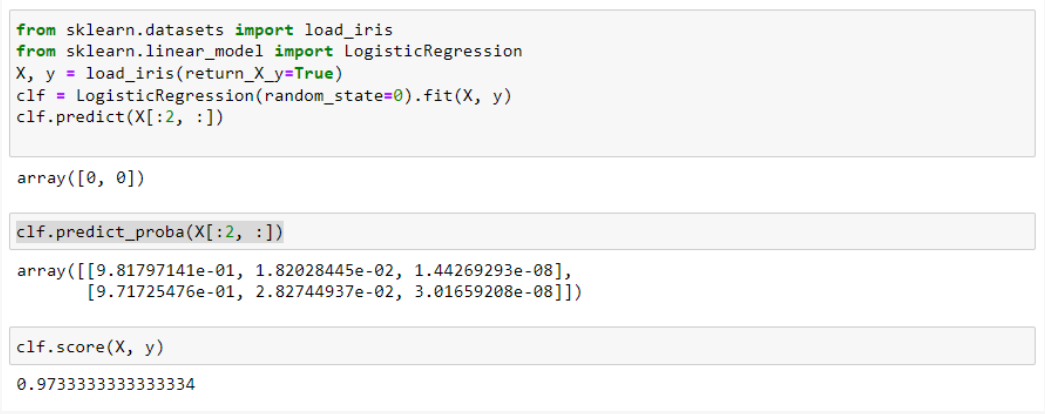
aplicação tem viabilidade em diferentes campos e uma abertura para aplicar diferentes formas de encaixe na modelagem, afirma Leite (2020).

De acordo com Leite (2020), A regressão logística é compreendida, basicamente, como uma regressão linear ajustada por um parâmetro (log). Ela foi uma maneira de resolver um problema comum dentro da forma linear: a existência de uma variável dependente categórica binário, como por exemplo 0 ou 1 (ao invés de variáveis contínuas). No modelo padrão de regressão linear, temos somente uma variável contínua, afirma Leite (2020).

Alguns aspectos são importantes ao considerar uma regressão logística, como o formato dos dados, a existência ou não de correlação entre as variáveis, entre outros. Em um primeiro ponto, enquanto em uma regressão linear não é (usualmente) necessário ter um banco de dados grande, já que no caso de a versão logística tem uma quantidade maior de observações, outros pontos a se considerar também estariam na classificação ser binária, ou seja, apresentar mais de uma variável (no caso duas) a serem relevantes, no sentido de determinar resultados e explicações claras. No entanto, podem ter mais n's.

A seguir, na imagem Xiiii, podemos observar uma simples implementação de regressão logística utilizando a biblioteca sklearn com o conjunto de dados Iris.

Figura 9 – Regressão logística.



A seguir são apresentados alguns parâmetros da biblioteca Sklearn:

decision_function(X)	Preveja pontuações de confiança para amostras.
densify()	Converta a matriz de coeficientes para o formato de matriz densa.
fit(X, y[, sample_weight])	Ajuste o modelo de acordo com os dados de treinamento fornecidos.
get_params([deep])	Obtenha parâmetros para este estimador.
predict(X)	Prever rótulos de classe para amostras em X.
predict_log_proba(X)	Prever o logaritmo das estimativas de probabilidade.
predict_proba(X)	Estimativas de probabilidade.
score(X, y[, sample_weight])	Retorne a precisão média nos dados e rótulos de teste fornecidos.
set_params(**params)	Defina os parâmetros deste estimador.
sparsify()	Converta a matriz de coeficientes para o formato esparso.

Redes Neurais Artificiais

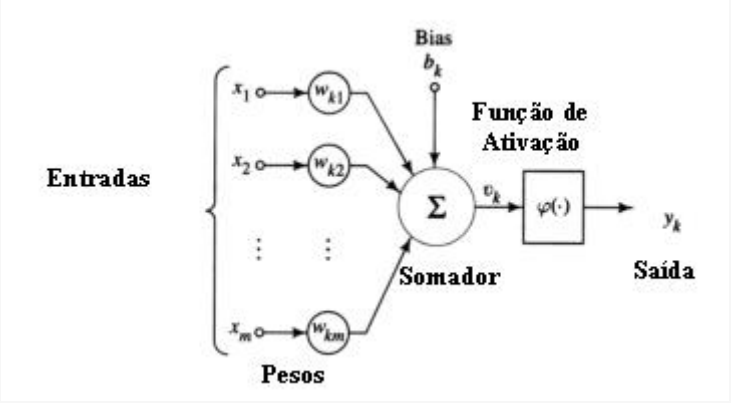
Sob o olhar computacional, pode-se dizer que em um neurônio é realizado o processamento sobre uma ou, geralmente, várias entradas, a fim de gerar uma saída.

O neurônio artificial é uma estrutura lógico-matemática que procura simular a forma, o comportamento e as funções de um neurônio biológico. Pode-se, a grosso modo, associar o dendrito à entrada, a soma ao processamento e o axônio à saída; portanto, o neurônio é considerado uma unidade fundamental processadora de informação.

Os dendritos são as entradas, cujas ligações com o corpo celular artificial são realizadas através de canais de comunicação que estão associados a um determinado peso (simulando as sinapses). Os estímulos captados pelas entradas são processados pela função soma, e o limiar de

disparo do neurônio biológico é substituído pela função de transferência. A figura a seguir apresenta esquematicamente um neurônio de McCullock e Pitts.

Figura 10 – Modelo de Neurônio Artificial.



Fonte: gsigma.

A Figura 10 representa a unidade fundamental do neurônio artificial, o Perceptron. Essa unidade é responsável por operar (calcular) as características de um conjunto de informações (dados).

O processo de cálculos, também conhecido como *feedforward,* é o andamento dos cômputos da esquerda para a direita. Nesta etapa, cada valor de entrada '*xi*' multiplica-se com o seu respectivo peso *wi,* posteriormente é feito um somatório de todas as multiplicações proferidas pelas entradas, como ilustra a equação 1.

$$\sum_{i=1}^{n} \mathrm{W}i\ xi, \qquad \text{(Eq.1)}$$

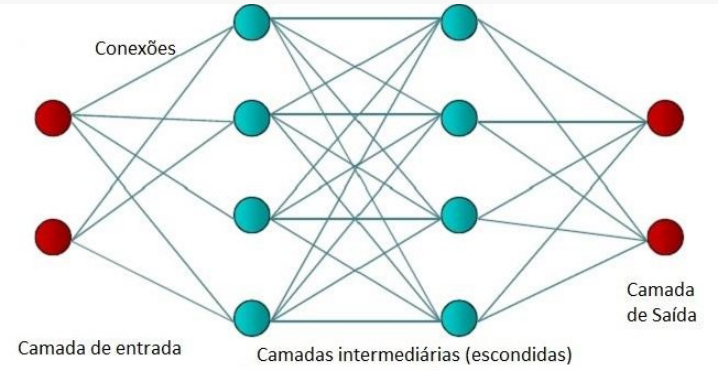
Ao obter o valor do somatório, é verificado o valor de não linearidade, para isso utiliza-se o que se denomina de função de ativação. Essa função possui esse nome devido ao fato que a mesma é responsável por "ativar" os neurônios que representem escolhas, ou características importantes ao problema.

O modelo perceptron é eficaz para tratar problemas lineares, no entanto, a maioria dos problemas não são linearmente separáveis, dessa

forma, depois de muitos anos de pesquisas, buscou-se conectar vários neurônios, formando assim uma rede neural artificial, ou seja, a multi-camada-perceptron (MLP).

Nesta arquitetura os neurônios são organizados em camadas e a informação é propagada de uma camada para outra, até que, ao final da rede, haja uma predição. A Figura 11, ilustra a representação da rede MLP com 2 entradas, 2 camadas escondidas ou intermediárias.

Figura 11 – Rede Neural Feedfoward.



Na ilustração da Figura 11, a camada formada pelos neurônios vermelhos mais à esquerda é a camada de entrada, que admite os inputs do modelo; a camada vermelha à direta é a camada da saída, onde são feitas as predições, e as camadas verdes são chamadas de *hidden layers* (camadas escondidas), onde cada neurônio é responsável por uma "característica" inerente à predição. A Figura 12 apresenta a forma como a rede MLP é criada utilizando a biblioteca sklearn.

É necessário para o processo de treinamento da rede neural importar a importação à biblioteca que operará a construção do modelo da rede, indicando as entradas, a quantidade de camadas escondidas e neurônios em cada uma das camadas.

Figura 12 – Implementação rede MLP.

```
from sklearn.neural_network import MLPClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(5, 2), random_state=1)
clf.fit(X, y)
```

```
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_state=1,
              solver='lbfgs')
```

Inicialmente define-se os dados de entradas e saída, sendo X e Y, respectivamente, posteriormente, pode-se apreciar também que no parâmetro **hidden_layer_sizes** possui duas camadas escondidas, no qual a primeira possui cinco e a segunda dois neurônios. Uma vez definido, o comando **clf.fit** inicia o processo de treinamento da rede neural.

Uma vez treinada a rede neural, pode-se prever a classificação dos novos dados de entrada.

Figura 13 – Previsão rede neural.

```
clf.predict([[2., 2.], [-1., -2.]])
```

```
array([1, 0])
```

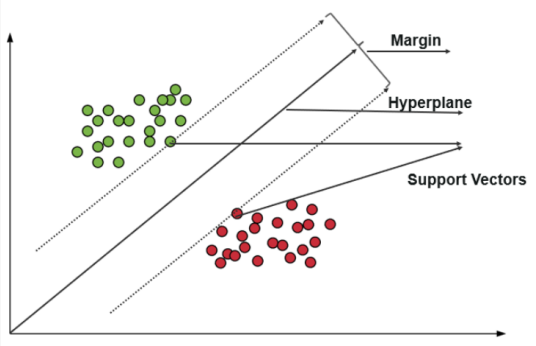
Pode-se observar que é possível prever classificações de dados utilizando redes neurais artificiais, entretanto, este modelo é utilizado em abordagem mais complexas de inferências.

Máquina de vetores de suporte - SVM

Uma SVM (*Suporte Vector Machine)* é um algoritmo que tenta fazer a adequação de uma linha entre as diferentes classes, de modo a maximizar a distância da linha até os pontos das classes. Dessa forma, ela tenta encontrar uma divisão consistente entre as classes. Os vetores de suporte são as divisas.

A figura a seguir representa a divisão realizada pelo SVM pelo hiperplano.

Figura 14 – Separação realizada pelo SVM.



Em geral, a SVM tem um bom desempenho e oferece suporte para espaços não lineares e lineares, utilizando o truque do kernel (*kenel trick*).

A Figura 15 mostra um exemplo de aplicação do SVM com uma base de dados simples.

Figura 15 – Construção do SVM.

```
from sklearn import svm
X = [[0, 0], [1, 1]]
y = [0, 1]
clf = svm.SVC()
clf.fit(X, y)
```

```
SVC()
```

Uma vez construído e treinado, pode-se prover a predição de classificação dos dados.

Figura 16 – Saída do SVM.

```
clf.predict([[2., 2.]])
```

```
array([1])
```

> Capítulo 2

Capítulo 2. Concorrência

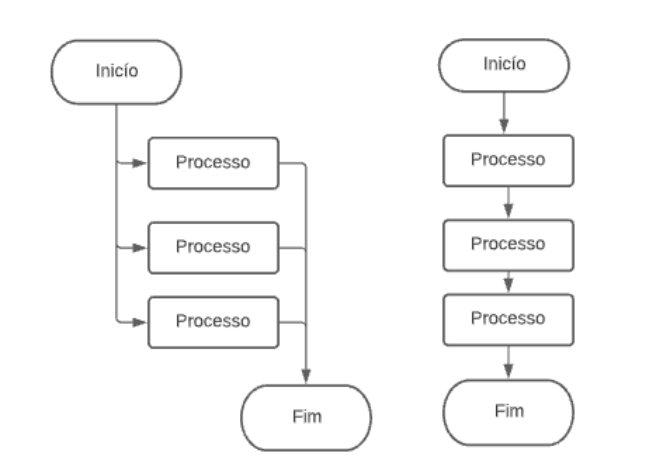
Estima-se que a quantidade de dados que precisam ser processados por programas de computador dobre a cada dois anos. A *Internacional Data Corporation* (IDC), por exemplo, estima que há 5.200 GB de dados para cada pessoa na Terra. Com esse volume impressionante de dados, surgem demandas insaciáveis por poder de computação e, embora inúmeras técnicas de computação estejam sendo desenvolvidas e utilizadas todos os dias, a programação simultânea continua sendo uma das maneiras mais importantes de processar dados com eficiência e precisão.

Enquanto alguns podem ficar intimidados quando a palavra simultaneidade aparece, a noção por trás dela é bastante intuitiva, e é muito comum, mesmo em um contexto de não programação. No entanto, isso não quer dizer que os **programas concorrentes** sejam tão simples quanto os sequenciais; eles são realmente mais difíceis de escrever e entender. No entanto, uma vez alcançada uma estrutura concorrente correta e eficaz, ocorrerá uma melhoria significativa no tempo de execução, como você verá mais adiante

Programação Concorrente vs Sequencial

Talvez a maneira mais óbvia de entender a programação concorrente seja compará-la à programação sequencial. Enquanto um programa sequencial está em um lugar por vez, em um programa concorrente, diferentes componentes estão em estados independentes ou semi-independentes. Isso significa que componentes em estados diferentes podem ser executados de forma independente e, portanto, ao mesmo tempo (já que a execução de um componente não depende do resultado de outro). A imagem a seguir ilustra as diferenças básicas entre esses dois tipos:

Figura 17 – Diferença entre Concorrência e programação Sequencial.



Uma vantagem imediata da simultaneidade é uma melhoria no tempo de execução. Novamente, como algumas tarefas são independentes e podem, portanto, ser concluídas ao mesmo tempo, é necessário menos tempo para o computador executar todo o programa.

Concorrência vs Paralelismo

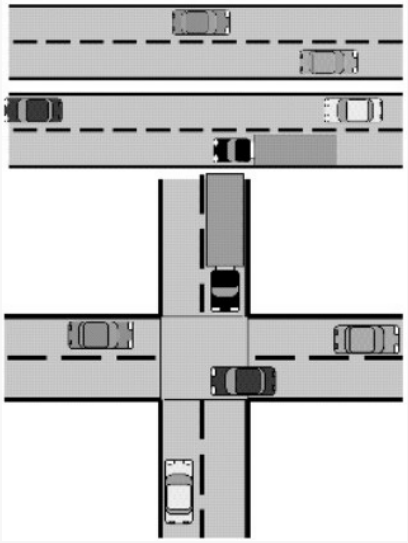
Neste ponto, se você já teve alguma experiência em programação paralela, pode estar se perguntando se simultaneidade é diferente de paralelismo.

A principal diferença entre programação simultânea e paralela é que, enquanto em programas paralelos existem vários fluxos de processamento (principalmente CPUs e núcleos) trabalhando independentemente de uma só vez, pode haver diferentes fluxos de processamento (principalmente threads) acessando e usando um recurso compartilhado ao mesmo tempo em programas simultâneos. Como esse recurso compartilhado pode ser lido e sobrescrito por qualquer um dos diferentes fluxos de processamento, às vezes é necessária alguma forma de coordenação, quando as tarefas que precisam ser executadas não são totalmente independentes umas das outras.

Em outras palavras, é importante que algumas tarefas sejam executadas após as outras, para garantir que os programas produzam o resultado correto.

A figura a seguir ilustra a diferença entre simultaneidade e paralelismo: enquanto no destaque superior, atividades paralelas (neste caso, carros) que não interagem entre sipodem ser executadas ao mesmo tempo, na seção inferior, algumas tarefas precisam esperar para que outros terminem antes de serem executadas.

Figura 18 – Diferenças entre concorrência e paralelismo.



Fonte: Nguyen, 2018.

Threads

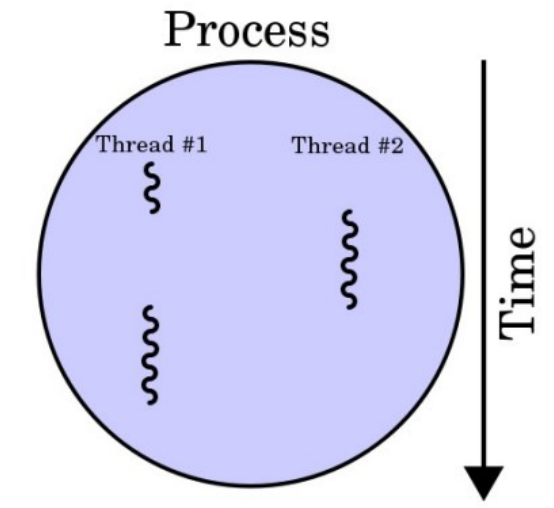
No campo da ciência da computação, uma thread de execução é a menor unidade de comandos de programação (código) que uma *schedule* (geralmente como parte de um sistema operacional) pode processar e gerenciar. Dependendo do sistema operacional, a implementação de threads e processos varia, mas um thread normalmente é um elemento (um componente) de um processo.

Threads vs Processos

Mais de uma thread pode ser implementada dentro do mesmo processo, na maioria das vezes executando concorrentemente e acessando/compartilhando os mesmos recursos, como memória. Processos separados não fazem isso. Threads no mesmo processo compartilham as instruções deste último (seu código) e contexto (os valores que suas variáveis referenciam em um dado momento). A principal diferença entre os dois conceitos é que um thread é normalmente um componente de um processo. Portanto, um processo pode incluir vários threads, que podem ser executados simultaneamente. Os threads também geralmente permitem o compartilhamento de recursos, como memória e dados, embora seja bastante raro que os processos o façam.

Em suma, uma thread é um componente independente de computação que é semelhante a um processo, mas as threads dentro de um processo podem compartilhar o espaço de endereço e, portanto, os dados desse processo.

Figura 19 – Processo com duas threads em execução.



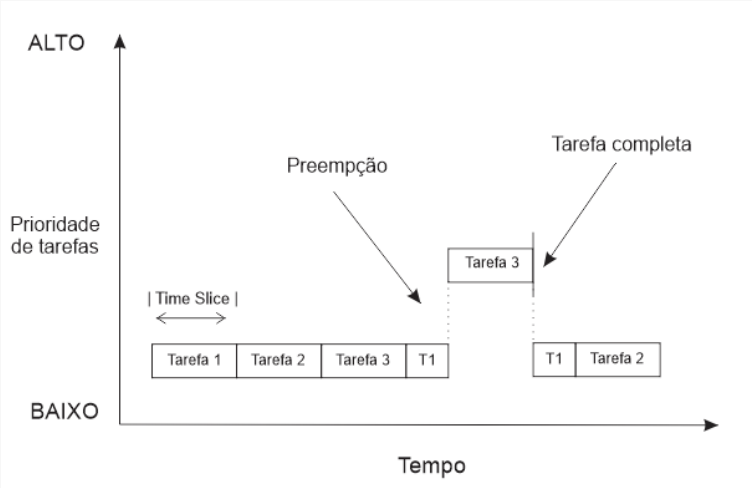
Fonte: Nguyen,2018.

Multithreads

Na ciência da computação, o single-threading é semelhante ao processamento sequencial tradicional, executando um único comando a qualquer momento. Por outro lado, multithreading implementa mais de uma thread para existir e executar em um único processo, simultaneamente. Ao permitir que vários threads acessem recursos/contextos compartilhados e sejam executados de forma independente, essa técnica de programação pode ajudar os aplicativos a ganhar velocidade na execução de tarefas independentes.

O multithreading pode ser obtido principalmente de duas maneiras. Em sistemas de processador único, o multithreading é normalmente implementado por meio de fatiamento de tempo, uma técnica que permite que a CPU alterne entre diferentes softwares executados em diferentes threads. Então, no tempo de divisão a CPU alterna sua execução tão rapidamente e com tanta frequência que os usuários geralmente percebem que o software está sendo executado em paralelo (por exemplo, quando você abre dois softwares diferentes ao mesmo tempo em um computador com um único processador).

Figura 20 – Demonstração da técnica de time slice.



Ao contrário dos sistemas de processador único, os sistemas com vários processadores ou núcleos podem implementar facilmente

multithreading, executando cada thread em um processo ou núcleo separado, simultaneamente.

Além disso, o fatiamento de tempo é uma opção, pois esses sistemas multiprocesso ou *multicore* podem ter apenas um processador/núcleo para alternar entre tarefas, embora isso geralmente não seja uma boa prática.

Os aplicativos *multithread* têm várias vantagens, em comparação com os aplicativos sequenciais tradicionais, alguns deles estão listados a seguir:

- **Tempo de execução mais rápido:**

Uma das principais vantagens da simultaneidade através de multithreading é a aceleração que é alcançada. Threads separadas no mesmo programa podem ser executadas simultaneamente ou em paralelo, se forem suficientemente independentes umas das outras.

- **Capacidade de resposta:**

Um programa de thread único pode processar apenas uma parte da entrada por vez; portanto, se o thread de execução principal bloquear em uma tarefa de longa duração (ou seja, uma entrada que requer computação e processamento pesados), todo o programa não poderá continuar com outra entrada e, portanto, parecerá ser congelado. Ao usar threads separados para realizar a computação e permanecer em execução para receber diferentes entradas do usuário simultaneamente, um programa multithread pode fornecer melhor capacidade de resposta.

- **Eficiência no consumo de recursos:**

Como mencionamos anteriormente, vários threads dentro do mesmo processo podem compartilhar e acessar os mesmos recursos. Consequentemente, os programas multithread podem atender e processar

muitas solicitações de dados de clientes simultaneamente, usando significativamente menos recursos do que seriam necessários ao usar programas de thread único ou multiprocesso. Isso também leva a uma comunicação mais rápida entre os threads.

Entretanto, as multithreads possuem também suas desvantagens:

- **Falhas:**

Mesmo que um processo possa conter várias threads, uma única operação ilegal em uma thread pode afetar negativamente o processamento de todas as outras threads do processo e, como resultado, pode travar todo o programa.

- **Sincronização:**

Embora compartilhar os mesmos recursos possa ser uma vantagem sobre a programação sequencial tradicional ou programas de multiprocessamento, também é necessária uma consideração cuidadosa para os recursos compartilhados. Normalmente, as threads devem ser coordenadas de maneira deliberada e sistemática, para que os dados compartilhados sejam computados e manipulados corretamente. Problemas não intuitivos que podem ser causados por coordenação de threads descuidada incluem *deadlocks, livelocks* e condições de corrida, todos os quais serão discutidos em capítulos futuros.

Para ilustrar o conceito, vamos exemplificar um código com várias threads, que pode ser acompanhado na imagem 2.4, a seguir. No código abaixo estamos usando o módulo de **threading** do Python como base da classe **MyThread.** Cada objeto desta classe tem um nome e um parâmetro de atraso. A função **run()**, que é chamada assim que uma nova thread é inicializada e iniciada, imprime uma mensagem inicial e, por sua vez, chama a função thread_count_down(). Esta função faz a contagem regressiva do

número 5 ao número 0, enquanto dorme entre as iterações por um número de segundos, especificado pelo parâmetro *delay*.

Figura 21 - Criação da classe threads.

```
#Por Nguyen, adaptado.
import threading
import time


class MyThread(threading.Thread):
    def __init__(self, name, delay):
        threading.Thread.__init__(self)
        self.name = name
        self.delay = delay

    def run(self):
        print('Iniciando thread %s.' % self.name)
        thread_count_down(self.name, self.delay)
        print('Finalizando thread %s.' % self.name)

def thread_count_down(name, delay):
    counter = 5

    while counter:
        time.sleep(delay)
        print('Thread %s contagem decrescente: %i...' % (name, counter))
        counter -= 1
```

Aqui, estamos inicializando e iniciando dois threads juntos, cada um com 0,5 segundos como parâmetro de atraso, e assim temos as seguintes saídas:

```
thread1 = MyThread('A', 0.5)
thread2 = MyThread('B', 0.5)

thread1.start()
thread2.start()

thread1.join()
thread2.join()


print('Fim.')
```

```
Iniciando thread A.
Iniciando thread B.
Thread B contagem decrescente: 5...
Thread A contagem decrescente: 5...
Thread B contagem decrescente: 4...
Thread A contagem decrescente: 4...
Thread B contagem decrescente: 3...Thread A contagem decrescente: 3...

Thread A contagem decrescente: 2...Thread B contagem decrescente: 2...

Thread A contagem decrescente: 1...Thread B contagem decrescente: 1...
Finalizando thread A.

Finalizando thread B.
Fim.
```

> Capítulo 3

Capítulo 3. Programação Reativa

Neste capitulo, aprenderemos a usar a biblioteca RxPy para criar programas assíncronos e baseados em eventos, implementando observáveis, observadores/assinantes e assuntos.

O que é Programação Reativa?

De certa forma, programação reativa não é uma coisa nova. Nossos eventos típicos de cliques são um fluxo de dados assíncrono que podemos observar e desencadear ações a partir dele. É assim que funciona, mas a Programação Reativa torna as coisas muito mais fáceis adicionando uma caixa de ferramentas de operadores para filtrar, criar, transformar e unificar qualquer um desses fluxos. Em apenas algumas linhas de código, podemos ter *requests* que recebem várias solicitações e as manuseiam em um processo assíncrono que serve a uma saída filtrada.

Por que Programação Reativa?

Os aplicativos da Web contêm muitas operações de banco de dados, chamadas de rede, retornos de chamadas aninhados e outras tarefas computacionalmente caras que podem levar muito tempo para serem concluídas (ou até mesmo bloquear outros tópicos até que seja feito). É aí que entra a Programação Reativa, nos dá a facilidade de converter quase tudo em fluxos (como variáveis, propriedades, entradas de usuários, caches etc.) para gerenciá-lo de forma assíncrona. Além disso, também nos dá uma maneira fácil de lidar com erros. Uma tarefa que geralmente é tarefa difícil dentro da programação assíncrona. Conforme Calderon (2018), a programação reativa torna nosso código mais flexível, legível, mantenedor e fácil de escrever.

Como funciona a Programação Reativa?

Para compreendermos, vamos levar em consideração o ReactiveX, a implementação mais famosa do paradigma de Programação Reativa. O ReactiveX é baseado principalmente em duas classes: A classe ***observable,*** que é a fonte de fluxos de dados ou eventos; e a classe que consome (ou reage) os elementos emitidos, ***Observer***.

Observable

Um pacote dos dados recebidos para que possam ser passados de um segmento para outro. Pode ser configurado para que regule quando fornecer dados. Por exemplo, ele pode ser acionado periodicamente ou apenas uma vez em seu ciclo de vida. Existem também várias funções que podem ser usadas para filtrar ou transformar o observável, para que o observador emita apenas determinados dados. Tudo isso é usado em vez de retornos de chamada, o que significa que nosso código se torna mais legível e menos falível.

Observer

O consumo do fluxo de dados emitido pelo ***observable*** pode ser múltiplo para cada item de dados emitido, a ser recebido pelo **observer**. 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():** quando há um elemento no fluxo de dados.
- **on_completed():** quando não vem mais itens, implica o fim da emissão.
- **on_error():** quando há um erro lançado do (também implica fim da emissão).

Não precisamos especificar os três tipos de eventos no código. Podemos escolher quais eventos observar usando os argumentos

nomeados, ou simplesmente fornecendo uma lambda para a função. Normalmente, na produção, queremos fornecer um manipulador para que os erros sejam explicitamente tratados pelo assinante.

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 22 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 22 – Implementação programação reativa.

```
from rx import create

def envia_strings(observer, scheduler):
    observer.on_next("IGTI 1")
    observer.on_next("IGTI 2")
    observer.on_next("IGTI 2")
    observer.on_next("IGTI 4")
    observer.on_next("IGTI 5")
    observer.on_completed()

source = create(envia_strings)

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

```
Recebido: IGTI 1
Recebido: IGTI 2
Recebido: IGTI 2
Recebido: IGTI 4
Recebido: IGTI 5
Finalizado!
```

> Capítulo 4

Capítulo 4. Introdução ao Pygame

O processo de desenvolver jogos é uma das mais recompensadoras áreas de programação, pelo cerne da criatividade e diversão. Neste capítulo reproduziremos o nostálgico jogo Space Invader, para tal, iremos utilizar a biblioteca Pygame, devido ao fato desta biblioteca possuir todas as características e funcionalidades da linguagem de programação Python na construção de jogos interativos.

Desenvolvendo o jogo Space Invader com Pygame

Inicialmente é necessário instalar a biblioteca do pygame, caso esteja utilizando o Jupyter notebook, execute o comando: !pip install pygame. Uma vez instalada, é necessário importar para seu projeto a biblioteca instalada, conforme a imagem a seguir.

Figura 23 – Importação e instalação dos módulos.

Fonte: Adaptado de MATTHES (2016).

Após a importação das bibliotecas, definimos a inicialização do pygame, juntamente com o tamanho da tela em que será apresentada o jogo e os elementos estruturais, como background, sons, títulos e ícones. Cabe salientar que os arquivos (imagens e áudios) devem ter o seu local especificado, neste exemplo os arquivos estão no mesmo local do executável.

Figura 24 – Elementos estruturais do jogo.

```
# Inicializa o módulo pygame
pygame.init()

# Define o tamanho da tela
screen = pygame.display.set_mode((800, 600))

# Plano de fundo
background = pygame.image.load('background.png')

# Sons
mixer.music.load("background.wav")
mixer.music.play(-1)

# Título e icone
pygame.display.set_caption("Space Invader")
icon = pygame.image.load('ufo.png')
pygame.display.set_icon(icon)
```

Fonte: Adaptado de MATTHES (2016).

Criado a estrutura do jogo, definiremos o a jogador e oponente (nave e alien), conforme imagem a seguir.

Figura 25 – Player e oponente.

```
# Nave
playerImg = pygame.image.load('Nave.png')
playerX = 370
playerY = 480
playerX_change = 0

# Nave
enemyImg = []
enemyX = []
enemyY = []
enemyX_change = []
enemyY_change = []
num_of_enemies = 6

for i in range(num_of_enemies):
    enemyImg.append(pygame.image.load('alien.png'))
    enemyX.append(random.randint(0, 736))
    enemyY.append(random.randint(50, 150))
    enemyX_change.append(4)
    enemyY_change.append(40)
```

Fonte: Adaptado de MATTHES (2016).

Observe que os parâmetros “playerX = 0 e playerY = 480”, definem a posição das ordenadas x e y da nave. Quando colocamos os “inimigos” em um vetor, como teremos mais de um por linhas, fazendo-os alterar suas posições ao longo do jogo.

Assim, definiremos agora as posições dos tiros e seus movimentos verticais de 10 em 10 px, e o posicionamento dos scores, juntamente com sua fonte, conforme a imagem a seguir:

Figura 26 – Posicionamento do tiro e definição de fontes.

```
# pronto - você não vê a bala na tela
# tiro- a bala está se movendo

bulletImg = pygame.image.load('bala.png')
bulletX = 0
bulletY = 480
bulletX_change = 0
bulletY_change = 10
bullet_state = "preparado"

# Score
score_value = 0
font = pygame.font.Font('freesansbold.ttf', 32)

#Game Over
over_font = pygame.font.Font('freesansbold.ttf', 64)


score_position_x = 10
score_position_y = 10
```

Fonte: Adaptado de MATTHES (2016).

Agora definiremos as funções de colisão, tiros, jogadores, texto do game over e pontuação, no qual o parâmetro “.blit” é o responsável pelo ato. A função isCollision verifica o posicionamento do vetor de inimigos e jogador e, caso sejam iguais, é disparada a colisão, como pode acompanhar na imagem a seguir.

Figura 27 – Funções de posicionamentos.

```
def show_score(x, y):
    score = font.render("Pontuação : " + str(score_value), True, (255, 255, 255))
    screen.blit(score, (x, y))


def game_over_text():
    over_text = over_font.render("GAME OVER", True, (255, 255, 255))
    screen.blit(over_text, (200, 250))


def player(x, y):
    screen.blit(playerImg, (x, y))


def enemy(x, y, i):
    screen.blit(enemyImg[i], (x, y))


def fire_bullet(x, y):
    global bullet_state
    bullet_state = "fogo"
    screen.blit(bulletImg, (x + 16, y + 10))


def isCollision(enemyX, enemyY, bulletX, bulletY):
    distance = math.sqrt(math.pow(enemyX - bulletX, 2) + (math.pow(enemyY - bulletY, 2)))
    if distance < 27:
        return True
    else:
        return False
```

Fonte: Adaptado de MATTHES (2016).

Agora procederemos a verificação das teclas que serão pressionadas, direita e esquerda para os movimentos da nave e a barra de espaço para disparar o tiro, conforme a seguir.

Figura 28 – Verificação de eventos de entrada.

```
# Rodando o jogo
running = True
while running:

    # RGB = Red, Green, Blue
    screen.fill((0, 0, 0))
    # Background Image
    screen.blit(background, (0, 0))
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        # se o pressionamento de tecla for pressionado, verifique se é direito ou esquerdo
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                playerX_change = -5
            if event.key == pygame.K_RIGHT:
                playerX_change = 5
            if event.key == pygame.K_SPACE:
                if bullet_state is "pronto":
                    bulletSound = mixer.Sound("laser.wav")
                    bulletSound.play()
                    # Obtenha a coordenada x atual da nave espacial
                    bulletX = playerX
                    fire_bullet(bulletX, bulletY)

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
                playerX_change = 0
```

Fonte: Adaptado de MATTHES (2016).

O pygame possui a condição de verificação de eventos, aplicado pela funcionalidade pygame.event.get, que possibilita o monitoramento de entrada de informações pelo teclado.

Desenvolvido isso, cria-se a verificação do movimento da nave juntamente com o posicionamento do inimigo, caso haja colisão.

Figura 29 – Verificação de colisão.

```
playerX += playerX_change
if playerX <= 0:
    playerX = 0
elif playerX >= 736:
    playerX = 736

# Movimento do inimigo
for i in range(num_of_enemies):

    # Game Over
    if enemyY[i] > 440:
        for j in range(num_of_enemies):
            enemyY[j] = 2000
        game_over_text()
        break

    enemyX[i] += enemyX_change[i]
    if enemyX[i] <= 0:
        enemyX_change[i] = 4
        enemyY[i] += enemyY_change[i]
    elif enemyX[i] >= 736:
        enemyX_change[i] = -4
        enemyY[i] += enemyY_change[i]

    # Colisão
    collision = isCollision(enemyX[i], enemyY[i], bulletX, bulletY)
    if collision:
        explosionSound = mixer.Sound("explosion.wav")
        explosionSound.play()
        bulletY = 480
        bullet_state = "pronto"
        score_value += 1
        enemyX[i] = random.randint(0, 736)
        enemyY[i] = random.randint(50, 150)

    enemy(enemyX[i], enemyY[i], i)
```

Fonte: Adaptado de MATTHES (2016).

Concluso essa etapa, necessita-se apenas a inclusão do movimento do projetil, e a chamada do posicionamento do player e score, conforme conclui-se na Figura 30.

Figura 30 – Posicionamento de pontuação e disparo.

```
# Mivimento do tiro
if bulletY <= 0:
    bulletY = 480
    bullet_state = "pronto"

if bullet_state is "fogo":
    fire_bullet(bulletX, bulletY)
    bulletY -= bulletY_change

player(playerX, playerY)
show_score(score_position_x, score_position_y)
pygame.display.update()
```

Fonte: Adaptado de MATTHES (2016).

Referências

BROWNLEE, Janson. *Master Machine Learning Algorithms:* Discover How They Works and Implament Them From Scratch. V1. Melbourne, Australia. 2016.

MATTHES, Eric. *Curso Intensivo de Python*. Uma introdução Prática e Baseada em Projetos à Programação. Novatec, 2016.