Atividade Prática

Disciplina	**MAP Bootcamp**
Atividade	**Rede Neural – Reconhecimento de Números**

1. Objetivos

Para iniciar os estudos de redes neurais, iremos fazer o uso de uma base de dados numéricos com 1797 amostras divididas dentro do intervalo de 0 a 9. A ideia é que vamos treinar nosso modelo a aprender as características de cada número e posteriormente conseguir fazer o reconhecimento de um número escrito a mão para teste.

2. Enunciado

Nesse exemplo em particular vamos testar dois modelos de processamento para identificação de imagens, pois simularemos aqui um erro bastante comum nesse meio que se dá quando temos que fazer identificação a partir de uma base de dados muito pequena, muito imprecisa que é a execução do modelo confiando em uma margem de precisão relativamente baixa.

Digits Dataset.py

```
1 from sklearn import datasets
2
```

Como sempre, damos início ao processo criando um arquivo, nesse caso de nome Digits Dataset.py, em seguida realizamos as primeiras importações necessárias, por hora, importaremos o módulo datasets da biblioteca sklearn. Assim você pode deduzir que inicialmente usaremos uma biblioteca de exemplo para darmos os primeiros passos.

```
3 base = datasets.load_digits()
4 entradas = base.data
5 saidas = base.target
```

Logo após criamos nossa variável de nome base, que por sua vez recebe todos os

dados do digits dataset por meio da função .load_digits( ) do módulo datasets. Em seguida criamos nossa variável entradas que recebe os atributos previsores por meio da instância base.data, da mesma forma criamos nossa variável saidas que recebe os dados alvo instanciando base.targets.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
saidas	int32	(1797,)	[0 1 2 ... 8 9 8]

Se o processo de importação ocorreu como esperado foram criadas as devidas variáveis, note que aqui temos a mesma quantidade de dados para nossas entradas e saídas, uma vez que faremos o processo de aprendizagem de máquina para que nosso interpretador aprenda a identificar e classificar números escritos a mão. Importante salientar que este número de 1797 amostras é muito pouco quando o assunto é classificação de imagens, o que acarretará problemas de imprecisão de nosso modelo quando formos executar alguns testes. O valor 64, da segunda coluna de nossa variável entradas diz respeito ao formato em que tais dados estão dispostos, aqui nesse dataset temos 1797 imagens que estão em formato 8x8 pixels, resultando em 64 dados em linhas e colunas que são referências a uma escala de preto e branco.

```
7 print(base.data[0])
8
```

Para ficar mais claro, por meio da função print( ) passando como parâmetro a amostra de número 0 de nossa base de dados, poderemos ver via console tal formato.

```
In [ ]: print(base.data[0])
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

Via console podemos finalmente visualizar tal formato. Cada valor é referente a uma posição dessa matriz 8x8, onde os valores estão dispostos em um intervalo de 0 até 16, onde 0 seria branco e 16 preto, com valores intermediários nessa escala para cada pixel que compõe essa imagem de um número escrito a mão.

```
9 print(base.images[0])
10
```

Da mesma forma podemos visualizar o formato de imagem salvo na própria base de dados. O método é muito parecido com o anterior, porém instanciamos a amostra número 0 de imagens.

```
In [ ]: print(base.images[0])
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

Via console podemos ver uma matriz 8x8 onde inclusive com o olhar um pouco mais atento podemos identificar que número é esse, referente a primeira amostra da base de dados.

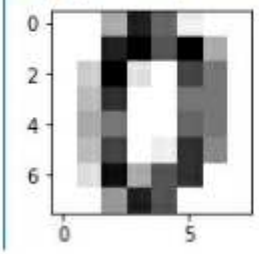
```
11 import matplotlib.pyplot as plt
12
```

Partindo para o que interessa, vamos fazer a plotagem deste número. Para isto, como de praxe, importamos a matplotlib.pyplot.

```
13 plt.figure(figsize = (2,2))
14 plt.imshow(base.images[0],
15            cmap = plt.cm.gray_r)
```

Em seguida, usando apenas o necessário para exibir nossa amostra, criamos essa pequena estrutura onde a função .figure( ) fará a leitura e plotagem com um tamanho definido em 2x2 (lembrando que isso não é pixels, apenas uma referência métrica interna a matplotlib). Na sequência a função .imshow( ) fica responsável por exibir nossa amostra parametrizada também em uma escala de cinza.

```
Out[5]: <matplotlib.image.AxesImage at 0x26db21af668>
```



Não havendo nenhum erro de sintaxe é finalmente exibido em nosso console o número 0, referente a primeira amostra de nossa base de dados. Note a baixa qualidade da imagem, tanto a baixa qualidade quanto o número relativamente pequeno de amostras são um grande problema quando se diz respeito em classificação de imagens. Revisando, toda imagem, internamente, para nosso sistema operacional e seus interpretadores, é um conjunto de referências a pixels e suas composições, aqui inicialmente estamos trabalhando com imagens de pequeno tamanho e em escala de cinza, mas independente disso, toda imagem possui um formato, a partir deste, uma matriz com dados de tonalidades de cinza (ou de vermelho, verde e azul no caso de imagens coloridas), que em algum momento será convertido para binário e código de máquina. A representação da imagem é uma abstração que o usuário vê em tela, desta disposição de pixels compondo uma figura qualquer.

```
17 from sklearn.model_selection import train_test_split
18 etreino,eteste,streino,steste = train_test_split(entradas,
19                                                  saidas,
20                                                  test_size = 0.1,
21                                                  random_state = 2)
```

Dando sequência, mesmo trabalhando sobre imagens podemos realizar aqueles testes de performance que já são velhos conhecidos nossos. Inicialmente importamos train_test_split, criamos suas respectivas variáveis, alimentamos a ferramenta com dados de entrada e de saída, assim como separamos as amostras em 90% para treino e 10% para teste, por fim, definimos que o método de separação das amostras seja randômico não pegando dados com proximidade igual ou menos a 2 números.

```
23 from sklearn import svm
24 classificador = svm.SVC()
25 classificador.fit(etreino,streino)
26 previsor = classificador.predict(eteste)
```

Separadas as amostras, hora de criar nosso modelo inicial de classificador, aqui neste exemplo inicialmente usaremos uma ferramenta muito usada quando o assunto é identificação e classificação de imagens chamada Suport Vector Machine, esta ferramenta está integrada na biblioteca sklearn e pode ser facilmente importada como importamos outras ferramentas em outros momentos. Em seguida criamos nosso classificador que inicializa a ferramenta svm.SVC( ), sem parâmetros mesmo, e codificada nessa sintaxe. Logo após por meio da função .fit( ) alimentamos nosso classificador com os dados de etreino e streino. Por fim, criamos nosso primeiro previsor, que com base nos dados encontrados em classificador fará as devidas comparações e previsões com os dados de eteste.

```
28 from sklearn import metrics
29 margem_acerto = metrics.accuracy_score(steste, previsor)
```

Criado o modelo, na sequência como de costume importamos metrics e criamos nossa variável responsável por fazer o cruzamento dos dados e o teste de performance do modelo. Então é criada a variável margem_acerto que por sua vez executa a ferramenta .accuracy_score( ) tendo como parâmetros os dados de steste e os encontrados em previsor.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
eteste	float64	(180, 64)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 1. ... 15. 1. 0.]
etreino	float64	(1617, 64)	[[0. 0. 10. ... 0. 0. 0.] [0. 0. 0. ... 16. 9. 0.]
margem_acerto	float64	1	0.5611111111111111
previsor	int32	(180,)	[4 5 9 ... 5 0 5]
saidas	int32	(1797,)	[0 1 2 ... 8 9 8]
steste	int32	(180,)	[4 0 9 ... 0 0 4]
streino	int32	(1617,)	[0 6 5 ... 1 1 5]

Selecionado e executado todo bloco de código anterior, via explorador de variáveis podemos nos ater aos dados apresentados nesses testes iniciais. O mais importante deles por hora, margem_acerto encontrou em seu processamento uma taxa de precisão de apenas 56%. Lembre-se que comentei em parágrafos anteriores, isto se dá por dois

motivos, base de dados pequena e imagens de baixa qualidade. De imediato o que isto significa? Nossos resultados não são nenhum pouco confiáveis nesse modelo, até podemos realizar as previsões, porém será interessante repetir os testes algumas vezes como prova real, como viés de confirmação das respostas encontradas dentro dessa margem de acertos relativamente baixa.

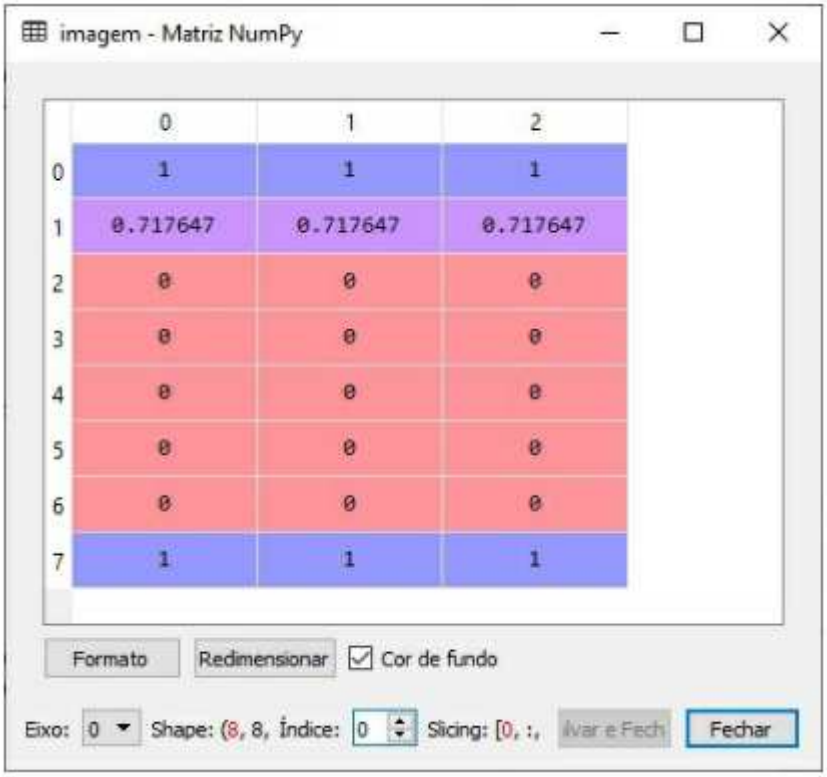
Leitura e reconhecimento de uma imagem.

```
31 import numpy as np
32 import matplotlib.image as mimg
```

Uma vez criado o modelo, mesmo identificada sua baixa precisão, podemos realizar testes reais para o mesmo, fazendo a leitura de um número escrito a mão a partir de uma imagem importada para nosso código. Como sabemos o número em questão esse processo pode ser considerado válido como teste, até mesmo com a nossa atual margem de acerto. Anteriormente havíamos importado a ferramenta pyplot da matplotlib, dessa vez a ferramenta que usaremos é a image, da própria matplotlib, o processo de importação é exatamente igual ao de costume, dessa vez referenciamos a ferramenta como mimg. Por fim, note que também importamos a biblioteca numpy.

```
34 imagem = mimg.imread('num2.png')
35 print(imagem)
```

Dando sequência, criamos nossa variável de nome imagem que recebe como atributo o conteúdo importado de num2.png por meio da função .imread( ). A seguir por meio do comando print( ) imprimimos imagem em nosso console.



Via explorador de variáveis podemos abrir a variável imagem e seu conteúdo, muita atenção ao rodapé da janela, uma vez que estamos visualizando aqui uma matriz, estes dados dispostos nessa grade 8x8 é apenas a informação de 1 pixel da imagem, por meio dos botões da parte inferior da janela podemos navegar pixel a pixel.

```
   ...: print(imagem)
[[[1.        1.        1.       ]
  [0.7176471 0.7176471 0.7176471]
  [0.        0.        0.       ]
  [0.        0.        0.       ]
  [0.        0.        0.       ]
  [0.        0.        0.       ]
  [0.        0.        0.       ]
  [1.        1.        1.       ]]

 [[1.        1.        1.       ]
  [0.        0.        0.       ]
  [1.        1.        1.       ]
  [1.        1.        1.       ]
```

Via console podemos ver em forma de lista todos dados também referentes a cada pixel. Raciocine que por hora estes formatos de dados nos impedem de aplicar qualquer tipo de função sobre os mesmos, sendo assim, precisamos fazer a devida conversão desses dados referentes a cada pixel para uma matriz que por sua vez contenha dados em forma de uma matriz, mais especificamente uma array do tipo

numpy.

```
37 def rgb2gray(rgb):
38     img_array = np.dot(rgb[...,:3],[0.299,0.587,0.114])
39     img_array = (16 - (img_array * 16)).astype(int)
40     img_array = img_array.flatten()
41     return img_array
```

Poderíamos criar uma função do zero que fizesse tal processo, porém vale lembrar que estamos trabalhando com Python, uma linguagem com uma enorme comunidade de desenvolvedores, em função disso com apenas uma rápida pesquisa no Google podemos encontrar uma função já pronta que atende nossos requisitos. Analisando a função em si, note que inicialmente é criada a função rgb2gray( ) que recebe como parâmetro a variável temporária rgb, uma vez que essa será substituída pelos arquivos que faremos a leitura e a devida identificação. Internamente temos uma variável de nome img_array, inicialmente ela recebe o produto escalar de todas as linhas e das 3 colunas do formato anterior (matriz de pixel) sobre os valores 0.299, 0.587 e 0.114, uma convenção quando o assunto é conversão para rgb (escala colorida red grey blue). Em seguida é feita a conversão desse valor resultado para tipo inteiro por meio da função .astype( ) parametrizada com int. Logo após é aplicada a função .flatten( ) que por sua vez irá converter a matriz atual por uma nova indexada no formato que desejamos, 64 valores em um intervalo entre 0 e 16 (escala de cinza) para que possamos aplicar o devido processamento que faremos a seguir.

```
43 rgb2gray(imagem)
44
```

Para ficar mais claro o que essa função faz, chamando-a e passando como parâmetro nossa variável imagem (que contém o arquivo lido anteriormente num2.png) podemos ver o resultado de todas conversões em nosso console.

```
In [  ]: rgb2gray(imagem)
Out[  ]:
array([ 0,  4, 16, 16, 16, 16, 16,  0,  0, 16,  0,  0,  0,  0,  0,  0,  0,
       16,  0,  0,  0,  0,  0,  0,  0,  7, 16, 16, 16,  6,  0,  0,  0,  0,
        0,  0,  0, 16,  0,  0,  0, 16,  0,  0,  0, 16, 16,  0,  0,  0, 16,
       16, 16, 16,  0,  0,  0,  0,  0, 16, 16,  0,  0,  0])
```

Aplicadas as conversões sobre num2.png que estava instanciada em nossa variável imagem, podemos notar que temos uma array com 64 referências de tonalidades de cinza em uma posição organizada para o devido reconhecimento.

```
identificador = svm.SVC()
identificador.fit(entradas,saidas)
previsor_id = identificador.predict([rgb2gray(imagem)])
print(previsor_id)
```

Criada toda estrutura inicial de nosso modelo, responsável por fazer até então a leitura e a conversão do arquivo de imagem para que seja feito o reconhecimento de caractere, hora de finalmente realizar os primeiros testes de reconhecimento e identificação. Aqui, inicialmente usaremos uma ferramenta chamada Suport Vector Machine. Esta ferramenta por sua vez usa uma série de métricas de processamento vetorial sobre os dados nela alimentados. Uma SVM possui a característica de por meio de aprendizado supervisionado (com base em treinamento via uma base de dados), aprender a reconhecer padrões e a partir destes realizar classificações. Inicialmente criamos nossa variável de nome identificador, que por sua vez inicia o módulo SVC da ferramenta svm. Em seguida por meio da função .fit( ) é feita a alimentação da mesma com os dados de entrada e saída, lembrando que nesse modelo estamos tratando de atributos previsores e das devidas saídas, já que estamos treinando nossa máquina para reconhecimento de imagens. Logo após criamos nossa variável de nome previsor_id, que por sua vez via função .predict( ) recebe nossa função rgb2gray( ) alimentada com nosso arquivo num2.png. Por fim, simplesmente imprimimos em nosso console o resultado obtido em previsor_id.

```
In [  ]: identificador = svm.SVC()
    ...: identificador.fit(entradas,saidas)
    ...: previsor_id = identificador.predict([rgb2gray(imagem)])
    ...: print(previsor_id)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\svm\base.py:193:
FutureWarning: The default value of gamma will change from 'auto' to 'scale' in
version 0.22 to account better for unscaled features. Set gamma explicitly to
'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
[5]
```

Selecionando e executando o bloco de código anterior podemos acompanhar via console o seu processamento e que, apesar de nossa margem de acerto relativamente baixa no processo de aprendizado, ele identificou corretamente que o número passado para identificação (nosso arquivo num2.png) era o número 5 escrito a mão. Da mesma forma podemos realizar mais testes de leitura, desde que alimentando nosso código

com imagens no mesmo padrão. Porém, para não nos estendermos muito nesse capítulo inicial, vamos realizar uma segunda verificação independente da SVM, para termos um segundo parâmetro de confirmação. Já sabemos que a leitura e identificação de nosso arquivo foi feita corretamente, mas caso houvesse ocorrido um erro de identificação esse segundo teste poderia nos ajudar de alguma forma. Em outro momento já usamos do modelo de regressão logística para classificar dados quanto seu agrupamento e cruzamento de dados via matriz, aqui nossas conversões vão de um arquivo de imagem para uma array numpy, sendo possível aplicar o mesmo tipo de processamento.

```
50 from sklearn.linear_model import LogisticRegression
51 logr = LogisticRegression()
52 logr.fit(etreino,streino)
53 previsor_logr = logr.predict(eteste)
54 acerto_logr = metrics.accuracy_score(steste, previsor_logr)
55 print(acerto_logr)
```

Como não havíamos usado nada deste modelo em nosso código atual, realizamos a importação da ferramenta LogisticRegression do módulo linear_model da biblioteca sklearn. Em seguida criamos uma variável logr que inicializa a ferramenta, na sequência alimentamos a ferramenta com os dados de etreino e streino, por fim criamos uma variável previsor_logr que faz as previsões sobre os dados de eteste. Também criamos uma variável de nome acerto_logr que mostra a taxa de precisão cruzando os dados de steste e previsor_logr, por fim, exibindo no console este resultado.

Nome	Tipo	Tamanho	Valor
acerto_logr	float64	1	0.9277777777777778
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
eteste	float64	(180, 64)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 1. ... 15. 1. 0.]
etreino	float64	(1617, 64)	[[0. 0. 10. ... 0. 0. 0.] [0. 0. 0. ... 16. 9. 0.]
imagem	float32	(8, 8, 3)	[[[1. 1. 1.] [0.7176471 0.7176471 0.7176471] .. ..
margem_acerto	float64	1	0.5611111111111111
previsor	int32	(180,)	[4 5 9 ... 5 0 5]
previsor_id	int32	(1,)	[5]

Via explorador de variáveis podemos ver que o modelo de regressão logística, aplicado em nossa base de dados, obteve uma margem de acertos de 92%, muito maior aos

56% da SVM que em teoria deveria retornar melhores resultados por ser especializada em reconhecimento e identificação de padrões para imagens. Novamente, vale lembrar que isso se dá porque temos uma base de dados muito pequena e com imagens de baixa qualidade, usadas apenas para fins de exemplo, em aplicações reais, tais margens de acerto tendem a ser o oposto do aqui apresentado.

```
57 regressor = LogisticRegression()
58 regressor.fit(entradas,saidas)
59 previsor_regl = regressor.predict([rgb2gray(imagem)])
60 print(previsor_regl)
```

Para finalizar, criamos nosso regressor por meio de uma variável homônima que inicializa a ferramenta, na sequência é alimentada com os dados contidos em entradas e saidas. Também criamos a variável previsor_regl que realiza as devidas previsões com base na nossa função rgb2gray( ) alimentada com nosso arquivo instanciado num2.png. Concluindo o processo imprimimos em nosso console o resultado obtido para previsor_regl.

```
In [  ]: regressor = LogisticRegression()
    ...: regressor.fit(entradas,saidas)
    ...: previsor_regl = regressor.predict([rgb2gray(imagem)])
    ...: print(previsor_regl)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:
432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:
469: FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Spec:
the multi_class option to silence this warning.
  "this warning.", FutureWarning)
[5]
```

Via console podemos acompanhar o processamento e como esperado, a previsão da identificação do arquivo, reconhecido como número 5 escrito a mão.

Código Completo:

```
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn import metrics
import numpy as np
import matplotlib.image as mimg
from sklearn.linear_model import LogisticRegression

base = datasets.load_digits()
entradas = base.data
saidas = base.target

plt.figure(figsize = (2,2))
plt.imshow(base.images[0],
           cmap = plt.cm.gray_r)

etreino,eteste,streino,steste = train_test_split(entradas,
                                                 saidas,
                                                 test_size = 0.1,
                                                 random_state = 2)
classificador = svm.SVC()
classificador.fit(etreino,streino)
previsor = classificador.predict(eteste)
margem_acerto = metrics.accuracy_score(steste, previsor)
```

```
imagem = mimg.imread('num2.png')

def rgb2gray(rgb):
    img_array = np.dot(rgb[...,:3],[0.299,0.587,0.114])
    img_array = (16 - (img_array * 16)).astype(int)
    img_array = img_array.flatten()
    return img_array

rgb2gray(imagem)

identificador = svm.SVC()
identificador.fit(entradas,saidas)
previsor_id = identificador.predict([rgb2gray(imagem)])
print(previsor_id)

logr = LogisticRegression()
logr.fit(etreino,streino)
previsor_logr = logr.predict(eteste)
acerto_logr = metrics.accuracy_score(steste, previsor_logr)
print(acerto_logr)

regressor = LogisticRegression()
regressor.fit(entradas,saidas)
previsor_regl = regressor.predict([rgb2gray(imagem)])
print(previsor_regl)
```

3. Conclusão

Rodamos um programa que faz o reconhecimento de caracteres pelo uso de redes neurais.