Nome : Arthur Felipe Reis Souza.
Data : 29/03/2024

Parte 1

Os dados foram gerados aleatóriamente, através da biblioteca numpy. Aamostra de dados 1 contém a média centralizada no ponto (2,2), e a amostra de dados 2 contém a média centralizada no ponto (4,4). A correlação entre ambas as amostras é nula, isso significa que os dados de uma amostra não tem uma relação com os dados da outra amostra, implicando que as variações de uma amostra não interferem na variação da outra amostra. Para evidenciar ambos os conjuntos, foi-se utilizado o desvio padrão 0.2 para ambas as amostras.

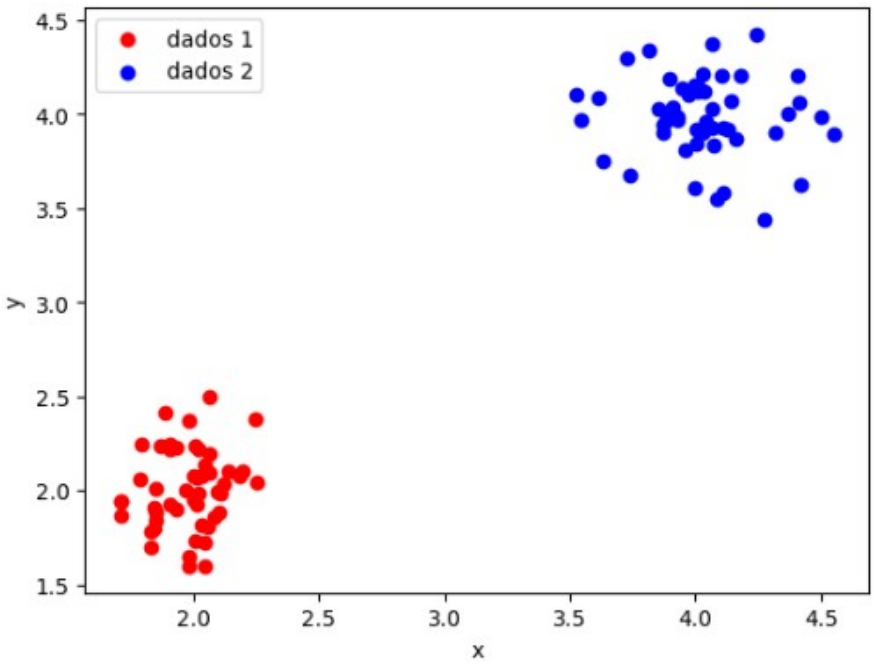
A imagem abaixo mostra o conjunto de dados gerados pela biblioteca numpy e plotados pela biblioteca matplotlib.

[15]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[16]:
```
xc1 = (0.2*np.random.normal(size = 100) + 2).reshape(-1, 2)
xc2 = (0.2*np.random.normal(size = 100) + 4).reshape(-1, 2)
```

[17]:
```
plt.scatter(xc1[:, 0], xc1[:, 1], color = 'red', label = 'dados 1')
plt.scatter(xc2[:, 0], xc2[:, 1], color = 'blue', label = 'dados 2')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.plot()
```

[17]:
```
[]
```



O modelo neuronal “perceptron” irá ser utilizado no exercício para separar a amostra de dados 1 da amostra de dados 2. O perceptron simples é um modelo neuronal que realiza uma separação linear nos dados de amostra. Quando utilizado em rede, chamamos esse tipo de rede de Multi Layer Perceptron. As redes MLP’s são consideradas aproximadoras universais de funções, e a mesma consegue realizar separações mais complexas do que as separações lineares que um perceptron simples pode realizar.

A imagem abaixo evidencia o treino do modelo neuronal perceptron.

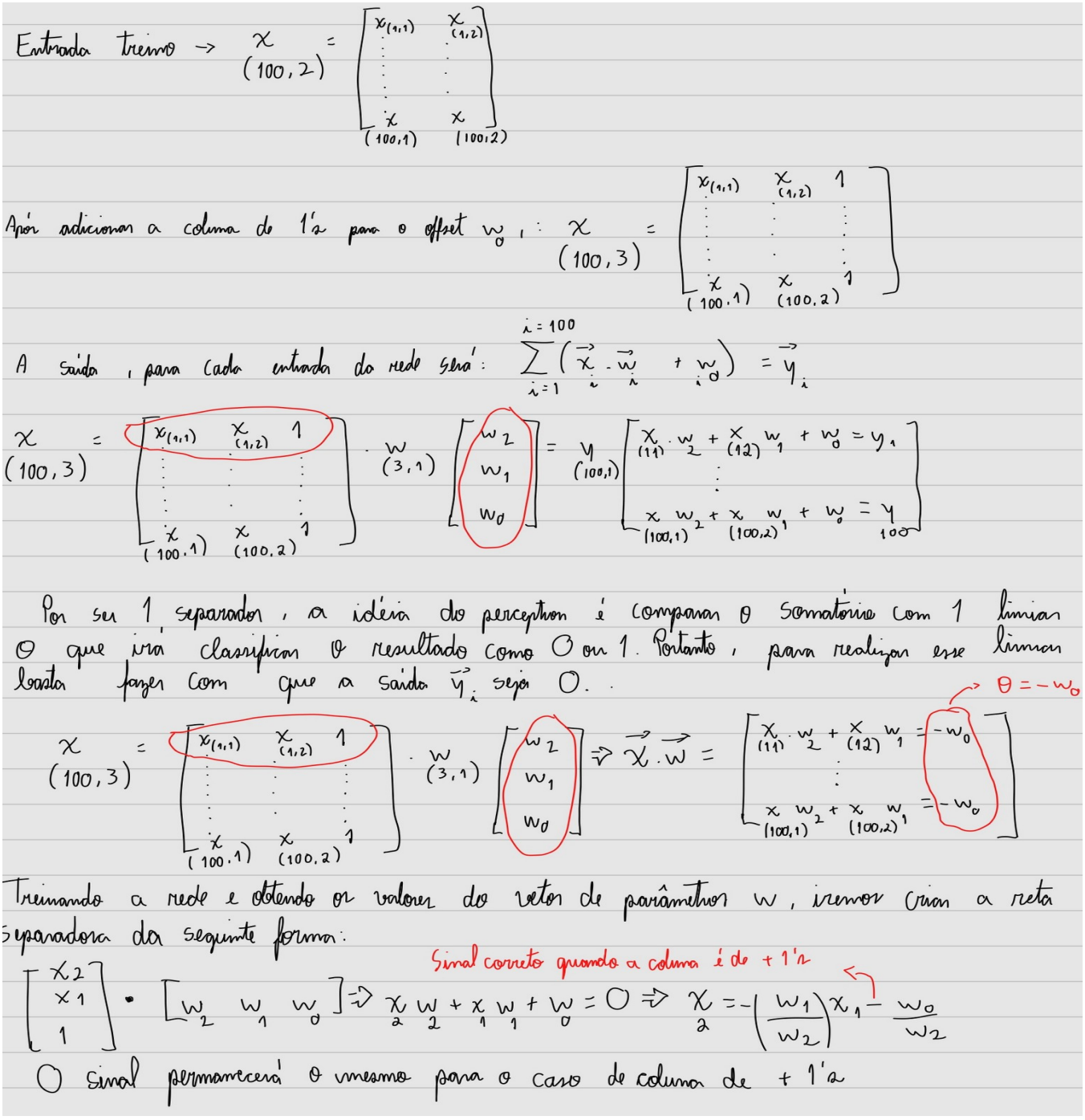
[19]:
```
def train_perceptron(x_inp : np.array, yi : np.array, learn_rate : float, tol : float, max_epochs : int, control_var : bool): # Fazendo o tr
    try:
        n_rows = x_inp.shape[0]
        n_cols = x_inp.shape[1]
    except Exception as error:
        if error == "IndexError":
            print("Now, you don't have cols, so we will change it...\n")
            n_cols = 1
        else:
            print(f"The error {error} is hapenning \n")
            print("Breaking the program...")
            sys.exit()
    finally:
        if control_var == True:
            w = np.random.uniform(size = n_cols + 1) - 0.5 # Iniciando o vetor de pesos com pesos aleatórios e terá 1 posição a mais.
            x_inp = np.column_stack([x_inp, np.ones_like(x_inp[:, 0])])
        else:
            w = np.random.uniform(size = n_cols) - 0.5 # Inicializando os pesos com valores aleatórios e não terá 1 posição a mais.

        n_epochs = 0
        err_epoch = tol + 1
        while ((n_epochs < max_epochs) and (tol < err_epoch)):
            err_grad = 0
            rand_order = np.random.permutation(n_rows)
            for i in range(n_rows):
                i_rand = rand_order[i]
                x_rand_val = x_inp[i_rand, :]
                y_hat = 1 if np.dot(x_rand_val, w) >=0 else 0
                err = (yi[i_rand] - y_hat)
                dw = (learn_rate*err*x_inp[i_rand,])
                w = w + dw
                err_grad = err_grad + err**2
            n_epochs += 1
        return w
```

[20]:
```
w = train_perceptron(xin, y_real, 0.01, 0.1, 10, True)
```

Os parâmetros da rede são : learning rate valendo 0.01, a tolerância valendo 0.1, e o número de épocas sendo 10.

O perceptron da função acima contém uma função de ativação degrau, que irá separar o espaço em regiões com o valor 0 ou com o valor 1. É válido citar que, pela função degrau não ser contínua, métodos de otimização como gradiente descendente não podem ser utilizados para otimizar a atualização dos pesos w de maneira iterativa. Uma função de ativação muito comum é a função sigmoidal, que é uma função contínua que serve como uma aproximação da função degrau e permite o uso do gradiente descendente.

Após o treinamento ser concluído, um vetor de parâmetros w foi obtido. Como a coluna de 1's extra teve o valor positivo, a equação final da reta que irá separar os conjutos de dados irá manter o valor de wo positivo. A lógica abaixo mostra a dedução da equação da reta que separa ambos os conjuntos de dados :

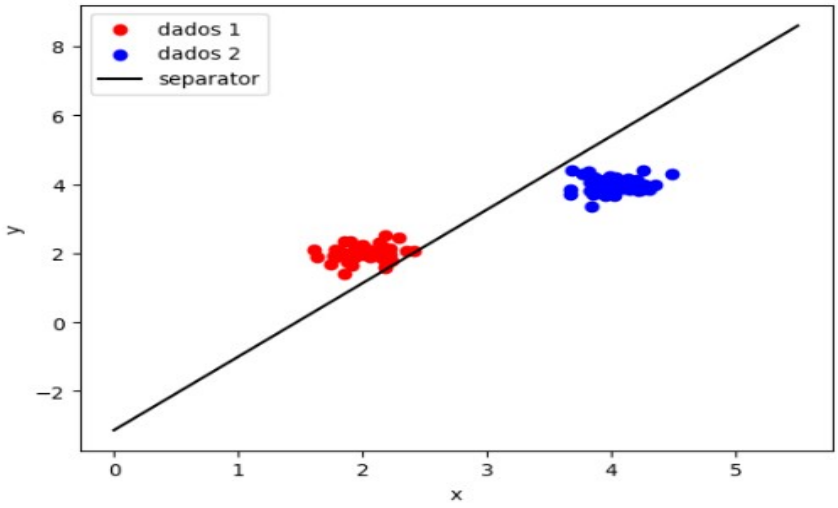


A imagem abaixo mostra o conjunto de dados sendo separados pela reta. Tudo que está a esquerda da reta será classificado como 0 e tudo que está a direita da reta será classificado como 1.

[63]:
```
t = np.linspace(start = 0, stop = 5.5, num = 200)
y = -(w[0] / w[1])*t - (w[2] / w[1])
```

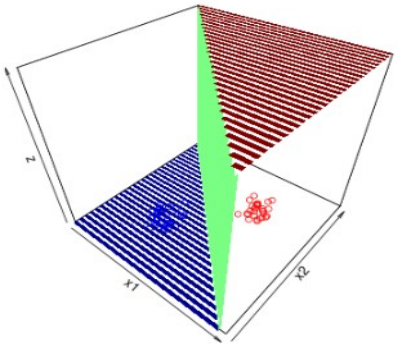
[64]:
```
plt.scatter(xc1[:, 0], xc1[:, 1], color = 'red', label = 'dados 1')
plt.scatter(xc2[:, 0], xc2[:, 1], color = 'blue', label = 'dados 2')
plt.xlabel('x')
plt.ylabel('y')
plt.plot(t, y, color = 'black', label = 'separator')
plt.legend()
plt.plot()
```

[64]:
```
[]
```



A parte final, consiste mostrar o hiperplano que separará ambas as classes. No fito de ampliar as técnicas de programação, foi-se utilizada a biblioteca rpy2 para fazer a conexão do código R com o código python no jupyter notebook. O código em R mostra como ficou a separação das 2 classes por meio de um hiperplano.

[106]:
```
%%R
library('plot3D')
xc1 <- matrix(0.3*rnorm(60) + 2, ncol = 2)
xc2 <- matrix(0.3*rnorm(60) + 4, ncol = 2)
seqx1x2 <- seq(0, 6, 0.2) # gera valores de 0 até 6 com passo de 0.2.
npgrid <- length(seqx1x2) # Pegando o tamanho da sequencia de valores gerados.
M <- matrix(nrow = npgrid, ncol = npgrid) # Criando 1 matriz quadrada.
ci <- 0
w <- as.matrix(c(1 ,1, 6)) # A matriz com os parâmetros
# Esses 2 valores de for alinhados gera os pares do plano (x1, x2) que irei plotar.
for (x1 in seqx1x2){
  ci <- ci + 1
  cj <- 0
  for (x2 in seqx1x2){
    cj <- cj + 1
    xin <- as.matrix(cbind(x1, x2, -1))
    M[ci, cj] <- 1*((xin %*% w) >= 0)
  }
}

ribbon3D(seqx1x2, seqx1x2, xlab = 'x1', ylab = 'x2', xlim = c(0, 6), ylim = c(0, 6), M, colkey =  F)
scatter3D(xc1[, 1], xc1[, 2], xlab = 'x1', ylab = 'x2', matrix(0, nrow = dim(xc1)[1]), add = T, col =  'blue', colkey = F)
scatter3D(xc2[, 1], xc2[, 2], xlab = 'x1', ylab = 'x2', matrix(0, nrow = dim(xc1)[1]), add = T, col =  'red', colkey = F)
```



Parte 2

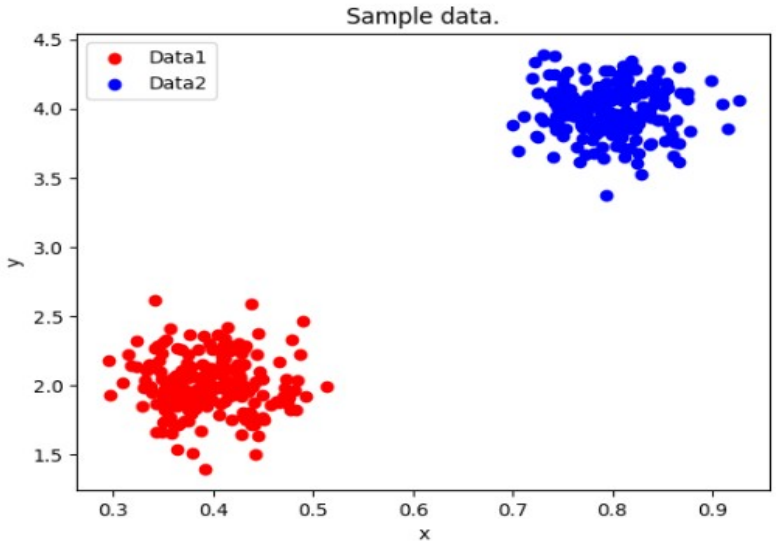
Agora, no fito de avaliar o desempenho do modelo através de um teste, foram geradas 400 pontos, sendo 200 de cada um dos 2 conjuntos. Dessas 400 amostras 70% são utilizadas para treino e 30% para teste. É válido comentar que, não se usa apenas dados de teste para avaliar um modelo. Esse modelo, antes de ser testado, tem que passar por uma validação, portanto é comum que parte dos dados de teste sejam utilizados para validar o modelo antes de testá-lo.

A imagem abaixo mostra a geração dos dados que serão utilizados para treino e para teste.

[4]:
```
xc1 = ((0.2*np.random.normal(size = 400) + 2).reshape(-1, 2))
xc2 = ((0.2*np.random.normal(size = 400) + 4).reshape(-1, 2))
```

[13]:
```
plt.scatter(0.2*xc1[:, 0],xc1[:, 1], color = 'red', label = 'Data1')
plt.scatter(0.2*xc2[:, 0],xc2[:, 1], color = 'blue', label = 'Data2')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title("Sample data.")
plt.plot()
```

[13]:
```
[]
```



Serão 140 dados de treino de cada uma das amostras, totalizando-se então 280 dados de treino. A imagem abaixo evidencia a separação dos dados de treino e os dados de teste. Antes de alimentar o perceptron simples com os dados de treino, embaralhamos o conjunto de dados para dar uma aleatoriedade na seleção das entradas.

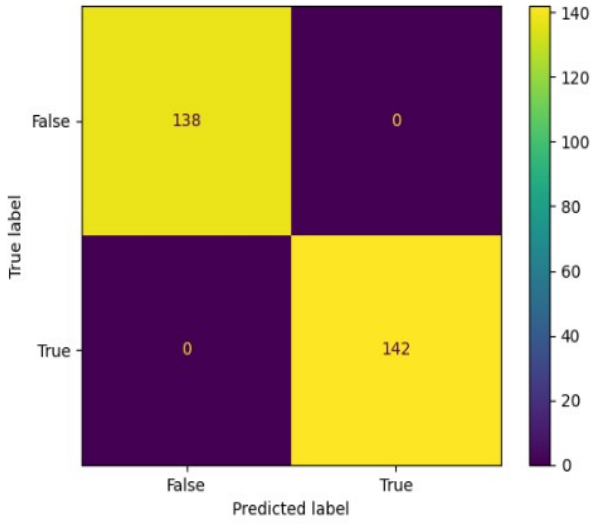
Embaralhando os indices e alterando a ordem para melhorar o treinamento.

:
```
zeros = np.zeros((xc1.shape[0]))
ones = np.ones((xc2.shape[0]))
y_real = np.concatenate((zeros, ones), axis = 0)
xin = np.vstack([xc1, xc2])
indices = np.arange(xin.shape[0])
np.random.shuffle(indices)
xin = xin[indices]
y_real = y_real[indices]
```

:
```
n_train = 280 # Serão 280 dados para teste.
x_train = xin[:n_train,]
y_train = y_real[:n_train]
x_test = xin[n_train:,]
y_test = y_real[n_train:]
```

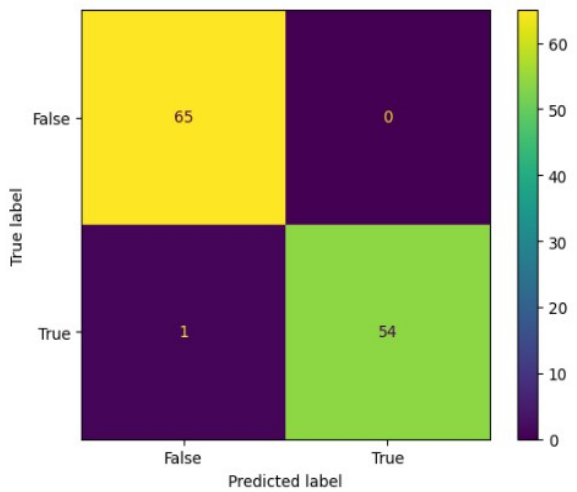
Após treinar o perceptron, foi criada uma matriz de confusão para fazer uma análise da performance do modelo sobre os dados utilizados no treinamento. A figura abaixo mostra a matriz de confusão e a acurácia do modelo sobre os dados de treino.

```
[98]: retlist = train_perceptron(x_train, y_train, 0.01, 0.1, 10, True)
      w = retlist[0]

[99]: yhatrain = yperceptron(x_train, w, True)
      acuracy = 1 - (np.transpose(y_train - yhatrain) @ (y_train - yhatrain)) / 280
      print(acuracy)

      1.0

[100]: from sklearn import metrics
       confusion_matrix = metrics.confusion_matrix(y_train, yhatrain)
       cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
       cm_display.plot()
       plt.show()
```



Analisando apenas os dados de treino, é possível afirmar que ele acertou todas as classificações corretamente para essa configuração de hiperparâmetros. Após fazer a mesma coisa para os dados de teste, que totalizam 30% do conjunto amostral total, obtemos :

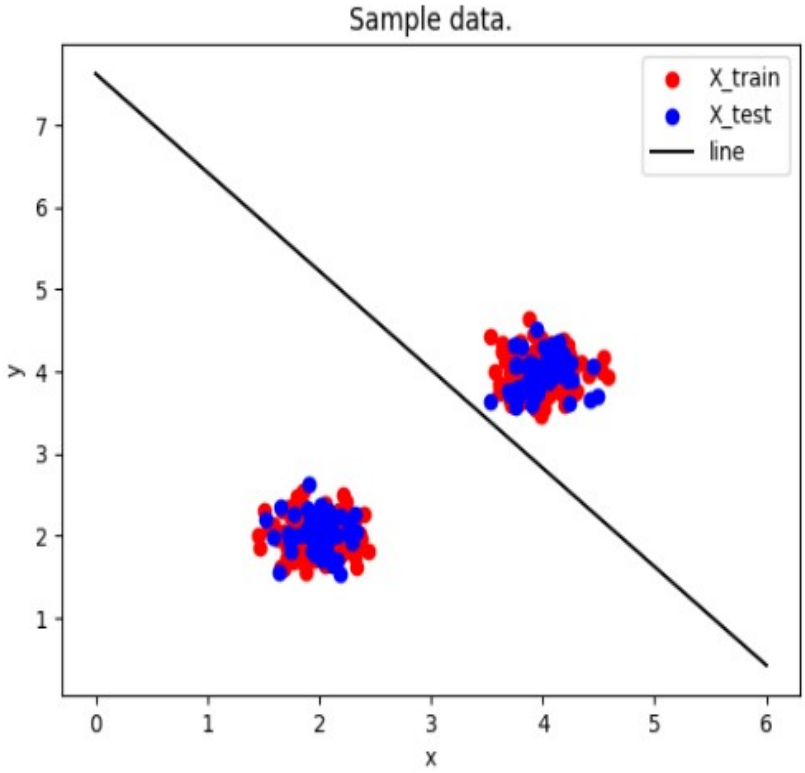
```
[129]: yhatest = yperceptron(x_test, w, True)
       acuracy = 1 - (np.transpose(y_test - yhatest) @ (y_test - yhatest)) / 120
       print(acuracy)

       0.9916666666666667

[130]: confusion_matrix = metrics.confusion_matrix(y_test, yhatest)
       cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
       cm_display.plot()
       plt.show()
```



A figura abaixo mostra a reta separadora, que separa corretamente ambos os conjuntos, tanto para os dados de treino (vermelho) quanto para os dados de teste (azul).

[232]:
```
t = np.linspace(start = 0, stop = 6, num = 200)
y = -(w[0]/w[1])*t - (w[2]/w[1])
plt.scatter(x_train[:, 0],x_train[:, 1], color = 'red', label = 'X_train')
plt.scatter(x_test[:, 0],x_test[:, 1], color = 'blue', label = 'X_test')
plt.plot(t, y, color = 'black', label = 'line')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title("Sample data.")
plt.plot()
w
```

[232]:
```
array([ 0.00773407,  0.00644657, -0.04913004])
```

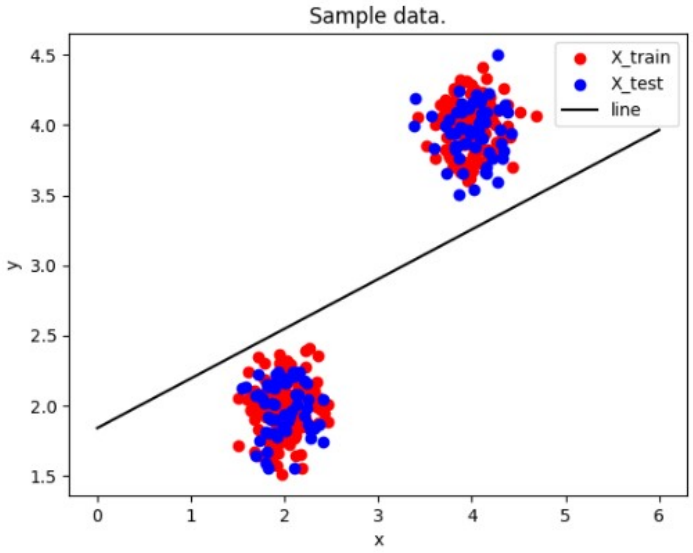


Portanto, é possível observar que a reta, obtida com o vetor de parâmetros w separa corretamente ambos os conjuntos. No entanto é possível afirmar que o treinamento pode ser otimizado de modo que a reta fique equidistante de ambos os pontos. Ao observar e imaginar como novos dados seriam classificados, é notório que dados da amostra 2, acima da reta, tem chances de ser classificados como da amostra 1. Isso inviesa o modelo, e amostras de dados do conjunto 2 tem grandes chances de serem erroneamente classificadas.

Após recompilar todo o código no jupyter notebook, e aumentando o número máximo de épocas, é possível ver a reta estando equidistante de ambos os conjuntos e os separando corretamente. Isso é resultado de um treinamento mais eficaz do que o visto anteriormente.

[28]:
```
t = np.linspace(start = 0, stop = 6, num = 200)
y = -(w[0]/w[1])*t - (w[2]/w[1])
plt.scatter(x_train[:, 0],x_train[:, 1], color = 'red', label = 'X_train')
plt.scatter(x_test[:, 0],x_test[:, 1], color = 'blue', label = 'X_test')
plt.plot(t, y, color = 'black', label = 'line')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title("Sample data.")
plt.plot()
w
```

[28]:
```
array([-0.07818021,  0.22050501, -0.40524321])
```



Parte 3

Na parte 3, é desejado que se use o perceptron simples para classificar o conjunto de dados Iris. O conjunto de dados Iris, contém 4 colunas que nos dão informações sobre a largura e o comprimento da pétala e da sépala de 3 espécies diferentes da planta Iris. Portanto o exercício consiste em, com base nas caracterísitca da planta, classificá-la com uma respectiva espécie. O perceptron é um separador linear, e há 3 espécies, tornando o problema não linearmente separável. Portanto iremos considerar o conjunto de amostras das classes 2 e 3 como se fosse apenas uma classe e utilizar o perceptron para separar os dados e classificá-los com base em suas características.

A imagem abaixo mostra como fica o gráfico das característica de cada amostra de planta.

---

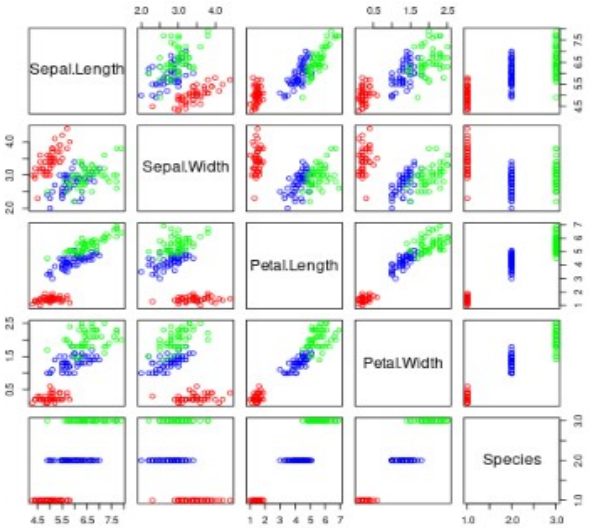
[2]:
```
import numpy as np
import pandas as pd
import rpy2.robjects as ro
from rpy2.robjects.packages import importr
from rpy2.robjects import pandas2ri
import matplotlib.pyplot as plt
from functools import partial
from rpy2.ipython import html
html.html_rdataframe=partial(html.html_rdataframe, table_class="docutils")
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

Usando a biblioteca ryp2 para ler o data set que é nativo da linguagem R.

[206]:
```
%%R
data(iris)
iris_df <- iris
print(rbind(iris[1,], iris[51,], iris[101,]))
plot(iris, col = c("red", "blue", "green")[unclass(iris$Species)])
print(dim(iris_df))
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
51           7.0         3.2          4.7         1.4 versicolor
101          6.3         3.3          6.0         2.5  virginica
[1] 150   5
```



A biblioteca rpy2 é utilizada para integrar o código R com o código python no ambiente jupyter notebook.

A imagem abaixo mostra a conversão de um dataset em R para um dataset pandas, em python. Também há a verificação de incoerências na conversão dos valores do dataset.

Convertendo os dados para 1 dataset pandas e verificando se houve alguma erro na conversão dos datasets.

[207]:
```
iris_df_rob = ro.r['iris']
with (ro.default_converter + pandas2ri.converter).context():
    iris_df_pandas = ro.conversion.get_conversion().rpy2py(iris_df_rob)
iris_df_pandas
```

[207]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
146	6.7	3.0	5.2	2.3	virginica
147	6.3	2.5	5.0	1.9	virginica
148	6.5	3.0	5.2	2.0	virginica
149	6.2	3.4	5.4	2.3	virginica
150	5.9	3.0	5.1	1.8	virginica

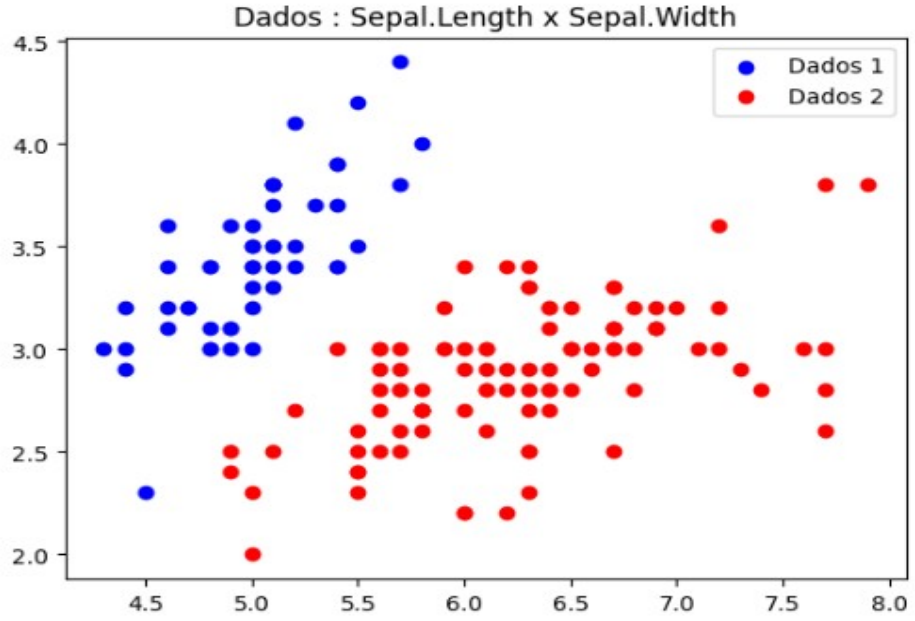
150 rows × 5 columns

[208]:
```
nan_counts = iris_df_pandas.isna().sum()
nan_counts
```

[208]:
```
Sepal.Length    0
Sepal.Width     0
Petal.Length    0
Petal.Width     0
Species         0
dtype: int64
```

É importante ressaltar que, existe toda uma análise e um pré processamento de dados por trás de técnicas de Machine Learning e Deep Learning. Há casos que é melhor eliminar linhas que contém valores NaN, há casos que podemos substituir pela média dos valores. O importante na hora de pré processar, é analisar se existem um ou mais valores que inviésam as respostas do modelo. Exemplos de dados que precisam ser pré processados antes de alimentar algum algoritmo de Machine Learning são : Colunas que contém valores discrepantes e que distoam da média dos outros valores, colunas que contém correlação nulas com todas as outras colunas (coluna ID em muitos datasets).

A imagem abaixo mostra ambos os conjuntos de dados já separados, e prontos para serem classificados.



Foram-se então, utilizadas 105 amostras aleatórias, cerca de 70% do total de amostras, para treinar o perceptron simples. O restante das amostras foram utilizadas para testá-lo.

Gerando os dados de treino e teste aleatóriamente...

[9]:
```
n_data_train = 105
random_data = np.random.permutation(y_real.shape[0]) - 1
xin = np.vstack([xc1, xc2])
x_train = xin[random_data[: n_data_train]]
y_train = y_real[random_data[: n_data_train]]
```

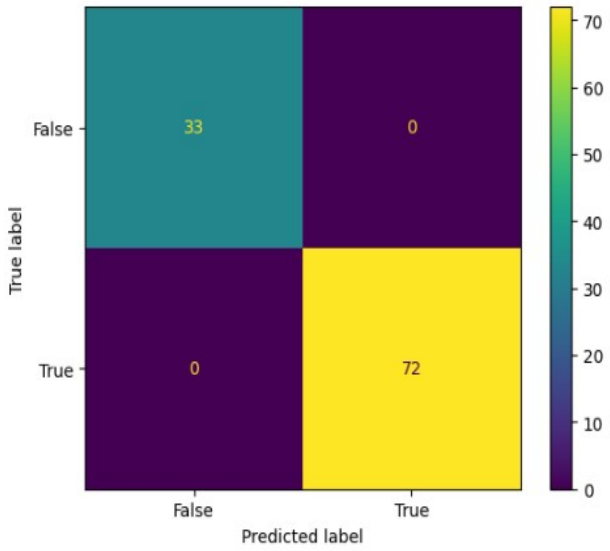
[10]:
```
retlist = train_perceptron(x_train, y_train, 0.01, 0.7, 100, True)
w = np.array(retlist[0])
lst_errors = np.array(retlist[1])
```

Os resultados obtidos com os dados de treino são destacados abaixo na matriz de confusão e na acurácia do perceptron simples sobre o conjunto de treinamento.

[16]:
```
y_hatrain = y_perceptron(x_train, w, True)
acuracy = 1 - (np.transpose(y_train - y_hatrain) @ (y_train - y_hatrain)) / (n_data_train)
print(f"The model has {acuracy*100}% of acuracy.")
```

```
The model has 100.0% of acuracy.
```

[17]:
```
from sklearn import metrics
confusion_matrix = metrics.confusion_matrix(y_train, y_hatrain)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
cm_display.plot()
plt.show()
```



Analisando os resultados, pode-se afirmar que a acurácia do perceptron sobre os dados de treinos é de cerca de 100%. É esperado que o mesmo ocorra com os dados de teste, visto que o dataset Iris contém a classe 1 distante das classes 2 e 3. A matriz de confusão destaca a relação entre as previsões do modelo e os valores reais das amostras. É válido falar também que, há uma análise mais profunda sobre a matriz de confusão e como as classificações estão certas ou erradas. Um certo problema e uma certa análise considera uma certa métrica de avaliação, que difere para outro problema. Uma rede que classifica uma pessoa que tem cancer como uma que não tem cancer é pior do que uma rede que classifica a pessoa como tendo cancer mesmo ela não tendo. Portanto, há pesos sobre as métricas que devem ser considerados na hora de utilizar a matriz de confusão para avaliar a performance do modelo.

A figura abaixo mostra os resultados para os dados de teste :

Gerando os dados de teste e testando o modelo.

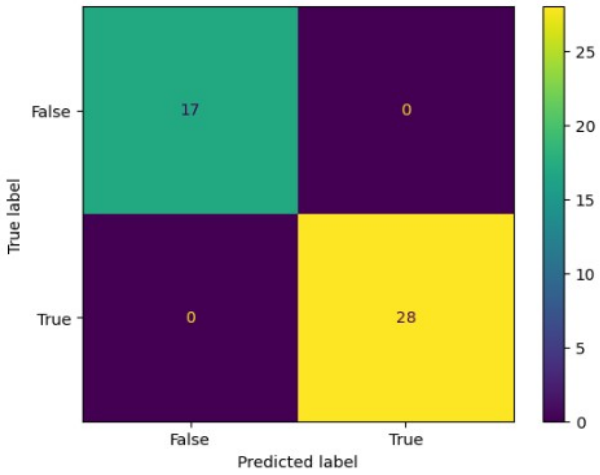
```
[18]: x_test = xin[random_data[n_data_train :]]
      y_test = y_real[random_data[n_data_train :]]
      y_hatest = y_perceptron(x_test, w, True)
```

```
[19]: acuracy = 1 - (np.transpose(y_test - y_hatest) @ (y_test - y_hatest)) / (150 - n_data_train) # 45 dados de teste.
      print(acuracy)
```

```
1.0
```

Plotando a matriz de confusão do treinamento.

```
[21]: from sklearn import metrics
      confusion_matrix = metrics.confusion_matrix(y_test, y_hatest)
      cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
      cm_display.plot()
      plt.show()
```



No fito de mostrar erros nos resultados do modelo, visto que a separação entre essas classes é muito fácil, foi então alterado a ordem das classes, comparando as classes 2 e 3 que estão mais próximas. Para ser mais objetivo, as imagens foram postas abaixo representando a acurácia e a matriz de confusão do modelo, utilizando 70% das amostras de treino e 30% de teste.

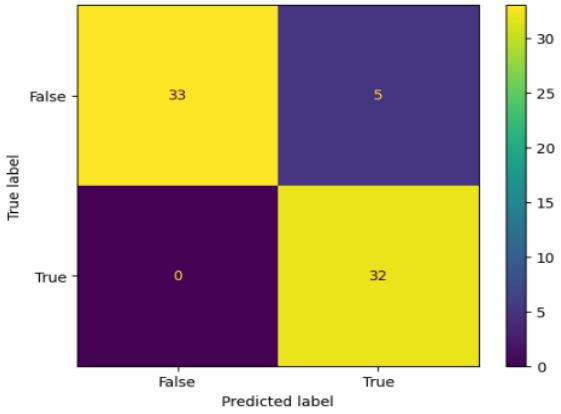
```
[31]: n_data_train = 70
      random_data = np.random.permutation(y_real.shape[0]) - 1
      xin = np.vstack([xc1, xc2])
      x_train = xin[random_data[: n_data_train]]
      y_train = y_real[random_data[: n_data_train]]
```

```
[32]: retlist = train_perceptron(x_train, y_train, 0.01, 0.7, 100, True)
      w = np.array(retlist[0])
      lst_errors = np.array(retlist[1])
```

```
[33]: y_hatrain = y_perceptron(x_train, w, True)
      acuracy = 1 - (np.transpose(y_train - y_hatrain) @ (y_train - y_hatrain)) / (n_data_train)
      print(f"The model has {acuracy*100}% of acuracy.")
```

```
The model has 92.85714285714286% of acuracy.
```

```
[34]: from sklearn import metrics
      confusion_matrix = metrics.confusion_matrix(y_train, y_hatrain)
      cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
      cm_display.plot()
      plt.show()
```



Gerando os dados de teste e testando o modelo.

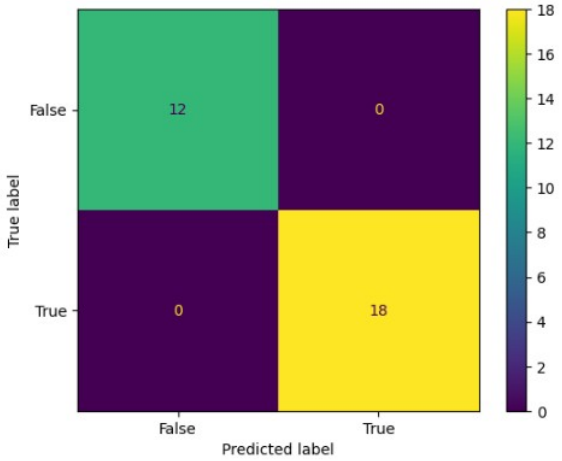
[36]:
```
x_test = xin[random_data[n_data_train :]]
y_test = y_real[random_data[n_data_train :]]
y_hatest = y_perceptron(x_test, w, True)
```

[37]:
```
acuracy = 1 - (np.transpose(y_test - y_hatest) @ (y_test - y_hatest)) / (100 - n_data_train) # 45 dados de teste.
print(acuracy)
```

```
1.0
```

Plotando a matriz de confusão do treinamento.

[38]:
```
from sklearn import metrics
confusion_matrix = metrics.confusion_matrix(y_test, y_hatest)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
cm_display.plot()
plt.show()
```



Portanto, no terceiro exercício foi possível observar o perceptron simples como um separador linear, e separando a planta Iris em espécies com base em suas características. O treinamento do perceptron para classificar as amostras de dados 1 e as amostras (2,3) ocorreu rapidamente e facilmente. Isso aconteceu porque os dados já estavam bem separados, devido a esse motivo, a acurácia sobre os treinos e sobre os testes foi de 100%. Já quando usa as amostras de dados 2 em comparação com as amostras de dados 3, pode-se notar uma reduzida na acurácia de treino. Isso acontece porque os dados estão mais próximos, tornando mais dificil separar as amostras com uma reta.

Parte 4

O exercício 4 é semelhante ao exercício 3. Agora, o dataset se chama “BreastCancer” e contém características celulares coletadas pelo Dr. Wolberg em sua clínica. Portanto, utilizaremos o conjunto de características para alimentar um perceptron simples, que irá predizer o resultado do tumor como 0 (benigno) ou 1 (maligno). O dataset é nativo da biblioteca mlbench em R e foi convertido para um dataset pandas através da biblioteca rpy2.

Usando a biblioteca rpy2 para ler a o dataset Breast Cancer.

[3]:
```
%%R
library(mlbench)
data(BreastCancer)
Breast_Cancerdf <- BreastCancer # R dataframe.
print(dim(Breast_Cancerdf))
```

```
[1] 699  11
In addition: Warning message:
In (function (package, help, pos = 2, lib.loc = NULL, character.only = FALSE,  :
  libraries ‘/usr/local/lib/R/site-library’, ‘/usr/lib/R/site-library’ contain no packages
```

Convertendo os dados para 1 dataset pandas.

[4]:
```
Breast_Cancer_R_df = ro.r['BreastCancer'] # Pegando o dataset R, e o colocando em 1 variável robject dataframe.
with (ro.default_converter + pandas2ri.converter).context():
    Breast_Cancer_2_pandas = ro.conversion.get_conversion().rpy2py(Breast_Cancer_R_df) # Convertendo em 1 dataset pandas.
Breast_Cancer_2_pandas
```

[4]:

	Id	Cl.thickness	Cell.size	Cell.shape	Marg.adhesion	Epith.c.size	Bare.nuclei	Bl.cromatin	Normal.nucleoli	Mitoses	Class
1	1000025	5	1	1	1	2	1	3	1	1	benign
2	1002945	5	4	4	5	7	10	3	2	1	benign
3	1015425	3	1	1	1	2	2	3	1	1	benign
4	1016277	6	8	8	1	3	4	3	7	1	benign
5	1017023	4	1	1	3	2	1	3	1	1	benign
...	...	...	...	...	...	...	...	...	...	...	...
695	776715	3	1	1	1	3	2	1	1	1	benign
696	841769	2	1	1	1	2	1	1	1	1	benign
697	888820	5	10	10	3	7	3	8	10	2	malignant
698	897471	4	8	6	4	3	4	10	6	1	malignant
699	897471	4	8	8	5	4	5	10	4	1	malignant

699 rows × 11 columns

Realizando uma análise sobre as características dos dados, é possível afirmar que há 16 valores NaN’s. Esses valores NaN’s não implicarão a linha que os contém ser removida. Os valores NaN’s serão substituidos pela média dos valores da coluna de características que os mesmos pertencem.

[177]:
```
Breast_Cancer_2_pandas.isna().sum() # Há 16 valores NaN na coluna Bare.nuclei.
```

[177]:
```
Id                0
Cl.thickness      0
Cell.size         0
Cell.shape        0
Marg.adhesion     0
Epith.c.size      0
Bare.nuclei      16
Bl.cromatin       0
Normal.nucleoli   0
Mitoses           0
Class             0
dtype: int64
```

[178]:
```
Breast_Cancer_2_pandas['Bare.nuclei'] = pd.to_numeric(Breast_Cancer_2_pandas['Bare.nuclei'])
max_value = Breast_Cancer_2_pandas['Bare.nuclei'].max()
min_value = Breast_Cancer_2_pandas['Bare.nuclei'].min()
median_value = Breast_Cancer_2_pandas['Bare.nuclei'].median()
print("Maximum value:", max_value)
print("Minimum value:", min_value)
print("Median value:", median_value)
```

```
Maximum value: 10.0
Minimum value: 1.0
Median value: 1.0
```

[179]:
```
Breast_Cancer_2_pandas['Bare.nuclei'] = Breast_Cancer_2_pandas['Bare.nuclei'].fillna(median_value).infer_objects(copy=False)
Breast_Cancer_2_pandas = Breast_Cancer_2_pandas.astype(int)
```

A separação dos dados de treino e os dados de teste é mostrada na figura abaixo :

Criando os dados que alimentarão o modelo para treino e teste.

[180]:
```
xin = np.array(Breast_Cancer_2_pandas.iloc[:, 1 : 10]) # Pegando todas as colunas com exceção da última e o id, o id não influencia.
y_real = np.array(Breast_Cancer_2_pandas.iloc[:, -1]) # Pegando a ultima coluna (label).
n_data_train = 490 # 70% dos dados sao de treino.
random_data = np.random.permutation(y_real.shape[0]) - 1 # Embaralhando a sequencia de dados de entrada para treino e teste.
x_train = xin[random_data[: n_data_train]]
y_train = y_real[random_data[: n_data_train]]
x_test = xin[random_data[n_data_train : ]]
y_test = y_real[random_data[n_data_train :]]
```

A figura abaixo mostra o treinamento do perceptron simples no conjunto de dados “BreastCancer”

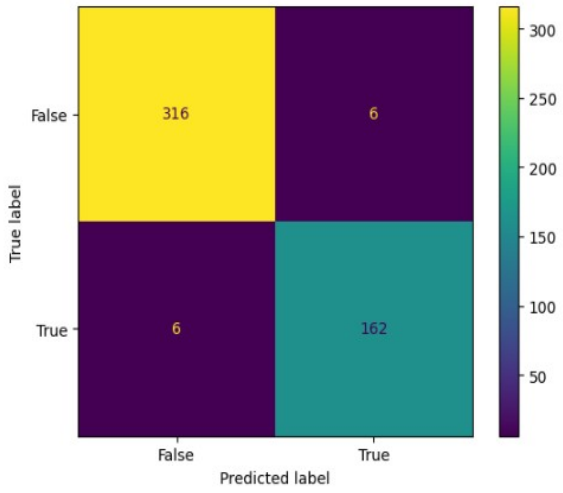
Treinando o meu perceptron.

[23]:
```
retlist = train_perceptron(x_train, y_train, 0.01, 0.7, 1000, True)
w = np.array(retlist[0])
lst_errors = np.array(retlist[1])
```

[28]:
```
yhatrain = y_perceptron(x_train, w, True)
acuracy = 1 - (np.transpose(y_train - yhatrain) @ (y_train - yhatrain)) / (n_data_train)
print(acuracy)
```

```
0.9755102040816327
```

[29]:
```
from sklearn import metrics
confusion_matrix = metrics.confusion_matrix(y_train, yhatrain)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
cm_display.plot()
plt.show()
```



A acurácia do modeo sobre os dados de treino é de aproximadamente 97%. O conjunto de amostras de treino contém cerca de 490 amostras e que foram selecionadas aleatóriamente para alimentar o perceptron simples no seu treinamento. O restante das amostras serão utilizadas para testar o perceptron simples.

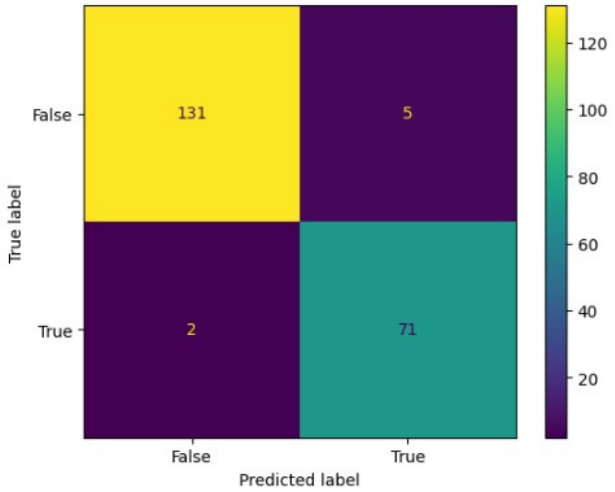
A figura abaixo mostra a acurácia e o desempenho do perceptron simples para os dados de teste. Foram utilizadas 210 amostras para testar o modelo, e a acurácia foi de aproximadamente 96,6%.

Testando o meu perceptron.

[30]:
```
y_hatest = y_perceptron(x_test, w, True)
acuracy = 1 - (np.transpose(y_test - y_hatest) @ (y_test - y_hatest)) / (699 - n_data_train)
print(acuracy)
```

```
0.9665071770334929
```

[31]:
```
confusion_matrix = metrics.confusion_matrix(y_test, y_hat)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
cm_display.plot()
plt.show()
```



[26]:
```
sum = 0
for (index,value) in enumerate(y_hat):
    if value != y_test[index]:
        sum += 1
print(f"There was {sum} errors...")
```

```
There was 7 errors...
```