Centro Universitário FEI

***Naive Bayes***

**Aula 4 - Exercícios**

Disciplina:

Tópicos Especiais em Aprendizagem (Prof. Reinaldo A. C. Bianchi)

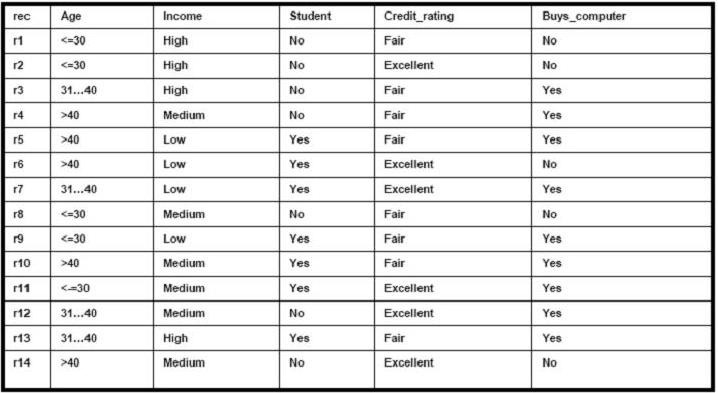
Aluno:

Claudio Aparecido Borges Junior (RA 120122-7)

São Bernardo do Campo

2o semestre / 2020

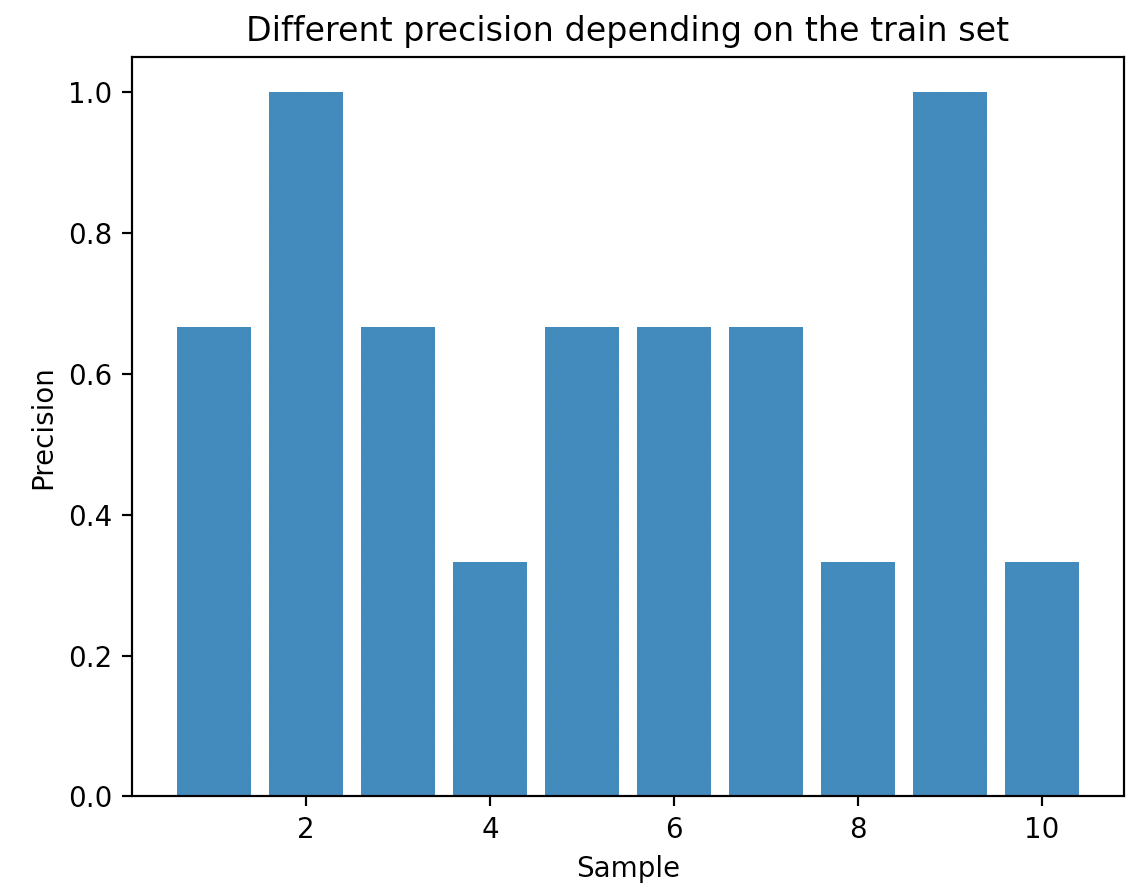
**EXERCÍCIO 1:** *Utilize o Naive Bayes para classificar:*

**

A implantação foi realizada utilizando a linguagem Python versão 3.6, com função de entrada ex1() presente no Anexo 1. A classe NaiveBayes foi criada para encapsular as ações necessárias para treinar e classificar utilizando o método de Bayes ingênuo.

Dois métodos são utilizados, o train e o predict. O primeiro treina o modelo utilizando um conjunto de N entradas de M atributos cada e uma lista de N classificações de K classes. O segundo prediz em qual das K classes o conjunto de entrada pertence utilizando o método ingênuo de Bayes.

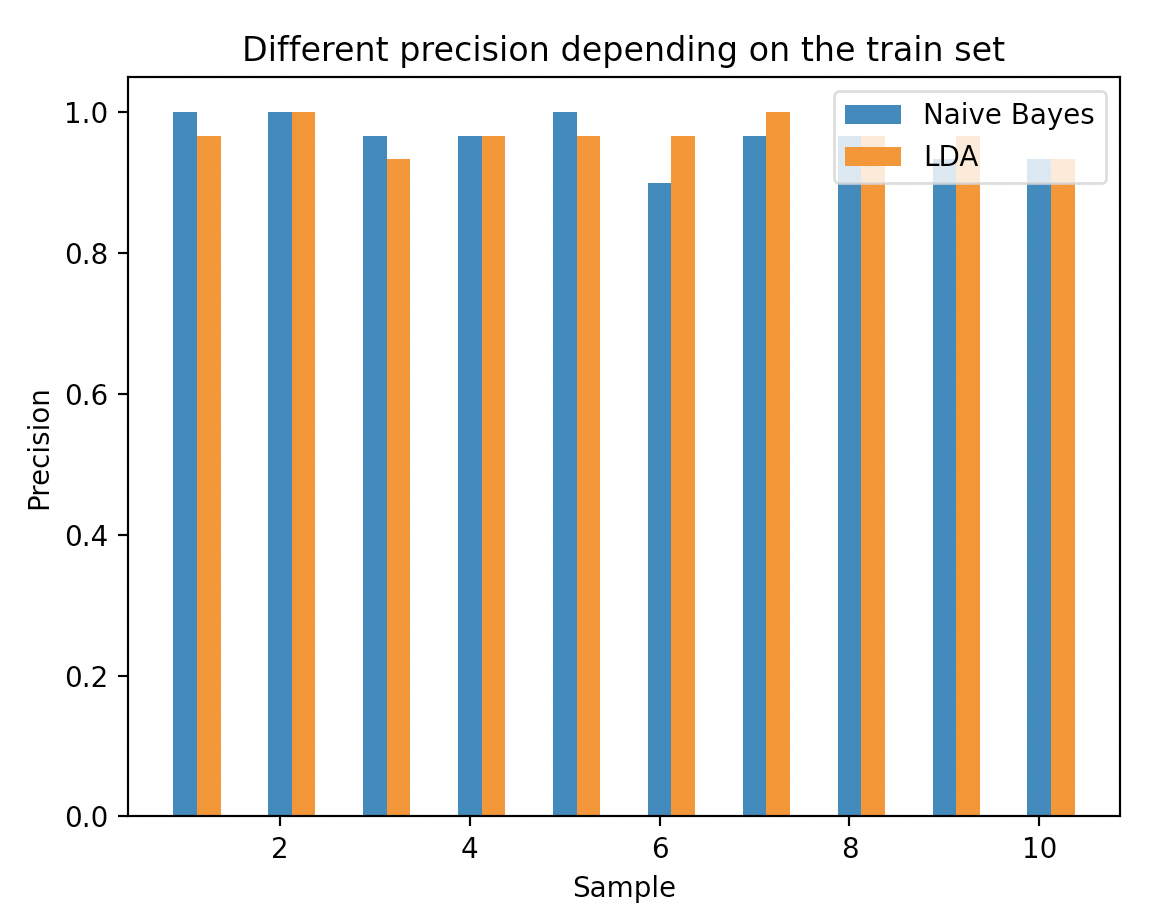
O conjunto de treinamento e teste pode interferir na precisão do resultado, em que a precisão é a razão da quantidade de acertos pela quantidade total a ser classificada. O gráfico abaixo mostra 10 amostras todas elas com conjuntos de treinamento e teste diferentes.



**EXERCÍCIO 2:** *Use o Naive Bayes Gaussiano para classificar o dataset Iris, e mais dois de sua escolha (os mesmos do trabalho anterior)*

**Conjunto de dados “Iris”**: <http://archive.ics.uci.edu/ml/datasets/Iris>

O conjunto de dados da Iris possui 4 atributes (largura e comprimento das sépalas e pétalas) e 3 classes (setosa, virgínica e versicolor). O método de Bayes ingênuo foi aplicado juntamente com método LDA e geraram as seguintes precisões a depender do conjunto de treinamento e teste:

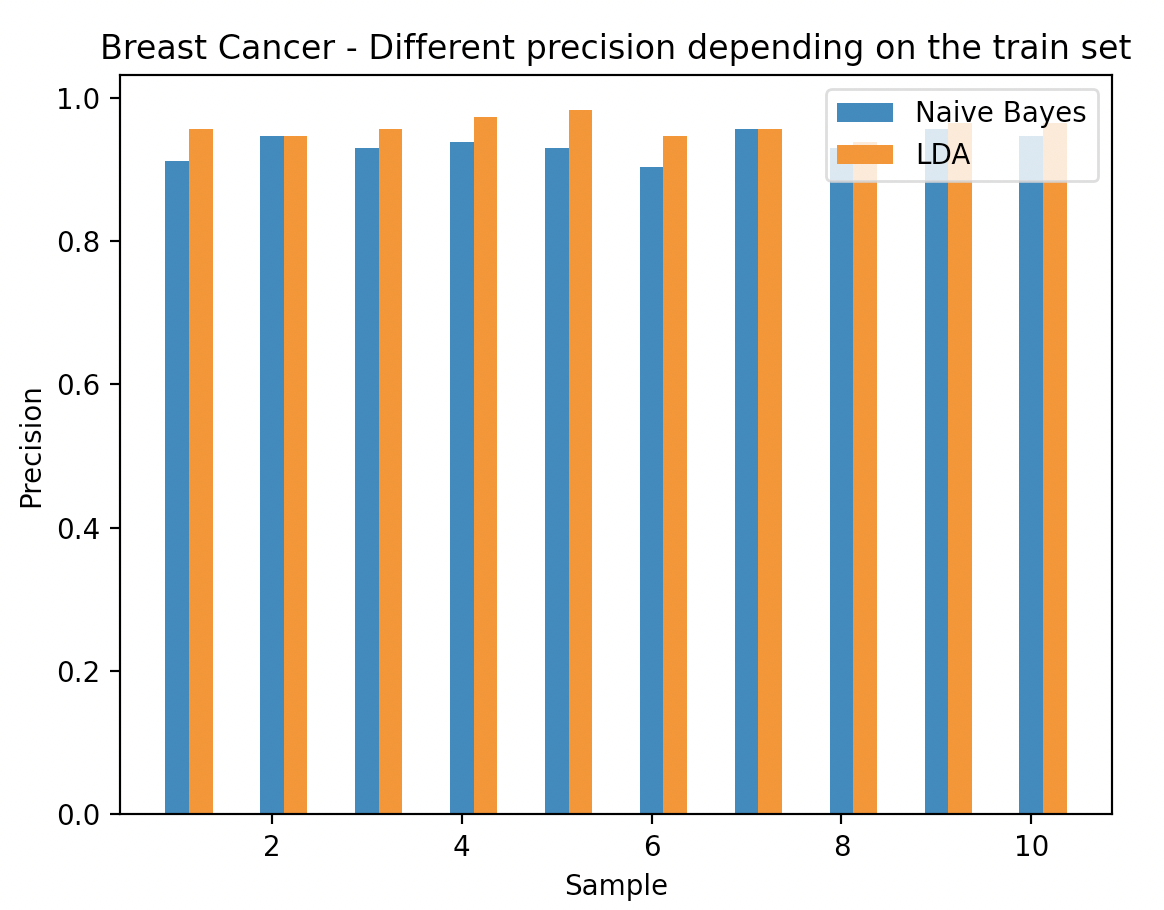
**

As precisões encontradas foram todas superiores a 0.9, e ambos os métodos, Bayes ingênuo e LDA, tiveram resultados semelhantes, ou seja, nenhum método alcançou uma precisão suporior a outro consistentemente.

O código fonte está no ANEXO I com função de entrada ex2\_iris().

**Conjunto de dados “Breast Cancer Wisconsin”**: <http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29>

Conjunto de dados sobre câncer de mama com 10 atributos e possíveis classificações, câncer maligno ou benigno. Uma preparação foi realizada no conjunto de testes onde as linhas que continham valores inválidos foram retiradas. O método de Bayes ingênuo foi aplicado juntamente com método LDA e geraram as seguintes precisões a depender do conjunto de treinamento e teste:

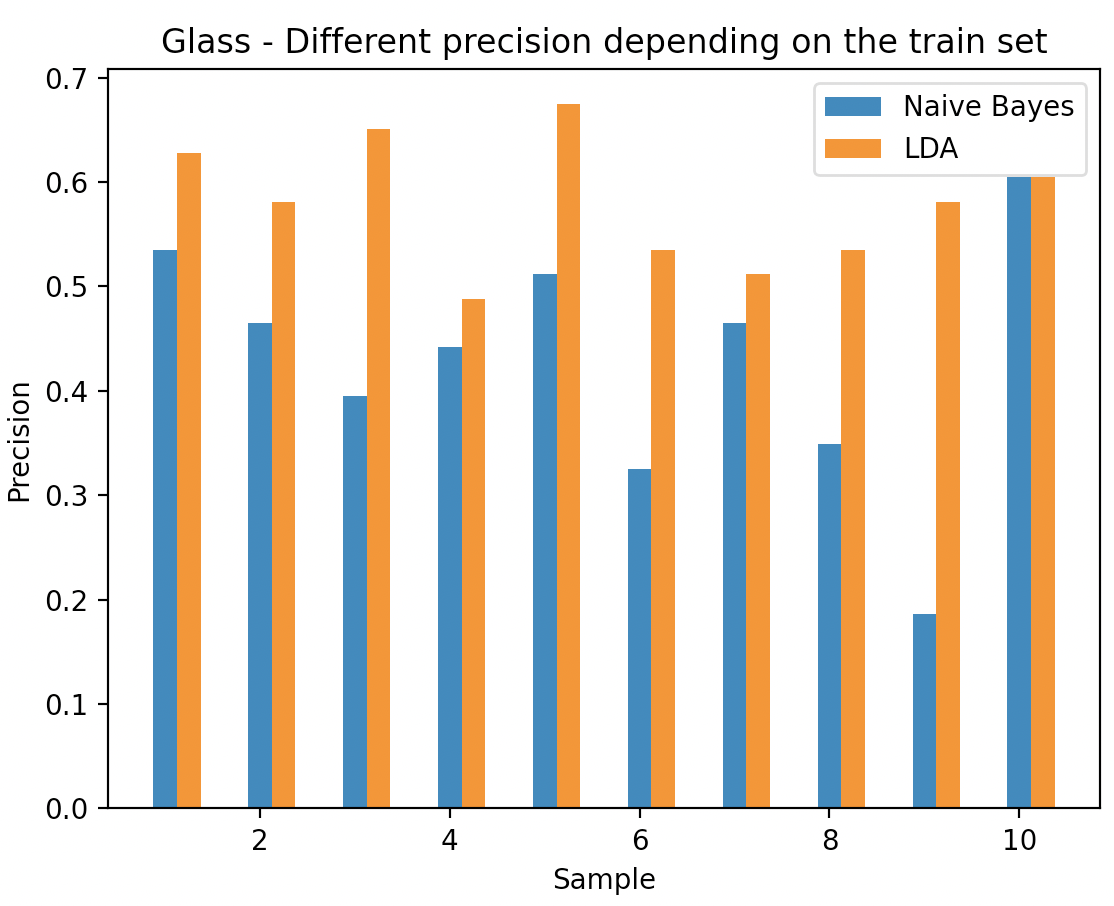
**

As precisões encontradas foram todas superiores a 0.9. Nesse caso, o LDA obteve resultado ligeiramente superior ao NB. Das 10 execuções, 8 tiveram o LDA com precisão maior e 2 com precisão igual.

O código fonte está no ANEXO I com função de entrada ex2\_breast\_cancer().

**Conjunto de dados “Glass Identification”**: <http://archive.ics.uci.edu/ml/datasets/Glass+Identification>

Conjunto de dados utilizado para treinamento de classificação de tipos de vidros em 7 categorias motivados por investigações criminológicas. O método de Bayes ingênuo foi aplicado juntamente com método LDA e geraram as seguintes precisões a depender do conjunto de treinamento e teste:



Nesse exemplo, o método de Bayes ingênuo obteve precisão pior do que o LDA, em especial no exemplo 9.

O código fonte está no ANEXO I com função de entrada ex2\_glass().

**ANEXO I – CÓDIGO FONTE**

import numpy as np  
from collections import Counter  
from collections import defaultdict  
import copy  
import math  
from matplotlib import pyplot as plt  
  
  
class NaiveBayes:  
 @staticmethod  
 def split\_train\_test(x, y, train\_size=0.8):  
 *""" Split the input and the output into train and test datasets* ***:param*** *x: dataset inputs* ***:param*** *y: dataset output* ***:param*** *train\_size: ranging from 0 to 1* ***:return****: the train and test x and y's  
 """* cutoff\_index = int(len(x) \* train\_size)  
 x\_train = x[:cutoff\_index, :]  
 y\_train = y[:cutoff\_index, :]  
 x\_test = x[cutoff\_index:, :]  
 y\_test = y[cutoff\_index:, :]  
 return x\_train, y\_train, x\_test, y\_test,  
  
 def \_\_init\_\_(self):  
 # Create a dictionary with all possible elements to be used across  
 # the naive bayes processing  
 self.nb\_dict = defaultdict(  
 lambda: defaultdict(lambda: defaultdict(int)))  
  
 # Init the probability models as None  
 self.p\_class = None  
 # Used for discrete values  
 self.p\_conditional = None  
 # Used for continuous values  
 self.mean = None  
 self.std = None  
  
 def train(self, x, y):  
 *""" Train the Naive Bayes for x and y* ***:param*** *x: input attributes* ***:param*** *y: output classes* ***:return****: nothing  
 """* x\_transposed = x.transpose()  
 y\_transposed = y.transpose()[0]  
  
 # Calculate the probability of each class  
 y\_count = Counter(y\_transposed)  
 p\_class = {c: y\_count[c] / len(y\_transposed) for c in y\_count}  
  
 # Calculate the conditional probabilities  
 # Count the number of times an attribute was seen for each class  
 attribute\_group\_count = copy.deepcopy(self.nb\_dict)  
 for i in range(len(x\_transposed)):  
 for j in range(len(x\_transposed[i])):  
 # Group -> class (Yes or No)  
 # Column -> The column of the table  
 # Attribute -> One of the distinct values of a column  
 group = y\_transposed[j]  
 column = i  
 attribute = x\_transposed[i][j]  
 # Account for it (sum 1)  
 attribute\_group\_count[group][column][attribute] += 1  
  
 # Change it to conditional probability  
 p\_conditional = copy.deepcopy(self.nb\_dict)  
 for group in attribute\_group\_count:  
 for col in attribute\_group\_count[group]:  
 for attribute in attribute\_group\_count[group][col]:  
 # Conditional probability is the number of times an  
 # attribute showed up for a giving class  
 count = attribute\_group\_count[group][col][attribute]  
 p\_conditional[group][col][attribute] = count / y\_count[  
 group]  
  
 # Make the class aware of the training  
 self.p\_conditional = p\_conditional  
 self.p\_class = p\_class  
  
 def predict(self, x):  
 *""" Predict the class of each element of x  
  
 Warning: this method must be called after training* ***:param*** *x: dataset input to be predicted* ***:return****: a list of classes  
 """* predictions = []  
 for i in range(len(x)):  
 # For each vector sample  
 attributes = x[i]  
 # Find the group that has the maximum probability (argmax)  
 chosen = None  
 max\_probability = 0;  
 for c in self.p\_class:  
 # Test with all possible groups  
  
 # Calculate the probability by multiplying the conditional  
 # probabilities and them multiply by the class probability  
 probability = 1  
 for j in range(len(attributes)):  
 p = self.p\_conditional[c][j][attributes[j]]  
 # If the probability is 0, use a very small value  
 probability \*= p if p > 0 else 1 / 10 \*\* 100  
 probability \*= self.p\_class[c]  
  
 # Find the argmax  
 if chosen is None or probability >= max\_probability:  
 chosen = c  
 max\_probability = probability  
 predictions.append(chosen)  
 return predictions  
  
 def train\_continuous(self, x, y):  
 *""" Train Naive Bayes with continuous values* ***:param*** *x: continuous input attributes* ***:param*** *y: output classes* ***:return****: nothing  
 """* x\_transposed = x.transpose()  
 y\_transposed = y.transpose()[0]  
  
 # Calculate the probability of each class  
 y\_count = Counter(y\_transposed)  
 p\_class = {c: y\_count[c] / len(y\_transposed) for c in y\_count}  
  
 # Calculate the mean of each class attribute  
 mean = defaultdict(lambda: defaultdict(float))  
 for c in p\_class:  
 x\_class = x[y\_transposed == c]  
 for i, row in enumerate(x\_class.transpose()):  
 mean[c][i] = sum(row) / len(row)  
  
 # Calculate the standard deviation of each class attribute  
 std = defaultdict(lambda: defaultdict(float))  
 for c in p\_class:  
 x\_class = x[y\_transposed == c]  
 for i, row in enumerate(x\_class.transpose()):  
 std[c][i] = math.sqrt(  
 sum([(elm - mean[c][i]) \*\* 2 for elm in row]) / (  
 len(row) - 1))  
 # Adjust standard deviation to be non zero (otherwise a division  
 # by 0 will be throw). Use a very high value to minimize the  
 # conditional probability  
 std[c][i] = std[c][i] if std[c][i] > 0 else 10 \*\* 100  
  
 self.p\_class = p\_class  
 self.mean = mean  
 self.std = std  
  
 def predict\_continuous(self, x):  
 *""" Predict the class of each element of x with continuous value  
  
 Warning: this method must be called after training* ***:param*** *x: dataset input to be predicted* ***:return****: a list of classes  
 """* predictions = []  
 for i in range(len(x)):  
 # For each vector sample  
 attributes = x[i]  
 # Find the group that has the maximum probability (argmax)  
 chosen = None  
 max\_probability = 0;  
 for c in self.p\_class:  
 # Test with all possible groups  
  
 # Calculate the probability by multiplying the conditional  
 # probabilities and them multiply by the class probability  
 probability = 1  
 for j in range(len(attributes)):  
 mean = self.mean[c][j]  
 sample = attributes[j]  
 std = self.std[c][j]  
 var = std \*\* 2  
 k = math.sqrt(2 \* math.pi)  
 # Apply the continuous formula using gaussian distribution  
 p = 1 / (k \* std) \* math.exp(  
 - (sample - mean) \*\* 2 / (2 \* var))  
 probability \*= p  
 # Multiply by the class probability  
 probability \*= self.p\_class[c]  
  
 # Find the argmax  
 if chosen is None or probability >= max\_probability:  
 chosen = c  
 max\_probability = probability  
 predictions.append(chosen)  
 return predictions  
  
  
def assertiveness(real, prediction):  
 *""" Calculate how good the prediction is.* ***:param*** *real: real values* ***:param*** *prediction: predicted values* ***:return****: quality measure ranging from [0 - 1] (inclusive)  
 """* err = sum(  
 [1 for a, b in zip(real, prediction) if a == b])  
 return err / len(prediction)  
  
  
def ex1():  
 # Input data, giving byt the excercise  
 data = np.array([  
 ['<=30', 'High', 'No', 'Fair', 'No', ],  
 ['<=30', 'High', 'No', 'Excellent', 'No', ],  
 ['31..40', 'High', 'No', 'Fair', 'Yes', ],  
 ['>40', 'Medium', 'No', 'Fair', 'Yes'],  
 ['>40', 'Low', 'Yes', 'Fair', 'Yes', ],  
 ['>40', 'Low', 'Yes', 'Excellent', 'No', ],  
 ['31..40', 'Low', 'Yes', 'Excellent', 'Yes', ],  
 ['<=30', 'Medium', 'No', 'Fair', 'No', ],  
 ['<=30', 'Low', 'Yes', 'Fair', 'Yes', ],  
 ['>40', 'Medium', 'Yes', 'Fair', 'Yes', ],  
 ['<=30', 'Medium', 'Yes', 'Excellent', 'Yes', ],  
 ['31..40', 'Medium', 'No', 'Excellent', 'Yes', ],  
 ['31..40', 'High', 'Yes', 'Fair', 'Yes', ],  
 ['>40', 'Medium', 'No', 'Excellent', 'No', ],  
 ])  
  
 # Run the model N times  
 precisions = []  
 for k in range(10):  
 # Suffle the data to decrease the bias  
 np.random.shuffle(data)  
  
 # Split the into X an Y where X are the input samples and Y are the output  
 # samples (what we want to classify)  
 x = data[:, :-1]  
 y = data[:, -1:]  
  
 # Split the dataset into training and test  
 x\_train, y\_train, x\_test, y\_test, = NaiveBayes.split\_train\_test(x, y)  
  
 nb = NaiveBayes()  
 nb.train(x\_train, y\_train)  
 prediction = nb.predict(x\_test)  
  
 # Calculate the precision  
 precision = assertiveness(y\_test.transpose()[0], prediction)  
 precisions.append(precision)  
 print('Naive Bayes assertiveness: ', precision)  
  
 fig = plt.figure()  
 plt.bar(list(range(1, len(precisions) + 1)), precisions)  
 plt.xlabel("Sample")  
 plt.ylabel("Precision")  
 plt.title("Different precision depending on the train set")  
  
  
def ex2\_iris():  
 # Read the iris.data  
 data = np.genfromtxt('datasets/iris.data', delimiter=',', dtype=None,  
 encoding=None)  
 # Run the model N times  
 precisions = []  
 precisions\_lda = []  
 for k in range(10):  
 # Suffle the data to decrease the bias  
 np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the  
 # output samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if i < len(row) - 1] for row in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == len(row) - 1] for row in  
 data])  
  
 # Split the dataset into training and test  
 x\_train, y\_train, x\_test, y\_test, = NaiveBayes.split\_train\_test(x, y)  
  
 nb = NaiveBayes()  
 nb.train\_continuous(x\_train, y\_train)  
 prediction = nb.predict\_continuous(x\_test)  
  
 # Calculate the precision  
 precision = assertiveness(y\_test.transpose()[0], prediction)  
 precisions.append(precision)  
  
 # Run LDA from the scikit learn just as a matter of comparison  
 from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis  
 clf = LinearDiscriminantAnalysis()  
 clf.fit(x\_train, y\_train)  
 prediction\_lda = clf.predict(x\_test)  
 precision\_lda = assertiveness(y\_test.transpose()[0], prediction\_lda)  
 precisions\_lda.append(precision\_lda)  
  
 # Plot both NB and LDA  
 fig = plt.figure()  
 fig\_x = list(range(1, len(precisions) + 1))  
 plt.bar([x for x in range(1, 11)], precisions, width=0.25,  
 label='Naive Bayes')  
 plt.bar([x + 0.25 for x in range(1, 11)], precisions\_lda, width=0.25,  
 label='LDA')  
 plt.xlabel("Sample")  
 plt.ylabel("Precision")  
 plt.title("Different precision depending on the train set")  
 plt.legend()  
 plt.show()  
  
  
def ex2\_breast\_cancer():  
 # Read the wpbc.data  
 data = np.genfromtxt('datasets/wdbc.data', delimiter=',', dtype=None,  
 encoding=None)  
  
 # Run the model N times  
 precisions = []  
 precisions\_lda = []  
 for k in range(10):  
 # Suffle the data to decrease the bias  
 np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the output  
 # samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if i >= 2] for row in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == 1] for row in  
 data])  
  
 # Split the dataset into training and test  
 x\_train, y\_train, x\_test, y\_test, = NaiveBayes.split\_train\_test(x, y)  
  
 nb = NaiveBayes()  
 nb.train\_continuous(x\_train, y\_train)  
 prediction = nb.predict\_continuous(x\_test)  
  
 # Calculate the precision  
 precision = assertiveness(y\_test.transpose()[0], prediction)  
 precisions.append(precision)  
  
 # Run LDA from the scikit learn just as a matter of comparison  
 from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis  
 clf = LinearDiscriminantAnalysis()  
 clf.fit(x\_train, y\_train)  
 prediction\_lda = clf.predict(x\_test)  
 precision\_lda = assertiveness(y\_test.transpose()[0], prediction\_lda)  
 precisions\_lda.append(precision\_lda)  
  
 # Plot both NB and LDA  
 fig = plt.figure()  
 fig\_x = list(range(1, len(precisions) + 1))  
 plt.bar([x for x in range(1, 11)], precisions, width=0.25,  
 label='Naive Bayes')  
 plt.bar([x + 0.25 for x in range(1, 11)], precisions\_lda, width=0.25,  
 label='LDA')  
 plt.xlabel("Sample")  
 plt.ylabel("Precision")  
 plt.title("Breast Cancer - Different precision depending on the train set")  
 plt.legend()  
 plt.show()  
  
  
def ex2\_glass():  
 # Read the wpbc.data  
 data = np.genfromtxt('datasets/glass.data', delimiter=',', dtype=None,  
 encoding=None)  
  
 # Run the model N times  
 precisions = []  
 precisions\_lda = []  
 for k in range(10):  
 # Suffle the data to decrease the bias  
 np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the output  
 # samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if 0 < i < len(row) - 1] for row  
 in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == len(row) - 1] for row in  
 data])  
  
 # Split the dataset into training and test  
 x\_train, y\_train, x\_test, y\_test, = NaiveBayes.split\_train\_test(x, y)  
  
 nb = NaiveBayes()  
 nb.train\_continuous(x\_train, y\_train)  
 prediction = nb.predict\_continuous(x\_test)  
  
 # Calculate the precision  
 precision = assertiveness(y\_test.transpose()[0], prediction)  
 precisions.append(precision)  
  
 # Run LDA from the scikit learn just as a matter of comparison  
 from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis  
 clf = LinearDiscriminantAnalysis()  
 clf.fit(x\_train, y\_train)  
 prediction\_lda = clf.predict(x\_test)  
 precision\_lda = assertiveness(y\_test.transpose()[0], prediction\_lda)  
 precisions\_lda.append(precision\_lda)  
  
 # Plot both NB and LDA  
 fig = plt.figure()  
 fig\_x = list(range(1, len(precisions) + 1))  
 plt.bar([x for x in range(1, 11)], precisions, width=0.25,  
 label='Naive Bayes')  
 plt.bar([x + 0.25 for x in range(1, 11)], precisions\_lda, width=0.25,  
 label='LDA')  
 plt.xlabel("Sample")  
 plt.ylabel("Precision")  
 plt.title("Glass - Different precision depending on the train set")  
 plt.legend()  
 plt.show()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 ex1()  
 ex2\_iris()  
 ex2\_breast\_cancer()  
 ex2\_glass()