Jorge Henrique Ferreira da Silva

2019006825

Link para notebook: https://github.com/Jhferr/knn_kdtree

Trabalho Prático 01 – Aplicações de algoritmos geométricos

Classificação por kNN (k-nearest neighbors)

Para classificação através do knn, a classe de cada ponto é definida como sendo a classe de maior frequência entre os k pontos mais próximos. Para armazenamento dos pontos é utilizado de uma árvore kd; dessa forma, após armazenados, a busca pelos k pontos mais próximos na árvore se dá pelo percorrimento da árvore até o suposto local em que o ponto alvo deveria ser inserido, encontrando-se o suposto ponto mais próximo e inserindo-o na lista dos k mais próximos. Caso não se tenha preenchido os k elementos ou a distância do último ponto inserido ao alvo seja maior que a distância perpendicular entre o alvo e valor da secção que divide o conjunto de pontos, percorre-se a árvore pelo outro ramo. Por fim, da lista gerada, é identificado a classe do ponto alvo pela frequência da ocorrência das classes.

Detalhes da implementação

Inicialmente, foi definido a classe "Node" que armazenará as informações dos nós na árvore: as coordenados do ponto (caso seja uma folha) ou o valor da coordenada do ponto médio para referência (caso contrário), a árvore à esquerda, a árvore à direita, o nível de profundidade na árvore e a classe do ponto (caso seja uma folha). Por sua vez, na construção da árvore, caso a lista de pontos contenha mais de um ponto, essa é ordenada de acordo com o eixo (módulo da profundidade atual pela dimensão do ponto), sendo armazenado o valor correspondente na posição do eixo do ponto médio, separado a lista em dois a partir do ponto médio, chamado a função que cria a árvore para ambas metades e retornado um nó com essas informações armazenadas nele. Caso a lista seja menor que dois, será retornado um nó folha contendo o ponto em análise, sua profundidade e classe. Sendo assim, apenas nas folhas será contido o ponto, enquanto nos pontos intermediários há apenas uma referência para percorrimento na árvore.

Definido como a árvore é construída, a função responsável pela execução do kNN e geração das estatísticas sobre o resultado recebe a lista de pontos a ser avaliada, o número k de vizinhos a serem encontrados e o identificador das classes. Por sua vez, essa função separa o conjunto de pontos em teste e treino seguindo proporção exigida (70-30) e constroí a árvore kd seguindo o algoritmo descrito acima para o conjunto de treino. Em seguida, para cada ponto no conjunto de teste, é chamada a função "knn", que será descrita abaixo, retornando a lista com os k vizinhos. Dessa lista, é identificado a classe mais recorrente dentre os pontos, atribuída ao ponto e

identificado classificação dentre possíveis identificadores em uma matriz de confusão (verdadeiro negativo, falso negativo, verdadeiro positivo ou falso negativo). Após a execução para cada ponto no conjunto de teste, as métricas para classificação são calculadas com os valores da matriz de confusão e essas, juntamente com a lista com a atribuição das classes, são retornadas pelo método, encerrando o procedimento.

A função "knn" é definida por uma série de chamadas a mesma função, percorrendo a árvore pelas ramificações de interesse até que os k vizinhos mais próximos sejam encontrados. Em uma descrição mais precisa, a função se chama recursivamente até que o nó avaliado seja um nó folha (suposto vizinho mais próximo do ponto alvo), caminhando pelas ramificações do nó avaliado até a suposta localização em que o ponto alvo deveria se situar na árvore. Ao se deparar com o nó folha, é preenchido a lista de prioridade de vizinhos caso a mesma não esteja completa; estando completa, é analisado a distância do ponto alvo ao ponto sendo avaliado: caso seja menor que a distância armazenada do ponto de maior distância na lista, esse é substituído pelos ponto avaliado e sua distância, sendo retornado pela função a lista de prioridade atualizada. Por fim, é "subido" na árvore analisando se o ponto inserido na lista é o mais próximo ou se a outra ramificação não analisada possa conter outro menos distante (caso raio da circunferência formada pelo último ponto inserido na lista e centro no ponto alvo seja maior ou igual a distância entre o ponto alvo e o divisor dos pontos armazenado no nó avaliado), e procurando pelos demais vizinhos para preenchimento da lista (caso não esteja completa), caminhando-se pelos ramos inexplorados da árvore (caso condição descrita no parênteses acima seja evidenciado).

Por critério de simplificação no manuseio e armazenamento de informações nos nós na árvore, foi escolhido a criação da classe "Node" como descrito anteriormente. Por sua vez, tanto para armazenamento dos dados a serem classificados e das classes atribuídas após a classificação, quanto para armazenamento e ordenação da lista de prioridade com os k vizinhos mais próximos, foi utilizado da estrutura "lista" e de seu próprio método de ordenação "sort()", evitando introdução de erros com estruturas mais complexas e menos adaptáveis. Além disso, o método para cálculo das estatísticas pela formação (desconstruída) da matriz de confusão busca maior eficiência facilitando a contagem e identificação de possíveis erros novamente simplificando as etapas e cálculos envolvidos.

```
import numpy as np
import random

class Node:
  def __init__(self, value, left_tree, right_tree, depth, nClass=-1):
    self.value = value # coordenadas do ponto, caso nó seja folha; valor de referência, ca
    self.left_tree = left_tree
    self.right_tree = right_tree
    self.depth = depth
    self.nClass = nClass # classe real, caso nó seja folha
```

```
def kdTree(pointList, depth = 0):
  if (len(pointList) > 1):
    try:
      dim = len(pointList[0])-1  # dimensão do ponto (excluindo informação da classe)
    except LookupError:
      return None
    else:
      dim = len(pointList[0])-1
      axis = depth % dim; # eixo atual
      pointList.sort(key=lambda lis: lis[axis])
      m = pointList[(-(len(pointList) // -2)) - 1][axis] # valor do ponto médio para perco
      left_child = kdTree(pointList[:(-(len(pointList) // -2))], depth + 1)
      right_child = kdTree(pointList[(-(len(pointList) // -2)):], depth + 1)
      return Node(m, left_child, right_child, depth) # nó intermediário
  else:
    dim = len(pointList[0])-1
    return Node(pointList[0][:-1], None, None, depth, pointList[0][dim]) # nó folha com in

def knn(nodeTree, priorityList, target, kN): #procura dos k vizinhos mais próximos
  if (nodeTree == None): #árvore vazia
    return priorityList
  if (type(nodeTree.value) == list):# nó folha
    distEuc = np.linalg.norm(np.asarray(nodeTree.value) - np.asarray(target[:-1])) # distâ
    if (len(priorityList)) < kN :
      priorityList.append([distEuc, nodeTree])
      priorityList.sort(key=lambda lis: lis[0])
    else:
      if priorityList[-1][0] > distEuc:
        priorityList[-1] = [distEuc, nodeTree]
        priorityList.sort(key=lambda lis: lis[0])
    return priorityList
  if (type(nodeTree.value) == float or type(nodeTree.value) == int):    # nó não folha
    dim = len(target) - 1
    axis = nodeTree.depth % dim;
    if (target[axis] < nodeTree.value):
      path = nodeTree.left_tree
      otherPath = nodeTree.right_tree
      priorityList = knn(path, priorityList, target, kN)
    else:
      otherPath = nodeTree.left_tree
      path = nodeTree.right_tree
      priorityList = knn(path, priorityList, target, kN)
    radiusPointSquared = priorityList[-1][0] * priorityList[-1][0] # quadrado do raio da c
    Dist = target[axis] - nodeTree.value
    if ((radiusPointSquared >= Dist * Dist) or len(priorityList) < kN):
      priorityList = knn(otherPath, priorityList, target, kN)
  return priorityList

def knn_statistics(points, kN, class1, class2):
  train = points[:int(len(points)*0.7)]
  test = points[int(len(points)*0.7):]
  tree = kdTree(train)
  tN = 0 # considerando classe 1 como 0 e classe 2 como 1 para estatísticas e cálculo da m
  fN = 0
  tP = 0
```

```
    fP= 0
    possClass = [] #vetor para armazenamento das classes atribuídas e das verdadeiras classe
    for i in test:
      pLis = [] # lista de prioridade obtida
      pLis = knn(tree, pLis, i, kN)
      count1 = 0
      count2 = 0
      for j in pLis:
        aux = j[1]
        if aux.nClass == class1:
          count1 = count1 + 1
        if aux.nClass == class2:
          count2 = count2 + 1
      if count1 >= count2: #classificar como classe 1
        possClass.append((i[-1], class1))
        if (i[-1] == class1):
          tN = tN + 1
        else:
          fN = fN + 1
      else:
        possClass.append((i[-1], class2))
        if (i[-1] == class2):
          tP = tP + 1
        else:
          fP = fP + 1
    precision = tP / (tP + fP)
    recall = tP / (tP + fN)
    accuracy = (tP + tN)/ (tP + tN + fP + fN)
    return precision, recall, accuracy, possClass
```

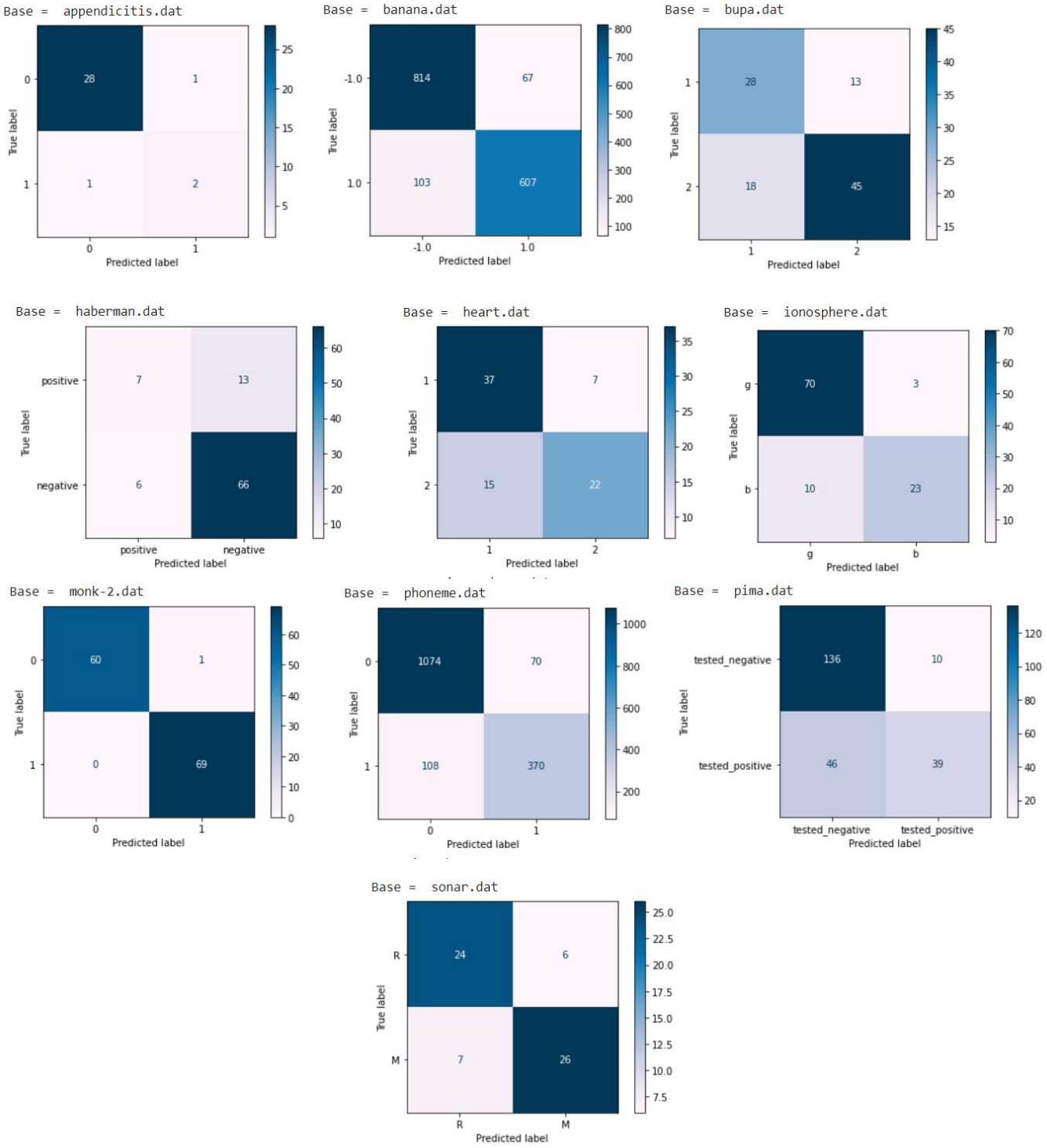
Análise dos experimentos

Para realização de experimentos, foram selecionadas 10 bases no site da ferramenta Keel. O código criado para a realização dos experimentos realiza as seguites operações para cada base: é utilizado do método "requests.get()" para pegar a base do github e extraído do arquivo o nome das classes dos pontos (sendo considerado a existência de apenas duas) e os pontos a serem avaliados, sendo esses últimos armazenados em uma lista de listas; a lista com os pontos é ordenada de forma aleatória através do comando "random.shuffle()", garantindo melhores resultados para os conjuntos de treino e teste; por fim, é executado o algoritmo de classificação pela função "knn_statistics()" várias vezes (para k variando de 2 a 10) para armazenamento das melhores estatísticas para um melhor k, sendo plotado a função de confusão para cada base através do método "confusion_matrix()" com dados retornados no cálculo do kNN.

Os dados obtidos são representados na tabela abaixo. Como verificado, apesar de haver uma considerável variação entre a acurácia notada entre as bases, os valores obtidos foram

promissores, com bases como "monk-2" apresentado resultados de aproximadamente 99.2% de

	Base Name	Best k	Accuracy	Precision	Recall
0	appendicitis.dat	2	0.937500	0.666667	0.666667
1	banana.dat	10	0.893149	0.900593	0.854930
2	bupa.dat	7	0.701923	0.775862	0.714286
3	haberman.dat	6	0.793478	0.835443	0.916667
4	heart.dat	3	0.728395	0.758621	0.594595
5	ionosphere.dat	3	0.877358	0.884615	0.696970
6	monk-2.dat	5	0.992308	0.985714	1.000000
7	phoneme.dat	3	0.890259	0.840909	0.774059
8	pima.dat	6	0.757576	0.795918	0.458824
9	sonar.dat	2	0.793651	0.812500	0.787879



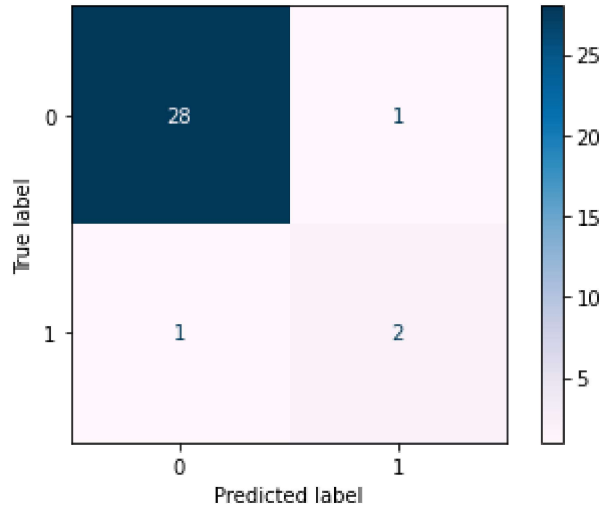

```
import matplotlib.pyplot as plt
import requests
import pandas as pd
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

tabela_comparativa = pd.DataFrame(columns=['Base Name','Best k','Accuracy','Precision','Re
Bases = ["appendicitis.dat", "banana.dat", "bupa.dat", "haberman.dat", "heart.dat", "ionos
for base in Bases:
  URL = 'https://raw.githubusercontent.com/Jhferr/Kdd/main/' + base
  Arquivo = requests.get(URL)
  file = Arquivo.text.split("\n")
  info = []
  data = []
```

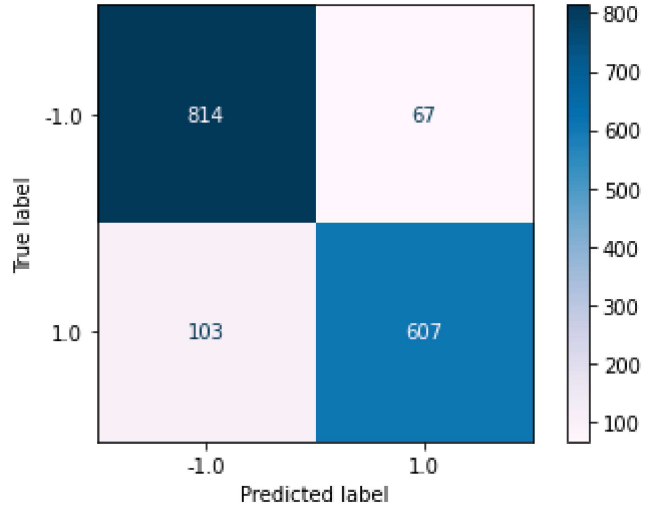
```
    for i in file:
      if "@" in i:
        info.append(i.strip().split("\n"))
      else:
        data.append(i.strip().split("\n"))
    Data = []
    data.pop()
    for i in data:
      i = i[0]
      line = i.replace(',',' ').split()
      line2 = [float(i) for i in line[:-1]]
      line2.append(line[-1])
      Data.append(line2)
    random.shuffle(Data)
    atributos = info[-4][0]
    x = atributos.find("{")
    y = atributos.find(",")
    z = atributos.find("}")
    class1 = atributos[x+1:y]
    class2 = atributos[y+1:z]
    class2 = class2.replace(" ", "")
    best = (0, 0)
    for nn in [2, 3, 4, 5, 6, 7, 8, 9, 10]:
      precision, recall, accuracy, classes = knn_statistics(Data, nn, class1, class2)
      if accuracy > best[0]:
        best = (accuracy, nn)
    precision, recall, accuracy, classes = knn_statistics(Data, best[1], class1, class2)
    nova_linha = {'Base Name':base,'Best k':best[1],'Accuracy':accuracy,'Precision':precisio
    tabela_comparativa = tabela_comparativa.append(nova_linha, ignore_index=True)
    print("Base = ", base)

    cm = confusion_matrix([c[0] for c in classes], [c[1] for c in classes], labels=(class1,
    dl = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=(class1, class2))
    dl.plot(cmap='PuBu')
    plt.show()
```

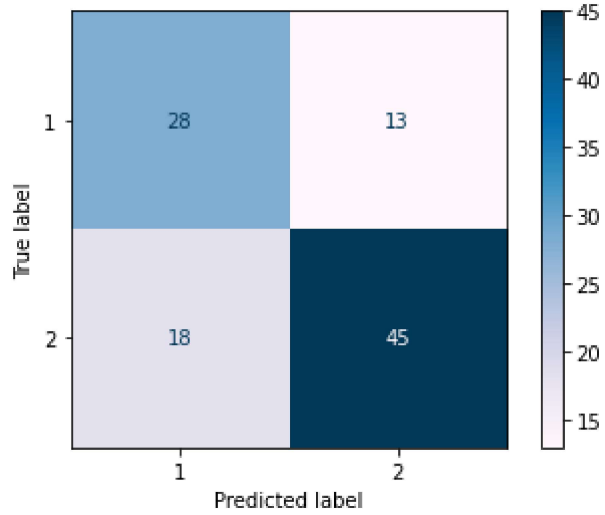
```
Base =  appendicitis.dat
```



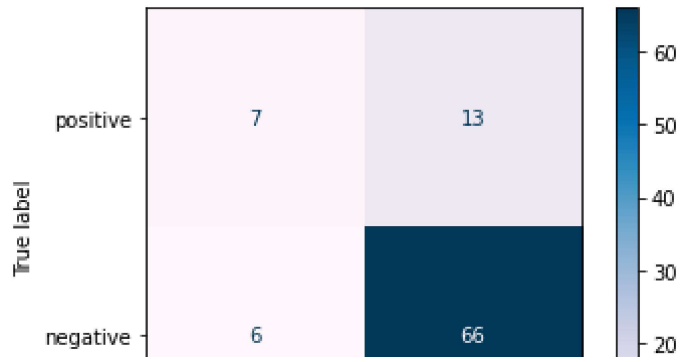

```
Base =  banana.dat
```

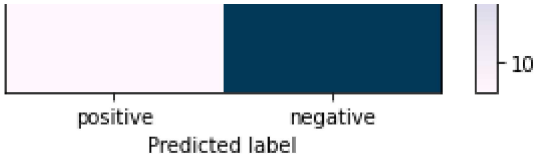



```
Base =  bupa.dat
```

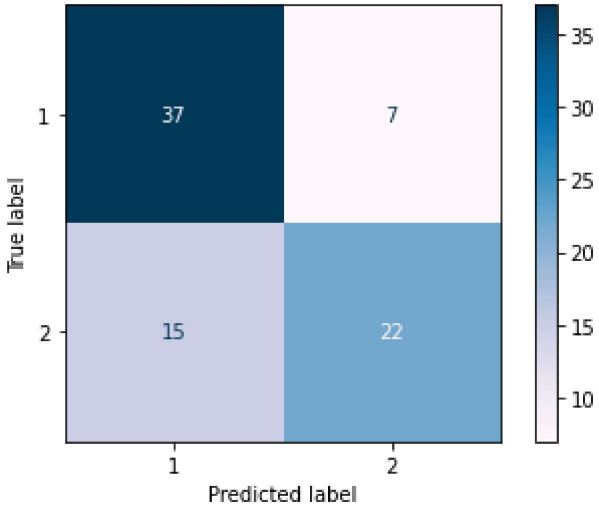



```
Base =  haberman.dat
```

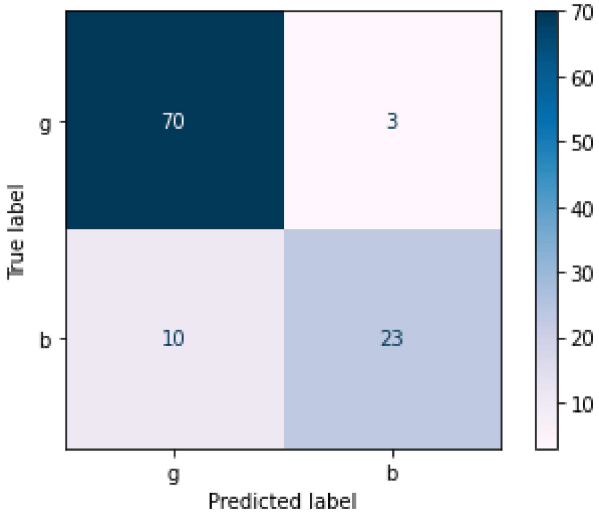




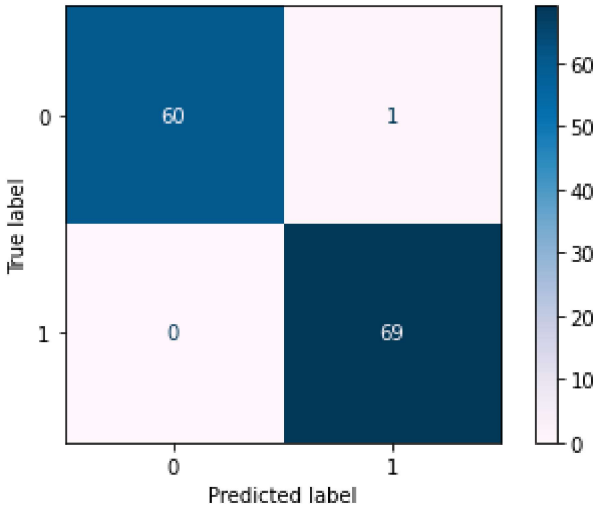
Base = heart.dat



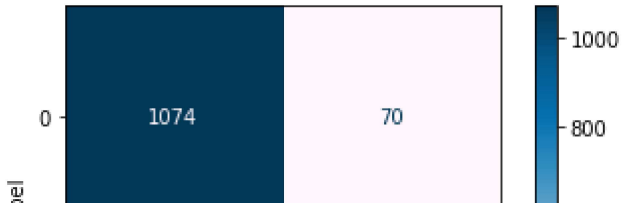
Base = ionosphere.dat

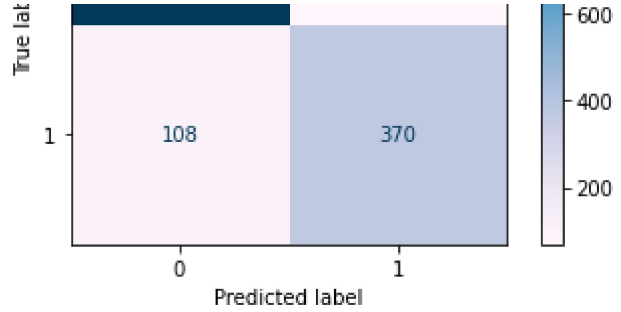


Base = monk-2.dat

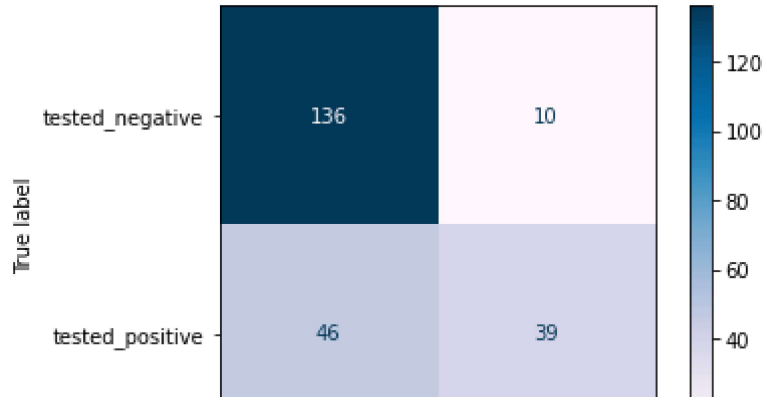


Base = phoneme.dat






```
Base =  pima.dat
```




```
tabela_comparativa.head(50)
```

	Base Name	Best k	Accuracy	Precision	Recall
0	appendicitis.dat	2	0.937500	0.666667	0.666667
1	banana.dat	10	0.893149	0.900593	0.854930
2	bupa.dat	7	0.701923	0.775862	0.714286
3	haberman.dat	6	0.793478	0.835443	0.916667
4	heart.dat	3	0.728395	0.758621	0.594595
5	ionosphere.dat	3	0.877358	0.884615	0.696970
6	monk-2.dat	5	0.992308	0.985714	1.000000
7	phoneme.dat	3	0.890259	0.840909	0.774059
8	pima.dat	6	0.757576	0.795918	0.458824
9	sonar.dat	2	0.793651	0.812500	0.787879

✓ 0s conclusão: 21:44

Não foi possível conectar-se ao serviço reCAPTCHA. Verifique sua conexão com a Internet e atualize a página para ver um desafio reCAPTCHA.