|  |  |
| --- | --- |
| **BFS**  import queue as Q  from RMP import \*  start='Arad'  goal='Bucharest'  result=''  def BFS(city, cityq, visitedq):      global result      if city==start:          result=result+' '+city      for eachcity in dict\_gn[city].keys():          if eachcity==goal:              result=result+' '+eachcity              return          if eachcity not in cityq.queue and eachcity not in visitedq.queue:              cityq.put(eachcity)              result=result+' '+eachcity      visitedq.put(city)      BFS(cityq.get(),cityq,visitedq)  def main():      cityq=Q.Queue()      visitedq=Q.Queue()      BFS(start, cityq, visitedq)      print("BFS Traversal from ",start," to ",goal," is: ")      print(result)    main() | **DFS**  import queue as Q  from RMP import dict\_gn  # Importing the city graph  start = 'Arad'  goal = 'Bucharest'  result = ''  def DLS(city, visited\_stack, stack\_limit, end\_limit):      global result      found = 0      result += city + ' '      visited\_stack.append(city)      if city == goal:          return 1      if stack\_limit == end\_limit:          return 0      for each\_city in dict\_gn[city].keys():          if each\_city not in visited\_stack:              found = DLS(each\_city, visited\_stack, stack\_limit + 1, end\_limit)              if found:                  return found  def IDDFS(city, visited\_stack, end\_limit):      global result      for i in range(0, end\_limit):          print("Searching at limit:", i)          found = DLS(city, visited\_stack, 0, i)          if found:              print("Found")              break          else:              print("Not Found!")              print('.......')              print(result)              result = ''              visited\_stack.clear()  # Clear visited\_stack instead of resetting it to an empty list  def main():      visited\_stack = []      IDDFS(start, visited\_stack, 9)      print("IDDFS Traversal from",start,"to",goal,"is:")  main() |

|  |  |
| --- | --- |
| **A\* Star**  import queue as Q  from RMP import \*  start='Arad'  goal='Bucharest'  result=' '  def get\_fn(citystr):      cities=citystr.split(",")      hn=gn=0      for ctr in range(0, len(cities)- 1 ):          gn=gn+dict\_gn[cities[ctr]][cities[ctr+1]]      hn=dict\_hn[cities[len(cities)-1]]      return(hn+gn)  def expand(cityq):      global result      tot,citystr, thiscity=cityq.get()      if thiscity==goal:          result=citystr+"::"+str(tot)          return      for cty in dict\_gn[thiscity]:           cityq.put((get\_fn(citystr+","+cty),citystr+","+cty,cty))      expand(cityq)  def main():      cityq=Q.PriorityQueue()      thiscity=start      cityq.put((get\_fn(start),start,thiscity))      expand(cityq)      print("The A\* path with the total is: ")      print(result)  main() | **Recursive best search**  import queue as Q  from RMP import \*  start='Arad'  goal='Bucharest'  result=''  def get\_fn(citystr):      cities=citystr.split(',')      hn=gn=0      for ctr in range(0,len(cities)-1):          gn=gn+dict\_gn[cities[ctr]][cities[ctr+1]]      hn=dict\_hn[cities[len(cities)-1]]      return(hn+gn)  def printout(cityq):      for i in range(0,cityq.qsize()):          print(cityq.queue[i])  def expand(cityq):      global result      tot,citystr,thiscity=cityq.get()      nexttot=999      if not cityq.empty():          nexttot,nextcitystr,nextthiscity=cityq.queue[0]      if thiscity==goal and tot<nexttot:          result=citystr+'::'+str(tot)          return      print("Expanded city------------------------------",thiscity)      print("Second best f(n)------------------------------",nexttot)      tempq=Q.PriorityQueue()      for cty in dict\_gn[thiscity]:              tempq.put((get\_fn(citystr+','+cty),citystr+','+cty,cty))      for ctr in range(1,3):          ctrtot,ctrcitystr,ctrthiscity=tempq.get()          if ctrtot<nexttot:              cityq.put((ctrtot,ctrcitystr,ctrthiscity))          else:              cityq.put((ctrtot,citystr,thiscity))              break      printout(cityq)      expand(cityq)  def main():      cityq=Q.PriorityQueue()      thiscity=start      cityq.put((999,"NA","NA"))      cityq.put((get\_fn(start),start,thiscity))      expand(cityq)      print(result)  main() |

|  |  |
| --- | --- |
| **Decision Tree**  import numpy as np  import pandas as pd  from sklearn.datasets import load\_iris  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn.tree import DecisionTreeClassifier  from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, classification\_report  iris = load\_iris()  iris\_df = pd.DataFrame(data=np.c\_[iris['data'], iris['target']], columns=iris['feature\_names'] + ['target'])  binary\_df = iris\_df[iris\_df['target'] != 2]  X = binary\_df.drop('target', axis=1)  y = binary\_df['target']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  logistic\_model = LogisticRegression()  logistic\_model.fit(X\_train, y\_train)  y\_pred\_logistic = logistic\_model.predict(X\_test)  print("Logistic Regression Metrics")  print("Accuracy:", accuracy\_score(y\_test, y\_pred\_logistic))  print("Precision:", precision\_score(y\_test, y\_pred\_logistic))  print("Recall:", recall\_score(y\_test, y\_pred\_logistic))  print("Classification Report:")  print(classification\_report(y\_test, y\_pred\_logistic)) | **Feed Forward**  from doctest import OutputChecker  import numpy as np  class NeuralNetwork():      def \_\_init\_\_(self):          np.random.seed()          self.synaptic\_weights=2\*np.random.random((3,1))-1      def sigmoid(self,x):          return 1/(1+np.exp(-x))      def sigmoid\_derivative(self,x):          return x\*(1-x)      def train(self, training\_inputs, training\_outputs, training\_iterations):          for iteration in range(training\_iterations):              output=self.think(training\_inputs)              error = training\_outputs-output              adjustments=np.dot(training\_inputs.T,error\*self.sigmoid\_derivative (output))              self.synaptic\_weights +=adjustments      def think(self, inputs):          inputs=inputs.astype(float)          output=self.sigmoid(np.dot(inputs, self.synaptic\_weights))          return output      def think(self, inputs):          inputs=inputs.astype(float)          output=self.sigmoid(np.dot(inputs, self.synaptic\_weights))          return output  if \_\_name\_\_ == "\_\_main\_\_":      neural\_network = NeuralNetwork()      print("Beginning Randomly Generated Weights: ")      print (neural\_network.synaptic\_weights)      training\_inputs = np.array([[0,0,1],                                  [1,1,1],                                  [1,0,1],                                  [0,1,1]])      training\_outputs = np.array([[0,1,1,0]]).T      neural\_network.train(training\_inputs, training\_outputs, 15000)      print("Ending weights After Training: ")      print (neural\_network.synaptic\_weights)      user\_input\_one = str(input("User Input One: "))      user\_input\_two = str(input("User Input Two: "))      user\_input\_three = str(input("User Input Three: "))      print("Considering New Situation:", user\_input\_one, user\_input\_two, user\_input\_three)      print(" New Output data: ")      print(neural\_network.think(np.array([user\_input\_one, user\_input\_two, user\_input\_three]))) |

|  |  |
| --- | --- |
| **SVM**  from sklearn import datasets  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import accuracy\_score  from sklearn.svm import SVC  iris = datasets.load\_iris()  X = iris.data  y = iris.target  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  svm\_classifier = SVC()  svm\_classifier.fit(X\_train, y\_train)  y\_pred = svm\_classifier.predict(X\_test)  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Accuracy: {accuracy \* 100:.2f}%") | **AssemblyLearnig**  import pandas  from sklearn import model\_selection  from sklearn.ensemble import AdaBoostClassifier  url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"  names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']  dataframe = pandas.read\_csv(url, names=names)  array = dataframe.values  X = array[:,0:8]  Y = array[:,8]  seed = 7  num\_trees = 30  model = AdaBoostClassifier(n\_estimators=num\_trees, random\_state=seed)  results = model\_selection.cross\_val\_score(model, X, Y)  print(results.mean()) |

|  |  |
| --- | --- |
| **KNN**  import numpy as np  from sklearn.datasets import load\_iris  from sklearn.model\_selection import train\_test\_split  from sklearn.neighbors import KNeighborsClassifier  from sklearn.metrics import accuracy\_score  iris = load\_iris()  X = iris.data  y = iris.target  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,  random\_state=42)  k = 3  knn = KNeighborsClassifier(n\_neighbors=k)  knn.fit(X\_train, y\_train)  y\_pred = knn.predict(X\_test)  accuracy = accuracy\_score(y\_test, y\_pred)  print(f'Accuracy: {accuracy \* 100:.2f}%') | **Naïve Bayes**  rwp\_examples = [      {'Alt': 'Y', 'Bar': 'N', 'Fri': 'N', 'Hun': 'Y', 'Pat': 'S', 'Price': '$$$', 'Rain': 'N', 'Res': 'Y', 'Type': 'F', 'Est': '0-10', 'ans': 'Y'},      {'Alt': 'Y', 'Bar': 'N', 'Fri': 'N', 'Hun': 'Y', 'Pat': 'F', 'Price': '$', 'Rain': 'N', 'Res': 'N', 'Type': 'T', 'Est': '30-60', 'ans': 'N'},      {'Alt': 'N', 'Bar': 'Y', 'Fri': 'N', 'Hun': 'N', 'Pat': 'S', 'Price': '$', 'Rain': 'N', 'Res': 'N', 'Type': 'B', 'Est': '0-10', 'ans': 'Y'}  ]  total\_exp = len(rwp\_examples)  def tot(attribute, value):      return sum(1 for ex in rwp\_examples if ex[attribute] == value)  def getProbab(attribute, attribval, value):      count = sum(1 for ex in rwp\_examples if ex[attribute] == attribval and ex['ans'] == value)      total\_ans\_value = tot('ans', value)      return count / total\_ans\_value if total\_ans\_value != 0 else 0  def calculate\_probability(attribute, values):      for val in values:          total\_attribute\_value = tot(attribute, val)          if total\_attribute\_value == 0:              print(f"No data for {attribute} = {val}")              continue          for ans in ['Y', 'N']:              prob = (getProbab(attribute, val, ans) \* tot('ans', ans) / total\_attribute\_value) \* 100              print(f"{ans}: Will Wait {prob:.2f}% if {attribute} = {val}")  def main():      conditions = {          'Alt': ['Y', 'N'],          'Est': ['0-10', '10-30', '30-60', '>60'],          'Pat': ['S', 'N', 'F'],          'Type': ['T']      }      for attr, vals in conditions.items():          print(f"\nProbability for Will Wait based on {attr}:")          calculate\_probability(attr, vals)  main() |