A*

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n is None:
            print('Path does not exist!')
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        open_set.remove(n)
        closed_set.add(n)
        if n in Graph_nodes:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
    print('Path does not exist!')
    return None
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
    H_dist = {'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'G': 0}
    return H_dist[n]
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)]
}
aStarAlgo('A', 'G')
```

basyian

```
import math
import ensamble
import csv
def encode_class(mydata):
    classes = []
    for i in range(len(mydata)):
        if mydata[i][-1] not in classes:
            classes.append(mydata[i][-1])
    for i in range(len(classes)):
        for j in range(len(mydata)):
            if mydata[j][-1] == classes[i]:
                mydata[j][-1] = i
    return mydata
def splitting(mydata, ratio):
    train_num = int(len(mydata) * ratio)
    train = []
    test = list(mydata)
    while len(train) < train_num:
        index = random.randrange(len(test))
        train.append(test.pop(index))
    return train, test
def groupUnderClass(mydata):
    mydict = {}
    for i in range(len(mydata)):
        if mydata[i][-1] not in mydict:
            mydict[mydata[i][-1]] = []
        mydict[mydata[i][-1]].append(mydata[i])
    return mydict
def mean(numbers):
    return sum(numbers) / float(len(numbers))
def std_dev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers)
- 1)
    return math.sqrt(variance)
def MeanAndStdDev(mydata):
    info = [(mean(attribute), std_dev(attribute)) for attribute in
zip(*mydata)]
    del info[-1]
    return info
def MeanAndStdDevForClass(mydata):
    info = {}
    mydict = groupUnderClass(mydata)
    for classValue, instances in mydict.items():
        info[classValue] = MeanAndStdDev(instances)
    return info
def calculateGaussianProbability(x, mean, stdev):
    expo = math.exp(-(math.pow(x - mean, 2) / (2 * math.pow(stdev, 2))))
    return (1 / (math.sqrt(2 * math.pi) * stdev)) * expo
def calculateClassProbabilities(info, test):
    probabilities = {}
    for classValue, classSummaries in info.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, std_dev = classSummaries[i]
            x = test[i]
            probabilities[classValue] *= calculateGaussianProbability(x,
mean, std_dev)
    return probabilities
```

```
def predict(info, test):
    probabilities = calculateClassProbabilities(info, test)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.items():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel
def getPredictions(info, test):
    predictions = []
    for i in range(len(test)):
        result = predict(info, test[i])
        predictions.append(result)
    return predictions
def accuracy_rate(test, predictions):
    correct = 0
    for i in range(len(test)):
        if test[i][-1] == predictions[i]:
            correct += 1
    return (correct / float(len(test))) * 100.0
filename = r'E:\pythonProject1\pima-indians-diabetes.csv'
with open(filename, "r") as file:
    mydata = list(csv.reader(file))
mydata = encode_class(mydata)
mydata = [[float(x) for x in row] for row in mydata]
ratio = 0.7
train_data, test_data = splitting(mydata, ratio)
print('Total number of examples:', len(mydata))
print('Out of these, training examples:', len(train_data))
print('Test examples:', len(test_data))
info = MeanAndStdDevForClass(train_data)
predictions = getPredictions(info, test_data)
accuracy = accuracy_rate(test_data, predictions)
print('Accuracy of your model is:', accuracy)
```

baysian net

```
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.inference import VariableElimination
from pgmpy.factors.discrete import TabularCPD
import numpy as np
model = BayesianModel([('C', 'S'), ('D', 'S')])
cpd_c = TabularCPD('C', 2, [[0.5], [0.5]])
cpd_d = TabularCPD('D', 2, [[0.5], [0.5]])
cpd_s = TabularCPD('S', 2, [[0.8, 0.6, 0.6, 0.2], [0.2, 0.4, 0.4, 0.8]],
                   evidence=['C', 'D'], evidence_card=[2, 2])
model.add_cpds(cpd_c, cpd_d, cpd_s)
data = np.random.randint(low=0, high=2, size=(5000, 2))
mle = MaximumLikelihoodEstimator(model, data)
model_fit = mle.fit()
infer = VariableElimination(model)
```

```
query = infer.query(['S'], evidence={'C': 1})
print(query)
```

Bfs

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = []
queue = []
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
print("Following is the BFS")
bfs(visited, graph, '5')
```

dfs

```
graph = {
  '5':['3','7'],
  '3':['2', '4'],
  '7':['8'],
  '2':[],
  '4':['8'],
  '8':[]
}
visited = set()
def dfs(visited,graph,node):
    if node not in visited:
```

```
        print(node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

build desion tree random forest(br)

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
X_train, X_test, y_train, y_test = train_test_split(df[iris.feature_names],
df['target'], test_size=0.3, random_state=0)
rfc = RandomForestClassifier(n_estimators=100, max_depth=2, random_state=0)
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
importances = rfc.feature_importances_
indices = list(range(len(importances)))
plt.bar(indices, importances, color='black')
plt.xticks(indices, iris.feature_names, rotation=90)
plt.title('Feature Importance')
plt.show()
```

cluster

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans, AgglomerativeClustering
import matplotlib.pyplot as plt
X, y = make_blobs(n_samples=100, centers=4, random_state=42)
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(X)
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_)
plt.title("K-Means Clustering")
plt.show()
hierarchical = AgglomerativeClustering(n_clusters=4)
hierarchical.fit(X)
plt.scatter(X[:, 0], X[:, 1], c=hierarchical.labels_)
plt.title("Hierarchical Clustering")
plt.show()
```

deepnn

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train / 255.0
x_test = x_test / 255.0
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10)
])
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print('Test accuracy:', test_acc)
```

dtree

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt
iris = load_iris()
clf = DecisionTreeClassifier(random_state=0)
clf.fit(iris.data, iris.target)
plt.figure(figsize=(10, 8))
plot_tree(clf, filled=True)
plt.show()
```

ensamble

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.3, random_state=0)
svc_model = SVC(kernel='linear', probability=True)
rf_model = RandomForestClassifier(n_estimators=10)
lr_model = LogisticRegression()
ensemble = VotingClassifier(estimators=[('svc', svc_model), ('rf',
rf_model), ('lr', lr_model)], voting='soft')
ensemble.fit(X_train, y_train)
```

```
y_pred = ensemble.predict(X_test)
print("Ensemble Accuracy:", ensemble.score(X_test, y_test))
```

linear

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
x = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 6, 8, 10])
regressor = LinearRegression()
regressor.fit(x, y)
y_pred = regressor.predict(x)
print('Coefficients:', regressor.coef_)
print('Intercept:', regressor.intercept_)
plt.scatter(x, y, color='black')
plt.plot(x, y_pred, color='blue', linewidth=3)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

logistic

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
X = np.array([[1, 2], [2, 3], [4, 5], [5, 6]])
y = np.array([0, 0, 1, 1])
classifier = LogisticRegression()
classifier.fit(X, y)
print('Coefficient:', classifier.coef_)
print('Intercept:', classifier.intercept_)
xx, yy = np.meshgrid(np.arange(0, 6, 0.01), np.arange(0, 8, 0.01))
z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
z = z.reshape(xx.shape)
plt.contourf(xx, yy, z, cmap=plt.cm.RdBu, alpha=0.8)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu_r, edgecolors='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Logistic Regression')
plt.show()
```

memory bound a*

```
from queue import PriorityQueue
def memory_bounded_astar(start, goal, heuristic, successors, memory_limit):
    frontier = PriorityQueue()
    frontier.put((0, start))
    g_scores = {start: 0}
    came_from = {start: None}
    while not frontier.empty():
```

```
        _, current = frontier.get()
        if current == goal:
            path = []
            while current is not None:
                path.append(current)
                current = came_from[current]
            return path[::-1]
        for next_node in successors(current):
            new_g_score = g_scores[current] + 1
            if next_node not in g_scores or new_g_score <
g_scores[next_node]:
                g_scores[next_node] = new_g_score
                f_score = new_g_score + heuristic(next_node, goal)
                frontier.put((f_score, next_node))
                came_from[next_node] = current
        if len(g_scores) > memory_limit:
            return None
    return None
def heuristic(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])
def successors(node):
    x, y = node
    return [(x + 1, y), (x, y + 1)]
start = (0, 0)
goal = (3, 3)
memory_limit = 50
path = memory_bounded_astar(start, goal, heuristic, successors,
memory_limit)
if path is not None:
    print("Path found:", path)
else:
    print("Memory limit exceeded, nopathfound.")
```

navie

```
from sklearn.datasets import load_iris
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.3, random_state=4)
nb = GaussianNB()
nb.fit(X_train, y_train)
y_pred = nb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}%".format(accuracy * 100))
```

nn

```
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train / 255.0
x_test = x_test / 255.0
model = keras.Sequential([
```

```
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print('Test accuracy:', test_acc)
```

rforest

```
import pandas as pd
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
data = pd.read_csv('data.csv')
X = data.drop(['target'], axis=1)
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
dt = DecisionTreeRegressor()
dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Decision Tree Mean Squared Error: {mse:.4f}")
rf = RandomForestRegressor()
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Random Forest Mean Squared Error: {mse:.4f}")
```

svm

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
data = pd.read_csv("apples_and_oranges.csv")
X = data.iloc[:, 0:2].values
Y = data.iloc[:, 2].values
le = LabelEncoder()
Y = le.fit_transform(Y)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=1)
classifier = SVC(kernel='rbf', random_state=1)
classifier.fit(X_train, Y_train)
```

```
Y_pred = classifier.predict(X_test)
cm = confusion_matrix(Y_test, Y_pred)
accuracy = float(cm.diagonal().sum()) / len(Y_test)
print("Accuracy of SVM for the given dataset: ", accuracy)
plt.figure(figsize=(7, 7))
X_set, Y_set = X_train, Y_train
X1, X2 = np.meshgrid(np.arange(start=X_set[:, 0].min()-1, stop=X_set[:,
0].max()+1, step=0.01),
                     np.arange(start=X_set[:, 1].min()-1, stop=X_set[:,
1].max()+1, step=0.01))
Z = classifier.predict(np.array([X1.ravel(), X2.ravel()]).T)
Z = Z.reshape(X1.shape)
plt.contourf(X1, X2, Z, alpha=0.75, cmap=ListedColormap(('black',
'white')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(Y_set)):
    plt.scatter(X_set[Y_set == j, 0], X_set[Y_set == j, 1],
color=ListedColormap(('red', 'orange'))(i), label=j)
plt.title('Apples Vs Oranges')
plt.xlabel('Weight in grams')
plt.ylabel('Size in cm')
plt.legend()
plt.show()
```