a* algorithm

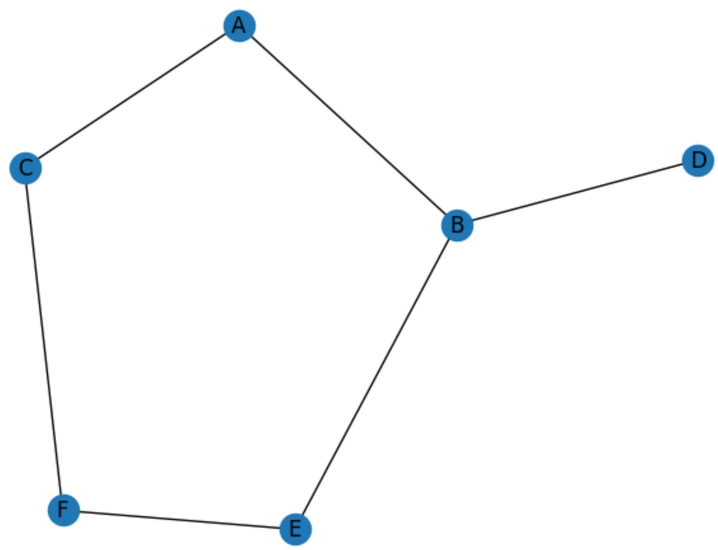
```
import heapq
import matplotlib.pyplot as plt
import networkx as nx

def draw_graph(graph):
    G = nx.Graph()
    for node, edges in graph.items():
        for edge in edges:
            G.add_edge(node, edge)
    nx.draw(G, with_labels=True)
    plt.show()

def a_star(graph, start, goal):
    queue = [(0, start, [])]
    seen = set()
    while queue:
        (cost, node, path) = heapq.heappop(queue)
        if node not in seen:
            path = path + [node]
            seen.add(node)
            if node == goal:
                return path
            for next_node in graph[node]:
                if next_node not in seen:
                    heapq.heappush(queue, (cost + 1, next_node, path))
    return []
```

```
graph = {'A': {'B', 'C'}, 'B': {'A', 'D', 'E'}, 'C': {'A', 'F'}, 'D': {'B'}, 'E': {'B', 'F'}, 'F': {'C', 'E'}}
draw_graph(graph)
print(a_star(graph, 'A', 'D'))
```

OUTPUT



['A', 'B', 'D']

jug problem

```
from collections import deque

def BFS(jug1, jug2, target):
    visited, path, queue = {}, [], deque([(0, 0)])

    while queue:
        state = queue.popleft()
```

```
        if state == target:
            if state not in visited: path.append(state)
            visited[state] = 1
            break
        if state not in visited:
            path.append(state)
            visited[state] = 1
            if state[0] > 0:
                queue.extend([(0, state[1]), (state[0] - min(state[0], jug2 - state[1]), state[1] + min(state[0],
jug2 - state[1]))])
            if state[1] > 0:
                queue.extend([(state[0], 0), (state[0] + min(state[1], jug1 - state[0]), state[1] - min(state[1],
jug1 - state[0]))])
            if state[0] < jug1: queue.append((jug1, state[1]))
            if state[1] < jug2: queue.append((state[0], jug2))

    print("No solution" if not visited.get(target) else "Steps:\n" + '\n'.join(map(str, path)))

BFS(4, 3, (2, 0))
```

OUITPUT

Steps: (0, 0)

(4, 0)

(0, 3)

(1, 3)

(4, 3)

(3, 0)

(1, 0)

(3, 3)

(0, 1)

(4, 2)

(4, 1)

(0, 2)

(2, 3)

(2, 0)

Classifier Building in Scikit-learn

Naive Bayes Classifier with Synthetic Dataset

In the first example, we will generate synthetic data using scikit-learn and train and evaluate the Gaussian Naive Bayes algorithm.

Generating the Dataset

Scikit-learn provides us with a machine learning ecosystem so that you can generate the dataset and evaluate various machine learning algorithms.

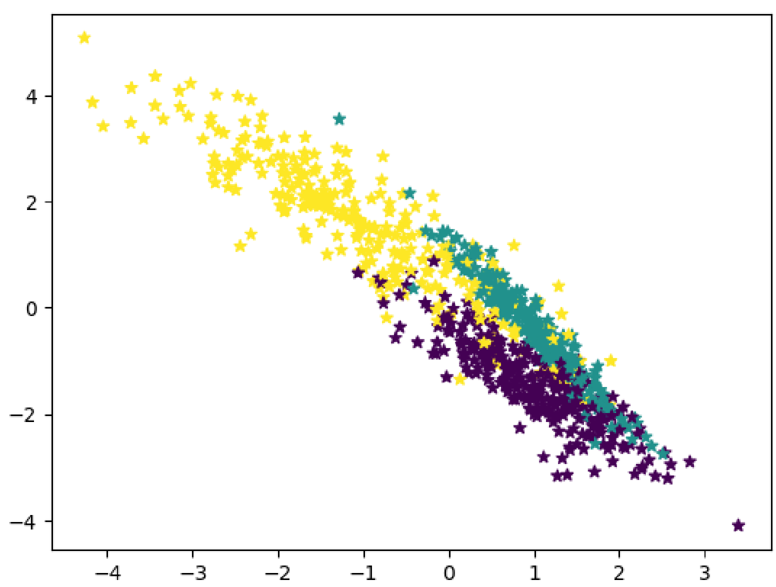
In our case, we are creating a dataset with six features, three classes, and 800 samples using the `make_classification` function.

```
from sklearn.datasets import make_classification
X, y = make_classification(
n_features=6, n_classes=3, n_samples=800, n_informative=2, random_state=1,
n_clusters_per_class=1, )
```

use matplotlib.pyplot's `scatter` function to visualize the dataset.

```
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], c=y, marker="*");
```

OUITPUT



Train Test Split

Before we start the training process, we need to split the dataset into training and testing for model evaluation.

```
from sklearn.model_selection import train_test_split X_train, X_test, y_train,
y_test = train_test_split( X, y, test_size=0.33, random_state=125 )
```

Model Building and Training

Build a generic Gaussian Naive Bayes and train it on a training dataset. After that, feed a random test sample to the model to get a predicted value.

```
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.naive_bayes import GaussianNB # Build a Gaussian Classifier model =
GaussianNB() # Model training model.fit(X_train, y_train) # Predict Output
predicted = model.predict([X_test[6]]) print("Actual Value:", y_test[6])
print("Predicted Value:", predicted[0])
```

Both actual and predicted values are the same.

OUITPUT

```
Actual Value: 0

Predicted Value: 0
```

Model Evaluation

We will not evolve the model on an unseen test dataset. First, we will predict the values for the test dataset and use them to calculate accuracy and F1 score.

```
from sklearn.metrics import ( accuracy_score, confusion_matrix,
ConfusionMatrixDisplay, f1_score, )
```

```
y_pred = model.predict(X_test) accuray = accuracy_score(y_pred, y_test) f1 =
f1_score(y_pred, y_test, average="weighted")

print("Accuracy:", accuray)

print("F1 Score:", f1)
```

Our model has performed fairly well with default hyperparameters.

OUITPUT

```
Accuracy: 0.8484848484848485

F1 Score: 0.8491119695890328
```

To visualize the Confusion matrix, we will use `confusion_matrix` to calculate the true positives and true negatives and `ConfusionMatrixDisplay` to display the confusion matrix with the labels.

```
labels = [0,1,2]

cm = confusion_matrix(y_test, y_pred, labels=labels)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
disp.plot();
```

k-Nearest Neighbors

Load the data

In this example, we use the iris dataset. We split the data into a train and test dataset.

```
from sklearn.datasets import load_iris

from sklearn.metrics import *
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

iris = load_iris(as_frame=True)
# Split the data into features (X) and target (y)
X = iris.data[["sepal length (cm)", "sepal width (cm)"]]
y = iris.target

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
stratify=y, random_state=0)
```

```
# Scale the features using StandardScaler scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Fitting and Evaluating the Model

We are now ready to train the model. For this, we'll use a fixed value of 3 for k, but we'll need to optimize this later on. We first create an instance of the kNN model, then fit this to our training data. We pass both the features and the target variable, so the model can learn.

```
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
```

The model is now trained! We can make predictions on the test dataset, which we can use later to score the model.

```
y_pred = knn.predict(X_test)
```

The simplest way to evaluate this model is by using accuracy. We check the predictions against the actual values in the test set and count up how many the model got right.

```
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

OUTPUT Accuracy: 0.7631578947368421
----
refer the link
https://github.com/suneet10/DataPreprocessing/blob/main/Data_Preprocessing.ipynb

Steps in Data Preprocessing:

In this article, We'll be covering the following steps:

- Importing the libraries
- Importing the dataset
- Taking care of missing data
- Encoding categorical data
- Normalizing the data
- Splitting the data into test and train

Step 1: Importing the libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Step 2: Importing the dataset

```
data = pd.read_csv('https://raw.githubusercontent.com/suneet10/DataPreprocessing/main/
Data.csv')
data
```

output

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
4	Germany	40.0	NaN	Yes
5	France	35.0	58000.0	Yes
6	Spain	NaN	52000.0	No

7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

In any dataset used for machine learning, there are two types of variables:

- •Independent variable
- •Dependent variable

The **independent variable** is the columns that we are going to use to predict the **dependent variable**, or in other words, the independent variable **affects** the dependent variable

Step 3: Handling the missing values

As you can see in our dataset we have two missing values one in the *Salary* column in the 5th Row and another in the *Age* column of the 7th row.

```
from sklearn.impute import SimpleImputer

# 'np.nan' signifies that we are targeting missing values
# and the strategy we are choosing is replacing it with 'mean'
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')

imputer.fit(data.iloc[:, 1:3])
data.iloc[:, 1:3] = imputer.transform(data.iloc[:, 1:3])

# print the dataset
data
```

output

	Country	Age	Salary	Purchased
0	France	44.000000	72000.000000	No
1	Spain	27.000000	48000.000000	Yes
2	Germany	30.000000	54000.000000	No
3	Spain	38.000000	61000.000000	No
4	Germany	40.000000	63777.777778	Yes
5	France	35.000000	58000.000000	Yes
6	Spain	38.777778	52000.000000	No

7	France	48.000000	79000.000000	Yes
8	Germany	50.000000	83000.000000	No
9	France	37.000000	67000.000000	Yes

Step 4: Encoding categorical data

In our case, we have two categorical columns, the *country* column, and the *purchased* column.

- **OneHot Encoding**

In the *country* column, we have three different categories: France, Germany, Spain. We can simply label France as 0, Germany as 1, and Spain as 2 but doing this might lead our machine learning model to interpret that there is some correlation between these numbers and the outcome.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])],
remainder='passthrough')
# [0] signifies the index of the column we are appliying the encoding on
data = pd.DataFrame(ct.fit_transform(data))
data
```

out put

	0	1	2	3	4	5
0	1.0	0.0	0.0	44.0	72000.0	No
1	0.0	0.0	1.0	27.0	48000.0	Yes
2	0.0	1.0	0.0	30.0	54000.0	No
3	0.0	0.0	1.0	38.0	61000.0	No
4	0.0	1.0	0.0	40.0	63777.777778	Yes
5	1.0	0.0	0.0	35.0	58000.0	Yes
6	0.0	0.0	1.0	38.777778	52000.0	No
7	1.0	0.0	0.0	48.0	79000.0	Yes
8	0.0	1.0	0.0	50.0	83000.0	No

```
9       1.0     0.0     0.0     37.0    67000.0       Yes
```

Label Encoding

In the last column, i.e. the purchased column, the data is in binary form meaning that there are only two outcomes either Yes or No. Therefore here we need to perform Label Encoding.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
data.iloc[:,-1] = le.fit_transform(data.iloc[:,-1])
# 'data.iloc[:,-1]' is used to select the column that we need to be encoded
data
```

output

```
        0       1       2       3       4       5
0       1.0     0.0     0.0     44.0    72000.0       0
1       0.0     0.0     1.0     27.0    48000.0       1
2       0.0     1.0     0.0     30.0    54000.0       0
3       0.0     0.0     1.0     38.0    61000.0       0
4       0.0     1.0     0.0     40.0    63777.777778        1
5       1.0     0.0     0.0     35.0    58000.0       1
6       0.0     0.0     1.0     38.777778     52000.0       0
7       1.0     0.0     0.0     48.0    79000.0       1
8       0.0     1.0     0.0     50.0    83000.0       0
9       1.0     0.0     0.0     37.0    67000.0       1
```

Step 5: Feature Scaling

Feature scaling is bringing all of the features on the dataset to the same scale, this is necessary while training a machine learning model because in some cases the **dominant features become so dominant that the other ordinary features are not even considered by the model**.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
data = pd.DataFrame(scaler.fit_transform(data))
data
```

output

	0	1	2	3 4	5	
0	1.0	0.0	0.0	0.739130	0.685714	0.0
1	0.0	0.0	1.0	0.000000	0.000000	1.0
2	0.0	1.0	0.0	0.130435	0.171429	0.0
3	0.0	0.0	1.0	0.478261	0.371429	0.0
4	0.0	1.0	0.0	0.565217	0.450794	1.0
5	1.0	0.0	0.0	0.347826	0.285714	1.0
6	0.0	0.0	1.0	0.512077	0.114286	0.0
7	1.0	0.0	0.0	0.913043	0.885714	1.0
8	0.0	1.0	0.0	1.000000	1.000000	0.0
9	1.0	0.0	0.0	0.434783	0.542857	1.0

Step 6: Splitting the dataset

Before we begin training our model there is one final step to go, which is splitting of the testing and training dataset. In machine learning, a **larger part** of the dataset is used to train the model, and a small part is used to test the trained model for finding out the accuracy and the efficiency of the model.

```
X = data.iloc[:, :-1].values y = data.iloc[:, -1].values

print("Independent Variable\n")

 print(X)

print("\nDependent Variable\n")

print(y)
```

output

Independent Variable

[[0.   1.   0.   0.   0.73913043 0.68571429]

```
[1.        0.        0.        1.        0.        0.        ]
[1.        0.        1.        0.        0.13043478 0.17142857]
[1.        0.        0.        1.        0.47826087 0.37142857]
[1.        0.        1.        0.        0.56521739 0.45079365]
[0.        1.        0.        0.        0.34782609 0.28571429]
[1.        0.        0.        1.        0.51207729 0.11428571]
[0.        1.        0.        0.        0.91304348 0.88571429]
[1.        0.        1.        0.        1.        1.        ]
[0.        1.        0.        0.        0.43478261 0.54285714]]
```

Dependent Variable

```
[0. 1. 0. 0. 1. 1. 0. 1. 0. 1.]
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
#'test_size=0.2' means 20% test data and 80% train data
print(X_train)
```

output

```
[[0.        1.        0.        0.        0.43478261 0.54285714]
 [0.        1.        0.        0.        0.34782609 0.28571429]
 [1.        0.        0.        1.        0.51207729 0.11428571]
 [0.        1.        0.        0.        0.73913043 0.68571429]
 [0.        1.        0.        0.        0.91304348 0.88571429]
 [1.        0.        1.        0.        0.56521739 0.45079365]
 [1.        0.        1.        0.        1.        1.        ]
 [1.        0.        0.        1.        0.        0.        ]]
```

```
print(X_test)
```

output

```
[[1.      0.      1.      0.      0.13043478 0.17142857]
 [1.         0.         0.         1.         0.47826087 0.37142857]]
```

ANN open this link and add some matter if needed

https://www.mltut.com/implementation-of-artificial-neural-network-in-python/

1.1 Import the Libraries-

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

1.2 Load the Dataset

```
dataset = pd.read_csv('Churn_Modelling_dataset.csv')
```

1.3 Split Dataset into X and Y

```
X = pd.DataFrame(dataset.iloc[:, 3:13].values)
y = dataset.iloc[:, 13].values
```

1.4 Encode Categorical Data

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X_2 = LabelEncoder() X.loc[:, 2] =
labelencoder_X_2.fit_transform(X.iloc[:, 2])
```

One Hot Encoding

```
labelencoder_X_1 = LabelEncoder() X.loc[:, 1] =
labelencoder_X_1.fit_transform(X.iloc[:, 1])
```

After applying label encoding, now it’s time to apply One Hot Encoding-

```
onehotencoder = OneHotEncoder(categorical_features = [1])
labelencoder_X_1 = LabelEncoder()
X.loc[:, 1] = labelencoder_X_1.fit_transform(X.iloc[:, 1])
X = onehotencoder.fit_transform(X).toarray()
X = X[:, 1:]
```

1.5 Split the X and Y Dataset into the Training set and Test set

```
from sklearn.model_selection import train_test_split X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

1.6 Perform Feature Scaling

```
from sklearn.preprocessing import StandardScaler sc = StandardScaler() X_train = sc.fit_transform(X_train) X_test = sc.transform(X_test)
```

2. Build Artificial Neural Network

2.1 Import the Keras libraries and packages

```
import keras
from keras.models import Sequential
from keras.layers import Dense
```

2.2 Initialize the Artificial Neural Network

```
classifier = Sequential()
```

2.3 Add the input layer and the first hidden layer

```
classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu', input_dim = 11))
```

2.4 Add the second hidden layer

```
classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))
```

2.5 Add the output layer

```
classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))
```

3. Train the ANN

The training part requires two steps- Compile the ANN, and Fit the ANN to the Training set. So let’s start with the first step-

3.1 Compile the ANN

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

3.2 Fit the ANN to the Training set

```
classifier.fit(X_train, y_train, batch_size = 10, nb_epoch = 100)
```

4. Predict the Test Set Results-

```
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.5)
```

5. Make the Confusion Matrix

```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test,y_pred)
```