**Steps :**

1. **Payload Detection ( Steganalysis)**
2. **Payload Extraction**
3. **Payload Classification**
4. **Payload Detection ( Steganalysis)**

**Spam:**

**Code:**

function F = spam686(IMG)

F = spam\_extract\_2(double(imread('output1.png')),3);

function F = spam\_extract\_2(X,T)

% horizontal left-right

D = X(:,1:end-1) - X(:,2:end);

L = D(:,3:end); C = D(:,2:end-1); R = D(:,1:end-2);

Mh1 = GetM3(L,C,R,T);

% horizontal right-left

D = -D;

L = D(:,1:end-2); C = D(:,2:end-1); R = D(:,3:end);

Mh2 = GetM3(L,C,R,T);

% vertical bottom top

D = X(1:end-1,:) - X(2:end,:);

L = D(3:end,:); C = D(2:end-1,:); R = D(1:end-2,:);

Mv1 = GetM3(L,C,R,T);

% vertical top bottom

D = -D;

L = D(1:end-2,:); C = D(2:end-1,:); R = D(3:end,:);

Mv2 = GetM3(L,C,R,T);

% diagonal left-right

D = X(1:end-1,1:end-1) - X(2:end,2:end);

L = D(3:end,3:end); C = D(2:end-1,2:end-1); R = D(1:end-2,1:end-2);

Md1 = GetM3(L,C,R,T);

% diagonal right-left

D = -D;

L = D(1:end-2,1:end-2); C = D(2:end-1,2:end-1); R = D(3:end,3:end);

Md2 = GetM3(L,C,R,T);

% minor diagonal left-right

D = X(2:end,1:end-1) - X(1:end-1,2:end);

L = D(1:end-2,3:end); C = D(2:end-1,2:end-1); R = D(3:end,1:end-2);

Mm1 = GetM3(L,C,R,T);

% minor diagonal right-left

D = -D;

L = D(3:end,1:end-2); C = D(2:end-1,2:end-1); R = D(1:end-2,3:end);

Mm2 = GetM3(L,C,R,T);

F1 = (Mh1+Mh2+Mv1+Mv2)/4;

F2 = (Md1+Md2+Mm1+Mm2)/4;

F = [F1;F2];

function M = GetM3(L,C,R,T)

% marginalization into borders

L = L(:); L(L<-T) = -T; L(L>T) = T;

C = C(:); C(C<-T) = -T; C(C>T) = T;

R = R(:); R(R<-T) = -T; R(R>T) = T;

% get cooccurences [-T...T]

M = zeros(2\*T+1,2\*T+1,2\*T+1);

for i=-T:T

C2 = C(L==i);

R2 = R(L==i);

for j=-T:T

R3 = R2(C2==j);

for k=-T:T

M(i+T+1,j+T+1,k+T+1) = sum(R3==k);

end

end

end

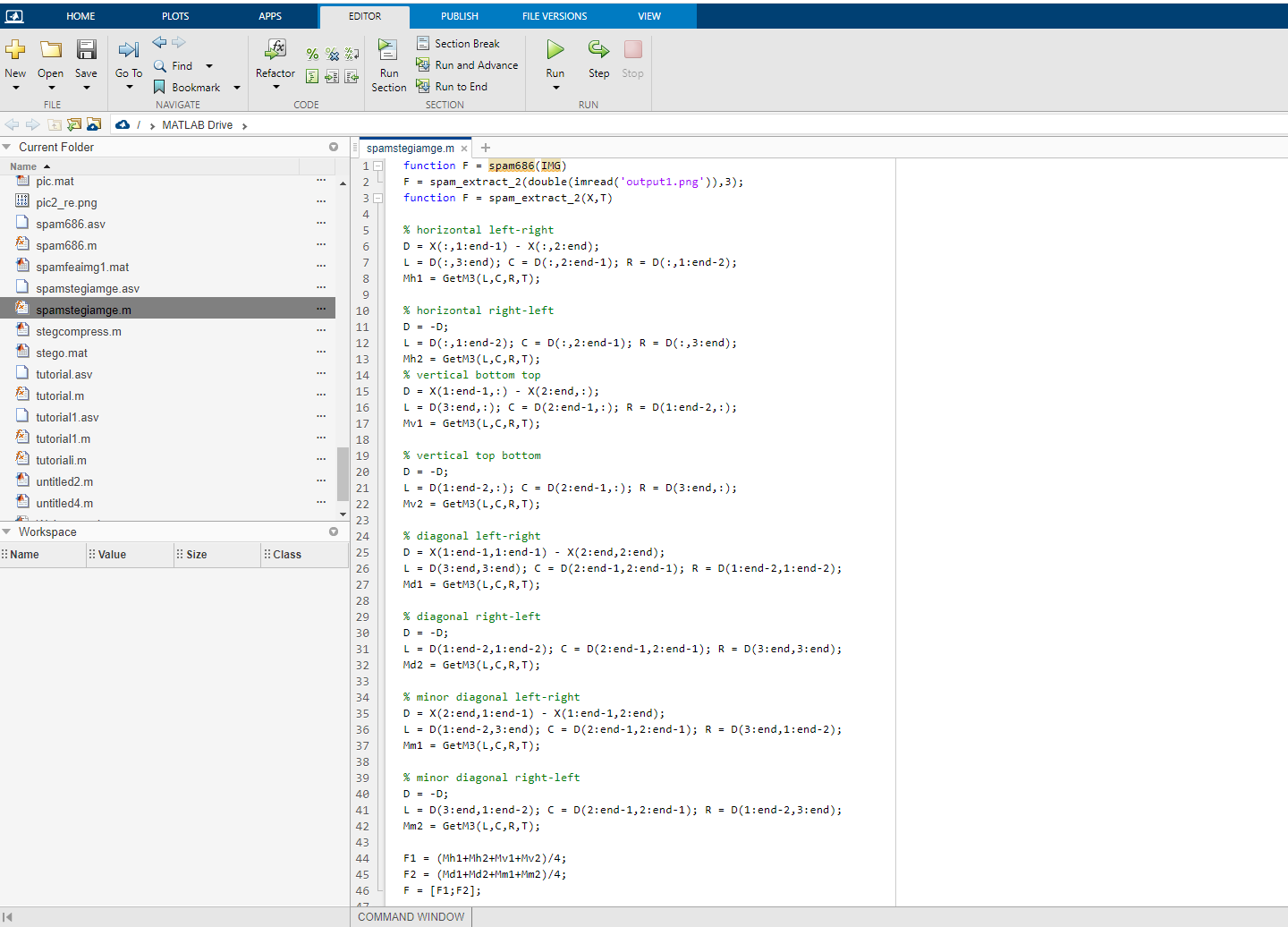
% normalization

M = M(:)/sum(M(:));

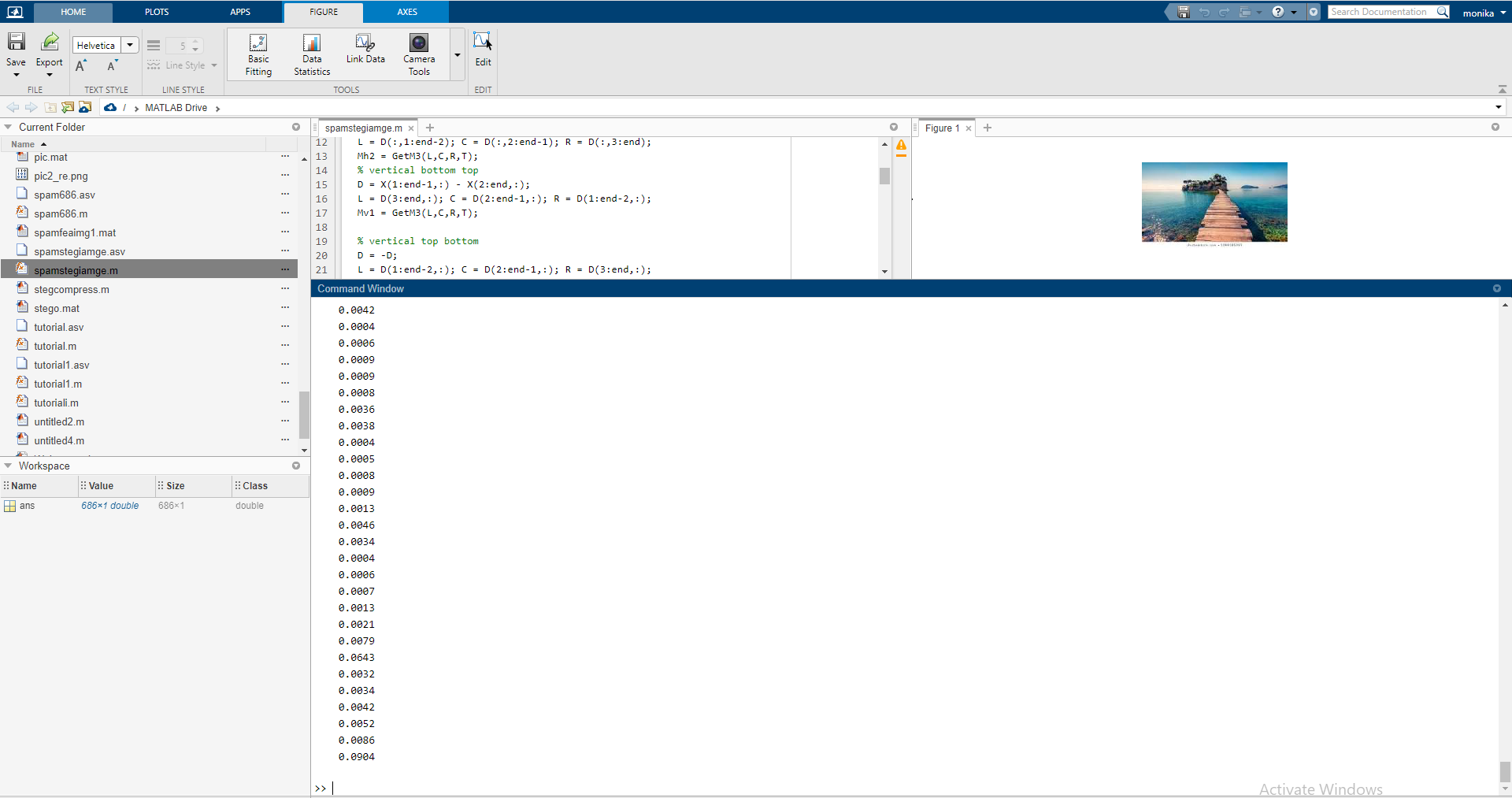
A = imread('output1.png');

imshow(A);

Code snap: //matlab



**Output snap:**

****

**Ant Colony:**

mport pandas as pd

import numpy as np

from numpy import inf

 import matplotlib.pyplot as plt

from sklearn.ensemble import ExtraTreesClassifier

#given values for the problems

# Loading the data

df = pd.read\_csv('stegotest.csv')

iteration = 100

n\_ants = 5

n\_citys = 5

# intialization part

m = n\_ants

n = n\_citys

e = .5         #evaporation rate

alpha = 1     #pheromone factor

beta = 2       #visibility factor

#calculating the visibility of the next city visibility(i,j)=1/d(i,j)

visibility = 1/d

visibility[visibility == inf ] = 0

#intializing pheromne present at the paths to the cities

pheromne = .1\*np.ones((m,n))

#intializing the rute of the ants with size rute(n\_ants,n\_citys+1)

#note adding 1 because we want to come back to the source city

rute = np.ones((m,n+1))

for ite in range(iteration):

    rute[:,0] = 1          #initial starting and ending positon of every ants '1' i.e city '1'

    for i in range(m):

        temp\_visibility = np.array(visibility)         #creating a copy of visibility

        for j in range(n-1):

            #print(rute)

            combine\_feature = np.zeros(5)     #intializing combine\_feature array to zero

            cum\_prob = np.zeros(5)            #intializing cummulative probability array to zeros

            cur\_loc = int(rute[i,j]-1)        #current city of the ant

            temp\_visibility[:,cur\_loc] = 0     #making visibility of the current city as zero

            p\_feature = np.power(pheromne[cur\_loc,:],beta)         #calculating pheromne feature

            v\_feature = np.power(temp\_visibility[cur\_loc,:],alpha)  #calculating visibility feature

            p\_feature = p\_feature[:,np.newaxis]                     #adding axis to make a size[5,1]

            v\_feature = v\_feature[:,np.newaxis]                     #adding axis to make a size[5,1]

                         combine\_feature = np.multiply(p\_feature,v\_feature)     #calculating the combine feature

            total = np.sum(combine\_feature)                        #sum of all the feature

                         probs = combine\_feature/total   #finding probability of element probs(i) = comine\_feature(i)/total

                         cum\_prob = np.cumsum(probs)     #calculating cummulative sum

            #print(cum\_prob)

            r = np.random.random\_sample()   #randon no in [0,1)

            #print(r)

            city = np.nonzero(cum\_prob>r)[0][0]+1       #finding the next city having probability higher then random(r)

            #print(city)

                         rute[i,j+1] = city              #adding city to route

        left = list(set([i for i in range(1,n+1)])-set(rute[i,:-2]))[0]     #finding the last untraversed city to route

                 rute[i,-2] = left                   #adding untraversed city to route

            rute\_opt = np.array(rute)               #intializing optimal route

         dist\_cost = np.zeros((m,1))             #intializing total\_distance\_of\_tour with zero

         for i in range(m):

                 s = 0

        for j in range(n-1):

                         s = s + d[int(rute\_opt[i,j])-1,int(rute\_opt[i,j+1])-1]   #calcualting total tour distance

                 dist\_cost[i]=s                      #storing distance of tour for 'i'th ant at location 'i'

            dist\_min\_loc = np.argmin(dist\_cost)             #finding location of minimum of dist\_cost

    dist\_min\_cost = dist\_cost[dist\_min\_loc]         #finging min of dist\_cost

    best\_route = rute[dist\_min\_loc,:]               #intializing current traversed as best route

    pheromne = (1-e)\*pheromne                       #evaporation of pheromne with (1-e)

    for i in range(m):

        for j in range(n-1):

            dt = 1/dist\_cost[i]

            pheromne[int(rute\_opt[i,j])-1,int(rute\_opt[i,j+1])-1] = pheromne[int(rute\_opt[i,j])-1,int(rute\_opt[i,j+1])-1] + dt

            #updating the pheromne with delta\_distance

            #delta\_distance will be more with min\_dist i.e adding more weight to that route  peromne

  print('route of all the ants at the end :')

# Loading the data

df = pd.read\_csv('malware.csv')

  # Seperating the dependent and independent variables

y = df['Class']

X = df.drop('Class', axis = 1)

  X.head()

# Building the model

extra\_tree\_forest = ExtraTreesClassifier(n\_estimators = 5,criterion ='entropy', max\_features = 2)

  # Training the model

extra\_tree\_forest.fit(X, y)

  # Computing the importance of each feature

feature\_importance = extra\_tree\_forest.feature\_importances\_

  # Normalizing the individual importances

feature\_importance\_normalized = np.std([tree.feature\_importances\_ for tree in extra\_tree\_forest.estimators\_],axis = 0)

# Plotting a Bar Graph to compare the models

plt.bar(X.columns, feature\_importance\_normalized)

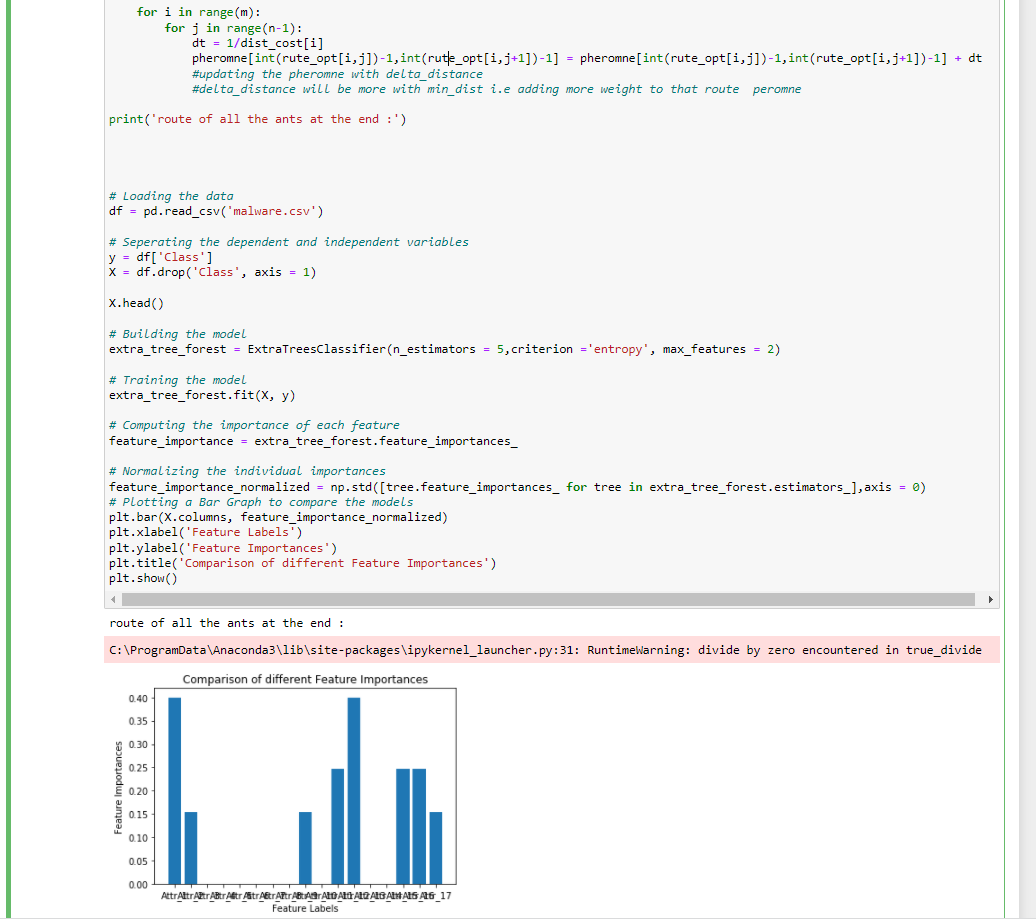
plt.xlabel('Feature Labels')

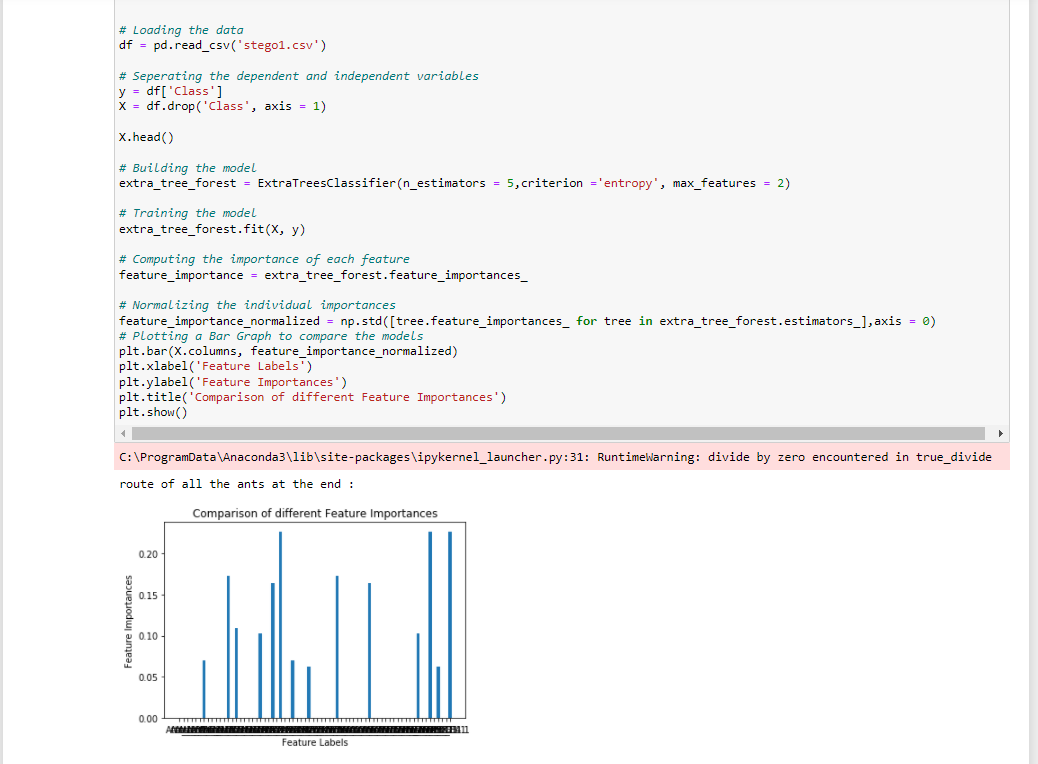
plt.ylabel('Feature Importances')

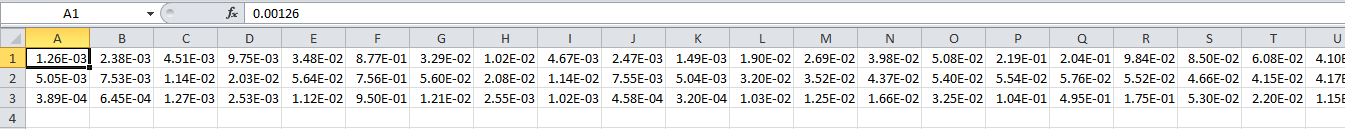
plt.title('Comparison of different Feature Importances')

plt.show()

**Output snaps:**

****

****



**Decision Tree:**

**Code:**

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.tree import DecisionTreeRegressor

#loading my train dataset into python

 train = pd.read\_csv('stegotraingod.csv')

test = pd.read\_csv('stego1.csv')

 #factors that will predict the price

desired\_factors = ['Attr\_1','Attr\_2','Attr\_3','Attr\_4','Attr\_5','Attr\_6','Attr\_7','Attr\_8','Attr\_9','Attr\_10','Attr\_11','Attr\_12','Attr\_13','Attr\_14','Attr\_15','Attr\_16','Attr\_17','Attr\_18','Attr\_19','Attr\_20','Attr\_21','Attr\_22','Attr\_23','Attr\_24','Attr\_25','Attr\_26','Attr\_27','Attr\_28','Attr\_29','Attr\_30','Attr\_31','Attr\_32','Attr\_33','Attr\_34','Attr\_35','Attr\_36','Attr\_37','Attr\_38','Attr\_39','Attr\_40','Attr\_41','Attr\_42','Attr\_43','Attr\_44','Attr\_45','Attr\_46','Attr\_47','Attr\_48','Attr\_49','Attr\_50','Attr\_51','Attr\_52','Attr\_53','Attr\_54','Attr\_55', 'Attr\_56','Attr\_57','Attr\_58','Attr\_59','Attr\_60','Attr\_61','Attr\_62','Attr\_63','Attr\_64', 'Attr\_65','Attr\_66','Attr67,'Attr\_68','Attr\_69',''Class']

  #set my model to DecisionTree

model = DecisionTreeRegressor()

 #set prediction data to factors that will predict, and set target to SalePrice

train\_data = train[desired\_factors]

test\_data = test[desired\_factors]

target = train.Class

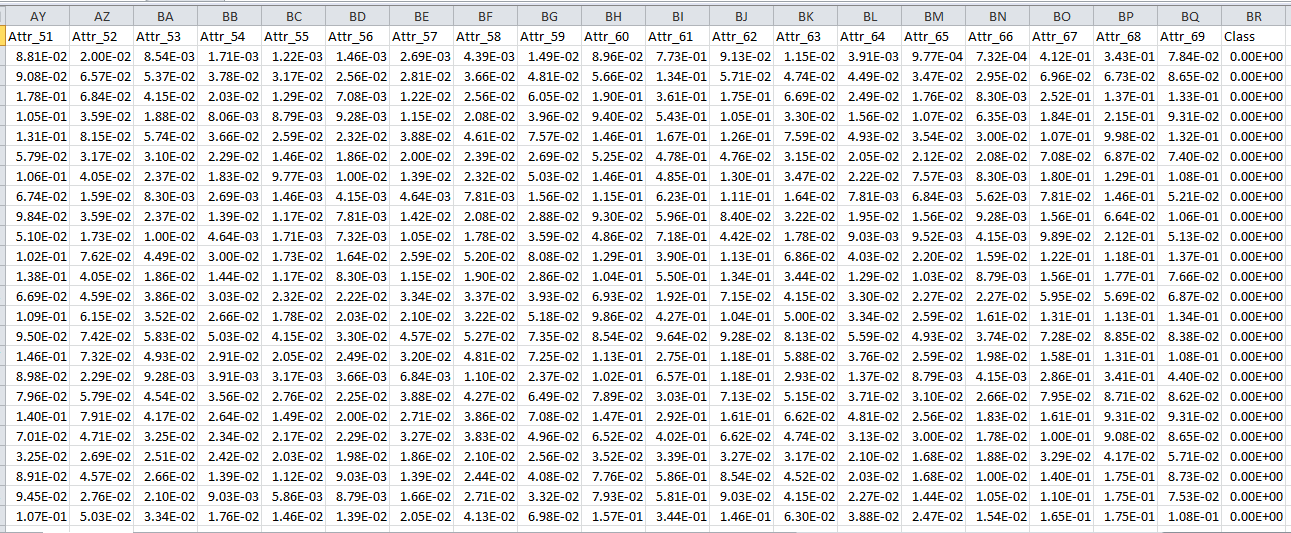
  #fitting model with prediction data and telling it my target

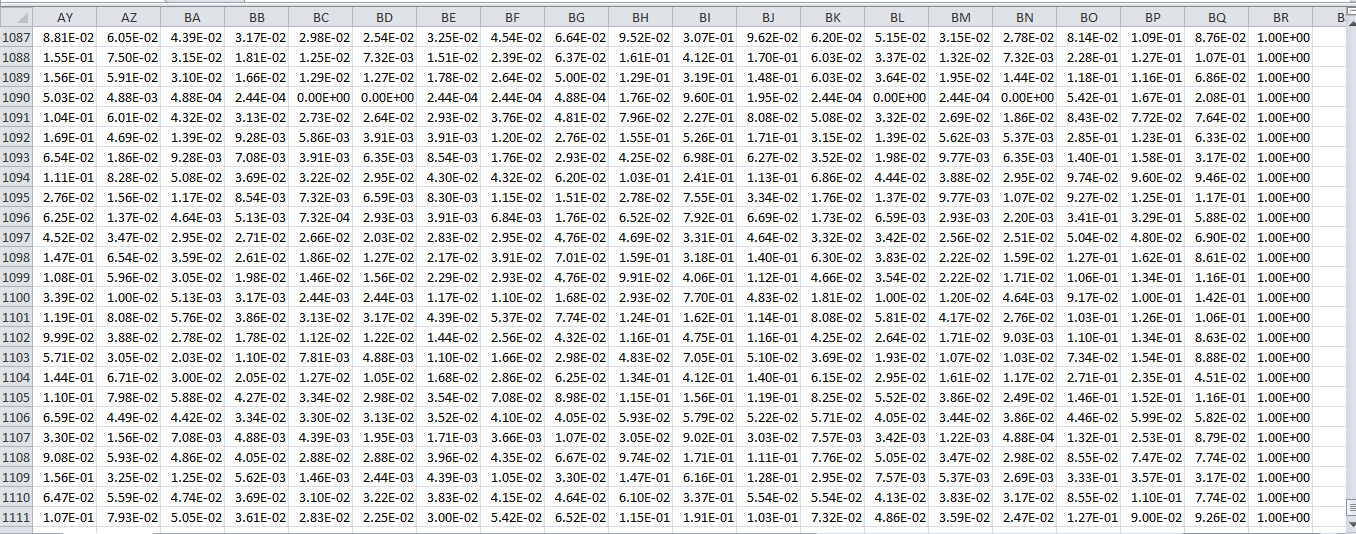
model.fit(train\_data, target)

  model.predict(test\_data.head())

**Output snaps:**

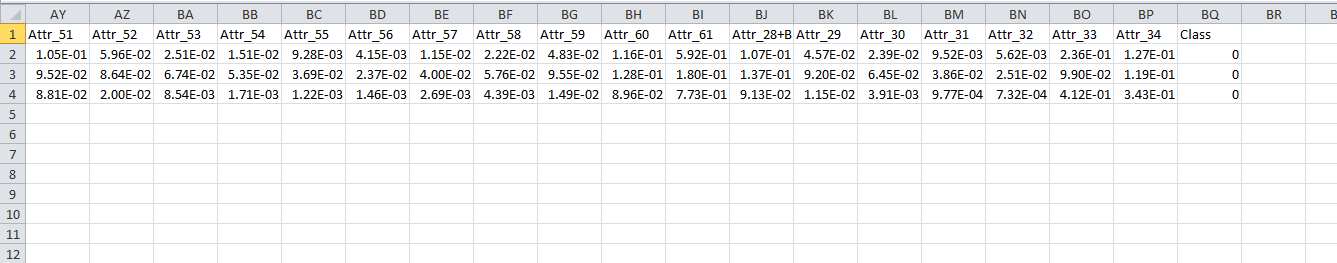
**Training:**





**Testing**

**Input snap**



**Output snap**

****

1. **Payload Extraction**

**Model 1:**

# Python program implementing Image Steganography

# PIL module is used to extract

# pixels of image and modify it

from PIL import Image

# Convert encoding data into 8-bit binary

# form using ASCII value of characters

def genData(data):

# list of binary codes

# of given data

newd = []

for i in data:

newd.append(format(ord(i), '08b'))

return newd

# Pixels are modified according to the

# 8-bit binary data and finally returned

def modPix(pix, data):

datalist = genData(data)

lendata = len(datalist)

imdata = iter(pix)

for i in range(lendata):

# Extracting 3 pixels at a time

pix = [value for value in imdata.\_\_next\_\_()[:3] +

imdata.\_\_next\_\_()[:3] +

imdata.\_\_next\_\_()[:3]]

# Pixel value should be made

# odd for 1 and even for 0

for j in range(0, 8):

if (datalist[i][j]=='0') and (pix[j]% 2 != 0):

if (pix[j]% 2 != 0):

pix[j] -= 1

elif (datalist[i][j] == '1') and (pix[j] % 2 == 0):

pix[j] -= 1

# Eigh^th pixel of every set tells

# whether to stop ot read further.

# 0 means keep reading; 1 means the

# message is over.

if (i == lendata - 1):

if (pix[-1] % 2 == 0):

pix[-1] -= 1

else:

if (pix[-1] % 2 != 0):

pix[-1] -= 1

pix = tuple(pix)

yield pix[0:3]

yield pix[3:6]

yield pix[6:9]

def encode\_enc(newimg, data):

w = newimg.size[0]

(x, y) = (0, 0)

for pixel in modPix(newimg.getdata(), data):

# Putting modified pixels in the new image

newimg.putpixel((x, y), pixel)

if (x == w - 1):

x = 0

y += 1

else:

x += 1

# Encode data into image

def encode():

img = input("Enter image name(with extension): ")

image = Image.open(img, 'r')

print("\n\nIMAGE IS READ SUCESSFULLY !!!!!!!!!!PIXEL VALUE OF GIVEN IMAGE IS DISPLAYED AS FOLLOWSS!!!!!")

print(image.histogram())

data = input("Enter data to be encoded : ")

if (len(data) == 0):

raise ValueError('Data is empty')

newimg = image.copy()

encode\_enc(newimg, data)

new\_img\_name = input("Enter the name of new image(with extension): ")

newimg.save(new\_img\_name, str(new\_img\_name.split(".")[1].upper()))

print("\n\nGIVEN MESSAGE IS SUCCESSFULLY HIDED INSIDE THE IMAGE !!!!!!!!!!!IMAGE PIXELS AFTER ENCRYPTION IS DISPALYED BELOW!!!!!!NOTIFY CHANGES")

print(newimg.histogram())

**# Decode the data in the image**

**def decode():**

**img = input("Enter image name(with extension) :")**

**image = Image.open(img, 'r')**

**data = ''**

**imgdata = iter(image.getdata())**

**while (True):**

**pixels = [value for value in imgdata.\_\_next\_\_()[:3] +**

**imgdata.\_\_next\_\_()[:3] +**

**imgdata.\_\_next\_\_()[:3]]**

**# string of binary data**

**binstr = ''**

**for i in pixels[:8]:**

**if (i % 2 == 0):**

**binstr += '0'**

**else:**

**binstr += '1'**

**data += chr(int(binstr, 2))**

**if (pixels[-1] % 2 != 0):**

**return data**

# Main Function

def main():

a = int(input(":: Welcome to Steganography ::\n" "1. Encode\n 2. Decode\n"))

if (a == 1):

encode()

elif (a == 2):

print("Decoded word- " + decode())

else:

raise Exception("Enter correct input")

# Driver Code

if \_\_name\_\_ == '\_\_main\_\_' :

# Calling main function

main()

**Model 2:**

**## ENCODING FILES**

out = open("C:/Users/MONIKA/Desktop/wrok 1 imp/malimage.png", "wb")

out.write(open("C:/Users/MONIKA/Desktop/wrok 1 imp/in.png", "rb").read())

out.write(open("C:/Users/MONIKA/Desktop/wrok 1 imp/mcpatcher.exe.zip", "rb").read())

out.close()

**###DECODING FILES**

## KALI LINUX

## i - input

## o - output

cd Desktop

foremost -t rar -i imagename.extention -o foldername for output

1. **Payload Classification**

**Binary Classifier BC Code:**

**Model 1:**

{

"cells": [

{

"cell\_type": "code",

"execution\_count": 8,

"id": "spectacular-jacob",

"metadata": {},

"outputs": [],

"source": [

"import torch\n",

"from utils import set\_seed\n",

"import os\n",

"import pickle\n",

"from sklearn.model\_selection import train\_test\_split\n",

"from torch.utils.data import DataLoader, Dataset, Subset\n",

"from torch import nn\n",

"from torch.nn import functional as F\n",

"import random\n",

"import matplotlib.pyplot as plt\n",

"import numpy as np\n",

"import pandas as pd\n",

"import seaborn as sns\n",

"from sklearn.metrics import auc, confusion\_matrix, roc\_curve\n",

"from torch import optim\n",

"from torch.nn.functional import sigmoid\n",

"from tqdm.auto import tqdm\n",

"from sklearn import metrics\n",

"\n",

"plt.rcParams.update(plt.rcParamsDefault)\n",

"set\_seed(42)\n",

"device = torch.device(\"cpu\")\n",

"\n",

"import warnings\n",

"warnings.filterwarnings('ignore')"

]

},

{

"cell\_type": "markdown",

"id": "premier-snowboard",

"metadata": {},

"source": [

"## Dataset"

]

},

{

"cell\_type": "code",

"execution\_count": 9,

"id": "mental-failure",

"metadata": {},

"outputs": [],

"source": [

"class MalwareDataset(Dataset):\n",

" def \_\_init\_\_(self, benign\_dir=\"data/benign\", malware\_dir=\"data/malware\"):\n",

" self.benign\_dir = benign\_dir\n",

" self.malware\_dir = malware\_dir\n",

" self.benign\_files = sorted(os.listdir(benign\_dir))\n",

" self.malware\_files = sorted(os.listdir(malware\_dir))\n",

"\n",

" def \_\_getitem\_\_(self, index):\n",

" try:\n",

" file\_dir = os.path.join(self.benign\_dir, self.benign\_files[index])\n",

" label = 0.0\n",

" except IndexError:\n",

" file\_dir = os.path.join(\n",

" self.malware\_dir, self.malware\_files[index - len(self.benign\_files)],\n",

" )\n",

" label = 1.0\n",

"\n",

" with open(file\_dir, \"rb\") as f:\n",

" file\_ = torch.tensor(pickle.load(f))\n",

" return file\_, label\n",

"\n",

" def \_\_len\_\_(self):\n",

" return len(self.benign\_files) + len(self.malware\_files)\n",

"\n",

"class UniLabelDataset(Dataset):\n",

" def \_\_init\_\_(self, data\_dir, is\_malware):\n",

" self.data\_dir = data\_dir\n",

" self.is\_malware = is\_malware\n",

" self.files = sorted(os.listdir(data\_dir))\n",

"\n",

" def \_\_getitem\_\_(self, index):\n",

" file\_dir = os.path.join(self.data\_dir, self.files[index])\n",

" with open(file\_dir, \"rb\") as f:\n",

" file\_ = torch.tensor(pickle.load(f))\n",

" return file\_, float(self.is\_malware)\n",

"\n",

" def \_\_len\_\_(self):\n",

" return len(self.files)\n",

"\n",

"\n",

"def collate\_fn(batch):\n",

" xs = pad\_sequence([x[0] for x in batch], max\_len=4096, padding\_value=256)\n",

" ys = torch.tensor([x[1] for x in batch])\n",

" return xs, ys\n",

"\n",

"\n",

"def pad\_sequence(sequences, max\_len=None, padding\_value=0):\n",

" batch\_size = len(sequences)\n",

" if max\_len is None:\n",

" max\_len = max([s.size(0) for s in sequences])\n",

" out\_tensor = sequences[0].new\_full((batch\_size, max\_len), padding\_value)\n",

" for i, tensor in enumerate(sequences):\n",

" length = tensor.size(0)\n",

" if max\_len > length:\n",

" out\_tensor[i, :length] = tensor\n",

" else:\n",

" out\_tensor[i, :max\_len] = tensor[:max\_len]\n",

" return out\_tensor\n",

"\n",

"\n",

"def train\_val\_test\_split(idx, val\_size, test\_size):\n",

" tv\_idx, test\_idx = train\_test\_split(idx, test\_size=test\_size, shuffle=True)\n",

" train\_idx, val\_idx = train\_test\_split(tv\_idx, test\_size=val\_size, shuffle=True)\n",

" return train\_idx, val\_idx, test\_idx\n",

"\n",

"\n",

"def make\_idx(dataset, val\_size, test\_size):\n",

" num\_benign = len(dataset.benign\_files)\n",

" num\_malware = len(dataset.malware\_files)\n",

" benign\_idx = range(num\_benign)\n",

" malware\_idx = range(num\_benign, num\_benign + num\_malware)\n",

" benign\_train\_idx, benign\_val\_idx, benign\_test\_idx = train\_val\_test\_split(\n",

" benign\_idx, val\_size, test\_size\n",

" )\n",

" malware\_train\_idx, malware\_val\_idx, malware\_test\_idx = train\_val\_test\_split(\n",

" malware\_idx, val\_size, test\_size\n",

" )\n",

" train\_idx = benign\_train\_idx + malware\_train\_idx\n",

" val\_idx = benign\_val\_idx + malware\_val\_idx\n",

" test\_idx = benign\_test\_idx + malware\_test\_idx\n",

" return train\_idx, val\_idx, test\_idx\n",

"\n",

"\n",

"def make\_loaders(batch\_size, val\_size, test\_size):\n",

" dataset = MalwareDataset()\n",

" train\_idx, val\_idx, test\_idx = make\_idx(dataset, val\_size, test\_size)\n",

" train\_dataset = Subset(dataset, indices=train\_idx)\n",

" val\_dataset = Subset(dataset, indices=val\_idx)\n",

" test\_dataset = Subset(dataset, indices=test\_idx)\n",

" train\_loader = make\_loader(train\_dataset, batch\_size)\n",

" val\_loader = make\_loader(val\_dataset, batch\_size)\n",

" test\_loader = make\_loader(test\_dataset, batch\_size)\n",

" return train\_loader, val\_loader, test\_loader\n",

"\n",

"\n",

"def make\_loader(dataset, batch\_size):\n",

" return DataLoader(\n",

" dataset, batch\_size=batch\_size, collate\_fn=collate\_fn, shuffle=True\n",

" )"

]

},

{

"cell\_type": "markdown",

"id": "german-holiday",

"metadata": {},

"source": [

"## Models"

]

},

{

"cell\_type": "code",

"execution\_count": 10,

"id": "secure-scout",

"metadata": {},

"outputs": [],

"source": [

"class MalConv(nn.Module):\n",

" def \_\_init\_\_(self, embed\_dim, max\_len, out\_channels, window\_size, dropout=0.5):\n",

" super(MalConv, self).\_\_init\_\_()\n",

" self.embed = nn.Embedding(257, embed\_dim)\n",

" self.dropout = nn.Dropout(dropout)\n",

" self.conv = nn.Conv1d(\n",

" in\_channels=embed\_dim,\n",

" out\_channels=out\_channels \* 2,\n",

" kernel\_size=window\_size,\n",

" stride=window\_size,\n",

" )\n",

" self.fc = nn.Linear(out\_channels, 1)\n",

"\n",

" def forward(self, x):\n",

" embedding = self.dropout(self.embed(x))\n",

" conv\_in = embedding.permute(0, 2, 1)\n",

" conv\_out = self.conv(conv\_in)\n",

" glu\_out = F.glu(conv\_out, dim=1)\n",

" values, \_ = glu\_out.max(dim=-1)\n",

" output = self.fc(values).squeeze(1)\n",

" return output\n",

" \n",

"class Conv\_RNN\_Custom(nn.Module):\n",

" def \_\_init\_\_(\n",

" self,\n",

" embed\_dim,\n",

" out\_channels,\n",

" window\_size,\n",

" module,\n",

" hidden\_size,\n",

" num\_layers,\n",

" bidirectional,\n",

" residual,\n",

" dropout=0.5,\n",

" ):\n",

" super(Conv\_RNN\_Custom, self).\_\_init\_\_()\n",

" assert module.\_\_name\_\_ in {\n",

" \"RNN\",\n",

" \"GRU\",\n",

" \"LSTM\",\n",

" }, \"`module` must be a `torch.nn` recurrent layer\"\n",

" self.residual = residual\n",

" self.embed = nn.Embedding(257, embed\_dim)\n",

" self.conv = nn.Conv1d(\n",

" in\_channels=embed\_dim,\n",

" out\_channels=out\_channels,\n",

" kernel\_size=window\_size,\n",

" stride=window\_size,\n",

" )\n",

" self.rnn = module(\n",

" input\_size=out\_channels,\n",

" hidden\_size=hidden\_size,\n",

" num\_layers=num\_layers,\n",

" bidirectional=bidirectional,\n",

" )\n",

" self.dropout = nn.Dropout(dropout)\n",

" rnn\_out\_size = (int(bidirectional) + 1) \* hidden\_size\n",

" if residual:\n",

" self.fc = nn.Linear(out\_channels + rnn\_out\_size, 1)\n",

" else:\n",

" self.fc = nn.Linear(rnn\_out\_size, 1)\n",

"\n",

" def forward(self, x):\n",

" embedding = self.dropout(self.embed(x))\n",

" conv\_in = embedding.permute(0, 2, 1)\n",

" conv\_out = self.conv(conv\_in)\n",

" if self.residual:\n",

" values, \_ = conv\_out.max(dim=-1)\n",

" conv\_out = conv\_out.permute(2, 0, 1)\n",

" rnn\_out, \_ = self.rnn(conv\_out)\n",

" fc\_in = rnn\_out[-1]\n",

" if self.residual:\n",

" fc\_in = torch.cat((fc\_in, values), dim=-1)\n",

" output = self.fc(fc\_in).squeeze(1)\n",

" return output"

]

},

{

"cell\_type": "markdown",

"id": "straight-theorem",

"metadata": {},

"source": [

"## Utility Functions"

]

},

{

"cell\_type": "code",

"execution\_count": 11,

"id": "comparative-funds",

"metadata": {},

"outputs": [],

"source": [

"def set\_seed(seed):\n",

" random.seed(seed)\n",

" os.environ[\"PYTHONHASHSEED\"] = str(seed)\n",

" np.random.seed(seed)\n",

" torch.manual\_seed(seed)\n",

" torch.backends.cudnn.deterministic = True\n",

" torch.backends.cudnn.benchmark = True\n",

"\n",

" \n",

"def count\_params(model, trainable\_only=True):\n",

" if trainable\_only:\n",

" return sum(p.numel() for p in model.parameters() if p.requires\_grad)\n",

" return sum(p.numel() for p in model.parameters())\n",

"\n",

"\n",

"def plot\_confusion\_matrix(model, test\_loader, save\_title, device, normalize=\"all\"):\n",

" y\_true, y\_pred = predict(model, test\_loader, device)\n",

" conf\_mat = confusion\_matrix(y\_true, y\_pred, normalize=normalize)\n",

" axis\_labels = (\"Benign\", \"Malware\")\n",

" df = pd.DataFrame(conf\_mat, index=axis\_labels, columns=axis\_labels)\n",

" plot = sns.heatmap(df, annot=True, cmap=\"Blues\")\n",

" plot.figure.savefig(os.path.join(\"imgs\", f\"{save\_title}\_conf\_mat.png\"), dpi=300)\n",

" plt.close(plot.figure)\n",

"\n",

"\n",

"def plot\_roc\_curve(models, test\_loader, save\_title, device):\n",

" fig, ax = plt.subplots()\n",

" ax.grid(linestyle=\"--\")\n",

" ax.set\_xlabel(\"False Positive Rate\")\n",

" ax.set\_ylabel(\"True Positive Rate\")\n",

" if isinstance(models, dict):\n",

" for label, model in models.items():\n",

" fpr, tpr, auc\_score = \_rates\_auc(model, test\_loader, device)\n",

" ax.plot(fpr, tpr, label=f\"{label} ({auc\_score:.2f})\")\n",

" else:\n",

" fpr, tpr, auc\_score = \_rates\_auc(models, test\_loader, device)\n",

" ax.plot(fpr, tpr, label=f\"{save\_title} ({auc\_score:.2f})\")\n",

" ax.plot([0, 1], [0, 1], linestyle=\"--\", label=\"Chance (0.5)\")\n",

" ax.legend(loc=\"best\")\n",

" fig.savefig(os.path.join(\"imgs\", f\"{save\_title}\_roc.png\"), dpi=300)\n",

" plt.close(fig)\n",

"\n",

"\n",

"def \_rates\_auc(model, test\_loader, device):\n",

" y\_true, y\_pred = predict(model, test\_loader, device, apply\_sigmoid=True)\n",

" fpr, tpr, \_ = metrics.roc\_curve(y\_true, y\_pred)\n",

" auc\_score = auc(fpr, tpr)\n",

" return fpr, tpr, auc\_score\n",

"\n",

"\n",

"@torch.no\_grad()\n",

"def predict(model, data\_loader, device, apply\_sigmoid=False, to\_numpy=True):\n",

" model.eval()\n",

" y\_true = []\n",

" y\_pred = []\n",

" for inputs, labels in tqdm(data\_loader, leave=False):\n",

" inputs = inputs.to(device)\n",

" outputs = model(inputs)\n",

" y\_true.append(labels)\n",

" y\_pred.append(outputs)\n",

" y\_true = torch.cat(y\_true).to(int)\n",

" if apply\_sigmoid:\n",

" y\_pred = sigmoid(torch.cat(y\_pred))\n",

" else:\n",

" y\_pred = (torch.cat(y\_pred) > 0).to(int)\n",

" if to\_numpy:\n",

" y\_true = y\_true.cpu().numpy()\n",

" y\_pred = y\_pred.cpu().numpy()\n",

" assert y\_true.shape == y\_pred.shape\n",

" model.train()\n",

" return y\_true, y\_pred\n",

"\n",

"\n",

"def get\_accuracy(model, data\_loader, device):\n",

" y\_true, y\_pred = predict(model, data\_loader, device, to\_numpy=False)\n",

" return 100 \* (y\_true == y\_pred).to(float).mean().item()\n",

"\n",

"\n",

"def plot\_train\_history(train\_loss\_history, val\_loss\_history, save\_title):\n",

" fig, ax = plt.subplots()\n",

" time\_ = range(len(train\_loss\_history))\n",

" ax.set\_xlabel(\"Epochs\")\n",

" ax.set\_ylabel(\"BCE Loss\")\n",

" ax.grid(linestyle=\"--\")\n",

" ax.plot(time\_, train\_loss\_history, color=\"blue\", label=\"train loss\")\n",

" ax.plot(time\_, val\_loss\_history, color=\"red\", label=\"val loss\")\n",

" ax.legend(loc=\"best\")\n",

" fig.savefig(os.path.join(\"figures\", f\"{save\_title}\_train\_history.png\"), dpi=300)\n",

" plt.close(fig)\n",

"\n",

"\n",

"def train(\n",

" model,\n",

" train\_loader,\n",

" val\_loader,\n",

" device,\n",

" save\_title,\n",

" lr=0.001,\n",

" patience=3,\n",

" num\_epochs=5,\n",

" verbose=True,\n",

"):\n",

" train\_loss\_history = []\n",

" val\_loss\_history = []\n",

" criterion = torch.nn.BCEWithLogitsLoss()\n",

" optimizer = optim.Adam(model.parameters(), lr=lr)\n",

" monitor = EarlyStopMonitor(patience)\n",

" scheduler = optim.lr\_scheduler.ReduceLROnPlateau(\n",

" optimizer, factor=0.5, patience=patience\n",

" )\n",

" for epoch in range(1, num\_epochs + 1):\n",

" model.train()\n",

" train\_loss = run\_epoch(model, train\_loader, device, criterion, optimizer)\n",

" train\_loss\_history.append(train\_loss)\n",

" model.eval()\n",

" with torch.no\_grad():\n",

" val\_loss = run\_epoch(model, val\_loader, device, criterion)\n",

" val\_loss\_history.append(val\_loss)\n",

" if verbose:\n",

" tqdm.write(\n",

" f\"Epoch [{epoch}/{num\_epochs}], \"\n",

" f\"Train Loss: {train\_loss:.4f}, \"\n",

" f\"Val Loss: {val\_loss:.4f}\"\n",

" )\n",

" scheduler.step(val\_loss)\n",

" if monitor.step(val\_loss):\n",

" break\n",

" if len(val\_loss\_history) == 1 or val\_loss < val\_loss\_history[-2]:\n",

" torch.save(\n",

" model.state\_dict(), os.path.join(\"checkpoints\", f\"{save\_title}.pt\"),\n",

" )\n",

" plot\_train\_history(train\_loss\_history, val\_loss\_history, save\_title)\n",

"\n",

"\n",

"def run\_epoch(model, data\_loader, device, criterion, optimizer=None):\n",

" total\_loss = 0\n",

" for inputs, labels in tqdm(data\_loader, leave=False):\n",

" inputs = inputs.to(device)\n",

" labels = labels.to(device)\n",

" outputs = model(inputs)\n",

" loss = criterion(outputs, labels)\n",

" if optimizer:\n",

" optimizer.zero\_grad()\n",

" loss.backward()\n",

" optimizer.step()\n",

" total\_loss += loss.item()\n",

" return total\_loss / len(data\_loader)\n",

"\n",

"\n",

"class EarlyStopMonitor:\n",

" def \_\_init\_\_(self, patience, mode=\"min\"):\n",

" assert mode in {\"min\", \"max\"}, \"`mode` must be one of 'min' or 'max'\"\n",

" self.log = []\n",

" self.mode = mode\n",

" self.count = 0\n",

" self.patience = patience\n",

"\n",

" def step(self, metric):\n",

" if not self.log:\n",

" self.log.append(metric)\n",

" return False\n",

" flag = metric > self.log[-1]\n",

" if flag == (self.mode == \"min\"):\n",

" self.count += 1\n",

" else:\n",

" self.count = 0\n",

" self.log.append(metric)\n",

" return self.count > self.patience"

]

},

{

"cell\_type": "markdown",

"id": "novel-couple",

"metadata": {},

"source": [

"## Training"

]

},

{

"cell\_type": "code",

"execution\_count": 12,

"id": "theoretical-marine",

"metadata": {},

"outputs": [],

"source": [

"batch\_size = 64\n",

"test\_size = val\_size = 0.2\n",

"\n",

"train\_loader, val\_loader, test\_loader = make\_loaders(batch\_size, val\_size, test\_size)"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "handmade-ancient",

"metadata": {},

"outputs": [],

"source": [

"malconv = MalConv(8, 4096, 128, 32).to(device)\n",

"train(malconv, train\_loader, val\_loader, device, \"malconv\")\n",

"torch.save(malconv.state\_dict(), os.path.join(\"weights\", \"malconv.pt\"))"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "conscious-sample",

"metadata": {},

"outputs": [],

"source": [

"gru\_bi = Conv\_RNN\_Custom(8, 128, 32, torch.nn.GRU, 256, 1, True, False).to(device)\n",

"train(gru\_bi, train\_loader, val\_loader, device, \"gru\_bi\")\n",

"torch.save(gru\_bi.state\_dict(), os.path.join(\"weights\", \"gru\_bi.pt\"))"

]

},

{

"cell\_type": "markdown",

"id": "trained-killing",

"metadata": {},

"source": [

"## Testing"

]

},

{

"cell\_type": "code",

"execution\_count": 6,

"id": "solid-midwest",

"metadata": {},

"outputs": [

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "91517400e7c54f8fab2debdcc41f9162",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/114 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "4729be2165494da681a478cb74f6dcec",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/114 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "66f5281124d54e778d7f557bfb5b8167",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/365 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"name": "stdout",

"output\_type": "stream",

"text": [

"Training accuracy : 98.00051488886982\n"

]

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "79d0a7e41a5e4da4ac7fd19791861975",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/92 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"name": "stdout",

"output\_type": "stream",

"text": [

"Validation accuracy : 97.99210571477605\n"

]

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "8a2f189574504be0b3d01db08f41b3ae",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/114 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"name": "stdout",

"output\_type": "stream",

"text": [

"Testing accuracy : 97.8448867536033\n"

]

}

],

"source": [

"malconv = MalConv(8, 4096, 128, 32).to(device)\n",

"PATH = os.path.join(\"weights\", \"malconv.pt\")\n",

"malconv.load\_state\_dict(torch.load(PATH, map\_location=torch.device('cpu')))\n",

"malconv.eval()\n",

"plot\_confusion\_matrix(malconv, test\_loader, \"malconv\", device)\n",

"plot\_roc\_curve(malconv, test\_loader, \"malconv\", device)\n",

"print(f\"Training accuracy : {get\_accuracy(malconv, train\_loader, device)}\")\n",

"print(f\"Validation accuracy : {get\_accuracy(malconv, val\_loader, device)}\")\n",

"print(f\"Testing accuracy : {get\_accuracy(malconv, test\_loader, device)}\")"

]

},

{

"cell\_type": "code",

"execution\_count": 7,

"id": "raised-identity",

"metadata": {},

"outputs": [

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "ccd662d23dad4db484e9868265cde127",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/114 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "f8a097c5b40b4f4aa3e28a968ea62beb",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/114 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "8ed7a70623024234b474a63b036f0095",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/365 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"name": "stdout",

"output\_type": "stream",

"text": [

"Training accuracy : 98.30515747017935\n"

]

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "001029819972451da8a40705601817ca",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/92 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"name": "stdout",

"output\_type": "stream",

"text": [

"Validation accuracy : 98.16372061094903\n"

]

},

{

"data": {

"application/vnd.jupyter.widget-view+json": {

"model\_id": "e5577cdebcbe4cccad958fab92b3d712",

"version\_major": 2,

"version\_minor": 0

},

"text/plain": [

" 0%| | 0/114 [00:00<?, ?it/s]"

]

},

"metadata": {},

"output\_type": "display\_data"

},

{

"name": "stdout",

"output\_type": "stream",

"text": [

"Testing accuracy : 98.38023335621139\n"

]

}

],

"source": [

"gru\_bi = Conv\_RNN\_Custom(8, 128, 32, torch.nn.GRU, 256, 1, True, False).to(device)\n",

"PATH = os.path.join(\"weights\", \"gru\_bi.pt\")\n",

"gru\_bi.load\_state\_dict(torch.load(PATH, map\_location=torch.device('cpu')))\n",

"gru\_bi.eval()\n",

"plot\_confusion\_matrix(gru\_bi, test\_loader, \"gru\_bi\", device)\n",

"plot\_roc\_curve(gru\_bi, test\_loader, \"gru\_bi\", device)\n",

"print(f\"Training accuracy : {get\_accuracy(gru\_bi, train\_loader, device)}\")\n",

"print(f\"Validation accuracy : {get\_accuracy(gru\_bi, val\_loader, device)}\")\n",

"print(f\"Testing accuracy : {get\_accuracy(gru\_bi, test\_loader, device)}\")"

]

}

],

"metadata": {

"kernelspec": {

"display\_name": "Python 3",

"language": "python",

"name": "python3"

},

"language\_info": {

"codemirror\_mode": {

"name": "ipython",

"version": 3

},

"file\_extension": ".py",

"mimetype": "text/x-python",

"name": "python",

"nbconvert\_exporter": "python",

"pygments\_lexer": "ipython3",

"version": "3.8.7"

}

},

"nbformat": 4,

"nbformat\_minor": 5

}

**Model 2:**

# first neural network with keras make predictions

from numpy import loadtxt

from keras.models import Sequential

from keras.layers import Dense

# load the dataset

dataset = loadtxt('malware1.csv', delimiter=',')

# split into input (X) and output (y) variables

X = dataset[:,4:8]

y = dataset[:,4]

# define the keras model

model = Sequential()

model.add(Dense(4, input\_dim=4, activation='relu'))

#model.add(Dense(8, activation='relu'))

model.add(Dense(1, activation='relu'))

# compile the keras model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# fit the keras model on the dataset

model.fit(X, y, epochs=150, batch\_size=10, verbose=0)

# make class predictions with the model

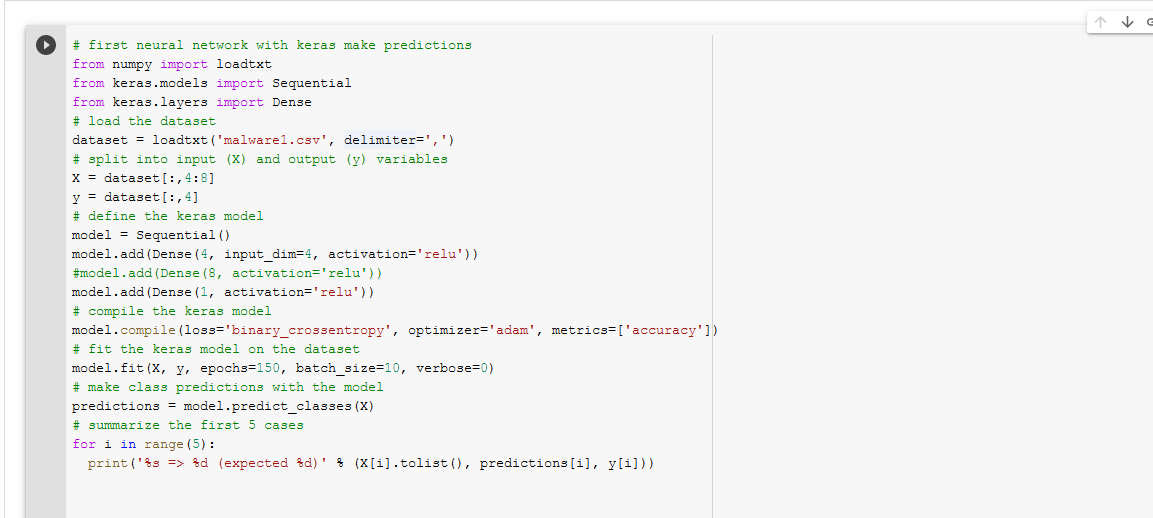
predictions = model.predict\_classes(X)

# summarize the first 5 cases

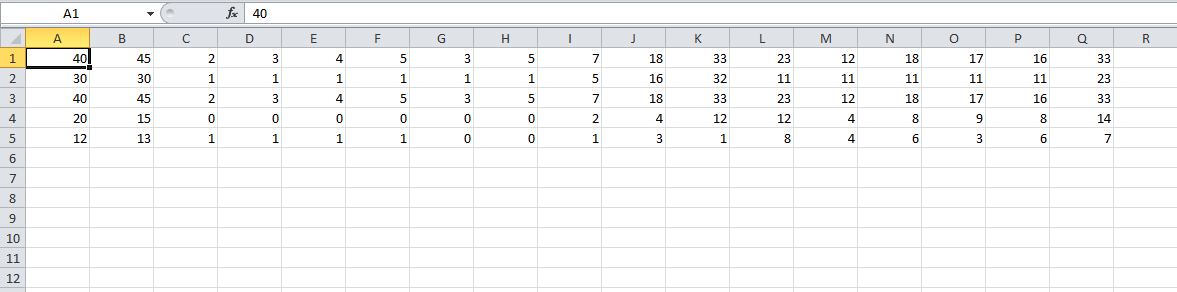
for i in range(5):

  print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))

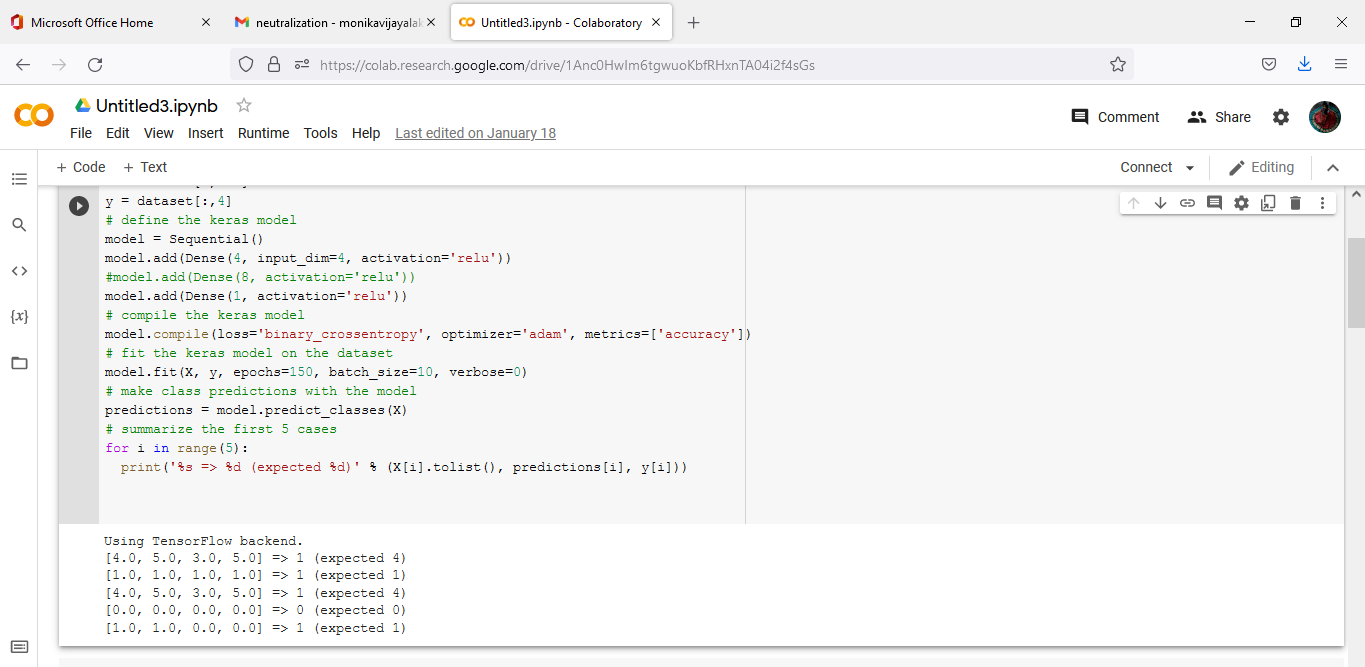
**Code snap**



**Input sample snap:**



**Output Sample**



**FCM Code:**

import pandas as pd

import numpy as np

import random

import operator

import math

df\_full = pd.read\_csv("malware.csv")

columns = list(df\_full.columns)

features = columns[:len(columns)-1]

class\_labels = list(df\_full[columns[-1]])

df = df\_full[features]

# Number of Attributes

num\_attr = len(df.columns) - 1

# Number of Clusters

k = 2

# Maximum number of iterations

MAX\_ITER = 100

# Number of data points

n = len(df)

# Fuzzy parameter

m = 2.00

def accuracy(cluster\_labels, class\_labels):

county = [0,0]

countn = [0,0]

tp = [0, 0]

tn = [0, 0]

fp = [0, 0]

fn = [0, 0]

#INITIALIZE FUZZY PARTITION OR FUZZY MEMBERSHIP MATRIC

for i in range(len(df)):

# Yes = 1, No = 0

if cluster\_labels[i] == 1 and class\_labels[i] == 'Yes':

tp[0] = tp[0] + 1

if cluster\_labels[i] == 0 and class\_labels[i] == 'No':

tn[0] = tn[0] + 1

if cluster\_labels[i] == 1 and class\_labels[i] == 'No':

fp[0] = fp[0] + 1

if cluster\_labels[i] == 0 and class\_labels[i] == 'Yes':

fn[0] = fn[0] + 1

for i in range(len(df)):

# Yes = 0, No = 1

if cluster\_labels[i] == 0 and class\_labels[i] == 'Yes':

tp[1] = tp[1] + 1

if cluster\_labels[i] == 1 and class\_labels[i] == 'No':

tn[1] = tn[1] + 1

if cluster\_labels[i] == 0 and class\_labels[i] == 'No':

fp[1] = fp[1] + 1

if cluster\_labels[i] == 1 and class\_labels[i] == 'Yes':

fn[1] = fn[1] + 1

a0 = float((tp[0] + tn[0]))/(tp[0] + tn[0] + fn[0] + fp[0])

a1 = float((tp[1] + tn[1]))/(tp[1] + tn[1] + fn[1] + fp[1])

p0 = float(tp[0])/(tp[0] + fp[0])

p1 = float(tp[1])/(tp[1] + fp[1])

r0 = float(tp[0])/(tp[0] + fn[0])

r1 = float(tp[1])/(tp[1] + fn[1])

accuracy = [a0\*100,a1\*100]

precision = [p0\*100,p1\*100]

recall = [r0\*100,r1\*100]

return accuracy, precision, recall

#STEP1: INITILAIZE MEMBERSHIP MATRIX

def initializeMembershipMatrix():

membership\_mat = list()

for i in range(n):

random\_num\_list = [random.random() for i in range(k)]

summation = sum(random\_num\_list)

temp\_list = [x/summation for x in random\_num\_list]

membership\_mat.append(temp\_list)

return membership\_mat

#STEP2: CALCULATE CLUSTER CENTER or CENTRIOD

def calculateClusterCenter(membership\_mat):

cluster\_mem\_val = list(zip(\*membership\_mat))

cluster\_centers = list()

for j in range(k):

x = list(cluster\_mem\_val[j])

xraised = [e \*\* m for e in x]

denominator = sum(xraised)

temp\_num = list()

for i in range(n):

data\_point = list(df.iloc[i])

prod = [xraised[i] \* val for val in data\_point]

temp\_num.append(prod)

numerator = list(map(sum,list(zip(\*temp\_num))))

center = [z/denominator for z in numerator]

cluster\_centers.append(center)

return cluster\_centers

#STEP3:UPDATE MEMBERSHIP MEMBERSHIP MATRIX

def updateMembershipValue(membership\_mat, cluster\_centers):

p = float(2/(m-1))

for i in range(n):

x = list(df.iloc[i])

distances = [np.linalg.norm(list(map(operator.sub, x, cluster\_centers[j]))) for j in range(k)]

for j in range(k):

den = sum([math.pow(float(distances[j]/distances[c]), p) for c in range(k)])

membership\_mat[i][j] = float(1/den)

return membership\_mat

def getClusters(membership\_mat):

cluster\_labels = list()

for i in range(n):

max\_val, idx = max((val, idx) for (idx, val) in enumerate(membership\_mat[i]))

cluster\_labels.append(idx)

return cluster\_labels

#STEP4: OBTAIN FUZZY MEMBERSHIP VALUE FOR EACH FUNCTION

def fuzzyCMeansClustering():

# Membership Matrix

membership\_mat = initializeMembershipMatrix()

curr = 0

while curr <= MAX\_ITER:

cluster\_centers = calculateClusterCenter(membership\_mat)

membership\_mat = updateMembershipValue(membership\_mat, cluster\_centers)

cluster\_labels = getClusters(membership\_mat)

curr += 1

print(membership\_mat)

return cluster\_labels, cluster\_centers

labels, centers = fuzzyCMeansClustering()

a,p,r = accuracy(labels, class\_labels)

**Code snap:**

