In [ ]:

```
# Importing the Library
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error, log_loss
from tqdm import tqdm_notebook
import seaborn as sns
import imageio
import time
from IPython.display import HTML

from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.datasets import make_blobs
```

In [ ]:

```
class FFSN_MultiClass: #Class for Feed-Forward Noural Network

  def __init__(self, n_inputs, n_outputs, hidden_sizes=[3]):
    self.nx = n_inputs
    self.ny = n_outputs
    self.nh = len(hidden_sizes)
    self.sizes = [self.nx] + hidden_sizes + [self.ny]

    self.W = {}
    self.B = {}
    for i in range(self.nh+1):
      self.W[i+1] = np.random.randn(self.sizes[i], self.sizes[i+1])
      self.B[i+1] = np.random.randn(1, self.sizes[i+1])

  def sigmoid(self, x): # Sigmoid Function
    return 1.0/(1.0 + np.exp(-x))

  def softmax(self, x): # Softmax Function for O/P
    exps = np.exp(x)
    return exps / np.sum(exps)

  def forward_pass(self, x): # Forward Pass
    self.A = {}
    self.H = {}
    self.H[0] = x.reshape(1, -1)
    for i in range(self.nh):
      self.A[i+1] = np.matmul(self.H[i], self.W[i+1]) + self.B[i+1]
      self.H[i+1] = self.sigmoid(self.A[i+1])
    self.A[self.nh+1] = np.matmul(self.H[self.nh], self.W[self.nh+1]) + self.
    self.H[self.nh+1] = self.softmax(self.A[self.nh+1])
    return self.H[self.nh+1]

  def predict(self, X): # Predict Function (Forward Pass)
    Y_pred = []
    for x in X:
      y_pred = self.forward_pass(x)
      Y_pred.append(y_pred)
    return np.array(Y_pred).squeeze()

  def grad_sigmoid(self, x): #Gradient of Sigmoid Function
    return x*(1-x)

  def cross_entropy(self,label,pred): # Entropy Loss Calculations
    yl=np.multiply(pred,label)
    yl=yl[yl!=0]
    yl=-np.log(yl)
    yl=np.mean(yl)
    return yl

  def grad(self, x, y): # Gradient Calculation
    self.forward_pass(x)
    self.dW = {}
    self.dB = {}
    self.dH = {}
    self.dA = {}
    L = self.nh + 1
```

```
    self.dA[L] = (self.H[L] - y)
    for k in range(L, 0, -1):
      self.dW[k] = np.matmul(self.H[k-1].T, self.dA[k])
      self.dB[k] = self.dA[k]
      self.dH[k-1] = np.matmul(self.dA[k], self.W[k].T)
      self.dA[k-1] = np.multiply(self.dH[k-1], self.grad_sigmoid(self.H[k-1]

  def fit(self, X, Y, epochs=100, initialize='True', learning_rate=0.01, disp

    if display_loss:
      loss = {}

    if initialize:
      for i in range(self.nh+1):
        self.W[i+1] = np.random.randn(self.sizes[i], self.sizes[i+1])
        self.B[i+1] = np.zeros((1, self.sizes[i+1]))

    for epoch in tqdm_notebook(range(epochs), total=epochs, unit="epoch"):
      dW = {}
      dB = {}
      for i in range(self.nh+1):
        dW[i+1] = np.zeros((self.sizes[i], self.sizes[i+1]))
        dB[i+1] = np.zeros((1, self.sizes[i+1]))
      for x, y in zip(X, Y):
        self.grad(x, y)
        for i in range(self.nh+1):
          dW[i+1] += self.dW[i+1]
          dB[i+1] += self.dB[i+1]

      m = X.shape[1]
      for i in range(self.nh+1):
        self.W[i+1] -= learning_rate * (dW[i+1]/m)
        self.B[i+1] -= learning_rate * (dB[i+1]/m)

      if display_loss:
        Y_pred = self.predict(X)
        loss[epoch] = self.cross_entropy(Y, Y_pred)

    if display_loss:
      plt.plot(loss.values())
      plt.xlabel('Epochs')
      plt.ylabel('CE')
      plt.show()
```

In [ ]:

```
# Importing the Dataset
data = pd.read_csv('Final_Refined_Encoded.csv')
```

In [ ]:

```
#data.head(20)
```

In [ ]:
```
# Replacing the Price according to FNN
bins = [1, 2, 3, 4]
#bins = [0, 50, 150, 250, 500, 1000]
names = [0, 1, 2, 3]

label_dict = dict(enumerate(names, 1))
price = pd.Series(np.vectorize(label_dict.get)(np.digitize(data['Price'], bir
#price
```

In [ ]:
```
data['Price'] = price
```

In [ ]:
```
# Saperating the Target Column
X, y = data.iloc[:, :-1], data.iloc[:, -1]
```

In [ ]:
```
# Test-Train Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ranc
```

In [ ]:
```
# Standard-Scalar
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [ ]:
```
#y_train.describe()
```

In [ ]:
```
# OneHotEncoding the output categories
enc = OneHotEncoder()
# 1 -> (1, 0, 0, 0, 0, 0), 2 -> (0, 1, 0, 0, 0, 0), 3 -> (0, 0, 1, 0, 0, 0),
y_OH_train = enc.fit_transform(np.expand_dims(y_train,1)).toarray()
y_OH_val = enc.fit_transform(np.expand_dims(y_test,1)).toarray()
print(y_OH_train.shape, y_OH_val.shape)
```

In [ ]:
```
# Innitilization & Training
ffsn_multi = FFSN_MultiClass(102,4,[102,50])
ffsn_multi.fit(X_train,y_OH_train,epochs=10000,learning_rate=.005,display_los
```

In [ ]:
```
# Predicting and Accuracy Calculation
Y_pred_train = ffsn_multi.predict(X_train)
Y_pred_train = np.argmax(Y_pred_train,1)

Y_pred_val = ffsn_multi.predict(X_test)
Y_pred_val = np.argmax(Y_pred_val,1)

accuracy_train = accuracy_score(Y_pred_train, y_train)
accuracy_val = accuracy_score(Y_pred_val, y_test)

print("Training accuracy", round(accuracy_train, 2))
print("Validation accuracy", round(accuracy_val, 2))
```

In [ ]:
```

```