<u>Assignment 2</u>

Smart Internz AI

Reg No : 20BCB0134

Name : Ritika Marlapalli

Email: ritika.marlapalli2020@vitstudent.ac.in

College: VIT Vellore

Campus : Vellore

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
```

```
df = pd.read_csv('drug200.csv')
```

```
# Task 1 : Read the dataset and do data pre-processing
```

```
label_encoder = LabelEncoder()
df['Sex'] = label_encoder.fit_transform(df['Sex'])
df['BP'] = label_encoder.fit_transform(df['BP'])
df['Cholesterol'] = label_encoder.fit_transform(df['Cholesterol'])
df['Drug'] = label_encoder.fit_transform(df['Drug'])
print(df.head())
```

```
   Age  Sex  BP  Cholesterol  Na_to_K  Drug
0   23    0   0            0   25.355     0
1   47    1   1            0   13.093     3
2   47    1   1            0   10.114     3
3   28    0   2            0    7.798     4
4   61    0   1            0   18.043     0
```

```
# Scale numerical variables

scaler = StandardScaler()
df[['Age', 'Na_to_K']] = scaler.fit_transform(df[['Age', 'Na_to_K']])
```

```
# Separate features and labels

x = df[['Age', 'Sex', 'BP', 'Cholesterol', 'Na_to_K']]
y = df['Drug']
```

```
# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 random_state=42)
```

```
print(X_train.shape)
print(y_test.shape)
```

```
(160, 5)
(40,)
```

```
# Task 2 : Build the ANN model with (input layer, min 3 hidden layers & output
 layer)
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
# Define the model architecture

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(5,)))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(5, activation='softmax'))
```

```
x = df.iloc[:,0:5]
y = df.iloc[:,5:]
print(x)
print(y)
```

```
          Age  Sex  BP  Cholesterol   Na_to_K
0   -1.291591    0   0            0  1.286522
1    0.162699    1   1            0 -0.415145
2    0.162699    1   1            0 -0.828558
3   -0.988614    0   2            0 -1.149963
4    1.011034    0   1            0  0.271794
..        ...  ...  ..          ...       ...
195  0.708057    0   1            0 -0.626917
196 -1.715759    1   1            0 -0.565995
197  0.465676    1   2            0 -0.859089
198 -1.291591    1   2            1 -0.286500
199 -0.261469    0   1            1 -0.657170

[200 rows x 5 columns]
     Drug
0       0
1       3
2       3
3       4
4       0


195     3
196     3
197     4
198     4
199     4
```

```python
# Compile the model

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',␣
↪metrics=['accuracy'])

y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
model.fit(X_train, y_train_encoded, epochs=20, batch_size=20,␣
↪validation_data=(X_test, y_test_encoded))
```

```
Epoch 1/20
8/8 [==============================] - 2s 38ms/step - loss: 1.4517 - accuracy:
0.5813 - val_loss: 1.3748 - val_accuracy: 0.4000
Epoch 2/20
8/8 [==============================] - 0s 6ms/step - loss: 1.2047 - accuracy:
0.5375 - val_loss: 1.1855 - val_accuracy: 0.4250
Epoch 3/20
8/8 [==============================] - 0s 9ms/step - loss: 1.0034 - accuracy:
0.6187 - val_loss: 1.0329 - val_accuracy: 0.5750
Epoch 4/20
8/8 [==============================] - 0s 9ms/step - loss: 0.8368 - accuracy:
0.7188 - val_loss: 0.8926 - val_accuracy: 0.6250
Epoch 5/20
8/8 [==============================] - 0s 6ms/step - loss: 0.7157 - accuracy:
0.7188 - val_loss: 0.8098 - val_accuracy: 0.6250
Epoch 6/20
8/8 [==============================] - 0s 8ms/step - loss: 0.6184 - accuracy:
0.7500 - val_loss: 0.7295 - val_accuracy: 0.7250
Epoch 7/20
8/8 [==============================] - 0s 6ms/step - loss: 0.5321 - accuracy:
0.8125 - val_loss: 0.6841 - val_accuracy: 0.7500
Epoch 8/20
8/8 [==============================] - 0s 10ms/step - loss: 0.4566 - accuracy:
0.8687 - val_loss: 0.6015 - val_accuracy: 0.8500
Epoch 9/20
8/8 [==============================] - 0s 7ms/step - loss: 0.3843 - accuracy:
0.9062 - val_loss: 0.5173 - val_accuracy: 0.8750
Epoch 10/20
8/8 [==============================] - 0s 8ms/step - loss: 0.3252 - accuracy:
```

```
0.9125 - val_loss: 0.4404 - val_accuracy: 0.8750
Epoch 11/20
8/8 [==============================] - 0s 7ms/step - loss: 0.2629 - accuracy:
0.9125 - val_loss: 0.3672 - val_accuracy: 0.8750
Epoch 12/20
8/8 [==============================] - 0s 7ms/step - loss: 0.2216 - accuracy:
0.9312 - val_loss: 0.3321 - val_accuracy: 0.8750
Epoch 13/20
8/8 [==============================] - 0s 7ms/step - loss: 0.1819 - accuracy:
0.9438 - val_loss: 0.2550 - val_accuracy: 0.9000
Epoch 14/20
8/8 [==============================] - 0s 10ms/step - loss: 0.1560 - accuracy:
0.9500 - val_loss: 0.2532 - val_accuracy: 0.9500
Epoch 15/20
8/8 [==============================] - 0s 6ms/step - loss: 0.1443 - accuracy:
0.9688 - val_loss: 0.1985 - val_accuracy: 0.9000
Epoch 16/20
8/8 [==============================] - 0s 9ms/step - loss: 0.1254 - accuracy:
0.9688 - val_loss: 0.1833 - val_accuracy: 0.9750
Epoch 17/20
8/8 [==============================] - 0s 6ms/step - loss: 0.0970 - accuracy:
0.9875 - val_loss: 0.1717 - val_accuracy: 1.0000
Epoch 18/20
8/8 [==============================] - 0s 9ms/step - loss: 0.0868 - accuracy:
0.9750 - val_loss: 0.1504 - val_accuracy: 0.9750
Epoch 19/20
8/8 [==============================] - 0s 6ms/step - loss: 0.0766 - accuracy:
1.0000 - val_loss: 0.1436 - val_accuracy: 0.9750
Epoch 20/20
8/8 [==============================] - 0s 7ms/step - loss: 0.0678 - accuracy:
0.9812 - val_loss: 0.1206 - val_accuracy: 0.9750
```

```
<keras.callbacks.History at 0x7fc722a7be20>
```

```
y_pred = model.predict(x_test)
y_pred
```

```
array([[4.13405127e-04, 1.27605614e-04, 2.03855492e-07, 7.50870770e-03,
        9.91949975e-01],
       [9.94201958e-01, 5.14725503e-03, 2.99533876e-05, 4.84759919e-04,
        1.36094895e-04],
       [2.79626124e-06, 1.99977421e-06, 5.16646413e-11, 6.72629918e-04,
        9.99322474e-01],
       [2.83280946e-03, 3.48852053e-02, 8.92015360e-03, 7.59812355e-01,
        1.93549350e-01],
       [9.99999940e-01, 3.28292191e-19, 1.42062910e-17, 8.46457494e-17,
        5.58904698e-17],
       [9.99691248e-01, 2.56415988e-05, 2.51631485e-04, 2.94335568e-05,
        2.17517095e-06],
       [9.99999940e-01, 3.61117553e-10, 4.05409484e-10, 1.11134280e-09,
        9.09846420e-10],
       [7.46123632e-03, 1.53253040e-05, 2.05253734e-08, 1.85971186e-02,
        9.73926246e-01],
       [4.89533022e-02, 8.14404786e-01, 6.96765035e-02, 5.54476641e-02,
        1.15178749e-02],
       [3.14717290e-05, 3.12856696e-06, 1.03769771e-07, 3.07339523e-03,
        9.96891856e-01],
       [8.33706290e-04, 9.44750011e-01, 4.69562830e-03, 4.91494723e-02,
        5.71190671e-04],
       [5.63477771e-03, 1.65499118e-03, 4.97897986e-07, 2.14239918e-02,
        9.71285701e-01],
       [9.99937952e-01, 3.12065737e-07, 1.05881973e-07, 2.78759489e-05,
        3.36685516e-05],
       [3.92728811e-03, 9.50904250e-01, 2.91301263e-03, 4.14308533e-02,
        8.24655988e-04],
       [2.11916384e-04, 1.94486752e-02, 9.77127016e-01, 3.20940185e-03,
        2.85138822e-06],
       [9.99888854e-01, 2.64876510e-10, 1.12958193e-11, 9.42942393e-07,
        1.01327441e-05],
       [1.60759955e-03, 1.64753329e-02, 9.78582621e-01, 3.29385232e-03,
        4.04419807e-05],
       [1.57631177e-06, 4.22669018e-07, 7.01798897e-10, 9.63229686e-04,
        9.99034703e-01],
       [3.98420263e-04, 1.10615864e-01, 1.25297796e-04, 5.62819958e-01,
        3.26040477e-01],
       [9.99999940e-01, 2.10215739e-14, 7.02131292e-14, 1.55016607e-11,
        5.87058735e-11],
       [8.40014219e-03, 1.10281460e-01, 8.65873754e-01, 1.37768965e-02,
        1.66778930e-03],
       [5.21895364e-02, 9.92505578e-04, 2.03632610e-03, 1.45251110e-01,
        7.99530506e-01],
       [8.76396836e-04, 2.67904103e-02, 9.21104662e-03, 4.60485995e-01,
```

```
        5.02636135e-01],
       [9.99999940e-01, 6.66354848e-15, 7.17282204e-14, 6.50112885e-13,
        1.00215138e-12],
       [9.99999940e-01, 5.00953337e-16, 5.93842814e-15, 3.52168192e-13,
        5.14562525e-12],
       [9.99999940e-01, 2.64196543e-15, 2.55897327e-14, 2.75578768e-13,
        3.84631481e-13],
       [1.00730290e-03, 5.72257526e-02, 1.34035340e-03, 6.65092647e-01,
        2.75333911e-01],
       [2.08249821e-05, 4.83725955e-07, 1.95186818e-11, 9.81732621e-04,
        9.98996973e-01],
       [9.99999940e-01, 3.62774255e-11, 6.37677827e-11, 1.92503111e-10,
        1.42245091e-10],
       [1.29936814e-01, 4.21307086e-05, 3.51125891e-06, 8.77872203e-03,
        8.61238778e-01],
       [9.99990046e-01, 5.69632475e-09, 3.74583742e-09, 6.74399985e-07,
        9.22276013e-06],
       [1.28411793e-05, 1.30465448e-01, 7.51612561e-06, 8.05001497e-01,
        6.45127445e-02],
       [1.78256020e-01, 1.00485990e-02, 5.48207936e-05, 3.79701257e-01,
        4.31939214e-01],
       [9.99999583e-01, 2.13776746e-10, 6.09901921e-11, 3.22761871e-08,
        3.09697043e-07],
       [1.15087496e-04, 8.31787109e-01, 1.56512201e-01, 1.14013907e-02,
        1.84151490e-04],
       [9.99999940e-01, 6.83683931e-14, 4.79056085e-13, 1.24546218e-12,
        5.17932702e-13],
       [1.88411415e-01, 1.24890450e-03, 5.95483556e-03, 1.63057938e-01,
        6.41326845e-01],
       [2.12751655e-03, 9.30602849e-01, 2.18930449e-02, 4.26748469e-02,
        2.70170020e-03],
       [9.99997914e-01, 6.68790108e-07, 2.85858519e-08, 8.10713004e-07,
        4.23714482e-07],
       [4.69133374e-04, 9.55850482e-01, 1.62037276e-02, 2.63245087e-02,
        1.15206011e-03]], dtype=float32)
```

```python
comp = pd.DataFrame(y_test_encoded)  # Creating a dataframe
comp.columns = ['Actual Value']  # Changing the column name
comp
```

	Actual Value
0	4
1	0
2	4
3	3
4	0
5	0

```
6       0
7       4
8       1
9       4
10      1
11      4
12      0
13      1
14      2
15      0
16      2
17      4
18      3
19      0
20      2
21      4
22      4
23      0
24      0
25      0
26      3
27      4
28      0
29      4
30      0
31      3
32      3
33      0
34      1
35      0
36      4
37      1
38      0
39      1
```

```python
# Print the model summary

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 64)                384

 dense_6 (Dense)             (None, 128)               8320


 dense_7 (Dense)             (None, 64)                8256

 dense_8 (Dense)             (None, 32)                2080

 dense_9 (Dense)             (None, 5)                 165

=================================================================
Total params: 19,205
Trainable params: 19,205
Non-trainable params: 0
_________________________________________________________________
```

```python
# Task 3 : Test the model with random data
```

```python
# Generate random data for testing

random_data = np.random.rand(1, 5)
random_data
```

```
array([[0.87039758, 0.52583504, 0.74177248, 0.71396893, 0.03728909]])
```

```python
# Make predictions

predictions = model.predict(random_data)
predictions
```

```
WARNING:tensorflow:6 out of the last 9 calls to <function
Model.make_predict_function.<locals>.predict_function at 0x7fc722bf49d0>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2)
passing tensors with different shapes, (3) passing Python objects instead of
tensors. For (1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid unnecessary
retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1 [==============================] - 0s 77ms/step
```

```
array([[9.9052775e-01, 3.0603227e-05, 6.6905326e-05, 1.3001083e-03,
        8.0746198e-03]], dtype=float32)
```

```python
# Get the predicted drug class

predicted_class = np.argmax(predictions)
```

```python
# Print the predicted class
```

```python
print("Predicted Drug Class :", predicted_class)
```

```
Predicted Drug Class : 0
```