**1) Import the libraries**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
import tensorflow as tf
import seaborn as sns

# Load the dataset
data = pd.read_csv('/content/diabetes.csv')
print(data.head())
```

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin
BMI  \
0            6      148             72             35        0  33.6

1            1       85             66             29        0  26.6

2            8      183             64              0        0  23.3

3            1       89             66             23       94  28.1

4            0      137             40             35      168  43.1


   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
```

```
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

```
data.isnull().sum()
```

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

**2) Normalize the features**

```
# Separate features and target variable
X = data.drop(columns=['Outcome'])  # Features
y = data['Outcome']  # Target variable

# Normalize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training, validation, and test sets
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y,
test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42)

# Print the shape of the datasets
print(f"Training set shape: {X_train.shape}")
print(f"Validation set shape: {X_val.shape}")
print(f"Test set shape: {X_test.shape}")
```

```
Training set shape: (537, 8)
Validation set shape: (115, 8)
Test set shape: (116, 8)
```

**3) Building the Neural Network Model**

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

# Build the neural network model
```

```
model = Sequential([
    Dense(12, input_shape=(X_train.shape[1],), activation='relu'),
    Dense(8, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/
dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━
┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━
┩
│ dense (Dense)                        │ (None, 12)                  │
108 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────
┤
│ dense_1 (Dense)                      │ (None, 8)                   │
104 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────
┤
│ dense_2 (Dense)                      │ (None, 1)                   │
9 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────
┘
```

```
 Total params: 221 (884.00 B)

 Trainable params: 221 (884.00 B)

 Non-trainable params: 0 (0.00 B)
```

**4) Evaluating the Model**

```
# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=10,
validation_data=(X_val, y_val), verbose=1)
```

```
Epoch 1/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8034 - loss:
0.3719 - val_accuracy: 0.7652 - val_loss: 0.5344
Epoch 2/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8452 - loss:
0.3559 - val_accuracy: 0.7652 - val_loss: 0.5365
Epoch 3/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8517 - loss:
0.3335 - val_accuracy: 0.7652 - val_loss: 0.5270
Epoch 4/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8682 - loss:
0.3042 - val_accuracy: 0.7652 - val_loss: 0.5354
Epoch 5/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8335 - loss:
0.3438 - val_accuracy: 0.7652 - val_loss: 0.5457
Epoch 6/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8603 - loss:
0.3305 - val_accuracy: 0.7652 - val_loss: 0.5336
Epoch 7/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8548 - loss:
0.3422 - val_accuracy: 0.7652 - val_loss: 0.5368
Epoch 8/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8463 - loss:
0.3602 - val_accuracy: 0.7652 - val_loss: 0.5449
Epoch 9/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8353 - loss:
0.3570 - val_accuracy: 0.7652 - val_loss: 0.5425
Epoch 10/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8829 - loss:
0.2869 - val_accuracy: 0.7652 - val_loss: 0.5401
Epoch 11/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8474 - loss:
0.3338 - val_accuracy: 0.7652 - val_loss: 0.5286
Epoch 12/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8551 - loss:
0.3091 - val_accuracy: 0.7652 - val_loss: 0.5383
Epoch 13/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8613 - loss:
0.3220 - val_accuracy: 0.7652 - val_loss: 0.5496
Epoch 14/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8539 - loss:
0.3395 - val_accuracy: 0.7652 - val_loss: 0.5279
Epoch 15/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8604 - loss:
0.3258 - val_accuracy: 0.7652 - val_loss: 0.5448
Epoch 16/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8623 - loss:
0.3256 - val_accuracy: 0.7652 - val_loss: 0.5440
Epoch 17/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8480 - loss:
```

```
0.3394 - val_accuracy: 0.7652 - val_loss: 0.5376
Epoch 18/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8565 - loss:
0.3309 - val_accuracy: 0.7652 - val_loss: 0.5558
Epoch 19/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8684 - loss:
0.3125 - val_accuracy: 0.7652 - val_loss: 0.5364
Epoch 20/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8612 - loss:
0.3245 - val_accuracy: 0.7652 - val_loss: 0.5437
Epoch 21/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8490 - loss:
0.3538 - val_accuracy: 0.7652 - val_loss: 0.5425
Epoch 22/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8625 - loss:
0.3365 - val_accuracy: 0.7652 - val_loss: 0.5444
Epoch 23/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8392 - loss:
0.3459 - val_accuracy: 0.7652 - val_loss: 0.5390
Epoch 24/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8337 - loss:
0.3508 - val_accuracy: 0.7652 - val_loss: 0.5487
Epoch 25/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8323 - loss:
0.3614 - val_accuracy: 0.7652 - val_loss: 0.5519
Epoch 26/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8246 - loss:
0.3542 - val_accuracy: 0.7652 - val_loss: 0.5466
Epoch 27/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8603 - loss:
0.3284 - val_accuracy: 0.7652 - val_loss: 0.5408
Epoch 28/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8582 - loss:
0.3398 - val_accuracy: 0.7565 - val_loss: 0.5390
Epoch 29/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8543 - loss:
0.3299 - val_accuracy: 0.7652 - val_loss: 0.5424
Epoch 30/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8402 - loss:
0.3507 - val_accuracy: 0.7652 - val_loss: 0.5415
Epoch 31/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8517 - loss:
0.3429 - val_accuracy: 0.7652 - val_loss: 0.5317
Epoch 32/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8618 - loss:
0.3131 - val_accuracy: 0.7652 - val_loss: 0.5375
Epoch 33/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8525 - loss:
0.3270 - val_accuracy: 0.7652 - val_loss: 0.5531
Epoch 34/100
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8464 - loss:
0.3481 - val_accuracy: 0.7652 - val_loss: 0.5377
Epoch 35/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8690 - loss:
0.3072 - val_accuracy: 0.7652 - val_loss: 0.5373
Epoch 36/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8623 - loss:
0.3209 - val_accuracy: 0.7565 - val_loss: 0.5388
Epoch 37/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8477 - loss:
0.3474 - val_accuracy: 0.7652 - val_loss: 0.5560
Epoch 38/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8460 - loss:
0.3441 - val_accuracy: 0.7652 - val_loss: 0.5424
Epoch 39/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8559 - loss:
0.3359 - val_accuracy: 0.7739 - val_loss: 0.5329
Epoch 40/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8536 - loss:
0.3262 - val_accuracy: 0.7652 - val_loss: 0.5492
Epoch 41/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8548 - loss:
0.3084 - val_accuracy: 0.7652 - val_loss: 0.5496
Epoch 42/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8563 - loss:
0.3098 - val_accuracy: 0.7652 - val_loss: 0.5660
Epoch 43/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8648 - loss:
0.3094 - val_accuracy: 0.7652 - val_loss: 0.5351
Epoch 44/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8676 - loss:
0.3137 - val_accuracy: 0.7652 - val_loss: 0.5522
Epoch 45/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8735 - loss:
0.3016 - val_accuracy: 0.7565 - val_loss: 0.5511
Epoch 46/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8614 - loss:
0.3244 - val_accuracy: 0.7652 - val_loss: 0.5511
Epoch 47/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8836 - loss:
0.2745 - val_accuracy: 0.7652 - val_loss: 0.5508
Epoch 48/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8568 - loss:
0.3251 - val_accuracy: 0.7478 - val_loss: 0.5320
Epoch 49/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8508 - loss:
0.3314 - val_accuracy: 0.7478 - val_loss: 0.5533
Epoch 50/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8570 - loss:
0.3136 - val_accuracy: 0.7478 - val_loss: 0.5495
```

```
Epoch 51/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8738 - loss:
0.2853 - val_accuracy: 0.7391 - val_loss: 0.5462
Epoch 52/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8592 - loss:
0.3178 - val_accuracy: 0.7478 - val_loss: 0.5524
Epoch 53/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8973 - loss:
0.2758 - val_accuracy: 0.7391 - val_loss: 0.5462
Epoch 54/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8548 - loss:
0.3318 - val_accuracy: 0.7565 - val_loss: 0.5502
Epoch 55/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8771 - loss:
0.2991 - val_accuracy: 0.7478 - val_loss: 0.5566
Epoch 56/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8596 - loss:
0.3297 - val_accuracy: 0.7478 - val_loss: 0.5624
Epoch 57/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8667 - loss:
0.3246 - val_accuracy: 0.7391 - val_loss: 0.5480
Epoch 58/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8507 - loss:
0.3338 - val_accuracy: 0.7478 - val_loss: 0.5420
Epoch 59/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8638 - loss:
0.3279 - val_accuracy: 0.7478 - val_loss: 0.5476
Epoch 60/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8429 - loss:
0.3315 - val_accuracy: 0.7478 - val_loss: 0.5709
Epoch 61/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8591 - loss:
0.3303 - val_accuracy: 0.7478 - val_loss: 0.5737
Epoch 62/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8403 - loss:
0.3385 - val_accuracy: 0.7478 - val_loss: 0.5532
Epoch 63/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8360 - loss:
0.3411 - val_accuracy: 0.7478 - val_loss: 0.5564
Epoch 64/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8746 - loss:
0.3089 - val_accuracy: 0.7478 - val_loss: 0.5519
Epoch 65/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8427 - loss:
0.3440 - val_accuracy: 0.7478 - val_loss: 0.5442
Epoch 66/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8689 - loss:
0.3255 - val_accuracy: 0.7478 - val_loss: 0.5479
Epoch 67/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8904 - loss:
```

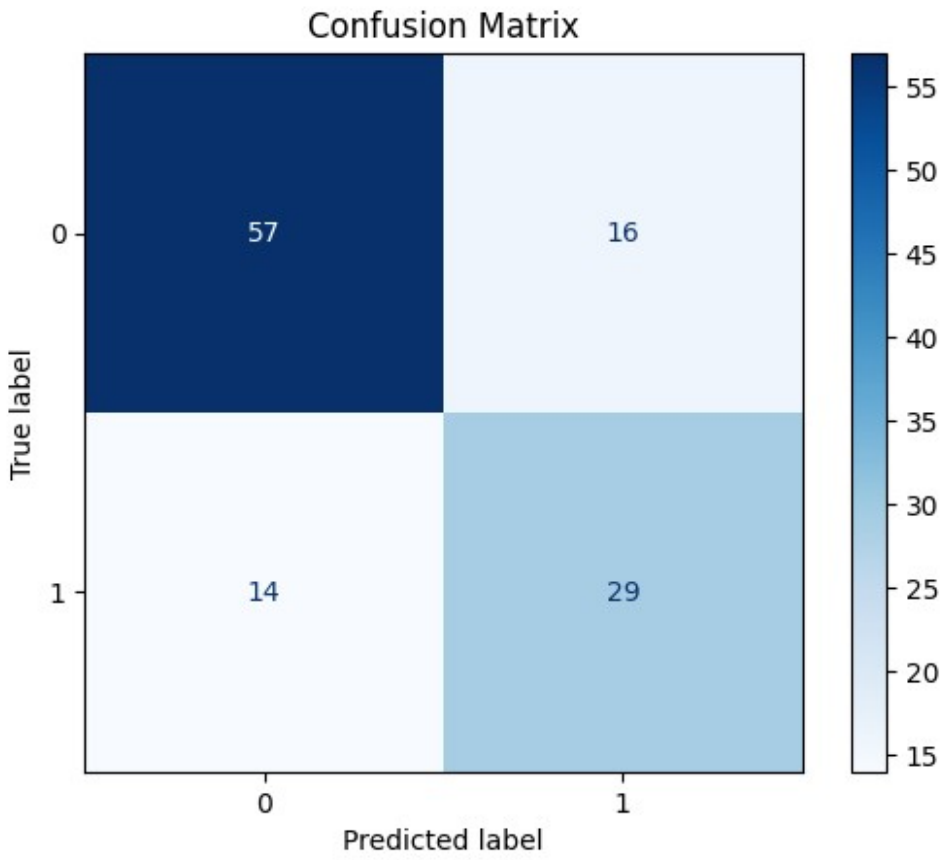
```
0.2833 - val_accuracy: 0.7478 - val_loss: 0.5686
Epoch 68/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8673 - loss:
0.2969 - val_accuracy: 0.7478 - val_loss: 0.5463
Epoch 69/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8470 - loss:
0.3101 - val_accuracy: 0.7478 - val_loss: 0.5551
Epoch 70/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8595 - loss:
0.3114 - val_accuracy: 0.7478 - val_loss: 0.5656
Epoch 71/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8627 - loss:
0.3124 - val_accuracy: 0.7565 - val_loss: 0.5429
Epoch 72/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8700 - loss:
0.3043 - val_accuracy: 0.7478 - val_loss: 0.5486
Epoch 73/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8548 - loss:
0.3074 - val_accuracy: 0.7478 - val_loss: 0.5604
Epoch 74/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8347 - loss:
0.3392 - val_accuracy: 0.7478 - val_loss: 0.5521
Epoch 75/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8724 - loss:
0.3012 - val_accuracy: 0.7478 - val_loss: 0.5528
Epoch 76/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8732 - loss:
0.3103 - val_accuracy: 0.7478 - val_loss: 0.5692
Epoch 77/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8600 - loss:
0.3121 - val_accuracy: 0.7478 - val_loss: 0.5602
Epoch 78/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8827 - loss:
0.3007 - val_accuracy: 0.7478 - val_loss: 0.5556
Epoch 79/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8502 - loss:
0.3189 - val_accuracy: 0.7478 - val_loss: 0.5528
Epoch 80/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8757 - loss:
0.2922 - val_accuracy: 0.7478 - val_loss: 0.5682
Epoch 81/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8794 - loss:
0.2879 - val_accuracy: 0.7391 - val_loss: 0.5851
Epoch 82/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8411 - loss:
0.3337 - val_accuracy: 0.7478 - val_loss: 0.5675
Epoch 83/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8427 - loss:
0.3404 - val_accuracy: 0.7652 - val_loss: 0.5470
Epoch 84/100
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8758 - loss:
0.2936 - val_accuracy: 0.7478 - val_loss: 0.5577
Epoch 85/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8675 - loss:
0.3131 - val_accuracy: 0.7478 - val_loss: 0.5647
Epoch 86/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8725 - loss:
0.2876 - val_accuracy: 0.7478 - val_loss: 0.5785
Epoch 87/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8613 - loss:
0.3088 - val_accuracy: 0.7391 - val_loss: 0.5693
Epoch 88/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8590 - loss:
0.3031 - val_accuracy: 0.7478 - val_loss: 0.5637
Epoch 89/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8692 - loss:
0.3026 - val_accuracy: 0.7565 - val_loss: 0.5593
Epoch 90/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8592 - loss:
0.3002 - val_accuracy: 0.7478 - val_loss: 0.5726
Epoch 91/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8726 - loss:
0.2945 - val_accuracy: 0.7478 - val_loss: 0.5691
Epoch 92/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8643 - loss:
0.3092 - val_accuracy: 0.7565 - val_loss: 0.5565
Epoch 93/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8590 - loss:
0.3106 - val_accuracy: 0.7478 - val_loss: 0.5805
Epoch 94/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8574 - loss:
0.3088 - val_accuracy: 0.7478 - val_loss: 0.5677
Epoch 95/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8589 - loss:
0.2984 - val_accuracy: 0.7478 - val_loss: 0.5647
Epoch 96/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8751 - loss:
0.3106 - val_accuracy: 0.7478 - val_loss: 0.5655
Epoch 97/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8579 - loss:
0.3030 - val_accuracy: 0.7565 - val_loss: 0.5653
Epoch 98/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8782 - loss:
0.2749 - val_accuracy: 0.7391 - val_loss: 0.5741
Epoch 99/100
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8762 - loss:
0.2917 - val_accuracy: 0.7478 - val_loss: 0.5734
Epoch 100/100
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8487 - loss:
0.3270 - val_accuracy: 0.7478 - val_loss: 0.5684
```

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Evaluate the model on the test data (as before)
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy:.4f}")

# Make predictions on the test data (as before)
y_pred = (model.predict(X_test) > 0.5).astype("int32")

# Calculate precision, recall, and F1 score (as before)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")

# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0,
1])
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```

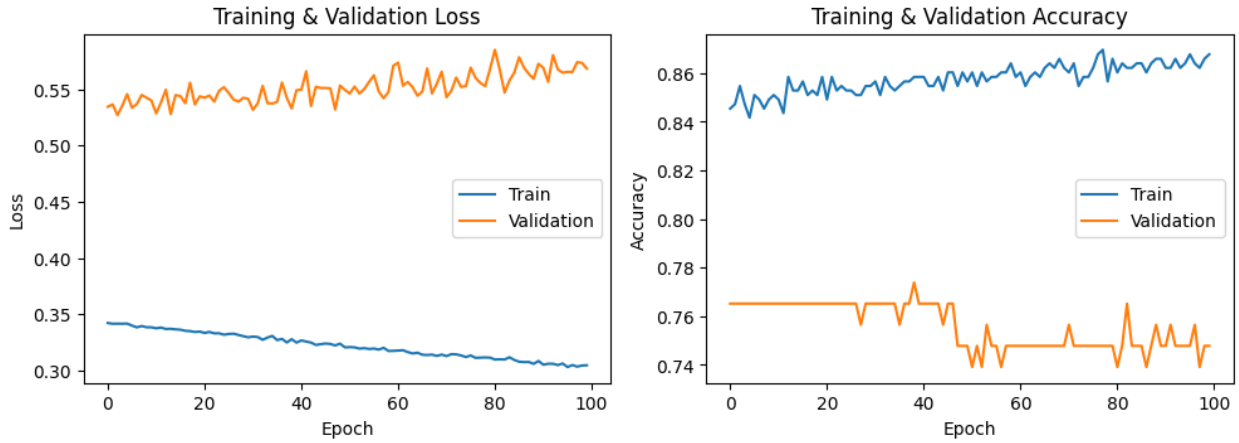
```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7205 - loss: 0.6174

Test Accuracy: 0.7414
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Precision: 0.6444
Recall: 0.6744
F1 Score: 0.6591
```




**5) Visualization and Analysis**

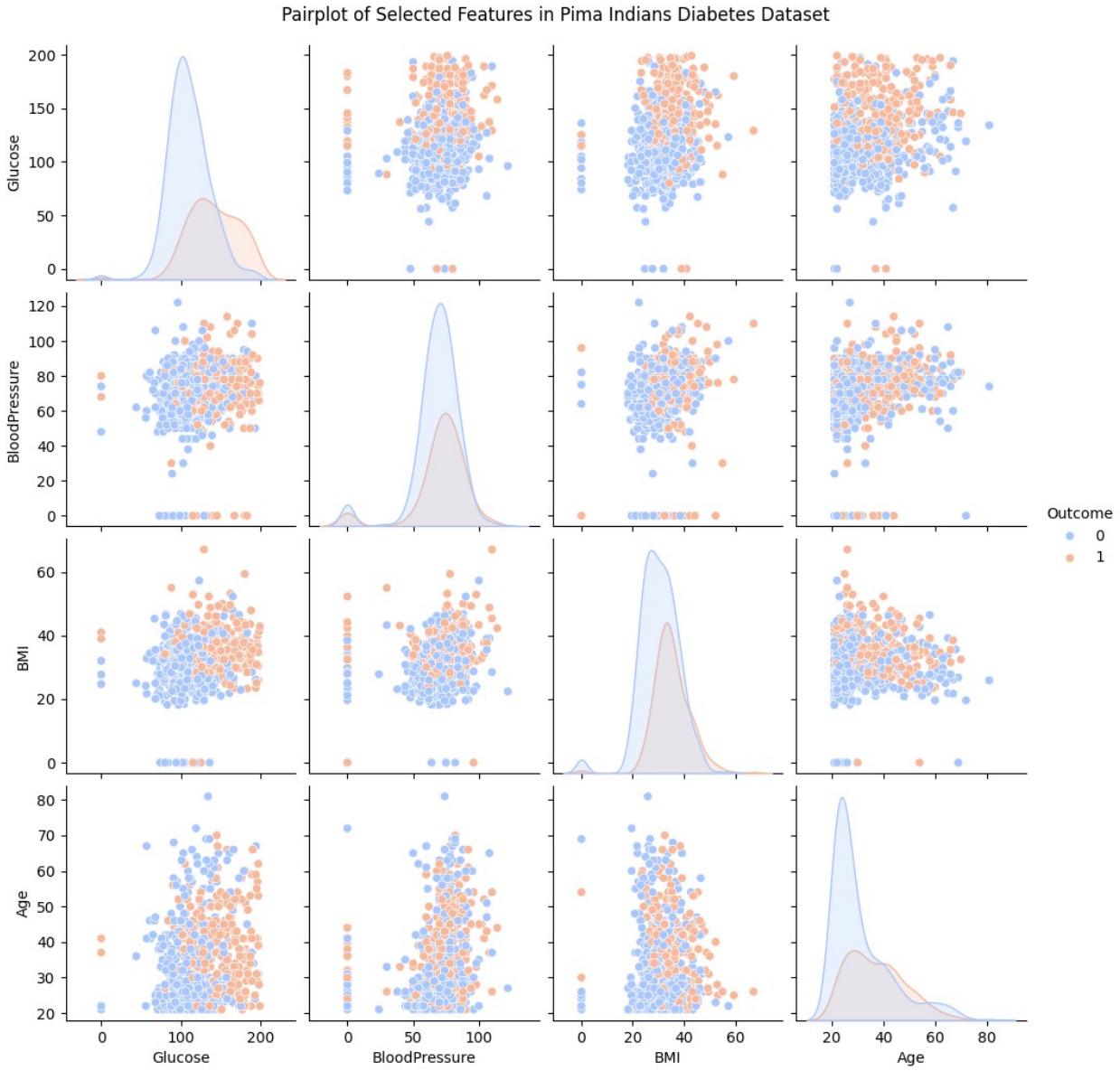
```
# Create a figure with multiple subplots
plt.figure(figsize=(15, 10))

# Plot training and validation loss
plt.subplot(3, 3, 1)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.title('Training & Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plot training and validation accuracy
plt.subplot(3, 3, 2)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.title('Training & Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

```
# Adjust layout and display the plots
plt.tight_layout()
plt.show()
```




```
targeted_columns = ['Glucose', 'BloodPressure', 'BMI', 'Age',
'Outcome']

# Create a pairplot for the selected columns
plt.figure(figsize=(12, 10))
sns.pairplot(data[targeted_columns], hue='Outcome', diag_kind='kde',
palette='coolwarm')
plt.suptitle('Pairplot of Selected Features in Pima Indians Diabetes
Dataset', y=1.02)
plt.show()
```

```
<Figure size 1200x1000 with 0 Axes>
```




### 6) Model Persistence

```
# Save the model
model.save('pima_diabetes_model.h5')

# Load the model
loaded_model = tf.keras.models.load_model('pima_diabetes_model.h5')

# Use the loaded model to make predictions
loaded_y_pred = (loaded_model.predict(X_test) > 0.5).astype("int32")

# Evaluate the loaded model
loaded_test_accuracy = accuracy_score(y_test, loaded_y_pred)
print(f"Loaded Model Test Accuracy: {loaded_test_accuracy:.4f}")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.

4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
Loaded Model Test Accuracy: 0.7328
```