AI BASED DIABETES PREDICTION SYSTEM

TEAM MEMBER

AU922321104035 : S. Santhiya Devi

PHASE 2 SUBMISSION DOCUMENT

Introduction :

Creating a diabetes prediction model using ensemble methods and deep learning architectures involves several steps. Below, I'll provide a comprehensive outline of the process, including code examples using Python and popular libraries like Scikit-Learn and TensorFlow/Keras.

Steps :

1. Random Forest :

- Random Forest is an ensemble method that combines multiple decision trees. It's particularly effective for classification tasks like diabetes prediction.

- Each tree in the forest is trained on a random subset of the data, and the final prediction is made by taking a majority vote or averaging the predictions of individual trees.

2. Gradient Boosting :

- Gradient Boosting algorithms like XGBoost, LightGBM, and CatBoost can be used for diabetes prediction.

- They build trees sequentially, where each tree corrects the errors made by the previous ones. This often leads to improved accuracy.

3. AdaBoost :

- AdaBoost is an ensemble method that combines multiple weak learners (e.g., shallow decision trees) into a strong learner.

- It assigns weights to each training sample and focuses on the samples that are misclassified by the previous models.

4. Stacking :

- Stacking involves training multiple base models and then combining their predictions using another model, often called a meta-learner or blender.

- For diabetes prediction, you could use a combination of models like logistic regression, support vector machines, or neural networks as base models, and then use a meta-learner to make the final prediction.

5. Bagging :

- Bagging, as seen in Random Forest, involves training multiple models independently on different subsets of the data and then averaging or taking a majority vote of their predictions.

- You can use bagging with various base classifiers, such as decision trees, support vector machines, or k-nearest neighbors.

6. Voting Classifier :

- A voting classifier combines the predictions of multiple base classifiers (e.g., logistic regression, decision trees, k-nearest neighbors) and selects the class with the most votes.

- You can use techniques like hard voting (majority vote) or soft voting (weighted average of class probabilities).

7. Ensemble of Neural Networks :

- You can create an ensemble of different neural network architectures or variations (e.g., CNN, LSTM, MLP) and combine their predictions.

- This can improve the model's ability to capture complex relationships in the data.

When applying ensemble methods for diabetes prediction, it's essential to preprocess the data, perform feature selection, and tune hyperparameters to achieve the best results. Additionally, use techniques like cross-validation to assess the ensemble's performance and prevent overfitting. The choice of ensemble method may depend on the size of your dataset, the computational resources available, and the specific characteristics of your diabetes prediction problem.

Source program :

```
import pandas as pd, numpy as np, seaborn as sns
```

```python
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
linkcode
```python
data = pd.read_csv("../input/diabetes-data-set/diabetes.csv")
```

In [3]:
```python
data.head()
```

Out[3]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

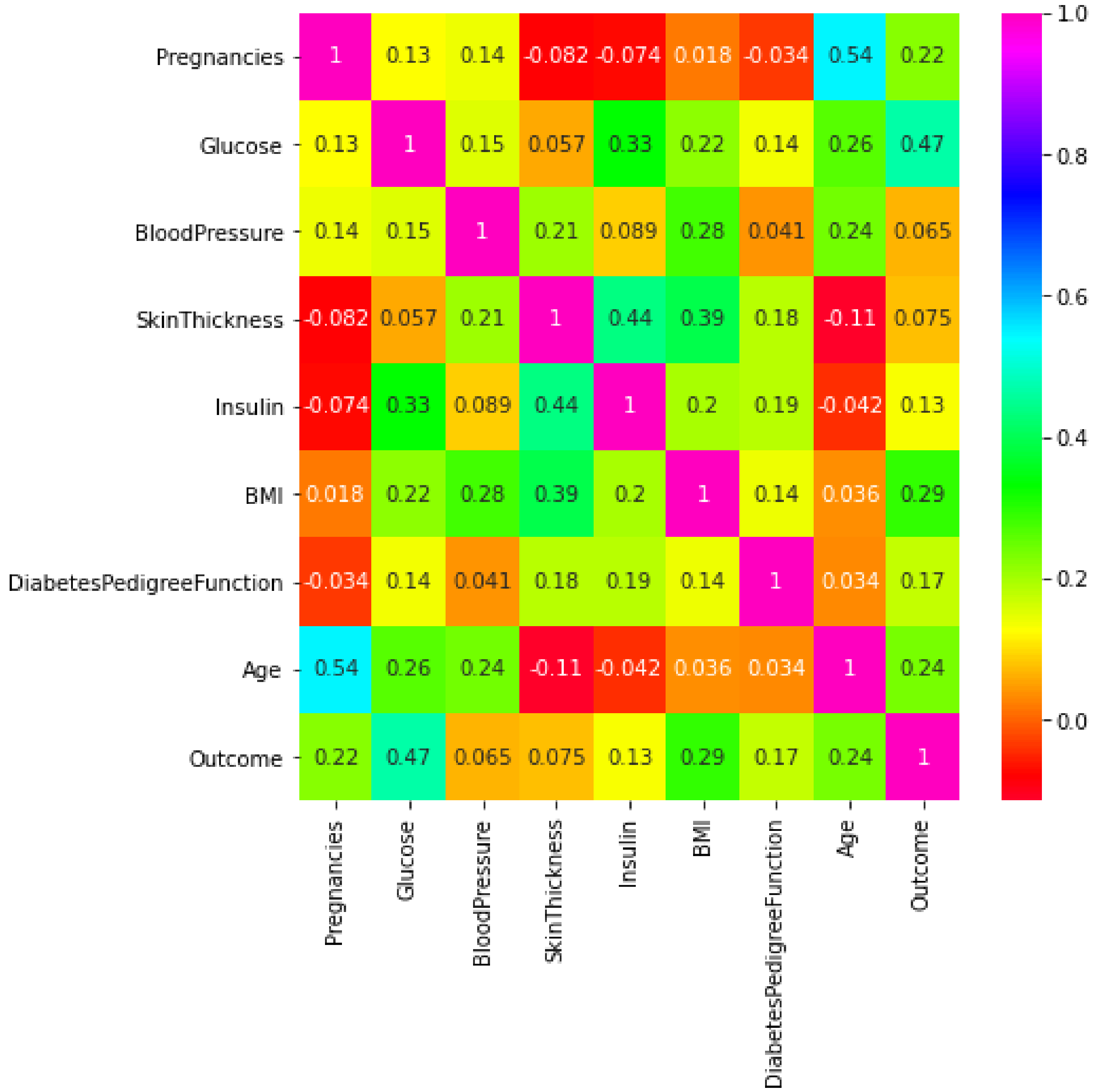
In [4]:
```python
data.info()
```

```
<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
feature-selection-techniques

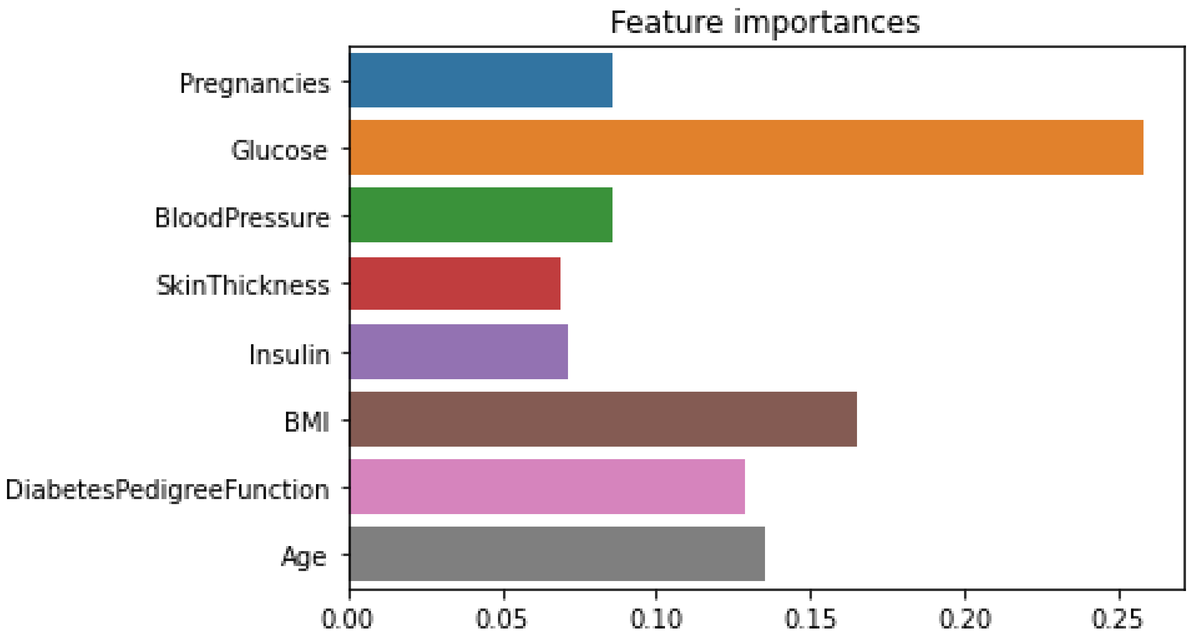
In [5]:

```
corrmat = data.corr()
top_corr_feat = corrmat.index
plt.figure(figsize=(7,7))
#plot heat map
g = sns.heatmap(data[top_corr_feat].corr(),annot=True,
          cmap='gist_rainbow')
```



In [6]:

```
#select from model technique for feature importance

from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier

feat = data.drop("Outcome",axis=1)
target = data["Outcome"]

feature_names = np.array(feat.columns)
RFC = RandomForestClassifier().fit(feat,target)
importance = np.abs(RFC.feature_importances_)
sns.barplot(x=importance, y=feature_names)
plt.title("Feature importances")
plt.show()
```



In [7]:

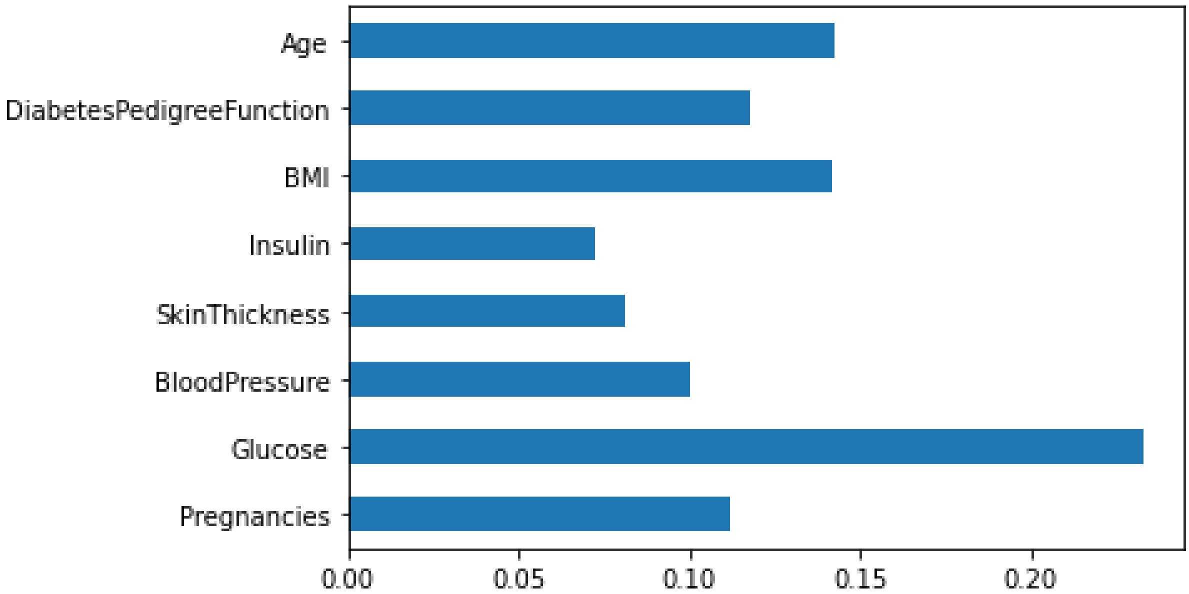
```
#feature importance
from sklearn.ensemble import ExtraTreesClassifier
model = ExtraTreesClassifier()
model.fit(feat,target)
model.feature_importances_
```

Out[7]:

```
array([0.11151513, 0.23269951, 0.09989779, 0.08119882, 0.07253244,
       0.14147394, 0.1179607 , 0.14272168])
```

In [8]:

```
feat_importance = pd.Series(model.feature_importances_, index=feat.columns)
feat_importance.plot(kind='barh')
plt.show()
```



In [9]:

```
#univariate selection
#apply selctkbest to selct top 5 features

from sklearn.feature_selection import SelectKBest, chi2

bestfeatures = SelectKBest(score_func = chi2, k=5)
```

```
fit = bestfeatures.fit(feat,target)

dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(feat.columns)
#concat the two dataframes for better viz
feat_scores = pd.concat([dfcolumns,dfscores],axis=1)
feat_scores.columns =['Feature', 'Score']
feat_scores.nlargest(5, 'Score') #top 5 features
```

Out[9]:

	Feature	Score
4	Insulin	2175.565273
1	Glucose	1411.887041
7	Age	181.303689
5	BMI	127.669343
0	Pregnancies	111.519691

In [10]:

```
report = feat_scores.nlargest(5, 'Score')
```

In [11]:

```
#use top features
optimum_features = report['Feature']
```

In [12]:

```
new_data = data.loc[0:,list(optimum_features)].join(data["Outcome"])
```

In [13]:

```
new_data.head()
```

Out[13]:

	Insulin	Glucose	Age	BMI	Pregnancies	Outcome
0	0	148	50	33.6	6	1
1	0	85	31	26.6	1	0

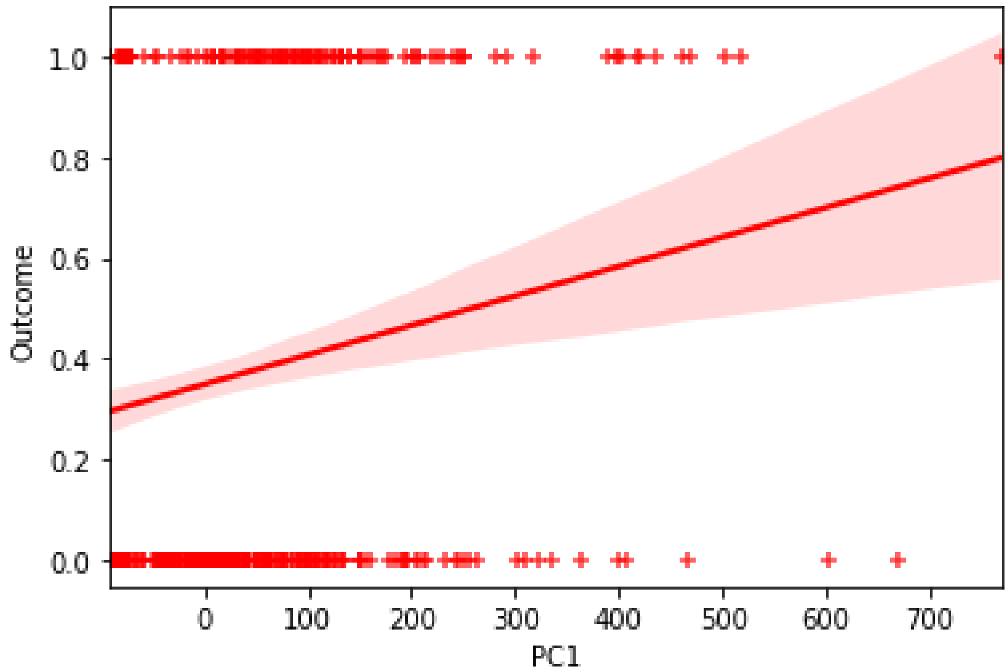
	Insulin	Glucose	Age	BMI	Pregnancies	Outcome
2	0	183	32	23.3	8	1
3	94	89	21	28.1	1	0
4	168	137	33	43.1	0	1

In [14]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=1)
X_pca = pca.fit_transform(new_data.drop('Outcome',axis=1))
PCA_df = pd.DataFrame(data = X_pca, columns = ['PC1'])
PCA_df = pd.concat([PCA_df, new_data['Outcome']], axis = 1)
PCA_df.head()
```

Out[14]:

	PC1	Outcome
0	-76.787155	1
1	-82.989683	0
2	-73.434318	1
3	10.995500	0
4	89.508314	1

In [15]:
```
sns.regplot(x=PCA_df['PC1'],
           y = PCA_df['Outcome'], color = 'red',
          marker = '+', fit_reg = True)
plt.show()
```



In [16]:
```
#split dataset into training and test set
from sklearn.model_selection import train_test_split
X = new_data.drop("Outcome",axis=1).values
y = new_data["Outcome"].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1, random_state = 45, shuffle = True, s
tratify = y)
```

In [17]:
```
from sklearn.ensemble import RandomForestClassifier as RFC,ExtraTreesClassifier as XTC
from sklearn.linear_model import LogisticRegression as LR
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import *
mcc= make_scorer(matthews_corrcoef)

def evaluate_model(cv):
   model = RFC()
   # evaluate the model
   scores = cross_val_score(model, X_train, y_train,
                    scoring= mcc,
                    cv=cv, n_jobs=-1)
   # return scores
   return scores.mean()
```

In [18]:
```
#iterate over a range of folds to get best K value:
```

```
folds = range(5,11)

# record mean and min/max of each set of results
means = list()
# evaluate each k value

for k in folds:
    # define the test condition
    cv = KFold(n_splits=k, shuffle=True, random_state=1)
    # evaluate k value
    k_mean = evaluate_model(cv)
    # report performance
    print('> folds=%d, rfc mean score = %.3f ' % (k, k_mean))
    # store mean accuracy
    means.append(k_mean)
```

```
> folds=5, rfc mean score = 0.454
> folds=6, rfc mean score = 0.438
> folds=7, rfc mean score = 0.441
> folds=8, rfc mean score = 0.437
> folds=9, rfc mean score = 0.427
> folds=10, rfc mean score = 0.445
```

In [19]:

```
#save randomforestclassif
model = RFC()
model.fit(X_train,y_train)
import pickle
model1 = pickle.dumps(model)
```

In [20]:

```
#evaluate extratreesclassif

def evaluate_model(cv):
    model = XTC()
    # evaluate the model
    scores = cross_val_score(model, X_train, y_train,
                             scoring= mcc,
                             cv=cv, n_jobs=-1)
    # return scores
    return scores.mean()
```

In [21]:

```
folds = range(5,11)

# record mean and min/max of each set of results
means = list()
# evaluate each k value

for k in folds:
    # define the test condition
    cv = KFold(n_splits=k, shuffle=True, random_state=1)
    # evaluate k value
    k_mean = evaluate_model(cv)
    # report performance
    print('> folds=%d, xtc mean score = %.3f ' % (k, k_mean))
    # store mean accuracy
    means.append(k_mean)
```

```
> folds=5, xtc mean score = 0.457
```

```
> folds=6, xtc mean score = 0.436
> folds=7, xtc mean score = 0.455
> folds=8, xtc mean score = 0.453
> folds=9, xtc mean score = 0.444
> folds=10, xtc mean score = 0.454
```

In [22]:

```
#save xtratreesclassif
model = XTC()
model.fit(X_train,y_train)
model2 = pickle.dumps(model)
```

In [23]:

```
rfc = pickle.loads(model1)
xtc = pickle.loads(model2)
```

In [24]:

```
#stack classifier with extratrees and randomforest as base estimators

from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression as LR
base_model, end_model = [('random_forest',rfc),('xtra_trees',xtc)], LR()
final_model = StackingClassifier(base_model,end_model, cv=10)
scores = cross_val_score(final_model, X_train, y_train,
                         scoring= mcc,
                         cv=10, n_jobs=-1)
scores.mean()
```

Out[24]:

```
0.459549190144178
```

In [25]:

```
final_model.fit(X_train, y_train)
# #save the final_model
model3 = pickle.dumps(final_model)
```

In [26]:

```
#run predictions with the 3 models
rfc_pred = rfc.predict(X_test)
xtc_pred = xtc.predict(X_test)
final_model_pred = final_model.predict(X_test)
```

In [27]:

```
from sklearn.metrics import classification_report as report, confusion_matrix as cm
print("report on random forest classifier : \n", report(y_pred=rfc_pred,y_true=y_test))
print('\n')
print("report on extra trees classifier : \n", report(y_pred=xtc_pred,y_true=y_test))
print('\n')
print("report on stacked classifier : \n", report(y_pred=final_model_pred,y_true=y_test))
```

```
report on random forest classifier :
              precision    recall  f1-score   support

           0       0.78      0.86      0.82        50
           1       0.68      0.56      0.61        27

    accuracy                           0.75        77
   macro avg       0.73      0.71      0.72        77
weighted avg       0.75      0.75      0.75        77
```

```
report on extra trees classifier :
              precision    recall  f1-score   support

           0       0.74      0.84      0.79        50
           1       0.60      0.44      0.51        27

    accuracy                           0.70        77
   macro avg       0.67      0.64      0.65        77
weighted avg       0.69      0.70      0.69        77



report on stacked classifier :
              precision    recall  f1-score   support

           0       0.77      0.88      0.82        50
           1       0.70      0.52      0.60        27

    accuracy                           0.75        77
   macro avg       0.74      0.70      0.71        77
weighted avg       0.75      0.75      0.74        77
```

In [28]:

```
print("matrix of random forest classifier : \n", cm(y_pred=rfc_pred,y_true=y_test,labels=[0,1]))
print('\n')
print("matrix of extra trees classifier : \n", cm(y_pred=xtc_pred,y_true=y_test,labels=[0,1]))
print('\n')
print("matrix of stacked classifier : \n", cm(y_pred=final_model_pred,y_true=y_test,labels=[0,1]))
```

```
matrix of random forest classifier :
 [[43  7]
 [12 15]]


matrix of extra trees classifier :
 [[42  8]
 [15 12]]


matrix of stacked classifier :
 [[44  6]
 [13 14]]
```

Deep learning architecture for diabetes prediction :

1.Data Collection and Preprocessing :

- Gather a comprehensive dataset that includes relevant features such as age, gender, family history, BMI, blood pressure, glucose levels, and other health-related factors. Ensure that the dat a is clean and well-structured.
- Split the dataset into training, validation, and test sets. Typically, you might use 70-80% for tr aining, 10-15% for validation, and the remaining for testing.

2. Feature Selection/Engineering :

- Analyze the dataset to identify which features are most relevant for diabetes prediction. You may need to perform feature selection or engineering to improve model performance.

3. Deep Learning Model :

- Choose a deep learning architecture suitable for this task. For a binary classification problem like diabetes prediction, a neural network with multiple hidden layers can work well. Some com mon choices include convolutional neural networks (CNNs) or recurrent neural networks (RNN s).
- Design the architecture by specifying the number of layers, neurons, and activation functions. Experiment with different architectures to find the best-performing one.

4. Training :
- Use the training dataset to train the deep learning model. Implement backpropagation and o ptimization techniques like stochastic gradient descent (SGD) or Adam to minimize the loss fun ction.
- Apply techniques like batch normalization and dropout to prevent overfitting.

5. Hyperparameter Tuning :

- Tune hyperparameters like learning rate, batch size, and the number of epochs to optimize m odel performance. You can use techniques like grid search or random search.

6. Validation :

- Monitor the model's performance on the validation set during training to detect overfitting or underfitting.
- Adjust the model architecture and hyperparameters as needed based on validation results.

7. Testing and Evaluation :

- Once the model is trained, evaluate its performance on the test dataset using metrics such a s accuracy, precision, recall, F1-score, and ROC AUC.
- Assess the model's ability to make predictions and its generalization to new, unseen data.

8. Deployment :

- If the model meets your performance criteria, deploy it in a real-world healthcare setting. Ens ure that it complies with data privacy regulations and ethical considerations.
- Develop a user-friendly interface for healthcare professionals to input patient data and receiv e predictions.

9. Monitoring and Maintenance :

- Continuously monitor the model's performance in the production environment. Retrain the m odel periodically with updated data to maintain accuracy.

10. Interpretability :- Consider methods to interpret the model's predictions, especially in healthc are applications where transparency is critical. Techniques like SHAP values or LIME can help e xplain model predictions.

Remember that building a deep learning model for healthcare applications like diabetes prediction requires rigorous data handling, privacy considerations, and thorough evaluation. Collaboration with healthcare experts and adherence to relevant regulations and ethics are crucial throughout the development process.

SOURCE CODE :

```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

/kaggle/input/pima-indians-diabetes-database/diabetes.csv

1. Importing and reading the data.

In [2]:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
sns.set()
plt.style.use('ggplot')
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split, KFold
```

In [3]:

```
df = pd.read_csv("/kaggle/input/pima-indians-diabetes-database/diabetes.csv")
df.head(5)
```

Out[3]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
4	0	137	40	35	168	43.1	2.288	33	1

In [4]:

```
print(df.shape)
```

(768, 9)

In [5]:

```
print(df.columns.tolist())
```

['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']

In [6]:

```
df.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
```

In [7]:

```
df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
Pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	17.00
Glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	199.00

	count	mean	std	min	25%	50%	75%	max
BloodPressure	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	122.00
SkinThickness	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	99.00
Insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	846.00
BMI	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	67.10
DiabetesPedigreeFunction	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	2.42
Age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	81.00
Outcome	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	1.00

1.1 Checking for number of 0 values in each column.

- Pregnanices can take 0 values and so can outcomes.
- But for other columns which contain 0 values, we will have to use some imputation strategy.

In [8]:

```
print('Number of 0s in each column\n')
for col in df:
  print(f"{col} : {(df[col]==0).sum()}")
```

```
Number of 0s in each column

Pregnancies : 111
Glucose : 5
BloodPressure : 35
SkinThickness : 227
```

Insulin : 374
BMI : 11
DiabetesPedigreeFunction : 0
Age : 0
Outcome : 500

1.2 Checking for NAN values.

In [9]:

```
df.isnull().sum()
```

Out[9]:

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

2. EDA and Feature Engineering

2.1 Imputation of 0 values.

- As there are a lot of columns with 0 as values, we will come up with appropriate imputation strategy after checking their distribution.
- But first, we need to replace all the 0 values with nan.
- It is also good practice to create a copy of the dataset and perform all this operations so that we dont mess with the original data.

In [10]:

```
#Creating a copy of the datset.
df_copy = df.copy(deep = True)
df_copy[['Glucose','BloodPressure','SkinThickness','Insulin','BMI']] = df_copy[['Glucose','BloodPressure','SkinThickness','Insulin','BMI']].replace(0,np.NaN)

#Checking
df_copy.isnull().sum()
```

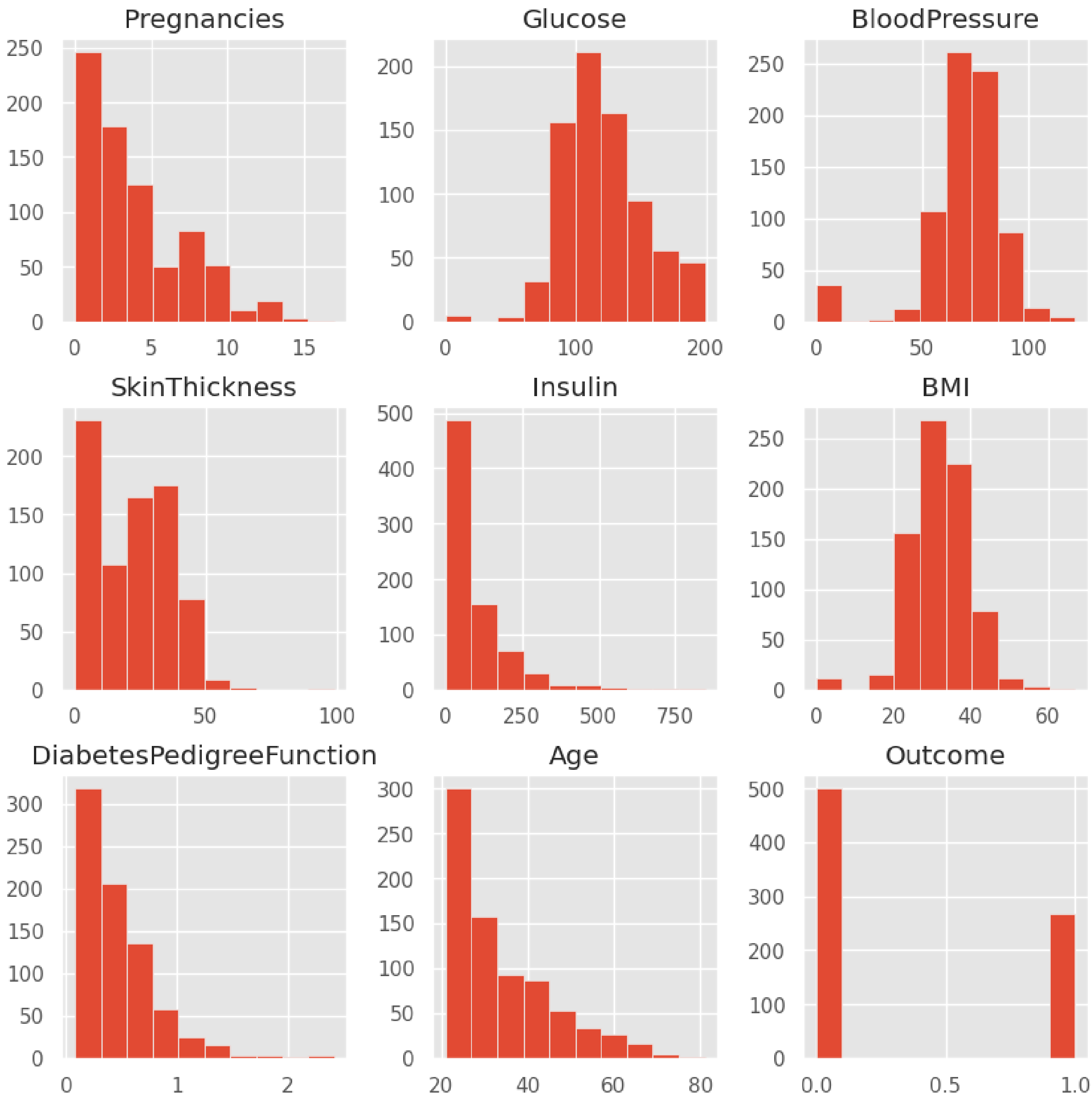
Out[10]:

```
Pregnancies                   0
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

Plotting the distribution of these columns to understand its skewness

In [11]:

```
ax = df.hist(figsize=(10,10))
```



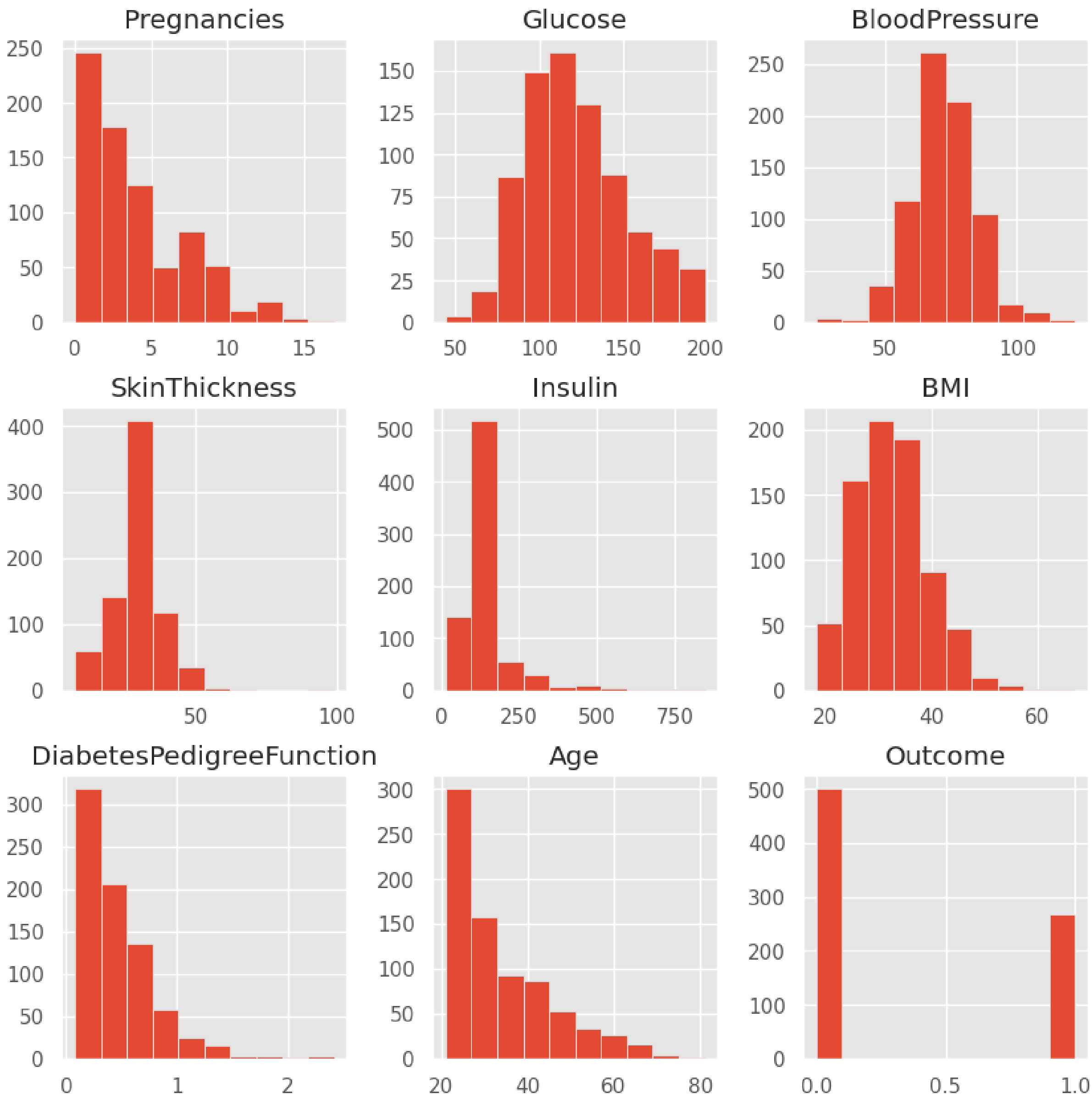
In [12]:

```
#Using mean strategy for glucose due to its central tendency.
df_copy['Glucose'].fillna(df_copy['Glucose'].mean(), inplace = True)

#Using mean strategy for blood pressure as well due to its central tendency.
df_copy['BloodPressure'].fillna(df_copy['BloodPressure'].mean(), inplace = True)

#Using median strategy for skin thickness as the data is skewed.
df_copy['SkinThickness'].fillna(df_copy['SkinThickness'].median(), inplace = True)
```

```
#Using median strategy for insulin as the data is skewed.
df_copy['Insulin'].fillna(df_copy['Insulin'].median(), inplace = True)

#Using mean strategy for BMI due to its central tendency.
df_copy['BMI'].fillna(df_copy['BMI'].mean(), inplace = True)
```

In [13]:

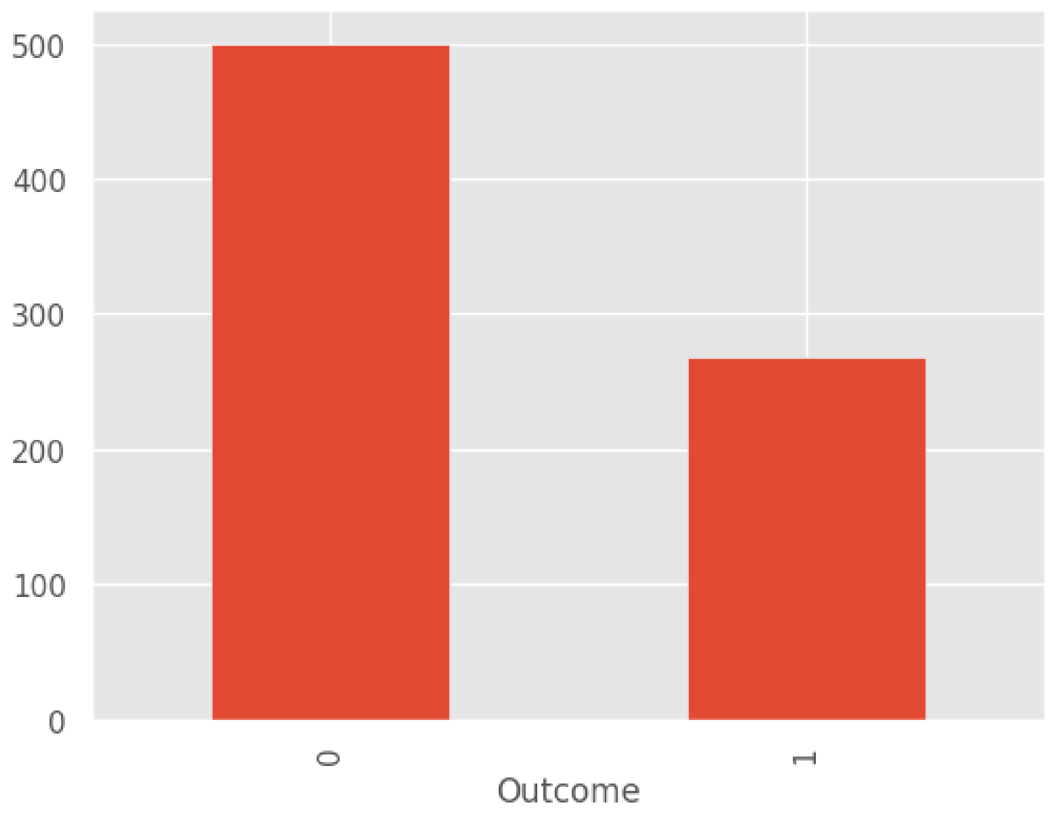
```
#Checking
ax = df_copy.hist(figsize=(10,10))
```



2.2 Comparing outcome values.

- We can see that the number of datapoints for patient not having diabetes is almost twice as that of patient having diabetes.
- This represents bias towards patient not having diabetes. (Imbalanced dataset)
- Due to this imbalance in the dataset, we need to come up with some sort of strategy to make sure that our model is not going to be biased against one particular class.
- There are several ways to counter this, I am going to use the stratified sampling technique.
- Some other ways include Sampling the dataset (Oversampling and undersampling), Ensemble methods etc.

In [14]:
```
ax = df_copy.Outcome.value_counts().plot(kind='bar')
```



2.3 Standardizing the data.

- Before we perform the stratified sampling, we need to standardize the data.

- We also need to define our features and label.

In [15]:
```
X = df_copy.drop('Outcome', axis=1)
y = df_copy.pop('Outcome')
X.head()
```

Out[15]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148.0	72.0	35.0	125.0	33.6	0.627	50
1	1	85.0	66.0	29.0	125.0	26.6	0.351	31
2	8	183.0	64.0	29.0	125.0	23.3	0.672	32
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33

In [16]:
```
y.head()
```

Out[16]:
```
0    1
1    0
2    1
3    0
4    1
Name: Outcome, dtype: int64
```

In [17]:
```
print(f'Shape of training data : {X.shape}')
print(f'Shape of test data : {y.shape}')
```

```
Shape of training data : (768, 8)
Shape of test data : (768,)
```

We need to scale the data as Neural Networks are prone to magnitude of values. It assigns higher importance to feature that has higher value.

In [18]:
```
#Scaling the data.
X_scaler = MinMaxScaler()
X = pd.DataFrame(X_scaler.fit_transform(X), columns=['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
    'BMI', 'DiabetesPedigreeFunction', 'Age'])
```

In [19]:
```
#Checking our standardized values
```

X.head()

Out[19]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	0.352941	0.670968	0.489796	0.304348	0.133413	0.314928	0.234415	0.483333
1	0.058824	0.264516	0.428571	0.239130	0.133413	0.171779	0.116567	0.166667
2	0.470588	0.896774	0.408163	0.239130	0.133413	0.104294	0.253629	0.183333
3	0.058824	0.290323	0.428571	0.173913	0.096154	0.202454	0.038002	0.000000
4	0.000000	0.600000	0.163265	0.304348	0.185096	0.509202	0.943638	0.200000

3. Model Building - 1

- Going with the fold value of 10 because it is one of the best starting values.

In [20]:

```
#Importing necessary modules.
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.optimizers import Adam
```

In [21]:

```
def model_constructor():
    model = Sequential()
    model.add(Dense(64, activation='relu', input_shape=(8,)))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    optimizer = Adam(learning_rate=0.01)
    #Compiling the model.
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
model = model_constructor()
model.summary()
```

Model: "sequential"

```
Layer (type)              Output Shape              Param #
=================================================================
dense (Dense)              (None, 64)               576

dense_1 (Dense)            (None, 32)                2080

dense_2 (Dense)            (None, 1)                33

=================================================================
Total params: 2,689
Trainable params: 2,689
Non-trainable params: 0
_________________________________________________________________
```

In [22]:

```
history = model.fit(X, y, validation_split=0.1, batch_size=32,epochs=100, verbose=2)
```

Epoch 1/100
22/22 - 1s - loss: 0.6346 - accuracy: 0.6700 - val_loss: 0.5556 - val_accuracy: 0.7662 - 1s/epoch - 59ms/step
Epoch 2/100
22/22 - 0s - loss: 0.5329 - accuracy: 0.7236 - val_loss: 0.4819 - val_accuracy: 0.7922 - 60ms/epoch - 3ms/step
Epoch 3/100
22/22 - 0s - loss: 0.4947 - accuracy: 0.7598 - val_loss: 0.4981 - val_accuracy: 0.7273 - 61ms/epoch - 3ms/step
Epoch 4/100
22/22 - 0s - loss: 0.5081 - accuracy: 0.7381 - val_loss: 0.4562 - val_accuracy: 0.8052 - 61ms/epoch - 3ms/step
Epoch 5/100
22/22 - 0s - loss: 0.4602 - accuracy: 0.7771 - val_loss: 0.5910 - val_accuracy: 0.7143 - 62ms/epoch - 3ms/step
Epoch 6/100
22/22 - 0s - loss: 0.4857 - accuracy: 0.7598 - val_loss: 0.4811 - val_accuracy: 0.7792 - 62ms/epoch - 3ms/step
Epoch 7/100
22/22 - 0s - loss: 0.4673 - accuracy: 0.7685 - val_loss: 0.4603 - val_accuracy: 0.7532 - 65ms/epoch - 3ms/step
Epoch 8/100
22/22 - 0s - loss: 0.4610 - accuracy: 0.7583 - val_loss: 0.4536 - val_accuracy: 0.8052 - 62ms/epoch - 3ms/step
Epoch 9/100
22/22 - 0s - loss: 0.4483 - accuracy: 0.7916 - val_loss: 0.4471 - val_accuracy: 0.8182 - 61ms/epoch - 3ms/step
Epoch 10/100
22/22 - 0s - loss: 0.4490 - accuracy: 0.7873 - val_loss: 0.4499 - val_accuracy: 0.7792 - 69ms/epoch - 3ms/step
Epoch 11/100
22/22 - 0s - loss: 0.4540 - accuracy: 0.7858 - val_loss: 0.4551 - val_accuracy: 0.8052 - 61ms/epoch - 3ms/step
Epoch 12/100
22/22 - 0s - loss: 0.4476 - accuracy: 0.7742 - val_loss: 0.4449 - val_accuracy: 0.8052 - 65ms/epoch - 3ms/step
Epoch 13/100
22/22 - 0s - loss: 0.4413 - accuracy: 0.7757 - val_loss: 0.4502 - val_accuracy: 0.7792 - 83ms/epoch - 4ms/step
Epoch 14/100

22/22 - 0s - loss: 0.4407 - accuracy: 0.7800 - val_loss: 0.4299 - val_accuracy: 0.7922 - 68ms/epoch - 3ms/step
Epoch 15/100
22/22 - 0s - loss: 0.4334 - accuracy: 0.7887 - val_loss: 0.4289 - val_accuracy: 0.7922 - 62ms/epoch - 3ms/step
Epoch 16/100
22/22 - 0s - loss: 0.4412 - accuracy: 0.7699 - val_loss: 0.4322 - val_accuracy: 0.7922 - 65ms/epoch - 3ms/step
Epoch 17/100
22/22 - 0s - loss: 0.4431 - accuracy: 0.7786 - val_loss: 0.4585 - val_accuracy: 0.7792 - 62ms/epoch - 3ms/step
Epoch 18/100
22/22 - 0s - loss: 0.4618 - accuracy: 0.7641 - val_loss: 0.4800 - val_accuracy: 0.7922 - 64ms/epoch - 3ms/step
Epoch 19/100
22/22 - 0s - loss: 0.4493 - accuracy: 0.7916 - val_loss: 0.4411 - val_accuracy: 0.8052 - 65ms/epoch - 3ms/step
Epoch 20/100
22/22 - 0s - loss: 0.4516 - accuracy: 0.7525 - val_loss: 0.4449 - val_accuracy: 0.7922 - 64ms/epoch - 3ms/step
Epoch 21/100
22/22 - 0s - loss: 0.4372 - accuracy: 0.7902 - val_loss: 0.4448 - val_accuracy: 0.7922 - 64ms/epoch - 3ms/step
Epoch 22/100
22/22 - 0s - loss: 0.4290 - accuracy: 0.7728 - val_loss: 0.4285 - val_accuracy: 0.7792 - 64ms/epoch - 3ms/step
Epoch 23/100
22/22 - 0s - loss: 0.4286 - accuracy: 0.7931 - val_loss: 0.4385 - val_accuracy: 0.8052 - 64ms/epoch - 3ms/step
Epoch 24/100
22/22 - 0s - loss: 0.4287 - accuracy: 0.7844 - val_loss: 0.4209 - val_accuracy: 0.8052 - 67ms/epoch - 3ms/step
Epoch 25/100
22/22 - 0s - loss: 0.4223 - accuracy: 0.7858 - val_loss: 0.4188 - val_accuracy: 0.8052 - 64ms/epoch - 3ms/step
Epoch 26/100
22/22 - 0s - loss: 0.4522 - accuracy: 0.7771 - val_loss: 0.4305 - val_accuracy: 0.7922 - 64ms/epoch - 3ms/step
Epoch 27/100
22/22 - 0s - loss: 0.4355 - accuracy: 0.7829 - val_loss: 0.4265 - val_accuracy: 0.7922 - 64ms/epoch - 3ms/step
Epoch 28/100
22/22 - 0s - loss: 0.4264 - accuracy: 0.7873 - val_loss: 0.4814 - val_accuracy: 0.7792 - 62ms/epoch - 3ms/step
Epoch 29/100
22/22 - 0s - loss: 0.4521 - accuracy: 0.7786 - val_loss: 0.4356 - val_accuracy: 0.8052 - 66ms/epoch - 3ms/step
Epoch 30/100
22/22 - 0s - loss: 0.4327 - accuracy: 0.7945 - val_loss: 0.4394 - val_accuracy: 0.8052 - 62ms/epoch - 3ms/step
Epoch 31/100
22/22 - 0s - loss: 0.4374 - accuracy: 0.7945 - val_loss: 0.5114 - val_accuracy: 0.7532 - 67ms/epoch - 3ms/step
Epoch 32/100
22/22 - 0s - loss: 0.4326 - accuracy: 0.7771 - val_loss: 0.4366 - val_accuracy: 0.8312 - 64ms/epoc

h - 3ms/step
Epoch 33/100
22/22 - 0s - loss: 0.4269 - accuracy: 0.7873 - val_loss: 0.4248 - val_accuracy: 0.8052 - 65ms/epoc
h - 3ms/step
Epoch 34/100
22/22 - 0s - loss: 0.4176 - accuracy: 0.8032 - val_loss: 0.4430 - val_accuracy: 0.8052 - 68ms/epoc
h - 3ms/step
Epoch 35/100
22/22 - 0s - loss: 0.4321 - accuracy: 0.7959 - val_loss: 0.4261 - val_accuracy: 0.8182 - 72ms/epoc
h - 3ms/step
Epoch 36/100
22/22 - 0s - loss: 0.4238 - accuracy: 0.8148 - val_loss: 0.4500 - val_accuracy: 0.7922 - 64ms/epoc
h - 3ms/step
Epoch 37/100
22/22 - 0s - loss: 0.4175 - accuracy: 0.8017 - val_loss: 0.4260 - val_accuracy: 0.8052 - 62ms/epoc
h - 3ms/step
Epoch 38/100
22/22 - 0s - loss: 0.4257 - accuracy: 0.7829 - val_loss: 0.4324 - val_accuracy: 0.8182 - 61ms/epoc
h - 3ms/step
Epoch 39/100
22/22 - 0s - loss: 0.4142 - accuracy: 0.7988 - val_loss: 0.4253 - val_accuracy: 0.8312 - 64ms/epoc
h - 3ms/step
Epoch 40/100
22/22 - 0s - loss: 0.4101 - accuracy: 0.7945 - val_loss: 0.4211 - val_accuracy: 0.8052 - 70ms/epoc
h - 3ms/step
Epoch 41/100
22/22 - 0s - loss: 0.4037 - accuracy: 0.8032 - val_loss: 0.4497 - val_accuracy: 0.8182 - 65ms/epoc
h - 3ms/step
Epoch 42/100
22/22 - 0s - loss: 0.4176 - accuracy: 0.7873 - val_loss: 0.4501 - val_accuracy: 0.7922 - 68ms/epoc
h - 3ms/step
Epoch 43/100
22/22 - 0s - loss: 0.4153 - accuracy: 0.7988 - val_loss: 0.4473 - val_accuracy: 0.8052 - 68ms/epoc
h - 3ms/step
Epoch 44/100
22/22 - 0s - loss: 0.4156 - accuracy: 0.7916 - val_loss: 0.4377 - val_accuracy: 0.8052 - 97ms/epoc
h - 4ms/step
Epoch 45/100
22/22 - 0s - loss: 0.4047 - accuracy: 0.7988 - val_loss: 0.4455 - val_accuracy: 0.8052 - 69ms/epoc
h - 3ms/step
Epoch 46/100
22/22 - 0s - loss: 0.4079 - accuracy: 0.8148 - val_loss: 0.4505 - val_accuracy: 0.7922 - 62ms/epoc
h - 3ms/step
Epoch 47/100
22/22 - 0s - loss: 0.4009 - accuracy: 0.8090 - val_loss: 0.4256 - val_accuracy: 0.8182 - 62ms/epoc
h - 3ms/step
Epoch 48/100
22/22 - 0s - loss: 0.4232 - accuracy: 0.7988 - val_loss: 0.4182 - val_accuracy: 0.8312 - 63ms/epoc
h - 3ms/step
Epoch 49/100
22/22 - 0s - loss: 0.4163 - accuracy: 0.7945 - val_loss: 0.4418 - val_accuracy: 0.8052 - 64ms/epoc
h - 3ms/step
Epoch 50/100
22/22 - 0s - loss: 0.4089 - accuracy: 0.7959 - val_loss: 0.4289 - val_accuracy: 0.8312 - 69ms/epoc
h - 3ms/step

Epoch 51/100
22/22 - 0s - loss: 0.4170 - accuracy: 0.7902 - val_loss: 0.4500 - val_accuracy: 0.7922 - 68ms/epoch - 3ms/step
Epoch 52/100
22/22 - 0s - loss: 0.4176 - accuracy: 0.7959 - val_loss: 0.4310 - val_accuracy: 0.8312 - 62ms/epoch - 3ms/step
Epoch 53/100
22/22 - 0s - loss: 0.4124 - accuracy: 0.8075 - val_loss: 0.4600 - val_accuracy: 0.8052 - 63ms/epoch - 3ms/step
Epoch 54/100
22/22 - 0s - loss: 0.4070 - accuracy: 0.7931 - val_loss: 0.4368 - val_accuracy: 0.7662 - 63ms/epoch - 3ms/step
Epoch 55/100
22/22 - 0s - loss: 0.4035 - accuracy: 0.8119 - val_loss: 0.4210 - val_accuracy: 0.8052 - 66ms/epoch - 3ms/step
Epoch 56/100
22/22 - 0s - loss: 0.4013 - accuracy: 0.8234 - val_loss: 0.4764 - val_accuracy: 0.7403 - 65ms/epoch - 3ms/step
Epoch 57/100
22/22 - 0s - loss: 0.4018 - accuracy: 0.8061 - val_loss: 0.4680 - val_accuracy: 0.7662 - 62ms/epoch - 3ms/step
Epoch 58/100
22/22 - 0s - loss: 0.3890 - accuracy: 0.8133 - val_loss: 0.4386 - val_accuracy: 0.7792 - 65ms/epoch - 3ms/step
Epoch 59/100
22/22 - 0s - loss: 0.3959 - accuracy: 0.8075 - val_loss: 0.4524 - val_accuracy: 0.8052 - 80ms/epoch - 4ms/step
Epoch 60/100
22/22 - 0s - loss: 0.3962 - accuracy: 0.8017 - val_loss: 0.4347 - val_accuracy: 0.8052 - 79ms/epoch - 4ms/step
Epoch 61/100
22/22 - 0s - loss: 0.3975 - accuracy: 0.8119 - val_loss: 0.4352 - val_accuracy: 0.7792 - 60ms/epoch - 3ms/step
Epoch 62/100
22/22 - 0s - loss: 0.3917 - accuracy: 0.8162 - val_loss: 0.4404 - val_accuracy: 0.8052 - 59ms/epoch - 3ms/step
Epoch 63/100
22/22 - 0s - loss: 0.3849 - accuracy: 0.8205 - val_loss: 0.4519 - val_accuracy: 0.7662 - 60ms/epoch - 3ms/step
Epoch 64/100
22/22 - 0s - loss: 0.3928 - accuracy: 0.8148 - val_loss: 0.4437 - val_accuracy: 0.7792 - 61ms/epoch - 3ms/step
Epoch 65/100
22/22 - 0s - loss: 0.3998 - accuracy: 0.8017 - val_loss: 0.4396 - val_accuracy: 0.7792 - 61ms/epoch - 3ms/step
Epoch 66/100
22/22 - 0s - loss: 0.3889 - accuracy: 0.8090 - val_loss: 0.4352 - val_accuracy: 0.7792 - 63ms/epoch - 3ms/step
Epoch 67/100
22/22 - 0s - loss: 0.3882 - accuracy: 0.8133 - val_loss: 0.4433 - val_accuracy: 0.7922 - 81ms/epoch - 4ms/step
Epoch 68/100
22/22 - 0s - loss: 0.3973 - accuracy: 0.8061 - val_loss: 0.4595 - val_accuracy: 0.7662 - 90ms/epoch - 4ms/step
Epoch 69/100

```
22/22 - 0s - loss: 0.3884 - accuracy: 0.8133 - val_loss: 0.4450 - val_accuracy: 0.7792 - 70ms/epoc
h - 3ms/step
Epoch 70/100
22/22 - 0s - loss: 0.3799 - accuracy: 0.8321 - val_loss: 0.4478 - val_accuracy: 0.8052 - 69ms/epoc
h - 3ms/step
Epoch 71/100
22/22 - 0s - loss: 0.3861 - accuracy: 0.8205 - val_loss: 0.4628 - val_accuracy: 0.8052 - 90ms/epoc
h - 4ms/step
Epoch 72/100
22/22 - 0s - loss: 0.3845 - accuracy: 0.8249 - val_loss: 0.4432 - val_accuracy: 0.7922 - 66ms/epoc
h - 3ms/step
Epoch 73/100
22/22 - 0s - loss: 0.3771 - accuracy: 0.8148 - val_loss: 0.4528 - val_accuracy: 0.8052 - 66ms/epoc
h - 3ms/step
Epoch 74/100
22/22 - 0s - loss: 0.3884 - accuracy: 0.8090 - val_loss: 0.4774 - val_accuracy: 0.7662 - 69ms/epoc
h - 3ms/step
Epoch 75/100
22/22 - 0s - loss: 0.3823 - accuracy: 0.8162 - val_loss: 0.4886 - val_accuracy: 0.7662 - 78ms/epoc
h - 4ms/step
Epoch 76/100
22/22 - 0s - loss: 0.3838 - accuracy: 0.8191 - val_loss: 0.4517 - val_accuracy: 0.7922 - 64ms/epoc
h - 3ms/step
Epoch 77/100
22/22 - 0s - loss: 0.3719 - accuracy: 0.8177 - val_loss: 0.5102 - val_accuracy: 0.7532 - 63ms/epoc
h - 3ms/step
Epoch 78/100
22/22 - 0s - loss: 0.4043 - accuracy: 0.8046 - val_loss: 0.4823 - val_accuracy: 0.7792 - 64ms/epoc
h - 3ms/step
Epoch 79/100
22/22 - 0s - loss: 0.4168 - accuracy: 0.8133 - val_loss: 0.4494 - val_accuracy: 0.8052 - 60ms/epoc
h - 3ms/step
Epoch 80/100
22/22 - 0s - loss: 0.3931 - accuracy: 0.8119 - val_loss: 0.4615 - val_accuracy: 0.7792 - 60ms/epoc
h - 3ms/step
Epoch 81/100
22/22 - 0s - loss: 0.3887 - accuracy: 0.8133 - val_loss: 0.4589 - val_accuracy: 0.7922 - 60ms/epoc
h - 3ms/step
Epoch 82/100
22/22 - 0s - loss: 0.3805 - accuracy: 0.8119 - val_loss: 0.4396 - val_accuracy: 0.7922 - 62ms/epoc
h - 3ms/step
Epoch 83/100
22/22 - 0s - loss: 0.3755 - accuracy: 0.8148 - val_loss: 0.4807 - val_accuracy: 0.7662 - 77ms/epoc
h - 4ms/step
Epoch 84/100
22/22 - 0s - loss: 0.3689 - accuracy: 0.8249 - val_loss: 0.4583 - val_accuracy: 0.7792 - 62ms/epoc
h - 3ms/step
Epoch 85/100
22/22 - 0s - loss: 0.3702 - accuracy: 0.8263 - val_loss: 0.4683 - val_accuracy: 0.8052 - 65ms/epoc
h - 3ms/step
Epoch 86/100
22/22 - 0s - loss: 0.3755 - accuracy: 0.8205 - val_loss: 0.4736 - val_accuracy: 0.7662 - 69ms/epoc
h - 3ms/step
Epoch 87/100
22/22 - 0s - loss: 0.3748 - accuracy: 0.8336 - val_loss: 0.4927 - val_accuracy: 0.8182 - 68ms/epoc
```

h - 3ms/step
Epoch 88/100
22/22 - 0s - loss: 0.3731 - accuracy: 0.8177 - val_loss: 0.4717 - val_accuracy: 0.8312 - 64ms/epoch - 3ms/step
Epoch 89/100
22/22 - 0s - loss: 0.3602 - accuracy: 0.8307 - val_loss: 0.4803 - val_accuracy: 0.7792 - 62ms/epoch - 3ms/step
Epoch 90/100
22/22 - 0s - loss: 0.3574 - accuracy: 0.8307 - val_loss: 0.5009 - val_accuracy: 0.7792 - 67ms/epoch - 3ms/step
Epoch 91/100
22/22 - 0s - loss: 0.3739 - accuracy: 0.8177 - val_loss: 0.4583 - val_accuracy: 0.8182 - 63ms/epoch - 3ms/step
Epoch 92/100
22/22 - 0s - loss: 0.3636 - accuracy: 0.8336 - val_loss: 0.5459 - val_accuracy: 0.7403 - 64ms/epoch - 3ms/step
Epoch 93/100
22/22 - 0s - loss: 0.3696 - accuracy: 0.8148 - val_loss: 0.4642 - val_accuracy: 0.8182 - 61ms/epoch - 3ms/step
Epoch 94/100
22/22 - 0s - loss: 0.3653 - accuracy: 0.8249 - val_loss: 0.4678 - val_accuracy: 0.7662 - 63ms/epoch - 3ms/step
Epoch 95/100
22/22 - 0s - loss: 0.3457 - accuracy: 0.8350 - val_loss: 0.4760 - val_accuracy: 0.7792 - 63ms/epoch - 3ms/step
Epoch 96/100
22/22 - 0s - loss: 0.3546 - accuracy: 0.8365 - val_loss: 0.4873 - val_accuracy: 0.7662 - 66ms/epoch - 3ms/step
Epoch 97/100
22/22 - 0s - loss: 0.3529 - accuracy: 0.8234 - val_loss: 0.5068 - val_accuracy: 0.7403 - 63ms/epoch - 3ms/step
Epoch 98/100
22/22 - 0s - loss: 0.3578 - accuracy: 0.8394 - val_loss: 0.4809 - val_accuracy: 0.8052 - 71ms/epoch - 3ms/step
Epoch 99/100
22/22 - 0s - loss: 0.3585 - accuracy: 0.8205 - val_loss: 0.4883 - val_accuracy: 0.7532 - 70ms/epoch - 3ms/step
Epoch 100/100
22/22 - 0s - loss: 0.3602 - accuracy: 0.8307 - val_loss: 0.4822 - val_accuracy: 0.7922 - 74ms/epoch - 3ms/step

In [23]:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.head()
```

Out[23]:

	loss	accuracy	val_loss	val_accuracy	epoch
0	0.634584	0.670043	0.555559	0.766234	0

	loss	accuracy	val_loss	val_accuracy	epoch
1	0.532897	0.723589	0.481906	0.792208	1
2	0.494692	0.759768	0.498138	0.727273	2
3	0.508134	0.738061	0.456181	0.805195	3
4	0.460164	0.777135	0.590974	0.714286	4

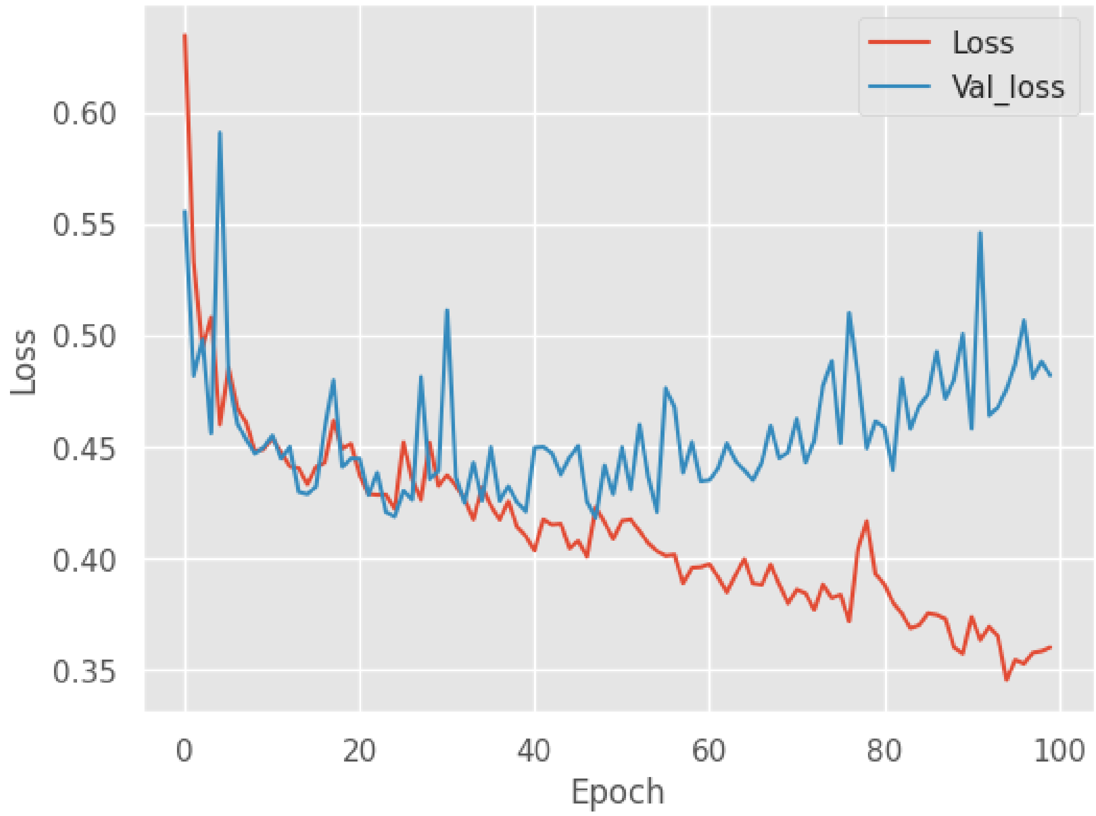
In [24]:

```
y_run = hist[['loss', 'val_loss']]
x_run = hist['epoch']
```

- Need to perform some hyperparameter tuning.

In [25]:

```
plt.plot(x_run,y_run)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Loss', 'Val_loss'], loc='upper right')
plt.show()
```



4. Model Building - 2 with SMOTE

- As we can see from previous model build, it does a decent job but not a good job.
- We can go for some other strategy to tackle the problem of imbalanced dataset.
- **Undersampling** is a technique where we take the same amount of data points from both classes. So if we have 100 data points out of which 20 are of the minority class, we take 20 data points from the majority class and discard rest of the data. Ofcourse this method has its drawback with the quantity of data.
- **Oversampling** has two types. In the first type, you just duplicate the minority class data to match the majority class data's numbers. The other type is called **SMOTE** (Synthetic Minority Oversampling Technique) where we use KNN to generate synthetic data similar to the minority class.
- **Ensemble** methods are another way to tackle this problem.

4.1 Oversampling using SMOTE

In [26]:

```
#Importing the necessary dependencies.
from imblearn.over_sampling import SMOTE
smote = SMOTE(sampling_strategy='minority')
```

In [27]:

```
#Showing the class imbalance.
y.value_counts()
```

Out[27]:

```
Outcome
0    500
```

```
1    268
Name: count, dtype: int64
```

- As we can see, we have a balanced dataset.

In [28]:
```
#Oversampling.
X_sm, y_sm = smote.fit_resample(X,y)
y_sm.value_counts()
```

Out[28]:
```
Outcome
1    500
0    500
Name: count, dtype: int64
```

4.2 Splitting the data

In [29]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_sm, y_sm, test_size = 0.1, stratify=y_sm, random_stat
e=42)
```

4.3 Training the ANN

In [30]:
```
model2 = model_constructor()
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 64)                576

 dense_4 (Dense)             (None, 32)                2080

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 2,689
Trainable params: 2,689
Non-trainable params: 0
_________________________________________________________________
```

In [31]:
```
history2 = model2.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, verbose=2, batch_si
ze=32)
```

```
Epoch 1/100
29/29 - 1s - loss: 0.6224 - accuracy: 0.6633 - val_loss: 0.5213 - val_accuracy: 0.7600 - 987ms/epo
ch - 34ms/step
Epoch 2/100
29/29 - 0s - loss: 0.5294 - accuracy: 0.7311 - val_loss: 0.5461 - val_accuracy: 0.7500 - 88ms/epoc
h - 3ms/step
Epoch 3/100
29/29 - 0s - loss: 0.5065 - accuracy: 0.7478 - val_loss: 0.4923 - val_accuracy: 0.7600 - 78ms/epoc
h - 3ms/step
Epoch 4/100
```

29/29 - 0s - loss: 0.4930 - accuracy: 0.7522 - val_loss: 0.4910 - val_accuracy: 0.7700 - 75ms/epoch - 3ms/step
Epoch 5/100
29/29 - 0s - loss: 0.4906 - accuracy: 0.7522 - val_loss: 0.5345 - val_accuracy: 0.7600 - 74ms/epoch - 3ms/step
Epoch 6/100
29/29 - 0s - loss: 0.4891 - accuracy: 0.7711 - val_loss: 0.4894 - val_accuracy: 0.7600 - 74ms/epoch - 3ms/step
Epoch 7/100
29/29 - 0s - loss: 0.4706 - accuracy: 0.7589 - val_loss: 0.4797 - val_accuracy: 0.7600 - 72ms/epoch - 2ms/step
Epoch 8/100
29/29 - 0s - loss: 0.4687 - accuracy: 0.7689 - val_loss: 0.4853 - val_accuracy: 0.7400 - 87ms/epoch - 3ms/step
Epoch 9/100
29/29 - 0s - loss: 0.4697 - accuracy: 0.7689 - val_loss: 0.5778 - val_accuracy: 0.7000 - 73ms/epoch - 3ms/step
Epoch 10/100
29/29 - 0s - loss: 0.4867 - accuracy: 0.7511 - val_loss: 0.4729 - val_accuracy: 0.7700 - 75ms/epoch - 3ms/step
Epoch 11/100
29/29 - 0s - loss: 0.4549 - accuracy: 0.7678 - val_loss: 0.4997 - val_accuracy: 0.7200 - 74ms/epoch - 3ms/step
Epoch 12/100
29/29 - 0s - loss: 0.4698 - accuracy: 0.7722 - val_loss: 0.4592 - val_accuracy: 0.7800 - 77ms/epoch - 3ms/step
Epoch 13/100
29/29 - 0s - loss: 0.4546 - accuracy: 0.7811 - val_loss: 0.4772 - val_accuracy: 0.7400 - 78ms/epoch - 3ms/step
Epoch 14/100
29/29 - 0s - loss: 0.4537 - accuracy: 0.7833 - val_loss: 0.4832 - val_accuracy: 0.6900 - 80ms/epoch - 3ms/step
Epoch 15/100
29/29 - 0s - loss: 0.4564 - accuracy: 0.7833 - val_loss: 0.4721 - val_accuracy: 0.7000 - 76ms/epoch - 3ms/step
Epoch 16/100
29/29 - 0s - loss: 0.4473 - accuracy: 0.7944 - val_loss: 0.4483 - val_accuracy: 0.7900 - 75ms/epoch - 3ms/step
Epoch 17/100
29/29 - 0s - loss: 0.4368 - accuracy: 0.7811 - val_loss: 0.4541 - val_accuracy: 0.7700 - 86ms/epoch - 3ms/step
Epoch 18/100
29/29 - 0s - loss: 0.4535 - accuracy: 0.7900 - val_loss: 0.4634 - val_accuracy: 0.7500 - 74ms/epoch - 3ms/step
Epoch 19/100
29/29 - 0s - loss: 0.4598 - accuracy: 0.7756 - val_loss: 0.4953 - val_accuracy: 0.7600 - 72ms/epoch - 2ms/step
Epoch 20/100
29/29 - 0s - loss: 0.4749 - accuracy: 0.7767 - val_loss: 0.4735 - val_accuracy: 0.7700 - 76ms/epoch - 3ms/step
Epoch 21/100
29/29 - 0s - loss: 0.4476 - accuracy: 0.7811 - val_loss: 0.4741 - val_accuracy: 0.7300 - 73ms/epoch - 3ms/step
Epoch 22/100
29/29 - 0s - loss: 0.4486 - accuracy: 0.7856 - val_loss: 0.4568 - val_accuracy: 0.7700 - 73ms/epoc

h - 3ms/step
Epoch 23/100
29/29 - 0s - loss: 0.4332 - accuracy: 0.7978 - val_loss: 0.4530 - val_accuracy: 0.7800 - 74ms/epoc
h - 3ms/step
Epoch 24/100
29/29 - 0s - loss: 0.4458 - accuracy: 0.7900 - val_loss: 0.4827 - val_accuracy: 0.7400 - 73ms/epoc
h - 3ms/step
Epoch 25/100
29/29 - 0s - loss: 0.4307 - accuracy: 0.7944 - val_loss: 0.4510 - val_accuracy: 0.7600 - 75ms/epoc
h - 3ms/step
Epoch 26/100
29/29 - 0s - loss: 0.4352 - accuracy: 0.7978 - val_loss: 0.4747 - val_accuracy: 0.7700 - 77ms/epoc
h - 3ms/step
Epoch 27/100
29/29 - 0s - loss: 0.4560 - accuracy: 0.7889 - val_loss: 0.4876 - val_accuracy: 0.7100 - 73ms/epoc
h - 3ms/step
Epoch 28/100
29/29 - 0s - loss: 0.4371 - accuracy: 0.7933 - val_loss: 0.4469 - val_accuracy: 0.8200 - 73ms/epoc
h - 3ms/step
Epoch 29/100
29/29 - 0s - loss: 0.4427 - accuracy: 0.7889 - val_loss: 0.4807 - val_accuracy: 0.7600 - 72ms/epoc
h - 2ms/step
Epoch 30/100
29/29 - 0s - loss: 0.4463 - accuracy: 0.7867 - val_loss: 0.4520 - val_accuracy: 0.7900 - 73ms/epoc
h - 3ms/step
Epoch 31/100
29/29 - 0s - loss: 0.4426 - accuracy: 0.7922 - val_loss: 0.4752 - val_accuracy: 0.7700 - 73ms/epoc
h - 3ms/step
Epoch 32/100
29/29 - 0s - loss: 0.4213 - accuracy: 0.7944 - val_loss: 0.4572 - val_accuracy: 0.7700 - 70ms/epoc
h - 2ms/step
Epoch 33/100
29/29 - 0s - loss: 0.4289 - accuracy: 0.8011 - val_loss: 0.4774 - val_accuracy: 0.7400 - 70ms/epoc
h - 2ms/step
Epoch 34/100
29/29 - 0s - loss: 0.4266 - accuracy: 0.7944 - val_loss: 0.4504 - val_accuracy: 0.7700 - 71ms/epoc
h - 2ms/step
Epoch 35/100
29/29 - 0s - loss: 0.4335 - accuracy: 0.7922 - val_loss: 0.5434 - val_accuracy: 0.7100 - 68ms/epoc
h - 2ms/step
Epoch 36/100
29/29 - 0s - loss: 0.4401 - accuracy: 0.7967 - val_loss: 0.4459 - val_accuracy: 0.7600 - 74ms/epoc
h - 3ms/step
Epoch 37/100
29/29 - 0s - loss: 0.4264 - accuracy: 0.8000 - val_loss: 0.4316 - val_accuracy: 0.8000 - 73ms/epoc
h - 3ms/step
Epoch 38/100
29/29 - 0s - loss: 0.4247 - accuracy: 0.8000 - val_loss: 0.4424 - val_accuracy: 0.7800 - 77ms/epoc
h - 3ms/step
Epoch 39/100
29/29 - 0s - loss: 0.4180 - accuracy: 0.8133 - val_loss: 0.4431 - val_accuracy: 0.7600 - 77ms/epoc
h - 3ms/step
Epoch 40/100
29/29 - 0s - loss: 0.4197 - accuracy: 0.8067 - val_loss: 0.4348 - val_accuracy: 0.7800 - 91ms/epoc
h - 3ms/step

Epoch 41/100
29/29 - 0s - loss: 0.4217 - accuracy: 0.8022 - val_loss: 0.4606 - val_accuracy: 0.7800 - 79ms/epoch - 3ms/step
Epoch 42/100
29/29 - 0s - loss: 0.4468 - accuracy: 0.7856 - val_loss: 0.4455 - val_accuracy: 0.8000 - 75ms/epoch - 3ms/step
Epoch 43/100
29/29 - 0s - loss: 0.4184 - accuracy: 0.8011 - val_loss: 0.4912 - val_accuracy: 0.7700 - 68ms/epoch - 2ms/step
Epoch 44/100
29/29 - 0s - loss: 0.4358 - accuracy: 0.8000 - val_loss: 0.4615 - val_accuracy: 0.7300 - 74ms/epoch - 3ms/step
Epoch 45/100
29/29 - 0s - loss: 0.4277 - accuracy: 0.7856 - val_loss: 0.4943 - val_accuracy: 0.7600 - 75ms/epoch - 3ms/step
Epoch 46/100
29/29 - 0s - loss: 0.4280 - accuracy: 0.8000 - val_loss: 0.4736 - val_accuracy: 0.7600 - 77ms/epoch - 3ms/step
Epoch 47/100
29/29 - 0s - loss: 0.4186 - accuracy: 0.8078 - val_loss: 0.4762 - val_accuracy: 0.7600 - 86ms/epoch - 3ms/step
Epoch 48/100
29/29 - 0s - loss: 0.4315 - accuracy: 0.7978 - val_loss: 0.4580 - val_accuracy: 0.7600 - 73ms/epoch - 3ms/step
Epoch 49/100
29/29 - 0s - loss: 0.4145 - accuracy: 0.8056 - val_loss: 0.4808 - val_accuracy: 0.7700 - 76ms/epoch - 3ms/step
Epoch 50/100
29/29 - 0s - loss: 0.4642 - accuracy: 0.7889 - val_loss: 0.4419 - val_accuracy: 0.7600 - 76ms/epoch - 3ms/step
Epoch 51/100
29/29 - 0s - loss: 0.4165 - accuracy: 0.8100 - val_loss: 0.4619 - val_accuracy: 0.7600 - 74ms/epoch - 3ms/step
Epoch 52/100
29/29 - 0s - loss: 0.4126 - accuracy: 0.7989 - val_loss: 0.4307 - val_accuracy: 0.7800 - 80ms/epoch - 3ms/step
Epoch 53/100
29/29 - 0s - loss: 0.4180 - accuracy: 0.8022 - val_loss: 0.4341 - val_accuracy: 0.7800 - 75ms/epoch - 3ms/step
Epoch 54/100
29/29 - 0s - loss: 0.4122 - accuracy: 0.8111 - val_loss: 0.4432 - val_accuracy: 0.7600 - 74ms/epoch - 3ms/step
Epoch 55/100
29/29 - 0s - loss: 0.4127 - accuracy: 0.8033 - val_loss: 0.4275 - val_accuracy: 0.8000 - 75ms/epoch - 3ms/step
Epoch 56/100
29/29 - 0s - loss: 0.4205 - accuracy: 0.8111 - val_loss: 0.4229 - val_accuracy: 0.8100 - 74ms/epoch - 3ms/step
Epoch 57/100
29/29 - 0s - loss: 0.4196 - accuracy: 0.7989 - val_loss: 0.4115 - val_accuracy: 0.7900 - 71ms/epoch - 2ms/step
Epoch 58/100
29/29 - 0s - loss: 0.4060 - accuracy: 0.8178 - val_loss: 0.4450 - val_accuracy: 0.7600 - 71ms/epoch - 2ms/step
Epoch 59/100

29/29 - 0s - loss: 0.4011 - accuracy: 0.8089 - val_loss: 0.4650 - val_accuracy: 0.7800 - 75ms/epoch - 3ms/step
Epoch 60/100
29/29 - 0s - loss: 0.4118 - accuracy: 0.8178 - val_loss: 0.4217 - val_accuracy: 0.7800 - 71ms/epoch - 2ms/step
Epoch 61/100
29/29 - 0s - loss: 0.4051 - accuracy: 0.8200 - val_loss: 0.4516 - val_accuracy: 0.7400 - 83ms/epoch - 3ms/step
Epoch 62/100
29/29 - 0s - loss: 0.4137 - accuracy: 0.8067 - val_loss: 0.4607 - val_accuracy: 0.7500 - 85ms/epoch - 3ms/step
Epoch 63/100
29/29 - 0s - loss: 0.4004 - accuracy: 0.8133 - val_loss: 0.4457 - val_accuracy: 0.7800 - 78ms/epoch - 3ms/step
Epoch 64/100
29/29 - 0s - loss: 0.4027 - accuracy: 0.8144 - val_loss: 0.4703 - val_accuracy: 0.7800 - 80ms/epoch - 3ms/step
Epoch 65/100
29/29 - 0s - loss: 0.3968 - accuracy: 0.8278 - val_loss: 0.4215 - val_accuracy: 0.8100 - 77ms/epoch - 3ms/step
Epoch 66/100
29/29 - 0s - loss: 0.3970 - accuracy: 0.8222 - val_loss: 0.4546 - val_accuracy: 0.8000 - 77ms/epoch - 3ms/step
Epoch 67/100
29/29 - 0s - loss: 0.3986 - accuracy: 0.8144 - val_loss: 0.4303 - val_accuracy: 0.7900 - 75ms/epoch - 3ms/step
Epoch 68/100
29/29 - 0s - loss: 0.3934 - accuracy: 0.8156 - val_loss: 0.4568 - val_accuracy: 0.8000 - 75ms/epoch - 3ms/step
Epoch 69/100
29/29 - 0s - loss: 0.4068 - accuracy: 0.8222 - val_loss: 0.4571 - val_accuracy: 0.7800 - 79ms/epoch - 3ms/step
Epoch 70/100
29/29 - 0s - loss: 0.3925 - accuracy: 0.8244 - val_loss: 0.4105 - val_accuracy: 0.7900 - 77ms/epoch - 3ms/step
Epoch 71/100
29/29 - 0s - loss: 0.3992 - accuracy: 0.8300 - val_loss: 0.4477 - val_accuracy: 0.7500 - 80ms/epoch - 3ms/step
Epoch 72/100
29/29 - 0s - loss: 0.3894 - accuracy: 0.8244 - val_loss: 0.4154 - val_accuracy: 0.8000 - 80ms/epoch - 3ms/step
Epoch 73/100
29/29 - 0s - loss: 0.4261 - accuracy: 0.8067 - val_loss: 0.4413 - val_accuracy: 0.7600 - 73ms/epoch - 3ms/step
Epoch 74/100
29/29 - 0s - loss: 0.3819 - accuracy: 0.8244 - val_loss: 0.4509 - val_accuracy: 0.8000 - 94ms/epoch - 3ms/step
Epoch 75/100
29/29 - 0s - loss: 0.3829 - accuracy: 0.8411 - val_loss: 0.4691 - val_accuracy: 0.7900 - 79ms/epoch - 3ms/step
Epoch 76/100
29/29 - 0s - loss: 0.4015 - accuracy: 0.8233 - val_loss: 0.4079 - val_accuracy: 0.7900 - 84ms/epoch - 3ms/step
Epoch 77/100
29/29 - 0s - loss: 0.3809 - accuracy: 0.8444 - val_loss: 0.4086 - val_accuracy: 0.8000 - 77ms/epoc

h - 3ms/step
Epoch 78/100
29/29 - 0s - loss: 0.3793 - accuracy: 0.8300 - val_loss: 0.4246 - val_accuracy: 0.7800 - 77ms/epoch - 3ms/step
Epoch 79/100
29/29 - 0s - loss: 0.3810 - accuracy: 0.8267 - val_loss: 0.4737 - val_accuracy: 0.7700 - 79ms/epoch - 3ms/step
Epoch 80/100
29/29 - 0s - loss: 0.3982 - accuracy: 0.8222 - val_loss: 0.4539 - val_accuracy: 0.7700 - 79ms/epoch - 3ms/step
Epoch 81/100
29/29 - 0s - loss: 0.3817 - accuracy: 0.8389 - val_loss: 0.4061 - val_accuracy: 0.8000 - 77ms/epoch - 3ms/step
Epoch 82/100
29/29 - 0s - loss: 0.3858 - accuracy: 0.8233 - val_loss: 0.4056 - val_accuracy: 0.8000 - 73ms/epoch - 3ms/step
Epoch 83/100
29/29 - 0s - loss: 0.3818 - accuracy: 0.8311 - val_loss: 0.4079 - val_accuracy: 0.8100 - 75ms/epoch - 3ms/step
Epoch 84/100
29/29 - 0s - loss: 0.3717 - accuracy: 0.8333 - val_loss: 0.4534 - val_accuracy: 0.8000 - 74ms/epoch - 3ms/step
Epoch 85/100
29/29 - 0s - loss: 0.3663 - accuracy: 0.8411 - val_loss: 0.4096 - val_accuracy: 0.8000 - 76ms/epoch - 3ms/step
Epoch 86/100
29/29 - 0s - loss: 0.3802 - accuracy: 0.8256 - val_loss: 0.4235 - val_accuracy: 0.7900 - 77ms/epoch - 3ms/step
Epoch 87/100
29/29 - 0s - loss: 0.3783 - accuracy: 0.8389 - val_loss: 0.4260 - val_accuracy: 0.8200 - 76ms/epoch - 3ms/step
Epoch 88/100
29/29 - 0s - loss: 0.3704 - accuracy: 0.8422 - val_loss: 0.4141 - val_accuracy: 0.8100 - 89ms/epoch - 3ms/step
Epoch 89/100
29/29 - 0s - loss: 0.3762 - accuracy: 0.8289 - val_loss: 0.4070 - val_accuracy: 0.8000 - 77ms/epoch - 3ms/step
Epoch 90/100
29/29 - 0s - loss: 0.3719 - accuracy: 0.8400 - val_loss: 0.4648 - val_accuracy: 0.8000 - 78ms/epoch - 3ms/step
Epoch 91/100
29/29 - 0s - loss: 0.3729 - accuracy: 0.8344 - val_loss: 0.4119 - val_accuracy: 0.8000 - 71ms/epoch - 2ms/step
Epoch 92/100
29/29 - 0s - loss: 0.3665 - accuracy: 0.8322 - val_loss: 0.4635 - val_accuracy: 0.7700 - 70ms/epoch - 2ms/step
Epoch 93/100
29/29 - 0s - loss: 0.3871 - accuracy: 0.8111 - val_loss: 0.4364 - val_accuracy: 0.7700 - 71ms/epoch - 2ms/step
Epoch 94/100
29/29 - 0s - loss: 0.3735 - accuracy: 0.8333 - val_loss: 0.4553 - val_accuracy: 0.8000 - 74ms/epoch - 3ms/step
Epoch 95/100
29/29 - 0s - loss: 0.3681 - accuracy: 0.8367 - val_loss: 0.4094 - val_accuracy: 0.7900 - 77ms/epoch - 3ms/step

```
Epoch 96/100
29/29 - 0s - loss: 0.3634 - accuracy: 0.8311 - val_loss: 0.4134 - val_accuracy: 0.8100 - 73ms/epoc
h - 3ms/step
Epoch 97/100
29/29 - 0s - loss: 0.3677 - accuracy: 0.8400 - val_loss: 0.4047 - val_accuracy: 0.7800 - 72ms/epoc
h - 2ms/step
Epoch 98/100
29/29 - 0s - loss: 0.3630 - accuracy: 0.8344 - val_loss: 0.5941 - val_accuracy: 0.7900 - 71ms/epoc
h - 2ms/step
Epoch 99/100
29/29 - 0s - loss: 0.3810 - accuracy: 0.8222 - val_loss: 0.4947 - val_accuracy: 0.7700 - 70ms/epoc
h - 2ms/step
Epoch 100/100
29/29 - 0s - loss: 0.3634 - accuracy: 0.8356 - val_loss: 0.4648 - val_accuracy: 0.7800 - 71ms/epoc
h - 2ms/step
```

In [32]:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.head()
```

Out[32]:

	loss	accuracy	val_loss	val_accuracy	epoch
0	0.634584	0.670043	0.555559	0.766234	0
1	0.532897	0.723589	0.481906	0.792208	1
2	0.494692	0.759768	0.498138	0.727273	2
3	0.508134	0.738061	0.456181	0.805195	3
4	0.460164	0.777135	0.590974	0.714286	4

In [33]:

```
y_run = hist[['loss', 'val_loss']]
x_run = hist['epoch']
```

- This has a better loss curve than the previous model.

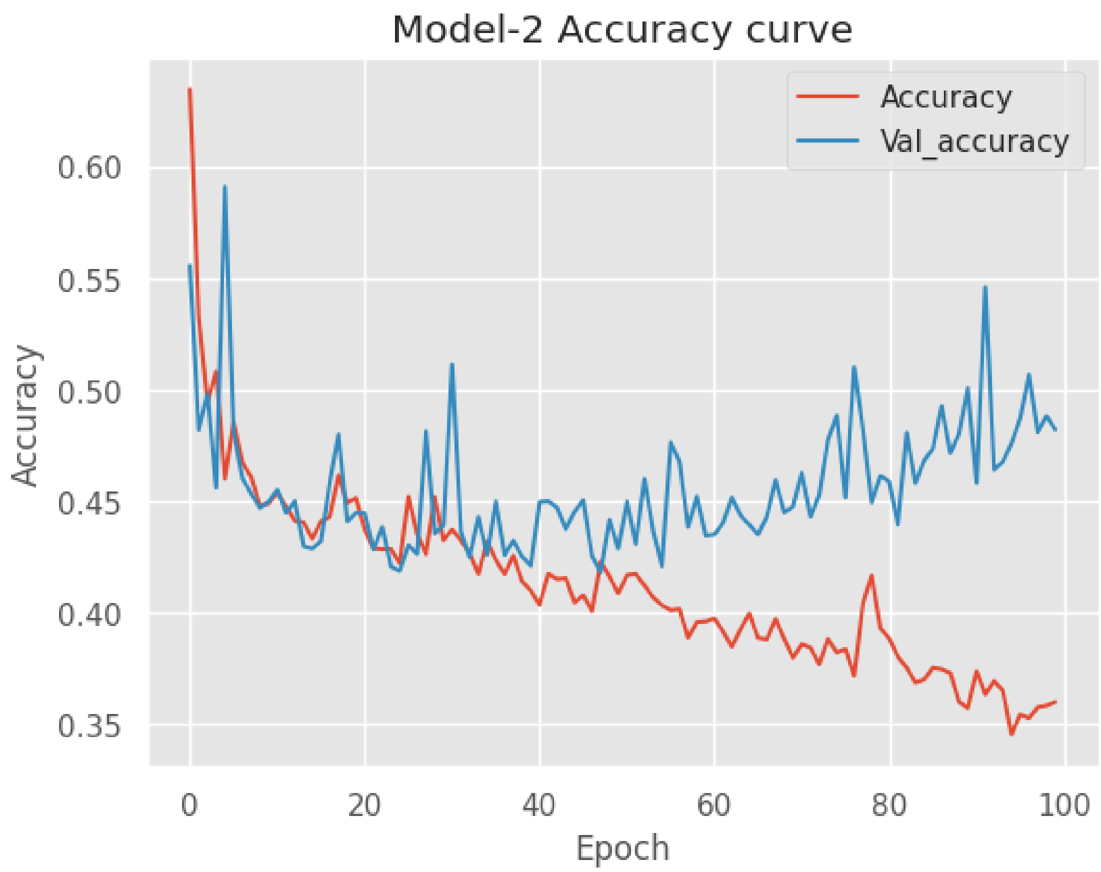
In [34]:

```
plt.plot(x_run,y_run)
plt.title('Model-2 Loss Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.legend(['Loss', 'Val_loss'], loc='upper right')
plt.show()
```

In [35]:
linkcode

```
y_acc = hist[['accuracy', 'val_accuracy']]
x_epoch = hist['epoch']
plt.plot(x_run,y_run)
plt.title('Model-2 Accuracy curve')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Accuracy', 'Val_accuracy'], loc='upper right')
plt.show()
```



CONCLUSION :

In the phase 2 conclusion,we will summarize the ensemble methods and deep learning architechure techniques for diabetes prediction.