In [1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Activation,Dense
```

```
C:\Users\Anusha V\anaconda3\lib\site-packages\scipy\__init__.py:155: UserW
arning: A NumPy version >=1.18.5 and <1.25.0 is required for this version
of SciPy (detected version 1.26.1
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

In [28]:
```

data=pd.read_csv(r"C:\Users\Anusha V\Downloads\heart1.csv")
```

In [29]:
```
data

```

Out[29]:

| | age | sex | cp | trestbps | chol | fbs | restecg | thalach | exang | oldpeak | slope | ca | thal | t |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 52 | 1 | 0 | 125 | 212 | 0 | 1 | 168 | 0 | 1.0 | 2 | 2 | 3 | |
| **1** | 53 | 1 | 0 | 140 | 203 | 1 | 0 | 155 | 1 | 3.1 | 0 | 0 | 3 | |
| **2** | 70 | 1 | 0 | 145 | 174 | 0 | 1 | 125 | 1 | 2.6 | 0 | 0 | 3 | |
| **3** | 61 | 1 | 0 | 148 | 203 | 0 | 1 | 161 | 0 | 0.0 | 2 | 1 | 3 | |
| **4** | 62 | 0 | 0 | 138 | 294 | 1 | 1 | 106 | 0 | 1.9 | 1 | 3 | 2 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **1020** | 59 | 1 | 1 | 140 | 221 | 0 | 1 | 164 | 1 | 0.0 | 2 | 0 | 2 | |
| **1021** | 60 | 1 | 0 | 125 | 258 | 0 | 0 | 141 | 1 | 2.8 | 1 | 1 | 3 | |
| **1022** | 47 | 1 | 0 | 110 | 275 | 0 | 0 | 118 | 1 | 1.0 | 1 | 1 | 2 | |
| **1023** | 50 | 0 | 0 | 110 | 254 | 0 | 0 | 159 | 0 | 0.0 | 2 | 0 | 2 | |
| **1024** | 54 | 1 | 0 | 120 | 188 | 0 | 1 | 113 | 0 | 1.4 | 1 | 1 | 3 | |

1025 rows × 14 columns

In [30]:
```
X = data.drop(columns=['target'])
y = data['target']

```

In [31]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2

```

In [32]:
```
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(X_train.shape[1],))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

```

In [ ]:
```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['a


model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=


```

```
Epoch 1/20
21/21 [==============================] - 2s 21ms/step - loss: 4.2801 - acc
uracy: 0.4649 - val_loss: 1.1501 - val_accuracy: 0.4390
Epoch 2/20
21/21 [==============================] - 0s 11ms/step - loss: 1.1572 - acc
uracy: 0.5091 - val_loss: 0.8599 - val_accuracy: 0.5183
Epoch 3/20
21/21 [==============================] - 0s 9ms/step - loss: 0.8155 - accu
racy: 0.6098 - val_loss: 0.8361 - val_accuracy: 0.5488
Epoch 4/20
21/21 [==============================] - 0s 9ms/step - loss: 0.7127 - accu
racy: 0.6463 - val_loss: 0.8186 - val_accuracy: 0.6037
Epoch 5/20
21/21 [==============================] - 0s 9ms/step - loss: 0.6476 - accu
racy: 0.6616 - val_loss: 0.8170 - val_accuracy: 0.5976
Epoch 6/20
21/21 [==============================] - 0s 9ms/step - loss: 0.6019 - accu
racy: 0.6905 - val_loss: 0.7779 - val_accuracy: 0.6280
Epoch 7/20
21/21 [==============================] - 0s 8ms/step - loss: 0.5743 - accu
racy: 0.7180 - val_loss: 0.7930 - val_accuracy: 0.6037
Epoch 8/20
21/21 [==============================] - 0s 10ms/step - loss: 0.5606 - acc
uracy: 0.7287 - val_loss: 0.7741 - val_accuracy: 0.6159
Epoch 9/20
21/21 [==============================] - 0s 8ms/step - loss: 0.5435 - accu
racy: 0.7363 - val_loss: 0.7611 - val_accuracy: 0.6220
Epoch 10/20
21/21 [==============================] - 0s 11ms/step - loss: 0.5296 - acc
uracy: 0.7409 - val_loss: 0.7394 - val_accuracy: 0.6280
Epoch 11/20
21/21 [==============================] - 0s 8ms/step - loss: 0.5383 - accu
racy: 0.7317 - val_loss: 0.7465 - val_accuracy: 0.6463
Epoch 12/20
21/21 [==============================] - 0s 8ms/step - loss: 0.5158 - accu
racy: 0.7332 - val_loss: 0.6861 - val_accuracy: 0.6402
Epoch 13/20
21/21 [==============================] - 0s 10ms/step - loss: 0.5326 - acc
uracy: 0.7363 - val_loss: 0.7073 - val_accuracy: 0.6524
Epoch 14/20
21/21 [==============================] - 0s 9ms/step - loss: 0.4983 - accu
racy: 0.7622 - val_loss: 0.6782 - val_accuracy: 0.6280
Epoch 15/20
21/21 [==============================] - 0s 9ms/step - loss: 0.4793 - accu
racy: 0.7744 - val_loss: 0.6642 - val_accuracy: 0.6402
Epoch 16/20
21/21 [==============================] - 0s 8ms/step - loss: 0.4748 - accu
racy: 0.7759 - val_loss: 0.6361 - val_accuracy: 0.6524
Epoch 17/20
13/21 [=================>............] - ETA: 0s - loss: 0.4944 - accurac
y: 0.7500
```

In [ ]:
```
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

In [ ]:
```

```

In [ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE

```

In [ ]:
```
# Assuming your dataset is in a CSV file
df = pd.read_csv(r"C:\Users\Anusha V\Desktop\diabetes.csv")

```

In [ ]:
```
# Check for missing values
print(df.isnull().sum())

# Check data types
print(df.dtypes)

```

In [ ]:
```
data.head(2)
```

In [ ]:
```
X = data.drop(columns=['count'])
y = data['count']
```

In [24]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2

```

In [25]:
```
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(X_train.shape[1],))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

```

In [26]:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['a


model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=


```

```
Epoch 1/20
5/5 [==============================] - 3s 182ms/step - loss: -0.4637 - acc
uracy: 0.1375 - val_loss: -5.4125 - val_accuracy: 0.1000
Epoch 2/20
5/5 [==============================] - 0s 63ms/step - loss: -1.1090 - accu
racy: 0.1375 - val_loss: -7.5187 - val_accuracy: 0.1000
Epoch 3/20
5/5 [==============================] - 0s 33ms/step - loss: -1.6182 - accu
racy: 0.1375 - val_loss: -9.8189 - val_accuracy: 0.1000
Epoch 4/20
5/5 [==============================] - 0s 30ms/step - loss: -2.2691 - accu
racy: 0.1375 - val_loss: -12.1385 - val_accuracy: 0.1000
Epoch 5/20
5/5 [==============================] - 0s 34ms/step - loss: -2.9196 - accu
racy: 0.1375 - val_loss: -14.5659 - val_accuracy: 0.1000
Epoch 6/20
5/5 [==============================] - 0s 34ms/step - loss: -3.5364 - accu
racy: 0.1375 - val_loss: -17.2265 - val_accuracy: 0.1000
Epoch 7/20
5/5 [==============================] - 0s 27ms/step - loss: -4.2658 - accu
racy: 0.1375 - val_loss: -19.9719 - val_accuracy: 0.1000
Epoch 8/20
5/5 [==============================] - 0s 24ms/step - loss: -5.0590 - accu
racy: 0.1375 - val_loss: -22.8575 - val_accuracy: 0.1000
Epoch 9/20
5/5 [==============================] - 0s 19ms/step - loss: -5.8365 - accu
racy: 0.1375 - val_loss: -25.9913 - val_accuracy: 0.1000
Epoch 10/20
5/5 [==============================] - 0s 23ms/step - loss: -6.7541 - accu
racy: 0.1375 - val_loss: -29.3319 - val_accuracy: 0.1000
Epoch 11/20
5/5 [==============================] - 0s 19ms/step - loss: -7.7242 - accu
racy: 0.1375 - val_loss: -33.0192 - val_accuracy: 0.1000
Epoch 12/20
5/5 [==============================] - 0s 37ms/step - loss: -8.6385 - accu
racy: 0.1375 - val_loss: -37.1766 - val_accuracy: 0.1000
Epoch 13/20
5/5 [==============================] - 0s 33ms/step - loss: -9.8316 - accu
racy: 0.1437 - val_loss: -41.3374 - val_accuracy: 0.1250
Epoch 14/20
5/5 [==============================] - 0s 20ms/step - loss: -10.9624 - acc
uracy: 0.1500 - val_loss: -45.8264 - val_accuracy: 0.1250
Epoch 15/20
5/5 [==============================] - 0s 20ms/step - loss: -12.1072 - acc
uracy: 0.1500 - val_loss: -50.7634 - val_accuracy: 0.1250
Epoch 16/20
5/5 [==============================] - 0s 27ms/step - loss: -13.5395 - acc
uracy: 0.1625 - val_loss: -55.8706 - val_accuracy: 0.1250
Epoch 17/20
5/5 [==============================] - 0s 24ms/step - loss: -14.8398 - acc
uracy: 0.1688 - val_loss: -61.6153 - val_accuracy: 0.1250
Epoch 18/20
5/5 [==============================] - 0s 29ms/step - loss: -16.1310 - acc
uracy: 0.1688 - val_loss: -68.0390 - val_accuracy: 0.1250
Epoch 19/20
5/5 [==============================] - 0s 23ms/step - loss: -17.9924 - acc
uracy: 0.1688 - val_loss: -74.4900 - val_accuracy: 0.1250
Epoch 20/20
5/5 [==============================] - 0s 25ms/step - loss: -19.7415 - acc
uracy: 0.1688 - val_loss: -81.4427 - val_accuracy: 0.1250
```

Out[26]: <keras.src.callbacks.History at 0x1fd40accfa0>

In [27]:
```
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

```
2/2 [==============================] - 0s 18ms/step - loss: -29.0556 - acc
uracy: 0.1600
Test accuracy: 0.1599999964237213
```

# Through a step-by-step process calculate TF/IDF for the given corpus and mention the words having highest value

Doc1: we are going to Mysore Doc2: Mysore is a famous place Doc3: we are going to famous place

In [2]:

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Define the corpus
corpus = [
    "we are going to Mysore",
    "Mysore is a famous place",
    "we are going to famous place"
]

#Create a TF-IDF Vectorizer
tfidf_vectorizer = TfidfVectorizer()

# Fit and transform the corpus
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)

#  Get feature names (words)
feature_names = tfidf_vectorizer.get_feature_names_out()

#Create a dictionary to store the TF-IDF values for each word in each d
tfidf_values = {}

#Loop through each document and each word to get the TF-IDF value
for doc_index, doc in enumerate(corpus):
    feature_index = tfidf_matrix[doc_index, :].nonzero()[1]
    tfidf_doc = zip(feature_index, [tfidf_matrix[doc_index, x] for x in

    for word_index, tfidf in tfidf_doc:
        word = feature_names[word_index]
        if word not in tfidf_values:
            tfidf_values[word] = [(doc_index, tfidf)]
        else:
            tfidf_values[word].append((doc_index, tfidf))

#Find the words with the highest TF-IDF values
highest_tfidf_words = {}
for word, values in tfidf_values.items():
    highest_tfidf_words[word] = max(values, key=lambda x: x[1])

# Print the words with the highest TF-IDF values
for word, (doc_index, tfidf) in highest_tfidf_words.items():
    print(f"Word: {word}, Document: {doc_index + 1}, TF-IDF Value: {tfi

```

```
C:\Users\Anusha V\anaconda3\lib\site-packages\scipy\__init__.py:155: UserW
arning: A NumPy version >=1.18.5 and <1.25.0 is required for this version
of SciPy (detected version 1.26.2
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"

Word: mysore, Document: 2, TF-IDF Value: 0.4598535287588349
Word: to, Document: 1, TF-IDF Value: 0.4472135954999579
Word: going, Document: 1, TF-IDF Value: 0.4472135954999579
Word: are, Document: 1, TF-IDF Value: 0.4472135954999579
Word: we, Document: 1, TF-IDF Value: 0.4472135954999579
Word: place, Document: 2, TF-IDF Value: 0.4598535287588349
Word: famous, Document: 2, TF-IDF Value: 0.4598535287588349
Word: is, Document: 2, TF-IDF Value: 0.6046521283053111
```

# Create a complete end to end Neural network for MNIST(classify handwritten numerals)

In [3]:

```python
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_d

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0

# One-hot encode the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# Build the neural network model
model = models.Sequential()
model.add(layers.Flatten(input_shape=(28, 28)))  # Flatten the 28x28 im
model.add(layers.Dense(128, activation='relu'))  # Hidden layer with 12
model.add(layers.Dropout(0.2))  # Dropout layer to reduce overfitting
model.add(layers.Dense(10, activation='softmax'))  # Output layer with

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

# Train the model
model.fit(train_images, train_labels, epochs=5, batch_size=64, validati

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test Accuracy: {test_acc}")

# Make predictions on a few test images
predictions = model.predict(test_images[:5])
predicted_labels = tf.argmax(predictions, axis=1)

# Print the predicted labels
print("Predicted Labels:", predicted_labels.numpy())

```

```
Epoch 1/5
938/938 [==============================] - 16s 14ms/step - loss: 0.3399 -
accuracy: 0.9026 - val_loss: 0.1606 - val_accuracy: 0.9515
Epoch 2/5
938/938 [==============================] - 10s 10ms/step - loss: 0.1634 -
accuracy: 0.9514 - val_loss: 0.1130 - val_accuracy: 0.9658
Epoch 3/5
938/938 [==============================] - 7s 7ms/step - loss: 0.1201 - ac
curacy: 0.9644 - val_loss: 0.0899 - val_accuracy: 0.9725
Epoch 4/5
938/938 [==============================] - 8s 8ms/step - loss: 0.0981 - ac
curacy: 0.9704 - val_loss: 0.0831 - val_accuracy: 0.9747
Epoch 5/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0827 - ac
curacy: 0.9750 - val_loss: 0.0765 - val_accuracy: 0.9772
313/313 [==============================] - 1s 4ms/step - loss: 0.0765 - ac
curacy: 0.9772
Test Accuracy: 0.9771999716758728
1/1 [==============================] - 0s 288ms/step
Predicted Labels: [7 2 1 0 4]
```

# N-grams are defined as the combinations of N Keywords together consider the given

“The greatest glory in Living lies not in never falling but in raising every Lies” Generate bi grams for the above Generate Tri grams for the above

In [4]:

```python
from nltk import ngrams
from nltk.tokenize import word_tokenize

# Given text
text = "The greatest glory in Living lies not in never falling but in r

# Tokenize the text into words
words = word_tokenize(text)

# Function to generate n-grams
def generate_ngrams(tokens, n):
    n_grams = ngrams(tokens, n)
    return [' '.join(gram) for gram in n_grams]

# Generate bi-grams
bi_grams = generate_ngrams(words, 2)
print("Bi-grams:", bi_grams)

# Generate tri-grams
tri_grams = generate_ngrams(words, 3)
print("Tri-grams:", tri_grams)

```

```
Bi-grams: ['The greatest', 'greatest glory', 'glory in', 'in Living', 'Liv
ing lies', 'lies not', 'not in', 'in never', 'never falling', 'falling bu
t', 'but in', 'in raising', 'raising every', 'every Lies']
Tri-grams: ['The greatest glory', 'greatest glory in', 'glory in Living',
'in Living lies', 'Living lies not', 'lies not in', 'not in never', 'in ne
ver falling', 'never falling but', 'falling but in', 'but in raising', 'in
raising every', 'raising every Lies']
```

In [ ]:

```
```