Srushti Ramesh Bhamre BC56

The IMDB sentiment classification dataset consists of 50,000 movie reviews from IMDB users that are labeled as either positive (1) or negative (0). The reviews are preprocessed and each one is encoded as a sequence of word indexes in the form of integers. The words within the reviews are indexed by their overall frequency within the dataset. For example, the integer "2" encodes the second most frequent word in the data. The 50,000 reviews are split into 25,000 for training and 25,000 for testing. Text Process word by word at diffrent timestamp (You may use RNN(Recurrent Neural Network) LSTM(Long Short-Term Memory) GRU(Gated Recurrent Unit)) convert input text to vector reprent input text DOMAIN: Digital content and entertainment industry CONTEXT: The objective of this project is to build a text classification model that analyses the customer's sentiments based on their reviews in the IMDB database. The model uses a complex deep learning model to build an embedding layer followed by a classification algorithm to analyse the sentiment of the customers. DATA DESCRIPTION: The Dataset of 50,000 movie reviews from IMDB, labelled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a sequence of word indexes (integers). For convenience, the words are indexed by their frequency in the dataset, meaning the for that has index 1 is the most frequent word. Use the first 20 words from each review to speed up training, using a max vocabulary size of 10,000. As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word. PROJECT OBJECTIVE: Build a sequential NLP classifier which can use input text parameters to determine the customer sentiments.

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
#loading imdb data with most frequent 10000 words

from keras.datasets import imdb
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000) # you may take top 10,000 word frequently review of movies
#consolidating data for EDA(exploratory data analysis: involves gathering all the relevant data into one place and preparing it for analysis )
data = np.concatenate((X_train, X_test), axis=0)
label = np.concatenate((y_train, y_test), axis=0)

```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz (https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz)
17464789/17464789 [==============================] - 10s 1us/step
```

In [3]:
```
print("Review is ",X_train[5]) # series of no converted word to vocabulory associated with index
print("Review is ",y_train[5]) # 0 indicating a negative review and 1 indicating a positive review.
```

```
Review is  [1, 778, 128, 74, 12, 630, 163, 15, 4, 1766, 7982, 1051, 2, 32, 85, 156, 45, 40, 148, 139, 121, 664, 665, 10, 10, 1361, 173, 4, 749, 2, 16, 3804, 8, 4, 226, 65, 12, 43, 127, 24, 2, 10, 10]
Review is  0
```

In [4]:
```
vocab=imdb.get_word_index() #The code you provided retrieves the word index for the IMDB dataset
print(vocab)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json (https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json)
1641221/1641221 [==============================] - 1s 1us/step
{'fawn': 34701, 'tsukino': 52006, 'nunnery': 52007, 'sonja': 16816, 'vani': 63951, 'woods': 1408, 'spiders': 16115, 'hanging': 2345, 'woody': 2289, 'trawling': 52008, "hold's": 52009, 'comically': 11307, 'localized': 40830, 'disobeying': 30568, "'royale": 52010, "harpo's": 40831, 'canet': 52011, 'aileen': 19313, 'acurately': 52012, "diplomat's": 52013, 'rickman': 25242, 'arranged': 6746, 'rumbustious': 52014, 'familiarness': 52015, "spider'": 52016, 'hahahah': 68804, "wood'": 52017, 'transvestism': 40833, "hangin'": 34702, 'bringing': 2338, 'seamier': 40834, 'wooded': 34703, 'bravora': 52018, 'grueling': 16817, 'wooden': 1636, 'wednesday': 16818, "'prix": 52019, 'altagracia': 34704, 'circuitry': 52020, 'crotch': 11585, 'busybody': 57766, "tart'n'tangy": 52021, 'burgade': 14129, 'thrace': 52023, "tom's": 11038, 'snuggles': 52025, 'francesco': 29114, 'complainers': 52027, 'templarios': 52125, '272': 40835, '273': 52028, 'zaniacs': 52130, '275': 34706, 'consenting': 27631, 'snuggled': 40836, 'inanimate': 15492, 'uality': 52030, 'bronte': 11926, 'errors': 4010, 'dialogs': 3230, "yomada's": 52031, "madman's": 34707, 'dialoge': 30585, 'usenet': 52033, 'videodrome': 40837, "kid'": 26338, 'pawed': 52034, "'girlfriend'": 30569, "'pleasure": 52035, "'reloaded'": 52036, "kazakos'": 40839, 'rocque': 52037, 'mailings': 52038, 'brainwashed': 11927, 'mcanally': 16819, "tom''": 52039, 'kurupt': 25243, 'affiliated': 21905, 'babaganoosh': 52040, "noe's": 40840, 'quart': 40841, 'kids': 359, 'uplifting': 5034, 'controversy': 7093, 'kida': 21906, 'kidd': 23379, "error'": 52041, 'neurologist': 52042, 'spotty': 18510, 'cobblers': 30570, 'projection': 9878, 'fastforwarding': 40842, 'sters': 52043, "eggar's": 52044, 'etherything': 52045, 'gateshead': 40843, 'airball': 34708, 'unsinkable': 25244, 'stern': 7180, "cervi's": 52046, 'dnd': 40844, 'dna': 11586, 'insecurity': 20598, "'reboot'": 52047, 'trelkovsky': 11037, 'jaekel': 52048, 'sidebars': 52049, "sforza's": 52050, 'distortions': 17633, 'mutinies': 52051, 'sermons': 30602, '7ft': 40846, 'boobage': 52052, "o'bannon's": 52053, 'populations': 23380, 'chulak': 52054, 'mesmerize': 27633, 'quinnell': 52055, 'yahoo': 10307, 'meteorologist': 52057, 'beswick': 42577, 'boorman': 15493, 'voicework': 40847, "ster'": 52058, 'blustering': 22922, 'hj': 52059, 'intake': 27634, 'morally': 5621, 'jumbling': 408
```

In [5]:
```
data #data is a numpy array that contains all the text data from the IMDB dataset, both the training and testing sets.
```

Out[5]:
```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284,
5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 2
2, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 6
6, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 595
2, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 1
94, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16,
4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]),
       list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954, 189, 102, 5, 207, 110,
3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163, 4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 490
1, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4, 1649, 26, 6853, 5, 163, 11, 3215, 2, 4, 1153, 9, 194, 775, 7, 8255,
2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853, 15, 349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175,
136, 50, 9, 4373, 228, 8255, 5, 2, 656, 245, 2350, 5, 4, 9837, 131, 152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 1
45, 23, 4, 1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16, 145, 95]),
       list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4, 2401, 311, 12, 16, 3711, 33, 75, 43, 1829, 29
6, 4, 86, 320, 35, 534, 19, 263, 4821, 1301, 4, 1873, 33, 89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11, 4, 1716, 43, 645, 662, 8, 257, 85, 1200,
42, 1228, 2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36, 69, 2, 780, 8, 106, 14, 6905, 1338, 18, 6, 22, 12, 215, 28, 610, 40, 6, 87, 326, 23, 2300, 2
1, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170, 23, 595, 116, 595, 1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534,
6, 227, 7, 129, 113]),
       ...,
       list([1, 13, 1408, 15, 8, 135, 14, 9, 35, 32, 46, 394, 20, 62, 30, 5093, 21, 45, 184, 78, 4, 1492, 910, 769, 2290, 2515, 395, 4257, 5, 1454, 11,
119, 2, 89, 1036, 4, 116, 218, 78, 21, 407, 100, 30, 128, 262, 15, 7, 185, 2280, 284, 1842, 2, 37, 315, 4, 226, 20, 272, 2942, 40, 29, 152, 60, 181, 8,
30, 50, 553, 362, 80, 119, 12, 21, 846, 5518]),
       list([1, 11, 119, 241, 9, 4, 840, 20, 12, 468, 15, 94, 3684, 562, 791, 39, 4, 86, 107, 8, 97, 14, 31, 33, 4, 2960, 7, 743, 46, 1028, 9, 3531, 5,
4, 768, 47, 8, 79, 90, 145, 164, 162, 50, 6, 501, 119, 7, 9, 4, 78, 232, 15, 16, 224, 11, 4, 333, 20, 4, 985, 200, 5, 2, 5, 9, 1861, 8, 79, 357, 4, 20,
47, 220, 57, 206, 139, 11, 12, 5, 55, 117, 212, 13, 1276, 92, 124, 51, 45, 1188, 71, 536, 13, 520, 14, 20, 6, 2302, 7, 470]),
       list([1, 6, 52, 7465, 430, 22, 9, 220, 2594, 8, 28, 2, 519, 3227, 6, 769, 15, 47, 6, 3482, 4067, 8, 114, 5, 33, 222, 31, 55, 184, 704, 5586, 2,
19, 346, 3153, 5, 6, 364, 350, 4, 184, 5586, 9, 133, 1810, 11, 5417, 2, 21, 4, 7298, 2, 570, 50, 2005, 2643, 9, 6, 1249, 17, 6, 2, 2, 21, 17, 6, 1211,
232, 1138, 2249, 29, 266, 56, 96, 346, 194, 308, 9, 194, 21, 29, 218, 1078, 19, 4, 78, 173, 7, 27, 2, 5698, 3406, 718, 2, 9, 6, 6907, 17, 210, 5, 3281,
5677, 47, 77, 395, 14, 172, 173, 18, 2740, 2931, 4517, 82, 127, 27, 173, 11, 6, 392, 217, 21, 50, 9, 57, 65, 12, 2, 53, 40, 35, 390, 7, 11, 4, 3567, 7,
4, 314, 74, 6, 792, 22, 2, 19, 714, 727, 5205, 382, 4, 91, 6533, 439, 19, 14, 20, 9, 1441, 5805, 1118, 4, 756, 25, 124, 4, 31, 12, 16, 93, 804, 34, 200
5, 2643])],
      dtype=object)
```

In [6]:
```
label #Label is a numpy array that contains all the sentiment labels from the IMDB dataset, both the training and testing sets.0 indicates a negativ
```

Out[6]:
```
array([1, 0, 0, ..., 0, 0, 0], dtype=int64)
```

In [7]:
```
X_train.shape
```

Out[7]:
```
(25000,)
```

In [8]:
```
X_test.shape
```

Out[8]:
```
(25000,)
```

In [9]:
```
y_train
```

Out[9]:
```
array([1, 0, 0, ..., 0, 1, 0], dtype=int64)
```

In [10]:
```
y_test # y_test is 25000, which indicates that it contains 25000 sentiment labels, one for each review in the testing set.
```

Out[10]:
```
array([0, 1, 1, ..., 0, 0, 0], dtype=int64)
```

In [11]:
```
def vectorize(sequences, dimension = 10000):
  # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1
    return results
```

In [12]:
```
# Now we split our data into a training and a testing set.
# The training set will contain  reviews and the testing set

test_x = data[:10000]
test_y = label[:10000]
train_x = data[10000:]
train_y = label[10000:]
```

In [13]:
```
test_x
```

Out[13]:
```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284,
       5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 2
       2, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 6
       6, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 595
       2, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 1
       94, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16,
       4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]),
       list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954, 189, 102, 5, 207, 110,
       3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163, 4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 490
       1, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4, 1649, 26, 6853, 5, 163, 11, 3215, 2, 4, 1153, 9, 194, 775, 7, 8255,
       2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853, 15, 349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175,
       136, 50, 9, 4373, 228, 8255, 5, 2, 656, 245, 2350, 5, 4, 9837, 131, 152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 1
       45, 23, 4, 1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16, 145, 95]),
       list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4, 2401, 311, 12, 16, 3711, 33, 75, 43, 1829, 29
       6, 4, 86, 320, 35, 534, 19, 263, 4821, 1301, 4, 1873, 33, 89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11, 4, 1716, 43, 645, 662, 8, 257, 85, 1200,
       42, 1228, 2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36, 69, 2, 780, 8, 106, 14, 6905, 1338, 18, 6, 22, 12, 215, 28, 610, 40, 6, 87, 326, 23, 2300, 2
       1, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170, 23, 595, 116, 595, 1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534,
       6, 227, 7, 129, 113]),
       ...,
       list([1, 14, 9, 6, 66, 327, 5, 1047, 20, 15, 4, 436, 223, 70, 358, 45, 44, 107, 2515, 5, 6, 1132, 37, 26, 623, 245, 8, 412, 19, 294, 334, 18, 6,
       117, 137, 21, 4, 1389, 92, 391, 5, 36, 1090, 5, 140, 8, 169, 4, 223, 23, 68, 205, 4, 1132, 9, 773, 5621, 5, 59, 456, 56, 8, 41, 403, 580, 9, 4, 1155, 9
       12, 37, 694, 6, 176, 44, 113, 23, 4, 1004, 7, 4, 6567, 2694, 9, 4, 922, 5, 2, 912, 37, 5190, 183, 276, 148, 289, 295, 23, 35, 1154, 5, 12, 166, 18, 6,
       654, 5, 253, 1061, 58, 50, 26, 57, 318, 302, 7, 4, 6849, 728, 38, 12, 218, 954, 33, 32, 45, 4, 118, 662, 1626, 20, 15, 207, 110, 38, 230, 45, 6, 66, 5
       2, 20, 18, 2166]),
       list([1, 14, 20, 9, 43, 160, 856, 206, 509, 21, 12, 100, 28, 77, 38, 76, 128, 54, 4, 1865, 216, 46, 36, 66, 887, 49, 3822, 339, 294, 40, 1798,
       2, 37, 93, 15, 530, 206, 720, 11, 3567, 17, 36, 847, 56, 4, 890, 39, 4, 4565, 62, 28, 679, 4, 753, 206, 844, 83, 142, 318, 88, 4, 360, 7, 4, 22, 16, 26
       59, 727, 21, 1753, 128, 74, 25, 62, 535, 39, 14, 552, 7, 20, 95, 385, 4, 477, 136, 4, 123, 180, 4, 31, 75, 69, 77, 1064, 18, 21, 16, 40, 149, 142, 39,
       4, 6, 768, 11, 4, 2084, 36, 1258, 5261, 164, 1936, 5, 36, 521, 187, 6, 313, 269, 8, 516, 257, 85, 172, 154, 172, 154]),
       list([1, 51, 527, 487, 5, 116, 57, 1613, 51, 25, 191, 97, 6, 52, 20, 19, 6, 686, 109, 7660, 12, 16, 224, 11, 2, 19, 532, 807, 10, 10, 38, 14, 55
       4, 271, 23, 6, 1189, 8, 67, 27, 336, 4, 554, 1655, 304, 6, 1707, 5, 4, 1811, 47, 6, 483, 1274, 5, 1442, 1696, 2817, 38, 4, 554, 6141, 11, 6, 2144, 5, 6
       095, 95, 29, 1129, 187, 4247, 11, 4, 5152, 366, 29, 214, 6590, 10, 10, 315, 15, 58, 29, 1860, 6, 2123, 1453, 4, 86, 58, 29, 1860, 12, 125, 11, 4, 2144,
       4, 333, 58, 29, 166, 6, 2, 46, 7, 6, 9459, 5, 9866, 4, 2123, 107, 665, 7, 1214, 541, 2, 46, 7, 4, 2, 4539, 1578, 72, 7, 6498, 2, 4, 3942, 9997, 10, 10,
       82, 4, 554, 1068, 8, 1968, 6, 2, 19, 727, 1901, 10, 10, 3288])],
      dtype=object)
```

In [14]:
```
test_y
```

Out[14]:
```
array([1, 0, 0, ..., 1, 0, 0], dtype=int64)
```

In [15]:
```
train_x
```

Out[15]:
```
array([list([1, 13, 104, 14, 9, 31, 7, 4, 4343, 7, 4, 3776, 3394, 2, 495, 103, 141, 87, 2048, 17, 76, 2, 44, 164, 525, 13, 197, 14, 16, 338, 4, 177, 1
       6, 6118, 5253, 2, 2, 2, 21, 61, 1126, 2, 16, 15, 36, 4621, 19, 4, 2, 157, 5, 605, 46, 49, 7, 4, 297, 8, 276, 11, 4, 621, 837, 844, 10, 10, 25, 43, 92,
       81, 2282, 5, 95, 947, 19, 4, 297, 806, 21, 15, 9, 43, 355, 13, 119, 49, 3636, 6951, 43, 40, 4, 375, 415, 21, 2, 92, 947, 19, 4, 2282, 1771, 14, 5, 106,
       2, 1151, 48, 25, 181, 8, 67, 6, 530, 9089, 1253, 7, 4, 2]),
       list([1, 14, 20, 16, 835, 835, 835, 51, 6, 1703, 56, 51, 6, 387, 180, 32, 812, 57, 2327, 6, 394, 437, 7, 676, 5, 58, 62, 24, 386, 12, 8, 61, 530
       1, 912, 37, 80, 106, 233]),
       list([1, 86, 125, 13, 62, 40, 8, 213, 46, 15, 137, 13, 244, 24, 35, 2809, 4, 96, 4, 3100, 16, 2400, 80, 2384, 129, 1663, 4633, 4, 2, 115, 2085,
       15, 2, 2, 165, 495, 9123, 18, 199, 4, 2, 88, 36, 70, 79, 35, 1271, 5, 4, 4824, 18, 24, 116, 23, 14, 17, 160, 2, 301, 2, 2799, 16, 2085, 9508, 4129, 36,
       343, 3973, 17, 4, 2, 37, 47, 965, 602, 5, 60, 80, 2, 11, 4, 3100, 63, 9, 43, 379, 48, 4, 2619, 69, 1668, 90, 8, 2, 15, 96, 36, 62, 28, 839, 11, 294, 99
       54, 900, 36, 62, 30, 43, 2254, 18, 35, 1271, 2, 14, 506, 16, 115, 1803, 3383, 1204, 44, 4, 2, 34, 4, 2, 2, 1373, 7, 4, 1494, 525, 5, 60, 54, 1803, 34,
       4, 4824, 3383, 1204, 40, 54, 12, 645, 29, 100, 24, 1527, 48, 15, 218, 3793, 824, 13, 92, 124, 51, 9, 5, 6, 3275, 2408, 62, 28, 1840, 35, 2, 10, 10, 132
       4, 347, 12, 517, 125, 73, 19, 4, 2, 7, 112, 4, 4069, 7, 6, 506, 19, 2, 21, 4, 3100, 166, 14, 20, 5028, 1297]),
       ...,
       list([1, 13, 1408, 15, 8, 135, 14, 9, 35, 32, 46, 394, 20, 62, 30, 5093, 21, 45, 184, 78, 4, 1492, 910, 769, 2290, 2515, 395, 4257, 5, 1454, 11,
       119, 2, 89, 1036, 4, 116, 218, 78, 21, 407, 100, 30, 128, 262, 15, 7, 185, 2280, 284, 1842, 2, 37, 315, 4, 226, 20, 272, 2942, 40, 29, 152, 60, 181, 8,
       30, 50, 553, 362, 80, 119, 12, 21, 846, 5518]),
       list([1, 11, 119, 241, 9, 4, 840, 20, 12, 468, 15, 94, 3684, 562, 791, 39, 4, 86, 107, 8, 97, 14, 31, 33, 4, 2960, 7, 743, 46, 1028, 9, 3531, 5,
       4, 768, 47, 8, 79, 90, 145, 164, 162, 50, 6, 501, 119, 7, 9, 4, 78, 232, 15, 16, 224, 11, 4, 333, 20, 4, 985, 200, 5, 2, 5, 9, 1861, 8, 79, 357, 4, 20,
       47, 220, 57, 206, 139, 11, 12, 5, 55, 117, 212, 13, 1276, 92, 124, 51, 45, 1188, 71, 536, 13, 520, 14, 20, 6, 2302, 7, 470]),
       list([1, 6, 52, 7465, 430, 22, 9, 220, 2594, 8, 28, 2, 519, 3227, 6, 769, 15, 47, 6, 3482, 4067, 8, 114, 5, 33, 222, 31, 55, 184, 704, 5586, 2,
       19, 346, 3153, 5, 6, 364, 350, 4, 184, 5586, 9, 133, 1810, 11, 5417, 2, 21, 4, 7298, 2, 570, 50, 2005, 2643, 9, 6, 1249, 17, 6, 2, 2, 21, 17, 6, 1211,
       232, 1138, 2249, 29, 266, 56, 96, 346, 194, 308, 9, 194, 21, 29, 218, 1078, 19, 4, 78, 173, 7, 27, 2, 5698, 3406, 718, 2, 9, 6, 6907, 17, 210, 5, 3281,
       5677, 47, 77, 395, 14, 172, 173, 18, 2740, 2931, 4517, 82, 127, 27, 173, 11, 6, 392, 217, 21, 50, 9, 57, 65, 12, 2, 53, 40, 35, 390, 7, 11, 4, 3567, 7,
       4, 314, 74, 6, 792, 22, 2, 19, 714, 727, 5205, 382, 4, 91, 6533, 439, 19, 14, 20, 9, 1441, 5805, 1118, 4, 756, 25, 124, 4, 31, 12, 16, 93, 804, 34, 200
       5, 2643])],
      dtype=object)
```

In [16]:
```
train_y
```

Out[16]:
```
array([0, 0, 0, ..., 0, 0, 0], dtype=int64)
```

In [17]:
```
print("Categories:", np.unique(label))
print("Number of unique words:", len(np.unique(np.hstack(data))))
# The hstack() function is used to stack arrays in sequence horizontally (column wise).

#>>> import numpy as np
#>>> x = np.array((3,5,7))
#>>> y = np.array((5,7,9))
#>>> np.hstack((x,y))
# array([3, 5, 7, 5, 7, 9])


# You can see in the output above that the dataset is labeled into two categories, either 0 or 1,
# which represents the sentiment of the review.

# The whole dataset contains 9998 unique words and the average review length is 234 words, with a standard deviation of 173 words.
```

```
Categories: [0 1]
Number of unique words: 9998
```

In [18]:
```
length = [len(i) for i in data]
print("Average Review length:", np.mean(length))
print("Standard Deviation:", round(np.std(length)))

# The whole dataset contains 9998 unique words and the average review length is 234 words, with a standard deviation of 173 words.
```

```
Average Review length: 234.75892
Standard Deviation: 173
```

In [19]:
```
# If you look at the data you will realize it has been already pre-processed.
# All words have been mapped to integers and the integers represent the words sorted by their frequency.
# This is very common in text analysis to represent a dataset like this.
# So 4 represents the 4th most used word,
# 5 the 5th most used word and so on...
# The integer 1 is reserved for the start marker,
# the integer 2 for an unknown word and 0 for padding.

# Let's look at a single training example:

print("Label:", label[0])
```

```
Label: 1
```

In [20]:
```
print(data[0])

```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 1
72, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 1
2, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33,
4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256,
4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 1
8, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113,
103, 32, 15, 16, 5345, 19, 178, 32]
```

In [21]:
```
# Retrieves a dict mapping words to their index in the IMDB dataset.
index = imdb.get_word_index()
# If there is a possibility of multiple keys with the same value, you will need to specify the desired behaviour in this case to lookup more than 2
# ivd = dict((v, k) for k, v in d.items())
# If you want to peek at the reviews yourself and see what people have actually written, you can reverse the process too:
reverse_index = dict([(value, key) for (key, value) in index.items()])
decoded = " ".join( [reverse_index.get(i - 3, "#") for i in data[0]] ) #The purpose of subtracting 3 from i is to adjust the indice,# to indicate th
print(decoded)
```

```
# this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being
there robert # is an amazing actor and now the same being director # father came from the same scottish island as myself so i loved the fact there was
a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it wa
s released for # and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what th
ey say if you cry at a film it must have been good and this definitely was also # to the two little boy's that played the # of norman and paul they wer
e just brilliant children are often left out of the # list i think because the stars that play them all grown up are such a big profile for the whole f
ilm but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and
was someone's life after all that was shared with us all
```

In [22]:
```
index
```

Out[22]:
```
{'fawn': 34701,
 'tsukino': 52006,
 'nunnery': 52007,
 'sonja': 16816,
 'vani': 63951,
 'woods': 1408,
 'spiders': 16115,
 'hanging': 2345,
 'woody': 2289,
 'trawling': 52008,
 "hold's": 52009,
 'comically': 11307,
 'localized': 40830,
 'disobeying': 30568,
 "'royale": 52010,
 "harpo's": 40831,
 'canet': 52011,
 'aileen': 19313,
 'acurately': 52012,
```

In [23]:
```
reverse_index
```

Out[23]:
```
{34701: 'fawn',
 52006: 'tsukino',
 52007: 'nunnery',
 16816: 'sonja',
 63951: 'vani',
 1408: 'woods',
 16115: 'spiders',
 2345: 'hanging',
 2289: 'woody',
 52008: 'trawling',
 52009: "hold's",
 11307: 'comically',
 40830: 'localized',
 30568: 'disobeying',
 52010: "'royale",
 40831: "harpo's",
 52011: 'canet',
 19313: 'aileen',
 52012: 'acurately',
```

In [24]:
```
decoded
```

Out[24]:
```
"# this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert # is an amazing actor and now the same being director # father came from the same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for # and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been good and this definitely was also # to the two little boy's that played the # of norman and paul they were just brilliant children are often left out of the # list i think because the stars that play them all grown up are such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all that was shared with us all"
```

In [25]:
```
#Adding sequence to data
# Vectorization is the process of converting textual data into numerical vectors and is a process that is usually applied once the text is cleaned.
data = vectorize(data)
label = np.array(label).astype("float32")
```

In [26]:
```
data
```

Out[26]:
```
array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       ...,
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.]])
```

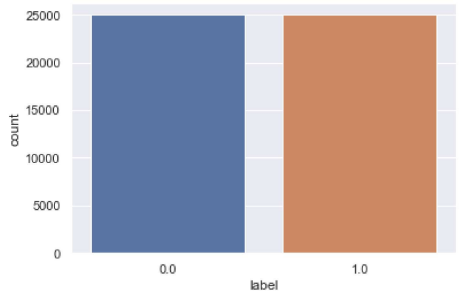
In [27]:
```
label
```

Out[27]:
```
array([1., 0., 0., ..., 0., 0., 0.], dtype=float32)
```

In [28]:
```
# Let's check distribution of data
# To create plots for EDA(exploratory data analysis)
import seaborn as sns #seaborn is a popular Python visualization library that is built on top of Matplotlib and provides a high-level interface for
sns.set(color_codes=True)
import matplotlib.pyplot as plt # %matplotlib to display Matplotlib plots inline with the notebook
%matplotlib inline
```

In [29]:
```
labelDF=pd.DataFrame({'label':label})
sns.countplot(x='label', data=labelDF)
# For below analysis it is clear that data has equel distribution of sentiments.This will help us building a good model.
```

Out[29]:
```
<AxesSubplot:xlabel='label', ylabel='count'>
```



In [30]:
```
# Creating train and test data set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data,label, test_size=0.20, random_state=1)
```

In [31]:
```
X_train.shape
```

Out[31]:
```
(40000, 10000)
```

In [32]:
```
X_test.shape
```

Out[32]:
```
(10000, 10000)
```

In [33]:
```
# Let's create sequential model,In deep learning, a Sequential model is a linear stack of layers, where you can simply add layers one after the other
from keras.utils import to_categorical
from keras import models
from keras import layers
```

In [34]:
```
model = models.Sequential()
# Input - Layer
# Note that we set the input-shape to 10,000 at the input-layer because our reviews are 10,000 integers long.
# The input-layer takes 10,000 as input and outputs it with a shape of 50.
model.add(layers.Dense(50, activation = "relu", input_shape=(10000, )))
# Hidden - Layers
# Please note you should always use a dropout rate between 20% and 50%. # here in our case 0.3 means 30% dropout we are using dropout to prevent over
# By the way, if you want you can build a sentiment analysis without LSTMs(Long Short-Term Memory networks), then you simply need to replace it by a
model.add(layers.Dropout(0.3, noise_shape=None, seed=None))
model.add(layers.Dense(50, activation = "relu")) #ReLU" stands for Rectified Linear Unit, and it is a commonly used activation function in neural ne
model.add(layers.Dropout(0.2, noise_shape=None, seed=None))
model.add(layers.Dense(50, activation = "relu"))
# Output- Layer
model.add(layers.Dense(1, activation = "sigmoid")) #adds another Dense layer to the model, but with a single neuron instead of 50,i.e. out put layer
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 50)                500050

 dropout (Dropout)           (None, 50)                0

 dense_1 (Dense)             (None, 50)                2550

 dropout_1 (Dropout)         (None, 50)                0

 dense_2 (Dense)             (None, 50)                2550

 dense_3 (Dense)             (None, 1)                 51

=================================================================
Total params: 505,201
Trainable params: 505,201
Non-trainable params: 0
_________________________________________________________________
```

In [35]:
```
#For early stopping
# Stop training when a monitored metric has stopped improving.
# monitor: Quantity to be monitored.
# patience: Number of epochs with no improvement after which training will be stopped.
import tensorflow as tf #TensorFlow provides a wide range of tools and features for data processing, model building, model training, and model evalu
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
```

In [36]:
```
# We use the “adam” optimizer, an algorithm that changes the weights and biases during training.
# During training, the weights and biases of a machine learning model are updated iteratively to minimize the difference between the model's predict
# We also choose binary-crossentropy as loss (because we deal with binary classification) and accuracy as our evaluation metric.

model.compile(
 optimizer = "adam",
 loss = "binary_crossentropy",
 metrics = ["accuracy"]
)
```

In [37]:
```
# Now we're able to train our model. We'll do this with a batch_size of 500 and only for two epochs because I recognized that the model overfits if w
# batch size defines the number of samples that will be propagated through the network.
# For instance, let's say you have 1050 training samples and you want to set up a batch_size equal to 100.
# The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network.
# Next, it takes the second 100 samples (from 101st to 200th) and trains the network again.
# We can keep doing this procedure until we have propagated all samples through of the network.
# Problem might happen with the last set of samples. In our example, we've used 1050 which is not divisible by 100 without remainder.
# The simplest solution is just to get the final 50 samples and train the network.
##The goal is to find the number of epochs that results in good performance on a validation dataset without overfitting to the training data.
results = model.fit(
 X_train, y_train,
 epochs= 2,
 batch_size = 500,
 validation_data = (X_test, y_test),
 callbacks=[callback]
)
```

```
Epoch 1/2
80/80 [==============================] - 11s 112ms/step - loss: 0.4031 - accuracy: 0.8217 - val_loss: 0.2551 - val_accuracy: 0.8951
Epoch 2/2
80/80 [==============================] - 3s 34ms/step - loss: 0.2178 - accuracy: 0.9166 - val_loss: 0.2514 - val_accuracy: 0.8973
```

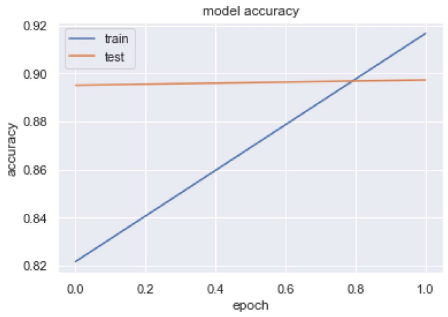
In [38]:
```
# Let's check mean accuracy of our model
print(np.mean(results.history["val_accuracy"])) # Good model should have a mean accuracy significantly higher than 50%
```

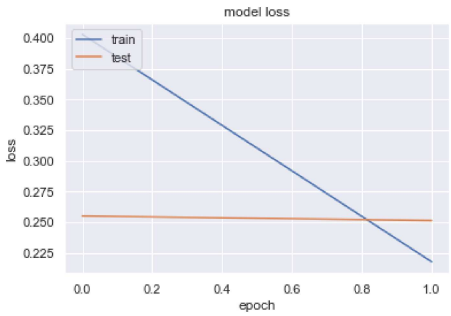
```
0.8962000012397766
```

In [39]:
```
#Let's plot training history of our model

# list all data in history
print(results.history.keys())
# summarize history for accuracy
plt.plot(results.history['accuracy']) #Plots the training accuracy of the model at each epoch.
plt.plot(results.history['val_accuracy']) #Plots the validation accuracy of the model at each epoch.
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(results.history['loss'])
plt.plot(results.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```





In [40]:
```
model.predict(X_test)
```

```
313/313 [==============================] - 1s 3ms/step
```

Out[40]:
```
array([[0.2554103 ],
       [0.97219026],
       [0.8446273 ],
       ...,
       [0.94011194],
       [0.9903155 ],
       [0.97657883]], dtype=float32)
```

The model's output analysis yields a single prediction value of 0.9865479 for a specific test data input. This value represents the predicted probability of the positive sentiment class (class 1) and, being close to 1, indicates high confidence in the model's positive class prediction for that input. The sigmoid activation function in the last layer maps the output values to the range [0,1], representing probabilities.