Text_classification___sentiment_analysis

November 24, 2021

Table of Contents

0.0.1 Data Preprocessing

1 Artificial Neural Networks Application

2 Vectorizers :Bag of words

2.0.1 Count Vectorizers

2.1 TF-IDF Vectorizer

3 Word Embedding : Time series model

3.1 Training own embedding

3.2 Using pretrained weights

[5]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/drive
```

Data Preprocessing

[10]:
```
import pandas as pd
import numpy as np
import tensorflow as tf
```

[11]:
```
dataset="drive/My Drive/IMDB Dataset.csv"
```

[12]:
```
df=pd.read_csv(dataset)
```

[13]:
```
df.head()
```

[13]:
```
                                                review sentiment
0  One of the other reviewers has mentioned that …  positive
1  A wonderful little production. <br /><br />The…  positive
2  I thought this was a wonderful way to spend ti…  positive
3  Basically there's a family where a little boy …  negative
4  Petter Mattei's "Love in the Time of Money" is…  positive
```

[14]:
```
df.describe()
```

[14]:
```
                                                  review sentiment
count                                              50000     50000
unique                                             49582         2
top     Loved today's show!!! It was a variety and not…  positive
freq                                                   5     25000
```

[15]:
```
df['sentiment'].value_counts()
```

[15]:
```
positive    25000
negative    25000
Name: sentiment, dtype: int64
```

[16]:
```
from bs4 import BeautifulSoup
def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()
```

[17]:
```
import re
def remove_between_square_brackets(text):
    return re.sub('\[[^]]*\]', '', text)
```

[18]:
```
def denoise_text(text):
    text = strip_html(text)
    text = remove_between_square_brackets(text)
    return text
```

[19]:
```
df['review']=df['review'].apply(denoise_text)
```

[20]:
```
df.head(20)
```

[20]:
```
                                                review sentiment
0   One of the other reviewers has mentioned that …  positive
1   A wonderful little production. The filming tec…  positive
2   I thought this was a wonderful way to spend ti…  positive
3   Basically there's a family where a little boy …  negative
4   Petter Mattei's "Love in the Time of Money" is…  positive
5   Probably my all-time favorite movie, a story o…  positive
6   I sure would like to see a resurrection of a u…  positive
7   This show was an amazing, fresh & innovative i…  negative
```

```
8   Encouraged by the positive comments about this…  negative
9   If you like original gut wrenching laughter yo…  positive
10  Phil the Alien is one of those quirky films wh…  negative
11  I saw this movie when I was about 12 when it c…  negative
12  So im not a big fan of Boll's work but then ag…  negative
13  The cast played Shakespeare.Shakespeare lost.I…  negative
14  This a fantastic movie of three prisoners who …  positive
15  Kind of drawn in by the erotic scenes, only to…  negative
16  Some films just simply should not be remade. T…  positive
17  This movie made it into one of my top 10 most …  negative
18  I remember this film,it was the first film i h…  positive
19  An awful film! It must have been up against so…  negative
```

[21]:
```
def remove_special_characters(text, remove_digits=True):
    pattern=r'[^a-zA-z0-9\s]'
    text=re.sub(pattern,'',text)
    return text
```

[22]:
```
def Convert_to_bin(text, remove_digits=True):
    if text=='positive':
      text= 1
    else:
      text=0
    return text
```

[23]:
```
df['review']=df['review'].apply(remove_special_characters)
```

[24]:
```
df['sentiment']=df['sentiment'].apply(Convert_to_bin)
```

[25]:
```
df.head(20)
```

[25]:

```
                                               review  sentiment
0   One of the other reviewers has mentioned that …          1
1   A wonderful little production The filming tech…          1
2   I thought this was a wonderful way to spend ti…          1
3   Basically theres a family where a little boy J…          0
4   Petter Matteis Love in the Time of Money is a …          1
5   Probably my alltime favorite movie a story of …          1
6   I sure would like to see a resurrection of a u…          1
7   This show was an amazing fresh  innovative ide…          0
8   Encouraged by the positive comments about this…          0
9   If you like original gut wrenching laughter yo…          1
10  Phil the Alien is one of those quirky films wh…          0
11  I saw this movie when I was about 12 when it c…          0
12  So im not a big fan of Bolls work but then aga…          0
13  The cast played ShakespeareShakespeare lostI a…          0
14  This a fantastic movie of three prisoners who …          1
```

```
15  Kind of drawn in by the erotic scenes only to …          0
16  Some films just simply should not be remade Th…          1
17  This movie made it into one of my top 10 most …          0
18  I remember this filmit was the first film i ha…          1
19  An awful film It must have been up against som…          0
```

[26]:
```
from sklearn.model_selection import train_test_split
```

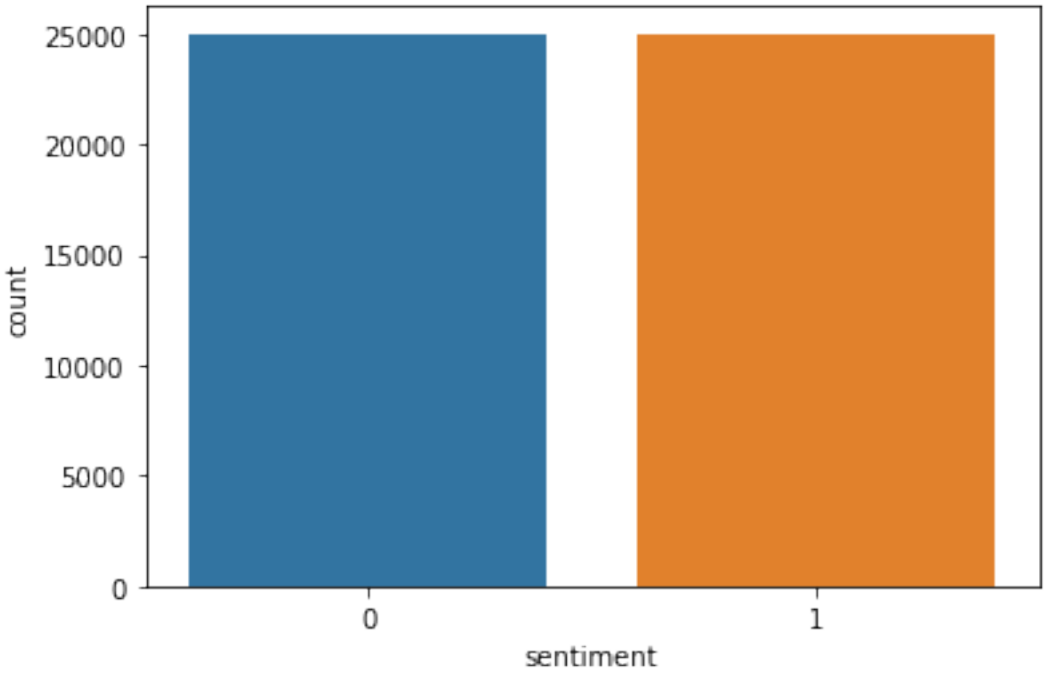
[27]:
```
X=df['review'].values
```

[28]:
```
Y=df['sentiment'].values
```

[29]:
```
import seaborn as sns
sns.countplot(df['sentiment'])
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```

[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1d591a2b70>
```



[30]:
```
X_train, X_test, Y_train, Y_test= train_test_split(X,Y, test_size=0.3)
```

0.1 Artificial Neural Networks Application

0.2 Vectorizers :Bag of words

Vectorizers: In this method, we create a single feature vector using all the words in the vocubulary. Each word is basically reagrded as a feature. So, the number of features is equal to the number of unique words in the vocab. Now, each sentence or review is a sample or record. Now, if the word is present in that sample it has some values and if the word is not present it is zero. Also called bag of words model

So, each sample has the same feature set size which is equal to the size of the vocabulary. Now, the vocabuary is basically made of the words in the train set. All the samples of the train and test set is fit using this vocabulary only. So, there may be some words in the test samples which are not present in the vocabulary, they are ignored.

Now, they form very sparse matrices or feature sets. Now, similar to a normal classification problem, the words become features of the record and the corresponding tag becomes the target value. So, it is actually like a common classification problem with number of features being equal to the distinct tokens in the training set.

This can be done in two ways:

1. Count Vectorizer: Here the count of a word in a particular sample or review. The count of that word becomes the value of the corresponding word feature. If a word in the vocab does not appear in the sample its value is 0.
2. TF-IDF Vectorizer: It is a better approach. It calculates two things term frequency and inverse document frequency. Term frequency= No. of times the word appears in the sample. IDF = log (number of time the word appears in the sample / number of time the word appears in the whole document). This helps to note some differences like the word "The" appears with same freq in almost all sentences while special words carrying significance like "good" don't. So, these TF and IDF terms are multiplied to obtain the vector formats for each sample.

It can also be done by TensorFlows Tokenizer.

For creating the matrix here, we need to use a tensorflow tokenizer, tokenizes the text to tokens. It can be done in mainly three ways: 1. Binary: X = tokenizer.sequences_to_matrix(x, mode='binary') : In this case, the value of a word feature is 1 if the word is present in the sample else zero.

2. Count: X = tokenizer.sequences_to_matrix(x, mode='count') : In this case, it is the number of times a word appear in the sentence.
3. TF-IDF: X = tokenizer.sequences_to_matrix(x, mode='tfidf') : In this we consider the TF of the word in the sample and IDF of the word in the sample with resect to the occurence of the word in the whole document.

Count Vectorizers

[ ]:
```
from sklearn.feature_extraction.text import CountVectorizer
```

[ ]:
```
vec = CountVectorizer()
```

[ ]:
```
vec.fit(X_train)
```

```
[ ]: CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                     dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                     lowercase=True, max_df=1.0, max_features=None, min_df=1,
                     ngram_range=(1, 1), preprocessor=None, stop_words=None,
                     strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                     tokenizer=None, vocabulary=None)
```

```
[ ]: x_train=vec.transform(X_train)
```

```
[ ]: x_test=vec.transform(X_test)
```

```
[ ]: x_train
```

```
[ ]: <35000x177470 sparse matrix of type '<class 'numpy.int64'>'
        with 4767914 stored elements in Compressed Sparse Row format>
```

```
[ ]: from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense
     from tensorflow.keras.optimizers import Adam
     model = Sequential()
     model.add(Dense(16, input_dim=x_train.shape[1], activation='relu'))
     model.add(Dense(16, activation='relu'))
     model.add(Dense(1, activation='sigmoid'))
     model.compile(loss='binary_crossentropy',optimizer='Adam',metrics=['accuracy'])
```

```
[ ]: model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                2839536
_________________________________________________________________
dense_1 (Dense)              (None, 16)                272
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 2,839,825
Trainable params: 2,839,825
Non-trainable params: 0
_________________________________________________________________
```

```
[ ]: history = model.fit(x_train, Y_train,epochs=100,verbose=True,batch_size=16)
```

```
Epoch 1/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.3146 -
accuracy: 0.8744
Epoch 2/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.1149 -
```

```
accuracy: 0.9576
Epoch 3/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0499 -
accuracy: 0.9821
Epoch 4/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0287 -
accuracy: 0.9897
Epoch 5/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0186 -
accuracy: 0.9942
Epoch 6/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0137 -
accuracy: 0.9959
Epoch 7/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0078 -
accuracy: 0.9974
Epoch 8/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0077 -
accuracy: 0.9977
Epoch 9/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0071 -
accuracy: 0.9978
Epoch 10/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0025 -
accuracy: 0.9993
Epoch 11/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0050 -
accuracy: 0.9985
Epoch 12/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0029 -
accuracy: 0.9991
Epoch 13/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0026 -
accuracy: 0.9993
Epoch 14/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0020 -
accuracy: 0.9995
Epoch 15/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0021 -
accuracy: 0.9992
Epoch 16/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0019 -
accuracy: 0.9993
Epoch 17/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0018 -
accuracy: 0.9995
Epoch 18/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.8614e-04 -
```

```
accuracy: 0.9999
Epoch 19/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0011 -
accuracy: 0.9998
Epoch 20/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0036 -
accuracy: 0.9992
Epoch 21/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0019 -
accuracy: 0.9995
Epoch 22/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.0364e-04 -
accuracy: 1.0000
Epoch 23/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.4194e-05 -
accuracy: 1.0000
Epoch 24/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.0649e-04 -
accuracy: 0.9999
Epoch 25/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.9919e-05 -
accuracy: 1.0000
Epoch 26/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.9021e-05 -
accuracy: 1.0000
Epoch 27/100
2188/2188 [==============================] - 6s 3ms/step - loss: 4.1639e-05 -
accuracy: 1.0000
Epoch 28/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.3546e-05 -
accuracy: 1.0000
Epoch 29/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0054 -
accuracy: 0.9995
Epoch 30/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0040 -
accuracy: 0.9994
Epoch 31/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0017 -
accuracy: 0.9995
Epoch 32/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0034 -
accuracy: 0.9993
Epoch 33/100
2188/2188 [==============================] - 6s 3ms/step - loss: 8.0544e-05 -
accuracy: 1.0000
Epoch 34/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.6258e-05 -
```

```
accuracy: 0.9999
Epoch 35/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.6887e-05 -
accuracy: 1.0000
Epoch 36/100
2188/2188 [==============================] - 6s 3ms/step - loss: 4.5846e-05 -
accuracy: 1.0000
Epoch 37/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.5241e-05 -
accuracy: 1.0000
Epoch 38/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.1134e-04 -
accuracy: 1.0000
Epoch 39/100
2188/2188 [==============================] - 6s 3ms/step - loss: 4.3324e-05 -
accuracy: 1.0000
Epoch 40/100
2188/2188 [==============================] - 6s 3ms/step - loss: 2.5094e-05 -
accuracy: 1.0000
Epoch 41/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0028 -
accuracy: 0.9994
Epoch 42/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0026 -
accuracy: 0.9994
Epoch 43/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0011 -
accuracy: 0.9997
Epoch 44/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.2710e-05 -
accuracy: 1.0000
Epoch 45/100
2188/2188 [==============================] - 6s 3ms/step - loss: 2.4858e-05 -
accuracy: 1.0000
Epoch 46/100
2188/2188 [==============================] - 6s 3ms/step - loss: 2.2871e-05 -
accuracy: 1.0000
Epoch 47/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.5169e-05 -
accuracy: 1.0000
Epoch 48/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.9690e-05 -
accuracy: 1.0000
Epoch 49/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.8087e-05 -
accuracy: 1.0000
Epoch 50/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.7954e-05 -
```

```
accuracy: 1.0000
Epoch 51/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.4394e-05 -
accuracy: 1.0000
Epoch 52/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.3059e-05 -
accuracy: 1.0000
Epoch 53/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.2217e-05 -
accuracy: 1.0000
Epoch 54/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.0851e-05 -
accuracy: 1.0000
Epoch 55/100
2188/2188 [==============================] - 6s 3ms/step - loss: 2.2537e-05 -
accuracy: 1.0000
Epoch 56/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.5112e-06 -
accuracy: 1.0000
Epoch 57/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.1919e-06 -
accuracy: 1.0000
Epoch 58/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.1334e-06 -
accuracy: 1.0000
Epoch 59/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.6236e-07 -
accuracy: 1.0000
Epoch 60/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0011 -
accuracy: 0.9997
Epoch 61/100
2188/2188 [==============================] - 6s 3ms/step - loss: 0.0068 -
accuracy: 0.9991
Epoch 62/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.9522e-04 -
accuracy: 0.9999
Epoch 63/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.1697e-07 -
accuracy: 1.0000
Epoch 64/100
2188/2188 [==============================] - 6s 3ms/step - loss: 4.8724e-07 -
accuracy: 1.0000
Epoch 65/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.2893e-07 -
accuracy: 1.0000
Epoch 66/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.8410e-07 -
```

```
accuracy: 1.0000
Epoch 67/100
2188/2188 [==============================] - 6s 3ms/step - loss: 8.6728e-08 -
accuracy: 1.0000
Epoch 68/100
2188/2188 [==============================] - 6s 3ms/step - loss: 4.5799e-08 -
accuracy: 1.0000
Epoch 69/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.7244e-08 -
accuracy: 1.0000
Epoch 70/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.2370e-09 -
accuracy: 1.0000
Epoch 71/100
2188/2188 [==============================] - 6s 3ms/step - loss: 3.7562e-09 -
accuracy: 1.0000
Epoch 72/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.6814e-09 -
accuracy: 1.0000
Epoch 73/100
2188/2188 [==============================] - 6s 3ms/step - loss: 8.2362e-10 -
accuracy: 1.0000
Epoch 74/100
2188/2188 [==============================] - 6s 3ms/step - loss: 4.5086e-10 -
accuracy: 1.0000
Epoch 75/100
2188/2188 [==============================] - 6s 3ms/step - loss: 2.9417e-10 -
accuracy: 1.0000
Epoch 76/100
2188/2188 [==============================] - 6s 3ms/step - loss: 2.0863e-10 -
accuracy: 1.0000
Epoch 77/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.6438e-10 -
accuracy: 1.0000
Epoch 78/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.3974e-10 -
accuracy: 1.0000
Epoch 79/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.2130e-10 -
accuracy: 1.0000
Epoch 80/100
2188/2188 [==============================] - 6s 3ms/step - loss: 1.1140e-10 -
accuracy: 1.0000
Epoch 81/100
2188/2188 [==============================] - 6s 3ms/step - loss: 9.6768e-11 -
accuracy: 1.0000
Epoch 82/100
2188/2188 [==============================] - 6s 3ms/step - loss: 9.0848e-11 -
```

```
accuracy: 1.0000
Epoch 83/100
2188/2188 [==============================] - 6s 3ms/step - loss: 9.1615e-11 -
accuracy: 1.0000
Epoch 84/100
2188/2188 [==============================] - 6s 3ms/step - loss: 8.6676e-11 -
accuracy: 1.0000
Epoch 85/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.6896e-11 -
accuracy: 1.0000
Epoch 86/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.8929e-11 -
accuracy: 1.0000
Epoch 87/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.5444e-11 -
accuracy: 1.0000
Epoch 88/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.0275e-11 -
accuracy: 1.0000
Epoch 89/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.5885e-11 -
accuracy: 1.0000
Epoch 90/100
2188/2188 [==============================] - 6s 3ms/step - loss: 7.0733e-11 -
accuracy: 1.0000
Epoch 91/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.1445e-11 -
accuracy: 1.0000
Epoch 92/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.4795e-11 -
accuracy: 1.0000
Epoch 93/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.4106e-11 -
accuracy: 1.0000
Epoch 94/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.8243e-11 -
accuracy: 1.0000
Epoch 95/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.1660e-11 -
accuracy: 1.0000
Epoch 96/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.3604e-11 -
accuracy: 1.0000
Epoch 97/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.3272e-11 -
accuracy: 1.0000
Epoch 98/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.2296e-11 -
```

```
accuracy: 1.0000
Epoch 99/100
2188/2188 [==============================] - 6s 3ms/step - loss: 5.3251e-11 -
accuracy: 1.0000
Epoch 100/100
2188/2188 [==============================] - 6s 3ms/step - loss: 6.2355e-11 -
accuracy: 1.0000
```

[ ]:
```
model.evaluate(x_train,Y_train)
```

```
1094/1094 [==============================] - 2s 2ms/step - loss: 6.4925e-11 -
accuracy: 1.0000
```

[ ]:
```
[6.492464899032768e-11, 1.0]
```

[ ]:
```
model.evaluate(x_test,Y_test)
```

```
469/469 [==============================] - 1s 2ms/step - loss: 3.1591 -
accuracy: 0.8731
```

[ ]:
```
[3.1590538024902344, 0.8730666637420654]
```

[ ]:
```
## Done
```

0.2.1 TF-IDF Vectorizer

[ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

[ ]:
```
vec = TfidfVectorizer()
```

[ ]:
```
vec.fit(X_train)
```

[ ]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_features=None,
                min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

[ ]:
```
x_train=vec.transform(X_train)
```

[ ]:
```
x_test=vec.transform(X_test)
```

[ ]:
```
x_train
```

[ ]:
```
<35000x176924 sparse matrix of type '<class 'numpy.float64'>'
        with 4778597 stored elements in Compressed Sparse Row format>
```

```
[ ]: from sklearn.linear_model import LogisticRegression
     lr = LogisticRegression()
     lr.fit(x_train, Y_train)
```

```
[ ]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                       intercept_scaling=1, l1_ratio=None, max_iter=100,
                       multi_class='auto', n_jobs=None, penalty='l2',
                       random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                       warm_start=False)
```

```
[ ]: score_test = lr.score(x_test, Y_test)
```

```
[ ]: score_test
```

```
[ ]: 0.8941333333333333
```

```
[ ]: score_train = lr.score(x_train, Y_train)
```

```
[ ]: score_train
```

```
[ ]: 0.9325428571428571
```

```
[ ]: ## Done
```

0.3 Word Embedding : Time series model

Word Embedding: In this method, the words are individually represented as a vector. In case of the bag of words all of the words made up a vector. Here, there are 100 words in a vocabulary, so, a specific word will be represented by a vector of size 100 where the index corresponding to that word will be equal to 1, and others will be 0.

So, Each sample having different number of words will basically have a different number of vectors, as each word is equal to a vector. Now, to feed a model we will need to have the same dimension for each sample, and as a result padding is needed to make the number of words in each sample equal to each other.

Basically in the bag of words or vectorizer approach, if we have 100 words in our total vocabulary, and a sample with 10 words and a sample with 15 words, after vectorization both the sample sizes would be an array of 100 words, but here for the 10 words it will be a (10 x 100) i,e 100 length vector for each of the 10 words and similarly for 15th one size will be (15 x 100). So, we need to find the longest sample and pad all others up to match the size.

We can do this in some ways:

One-Hot encoding: It is just taking the size of the vocabulary and making an array of that size with 0's at all indices and 1 at only the index of the word. But this things provides us with a very less information.

The Second Choice is word embeddings.

The one hot encoder is a pretty hard coded approach. It is of a very high dimension and sparse with a very low amount of data. **Embedding is a way to create a dense vector representation out of the sparse representations.** It is of a lower dimension and helps to capture much more informations. It more like captures the relation and similarities between words using how they appear close to each other. For example, king, queen, men and women will have some relations.

Say, we are having 10k words are being embedded in a 300 dimensional embedding space. To do this, we declare the number of nodes in the embedding layer =300. Now, each word of the 10k words enter the embedding layer as a 10k sized individual vector, Now, each of the words will be placed in 300 dimensional plane based on their similarities with one another which is decieded by several factors, like the order in which the words occur. Now, being placed in 300 Dimensional plane the words will have a 300 length tuple to represent it which are actually the coordinates of the point on the 300 dimensional plane. So, this 300 dimensional tuple becomes the new feature set or representing vector for the word.

So, the vector for the word decreased from 10k to 300. The tuples serve as feature vectors between two words and the cosine angle between the vectors represent the similarity between the two words.

We can do this in two ways:

1. Using our own embeddings
2. Using pretrained embeddings

Making our own embedding

Now, for the embedding, we need to send each sample through an embedding layer first then move to make them dense using embedding. These embedding layers see how the words are used, i.e, it tries to see if two words always occur together or are used in contrast. After judging all these factors the layer places the word in a position one the n-dimensional embedding space.

Using pretrained embedded matrices

We can use pretrained word embeddings like word2vec by google and GloveText by standford.They are trained on huge corpuses with billions of examples and words. Now, they have billions of words we have only a 10k so, training our model with a billion words will be very inefficient, So, we need to just select out our required word's embeddings from their pretrained embeddings.

Now, **How are these embeddings found?**

For google's word2vec implementations, there are two ways:

1. Continous bag of words
2. Spin Gram.

Both of these algorithms actually use a Neural Network with a single hidden layer to generate the embedding.

Now, for CBOW, the context of the words , i.e, the words befor and after the required words are fed to the neural network, and the model is needed to predict the word.

For the Spin-Gram, the words are given and the model has to predict the context words.

In both cases, the feature vectors or encoded vectors of the words are fed to the input. The output has a softmax layer with number of nodes equal to the vocabulary size, which gives the percentage of prediction for each word. Though we don't use the output layer actually.

We go for the weight matrix produced in the hidden layer. **The number of nodes in the hidden layer is equal to the embedding dimension.** So, say if there are 10k words in a vocabulary and 300 nodes in the ghidden layer, each node in the hidden layer will have an array of weights of dimension of 10k for each word after training.

Because the neural network units work on

y=f(w1x1+w2x2+…….wnxn)

Here x1, x2…… xn are the words and so n= number of words in vocabulary=10k.

So for 10k x's there will be 10k w's. Now foe 1 node there are 10k length weight matrix. For 300 combined we have a matrix of 300 x 10k wieghts. Now, if we concatenate, we will have 300 rows and 10k columns. Let's transpose the matrix. We will get 300 columns and 10k rows. Each row represents a word, and the 300 column values represent a 300 length wieght vector for that word.

This weight vector is the obtained embedding of length 300.

0.3.1 Training own embedding

[ ]:
```
from keras.preprocessing.text import Tokenizer
```

```
Using TensorFlow backend.
```

[ ]:
```
tokenizer = Tokenizer(num_words=10000)
```

[ ]:
```
tokenizer.fit_on_texts(X_train)
```

1) tokenize.fit_on_text() –» Creates the vocabulary index based on word frequency. For example, if you had the phrase "My dog is different from your dog, my dog is prettier", word_index["dog"] = 0, word_index["is"] = 1 (dog appears 3 times, is appears 2 times)

2) tokenize.text_to_sequence() –» Transforms each text into a sequence of integers. Basically if you had a sentence, it would assign an integer to each word from your sentence. You can access tokenizer.word_index() (returns a dictionary) to verify the assigned integer to your word.

[ ]:
```
x_train = tokenizer.texts_to_sequences(X_train)
```

[ ]:
```
x_test = tokenizer.texts_to_sequences(X_test)
```

[ ]:
```
vocab = len(tokenizer.word_index) + 1
```

+1 for padding

[ ]:
```
from keras.preprocessing.sequence import pad_sequences
```

[ ]:
```
maxlen = 100
```

[ ]:
```
x_train = pad_sequences(x_train, padding='post', maxlen=maxlen)
x_test = pad_sequences(x_test, padding='post', maxlen=maxlen)
```

[ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding,Dense, Activation, MaxPool1D
from tensorflow.keras.optimizers import Adam
emb_dim=100

model= Sequential()
model.add(Embedding(input_dim=vocab, output_dim=emb_dim, input_length=maxlen))
model.add(MaxPool1D())
model.add(Dense(16,activation="relu"))
model.add(Dense(16,activation="relu"))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='Adam',loss='binary_crossentropy',metrics=['accuracy'])
```

Maxpooling is used to convert the sparse matrix denser

[ ]:
```
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 100, 100)          17754200
_________________________________________________________________
max_pooling1d_4 (MaxPooling1 (None, 50, 100)           0
_________________________________________________________________
dense_24 (Dense)             (None, 50, 16)            1616
_________________________________________________________________
dense_25 (Dense)             (None, 50, 16)            272
_________________________________________________________________
dense_26 (Dense)             (None, 50, 1)             17
=================================================================
Total params: 17,756,105
Trainable params: 17,756,105
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
history = model.fit(x_train, Y_train,epochs=35,verbose=True,batch_size=16)
```

Here at each step the model weights as well as the embedding dimensions are getting corrected

[ ]:
```
test_score=model.evaluate(x_test,Y_test)
```

[ ]:
```
test_score
```

[ ]:
```
train_score=model.evaluate(x_train,Y_train)
```

[ ]:
```
train_score
```

Train accuray 83.7 Test accuracy 77.4

[51]:
```
## Done
```

0.3.2 Using pretrained weights

[6]:
```
file_path="drive/My Drive/glove.6B/"
```

[7]:
```
import os
files=os.listdir(file_path)
```

[8]:
```
files
```

[8]:
```
['glove.6B.100d.txt',
 'glove.6B.200d.txt',
 'glove.6B.50d.txt',
 'glove.6B.300d.txt']
```

So, here are four files each of a different embedding size 50, 100, 200, 300

We will go for 50.

Now, here there are 6 billion words we have much less words than that so what we will do is, we will find our words and pick the weights from the pretrained words

[31]:
```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(X_train)
```

[32]:
```
x_train = tokenizer.texts_to_sequences(X_train)
x_test = tokenizer.texts_to_sequences(X_test)
```

[44]:
```
from keras.preprocessing.sequence import pad_sequences
maxlen=100
x_train = pad_sequences(x_train, padding='post', maxlen=maxlen)
x_test = pad_sequences(x_test, padding='post', maxlen=maxlen)
```

```
Using TensorFlow backend.
```

[33]:
```
emb_dim=50
vocab=len(tokenizer.word_index)+1
emb_mat= np.zeros((vocab,emb_dim))
```

Initializing a zero matrix for each word, they will be compared to have their final embedding

[38]:
```
with open(file_path+'glove.6B.50d.txt') as f:
        for line in f:
            word, *emb = line.split()
            if word in tokenizer.word_index:
              ind=tokenizer.word_index[word]
              emb_mat[ind]=np.array(emb,dtype="float32")[:emb_dim]
```

This is an extractor for the task, so we have the embeddings and the words in a line. So, we just compare the words pick out the indices in our dataset. Take the vectors and place it in the embedding matrix at a index corresponding to the index of the word in our dataset.

We have used a *emb because the embedding matrix is variant in size.

[41]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding,Dense, Activation, MaxPool1D
from tensorflow.keras.optimizers import Adam
emb_dim=50
maxlen=100
model= Sequential()
model.add(Embedding(input_dim=vocab, output_dim=emb_dim,weights=[emb_mat],␣
 ↪input_length=maxlen,trainable=False))
model.add(MaxPool1D())
model.add(Dense(16,activation="relu"))
model.add(Dense(16,activation="relu"))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='Adam',loss='binary_crossentropy',metrics=['accuracy'])
```

[42]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 50)           8839300
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 50, 50)            0
_________________________________________________________________
dense (Dense)                (None, 50, 16)            816
_________________________________________________________________
dense_1 (Dense)              (None, 50, 16)            272
_________________________________________________________________
dense_2 (Dense)              (None, 50, 1)             17
=================================================================
Total params: 8,840,405
Trainable params: 1,105
Non-trainable params: 8,839,300
_________________________________________________________________
```

[46]:
```
history = model.fit(x_train, Y_train,epochs=50,verbose=True,batch_size=16)
```

```
Epoch 1/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6815 -
accuracy: 0.5536
Epoch 2/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6809 -
accuracy: 0.5552
```

```
Epoch 3/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6806 -
accuracy: 0.5557
Epoch 4/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6801 -
accuracy: 0.5567
Epoch 5/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6797 -
accuracy: 0.5575
Epoch 6/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6795 -
accuracy: 0.5582
Epoch 7/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6794 -
accuracy: 0.5584
Epoch 8/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6792 -
accuracy: 0.5589
Epoch 9/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6789 -
accuracy: 0.5599
Epoch 10/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6787 -
accuracy: 0.5602
Epoch 11/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6786 -
accuracy: 0.5604
Epoch 12/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6786 -
accuracy: 0.5604
Epoch 13/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6785 -
accuracy: 0.5609
Epoch 14/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6786 -
accuracy: 0.5603
Epoch 15/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6782 -
accuracy: 0.5618
Epoch 16/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6782 -
accuracy: 0.5615
Epoch 17/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6781 -
accuracy: 0.5619
Epoch 18/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6780 -
accuracy: 0.5620
```

```
Epoch 19/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6779 -
accuracy: 0.5620
Epoch 20/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6780 -
accuracy: 0.5621
Epoch 21/50
2188/2188 [==============================] - 7s 3ms/step - loss: 0.6780 -
accuracy: 0.5618
Epoch 22/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6779 -
accuracy: 0.5621
Epoch 23/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6777 -
accuracy: 0.5626
Epoch 24/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6779 -
accuracy: 0.5623
Epoch 25/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6776 -
accuracy: 0.5630
Epoch 26/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6779 -
accuracy: 0.5626
Epoch 27/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6776 -
accuracy: 0.5624
Epoch 28/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6777 -
accuracy: 0.5629
Epoch 29/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6776 -
accuracy: 0.5630
Epoch 30/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6776 -
accuracy: 0.5628
Epoch 31/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6774 -
accuracy: 0.5633
Epoch 32/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6776 -
accuracy: 0.5628
Epoch 33/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6773 -
accuracy: 0.5637
Epoch 34/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6774 -
accuracy: 0.5635
```

```
Epoch 35/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6774 -
accuracy: 0.5637
Epoch 36/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6774 -
accuracy: 0.5634
Epoch 37/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6774 -
accuracy: 0.5635
Epoch 38/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6772 -
accuracy: 0.5640
Epoch 39/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6770 -
accuracy: 0.5647
Epoch 40/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6773 -
accuracy: 0.5638
Epoch 41/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6772 -
accuracy: 0.5643
Epoch 42/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6772 -
accuracy: 0.5638
Epoch 43/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6771 -
accuracy: 0.5640
Epoch 44/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6771 -
accuracy: 0.5646
Epoch 45/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6772 -
accuracy: 0.5644
Epoch 46/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6771 -
accuracy: 0.5645
Epoch 47/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6770 -
accuracy: 0.5645
Epoch 48/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6772 -
accuracy: 0.5637
Epoch 49/50
2188/2188 [==============================] - 8s 4ms/step - loss: 0.6770 -
accuracy: 0.5646
Epoch 50/50
2188/2188 [==============================] - 8s 3ms/step - loss: 0.6771 -
accuracy: 0.5646
```

[47]:
```
test_score=model.evaluate(x_test,Y_test)
```

```
469/469 [==============================] - 1s 3ms/step - loss: 0.6773 -
accuracy: 0.5639
```

[48]:
```
test_score
```

[48]: [0.6773159503936768, 0.563922643661499]

[49]:
```
train_score=model.evaluate(x_train,Y_train)
```

```
1094/1094 [==============================] - 3s 3ms/step - loss: 0.6767 -
accuracy: 0.5651
```

[50]:
```
train_score
```

[50]: [0.6766629815101624, 0.5651329159736633]

[52]:
```
## Done
```