In [1]:

```
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt

import sqlite3
import pandas as pd
import numpy as np
import seaborn as sns
import nltk
from tqdm import tqdm
from bs4 import BeautifulSoup
import re
import datetime
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
import itertools
import itertools
import collections
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from keras.models import Sequential
from sklearn.model_selection import train_test_split
```

```
Using TensorFlow backend.
```

In [0]:

```
!pip install PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
#Authenticate and create the PyDrive client

auth.authenticate_user()
gauth=GoogleAuth()
gauth.credentials=GoogleCredentials.get_application_default()
drive=GoogleDrive(gauth)
```

In [0]:

```
link ="https://drive.google.com/open?id=18yHOyLnrSgzAabvXoev4C2yjzSThegLB"
fluff, id = link.split('=')
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('database.sqlite')
```

Data Read and Cleanup

In [6]:

```
# Load the data from .sqlite file

db=sqlite3.connect('database.sqlite')

# select all reviews from given dataset
# we are considering a review is positive or negative on the basis of the Score column which is nothing but a rating given
# by a customer for a product. If a score >3 it is considered as positive elseif score<3 it is negative and score=3 is neutral
# Therefore all reviews which are having score other than 3 are taken into account.

filtered_data=pd.read_sql_query("""
```

```
SELECT *
FROM Reviews WHERE Score!=3""",db)

# Replace this numbers in Score column as per our assumptions i.e replace 3+ with positive 1 and 3- wit
h negative 0
def partition(x):
    if x < 3:
        return 0
    return 1

# changing reviews with score less than 3 to be positive (1) and vice-versa
actualScore = filtered_data['Score']
positiveNegative = actualScore.map(partition)
filtered_data['Score'] = positiveNegative
print(filtered_data.shape)
```

```
(525814, 10)
```

In [0]:

```
# converting datestamp into string representable form as YYYY-MM-DD
filtered_data["Time"] = filtered_data["Time"].map(lambda t: datetime.datetime.fromtimestamp(t).strftime
('%Y-%m-%d'))
```

In [8]:

```
# There is lot of duplicate data present as we can see above productId B007OSBE1U
# have multiple duplicate reviews this is what we need to avoid.

# so first step is to sort the data and then remove duplicate entries so that only
# one copy of them should be remain in our data.
dup_free=filtered_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"})
# dup_free.head()
# This is shape of our dataset of 100k datapoints after removal of dups
dup_free.shape
```

Out[8]:

```
(364173, 10)
```

In [0]:

```
final_filtered_data=dup_free[dup_free.HelpfulnessNumerator<=dup_free.HelpfulnessDenominator]
```

In [10]:

```
final_filtered_data.shape
```

Out[10]:

```
(364171, 10)
```

In [11]:

```
((final_filtered_data['Id'].size*1.0)/(filtered_data['Id'].size*(1.0)))*100
```

Out[11]:

```
69.25852107399194
```

so after data cleanup we left with 69.25%data of 525k datapoints

In [0]:

```
filtered_data=filtered_data.sort_values(by='Time').reset_index(drop=True)
```

In [13]:

```
final=filtered_data.sample(frac=0.18,random_state=2)
final.shape
```

Out[13]:

```
(94647, 10)
```

In [14]:

```
print("Positive Reviews: ",final[final.Score ==1].shape[0])
print("Positive Reviews: ",final[final.Score ==0].shape[0])
```

```
Positive Reviews:  79928
Positive Reviews:  14719
```

Dataset seems balanced now with 52% positive reviews and 48% negative reviews

In [15]:

```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Out[15]:

```
True
```

Text Preprocessing

In [0]:

```
# Now we have already done with data cleanup part. As in our dataset most cruicial or I can say most de
terminant feature
# from which we can say it is positive or negative review is review Text.
# So we are need to perform some Text Preprocessing on it before we actually convert it into word vecto
r or vectorization

# I am creating some precompiled objects for our regular expressions cause it will be used for over ~64
K times (in our case)
# as it seems fast but using regular expression is CPU expensive task so it would be faster to use prec
ompiled search objects.

_wont  = re.compile(r"won't")
_cant  = re.compile(r"can\'t")
_not   = re.compile(r"n\'t")
_are   = re.compile(r"\'re")
_is    = re.compile(r"\'s")
_would = re.compile(r"\'d")
_will  = re.compile(r"\'ll")
_have  = re.compile(r"\'ve")
_am    = re.compile(r"\'m")

# we are ignoring "not" from stopwords as "not" plays important role for semantic analysis as it can al
one change the
# meaning of whole sentence
stopWords = set(stopwords.words('english'))
sw=stopWords.copy()
sw.discard('not')

def expand_abbrevated_words(phrase):
    phrase = re.sub(_wont, "will not", phrase)
    phrase = re.sub(_cant, "can not", phrase)
    phrase = re.sub(_not, " not", phrase)
    phrase = re.sub(_are, " are", phrase)
    phrase = re.sub(_is, " is", phrase)
```

```
    phrase = re.sub(_would, " would", phrase)
    phrase = re.sub(_will, " will", phrase)
    phrase = re.sub(_have, " have", phrase)
    phrase = re.sub(_am, " am", phrase)
    return phrase

# As this dataset is web scrapped from amazon.com while scrapping there might be a good chance that we are getting some garbage
# characters/words/sentences in our Text data like html tags,links, alphanumeric characters so we ought to remove them
def remove_unwanted_char(data):
    processed_data=[]
    for sentence in tqdm(data):
        sentence = re.sub(r"http\S+", "", sentence) # this will remove links
        sentence = BeautifulSoup(sentence, 'lxml').get_text()
        sentence = re.sub("\S*\d\S*", "", sentence).strip() #remove alphanumeric words
        sentence = re.sub('[^A-Za-z]+', ' ', sentence) #remove special characters
        sentence =  expand_abbrevated_words(sentence)
        # we need to convert everything into lower case because I dont want my model to treat same word differently
        # if it appears in the begining of sentence and somewhere middle of sentence.
        # Also remove stopword froms from sentences
        sentence =" ".join(j.lower() for j in sentence.split() if j.lower() not in sw)
        processed_data.append(sentence)
    return processed_data

def preprocess_my_data(data):
    return remove_unwanted_char(data)
```

In [17]:

```
data_to_be_processed=final['Text'].values
processed_data=preprocess_my_data(data_to_be_processed)
label=final['Score']
print(len(processed_data))
```

```
100%|██████████| 94647/94647 [00:37<00:00, 2524.48it/s]
```

```
94647
```

In [18]:

```
final['CleanedText']=processed_data
print(processed_data[0])
```

```
tried several times get good coconut flavored coffee little success boyer trick great coffee good amoun
t coconut flavor highly recommend
```

In [0]:

```
reviews=processed_data.copy()
```

In [0]:

```
vocab=[x.split() for x in reviews] # splitting sentences to words
```

In [0]:

```
vocab=list(itertools.chain.from_iterable(vocab)) # getting vocabulary
```

In [0]:

```
word_freq=collections.Counter(vocab) # word to frequency count dictionary
```

In [23]:

```
print("Size of Vocabulory : ",len(word_freq))
```

```
Size of Vocabulory :  54224
```

In [0]:

```
frequency=np.array(list(word_freq.values()))
words=list(word_freq.keys())
```

In [0]:

```
frequency=list(np.argsort(frequency))
```

In [0]:

```
frequency.reverse() # to get words in descending order according to frequency
```

In [0]:

```
word_to_frequency_index=dict()
# Assigning each word to its corresponding index according to it occurence frequency in descending orde
r
for count, index in enumerate(frequency,1):
  word_to_frequency_index.update({words[index]:count})
```

In [0]:

```
top_word_freq=5000
```

In [0]:

```
# Replace each word in review with its index of occurence frequency (only top 5000 words will be replac
ed and others are ignored)
def replace_word_to_occurenece_index(review):
  updated_rev=list()
  for word in review.split():
    count=word_to_frequency_index[word]
    if count<=top_word_freq:
      updated_rev.append(count)

  return np.array(updated_rev)
```

In [0]:

```
reviews=list(map(replace_word_to_occurenece_index,reviews))
```

In [0]:

```
reviews=np.array(reviews)
```

In [32]:

```
len(reviews)
```

Out[32]:

```
94647
```

In [33]:

```
len(label)
```

Out[33]:

94647

In [0]:

```
max_review_length = 600
reviews = sequence.pad_sequences(reviews, maxlen=max_review_length)
```

In [35]:

```
x_tr, x_test, y_tr, y_test = train_test_split(reviews, label, test_size=0.3, random_state=0)
print("Sizes of Train,test dataset after split: {0} , {1}".format(len(x_tr),len(x_test)))
```

```
Sizes of Train,test dataset after split: 66252 , 28395
```

In [36]:

```
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_word_freq+1, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

```
W0902 08:07:16.445701 139849670682496 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-pa
ckages/keras/backend/tensorflow_backend.py:66: The name tf.get_default_graph is deprecated. Please use
tf.compat.v1.get_default_graph instead.

W0902 08:07:16.483319 139849670682496 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-pa
ckages/keras/backend/tensorflow_backend.py:541: The name tf.placeholder is deprecated. Please use tf.co
mpat.v1.placeholder instead.

W0902 08:07:16.490946 139849670682496 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-pa
ckages/keras/backend/tensorflow_backend.py:4432: The name tf.random_uniform is deprecated. Please use t
f.random.uniform instead.

W0902 08:07:16.844098 139849670682496 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-pa
ckages/keras/optimizers.py:793: The name tf.train.Optimizer is deprecated. Please use tf.compat.v1.trai
n.Optimizer instead.

W0902 08:07:16.867537 139849670682496 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-pa
ckages/keras/backend/tensorflow_backend.py:3657: The name tf.log is deprecated. Please use tf.math.log
instead.

W0902 08:07:16.874839 139849670682496 deprecation.py:323] From /usr/local/lib/python3.6/dist-packages/t
ensorflow/python/ops/nn_impl.py:180: add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.
array_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, 600, 32)           160032    
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               53200     
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101       
=================================================================
Total params: 213,333
Trainable params: 213,333
Non-trainable params: 0
_________________________________________________________________
None
```

In [37]:

```
history=model.fit(x_tr, y_tr, nb_epoch=10, batch_size=64)
```

```
W0902 08:07:17.943202 139849670682496 deprecation_wrapper.py:119] From /usr/local/lib/python3.6/dist-pa
ckages/keras/backend/tensorflow_backend.py:1033: The name tf.assign_add is deprecated. Please use tf.co
mpat.v1.assign_add instead.
```

```
Epoch 1/10
66252/66252 [==============================] - 847s 13ms/step - loss: 0.2430 - acc: 0.9065
Epoch 2/10
66252/66252 [==============================] - 876s 13ms/step - loss: 0.1733 - acc: 0.9341
Epoch 3/10
66252/66252 [==============================] - 872s 13ms/step - loss: 0.1529 - acc: 0.9418
Epoch 4/10
66252/66252 [==============================] - 867s 13ms/step - loss: 0.1352 - acc: 0.9494
Epoch 5/10
66252/66252 [==============================] - 865s 13ms/step - loss: 0.1180 - acc: 0.9564
Epoch 6/10
66252/66252 [==============================] - 865s 13ms/step - loss: 0.0992 - acc: 0.9642
Epoch 7/10
66252/66252 [==============================] - 868s 13ms/step - loss: 0.0849 - acc: 0.9707
Epoch 8/10
66252/66252 [==============================] - 868s 13ms/step - loss: 0.0740 - acc: 0.9756
Epoch 9/10
66252/66252 [==============================] - 871s 13ms/step - loss: 0.0607 - acc: 0.9802
Epoch 10/10
66252/66252 [==============================] - 872s 13ms/step - loss: 0.0505 - acc: 0.9839
```

In [0]:

```
# Final evaluation of the model
scores = model.evaluate(x_test, y_test, verbose=0)
```

In [41]:

```
print("Accuracy on Test Data : ",scores[1])
```

```
Accuracy on Test Data :  0.9218524388201452
```

Summary

We got **92.18%** accuracy on Test data

Conclusion

1. Traditional RNN's cannot remember the Long Term relationships between words i.e simple RNN's suffer from vanishing gradients problem as no of features or dimensions increases.
2. In case of Amazon fine food reviews as words in sentences increases our model will start facing vanishing gradient problem.
3. LSTM's are remedy for this problem as it has no prblem remembering the dependencies of previous inputs with the future inputs.
4. LSTMS comprised of :
    - Cell State
    - Forget gate layer
    - Input gate layer
    - Output Layer
5. Key thing in LSTM is Cell state.
6. Forget gate layer takes care of which information need to be dropped
7. Input gate takes care of which new information need to be passed and replaced with the information dropped
8. output layer decides what information to be fed to next LSTM unit