Emotion Analysis with PyTorch

Submitted in partial fulfillment of the requirements of

Mini Project (CSM401)

for

Second Year of Computer Engineering

By

Dishant 19102A0003

Under the Guidance of

Prof. Sanjeev Dwivedi

Department of Computer Engineering

Vidyalankar Institute of Technology
Wadala(E), Mumbai-400437

University of Mumbai

2020-21

Abstract

In our daily life, we go through different situations and develop a feeling about it. Emotion is a strong feeling about a human's situation or relation with others. These feelings and express Emotions are expressed as facial expression. The primary emotion levels are of six types namely; **Love, Joy, Anger, Sadness, Fear, and Surprise**. Human expresses emotion in different ways including facial expression, speech, gestures/actions, and written text. This project mainly focuses on emotions hidden in a written text.

As technology progresses, the internet is now commonly used on PCs, tablets, and smartphones. This generates a huge amount of data, especially textual data. It has become impossible to manually analyse all the data for a specific purpose. New research directions have emerged from automatic data analysis like automatic emotion analysis. Emotion analysis has attracted researchers' attention because of its applications in different fields. For example, security agencies can track emails/messages/blogs, etc., and detect suspicious activities.

The business communities nowadays prefer to use emotional marketing. In emotional marketing, they try to stimulate customers' emotions to buy products or services. Some emotional marketing advertisements are given at the link.

This project aims at analysing the given text for these 6 broad categories of emotions and classifies the text into these emotions by implementing a 3-layered (2 hidden 1 output layers) Artificial Neural Network using PyTorch classes. The tf-idf vectors of the words in the training set are passed as training features to the model to predict one out of these 6 major emotions.

Also, the relative percentage of each emotion in a particular text is also plotted as pyplot graphs to provide a clearer understanding about the given statement.

Table of Contents

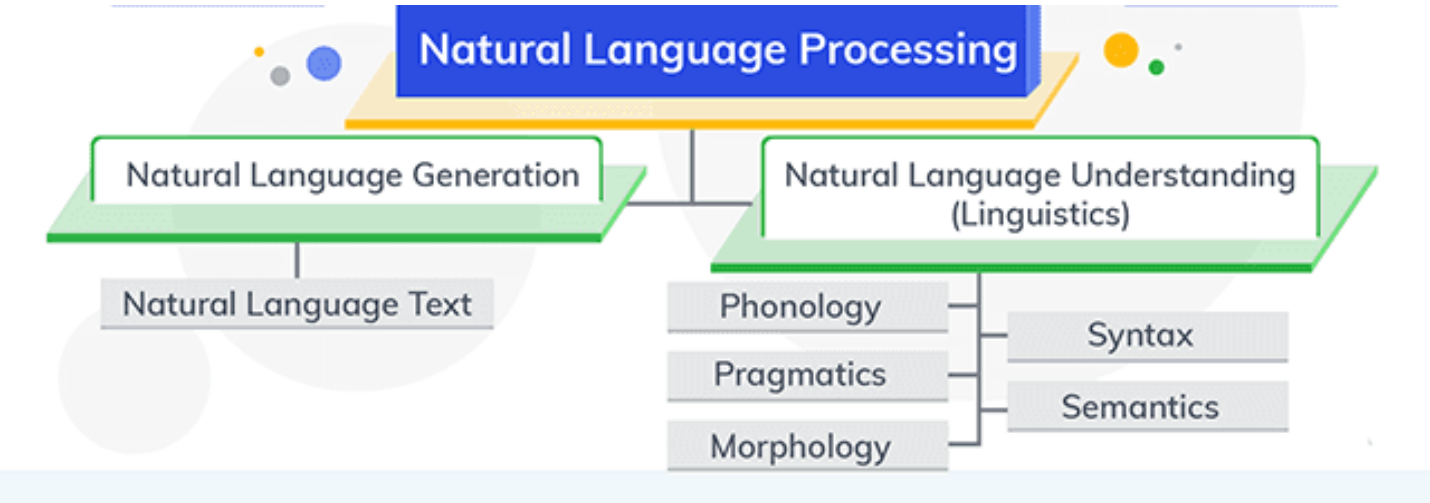
Introduction

Humans have emotions, which correspond to their mental statuses and behavioural patterns. **There are two dimensions to the emotions felt by humans: an internal and an external dimension**.

The internal or subjective component of emotions is the one that the individual immediately perceives. The individual can't share this component with others, which therefore remains within that individual's direct experience

It corresponds to the psychological status of the person, but also to the activation of emotion-specific neural circuits of the brain. **This internal dimension is not observable through the methodology of content analysis and it's, for all purposes, a black box**.

There's also an external or shared dimension to emotions though. **As the individual feels emotions, they share, voluntarily or not, this external component with the rest of the environment**. This sharing is done by emitting signals that take the form of facial expressions; gestures; attitudes; and written texts or chats. The focus of this project are only the written texts.



The question then becomes, how can we use these texts in order to study the black box or internal component of emotions; that is, the subjective emotional status. To do this, we need an additional hypothesis that justifies the study of emotions by means of emotional signals.

The underlying hypothesis is, therefore, that some kind of **unique mapping f: (words)→(emotions)** exists, and that we can learn about it by means of machine learning.

In Natural Language Processing, we don't observe the whole set of emotional behaviours associated with an individual. Instead, **we select only a small behavioural subset, the individual words or tokens, and imagine that they represent well the overall emotional behaviour of the individual**.

Most of the emotional responses of humans are simple and stereotypical. This means that, if we can learn what signals are produced by humans in response to a given emotional status, this same signal will likely be used in future situations, where the same emotional status will be experienced. This is because words, the verbal signals, don't change their meaning frequently or rapidly.

The reason why natural languages generally stay unchanged for decades is that they map well the relationship between syntactic value and semantic meaning of words. While words may change their meaning, they do so slowly enough that their capacity to map internal states of the individuals doesn't change in the short term.

Furthermore, while humans have the capacity to reassign meaning to words "on-the-go", they generally don't do that often. If a word is used in association with a given emotion in one context, that word will generally be associated with the same emotion in any other context.

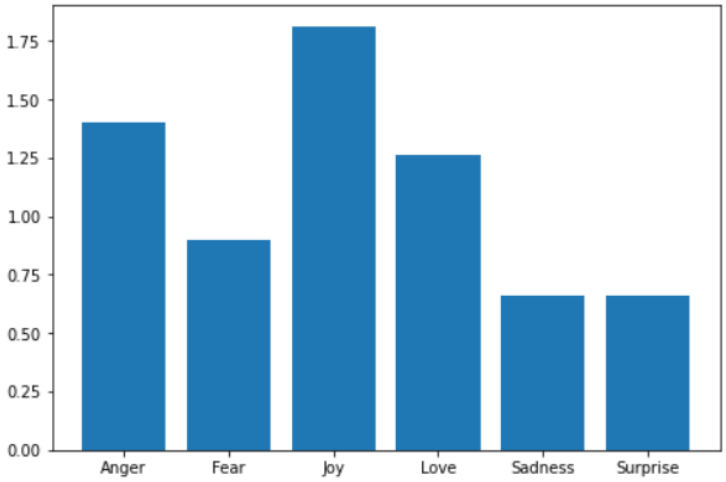
Problem Definition

Our problem statement is to analyze the 6 major emotions hidden in a text, plot their relative percentages, and predict the overall dominating emotion in that text. This will be achieved by training 16000 statements, each followed by its annotated emotion, in English, implementing Artificial Neural Networks and providing various words/tokens acting as features, with weights using suitable text processing techniques.

For Example, a sample problem and it's analysis:

```
para="Wow man! I am so surprised"
predict(para)
```

```
tensor([[-1.5948, -2.1019, -1.1845, -1.7394, -2.3363, -2.3371]])
```




```
tensor(2)
Emotion: joy
```

As clearly perceived, relative presence of each emotion is displayed in this given text, and 'Joy' is predicted as the dominating emotion.

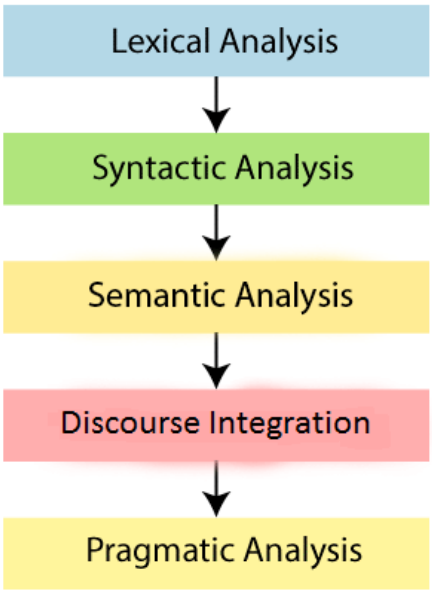
Literature Survey

Understanding **Natural Language Processing**

Natural Language Processing, usually shortened as NLP, is a branch of artificial intelligence that deals with the interaction between computers and humans using the natural language.

The ultimate objective of NLP is to read, decipher, understand, and make sense of the human languages in a manner that is valuable.

There are the following five phases of NLP:



Following are the steps to build an NLP pipeline:

Step1: Sentence Segmentation

Sentence Segment is the first step for building the NLP pipeline. It breaks the paragraph into separate sentences.

Example: Consider the following paragraph -

Independence Day is one of the important festivals for every Indian citizen. It is celebrated on the 15th of August each year ever since India got independence from the British rule. The day celebrates independence in the true sense.

Sentence Segment produces the following result:

1. "Independence Day is one of the important festivals for every Indian citizen."
2. "It is celebrated on the 15th of August each year ever since India got independence from the British rule."
3. "This day celebrates independence in the true sense."

Step2: Remove Punctuation and transform sentence to Lower Case

Punctuation removal and lowering the case is the second step for building the NLP pipeline. It removes all the various punctuations in a sentence, using python string libraries.

Example: Consider the following paragraph -

“OMG! This was just an amazing experience! What’s your opinion? Were you there @the location?”

This produces the following result:

"omg this was just amazing experience whats your opinion were you there the location"

Step3: Word Tokenization

Word Tokenizer is used to break the sentence into separate words or tokens.

Example: Consider the following -

“omg this was just amazing experience whats your opinion were you there the location”

This produces the following result:

[“omg”,”this”,”was”,”just”,”amazing”,”experience”,”whats”,”your”,”opinion”,”were”,”you”,”there”,”the”,”location”]

Step4: StopWords Removal

In English, there are a lot of words that appear very frequently like "is", "and", "the", and "a". NLP pipelines will flag these words as stop words. **Stop words** might be filtered out before doing any statistical analysis.

Example: Consider the following -

[“omg”,”this”,”was”,”just”,”amazing”,”experience”,”whats”,”your”,”opinion”,”were”,”you”,”there”,”the”,”location”]

This produces the following result:

[“omg”,”amazing”,”experience”,”whats”,”opinion”,”location”]

Step5: Lemmatization and Stemming

Stemming is used to normalize words into its base form or root form. For example, celebrates, celebrated and celebrating, all these words are originated with a single root word "celebrate." The big problem with stemming is that sometimes it produces the root word which may not have any meaning.

Lemmatization is quite similar to the Stemming. It is used to group different inflected forms of the word, called Lemma. The main difference between Stemming and lemmatization is that it produces the root word, which has a meaning.

Example: Consider the following -

[“omg”,”amazing”,”experience”,”whats”,”opinion”,”location”]

This produces the following result:

[“omg”,”amaz”,”experi”,”what”,”opinion”,”locat”]

Step6: TF-IDF Vectorization (Term Frequency-Inverse Document Frequency)

TF-IDF stands for Term Frequency Inverse Document Frequency of records. It can be defined as the calculation of how relevant a word in a series or corpus is to a text. The meaning increases proportionally to the number of times in the text a word appears but is compensated by the word frequency in the corpus (data-set).

Terminologies:

- **Term Frequency:** In document d, the frequency represents the number of instances of a given word t. Therefore, we can see that it becomes more relevant when a word appears in the text, which is rational. Since the ordering of terms is not significant, we can use a vector to describe the text in the bag of term models. For each specific term in the paper, there is an entry with the value being the term frequency.

The weight of a term that occurs in a document is simply proportional to the term frequency.

tf(t,d) = count of t in d / number of words in d

- **Document Frequency:** This tests the meaning of the text, which is very similar to TF, in the whole corpus collection. The only difference is that in document d, TF is the frequency counter for a term t, while df is the number of occurrences in the document set N of the term t. In other words, the number of papers in which the word is present is DF.

df(t) = occurrence of t in documents

- **Inverse Document Frequency:** Mainly, it tests how relevant the word is. The key aim of the search is to locate the appropriate records that fit the demand. Since tf considers all terms equally significant, it is therefore not only possible to use the term frequencies to measure the weight of the term in the paper. First, find the document frequency of a term t by counting the number of documents containing the term:

df(t) = N(t)

where

df(t) = Document frequency of a term t

N(t) = Number of documents containing the term t

Term frequency is the number of instances of a term in a single document only; although the frequency of the document is the number of separate documents in which the term appears, it depends on the entire corpus. Now let's look at the definition of the frequency of the inverse paper. The IDF of the word is the number of documents in the corpus separated by the frequency of the text.

idf(t) = N/ df(t) = N/N(t)

The more common word is supposed to be considered less significant, but the element (most definite integers) seems too harsh. We then take the logarithm (with base 2) of

the inverse frequency of the paper. So the if of the term t becomes:

idf(t) = log(N/ df(t))

- **Computation:** Tf-idf is one of the best metrics to determine how significant a term is to a text in a series or a corpus. tf-idf is a weighting system that assigns a weight to each word in a document based on its term frequency (tf) and the reciprocal document frequency (tf) (idf). The words with higher scores of weight are deemed to be more significant.

Usually, the tf-idf weight consists of two terms-

1. **Normalized Term Frequency (tf)**
2. **Inverse Document Frequency (idf)**

tf-idf(t, d) = tf(t, d) * idf(t)

Example: Consider the following -

[“om shanti om”,”om”,”film”]

This produces the following result:

Tf-idf in matrix form:

[[0.54935123,0.83559154,0.]
 [0. , 1. , 0.]
 [0. , 0. , 1.]]

PyTorch Tensor

A torch.Tensor is a multi-dimensional matrix containing elements of a single data type.

A PyTorch Tensor is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic n-dimensional array to be used for arbitrary numeric computation.

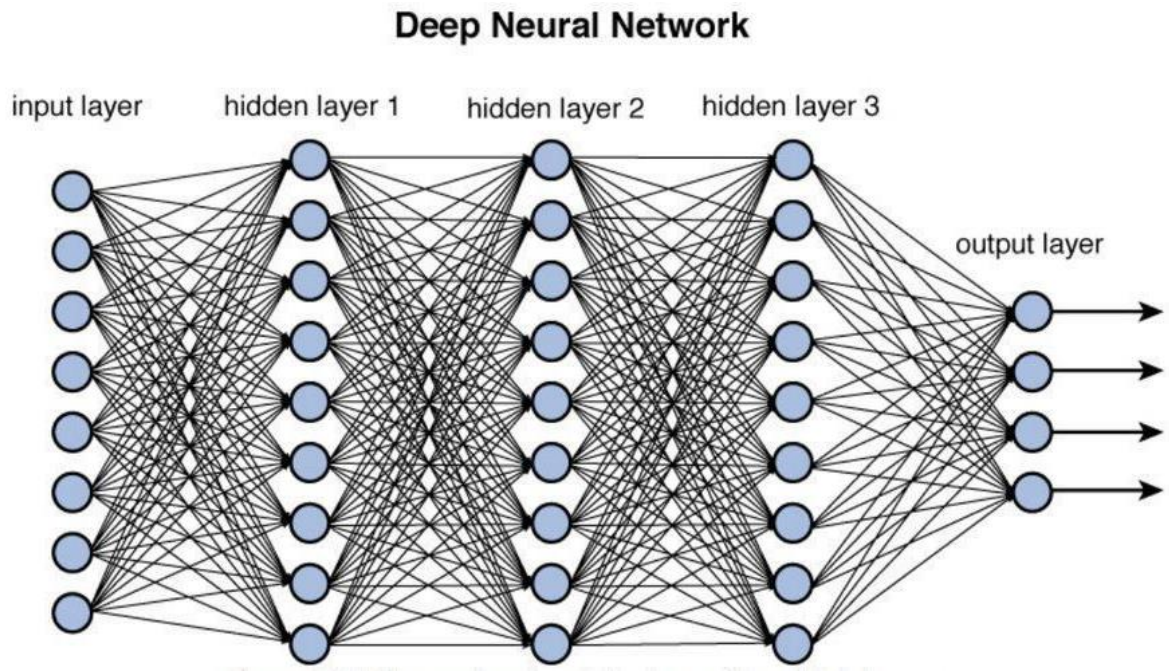
The biggest difference between a numpy array and a PyTorch Tensor is that a PyTorch Tensor can run on either CPU or GPU. To run operations on the GPU, just cast the Tensor to a CUDA datatype.

```
>>> torch.tensor([[1., -1.], [1., -1.]])
tensor([[ 1.0000, -1.0000],
        [ 1.0000, -1.0000]])
```

Neural Networks

Neural networks reflect the behaviour of the human brain, allowing computer programs to recognize patterns and solve common problems in the fields of AI, machine learning, and deep learning. Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.



Neural networks can be constructed using the `torch.nn` package.

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: `weight = weight - learning_rate * gradient`

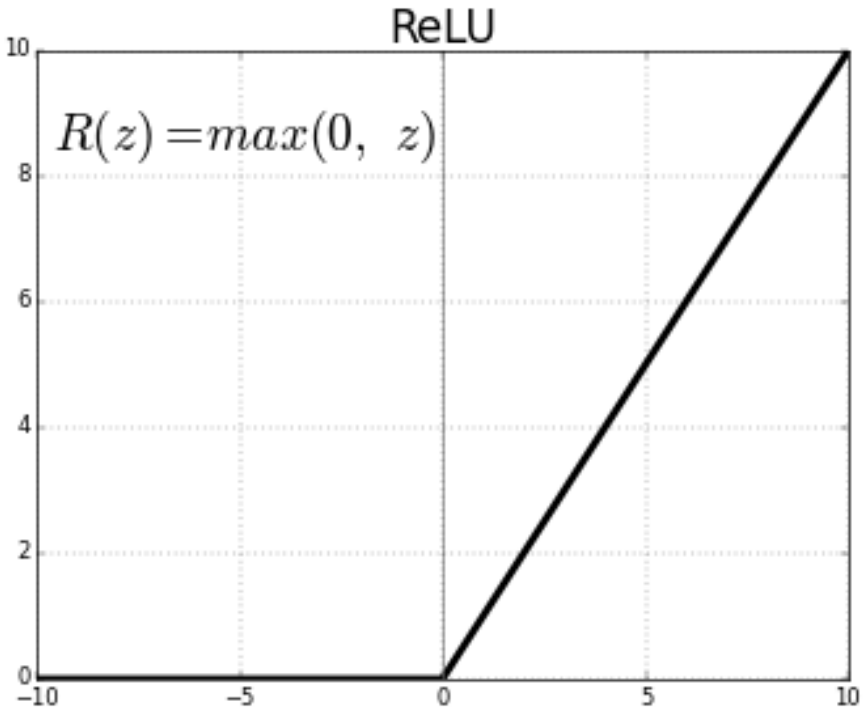
Rectified Linear Activation Function

In order to use stochastic gradient descent with backpropagation of errors to train deep neural networks, an activation function is needed that looks and acts like a linear function, but is, in fact, a nonlinear function allowing complex relationships in the data to be learned.
The function must also provide more sensitivity to the activation sum input and avoid easy saturation.

The solution had been bouncing around in the field for some time, although was not highlighted until papers in 2009 and 2011 shone a light on it.

The solution is to use the rectified linear activation function, or ReL for short.

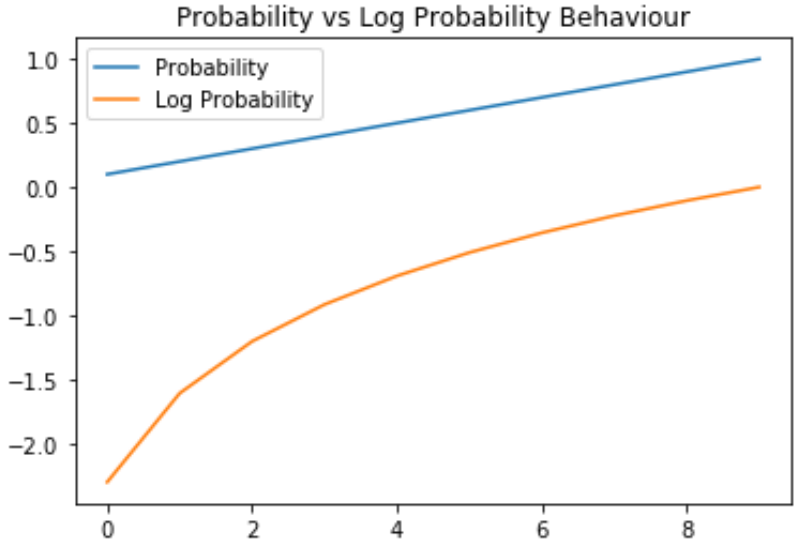
A node or unit that implements this activation function is referred to as a **rectified linear activation unit**, or ReLU for short. Often, networks that use the rectifier function for the hidden layers are referred to as rectified networks.
Adoption of ReLU may easily be considered one of the few milestones in the deep learning revolution, e.g. the techniques that now permit the routine development of very deep neural networks.



Log Softmax

Output will be measured as probability distribution. The mathematical function to be used here is log_softmax function in PyTorch.

$$LogSoftmax(x)_i = log(\frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}})$$



Implementation

Importing necessary packages:

```
import nltk
import torch
import matplotlib.pyplot as plt
import math
import pickle
```

```
[ ] nltk.download('stopwords')
```

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

```
[ ] from nltk.corpus import stopwords
```

```
import torch.nn as nn
import torch.nn.functional as F
```

Reading Training Set as a file from Google Drive:

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

```
Mounted at /content/drive
```

```
path = "/content/drive/MyDrive/NLP_Project/train.txt"
with open(path, 'r') as file:
  f=file.read()
```

```
[ ] f
```

Splitting the file into different sentences and storing them in a separate python list:

```
[ ] sentences=f.split('\n')
    sentences
```

Converting the sentences into Lower Case and store it in a separate python list:

```
lower_sent=[]
for sent in sentences:
  sent=sent.lower()
  lower_sent.append(sent)
lower_sent
```

Now, each sentence has a comment and it's annotated emotion, separated by ';', a semicolon. This is split into 2 different items and stored in different python lists, storing comment in one and its respective emotion in another list.

```
[11] comment=[]
     emotion=[]
     for sent in lower_sent:
       s=sent.split(';')
       comment.append(s[0])
       emotion.append(s[1])
```

```
[12] comment
```

```
['i didnt feel humiliated',
 'i can go from feeling so hopeless to so damned hopeful just from being around someone who cares and is awake',
 'im grabbing a minute to post i feel greedy wrong',
 'i am ever feeling nostalgic about the fireplace i will know that it is still on the property',
 'i am feeling grouchy',
 'ive been feeling a little burdened lately wasnt sure why that was',
```

Tokenizing each sentence in its individual words and storing them in a python list:

```
[13] comm_words=[]
     for comm in comment:
       comm_words.append(comm.split())
     comm_words
```

```
 ['i',
  'feel',
  'like',
  'not',
  'enough',
  'people',
  'my',
  'age',
  'actually',
  'think',
  'that',
  'most',
  'are',
  'pretty',
  'devastated',
  'that',
  'their',
  's',
  'have',
  'come',
  'and',
  'gone'],
 ['i',
  'get',
  'home',
  'i',
  'laze',
  'around',
```

Cleaning the text and removing all the stop words:

```
from nltk.corpus import stopwords
```

```
cleaned_text=[]
for comm in comm_words:
  words=[]
  for w in comm:
    if w not in set(stopwords.words('english')):
      words.append(w)
  cleaned_text.append(words)
cleaned_text
```

```
[['didnt', 'feel', 'humiliated'],
 ['go',
  'feeling',
  'hopeless',
  'damned',
  'hopeful',
  'around',
  'someone',
  'cares',
  'awake'],
 ['im', 'grabbing', 'minute', 'post', 'feel', 'greedy', 'wrong'],
 ['ever', 'feeling', 'nostalgic', 'fireplace', 'know', 'still', 'property'],
 ['feeling', 'grouchy'],
 ['ive', 'feeling', 'little', 'burdened', 'lately', 'wasnt', 'sure'],
 ['ive',
  'taking',
  'milligrams',
  'times',
  'recommended',
  'amount',
  'ive',
  'fallen',
  'asleep',
  'lot',
  'faster',
  'also',
  'feel',
  'like',
  'funny'],
 ['feel', 'confused', 'life', 'teenager', 'jaded', 'year', 'old', 'man'],
```

Performing Lemmatization and Stemming on obtained words:

```
[15] nltk.download('wordnet')
```

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

```
from nltk.stem import WordNetLemmatizer
lemmatizer=WordNetLemmatizer()
lemm_text=[]
for text in cleaned_text:
  words=[]
  for w in text:
    words.append(lemmatizer.lemmatize(w))
  lemm_text.append(words)
lemm_text
```

```
[16] ['set',
      'mind',
      'wanting',
      'specific',
      'item',
      'needing',
      'specific',
      'event',
      'specific',
      'time',
      'find',
      'ill',
      'end',
      'spending',
      'want',
      'feel',
      'pressured',
      'constraint'],
     ['written',
      'prayer',
      'journal',
      'morning',
      'meditating',
      'greatness',
      'lord',
      'psalm',
      'written',
      'closing',
      'may',
      'feel',
      'tender',
      'care',
      'today'],
     ['depressed', 'actually', 'feeling', 'inspired'],
```

```
from nltk.stem import PorterStemmer
stemmer=PorterStemmer()
stem_text=[]
for text in lemm_text:
  words=''
  for w in text:
    words=words+stemmer.stem(w)+' '
  stem_text.append(words)
stem_text
```

```
['didnt feel humili ',
 'go feel hopeless damn hope around someon care awak ',
 'im grab minut post feel greedi wrong ',
 'ever feel nostalg fireplac know still properti ',
 'feel grouchi ',
 'ive feel littl burden late wasnt sure ',
 'ive take milligram time recommend amount ive fallen asleep lot faster also feel like funni ',
 'feel confus life teenag jade year old man ',
 'petrona year feel petrona perform well made huge profit ',
 'feel romant ',
 'feel like make suffer see mean someth ',
 'feel run divin experi expect type spiritu encount ',
 'think easiest time year feel dissatisfi ',
 'feel low energi thirsti ',
 'immens sympathi gener point possibl proto writer tri find time write corner life sign agent let alon publish contract feel littl preciou
 'feel reassur anxieti side ',
 'didnt realli feel embarrass ',
 'feel pretti pathet time ',
 'start feel sentiment doll child began collect vintag barbi doll sixti ',
 'feel compromis skeptic valu everi unit work put ',
 'feel irrit reject without anyon anyth say anyth ',
 'feel complet overwhelm two strategi help feel ground pour heart journal form letter god end list five thing grate ',
 'feel amus delight ',
```

Viewing the major emotions annotated in training set sentences:

```
[18] emotion_set=set(emotion)
```

```
[19] emotion_set
```

```
{'anger', 'fear', 'joy', 'love', 'sadness', 'surprise'}
```

Saving emotions in a python dictionary to be used in future, in emotion prediction:

```
emotion_dict={'anger':0, 'fear':1, 'joy':2, 'love':3, 'sadness':4, 'surprise':5}
emo_y=[]
for em in emotion:
  emo_y.append(emotion_dict[em])
emo_y
```

```
1,
4,
2,
4,
0,
2,
2,
4,
2,
3,
2,
2,
2,
0,
1,
4,
```

Importing TfidfVectorizer from scikit_learn package and transforming stemmed sentences into tf-idf feature list:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer=TfidfVectorizer()
X=vectorizer.fit_transform(stem_text)
print(vectorizer.get_feature_names())
```

```
['aa', 'aaaaaaand', 'aaaaand', 'aaaand', 'aac', 'aahhh', 'aaron', 'ab', 'abandon', 'abat
```

Converting Feature list to tf-idf matrix and further transforming numpy array to tf-idf torch.tensor tensors:

```
[22] train_x=X.toarray()
```

```
[23] train_x
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

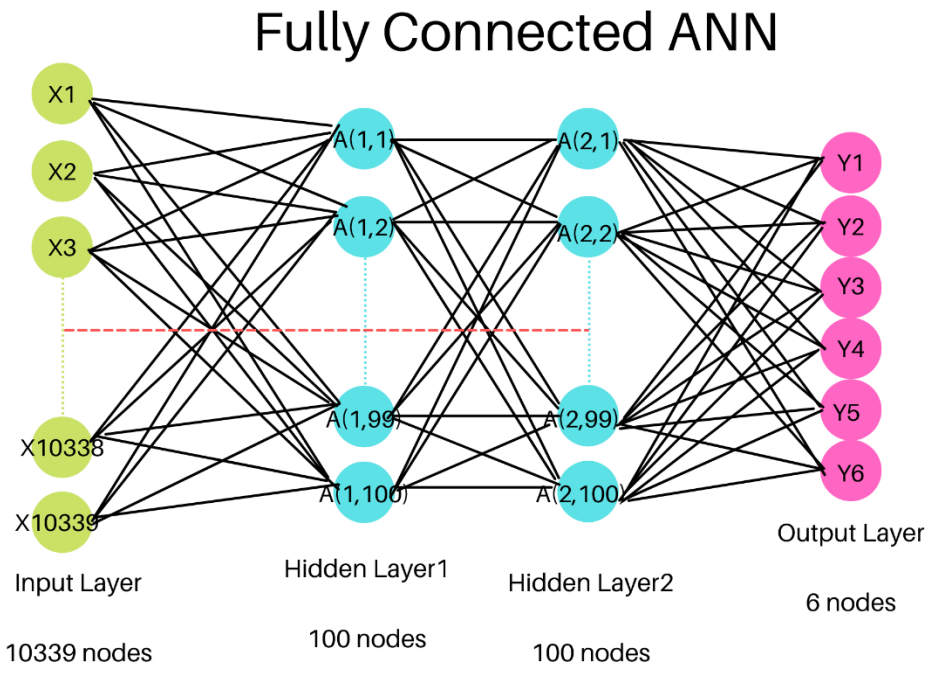
```
train_x.shape
```

```
(16000, 10339)
```

```
[24] train_tensor=torch.Tensor(train_x)
     train_tensor
```

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
```

Constructing a 3-layered Fully Connected Neural Network with 2 hidden using torch.nn classes:




```
[27] class Net2(nn.Module):
         def __init__(self):
             super().__init__()#initialize nn.Module
             self.fc1=nn.Linear(10339,100)#nn.Linear() for fully connected Layers.
             self.fc2=nn.Linear(100,100)
             self.fc3=nn.Linear(100,6)
         def forward(self,x):
             x=F.relu(self.fc1(x))#relu() is an activation function
             x=F.relu(self.fc2(x))
             # x=F.relu(self.fc3(x))
             x=self.fc3(x)
             return F.log_softmax(x,dim=1)#log_softmax() for probability distribution

     net=Net2()
     print(net)
```

```
Net2(
  (fc1): Linear(in_features=10339, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=6, bias=True)
)
```

Converting emotion list to tensor:

```
y=torch.tensor(emo_y,dtype=torch.long)
y
```

```
tensor([4, 4, 0,  ..., 2, 0, 4])
```

Optimizing the model by Backward Propagation:

```
import torch.optim as optim
optimizer=optim.Adam(net.parameters(),lr=0.001)

EPOCHS=100

for epoch in range(EPOCHS):
  if epoch%10==0:
    net.zero_grad()
  output=net(train_tensor)
  loss=F.nll_loss(output,y)
  loss.backward()
  optimizer.step()
  if epoch%10==0:
    print(loss)
  # if loss<0.6:
  #   break
```

```
tensor(1.8103, grad_fn=<NllLossBackward>)
tensor(1.7632, grad_fn=<NllLossBackward>)
tensor(1.6918, grad_fn=<NllLossBackward>)
tensor(1.5833, grad_fn=<NllLossBackward>)
tensor(1.4683, grad_fn=<NllLossBackward>)
tensor(1.3519, grad_fn=<NllLossBackward>)
tensor(1.2003, grad_fn=<NllLossBackward>)
tensor(1.0133, grad_fn=<NllLossBackward>)
tensor(0.8288, grad_fn=<NllLossBackward>)
tensor(0.6671, grad_fn=<NllLossBackward>)
```

Evaluating Training Set Accuracy:

```
correct=0
total=0

with torch.no_grad():
  output=net(train_tensor)
  for idx,i in enumerate(output):
    if torch.argmax(i)==y[idx]:#torch.argmax() returns original value given to log_softmax()
      correct+=1
    total+=1

print("Accuracy: ",round(correct/total,3))
print(correct,total,sep='  ')
```

```
Accuracy:  0.86
13767  16000
```

Clubbing text preprocessing together to evaluate test and validation datasets:

```
def preprocess(f):
  sentences=f.split('\n')
  sentences.pop()
  lower_sent=[]
  for sent in sentences:
    sent=sent.lower()
    lower_sent.append(sent)
  comment=[]
  emotion=[]
  for sent in lower_sent:
    s=sent.split(';')
    comment.append(s[0])
    emotion.append(s[1])
  punct_sent=[]
  for comm in comment:
    for w in comm:
      if w in punctuation:
        comm=comm.replace(w,'')
    punct_sent.append(comm)
  comm_words=[]
  for comm in punct_sent:
    comm_words.append(comm.split())
  cleaned_text=[]
  for comm in comm_words:
    words=[]
    for w in comm:
      if w not in set(stopwords.words('english')):
        words.append(w)
    cleaned_text.append(words)
  # from nltk.stem import WordNetLemmatizer
  lemmatizer=WordNetLemmatizer()
  lemm_text=[]
```

```
for text in cleaned_text:
  words=[]
  for w in text:
    words.append(lemmatizer.lemmatize(w))
  lemm_text.append(words)
# from nltk.stem import PorterStemmer
stemmer=PorterStemmer()
stem_text=[]
for text in lemm_text:
  words=''
  for w in text:
    words=words+stemmer.stem(w)+' '
  stem_text.append(words)
num_rows=len(stem_text)
emotion_set=set(emotion)
emotion_dict={'anger':0, 'fear':1, 'joy':2, 'love':3, 'sadness':4, 'surprise':5}
emo_y=[]
for em in emotion:
  emo_y.append(emotion_dict[em])
y=torch.tensor(emo_y,dtype=torch.long)
vectorizer=TfidfVectorizer()
X=vectorizer.fit_transform(stem_text)
print(vectorizer.get_feature_names())
feature_arr=vectorizer.get_feature_names()
print(len(vectorizer.get_feature_names()))
tfidf_matrix=vectorizer.fit_transform(stem_text)
# print(tfidf_matrix)
# return feature_arr
test_tens=torch.zeros([num_rows,10339])
```

```
for doc in range(num_rows):
  feature_index = tfidf_matrix[doc,:].nonzero()[1]
  tfidf_scores = zip(feature_index, [tfidf_matrix[doc, x] for x in feature_index])
  for w, s in [(feature_arr[i], s) for (i, s) in tfidf_scores]:
    # print(w, s,sep="   ")
    try:
      test_tens[doc][feature_indices[w]]=s
    except:
      continue
# return feature_arr,X.toarray()
train_x=X.toarray()
train_tensor=torch.Tensor(train_x)
# return test_tens,train_tensor
# # net=make_model(train_tensor[0].shape[0])
correct=0
total=0

with torch.no_grad():
  output=net(test_tens)
  for idx,i in enumerate(output):
    if torch.argmax(i)==y[idx]:#torch.argmax() returns original value given to log_softmax()
      correct+=1
    total+=1

print("Accuracy: ",round(correct/total,3))
print("Correct Predictions: {}  Total: {}".format(correct,total))
```

Evaluating Test Set Accuracy:

```
[33] with open("/content/drive/MyDrive/NLP_Project/test.txt",'r') as file:
       f=file.read()
       preprocess(f)
     # train_features,X_train=preprocess(f)
```

```
['aaaah', 'abandon', 'abba', 'abil', 'abit', 'abl', 'ablo', 'abou', 'absenc', 'absolout', 'abs
3549
Accuracy:  0.681
Correct Predictions: 1363  Total: 2000
```

Evaluating Validation Set Accuracy:

```
with open("/content/drive/MyDrive/NLP_Project/val.txt",'r') as file:
  f=file.read()
  preprocess(f)
# train_features,X_train=preprocess(f)
```

```
['abandon', 'abil', 'abit', 'abl', 'absolut', 'absorpt', 'absurd', 'abus', 'abyss', 'ac
3578
Accuracy:  0.682
Correct Predictions: 1365  Total: 2000
```

Clubbing Text Preprocessing functions to predict emotions on User input text:

```
def predict(f):
  sentences=f.split('\n')
  lower_sent=[]
  for sent in sentences:
    sent=sent.lower()
    lower_sent.append(sent)
  punct_sent=[]
  for comm in lower_sent:
    for w in comm:
      if w in punctuation:
        comm=comm.replace(w,'')
    punct_sent.append(comm)
  comm_words=[]
```

```
for comm in punct_sent:
  comm_words.append(comm.split())
cleaned_text=[]
for comm in comm_words:
  words=[]
  for w in comm:
    if w not in set(stopwords.words('english')):
      words.append(w)
  cleaned_text.append(words)
# from nltk.stem import WordNetLemmatizer
lemmatizer=WordNetLemmatizer()
lemm_text=[]
for text in cleaned_text:
  words=[]
  for w in text:
    words.append(lemmatizer.lemmatize(w))
  lemm_text.append(words)
# from nltk.stem import PorterStemmer
stemmer=PorterStemmer()
stem_text=[]
for text in lemm_text:
  words=''
  for w in text:
    words=words+stemmer.stem(w)+' '
  stem_text.append(words)
num_rows=len(stem_text)
emotion_dict={'anger':0, 'fear':1, 'joy':2, 'love':3, 'sadness':4, 'surprise':5}
vectorizer=TfidfVectorizer()
X=vectorizer.fit_transform(stem_text)
# print(vectorizer.get_feature_names())
feature_arr=vectorizer.get_feature_names()
# print(len(vectorizer.get_feature_names()))
tfidf_matrix=vectorizer.fit_transform(stem_text)
```

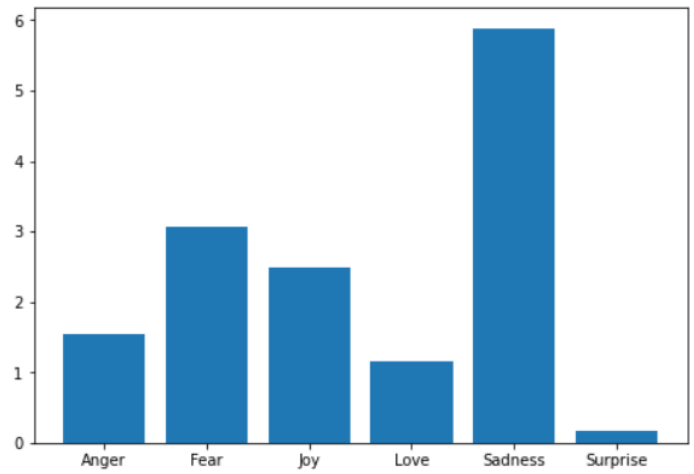
```
test_tens=torch.zeros([num_rows,10339])
for doc in range(num_rows):
  feature_index = tfidf_matrix[doc,:].nonzero()[1]
  tfidf_scores = zip(feature_index, [tfidf_matrix[doc, x] for x in feature_index])
  for w, s in [(feature_arr[i], s) for (i, s) in tfidf_scores]:
    # print(w, s,sep="   ")
    try:
      test_tens[doc][feature_indices[w]]=s
    except:
      continue
# return feature_arr,X.toarray()
train_x=X.toarray()
train_tensor=torch.Tensor(train_x)
```

```
from math import ceil
with torch.no_grad():
  output=net(test_tens)
  print(output)
  measure=[]
  val=[]
  for x in output[0]:
    val.append(abs(x.item()))
  for x in output[0]:
    # measure.append((sum-abs(x.item()))*100/sum)
    measure.append(ceil(max(val))-abs(x.item()))
  fig = plt.figure()
  ax = fig.add_axes([0,0,1,1])
  emot = ['Anger', 'Fear', 'Joy', 'Love', 'Sadness','Surprise']
  ax.bar(emot,measure)
  plt.show()
  for idx,i in enumerate(output):
    p=torch.argmax(i)
    print(p)
    for k in emotion_dict.keys():
      if emotion_dict[k]==p:
        print("Emotion: {}".format(k))
        break
```

Predicting Emotions on a random statement and plotting matplotlib.pyplot graph depicting relative percentage of each emotion in a particular statement:

```
para="How can i be so stupid! I don't know what to do, i am so sooooo confused. Oh go help me pls!"
predict(para)
```

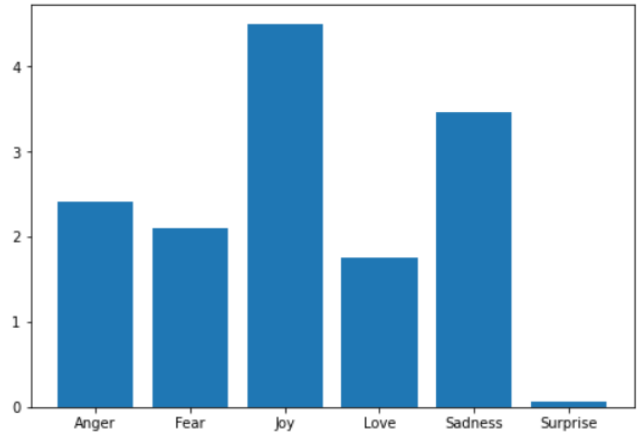
```
tensor([[-4.4587, -2.9417, -3.5173, -4.8419, -0.1108, -5.8312]])
```




```
tensor(4)
Emotion: sadness
```

```
para="I am so elated after hearing this news. This is  really something else, but something so beautiful!"
predict(para)
```

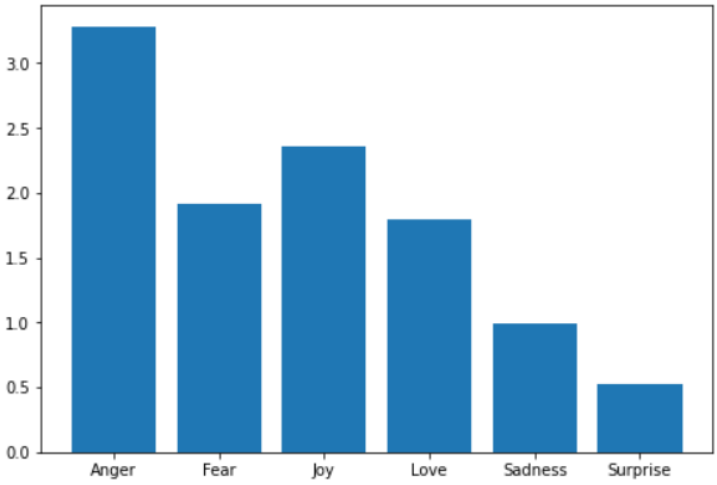
```
tensor([[-2.5940, -2.8934, -0.4951, -3.2541, -1.5392, -4.9360]])
```




```
tensor(2)
Emotion: joy
```

```
para="My frustration is on its peak. I feel like beating him so hard"
predict(para)
```

```
tensor([[-0.7133, -2.0816, -1.6366, -2.2056, -3.0054, -3.4764]])
```




```
tensor(0)
Emotion: anger
```

Conclusion

We have successfully trained a **3-layer deep fully connected Artificial Neural Network** using PyTorch classes and helper functions and optimizing the **loss to around 0.6 or 66%**, providing us with a **Training set accuracy of 0.86 or 86%**. The model correctly predicts 13767 statements' emotions out of 16000 total statements.

The **test set and validation set accuracy** obtained is as **0.681 or 68.1% and 0.682 or 68.2%** respectively.

Outputs are predicted in the form of **log-SoftMax probability distribution** of all the 6 emotions, and the statement is finally annotated with the **emotion having the best argmax() value**.

Our observations regarding test and validation set accuracies conclude that the **model has also undergone some degree of Overfitting,** due to overtraining of the model, unbalanced data with uneven count of emotion-annotated statements.

This problem of overfitting can be easily rectified and reduced by following approaches:

1. Cross-validation.
2. Training with more data.
3. Training with properly balanced data.
4. Removing features.
5. Early stopping.
6. Regularization.
7. Ensembling.

References

1. https://pytorch.org/
2. http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
3. https://www.kaggle.com/praveengovi/emotions-dataset-for-nlp?select=train.txt

Certifications:

https://www.udemy.com/certificate/UC-5dd381b9-560a-4a0f-b86d-5a441b05f003/