Lab Guide

Natural Language Processing Labs

December 10, 2020

Natural Language Processing Labs

Lab 1: Text Classification using Word Embeddings.

- *Learn and Practice the code for text classification using Word Embeddings.*

Lab 1: Input Code

Notebook Link - https://colab.research.google.com/drive/1CmpASpjS0uXfugmSa5RYmG7BdC4dVq_t?usp=sharing

Dataset - https://drive.google.com/file/d/1oJaLF27NsmjdZhuqkiTy-50mwMkHCHn0/view?usp=sharing

Word Embeddings - https://drive.google.com/file/d/1piCBW3pbxn9HBWq8CYd7jI9Gf1Y6lU7K/view?usp=sharing

```
from google.colab import drive
drive.mount('/content/drive')
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM,Dense,Bidirectional,Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
from string import punctuation
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from tqdm import tqdm
tqdm.pandas()
df = pd.read_csv('/content/drive/MyDrive/Datasets/Quora Text
Classification Data.csv')
df.head()

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

```
nltk.download('wordnet')
stop_words = stopwords.words('english')+list(punctuation)
lem = WordNetLemmatizer()
def cleaning(text):
  text = text.lower()
  words = word_tokenize(text)
  words = [w for w in words if w not in stop_words]
  words = [lem.lemmatize(w) for w in words]
  return ' '.join(words)
df['Clean Text'] = df['question_text'].progress_apply(cleaning)
!unzip '/content/drive/MyDrive/Word Embeddings/glove.42B.300d.zip'
embedding_values = {}
f = open('/content/glove.42B.300d.txt')
for line in tqdm(f):
  value = line.split(' ')
  word = value[0]
  coef = np.array(value[1],dtype = "float32")
  if coef is not None:
    embedding_values[word] = coef
tokenizer = Tokenizer()
x = df['Clean Text']
y = df['target']

tokenizer.fit_on_texts(x)


seq = tokenizer.texts_to_sequences(x)
pad_seq = pad_sequences(seq,maxlen = 300)

vocab_size = len(tokenizer.word_index)+1
print(vocab_size)
embedding_matrix = np.zeros((vocab_size,300))
for word, i in tqdm(tokenizer.word_index.items()):
  value = embedding_values.get(word)
  if value is not None:
    embedding_matrix[i] = value
model = Sequential()
model.add(Embedding(vocab_size,300,input_length=300,weights =
[embedding_matrix],trainable = False))
model.add(LSTM(50,return_sequences=False))
```

```
model.add(Dense(128,activation ='relu'))
model.add(Dense(1,activation= 'sigmoid'))
model.compile(optimizer = 'adam',loss='binary_crossentropy',metrics =
['accuracy'])

history = model.fit(pad_seq,y,validation_split=0.2,epochs = 5)

train_acc = history.history['accuracy']
train_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
epochs = range(1,6)

plt.plot(epochs,train_acc,label = 'Train Accuracy')
plt.plot(epochs,val_acc,label = 'Validation Accuracy')
plt.legend()
plt.show()

plt.plot(epochs,train_loss,label = 'Train Loss')
plt.plot(epochs,val_loss,label = 'Validation Loss')
plt.legend()
plt.show()
```

Lab 2: Find Synonyms and antonyms using Word Embeddings.

- *Learn and Practice the code for finding Synonyms and Antonyms using Word Embeddings.*

Lab 2: Input Code

Notebook Link - https://colab.research.google.com/drive/1LjeE3wvyF5jp7bgaVo9FJ9gBAId9cN-Y?usp=sharing

```
import gensim.downloader
```

```
import warnings
warnings.filterwarnings('ignore')
# Show all available models in gensim-data
print(list(gensim.downloader.info()['models'].keys()))
word2vec = gensim.downloader.load('word2vec-google-news-300')
word2vec.most_similar('technology')
word2vec.most_similar('Science')
word2vec.most_similar('arts')
word2vec.similarity('hot', 'cold')
```

Lab 3: Introduction to Topic Modelling.

- *Introduction to Topic Modelling.*

Lab 3: Input Code

Notebook Link - https://colab.research.google.com/drive/13BaKT5bw4slZ9nVInt--4UckpD9BlQNE?usp=sharing

```
import pandas as pd
import re
import gensim
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from string import punctuation
from gensim.corpora import Dictionary
from nltk.tokenize import word_tokenize
from gensim.models.ldamodel import LdaModel,CoherenceModel
import pyLDAvis
import pyLDAvis.gensim
import matplotlib.pyplot as plt
```

```
%matplotlib inline
df =
pd.read_json('https://raw.githubusercontent.com/selva86/datasets/master/ne
wsgroups.json')
df.head()
def removing_email(text):
    text = re.sub('\S*@\S*\s',' ',text)
    return text
def only_words(text):
    text = re.sub('\W+',' ',text)
    return text
stop_words =
list(set(stopwords.words('english')))+list(punctuation)+['\n','----','----
\n\n\n\n\n']
lem = WordNetLemmatizer()
def cleaning(text):
    text = text.lower()
    words = word_tokenize(text)
    words = [w for w in words if w not in stop_words]
    words = [w for w in words if len(w)>=3]
    lemma = [lem.lemmatize(w,'v') for w in words]
    return lemma
df['without email'] = df['content'].apply(removing_email)
df['only words'] = df['without email'].apply(only_words)
df['clean content'] = df['only words'].apply(cleaning)
df.head()
clean_doc = list(df['clean content'].values)
dictionary = Dictionary(clean_doc)
corpus = [dictionary.doc2bow(doc) for doc in clean_doc]
ldamodel =
LdaModel(corpus=corpus,id2word=dictionary,num_topics=5,random_state=42,upd
ate_every=1,passes=50, chunksize=100)
print(ldamodel.print_topics())
print(ldamodel.log_perplexity(corpus))
coherence =
CoherenceModel(ldamodel,texts=clean_doc,dictionary=dictionary,coherence='c
_v')
coherence.get_coherence()
```

```
coherence =
CoherenceModel(ldamodel,texts=clean_doc,dictionary=dictionary,coherence='u
_mass')
coherence.get_coherence()
pyLDAvis.enable_notebook()
vis = pyLDAvis.gensim.prepare(ldamodel, corpus, dictionary)
vis
```

Lab 4: Converting a Foreign Language to English using Machine Translation(German to English).

- *Learning Machine Translation.*

Lab 4: Input Code

Notebook Link -
https://colab.research.google.com/drive/1ZHK8LoC9Y3gIRFQoqUxH3-Lja7nfhvA0?usp=sharing

Dataset Link -
English Text Corpus - https://nlp.stanford.edu/projects/nmt/data/wmt14.en-de/train.en
German Text Corpus - https://nlp.stanford.edu/projects/nmt/data/wmt14.en-de/train.de
English Vocabulary - https://nlp.stanford.edu/projects/nmt/data/wmt14.en-de/vocab.50K.en
German Vocabulary - https://nlp.stanford.edu/projects/nmt/data/wmt14.en-de/vocab.50K.de
English Word Embeddings -
https://github.com/thushv89/exercises_thushv_dot_com/blob/master/en-embeddings.npy
German Word Embeddings -
https://github.com/thushv89/exercises_thushv_dot_com/blob/master/de-embeddings.npy

```
%matplotlib inline
import math
```

```
import numpy as np
import os
import random
import tensorflow as tf
from matplotlib import pylab
from collections import Counter
import csv

# Seq2Seq Items
import tensorflow.contrib.seq2seq as seq2seq
from tensorflow.python.ops.rnn_cell import LSTMCell
from tensorflow.python.ops.rnn_cell import MultiRNNCell
from tensorflow.contrib.seq2seq.python.ops import attention_wrapper
from tensorflow.python.layers.core import Dense
vocab_size= 50000
num_units = 128
input_size = 128
batch_size = 16
source_sequence_length=40
target_sequence_length=60
decoder_type = 'basic' # could be basic or attention
sentences_to_read = 50000
src_dictionary = dict()
with open('vocab.50K.de.txt', encoding='utf-8') as f:
    for line in f:
        #we are discarding last char as it is new line char
        src_dictionary[line[:-1]] = len(src_dictionary)

src_reverse_dictionary =
dict(zip(src_dictionary.values(),src_dictionary.keys()))

print('Source')
print('\t',list(src_dictionary.items())[:10])
print('\t',list(src_reverse_dictionary.items())[:10])
print('\t','Vocabulary size: ', len(src_dictionary))

tgt_dictionary = dict()
with open('vocab.50K.en.txt', encoding='utf-8') as f:
    for line in f:
        #we are discarding last char as it is new line char
```

```
        tgt_dictionary[line[:-1]] = len(tgt_dictionary)

tgt_reverse_dictionary =
dict(zip(tgt_dictionary.values(),tgt_dictionary.keys()))

print('Target')
print('\t',list(tgt_dictionary.items())[:10])
print('\t',list(tgt_reverse_dictionary.items())[:10])
print('\t','Vocabulary size: ', len(tgt_dictionary))

source_sent = []
target_sent = []

test_source_sent = []
test_target_sent = []


with open('train.de', encoding='utf-8') as f:
    for l_i, line in enumerate(f):
        # discarding first 20 translations as there was some
        # english to english translations found in the first few. which
are wrong
        if l_i<50:
            continue
        source_sent.append(line)
        if len(source_sent)>=sentences_to_read:
            break


with open('train.en', encoding='utf-8') as f:
    for l_i, line in enumerate(f):
        if l_i<50:
            continue

        target_sent.append(line)
        if len(target_sent)>=sentences_to_read:
            break
```

```
assert len(source_sent)==len(target_sent),'Source: %d, Target:
%d'%(len(source_sent),len(target_sent))

print('Sample translations (%d)'%len(source_sent))
for i in range(0,sentences_to_read,10000):
    print('(',i,') DE: ', source_sent[i])
    print('(',i,') EN: ', target_sent[i])

def split_to_tokens(sent,is_source):
    #sent = sent.replace('-',' ')
    sent = sent.replace(',',' ,')
    sent = sent.replace('.',' .')
    sent = sent.replace('\n',' ')

    sent_toks = sent.split(' ')
    for t_i, tok in enumerate(sent_toks):
        if is_source:
            if tok not in src_dictionary.keys():
                sent_toks[t_i] = '<unk>'
        else:
            if tok not in tgt_dictionary.keys():
                sent_toks[t_i] = '<unk>'
    return sent_toks

# Let us first look at some statistics of the sentences
source_len = []
source_mean, source_std = 0,0
for sent in source_sent:
    source_len.append(len(split_to_tokens(sent,True)))

print('(Source) Sentence mean length: ', np.mean(source_len))
print('(Source) Sentence stddev length: ', np.std(source_len))

target_len = []
target_mean, target_std = 0,0
for sent in target_sent:
    target_len.append(len(split_to_tokens(sent,False)))

print('(Target) Sentence mean length: ', np.mean(target_len))
print('(Target) Sentence stddev length: ', np.std(target_len))
```

```
train_inputs = []
train_outputs = []
train_inp_lengths = []
train_out_lengths = []

max_tgt_sent_lengths = 0

src_max_sent_length = 41
tgt_max_sent_length = 61
for s_i, (src_sent, tgt_sent) in enumerate(zip(source_sent,target_sent)):

    src_sent_tokens = split_to_tokens(src_sent,True)
    tgt_sent_tokens = split_to_tokens(tgt_sent,False)

    num_src_sent = []
    for tok in src_sent_tokens:
        num_src_sent.append(src_dictionary[tok])

    num_src_set = num_src_sent[::-1] # we reverse the source sentence.
This improves performance
    num_src_sent.insert(0,src_dictionary['<s>'])
    train_inp_lengths.append(min(len(num_src_sent)+1,src_max_sent_length))

    # append until the sentence reaches max length
    if len(num_src_sent)<src_max_sent_length:
        num_src_sent.extend([src_dictionary['</s>'] for _ in
range(src_max_sent_length - len(num_src_sent))])
    # if more than max length, truncate the sentence
    elif len(num_src_sent)>src_max_sent_length:
        num_src_sent = num_src_sent[:src_max_sent_length]
    assert len(num_src_sent)==src_max_sent_length,len(num_src_sent)

    train_inputs.append(num_src_sent)

    num_tgt_sent = [tgt_dictionary['</s>']]
    for tok in tgt_sent_tokens:
        num_tgt_sent.append(tgt_dictionary[tok])

    train_out_lengths.append(min(len(num_tgt_sent)+1,tgt_max_sent_length))
```

```
    if len(num_tgt_sent)<tgt_max_sent_length:
        num_tgt_sent.extend([tgt_dictionary['</s>'] for _ in
range(tgt_max_sent_length - len(num_tgt_sent))])
    elif len(num_tgt_sent)>tgt_max_sent_length:
        num_tgt_sent = num_tgt_sent[:tgt_max_sent_length]

    train_outputs.append(num_tgt_sent)
    assert len(train_outputs[s_i])==tgt_max_sent_length, 'Sent length
needs to be 60, but is %d'%len(binned_outputs[s_i])

assert len(train_inputs)  == len(source_sent),\
        'Size of total bin elements: %d, Total sentences: %d'\
                %(len(train_inputs),len(source_sent))

print('Max sent lengths: ', max_tgt_sent_lengths)


train_inputs = np.array(train_inputs, dtype=np.int32)
train_outputs = np.array(train_outputs, dtype=np.int32)
train_inp_lengths = np.array(train_inp_lengths, dtype=np.int32)
train_out_lengths = np.array(train_out_lengths, dtype=np.int32)
print('Samples from bin')
print('\t',[src_reverse_dictionary[w]  for w in
train_inputs[0,:].tolist()])
print('\t',[tgt_reverse_dictionary[w]  for w in
train_outputs[0,:].tolist()])
print('\t',[src_reverse_dictionary[w]  for w in
train_inputs[10,:].tolist()])
print('\t',[tgt_reverse_dictionary[w]  for w in
train_outputs[10,:].tolist()])
print()
print('\tSentences ',train_inputs.shape[0])
input_size = 128

class DataGeneratorMT(object):

    def __init__(self,batch_size,num_unroll,is_source):
        self._batch_size = batch_size
        self._num_unroll = num_unroll
        self._cursor = [0 for offset in range(self._batch_size)]
```

```
        self._src_word_embeddings = np.load('de-embeddings.npy')

        self._tgt_word_embeddings = np.load('en-embeddings.npy')

        self._sent_ids = None

        self._is_source = is_source


    def next_batch(self, sent_ids, first_set):

        if self._is_source:
            max_sent_length = src_max_sent_length
        else:
            max_sent_length = tgt_max_sent_length
        batch_labels_ind = []
        batch_data = np.zeros((self._batch_size),dtype=np.float32)
        batch_labels = np.zeros((self._batch_size),dtype=np.float32)

        for b in range(self._batch_size):

            sent_id = sent_ids[b]

            if self._is_source:
                sent_text = train_inputs[sent_id]

                batch_data[b] = sent_text[self._cursor[b]]
                batch_labels[b]=sent_text[self._cursor[b]+1]

            else:
                sent_text = train_outputs[sent_id]

                batch_data[b] = sent_text[self._cursor[b]]
                batch_labels[b] = sent_text[self._cursor[b]+1]

            self._cursor[b] = (self._cursor[b]+1)%(max_sent_length-1)

        return batch_data,batch_labels
```

```
    def unroll_batches(self,sent_ids):

        if sent_ids is not None:

            self._sent_ids = sent_ids

            self._cursor = [0 for _ in range(self._batch_size)]

        unroll_data,unroll_labels = [],[]
        inp_lengths = None
        for ui in range(self._num_unroll):

            data, labels = self.next_batch(self._sent_ids, False)

            unroll_data.append(data)
            unroll_labels.append(labels)
            inp_lengths = train_inp_lengths[sent_ids]
        return unroll_data, unroll_labels, self._sent_ids, inp_lengths

    def reset_indices(self):
        self._cursor = [0 for offset in range(self._batch_size)]

# Running a tiny set to see if the implementation correct
dg = DataGeneratorMT(batch_size=5,num_unroll=40,is_source=True)
u_data, u_labels, _, _ = dg.unroll_batches([0,1,2,3,4])

print('Source data')
for _, lbl in zip(u_data,u_labels):
    print([src_reverse_dictionary[w] for w in lbl.tolist()])


# Running a tiny set to see if the implementation correct
dg = DataGeneratorMT(batch_size=5,num_unroll=60,is_source=False)
u_data, u_labels, _, _ = dg.unroll_batches([0,2,3,4,5])
print('\nTarget data batch (first time)')
for d_i,(_, lbl) in enumerate(zip(u_data,u_labels)):
    #if d_i>5 and d_i < 35:
    #    continue
```

```
    print([tgt_reverse_dictionary[w] for w in lbl.tolist()])

print('\nTarget data batch (non-first time)')
u_data, u_labels, _, _ = dg.unroll_batches(None)
for d_i,(_, lbl) in enumerate(zip(u_data,u_labels)):

    #if d_i>5 and d_i < 35:
    #    continue

    print([tgt_reverse_dictionary[w] for w in lbl.tolist()])

tf.reset_default_graph()

enc_train_inputs = []
dec_train_inputs = []

# Need to use pre-trained word embeddings
encoder_emb_layer = tf.convert_to_tensor(np.load('de-embeddings.npy'))
decoder_emb_layer = tf.convert_to_tensor(np.load('en-embeddings.npy'))

# Defining unrolled training inputs
for ui in range(source_sequence_length):
    enc_train_inputs.append(tf.placeholder(tf.int32,
shape=[batch_size],name='enc_train_inputs_%d'%ui))

dec_train_labels=[]
dec_label_masks = []
for ui in range(target_sequence_length):
    dec_train_inputs.append(tf.placeholder(tf.int32,
shape=[batch_size],name='dec_train_inputs_%d'%ui))
    dec_train_labels.append(tf.placeholder(tf.int32,
shape=[batch_size],name='dec-train_outputs_%d'%ui))
    dec_label_masks.append(tf.placeholder(tf.float32,
shape=[batch_size],name='dec-label_masks_%d'%ui))

encoder_emb_inp = [tf.nn.embedding_lookup(encoder_emb_layer, src) for src
in enc_train_inputs]
encoder_emb_inp = tf.stack(encoder_emb_inp)
```

```
decoder_emb_inp = [tf.nn.embedding_lookup(decoder_emb_layer, src) for src
in dec_train_inputs]
decoder_emb_inp = tf.stack(decoder_emb_inp)

enc_train_inp_lengths = tf.placeholder(tf.int32,
shape=[batch_size],name='train_input_lengths')
dec_train_inp_lengths = tf.placeholder(tf.int32,
shape=[batch_size],name='train_output_lengths')
encoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

initial_state = encoder_cell.zero_state(batch_size, dtype=tf.float32)

encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
    encoder_cell, encoder_emb_inp, initial_state=initial_state,
    sequence_length=enc_train_inp_lengths,
    time_major=True, swap_memory=True)
# Build RNN cell
decoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

projection_layer = Dense(units=vocab_size, use_bias=True)

# Helper
helper = tf.contrib.seq2seq.TrainingHelper(
    decoder_emb_inp, [tgt_max_sent_length-1 for _ in range(batch_size)],
time_major=True)

# Decoder
if decoder_type == 'basic':
    decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)

elif decoder_type == 'attention':
    decoder = tf.contrib.seq2seq.BahdanauAttention(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)

# Dynamic decoding
outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(
    decoder, output_time_major=True,
```

```
    swap_memory=True
)


logits = outputs.rnn_output

crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=dec_train_labels, logits=logits)
loss = (tf.reduce_sum(crossent*tf.stack(dec_label_masks)) /
(batch_size*target_sequence_length))

train_prediction = outputs.sample_id
print('Defining Optimizer')
# Adam Optimizer. And gradient clipping.
global_step = tf.Variable(0, trainable=False)
inc_gstep = tf.assign(global_step,global_step + 1)
learning_rate = tf.train.exponential_decay(
    0.01, global_step, decay_steps=10, decay_rate=0.9, staircase=True)

with tf.variable_scope('Adam'):
    adam_optimizer = tf.train.AdamOptimizer(learning_rate)

adam_gradients, v = zip(*adam_optimizer.compute_gradients(loss))
adam_gradients, _ = tf.clip_by_global_norm(adam_gradients, 25.0)
adam_optimize = adam_optimizer.apply_gradients(zip(adam_gradients, v))

with tf.variable_scope('SGD'):
    sgd_optimizer = tf.train.GradientDescentOptimizer(learning_rate)

sgd_gradients, v = zip(*sgd_optimizer.compute_gradients(loss))
sgd_gradients, _ = tf.clip_by_global_norm(sgd_gradients, 25.0)
sgd_optimize = sgd_optimizer.apply_gradients(zip(sgd_gradients, v))

sess = tf.InteractiveSession()


if not os.path.exists('logs'):
    os.mkdir('logs')
log_dir = 'logs'
```

```
bleu_scores_over_time = []
loss_over_time = []
tf.global_variables_initializer().run()

src_word_embeddings = np.load('de-embeddings.npy')
tgt_word_embeddings = np.load('en-embeddings.npy')

# Defining data generators
enc_data_generator =
DataGeneratorMT(batch_size=batch_size,num_unroll=source_sequence_length,is
_source=True)
dec_data_generator =
DataGeneratorMT(batch_size=batch_size,num_unroll=target_sequence_length,is
_source=False)

num_steps = 10001
avg_loss = 0

bleu_labels, bleu_preds = [],[]

print('Started Training')

for step in range(num_steps):

    # input_sizes for each bin: [40]
    # output_sizes for each bin: [60]
    print('.',end='')
    if (step+1)%100==0:
        print('')

    sent_ids =
np.random.randint(low=0,high=train_inputs.shape[0],size=(batch_size))
    # ====================== ENCODER DATA COLLECTION
==================================================

    eu_data, eu_labels, _, eu_lengths =
enc_data_generator.unroll_batches(sent_ids=sent_ids)

    feed_dict = {}
    feed_dict[enc_train_inp_lengths] = eu_lengths
```

```
    for ui,(dat,lbl) in enumerate(zip(eu_data,eu_labels)):
        feed_dict[enc_train_inputs[ui]] = dat

    # ====================== DECODER DATA COLLECITON
==========================
    # First step we change the ids in a batch
    du_data, du_labels, _, du_lengths =
dec_data_generator.unroll_batches(sent_ids=sent_ids)

    feed_dict[dec_train_inp_lengths] = du_lengths
    for ui,(dat,lbl) in enumerate(zip(du_data,du_labels)):
        feed_dict[dec_train_inputs[ui]] = dat
        feed_dict[dec_train_labels[ui]] = lbl
        feed_dict[dec_label_masks[ui]] = (np.array([ui for _ in
range(batch_size)])<du_lengths).astype(np.int32)

    # ====================== OPTIMIZATION ==========================
    if step < 10000:
        _,l,tr_pred = sess.run([adam_optimize,loss,train_prediction],
feed_dict=feed_dict)
    else:
        _,l,tr_pred = sess.run([sgd_optimize,loss,train_prediction],
feed_dict=feed_dict)
    tr_pred = tr_pred.flatten()


    if (step+1)%250==0:

        print('Step ',step+1)

        print_str = 'Actual: '
        for w in np.concatenate(du_labels,axis=0)[::batch_size].tolist():
            print_str += tgt_reverse_dictionary[w] + ' '
            if tgt_reverse_dictionary[w] == '</s>':
                break

        print(print_str)
        print()

        print_str = 'Predicted: '
```

```
        for w in tr_pred[::batch_size].tolist():
            print_str += tgt_reverse_dictionary[w] + ' '
            if tgt_reverse_dictionary[w] == '</s>':
                break
        print(print_str)

        print('\n')

        rand_idx = np.random.randint(low=1,high=batch_size)
        print_str = 'Actual: '
        for w in
np.concatenate(du_labels,axis=0)[rand_idx::batch_size].tolist():
            print_str += tgt_reverse_dictionary[w] + ' '
            if tgt_reverse_dictionary[w] == '</s>':
                break
        print(print_str)


        print()
        print_str = 'Predicted: '
        for w in tr_pred[rand_idx::batch_size].tolist():
            print_str += tgt_reverse_dictionary[w] + ' '
            if tgt_reverse_dictionary[w] == '</s>':
                break
        print(print_str)
        print()

    avg_loss += l

    #sess.run(reset_train_state) # resetting hidden state for each batch

    if (step+1)%500==0:
        print('============= Step ', str(step+1), ' =============')
        print('\t Loss: ',avg_loss/500.0)

        loss_over_time.append(avg_loss/500.0)

        avg_loss = 0.0
        sess.run(inc_gstep)
```

Lab 5: Twitter Sentiment Analysis.

- *Performing Sentiment Analysis on Twitter Dataset.*

Lab 5: Input Code

Notebook Link - https://colab.research.google.com/drive/1lxDe9lteF3UihKiSBFezW7KCY2EyaSBB?usp=sharing
Dataset - https://drive.google.com/file/d/1GbgYsudN9EkbzOdwjO9eu2yQlwj2azvM/view?usp=sharing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import tensorflow as tf
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from gensim.models import word2vec
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LogisticRegression
dd =
pd.read_csv('/content/training.1600000.processed.noemoticon.csv',encoding
= "ISO-8859-1")
dd.columns = ['sentiment','id','date','query','special','text']
dd.head()
dd.drop(['id','date','query','special'],axis = 1,inplace = True)
df = dd.sample(100000)
df['Cleaned'] = df['text'].str.replace('@','')
df['Cleaned'] = df['Cleaned'].str.replace(r'http\S+','')
df['Cleaned'] = df['Cleaned'].str.replace('[^a-zA-Z]',' ')
```

```
stopwords = stopwords.words('english')
def remove_stopwords(text):
    clean_text=' '.join([word for word in text.split() if word not in
stopwords])
    return clean_text
df['Cleaned'] = df['Cleaned'].apply(lambda text :
remove_stopwords(text.lower()))
df['Cleaned'] = df['Cleaned'].apply(lambda x : x.split())
df.head()
sns.countplot(df.sentiment)
wordnet=WordNetLemmatizer()
df['Cleaned'] = df['Cleaned'].apply(lambda x : [wordnet.lemmatize(i) for i
in x])
df['Cleaned'] = df['Cleaned'].apply(lambda x : ' '.join([w for w in x]))
df['Cleaned'] = df['Cleaned'].apply(lambda x : ' '.join([w for w in
x.split()]))
df.head()
cv = CountVectorizer(max_features = 2500)
x = cv.fit_transform(df['Cleaned']).toarray()
x.shape
x_train,x_test,y_train,y_test =
train_test_split(x,df['sentiment'],test_size = 0.2,random_state = 42)
%%time
model = RandomForestClassifier()
model.fit(x_train,y_train)
model.score(x_train,y_train)
model.score(x_test,y_test)
%%time
reg = LogisticRegression()
reg.fit(x_train,y_train)
reg.score(x_train,y_train)
reg.score(x_test,y_test)

tf = TfidfVectorizer(max_features = 2500)
z = tf.fit_transform(df['Cleaned']).toarray()
z.shape
z_train,z_test,y_train,y_test =
train_test_split(z,df['sentiment'],test_size = 0.2,random_state = 42)
%%time
model1 = RandomForestClassifier()
```

```
model1.fit(z_train,y_train)
model1.score(z_train,y_train)
model1.score(z_test,y_test)
%%time
reg1 = LogisticRegression()
reg1.fit(z_train,y_train)
reg1.score(z_train,y_train)
reg1.score(z_test,y_test)
scores = pd.DataFrame({'Bow(RF)': model.score(x_test,y_test),
                       'Bow(LR)': reg.score(x_test,y_test),
                       'TF(RF)': model1.score(z_test,y_test),
                       'TF(LR)': reg1.score(z_test,y_test)},
                      index = [0])
scores.T.plot(kind = 'bar')
```

Lab 6: Explaining Lemmatization, PoS Tagging, Stemming and Tokenization using an Example.

- *Explaining lemmatization, PoS tagging, Stemming and Tokenization.*

Lab 6: Input Code

Notebook Link - https://colab.research.google.com/drive/1TVJoJSRnhbTYvRgv55_k_hVneilbme-b?usp=sharing

```
sentence = 'My name is John and I am learning Natural Language Processing today.'
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer,PorterStemmer
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
words = word_tokenize(sentence)
```

```
print(words)
stemming = PorterStemmer()
sentences = [stemming.stem(w.lower()) for w in words]
print(sentences)
lem = WordNetLemmatizer()
lemmatized_words = [lem.lemmatize(w.lower(),'v') for w in words]
print(lemmatized_words)
print(nltk.pos_tag(words))
```

Lab 7: Understanding Dependency Parsing in a given sentence.

- *Understanding dependency parsing.*

Lab 7: Input Code

Notebook Link - https://colab.research.google.com/drive/1ybsFWB2SfuYKMrUW1LQArdl0LHp6qwzP?usp=sharing

```
import spacy
nlp = spacy.load('en_core_web_sm')

piano_text = 'Gus is learning piano'
piano_doc = nlp(piano_text)
for token in piano_doc:
  print (token.text, token.tag_, token.head.text, token.dep_)

from spacy import displacy
about_interest_text = ('He is interested in learning'' Natural Language Processing.')
about_interest_doc = nlp(about_interest_text)
displacy.render(about_interest_doc, style='dep', jupyter=True)
```

Lab 8: Perform Speech to Text Conversion using PyAudio and Google Speech Recognition.

- *Performing Speech to Text Conversion.*

Lab 8: Input Code

Notebook Link - https://colab.research.google.com/drive/1kJxBIsswieYjYqwUt7kR-PUkBAWn07jo?usp=sharing

```
import speech_recognition as sr
import pyttsx3

# Initialize the recognizer
r = sr.Recognizer()

# Function to convert text to
# speech
def SpeakText(command):

  # Initialize the engine
  engine = pyttsx3.init()
  engine.say(command)
  engine.runAndWait()


# Loop infinitely for user to
# speak

while(1):

  # Exception handling to handle
  # exceptions at the runtime
  try:

    # use the microphone as source for input.
    with sr.Microphone() as source2:
```

```
        # wait for a second to let the recognizer
        # adjust the energy threshold based on
        # the surrounding noise level
        r.adjust_for_ambient_noise(source2, duration=0.2)

        #listens for the user's input
        audio2 = r.listen(source2)

        # Using ggogle to recognize audio
        MyText = r.recognize_google(audio2)
        MyText = MyText.lower()

        print("Did you say "+MyText)
        SpeakText(MyText)

    except sr.RequestError as e:
      print("Could not request results; {0}".format(e))

    except sr.UnknownValueError:
      print("unknown error occured")
```

Lab 9: Creating Custom Speech Recognition Corpus.

- *Creating Custom Speech Recognition Dataset.*

Lab 9: Input Code

Notebook Link -
https://colab.research.google.com/drive/1exllvRotRSG4N665PwY9eKVthRdsuroR?usp=sharing

Sample Dataset Link -
https://drive.google.com/file/d/1fSrmrLjFNHNeIxWxAS2lE5cTCP8T7sjm/view?usp=sharing

Complete Dataset Link -
https://librivox.org/search?primary_key=0&search_category=language&search_page=1&search_form=get_results

```
from google.colab import drive
drive.mount('/content/drive')
!pip install librosa==0.6.0
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.style as ms
import librosa.display
ms.use('seaborn-muted')
%matplotlib inline
import IPython.display as ipd
import librosa
data, sr = librosa.load('/content/drive/MyDrive/Datasets/Music.mp3')
print(data.shape)
S = librosa.feature.melspectrogram(data,sr)
log_S = librosa.power_to_db(S,ref = np.max)
plt.plot(log_S)
plt.show()
chromagram = librosa.feature.chroma_stft(y = data,sr = sr)
plt.figure(figsize = (15,5))
librosa.display.specshow(chromagram,x_axis = 'time',y_axis = 'chroma')
onset_env = librosa.onset.onset_strength(data,sr=sr)
tempo = librosa.beat.tempo(onset_env,sr=sr)
tempo
y_harmonic,y_percussive = librosa.effects.hpss(data)
```

```
tempo, beats = librosa.beat.beat_track(y = y_percussive,sr=sr)
print(tempo)
print(beats)
```

Lab 10: Introduction to Dynamic Memory Network.

- *Performing Question Answering Task using Dynamic Memory Network.*

Lab 10: Input Code

Notebook Link - https://colab.research.google.com/drive/1Ce0DGjSSgvD5R9Un291GgsWyy1ebPPfA?usp=sharing

Dataset Link - https://github.com/SeanLee97/nlp_learning/tree/master/reading_comprehension/corpus/bAbI/en-10k

```
!pip3 install http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl
!pip3 install torchvision

import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.optim as optim
import torch.nn.functional as F
import nltk
import random
import numpy as np
from collections import Counter, OrderedDict
import nltk
from copy import deepcopy
import os
import re
import unicodedata
```

```
flatten = lambda l: [item for sublist in l for item in sublist]

from torch.nn.utils.rnn import PackedSequence, pack_padded_sequence
random.seed(1024)
USE_CUDA = torch.cuda.is_available()
gpus = [0]
torch.cuda.set_device(gpus[0])

FloatTensor = torch.cuda.FloatTensor if USE_CUDA else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if USE_CUDA else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if USE_CUDA else torch.ByteTensor
def getBatch(batch_size, train_data):
    random.shuffle(train_data)
    sindex=0
    eindex=batch_size
    while eindex < len(train_data):
        batch = train_data[sindex: eindex]
        temp = eindex
        eindex = eindex + batch_size
        sindex = temp
        yield batch

    if eindex >= len(train_data):
        batch = train_data[sindex:]
        yield batch
def pad_to_batch(batch, w_to_ix):
    fact,q,a = list(zip(*batch))
    max_fact = max([len(f) for f in fact])
    max_len = max([f.size(1) for f in flatten(fact)])
    max_q = max([qq.size(1) for qq in q])
    max_a = max([aa.size(1) for aa in a])

    facts, fact_masks, q_p, a_p = [], [], [], []
    for i in range(len(batch)):
        fact_p_t = []
        for j in range(len(fact[i])):
            if fact[i][j].size(1) < max_len:
                fact_p_t.append(torch.cat([fact[i][j],
Variable(LongTensor([w_to_ix['<PAD>']] * (max_len -
fact[i][j].size(1)))).view(1, -1)], 1))
```

```
            else:
                fact_p_t.append(fact[i][j])

        while len(fact_p_t) < max_fact:
            fact_p_t.append(Variable(LongTensor([w_to_ix['<PAD>']] *
max_len)).view(1, -1))

        fact_p_t = torch.cat(fact_p_t)
        facts.append(fact_p_t)
        fact_masks.append(torch.cat([Variable(ByteTensor(tuple(map(lambda
s: s ==0, t.data))), volatile=False) for t in
fact_p_t]).view(fact_p_t.size(0), -1))

        if q[i].size(1) < max_q:
            q_p.append(torch.cat([q[i],
Variable(LongTensor([w_to_ix['<PAD>']] * (max_q - q[i].size(1)))).view(1,
-1)], 1))
        else:
            q_p.append(q[i])

        if a[i].size(1) < max_a:
            a_p.append(torch.cat([a[i],
Variable(LongTensor([w_to_ix['<PAD>']] * (max_a - a[i].size(1)))).view(1,
-1)], 1))
        else:
            a_p.append(a[i])

    questions = torch.cat(q_p)
    answers = torch.cat(a_p)
    question_masks = torch.cat([Variable(ByteTensor(tuple(map(lambda s: s
==0, t.data))), volatile=False) for t in
questions]).view(questions.size(0), -1)

    return facts, fact_masks, questions, question_masks, answers
def prepare_sequence(seq, to_index):
    idxs = list(map(lambda w: to_index[w] if to_index.get(w) is not None
else to_index["<UNK>"], seq))
    return Variable(LongTensor(idxs))
data = open('qa5_three-arg-relations_train.txt').readlines()
data = [d[:-1] for d in data]
```

```
train_data = []
fact=[]
qa=[]
for d in data:
    index=d.split(' ')[0]
    if(index=='1'):
        fact=[]
        qa=[]
    if('?' in d):
        temp = d.split('\t')
        ques = temp[0].strip().replace('?', '').split(' ')[1:] + ['?']
        ans=temp[1].split() + ['</s>']
        temp_s = deepcopy(fact)
        train_data.append([temp_s, ques, ans])
    else:
        fact.append(d.replace('.', '').split(' ')[1:] + ['</s>'])
fact,q,a = list(zip(*train_data))
vocab = list(set(flatten(flatten(fact)) + flatten(q) + flatten(a)))
word_to_index={'<PAD>': 0, '<UNK>': 1, '<s>': 2, '</s>': 3}
for vo in vocab:
    if word_to_index.get(vo) is None:
        word_to_index[vo] = len(word_to_index)
index_to_word = {v:k for k, v in word_to_index.items()}
for s in train_data:
    for i, fact in enumerate(s[0]):
        s[0][i] = prepare_sequence(fact, word_to_index).view(1, -1)
    s[1] = prepare_sequence(s[1], word_to_index).view(1, -1)
    s[2] = prepare_sequence(s[2], word_to_index).view(1, -1)
class DMN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
dropout_p=0.1):
        super(DMN, self).__init__()

        self.hidden_size=hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.fact_gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.ques_gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.attn_weights = nn.Sequential(nn.Linear(4*hidden_size,
hidden_size), nn.Tanh(), nn.Linear(hidden_size, 1), nn.Softmax())
```

```
        self.epsisodic_grucell = nn.GRUCell(hidden_size, hidden_size)
        self.memory_grucell = nn.GRUCell(hidden_size, hidden_size)
        self.ans_grucell = nn.GRUCell(2*hidden_size, hidden_size)

        self.ans_fc = nn.Linear(hidden_size, output_size)

        self.dropout = nn.Dropout(dropout_p)

    def init_hidden(self, inputs):
        hidden = Variable(torch.zeros(1, inputs.size(0),
self.hidden_size))
        return hidden.cuda() if USE_CUDA else hidden

    def init_weight(self):
        nn.init.xavier_uniform(self.embedding.state_dict()['weight'])

        for name, param in self.fact_gru.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.ques_gru.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.attn_weights.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.epsisodic_grucell.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.memory_grucell.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.ans_grucell.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)

        nn.init.xavier_normal(self.ans_fc.state_dict()['weight'])
        self.ans_fc.bias.data.fill_(0)

    def forward(self, facts, facts_masks, question, question_masks,
num_decode, episodes=3, is_training=True):
        #input module
        concated=[]
        for fact, fact_mask in zip(facts, facts_masks):
            embedded = self.embedding(fact)
            if(is_training):
```

```
                embedded = self.dropout(embedded)
            hidden = self.init_hidden(fact)
            output, hidden = self.fact_gru(embedded, hidden)
            hidden_real = []
            for i, o in enumerate(output):
                length = fact_mask[i].data.tolist().count(0)
                hidden_real.append(o[length-1])
            concated.append(torch.cat(hidden_real).view(fact.size(0),
-1).unsqueeze(0))
        encoded_facts = torch.cat(concated)
        #question module
        hidden=self.init_hidden(question)

        embedded = self.embedding(question)
        if(is_training):
                embedded = self.dropout(embedded)
        output, hidden = self.ques_gru(embedded, hidden)

        if is_training == True:
            real_question = []
            for i, o in enumerate(output): # B,T,D
                real_length = question_masks[i].data.tolist().count(0)

                real_question.append(o[real_length - 1])

            encoded_question =
torch.cat(real_question).view(questions.size(0), -1) # B,D
        else: # for inference mode
            encoded_question = hidden.squeeze(0) # B,D

        #episodic memory module

        memory = encoded_question
        T_C = encoded_facts.size(1)
        B = encoded_facts.size(0)
        for i in range(episodes):
            hidden = self.init_hidden(encoded_facts.transpose(0,
1)[0]).squeeze(0) # B,D
            for t in range(T_C):
```

```
                z = torch.cat([
                                        encoded_facts.transpose(0, 1)[t] *
encoded_question, # B,D , element-wise product
                                        encoded_facts.transpose(0, 1)[t] *
memory, # B,D , element-wise product

torch.abs(encoded_facts.transpose(0,1)[t] - encoded_question), # B,D

torch.abs(encoded_facts.transpose(0,1)[t] - memory) # B,D
                                    ], 1)
                g_t = self.attn_weights(z) # B,1 scalar
                hidden = g_t *
self.epsisodic_grucell(encoded_facts.transpose(0, 1)[t], hidden) + (1 -
g_t) * hidden

            e = hidden
            memory = self.memory_grucell(e, memory)

        # Answer Module
        answer_hidden = memory
        start_decode = Variable(LongTensor([[word_to_index['<s>']] *
memory.size(0)])).transpose(0, 1)
        y_t_1 = self.embedding(start_decode).squeeze(1) # B,D

        decodes = []
        for t in range(num_decode):
            answer_hidden = self.ans_grucell(torch.cat([y_t_1,
encoded_question], 1), answer_hidden)
            decodes.append(F.log_softmax(self.ans_fc(answer_hidden),1))
        return torch.cat(decodes, 1).view(B * num_decode, -1)
HIDDEN_SIZE = 80
BATCH_SIZE = 64
LR = 0.001
EPOCH = 50
NUM_EPISODE = 3
EARLY_STOPPING = False
model = DMN(len(word_to_index), HIDDEN_SIZE, len(word_to_index))
model.init_weight()
if USE_CUDA:
    model = model.cuda()
```

```
loss_function = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.Adam(model.parameters(), lr=LR)
 for epoch in range(EPOCH):
    losses = []
    if EARLY_STOPPING:
        break

    for i,batch in enumerate(getBatch(BATCH_SIZE, train_data)):
        facts, fact_masks, questions, question_masks, answers =
pad_to_batch(batch, word_to_index)

        model.zero_grad()
        pred = model(facts, fact_masks, questions, question_masks,
answers.size(1), NUM_EPISODE, True)
        loss = loss_function(pred, answers.view(-1))
        losses.append(loss.data.tolist()[0])

        loss.backward()
        optimizer.step()

        if i % 100 == 0:
            print("[%d/%d] mean_loss : %0.2f" %(epoch, EPOCH,
np.mean(losses)))

            if np.mean(losses) < 0.01:
                EARLY_STOPPING = True
                print("Early Stopping!")
                break
            losses = []
torch.save(model, 'DMN.pkl')
# Uncomment to load the existing model
# model = torch.load('DMN.pkl')
def pad_to_fact(fact, x_to_ix): # this is for inference

    max_x = max([s.size(1) for s in fact])
    x_p = []
    for i in range(len(fact)):
        if fact[i].size(1) < max_x:
```

```
            x_p.append(torch.cat([fact[i],
Variable(LongTensor([x_to_ix['<PAD>']] * (max_x -
fact[i].size(1)))).view(1, -1)], 1))
        else:
            x_p.append(fact[i])

    fact = torch.cat(x_p)
    fact_mask = torch.cat([Variable(ByteTensor(tuple(map(lambda s: s ==0,
t.data))), volatile=False) for t in fact]).view(fact.size(0), -1)
    return fact, fact_mask
data = open('qa5_three-arg-relations_test.txt').readlines()
data = [d[:-1] for d in data]
test_data = []
fact=[]
qa=[]
for d in data:
    index=d.split(' ')[0]
    if(index=='1'):
        fact=[]
        qa=[]
    if('?' in d):
        temp = d.split('\t')
        ques = temp[0].strip().replace('?', '').split(' ')[1:] + ['?']
        ans=temp[1].split() + ['</s>']
        temp_s = deepcopy(fact)
        test_data.append([temp_s, ques, ans])
    else:
        fact.append(d.replace('.', '').split(' ')[1:] + ['</s>'])
for t in test_data:
    for i, fact in enumerate(t[0]):
        t[0][i] = prepare_sequence(fact, word_to_index).view(1, -1)

    t[1] = prepare_sequence(t[1], word_to_index).view(1, -1)
    t[2] = prepare_sequence(t[2], word_to_index).view(1, -1)
accuracy = 0
for t in test_data:
    fact, fact_mask = pad_to_fact(t[0], word_to_index)
    question = t[1]
    question_mask = Variable(ByteTensor([0] * t[1].size(1)),
requires_grad=False).unsqueeze(0)
```

```
    answer = t[2].squeeze(0)

    model.zero_grad()
    pred = model([fact], [fact_mask], question, question_mask,
answer.size(0), NUM_EPISODE, False)
    if pred.max(1)[1].data.tolist() == answer.data.tolist():
        accuracy += 1

print(accuracy/len(test_data) * 100)
t = random.choice(test_data)
fact, fact_mask = pad_to_fact(t[0], word_to_index)
question = t[1]
question_mask = Variable(ByteTensor([0] * t[1].size(1)),
requires_grad=False).unsqueeze(0)
answer = t[2].squeeze(0)

model.zero_grad()
pred = model([fact], [fact_mask], question, question_mask, answer.size(0),
NUM_EPISODE, False)

print("Facts : ")
print('\n'.join([' '.join(list(map(lambda x: index_to_word[x],f))) for f
in fact.data.tolist()]))
print("")
print("Question : ",' '.join(list(map(lambda x: index_to_word[x],
question.data.tolist()[0]))))
print("")
print("Answer : ",' '.join(list(map(lambda x: index_to_word[x],
answer.data.tolist()))))
print("Prediction : ",' '.join(list(map(lambda x: index_to_word[x],
pred.max(1)[1].data.tolist()))))
```

Lab 11: Speech Recognition using Deep Learning.

- *Performing Speech Recognition using Deep Learning.*

Lab 11: Input Code

Notebook Link - https://colab.research.google.com/drive/1m8NI-zn_Y2nZO1MVR8Sp1E27Ekftg34b?usp=sharing

Dataset Link - https://drive.google.com/file/d/1wWsrN2Ep7x6lWqOXfr4rpKGYrJhWc8z7/view

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

import librosa
import soundfile
import os, glob, pickle
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

#DataFlair - Extract features (mfcc, chroma, mel) from a sound file
def extract_feature(file_name, mfcc, chroma, mel):
    with soundfile.SoundFile(file_name) as sound_file:
        X = sound_file.read(dtype="float32")
        sample_rate=sound_file.samplerate
        if chroma:
            stft=np.abs(librosa.stft(X))
        result=np.array([])
        if mfcc:
            mfccs=np.mean(librosa.feature.mfcc(y=X, sr=sample_rate,
n_mfcc=40).T, axis=0)
            result=np.hstack((result, mfccs))
        if chroma:
            chroma=np.mean(librosa.feature.chroma_stft(S=stft,
sr=sample_rate).T,axis=0)
            result=np.hstack((result, chroma))
if mel:
            mel=np.mean(librosa.feature.melspectrogram(X,
sr=sample_rate).T,axis=0)
            result=np.hstack((result, mel))
return result

#DataFlair - Emotions in the RAVDESS dataset
```

```
emotions={
  '01':'neutral',
  '02':'calm',
  '03':'happy',
  '04':'sad',
  '05':'angry',
  '06':'fearful',
  '07':'disgust',
  '08':'surprised'
}

#DataFlair - Emotions to observe
observed_emotions=['calm', 'happy', 'fearful', 'disgust']

#DataFlair - Load the data and extract features for each sound file
def load_data(test_size=0.2):
    x,y=[],[]
    for file in glob.glob("/content/drive/My Drive/Datasets/Voice
Samples/Actor_*/*.wav"):
        file_name=os.path.basename(file)
        emotion=emotions[file_name.split("-")[2]]
        if emotion not in observed_emotions:
            continue
        feature=extract_feature(file, mfcc=True, chroma=True, mel=True)
        x.append(feature)
        y.append(emotion)
    return train_test_split(np.array(x), y, test_size=test_size,
random_state=9)

#DataFlair - Split the dataset
x_train,x_test,y_train,y_test=load_data(test_size=0.25)

#DataFlair - Get the shape of the training and testing datasets
print((x_train.shape[0], x_test.shape[0]))

#DataFlair - Get the number of features extracted
print(f'Features extracted: {x_train.shape[1]}')

#DataFlair - Initialize the Multi Layer Perceptron Classifier
```

```
model=MLPClassifier(alpha=0.01, batch_size=256, epsilon=1e-08,
hidden_layer_sizes=(300,), learning_rate='adaptive', max_iter=500)

#DataFlair - Train the model
model.fit(x_train,y_train)

#DataFlair - Predict for the test set
y_pred=model.predict(x_test)

#DataFlair - Calculate the accuracy of our model
accuracy=accuracy_score(y_true=y_test, y_pred=y_pred)

#DataFlair - Print the accuracy
print("Accuracy: {:.2f}%".format(accuracy*100))
```

Lab 12: Dialog Generation using Deep Learning.

- *Performing Dialog Generation using Deep Learning.*

Lab 12: Input Code

Notebook Link - https://colab.research.google.com/drive/159zyaDB-7BLB5I94kICzFT4nMYPriGtP?usp=sharing

Dataset Link - https://drive.google.com/file/d/1wWsrN2Ep7x6lWqOXfr4rpKGYrJhWc8z7/view

```
from google.colab import drive
drive.mount('/content/drive')
import keras
import json
from datetime import datetime
```

```
import numpy as np
dialogues_path = "/content/drive/MyDrive/Datasets/movie_lines.txt"
VOCAB_SIZE = 5000 # len(keras_tokenizer.word_index) + 1
print(VOCAB_SIZE)
EMBEDDING_DIM = 500
from keras.preprocessing.text import Tokenizer
from statistics import median
EOS_TOKEN = "~e"
dialogue_lines = list()
with open(dialogues_path) as dialogues_file:
    for line in dialogues_file:
        line = line.strip().lower()
        split_line = line.split(' +++$+++ ')
        try:
            dialogue_lines.append(split_line[4] + " " + EOS_TOKEN)
        except IndexError:
            pass
#             print("Skipped line " + line)
dialogue_lines[:10]
keras_tokenizer = Tokenizer(num_words=VOCAB_SIZE,
filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}\t\n')
keras_tokenizer.fit_on_texts(dialogue_lines)
text_sequences = keras_tokenizer.texts_to_sequences(dialogue_lines)[:2000]
MAX_SEQUENCE_LENGTH = int(median(len(sequence) for sequence in
text_sequences))
print(MAX_SEQUENCE_LENGTH)
from keras import backend as K
from keras.engine.topology import Layer
from keras.layers import Input, Dense, RepeatVector, LSTM, Conv1D,
Masking, Embedding
from keras.layers.wrappers import TimeDistributed, Bidirectional
from keras.models import Model
from keras.preprocessing.sequence import pad_sequences
x_train = pad_sequences(text_sequences, maxlen=MAX_SEQUENCE_LENGTH,
padding='post',
                        truncating='post', value=0)
x_train.shape
x_train_rev = list()
for x_vector in x_train:
    x_rev_vector = list()
```

```
    for index in x_vector:
        char_vector = np.zeros(VOCAB_SIZE)
        char_vector[index] = 1
        x_rev_vector.append(char_vector)
    x_train_rev.append(np.asarray(x_rev_vector))
x_train_rev = np.asarray(x_train_rev)
x_train_rev.shape
def get_seq2seq_model():
    main_input = Input(shape=x_train[0].shape, dtype='float32',
name='main_input')
    print(main_input)

    embed_1 = Embedding(input_dim=VOCAB_SIZE, output_dim=EMBEDDING_DIM,
                        mask_zero=True, input_length=MAX_SEQUENCE_LENGTH)
(main_input)
    print(embed_1)

    lstm_1 = Bidirectional(LSTM(2048, name='lstm_1'))(embed_1)
    print(lstm_1)

    repeat_1 = RepeatVector(MAX_SEQUENCE_LENGTH, name='repeat_1')(lstm_1)
    print(repeat_1)

    lstm_3 = Bidirectional(LSTM(2048, return_sequences=True,
name='lstm_3'))(repeat_1)
    print(lstm_3)

    softmax_1 = TimeDistributed(Dense(VOCAB_SIZE,
activation='softmax'))(lstm_3)
    print(softmax_1)

    model = Model(main_input, softmax_1)
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model
seq2seq_model = get_seq2seq_model()
seq2seq_model.fit(x_train, x_train_rev, batch_size=128, epochs=20)
predictions = seq2seq_model.predict(x_train)
```

```
index2word_map = inv_map = {v: k for k, v in
keras_tokenizer.word_index.items()}
def sequence_to_str(sequence):
    word_list = list()
    for element in sequence:
#         if amax(element) < max_prob:
#             continue
        index = np.argmax(element) + 1
        word = index2word_map[index]
        word_list.append(word)

    return word_list
predictions_file_path = \
    "/content/" + datetime.now().strftime('%Y-%m-%d-%H-%M-%S') + ".txt"
with open(predictions_file_path, 'w') as predictions_file:
    for i in range(len(predictions)):
        predicted_word_list = sequence_to_str(predictions[i])
        actual_len = len(dialogue_lines[i])

        actual_sentence = "Actual: " +
dialogue_lines[i][:len(dialogue_lines[i])-3]

        generated_sentence = ""
        for word in predicted_word_list:
            if word == EOS_TOKEN:
                predictions_file.write('\n')
                break
            generated_sentence += word + " "

        sent_dict = dict()
        sent_dict["actual"] = actual_sentence.strip()
        sent_dict["generated"] = generated_sentence.strip()

        predictions_file.write(json.dumps(sent_dict, sort_keys=True,
indent=2, separators=(',', ': ')))
        predictions_file.write("\n")
```