

**NLP with Deep Learning**

**LAB FILE**

**Bachelor OF Technology**

**(Academic Session: 2019-2023)**

**Name of Student:** Divvela Vishnu Sai Kumar

**Enrollment No. :** A076119819005

**Branch:** B.Tech (AI) 7th Sem

**Course** **Name**: NLP with Deep learning

**Course Code**: CSE468

**Faculty Name:** Prof. Dr. Garima Srivastava

**Faculty** **Name**: Dr Pooja Khanna

1

**Department of Computer Science & Engineering**

**Amity School of Engineering & Technology**

**Amity University, Uttar Pradesh**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO.** | **Program Title** | **Date** | **Remarks** |
| 1 | Perform Classification with word vectors. |  |  |
| 2 | Implement Neural Network Bigram Model. |  |  |
| 3 | Implementation of word2vec using tensorflow. |  |  |
| 4 | Implement GloVe using tensorflow gradient descent and Alternating Least Squares. |  |  |
| 5 | Visualizing data with analogies with t-SNE. |  |  |
| 6 | Visualizing country analogies using embedding projectors |  |  |
| 7 | Perform Pointwise Mutual Information. |  |  |
| 8 | Implement Recursive Neural Tensor Network using tensorflow. |  |  |

**PRACTICAL-WORK**

Practical-1

OBJECTIVE:

Perform Classification with word vectors.

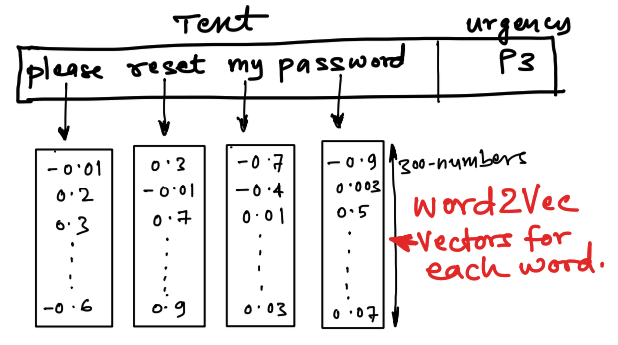
Tools Used:

* Python
* Vs code editor
* Tensorflow
* Jupyter Notebook

Description:

Word2vec is not a single algorithm but a combination of two techniques – **CBOW**(Continuous bag of words) and **Skip-gram** model. Both of these are shallow neural networks that map word(s) to the target variable which is also a word(s). Both of these techniques learn weights of the neural network which acts as word vector representations.

Basically each word is represented as a vector of numbers.



Sample word2vec vectors

**CBOW**

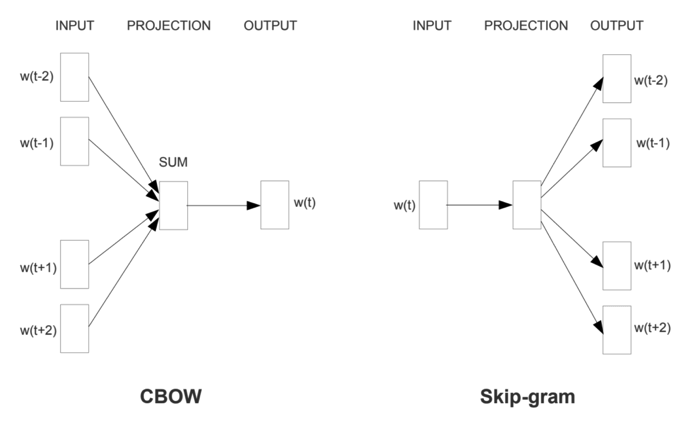
CBOW(Continuous bag of words) predicts the probability of a word to occur given the words surrounding it. We can consider a single word or a group of words.

**Skip-gram model**

The Skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the center word)

**Which one should be used?**

For a large corpus with higher dimensions, it is better to use skip-gram but it is slow to train. Whereas CBOW is better for small corpus and is faster to train too.



CBOW vs Skip-Gram

**Word2Vec** vectors are basically a form of word representation that bridges the human understanding of language to that of a machine.

Code:

# Read in the data and clean up column names

import gensim

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

pd.set\_option('display.max\_colwidth', 100)

messages = pd.read\_csv('supportTicketData.csv')

# messages = messages.drop(labels = ["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"], axis = 1)

messages.columns = ["text", "label"]

messages.head()

messages = messages[messages['label']!='P3']

# Clean data using the built in cleaner in gensim

messages['text\_clean'] = messages['text'].apply(lambda x: gensim.utils.simple\_preprocess(x))

messages.head()

# Encoding the label column

messages['label']=messages['label'].map({'P1':1,'P2':2})

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split (messages['text\_clean'], messages['label'] , test\_size=0.2)

# Train the word2vec model

w2v\_model = gensim.models.Word2Vec(X\_train,

                                   vector\_size=100,

                                   window=5,

                                   min\_count=2)

w2v\_model.wv.index\_to\_key

# Find the most similar words to "cabin" based on word vectors from our trained model

w2v\_model.wv.most\_similar('proceed')

words = set(w2v\_model.wv.index\_to\_key )

X\_train\_vect = np.array([np.array([w2v\_model.wv[i] for i in ls if i in words])

                         for ls in X\_train])

X\_test\_vect = np.array([np.array([w2v\_model.wv[i] for i in ls if i in words])

                         for ls in X\_test])

# Compute sentence vectors by averaging the word vectors for the words contained in the sentence

X\_train\_vect\_avg = []

for v in X\_train\_vect:

    if v.size:

        X\_train\_vect\_avg.append(v.mean(axis=0))

    else:

        X\_train\_vect\_avg.append(np.zeros(100, dtype=float))

X\_test\_vect\_avg = []

for v in X\_test\_vect:

    if v.size:

        X\_test\_vect\_avg.append(v.mean(axis=0))

    else:

        X\_test\_vect\_avg.append(np.zeros(100, dtype=float))

# Instantiate and fit a basic Random Forest model on top of the vectors

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier()

rf\_model = rf.fit(X\_train\_vect\_avg, y\_train.values.ravel())

# Use the trained model to make predictions on the test data

y\_pred = rf\_model.predict(X\_test\_vect\_avg)

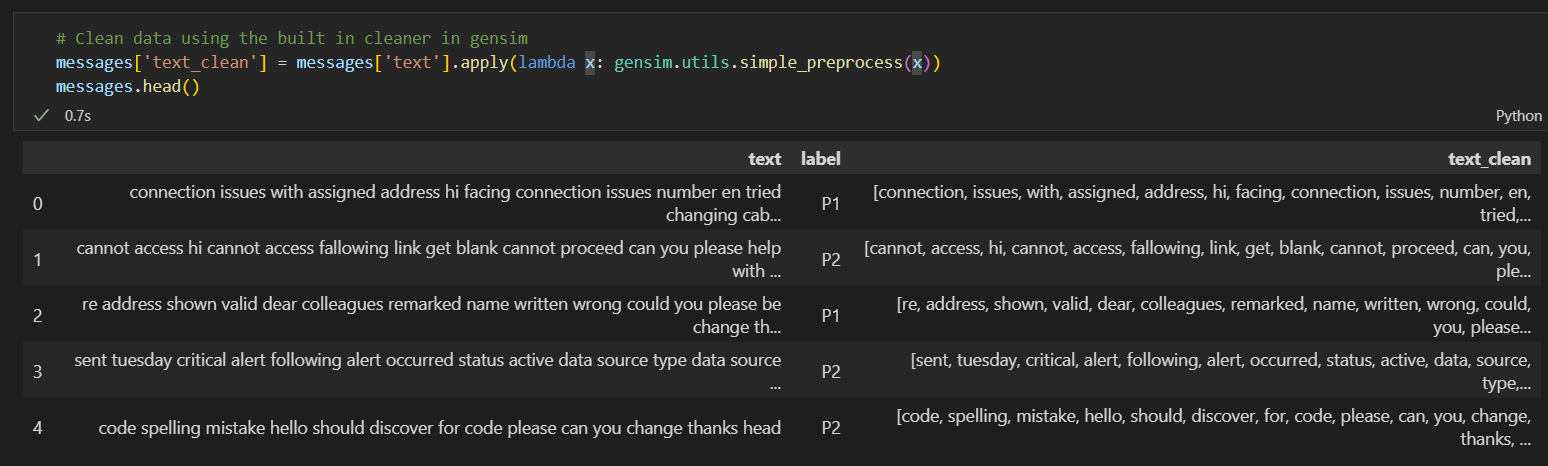
from sklearn.metrics import precision\_score, recall\_score

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

print('Precision: {} / Recall: {} / Accuracy: {}'.format(round(precision, 3), round(recall, 3), round((y\_pred==y\_test).sum()/len(y\_pred), 3)))

Output:



Text

Description automatically generated

Text

Description automatically generated

Practical-2

OBJECTIVE:

Implement Neural Network Bigram Model

Tools Used:

* Python
* Vs code editor
* Tensorflow
* Jupyter Notebook

Description:

Statistical language models, in its essence, are the type of models that assign probabilities to the sequences of words. In this article, we’ll understand the simplest model that assigns probabilities to sentences and sequences of words, the n-gram

You can think of an N-gram as the sequence of N words, by that notion, a 2-gram (or bigram) is a two-word sequence of words like “please turn”, “turn your”, or ”your homework”, and a 3-gram (or trigram) is a three-word sequence of words like “please turn your”, or “turn your homework”

Intuitive Formulation

Let’s start with equation P(w|h), the probability of word w, given some history, h. For example,



Here, w = The, h = its water is so transparent that



**The Bigram Model**

As the name suggests, the bigram model approximates the probability of a word given all the previous words by using only the conditional probability of one preceding word. In other words, you approximate it with the probability: P(the | that)

**For example -**

In the sentence "DEV is awesome and user friendly" the bigrams are :

"DEV is", "is awesome", "awesome and", "and user", "user friendly"

In this code the readData() function is taking four sentences which form the corpus. The sentences are

This is a dog

This is a cat

I love my cat

This is my name

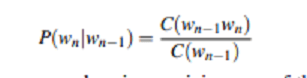
and these sentences are split to find the atomic words which form the vocabulary.

Then there is a function createBigram() which finds all the possible Bigrams the Dictionary of Bigrams and Unigrams along with their frequency i.e. how many times they occur in the corpus.

Then the function calcBigramProb() is used to calculate the probability of each bigram. The formula for which is

Alt Text

It is in terms of probability we then use count to find the probability. Which is basically



Then we use these probabilities to find the probability of next word by using the chain rule or we find the probability of the sentence like we have used in this program. We find the probability of the sentence "This is my cat" in the program given below.

Code:

def readData():

    data = ['This is a  dog','This is a cat','I love my cat','This is my name ']

    dat=[]

    for i in range(len(data)):

        for word in data[i].split():

            dat.append(word)

    print(dat)

    return dat

def createBigram(data):

   listOfBigrams = []

   bigramCounts = {}

   unigramCounts = {}

   for i in range(len(data)-1):

      if i < len(data) - 1 and data[i+1].islower():

         listOfBigrams.append((data[i], data[i + 1]))

         if (data[i], data[i+1]) in bigramCounts:

            bigramCounts[(data[i], data[i + 1])] += 1

         else:

            bigramCounts[(data[i], data[i + 1])] = 1

      if data[i] in unigramCounts:

         unigramCounts[data[i]] += 1

      else:

         unigramCounts[data[i]] = 1

   return listOfBigrams, unigramCounts, bigramCounts

def calcBigramProb(listOfBigrams, unigramCounts, bigramCounts):

    listOfProb = {}

    for bigram in listOfBigrams:

        word1 = bigram[0]

        word2 = bigram[1]

        listOfProb[bigram] = (bigramCounts.get(bigram))/(unigramCounts.get(word1))

    return listOfProb

if \_\_name\_\_ == '\_\_main\_\_':

    data = readData()

    listOfBigrams, unigramCounts, bigramCounts = createBigram(data)

    print("\n All the possible Bigrams are ")

    print(listOfBigrams)

    print("\n Bigrams along with their frequency ")

    print(bigramCounts)

    print("\n Unigrams along with their frequency ")

    print(unigramCounts)

    bigramProb = calcBigramProb(listOfBigrams, unigramCounts, bigramCounts)

    print("\n Bigrams along with their probability ")

    print(bigramProb)

    inputList="This is my cat"

    splt=inputList.split()

    outputProb1 = 1

    bilist=[]

    bigrm=[]

    for i in range(len(splt) - 1):

        if i < len(splt) - 1:

            bilist.append((splt[i], splt[i + 1]))

    print("\n The bigrams in given sentence are ")

    print(bilist)

    for i in range(len(bilist)):

        if bilist[i] in bigramProb:

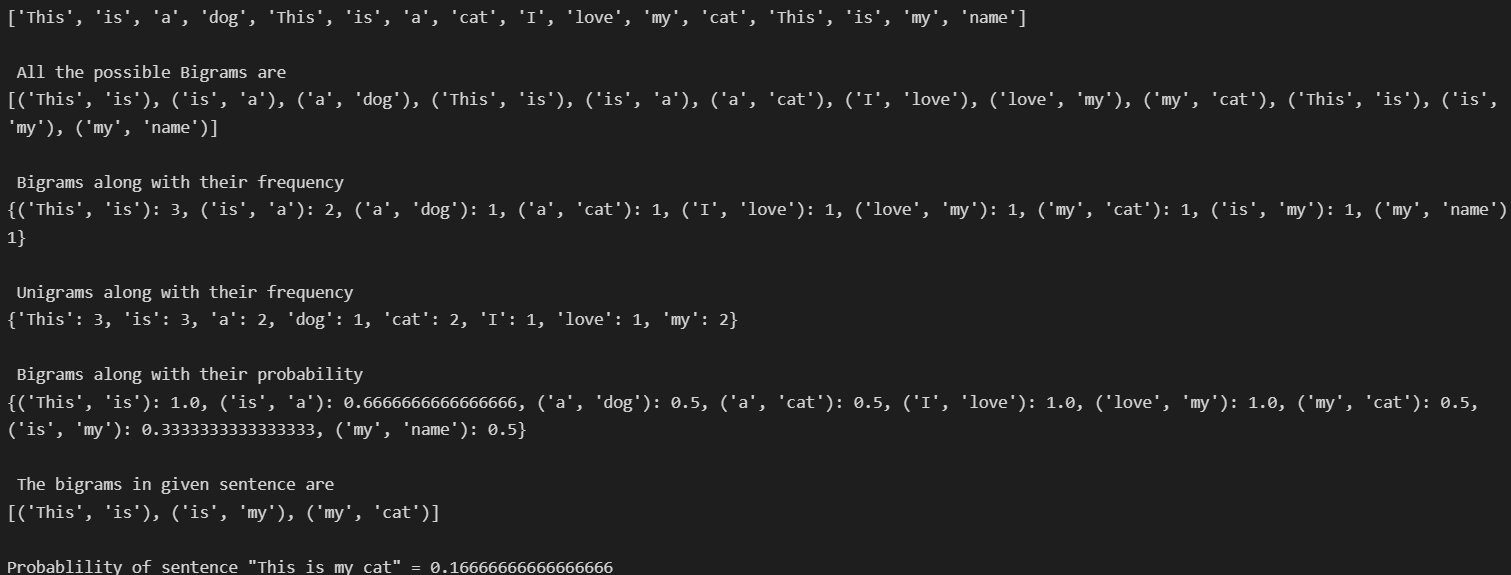
            outputProb1 \*= bigramProb[bilist[i]]

        else:

            outputProb1 \*= 0

    print('\n' + 'Probablility of sentence \"This is my cat\" = ' + str(outputProb1))

Output:



Practical-3

OBJECTIVE:

Implementation of word2vec using tensorflow.

Tools Used:

* Python
* Vs code editor
* Tensorflow
* Jupyter Notebook

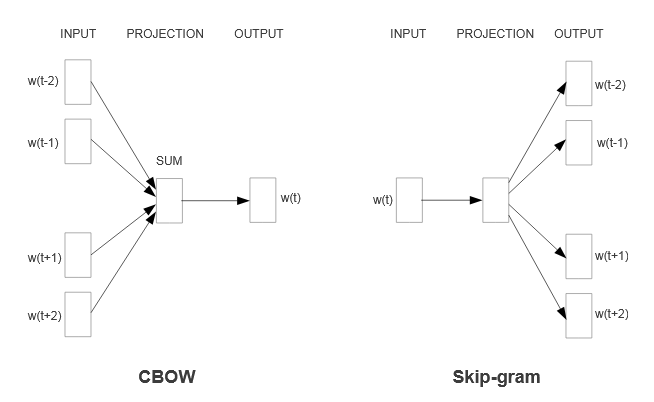
Description:

Word2Vec is a classical method that creates word embeddings in the field of Natural Language Processing (NLP). Word2vec takes in words from a large corpus of texts as input and learns to give out their vector representation. In the same way CNNs extract features from images, the word2vec algorithm extracts features from the text for particular words. Using those features, word2vec creates vectors that represent a word in the vector space. These vectors are chosen using the cosine similarity function, which indicates the semantic similarity between words.

Cosine similarity of 1 would mean that the angle between two words is 0; and would denote that the words are similar. Two similar words will occupy locations close to each other in that vector space, whereas words that are very different will occupy far away spaces. In that way, using its ability with linear algebra, the algorithm can recognize context and words that have similar meanings.

For example, the words “intelligent” and “smart” would appear closer together in this vector space, whereas the words “engine” and “car” will be far from “intelligent” and “smart”. This is because these words have that contextual understanding within a vector space.

#### **Model Architecture**



The algorithm uses a neural network architecture that consists of two learning models:

1. Continuous Bag-of-Words model (CBOW)

In this approach, the model uses context words to predict the target words. The input may be a group of words or a single word. It predicts a missing word given a window of context words or word sequence.

Suppose we have the sentence: An apple is green in color. If we remove the word “green” from the sentence and leave it blank, the model should predict the missing word.

It is referred to as the Bag of Words (BOW) model as the word order in history doesn’t influence the outcome. Further denoting the BOW model as continuous, i.e., Continous Bag Of Words (CBOW), means that the model uses distributed representation of the context continuously.

2. Continuous skip-gram model

The continuous skip-gram model works the other way around. It uses the target words to predict the context words. It involves training a neural network to learn the weights of the hidden layer. These learned weights correspond to the word vectors that we are trying to learn.

Code:

import numpy as np

import tensorflow as tf

raw\_text = 'He is the king . The king is royal . She is the royal  queen '

raw\_text = raw\_text.lower()

My\_words = []

for word in raw\_text.split():

    if word != '.':

        My\_words.append(word)

My\_words = set(My\_words)

My\_word2int = {}

My\_int2word = {}

vocabulary\_size = len(My\_words)

for i,word in enumerate(My\_words):

    My\_word2int[word] = i

raw\_text\_2 = raw\_text.split('.')

sentences = []

for sentence in raw\_text\_2:

    sentences.append(sentence.split())

print("The list of sentenses is:",sentences)

king\_data = []

Set\_window\_size = 2

for sentence in sentences:

    for word\_index, word in enumerate(sentence):

        for nb\_word in sentence[max(word\_index - Set\_window\_size, 0) : min(word\_index + Set\_window\_size, len(sentence)) + 1] :

            if nb\_word != word:

                king\_data.append([word, nb\_word])

print("The list of word pairs is:",king\_data)

def one\_hot(index\_data\_point, vocabulary\_size):

    temp = np.zeros(vocabulary\_size)

    temp[index\_data\_point] = 1

    return temp

x\_train\_data = [] # this is for input word

y\_train\_data = [] # this is for output word

for word\_data in king\_data:

    x\_train\_data.append(one\_hot(My\_word2int[word\_data[0]], vocabulary\_size))

    y\_train\_data.append(one\_hot(My\_word2int[word\_data[1]], vocabulary\_size))

print("This is the shape for x\_train and y\_train:",len(x\_train\_data), len(y\_train\_data))

tf.compat.v1.disable\_eager\_execution()

x\_new = tf.compat.v1.placeholder(tf.float32, shape=(None, vocabulary\_size))

label\_y = tf.compat.v1.placeholder(tf.float32, shape=(None, vocabulary\_size))

dim\_embbed = 5

W1\_data = tf.Variable(tf.compat.v1.random\_normal([vocabulary\_size, dim\_embbed]))

b1\_data = tf.Variable(tf.compat.v1.random\_normal([dim\_embbed])) #bias

representation\_hidden = tf.add(tf.matmul(x\_new,W1\_data), b1\_data)

W2\_data = tf.Variable(tf.compat.v1.random\_normal([dim\_embbed, vocabulary\_size]))

b2\_data = tf.Variable(tf.compat.v1.random\_normal([vocabulary\_size]))

Make\_prediction = tf.nn.softmax(tf.add( tf.matmul(representation\_hidden, W2\_data), b2\_data))

sess = tf.compat.v1.Session()

initialize\_var = tf.compat.v1.global\_variables\_initializer()

sess.run(initialize\_var)

# define the loss function:

cross\_entropy\_loss = tf.reduce\_mean(-tf.reduce\_sum(label\_y \* tf.compat.v1.log(Make\_prediction), axis=[1]))

# define the training step:

Step\_train = tf.compat.v1.train.GradientDescentOptimizer(0.1).minimize(cross\_entropy\_loss)

iters = 10000

# train for defined iterations

for ele in range(iters):

    sess.run(Step\_train, feed\_dict={x\_new: x\_train\_data, label\_y: y\_train\_data})

    print('The loss is : ', sess.run(cross\_entropy\_loss, feed\_dict={x\_new: x\_train\_data, label\_y: y\_train\_data}))

Output:

