**Practical File**

Course Title: NLP

Course Code: CSE468

Credit Units: 03



Department of Computer Science and Engineering

Amity Institute of Engineering and Technology

Uttar Pradesh, Noida

Name: Amarjeet Singh Chauhan

Class: 7CSE13X

Enrollment No: A023119819028

Index

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S.**  **No.** | **Category of Assignment** | **Code** | **Program**  **No.** | **Name of Program** | **Date of Allotment** | **Max. Marks** | **Marks Obtained** | **Sig. of Faculty** |
| 1 | Mandatory Experiment | LR  (10) | 1 | Word2Vec Implementation using Numpy | 19/07/22 | 1 |  |  |
| 2 | 2 | Word2Vec implementation using TensorFlow | 26/07/22 | 1 |  |  |
| 3 | 3 | Visualizing data with analogies with t-SNE | 2/08/22 | 1 |  |  |
| 4 | 4 | Visualizing data with analogies with PCA | 09/08/2022 | 1 |  |  |
| 5 | 5 | Neural Network Bigram Model | 16/08/2022 | 1 |  |  |
| 6 | 6 | GLOVE using numpy gradient descent | 23/08/2022 | 1 |  |  |
| 7 | 7 | GLOVE using Alternative Least Squares | 30/08/2022 | 1 |  |  |
| 8 | 8 | Embedding projectors | 06/09/2022 | 1 |  |  |
| 9 | 9 | Point wise Mutual Information. | 13/09/2022 | 1 |  |  |
| 10 | 10 | Classification with word vectors | 20/09/2022 | 1 |  |  |
| 11 | Performance | PR (10) |  |  |  | 10 |  |  |

**Experiment-1**

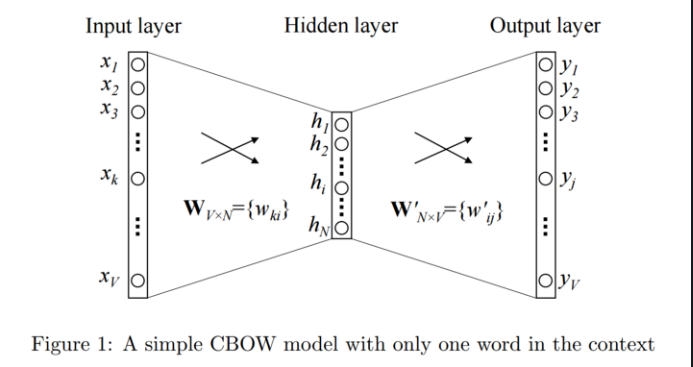
**Aim-**Implementation of word2vec using NumPy.

**Theory-** NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Word2Vec is a method to construct such an embedding. It can be obtained using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW). word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embedding’s from large datasets. Embedding’s learned through word2vec have proven to be successful on a variety of downstream natural language processing tasks.

Continuous bag-of-words model: predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

Continuous skip-gram model: predicts words within a certain range before and after the current word in the same sentence.



Code-

import numpy as np

import pandas as pd

import re

from collections import defaultdict

class word2vec():

def \_init\_ (self):

self.n = settings['n']

self.eta = settings['learning\_rate']

self.epochs = settings['epochs']

self.window = settings['window\_size']

pass

# GENERATE TRAINING DATA

def generate\_training\_data(self, settings, corpus):

# GENERATE WORD COUNTS

word\_counts = defaultdict(int)

for row in corpus:

for word in row:

word\_counts[word] += 1

self.v\_count = len(word\_counts.keys())

# GENERATE LOOKUP DICTIONARIES

self.words\_list = sorted(list(word\_counts.keys()),reverse=False)

self.word\_index = dict((word, i) for i, word in enumerate(self.words\_list))

self.index\_word = dict((i, word) for i, word in enumerate(self.words\_list))

training\_data = []

# CYCLE THROUGH EACH SENTENCE IN CORPUS

for sentence in corpus:

sent\_len = len(sentence)

# CYCLE THROUGH EACH WORD IN SENTENCE

for i, word in enumerate(sentence):

#w\_target = sentence[i]

w\_target = self.word2onehot(sentence[i])

# CYCLE THROUGH CONTEXT WINDOW

w\_context = []

for j in range(i-self.window, i+self.window+1):

if j!=i and j<=sent\_len-1 and j>=0:

w\_context.append(self.word2onehot(sentence[j]))

training\_data.append([w\_target, w\_context])

return np.array(training\_data)

# SOFTMAX ACTIVATION FUNCTION

def softmax(self, x):

e\_x = np.exp(x - np.max(x))

return e\_x / e\_x.sum(axis=0)

# CONVERT WORD TO ONE HOT ENCODING

def word2onehot(self, word):

word\_vec = [0 for i in range(0, self.v\_count)]

word\_index = self.word\_index[word]

word\_vec[word\_index] = 1

return word\_vec

# FORWARD PASS

def forward\_pass(self, x):

h = np.dot(self.w1.T, x)

u = np.dot(self.w2.T, h)

y\_c = self.softmax(u)

return y\_c, h, u

# BACKPROPAGATION

def backprop(self, e, h, x):

dl\_dw2 = np.outer(h, e)

dl\_dw1 = np.outer(x, np.dot(self.w2, e.T))

# UPDATE WEIGHTS

self.w1 = self.w1 - (self.eta \* dl\_dw1)

self.w2 = self.w2 - (self.eta \* dl\_dw2)

pass

# TRAIN W2V model

def train(self, training\_data):

# INITIALIZE WEIGHT MATRICES

self.w1 = np.random.uniform(-0.8, 0.8, (self.v\_count, self.n)) # embedding matrix

self.w2 = np.random.uniform(-0.8, 0.8, (self.n, self.v\_count)) # context matrix

# CYCLE THROUGH EACH EPOCH

for i in range(0, self.epochs):

self.loss = 0

# CYCLE THROUGH EACH TRAINING SAMPLE

for w\_t, w\_c in training\_data:

# FORWARD PASS

y\_pred, h, u = self.forward\_pass(w\_t)

# CALCULATE ERROR

EI = np.sum([np.subtract(y\_pred, word) for word in w\_c], axis=0)

# BACKPROPAGATION

self.backprop(EI, h, w\_t)

# CALCULATE LOSS

self.loss += -np.sum([u[word.index(1)] for word in w\_c]) + len(w\_c) \* np.log(np.sum(np.exp(u)))

#self.loss += -2\*np.log(len(w\_c)) -np.sum([u[word.index(1)] for word in w\_c]) + (len(w\_c) \* np.log(np.sum(np.exp(u))))

print('EPOCH:',i, 'LOSS:', self.loss)

# input a word, returns a vector (if available)

def word\_vec(self, word):

w\_index = self.word\_index[word]

v\_w = self.w1[w\_index]

return v\_w

# input a vector, returns nearest word(s)

def vec\_sim(self, vec, top\_n):

# CYCLE THROUGH VOCAB

word\_sim = {}

for i in range(self.v\_count):

v\_w2 = self.w1[i]

theta\_num = np.dot(vec, v\_w2)

theta\_den = np.linalg.norm(vec) \* np.linalg.norm(v\_w2)

theta = theta\_num / theta\_den

word = self.index\_word[i]

word\_sim[word] = theta

words\_sorted = sorted(word\_sim.items(), key=lambda x: x[1], reverse=True)

for word, sim in words\_sorted[:top\_n]:

print(word, sim)

# input word, returns top [n] most similar words

def word\_sim(self, word, top\_n):

w1\_index = self.word\_index[word]

v\_w1 = self.w1[w1\_index]

# CYCLE THROUGH VOCAB

word\_sim = {}

for i in range(self.v\_count):

v\_w2 = self.w1[i]

theta\_num = np.dot(v\_w1, v\_w2)

theta\_den = np.linalg.norm(v\_w1) \* np.linalg.norm(v\_w2)

theta = theta\_num / theta\_den

word = self.index\_word[i]

word\_sim[word] = theta

words\_sorted = sorted(word\_sim.items(), key=lambda x: x[1], reverse=True)

for word, sim in words\_sorted[:top\_n]:

print(word, sim)

#--- EXAMPLE RUN --------------------------------------------------------------+

settings = {}

settings['n'] = 5 # dimension of word embeddings

settings['window\_size'] = 2 # context window +/- center word

settings['min\_count'] = 0 # minimum word count

settings['epochs'] = 600 # number of training epochs

settings['neg\_samp'] = 10 # number of negative words to use during training

settings['learning\_rate'] = 0.01 # learning rate

np.random.seed(0) # set the seed for reproducibility

corpus = [['this','mobile','is','good','not','affordable']]

# INITIALIZE W2V MODEL

w2v = word2vec()

# generate training data

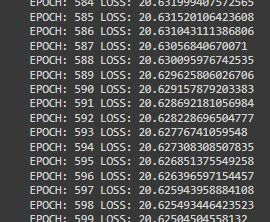
training\_data = w2v.generate\_training\_data(settings, corpus)

# train word2vec model

w2v.train(training\_data)

**Output-**





**Results-** Successfully implemented word2vec.

**Experiment-2**

**Aim-** Implementation of word2vec using tensorflow.

**Theory-** Word2Vec is a method to construct such an embedding. It can be obtained using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW). word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embedding’s from large datasets. Embedding’s learned through word2vec have proven to be successful on a variety of downstream natural language processing tasks.

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

**Code-**

from \_future\_ import absolute\_import, division, print\_function, unicode\_literals

import tensorflow as tf

import numpy as np

class Word2Vec:

def \_init\_(self, vocab\_size=0, embedding\_dim=16, optimizer='sgd', epochs=10000):

self.vocab\_size=vocab\_size

self.embedding\_dim=5

self.epochs=epochs

if optimizer=='adam':

self.optimizer = tf.optimizers.Adam()

else:

self.optimizer = tf.optimizers.SGD(learning\_rate=0.1)

def train(self, x\_train=None, y\_train=None):

self.W1 = tf.Variable(tf.random.normal([self.vocab\_size, self.embedding\_dim]))

self.b1 = tf.Variable(tf.random.normal([self.embedding\_dim])) #bias

self.W2 = tf.Variable(tf.random.normal([self.embedding\_dim, self.vocab\_size]))

self.b2 = tf.Variable(tf.random.normal([self.vocab\_size]))

for \_ in range(self.epochs):

with tf.GradientTape() as t:

hidden\_layer = tf.add(tf.matmul(x\_train,self.W1),self.b1)

output\_layer = tf.nn.softmax(tf.add( tf.matmul(hidden\_layer, self.W2), self.b2))

cross\_entropy\_loss = tf.reduce\_mean(-tf.math.reduce\_sum(y\_train \* tf.math.log(output\_layer), axis=[1]))

grads = t.gradient(cross\_entropy\_loss, [self.W1, self.b1, self.W2, self.b2])

self.optimizer.apply\_gradients(zip(grads,[self.W1, self.b1, self.W2, self.b2]))

if(\_ % 1000 == 0):

print(cross\_entropy\_loss)

def vectorized(self, word\_idx):

return (self.W1+self.b1)[word\_idx]

corpus\_raw = 'this mobile is good this mobile is not good this mobile is affordable'

# convert to lower case

corpus\_raw = corpus\_raw.lower()

# raw sentences is a list of sentences.

raw\_sentences = corpus\_raw.split('.')

sentences = []

for sentence in raw\_sentences:

sentences.append(sentence.split())

data = []

WINDOW\_SIZE = 2

for sentence in sentences:

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

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

if nb\_word != word:

data.append([word, nb\_word])

words = []

for word in corpus\_raw.split():

if word != '.': # because we don't want to treat . as a word

words.append(word)

words = set(words) # so that all duplicate words are removed

word2int = {}

int2word = {}

vocab\_size = len(words) # gives the total number of unique words

for i,word in enumerate(words):

word2int[word] = i

int2word[i] = word

# function to convert numbers to one hot vectors

def to\_one\_hot(data\_point\_index, vocab\_size):

temp = np.zeros(vocab\_size)

temp[data\_point\_index] = 1

return temp

x\_train = [] # input word

y\_train = [] # output word

for data\_word in data:

x\_train.append(to\_one\_hot(word2int[ data\_word[0] ], vocab\_size))

y\_train.append(to\_one\_hot(word2int[ data\_word[1] ], vocab\_size))

# convert them to numpy arrays

x\_train = np.asarray(x\_train, dtype='float32')

y\_train = np.asarray(y\_train, dtype='float32')

w2v = Word2Vec(vocab\_size=vocab\_size, optimizer='adam', epochs=10000)

w2v.train(x\_train, y\_train)

w2v.vectorized(word2int['mobile'])

sentences = sentences[0]

vectors = []

for i in sentences:

vectors.append(w2v.vectorized(word2int[i]))

from sklearn.manifold import TSNE

from sklearn import preprocessing

model = TSNE(n\_components=2, random\_state=0)

np.set\_printoptions(suppress=True)

vectors = model.fit\_transform(vectors)

normalizer = preprocessing.Normalizer()

vectors = normalizer.fit\_transform(vectors, 'l2')

import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set\_xlim(left=-1, right=1)

ax.set\_ylim(bottom=-1, top=1)

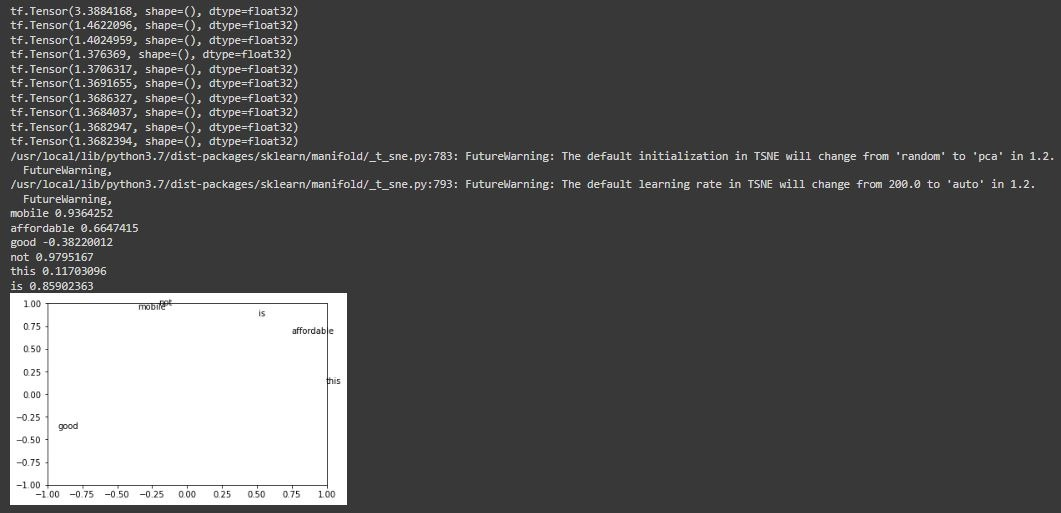
for word in words:

print(word, vectors[word2int[word]][1])

ax.annotate(word, (vectors[word2int[word]][0],vectors[word2int[word]][1] ))

plt.show()

**Output-**



Results- Successfully implemented word2vec using tensorflow.

**Experiment-3**

**Aim-** Visualizing data with analogies with t-SNE.

**Theory-** t-Distributed Stochastic Neighbor Embedding (t-SNE) is an unsupervised, non-linear technique primarily used for data exploration and visualizing high-dimensional data. In simpler terms, t-SNE gives you a feel or intuition of how the data is arranged in a high-dimensional space.

t-SNE is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high. This will suppress some noise and speed up the computation of pairwise distances between samples.

**Code-**

from sklearn.manifold import TSNE

from keras.datasets import mnist

from sklearn.datasets import load\_iris

from numpy import reshape

import seaborn as sns

import pandas as pd

iris = load\_iris()

x = iris.data

y = iris.target

tsne = TSNE(n\_components=2, verbose=1, random\_state=123)

z = tsne.fit\_transform(x)

df = pd.DataFrame()

df["y"] = y

df["comp-1"] = z[:,0]

df["comp-2"] = z[:,1]

sns.scatterplot(x="comp-1", y="comp-2", hue=df.y.tolist(),

palette=sns.color\_palette("hls", 3),

data=df).set(title="Iris data T-SNE projection")

(x\_train, y\_train), (\_ , \_) = mnist.load\_data()

x\_train = x\_train[:3000]

y\_train = y\_train[:3000]

print(x\_train.shape)

x\_mnist = reshape(x\_train, [x\_train.shape[0], x\_train.shape[1]\*x\_train.shape[2]])

print(x\_mnist.shape)

tsne = TSNE(n\_components=2, verbose=1, random\_state=123)

z = tsne.fit\_transform(x\_mnist)

df = pd.DataFrame()

df["y"] = y\_train

df["comp-1"] = z[:,0]

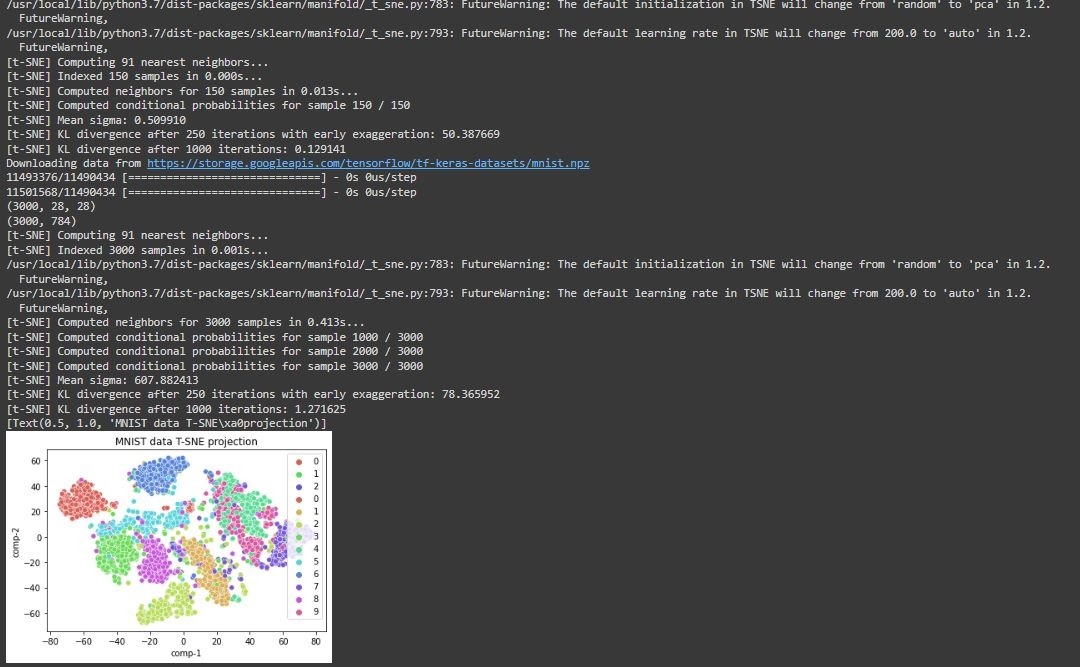
df["comp-2"] = z[:,1]

sns.scatterplot(x="comp-1", y="comp-2", hue=df.y.tolist(),

palette=sns.color\_palette("hls", 10),

data=df).set(title="MNIST data T-SNE projection")

**Output-**



**Results-** Successfully visualized data using t-SNE.

Experiment-4

**Aim-** Visualizing data with analogies with PCA

**Theory-** Principal Component analysis is a technique for feature extraction — so it combines our input variables in a specific way, then we can drop the “least important” variables while still retaining the most valuable parts of all of the variables! As an added benefit, each of the “new” variables after PCA are all independent of one another. This is a benefit because the assumptions of a linear model require our independent variables to be independent of one another. If we decide to fit a linear regression model with these “new” variables, this assumption will necessarily be satisfied.

**Code-**

import pandas as pd

import numpy as np

import seaborn as sb

from sklearn import preprocessing

import matplotlib.pyplot as plt

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import r2\_score

df = pd.read\_csv("/content/CarPrice\_1.csv", encoding = 'unicode\_escape')

print(df)

label\_encoder = preprocessing.LabelEncoder()

cols\_tbe = ['CarName', 'fueltype', 'aspiration', 'doornumber', 'carbody', 'drivewheel', 'enginelocation', 'fuelsystem', 'enginetype', 'cylindernumber']

for col\_tbe in cols\_tbe:

df[col\_tbe] = df[col\_tbe].astype('|S')

df[col\_tbe] = label\_encoder.fit\_transform(df[col\_tbe])

X = df.iloc[:,0:-1].values

y = df.iloc[:,-1].values

X\_std = preprocessing.StandardScaler().fit\_transform(X)

cov\_m = np.cov(X\_std, rowvar=False)

cov\_m = np.cov(X\_std.T)

eig\_vals, eig\_vecs = np.linalg.eig(cov\_m)

eig\_pairs = [(np.abs(eig\_vals[i]), eig\_vecs[:,i]) for i in range(len(eig\_vals))]

eig\_pairs.sort()

eig\_pairs.reverse()

tot = sum(eig\_vals)

var\_exp = [(i / tot)\*100 for i in sorted(eig\_vals, reverse=True)]

cum\_var\_exp = np.cumsum(var\_exp)

matrix\_w = np.hstack((eig\_pairs[0][1].reshape(25,1),

eig\_pairs[1][1].reshape(25,1),

eig\_pairs[2][1].reshape(25,1),

eig\_pairs[3][1].reshape(25,1),

eig\_pairs[4][1].reshape(25,1),

eig\_pairs[5][1].reshape(25,1),

eig\_pairs[6][1].reshape(25,1),

eig\_pairs[7][1].reshape(25,1),

eig\_pairs[8][1].reshape(25,1),

eig\_pairs[9][1].reshape(25,1),

eig\_pairs[10][1].reshape(25,1),

eig\_pairs[11][1].reshape(25,1),

eig\_pairs[12][1].reshape(25,1),

eig\_pairs[13][1].reshape(25,1),

eig\_pairs[14][1].reshape(25,1)))

Y = X\_std.dot(matrix\_w)

principalDf = pd.DataFrame(data = Y, columns = ['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10', 'PC11', 'PC12', 'PC13', 'PC14', 'PC15'])

final\_df = pd.concat([principalDf, pd.DataFrame(y, columns=['price'])], axis = 1)

print(final\_df)

data = final\_df.values

X,y = data[:,:-1], data[:,-1]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 0)

regr = LinearRegression()

regr.fit(X\_train, y\_train)

y\_pred = regr.predict(X\_test)

print('R2 Score:',r2\_score(y\_test, y\_pred))

fig = plt.figure()

m, b = np.polyfit(y\_test, y\_pred, 1)

plt.plot(y\_test, m\*y\_test + b, color='red')

plt.scatter(y\_test, y\_pred, alpha=.5)

fig.suptitle('Actual vs Predicted Price', fontsize = 20)

plt.xlabel('Actual', fontsize = 18)

plt.ylabel('Predicted', fontsize = 18)

plt.show()

data = df.values

X,y = data[:,:-1], data[:,-1]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 0)

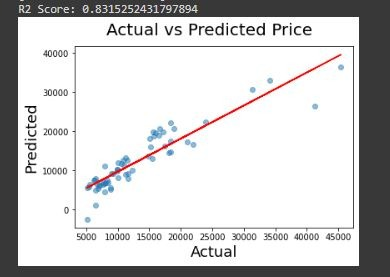
regr = LinearRegression()

regr.fit(X\_train, y\_train)

y\_pred = regr.predict(X\_test)

print('R2 Score (without using PCA):',r2\_score(y\_test, y\_pred))

**Output-**





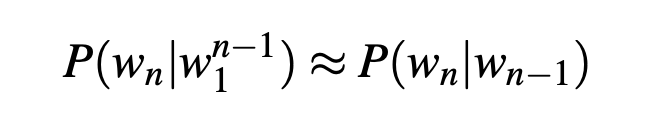
**Results-** Successfully visualized data using PCA.

**Experiment-5**

**Aim**- To implement Neural Network Bigram Model.

**Theory**- 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)

And so, when you use a bigram model to predict the conditional probability of the next word, you are thus making the following approximation:



**Code**-

# imports

import string

import random

import nltk

nltk.download('punkt')

nltk.download('stopwords')

nltk.download('reuters')

from nltk.corpus import reuters

from nltk import FreqDist

# input the reuters sentences

sents  =reuters.sents()

# write the removal characters such as : Stopwords and punctuation

stop\_words = set(stopwords.words('english'))

string.punctuation = string.punctuation +'"'+'"'+'-'+'''+'''+'—'

string.punctuation

removal\_list = list(stop\_words) + list(string.punctuation)+ ['lt','rt']

removal\_list

# generate unigrams bigrams trigrams

unigram=[]

bigram=[]

trigram=[]

tokenized\_text=[]

for sentence in sents:

  sentence = list(map(lambda x:x.lower(),sentence))

  for word in sentence:

        if word== '.':

            sentence.remove(word)

        else:

            unigram.append(word)

  tokenized\_text.append(sentence)

  bigram.extend(list(ngrams(sentence, 2,pad\_left=True, pad\_right=True)))

  trigram.extend(list(ngrams(sentence, 3, pad\_left=True, pad\_right=True)))

# remove the n-grams with removable words

def remove\_stopwords(x):

    y = []

    for pair in x:

        count = 0

        for word in pair:

            if word in removal\_list:

                count = count or 0

            else:

                count = count or 1

        if (count==1):

            y.append(pair)

    return (y)

unigram = remove\_stopwords(unigram)

bigram = remove\_stopwords(bigram)

trigram = remove\_stopwords(trigram)

# generate frequency of n-grams

freq\_bi = FreqDist(bigram)

freq\_tri = FreqDist(trigram)

d = defaultdict(Counter)

for a, b, c in freq\_tri:

    if(a != None and b!= None and c!= None):

      d[a, b] += freq\_tri[a, b, c]

# Next word prediction

s=''

def pick\_word(counter):

    "Chooses a random element."

    return random.choice(list(counter.elements()))

prefix = "he", "said"

print(" ".join(prefix))

s = " ".join(prefix)

for i in range(19):

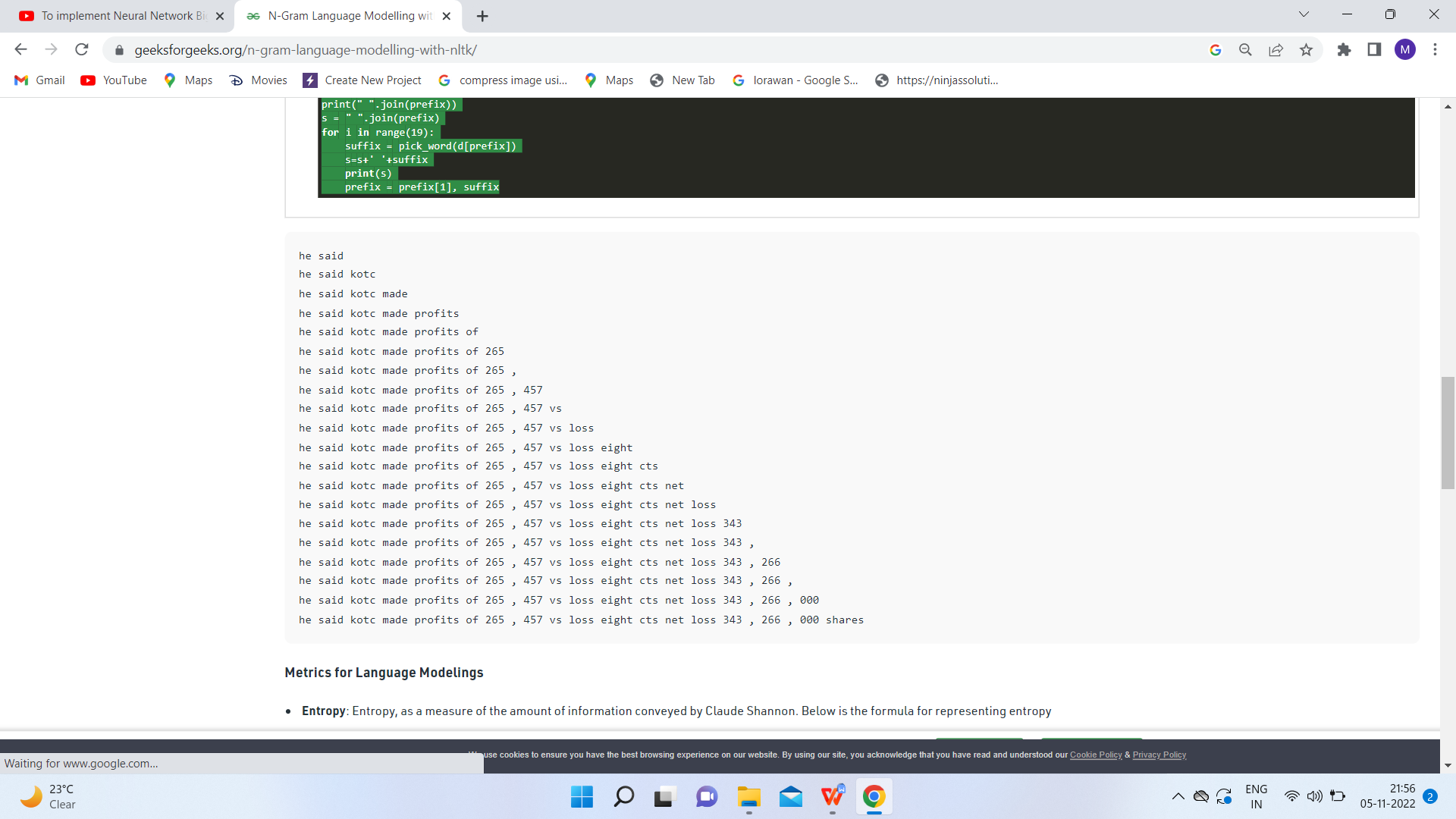
    suffix = pick\_word(d[prefix])

    s=s+' '+suffix

    print(s)

    prefix = prefix[1], suffix

**Output**-



**Result**- All the concepts related to the aim are fully understood and applied correctly.

**Experiment-6**

**Aim**- To implement GLOVE using numpy gradient descent.

**Theory**- GloVe is a word vector technique that rode the wave of word vectors after a brief silence. Just to refresh, word vectors put words to a nice vector space, where similar words cluster together and different words repel. The advantage of GloVe is that, unlike Word2vec, GloVe does not rely just on local statistics (local context information of words), but incorporates global statistics (word co-occurrence) to obtain word vectors.

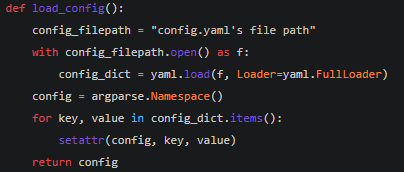
**Code**-

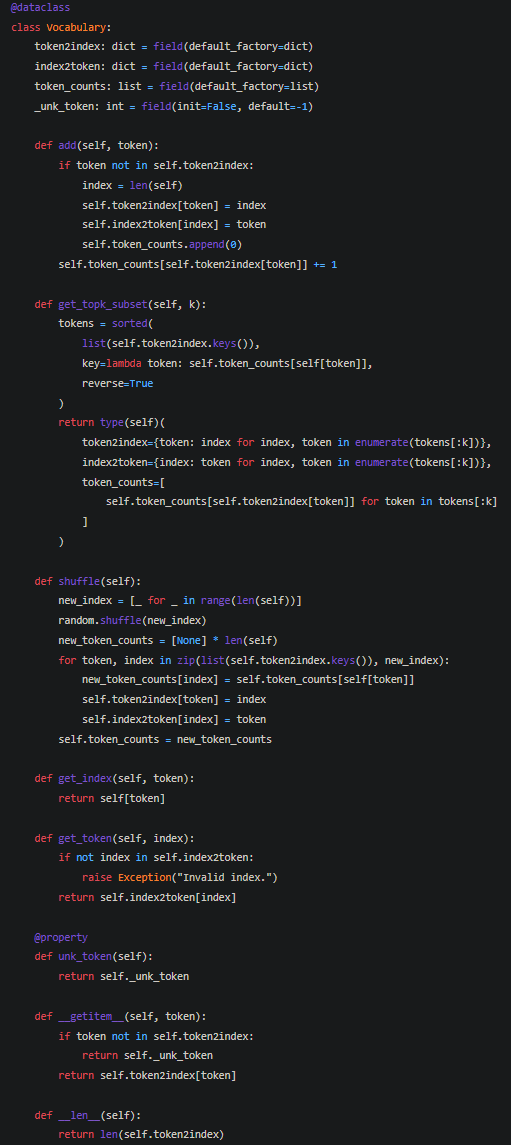
import gensim.downloader as api

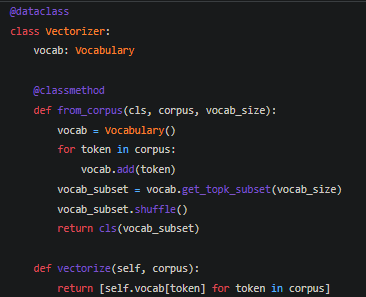
dataset = api.load("text8")

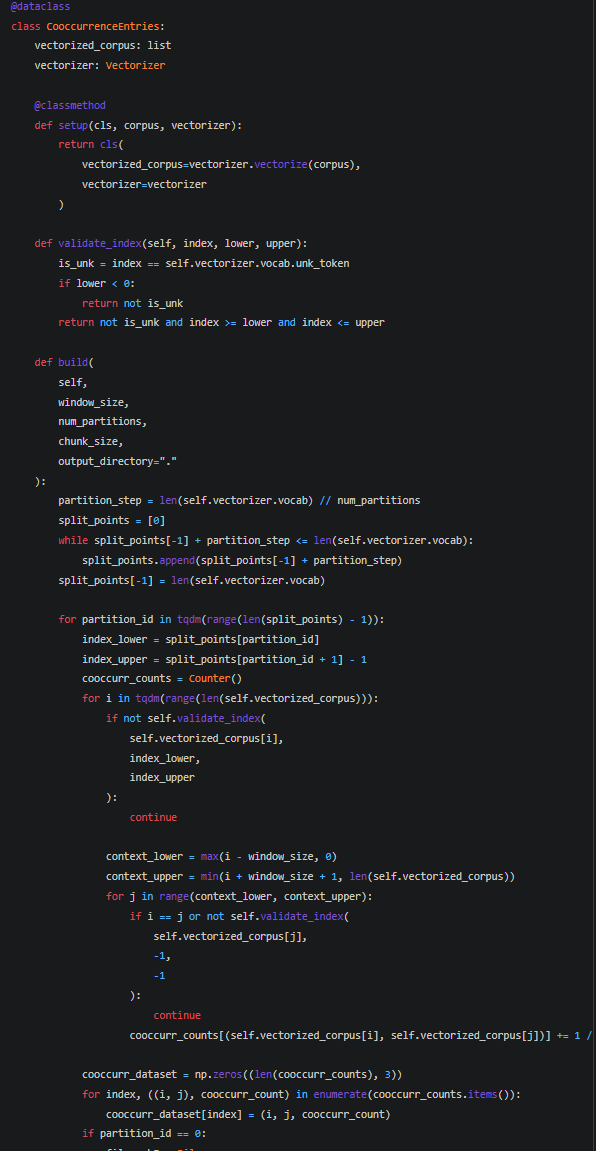
import itertools

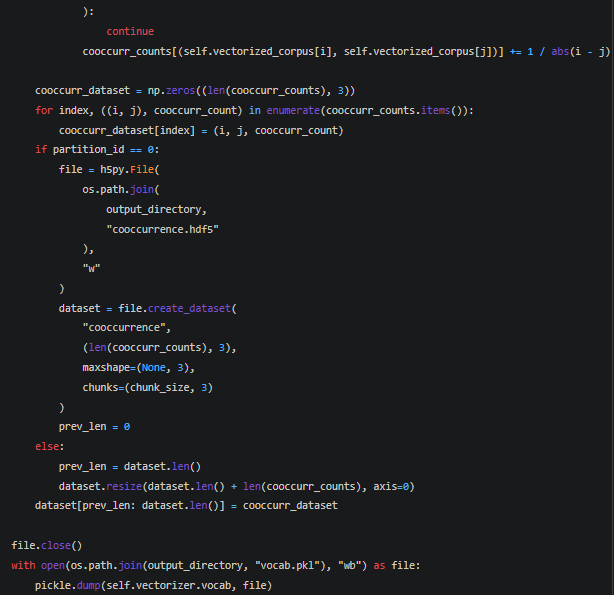
corpus = list(itertools.chain.from\_iterable(dataset))







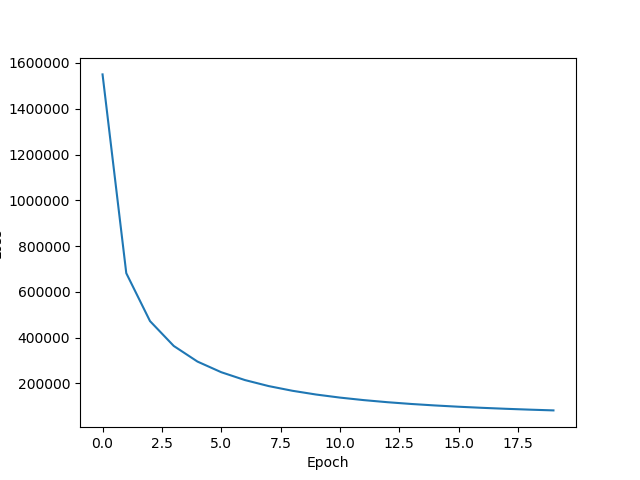


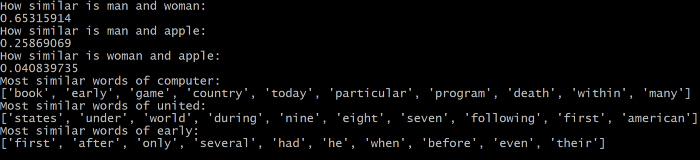






**Outpu**t-





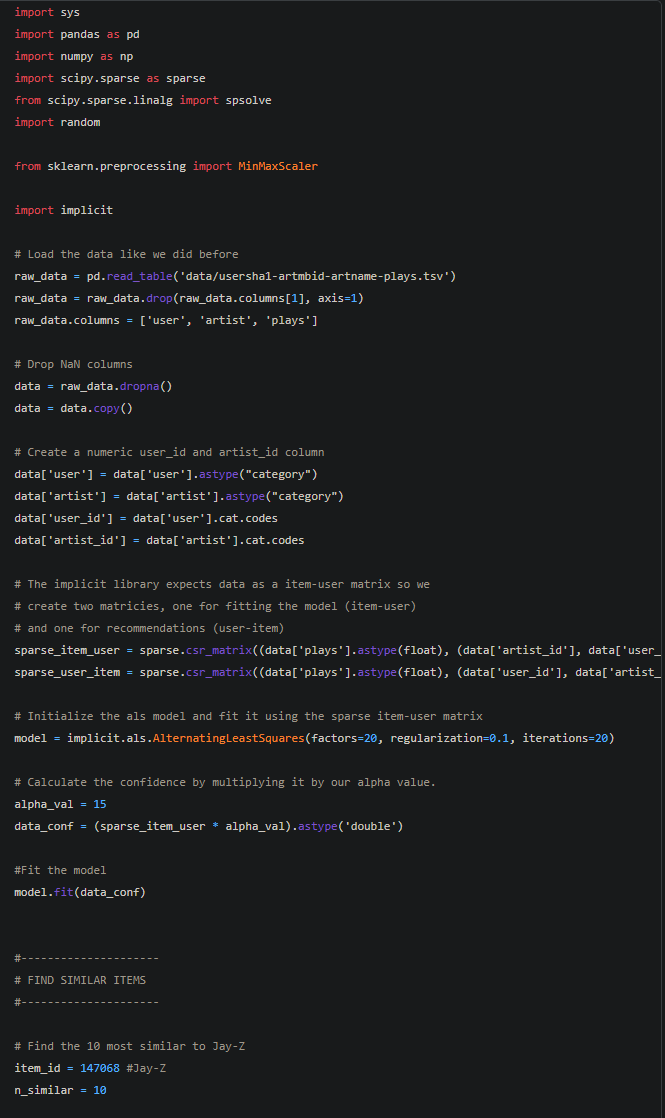
**Result**- All the concepts related to the aim are fully understood and applied correctly.

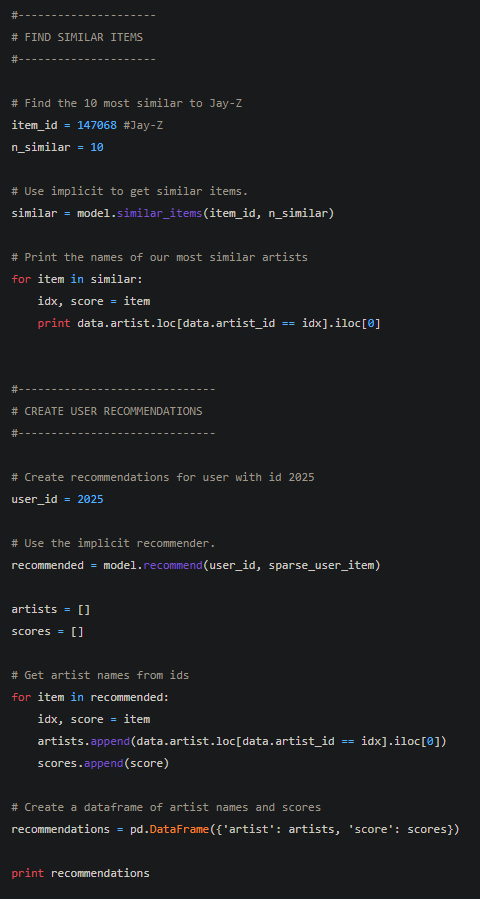
**Experiment-7**

**Aim**- To implement GLOVE using Alternative Least Squares.

**Theory**- Alternating Least Square (ALS) is also a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for a larges-scale collaborative filtering problems. ALS is doing a pretty good job at solving scalability and sparseness of the Ratings data, and it’s simple and scales well to very large datasets.

**Code**-





**Output**-



**Result**- All the concepts related to the aim are fully understood and applied correctly.

**Experiment-8**

**Aim**- To visualize the country analogies using embedding projectors.

**Theory**- The TensorBoard embedding projector is a very powerful tool in data analysis, specifically for interpreting and visualizing low-dimensional embedding’s. In order to do so, first, it applies a dimensionality reduction algorithm to the input embedding’s, between UMAP, T-SNE, PCA, or a custom one, to reduce their dimension to three and be able to render them in a three-dimensional space. Once the map is generated, this tool can be used, for example, to search for specific keywords associated with the embedding’s or highlight similar points in space. Ultimately, its goal is to provide a way to better interpret the embedding’s that our machine learning model is generating, to check if the similar ones according to our definition are plotted nearby in the 3D space.

**Code**-

try:  
  # %tensorflow\_version only exists in Colab.  
  %tensorflow\_version 2.x  
except Exception:  
  pass  
  
%load\_ext tensorboard

import os  
import tensorflow as tf  
import tensorflow\_datasets as tfds  
from tensorboard.plugins import projector

(train\_data, test\_data), info = tfds.load(  
    "imdb\_reviews/subwords8k",  
    split=(tfds.Split.TRAIN, tfds.Split.TEST),  
    with\_info=True,  
    as\_supervised=True,  
)  
encoder = info.features["text"].encoder  
  
# Shuffle and pad the data.  
train\_batches = train\_data.shuffle(1000).padded\_batch(  
    10, padded\_shapes=((None,), ())  
)  
test\_batches = test\_data.shuffle(1000).padded\_batch(  
    10, padded\_shapes=((None,), ())  
)  
train\_batch, train\_labels = next(iter(train\_batches))

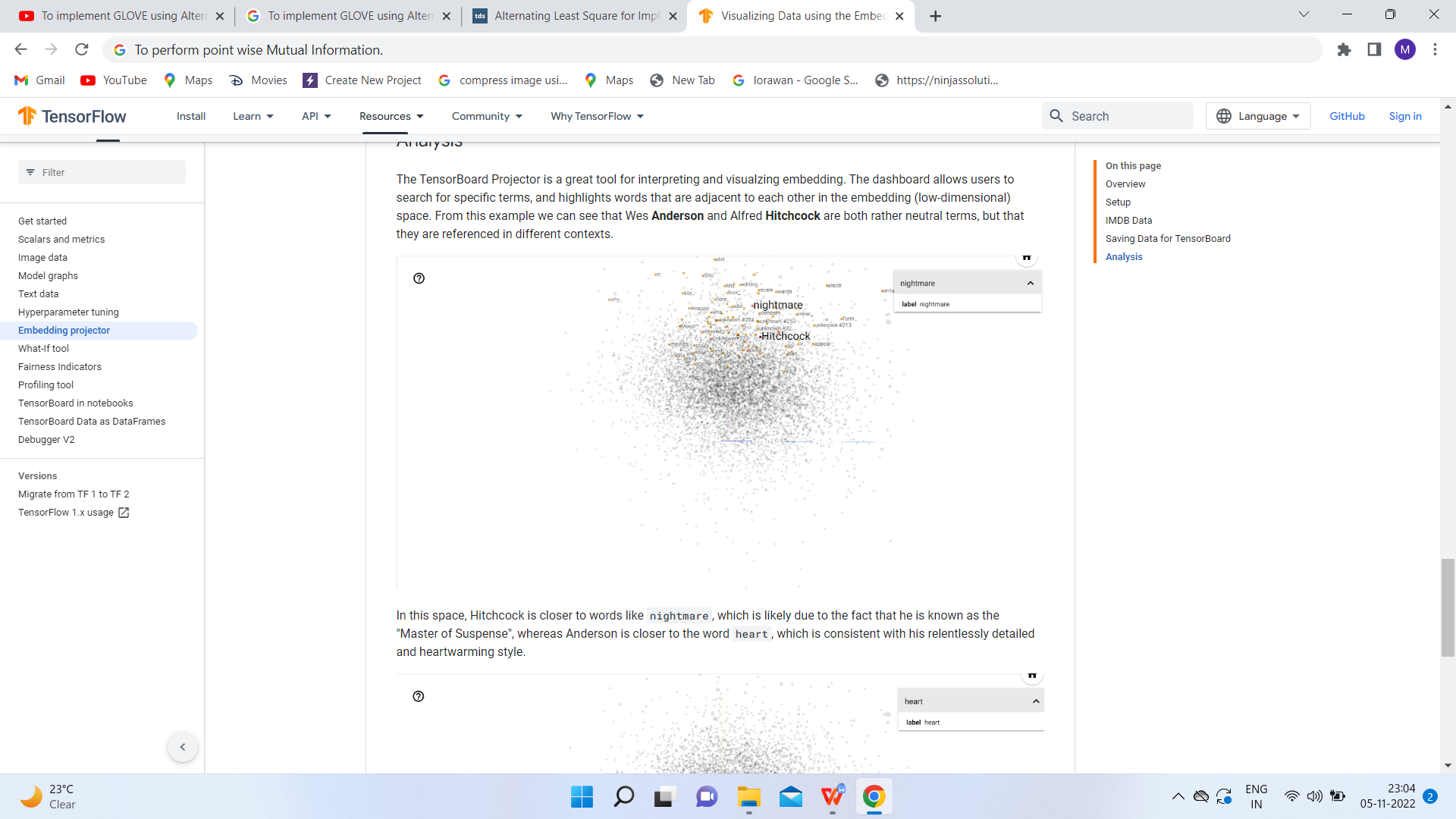
# Create an embedding layer.  
embedding\_dim = 16  
embedding = tf.keras.layers.Embedding(encoder.vocab\_size, embedding\_dim)  
# Configure the embedding layer as part of a keras model.  
model = tf.keras.Sequential(  
    [  
        embedding, # The embedding layer should be the first layer in a model.  
        tf.keras.layers.GlobalAveragePooling1D(),  
        tf.keras.layers.Dense(16, activation="relu"),  
        tf.keras.layers.Dense(1),  
    ]  
)  
  
# Compile model.  
model.compile(  
    optimizer="adam",  
    loss=tf.keras.losses.BinaryCrossentropy(from\_logits=True),  
    metrics=["accuracy"],  
)  
  
# Train model for one epoch.  
history = model.fit(  
    train\_batches, epochs=1, validation\_data=test\_batches, validation\_steps=20  
)

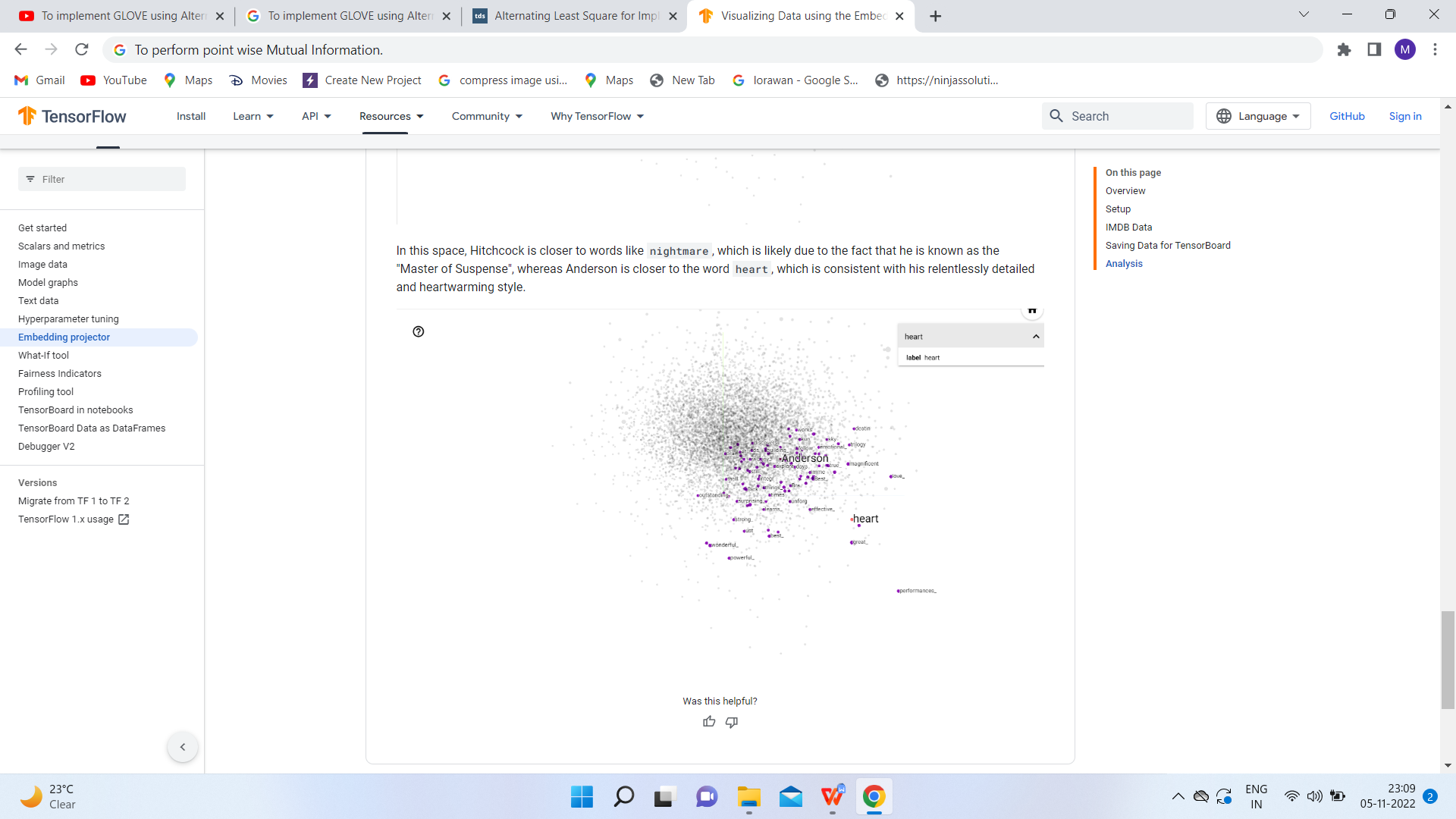
# Set up a logs directory, so Tensorboard knows where to look for files.  
log\_dir='/logs/imdb-example/'  
if not os.path.exists(log\_dir):  
    os.makedirs(log\_dir)  
  
# Save Labels separately on a line-by-line manner.  
with open(os.path.join(log\_dir, 'metadata.tsv'), "w") as f:  
  for subwords in encoder.subwords:  
    f.write("{}\n".format(subwords))  
  # Fill in the rest of the labels with "unknown".  
  for unknown in range(1, encoder.vocab\_size - len(encoder.subwords)):  
    f.write("unknown #{}\n".format(unknown))  
  
  
# Save the weights we want to analyze as a variable. Note that the first  
# value represents any unknown word, which is not in the metadata, here  
# we will remove this value.  
weights = tf.Variable(model.layers[0].get\_weights()[0][1:])  
# Create a checkpoint from embedding, the filename and key are the  
# name of the tensor.  
checkpoint = tf.train.Checkpoint(embedding=weights)  
checkpoint.save(os.path.join(log\_dir, "embedding.ckpt"))  
  
# Set up config.  
config = projector.ProjectorConfig()  
embedding = config.embeddings.add()  
# The name of the tensor will be suffixed by `/.ATTRIBUTES/VARIABLE\_VALUE`.  
embedding.tensor\_name = "embedding/.ATTRIBUTES/VARIABLE\_VALUE"  
embedding.metadata\_path = 'metadata.tsv'  
projector.visualize\_embeddings(log\_dir, config)

# Now run tensorboard against on log data we just saved.  
%tensorboard --logdir /logs/imdb-example/

**Output**-







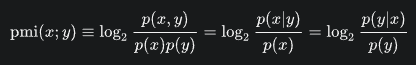
**Result**- All the concepts related to the aim are fully understood and applied correctly.

**Experiment-9**

**Aim**- To perform point wise Mutual Information.

**Theory**- In statistics, probability theory and information theory, pointwise mutual information (PMI), or point mutual information, is a measure of association. It compares the probability of two events occurring together to what this probability would be if the events were independent.

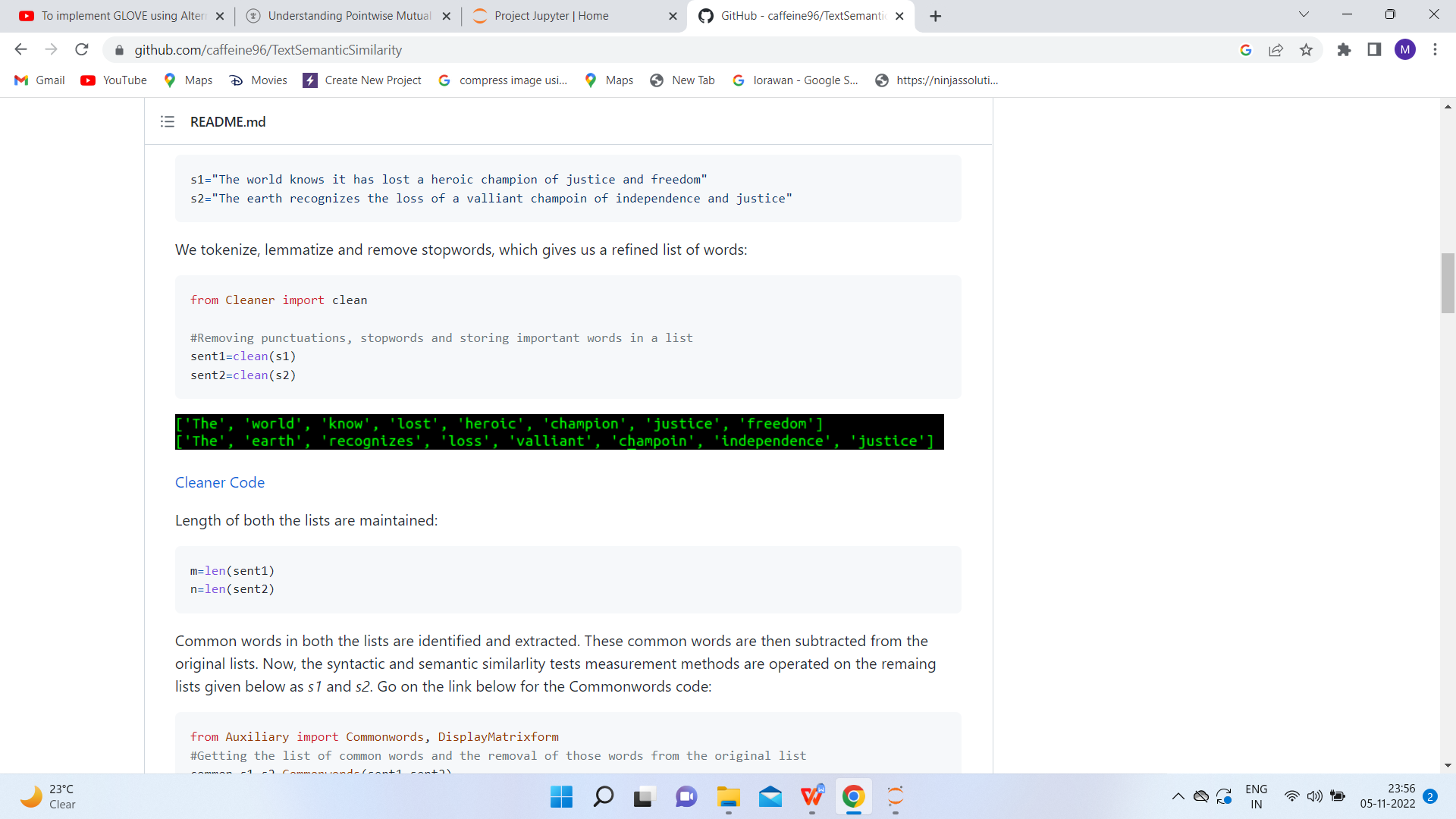
PMI (especially in its positive pointwise mutual information variant) has been described as "one of the most important concepts in NLP", where it "draws on the intuition that the best way to weigh the association between two words is to ask how much more the two words co-occur in a corpus than we would have a priori expected them to appear by chance.



**Code**-

from Cleaner import clean

#Removing punctuations, stopwords and storing important words in a listsent1=clean(s1)sent2=clean(s2)



m=len(sent1)

n=len(sent2)

from Auxiliary import Commonwords, DisplayMatrixform

#Getting the list of common words and the removal of those words from the original list

common,s1,s2=Commonwords(sent1,sent2)

IMG_256

synmat={}if len(s2)<len(s1):

for i in s2:

synmat[i]={}

for j in s1:

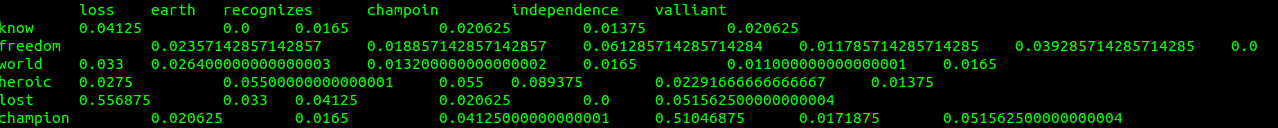
synmat[i][j]=SyntacticSimilarity(i,j)else:

for i in s1:

synmat[i]={}

for j in s2:

synmat[i][j]=SyntacticSimilarity(i,j)



if len(s2)<len(s1):

for i in s2:

semmat[i]={}

for j in s1:

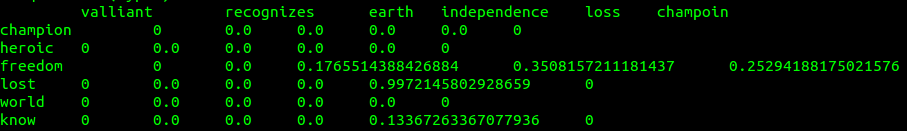
semmat[i][j]=SemanticSimilarity(i,j,a,delta,gamma)else:

for i in s1:

semmat[i]={}

for j in s2:

semmat[i][j]=SemanticSimilarity(i,j,a,delta,gamma)



IMG_256

sentencesimilarity= (len(common)+rhosum)\*(m+n)/(2\*m\*n)

print(sentencesimilarity)

IMG_256

#Lemmatizing words and adding to the final array if they are not stopwordsfor w in words:

if w not in stop\_words:

w=lemmatizer.lemmatize(w)

filtered\_text.append(w)

IMG_256

typefr=collections.Counter()

for w in filtered\_text:

typefr[w]+=1

neighboursw1=collections.Counter()

n2w1=[]

for i in range(len(filtered\_text)):

if w1==filtered\_text[i]:

neighboursw1[filtered\_text[i]]+=1

for j in range(0,a+1):

neighboursw1[filtered\_text[i+j]]+=1

neighboursw1[filtered\_text[i-j]]+=1

pmiw1={}

for t in neighboursw1.keys():

pmiw1[t]= math.log(neighboursw1[t]\*m/(typefr[t]\*typefr[w1]),2)pmiw1\_sorted = sorted(pmiw1, key=pmiw1.get, reverse=True)

b1= math.floor((math.pow(math.log10(typefr[w1]),2)\* math.log(len(unique),2))/delta)

for i in range(0,b1):

for j in range(0,b2):

if pmiw1\_sorted[i]==pmiw2\_sorted[j]:

betasumw1+=math.pow(pmiw2[pmiw1\_sorted[i]],gamma)

similarity= betasumw1/b1 + betasumw2/b2

target=open("Lambda.txt","r")

lmbda=float(target.read())

target.close()

similarity,lmbda= normalized\_similarity(similarity,lmbda)

target=open("Lambda.txt","w")

target.write(str(lmbda))

target.close()

return similarity

#Returning Normalized Similaritydef normalized\_similarity(similarity,lmbda):

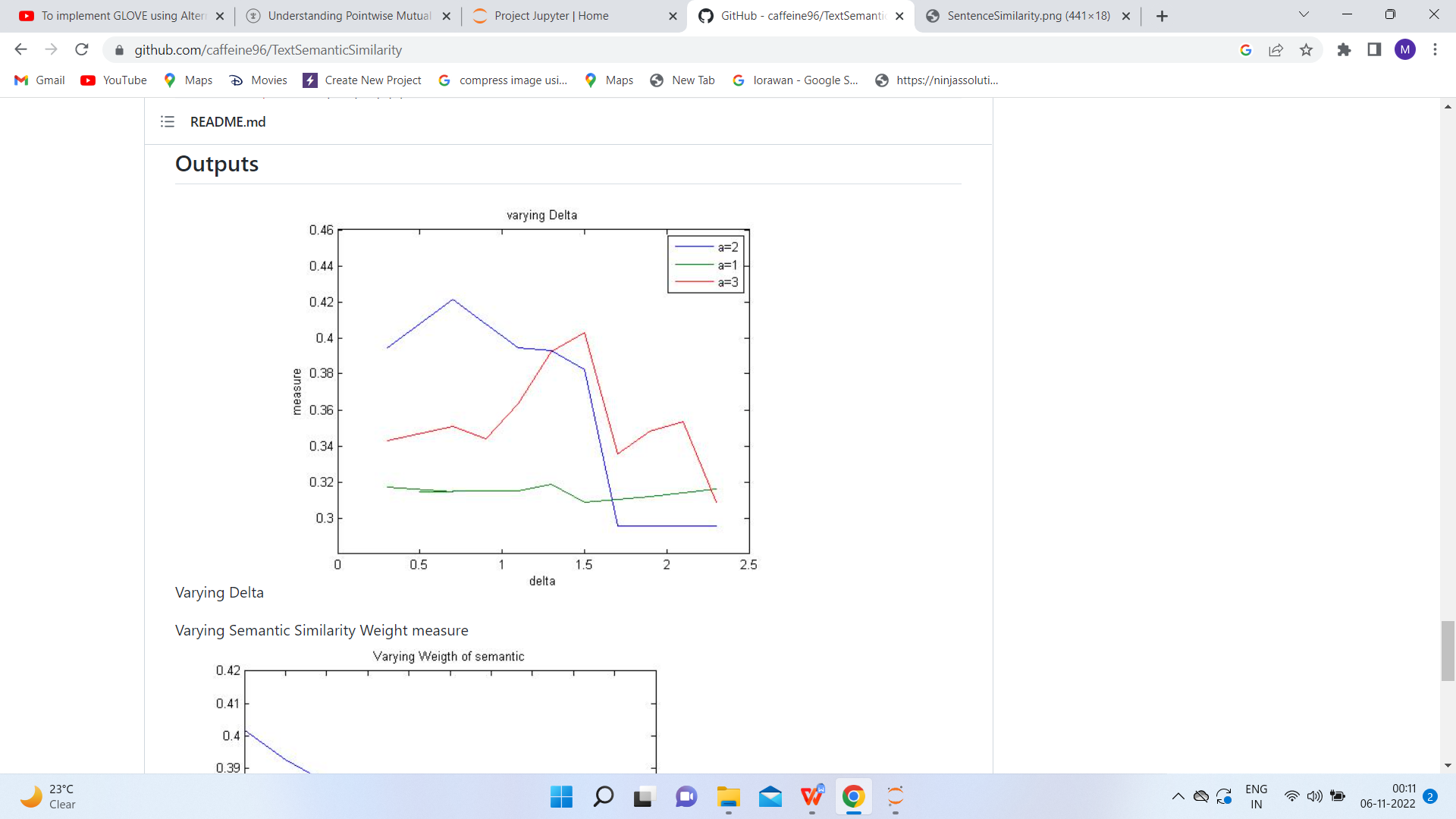
if similarity>lmbda:

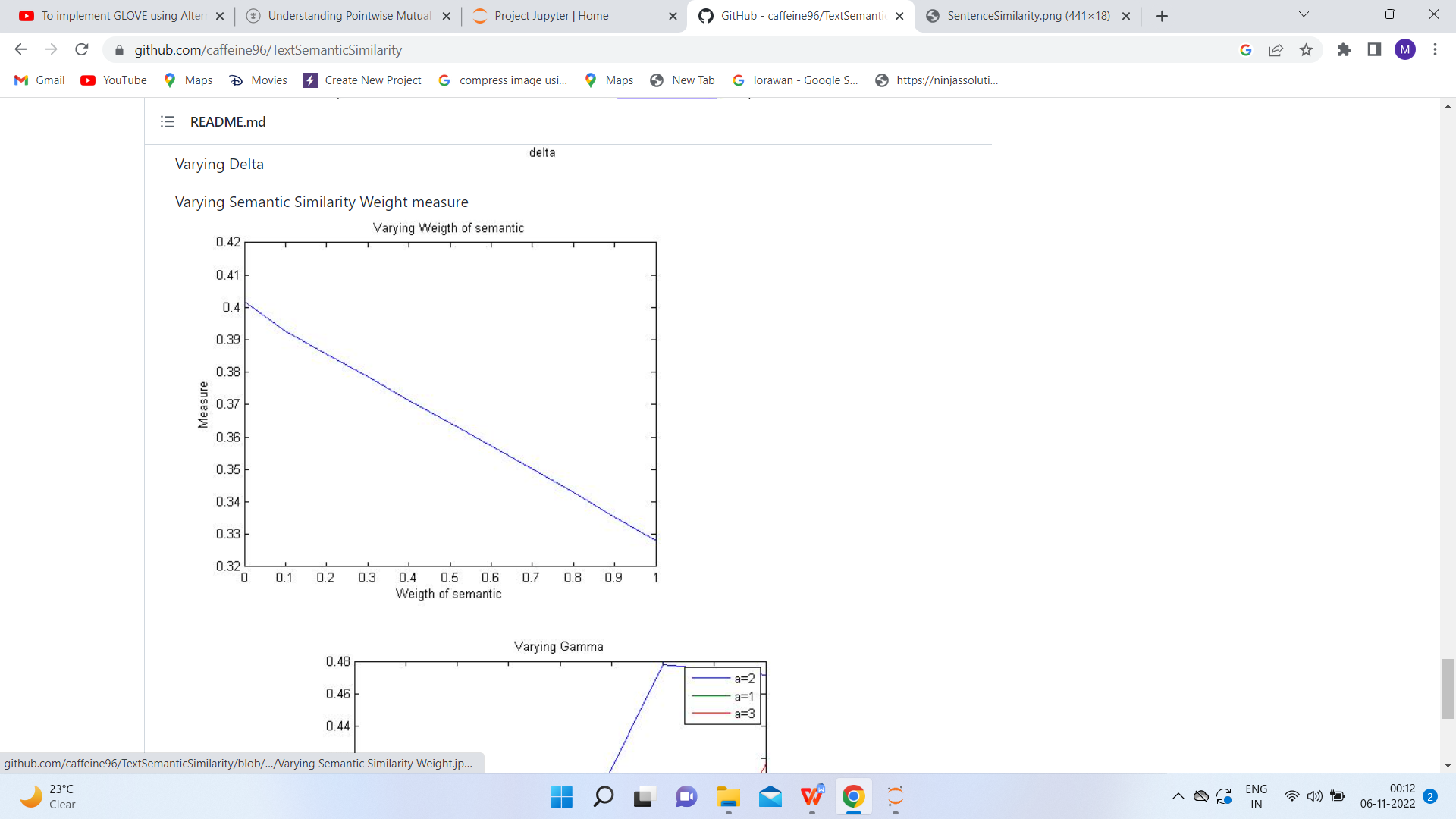
lmbda=math.ceil(similarity)

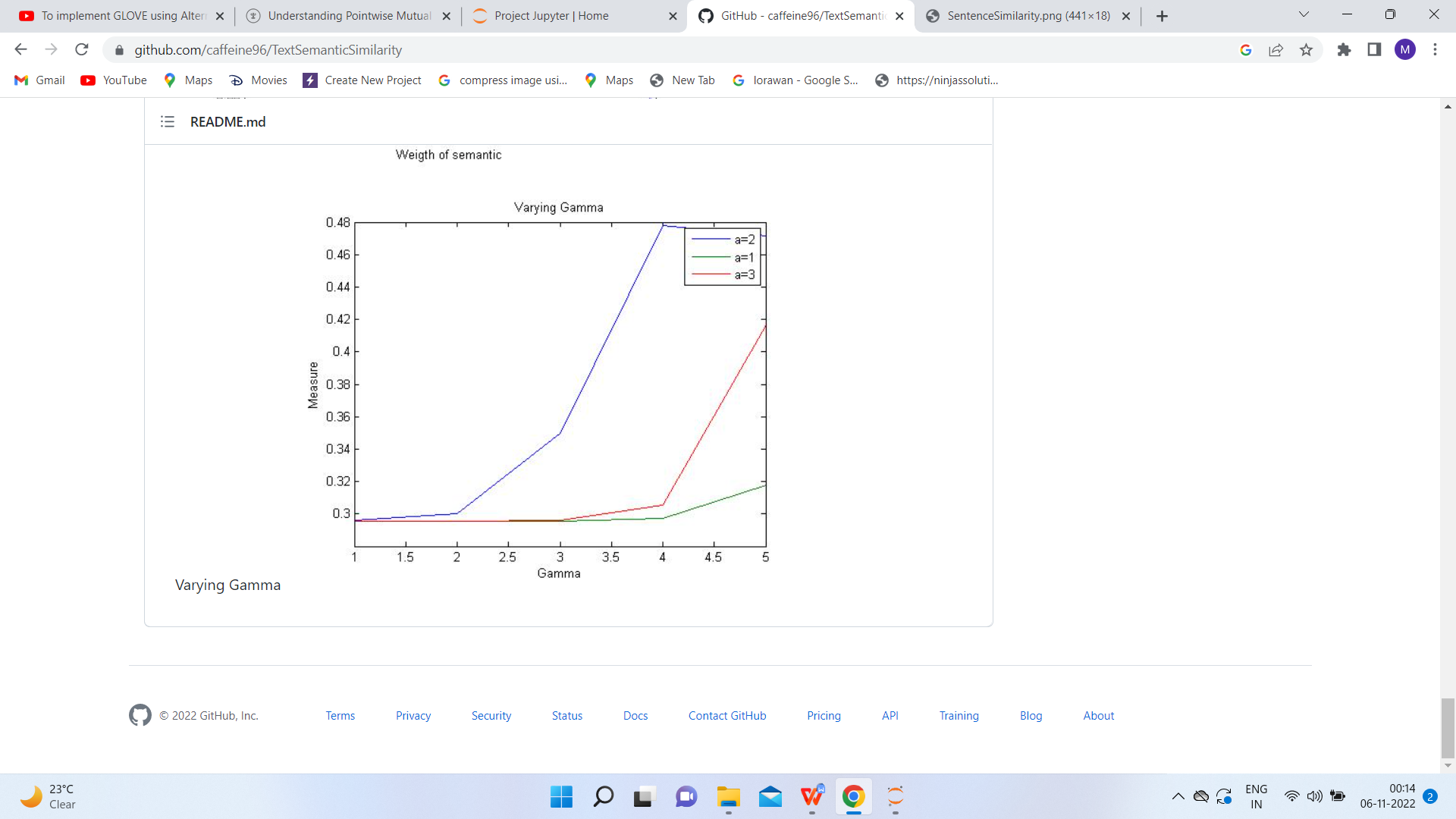
similarity/=lmbda

return similarity,lmbda

**Output**-







**Results**- All the concepts related to the aim are fully understood and applied correctly.

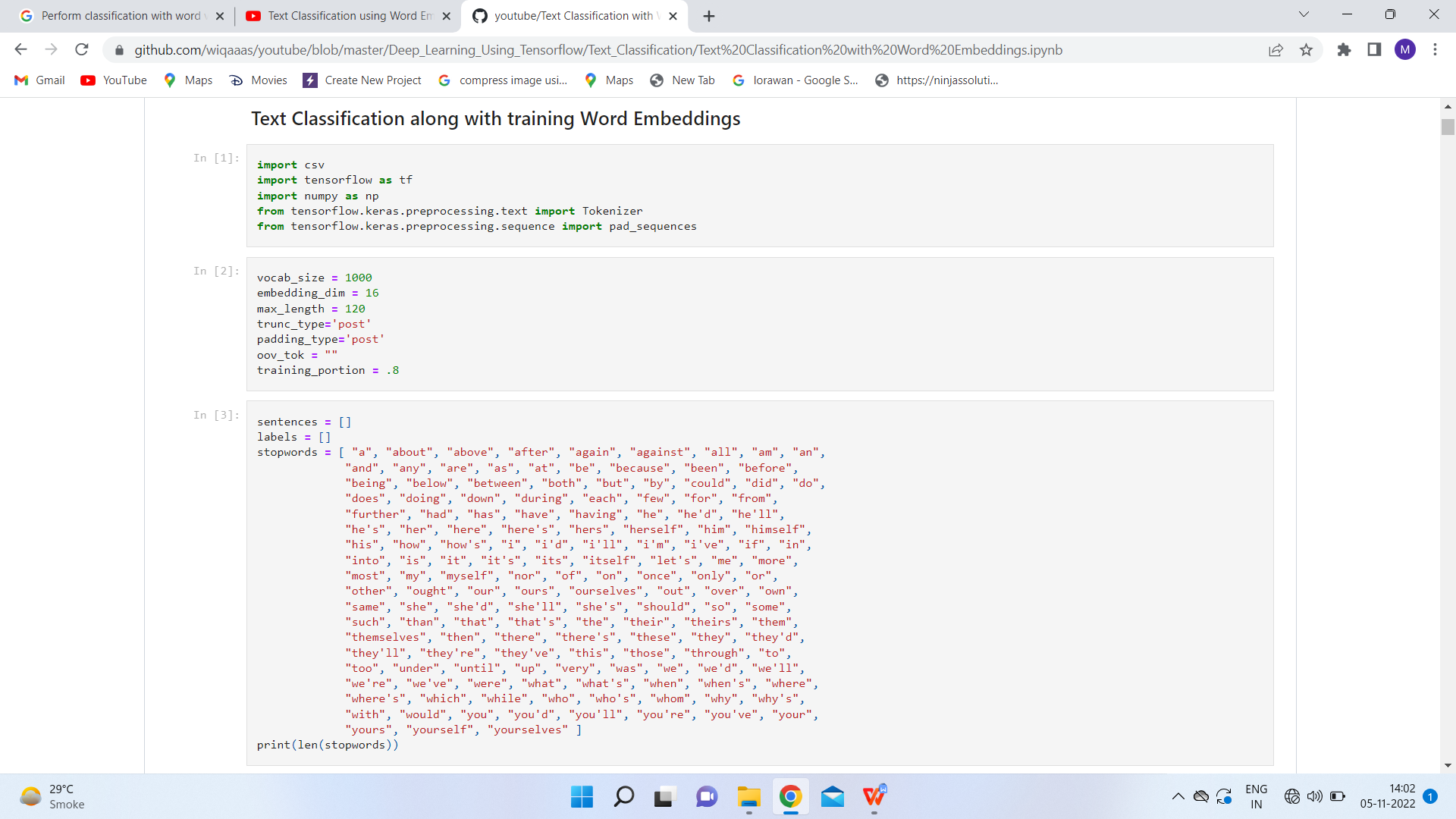
**Experiment-10**

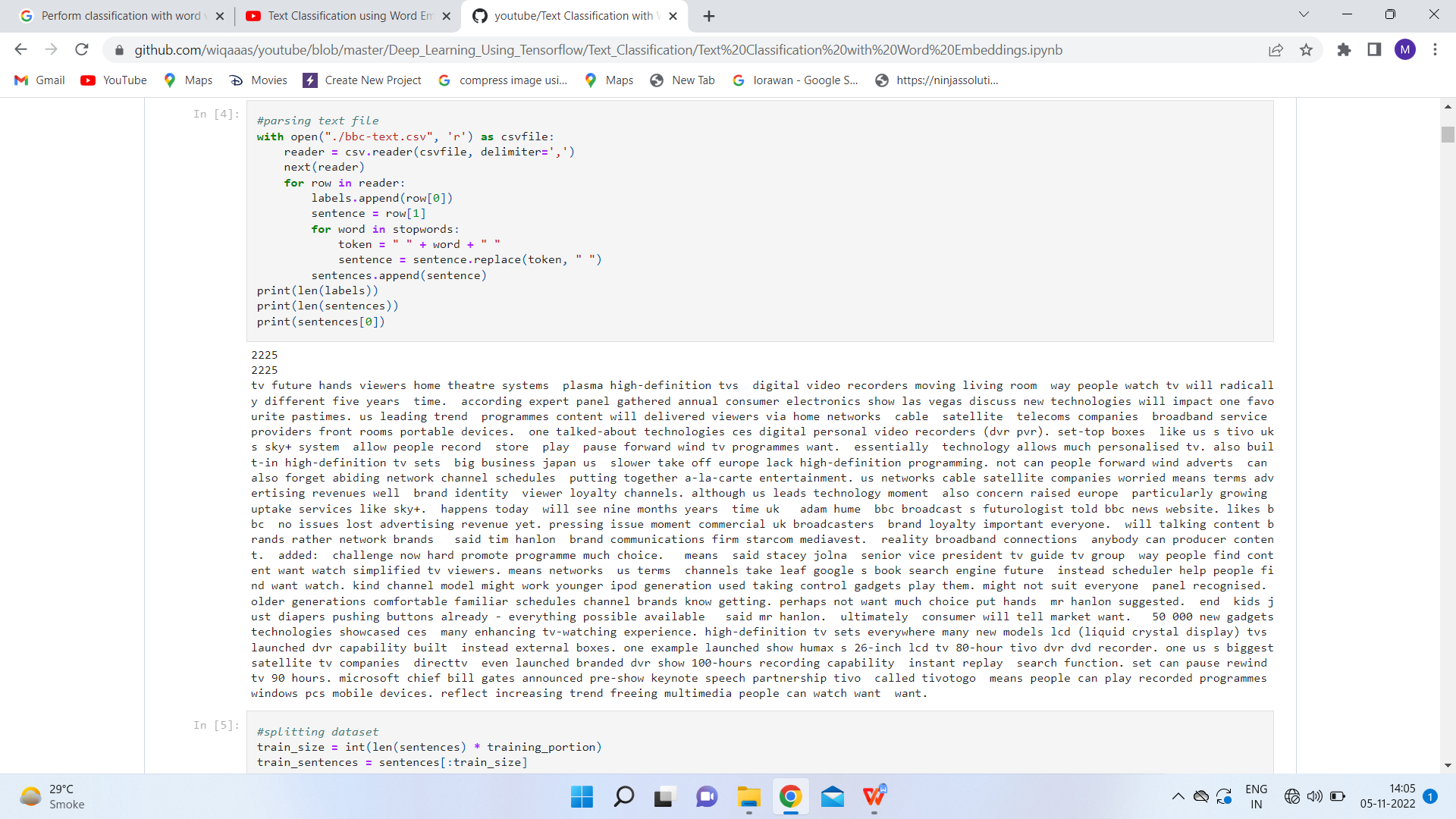
**Aim**- To perform classification with word vectors.

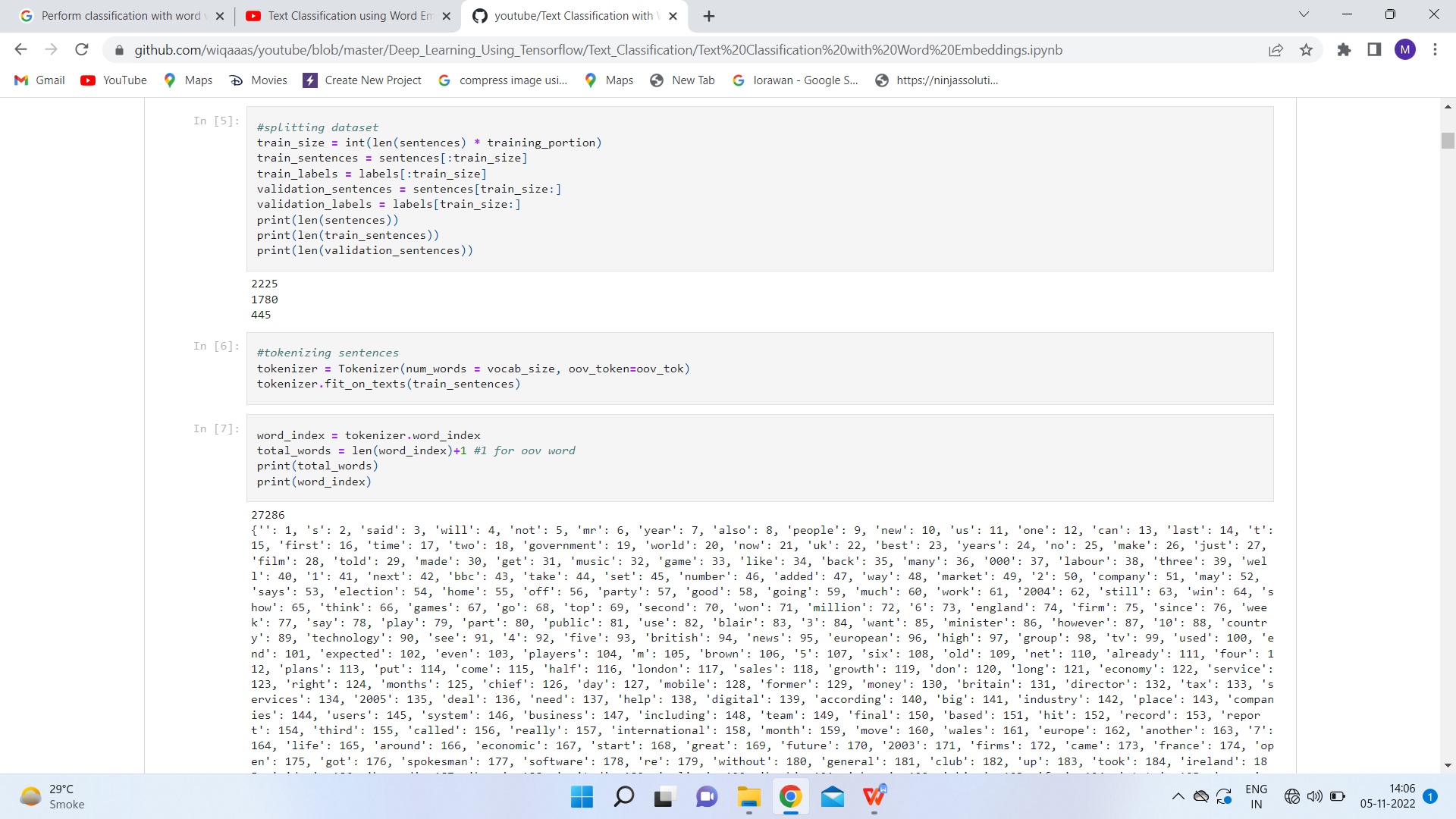
**Theory**- A word vector is an attempt to mathematically represent the meaning of a word. In essence, a computer goes through some text (ideally a lot of text) and calculates how often words show up next to each other. These frequencies are represented with numbers. So if the word ‘good’ always shows up next to the word ‘friend’ then part of the word vector for ‘good’ will reflect that connection. A given word will have a vast number of such values, usually in the hundreds and sometimes in the thousands.

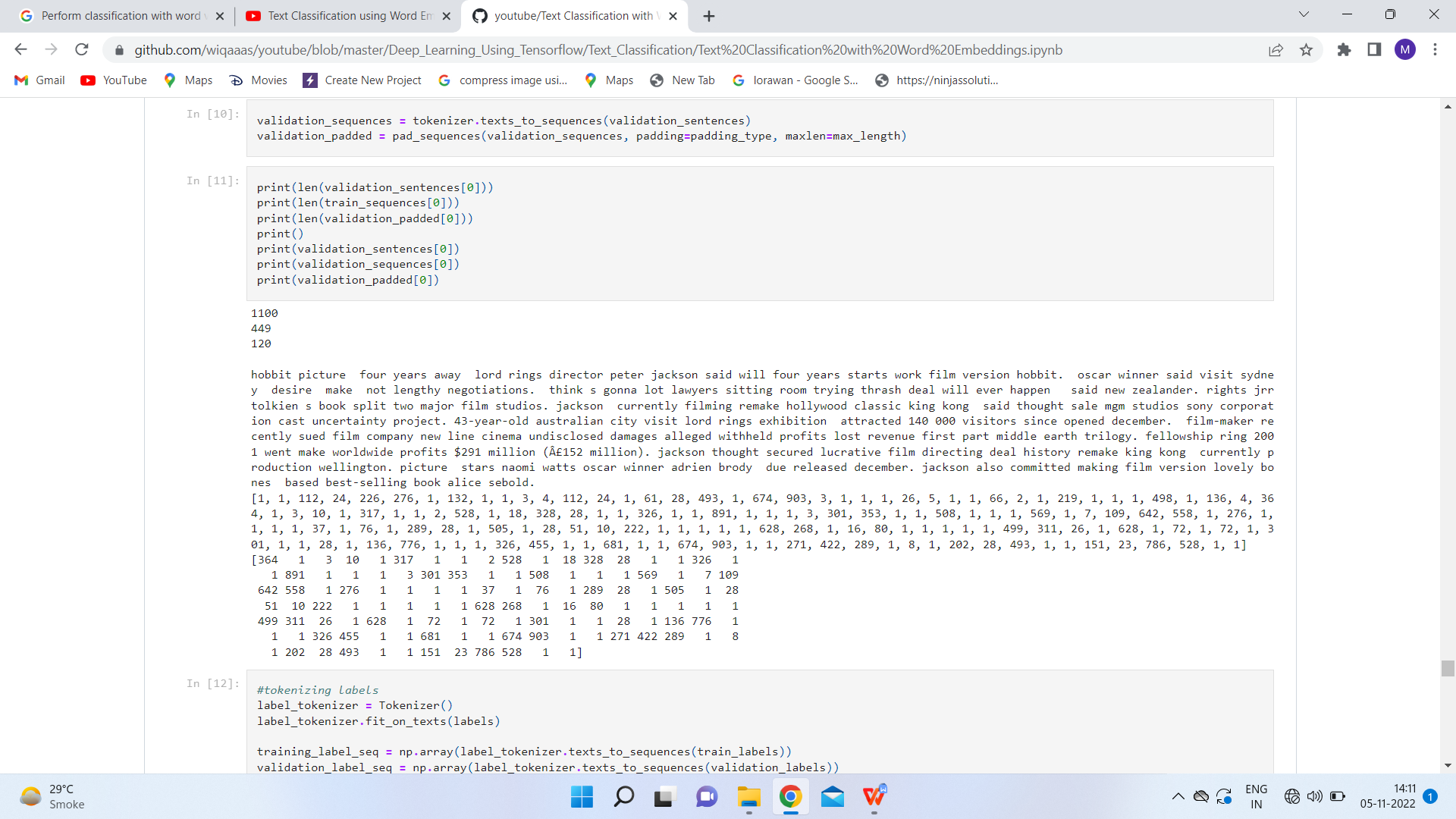
Once you have this set of numbers for a word, you can compare the vectors of different words. For example, you could compare the different vectors for the words ‘banana,’ ‘kiwi,’ and ‘raincloud.’ Since the vectors are mathematical objects, you can calculate the numerical similarity of different vectors. And since ‘banana’ and ‘kiwi’ are used in a lot of similar contexts, and are not used in the contexts where ‘raincloud’ shows up, their vectors will be closer to each other and farther from the vector for ‘raincloud.’

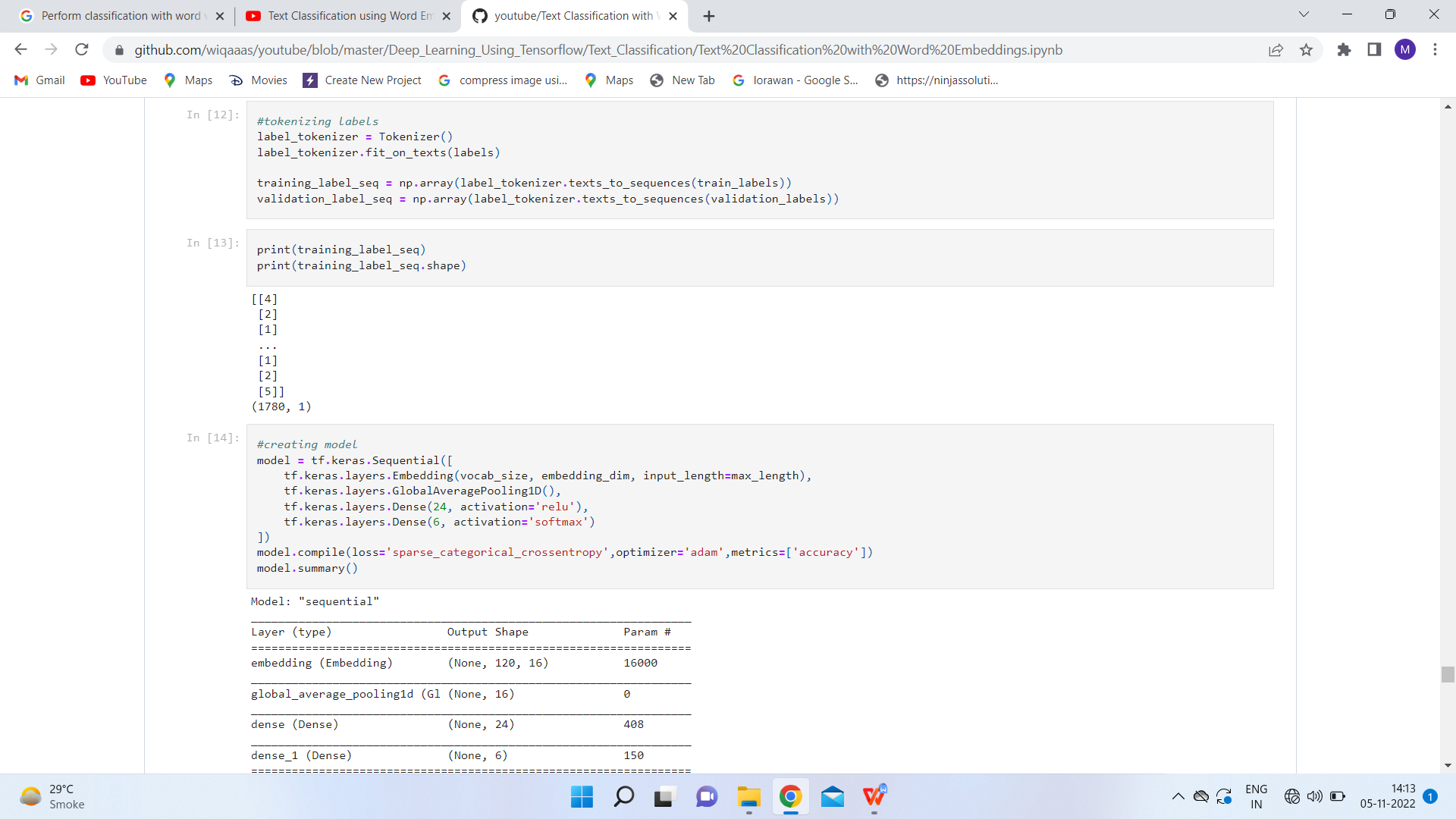
**Code**-

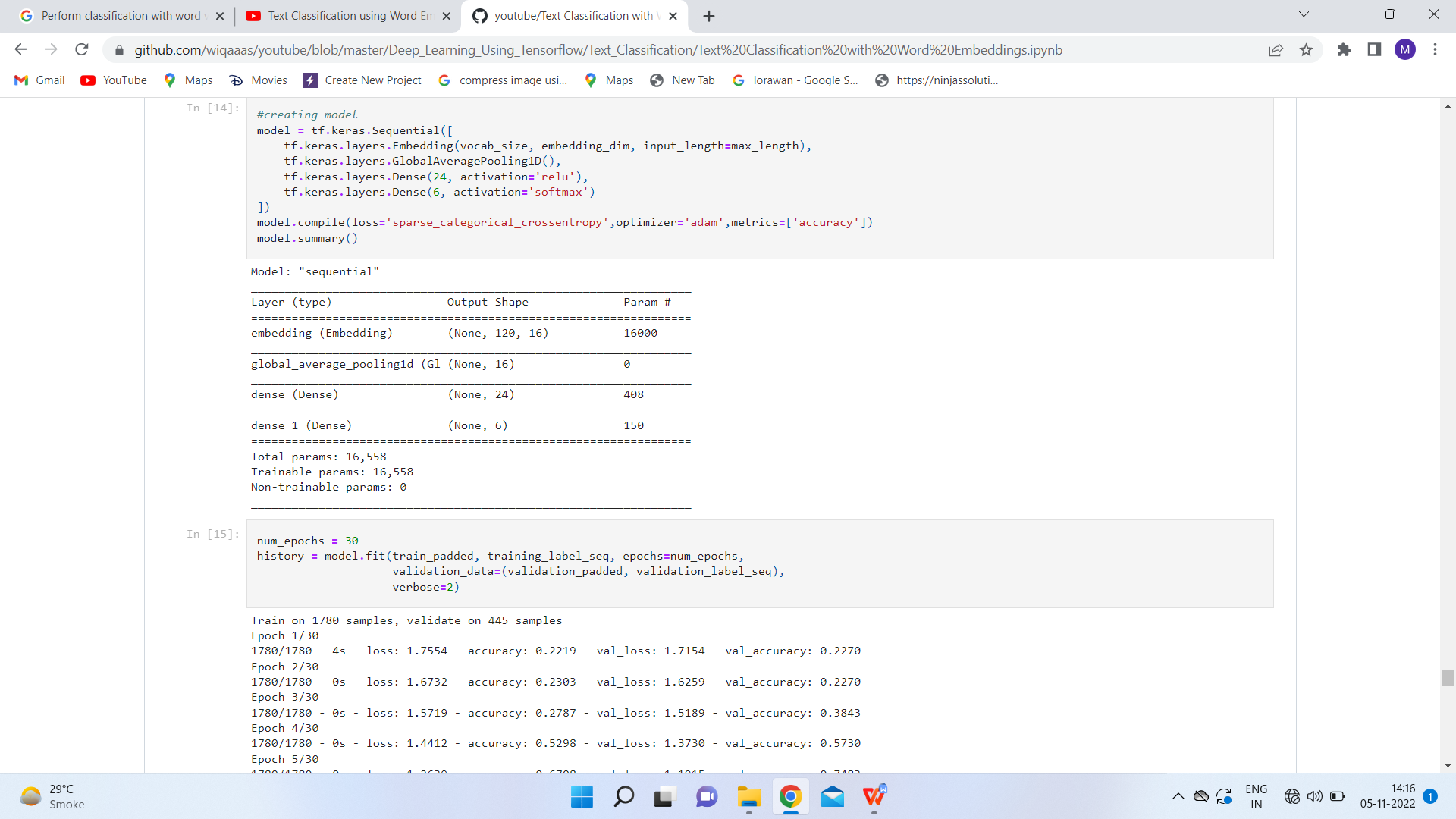


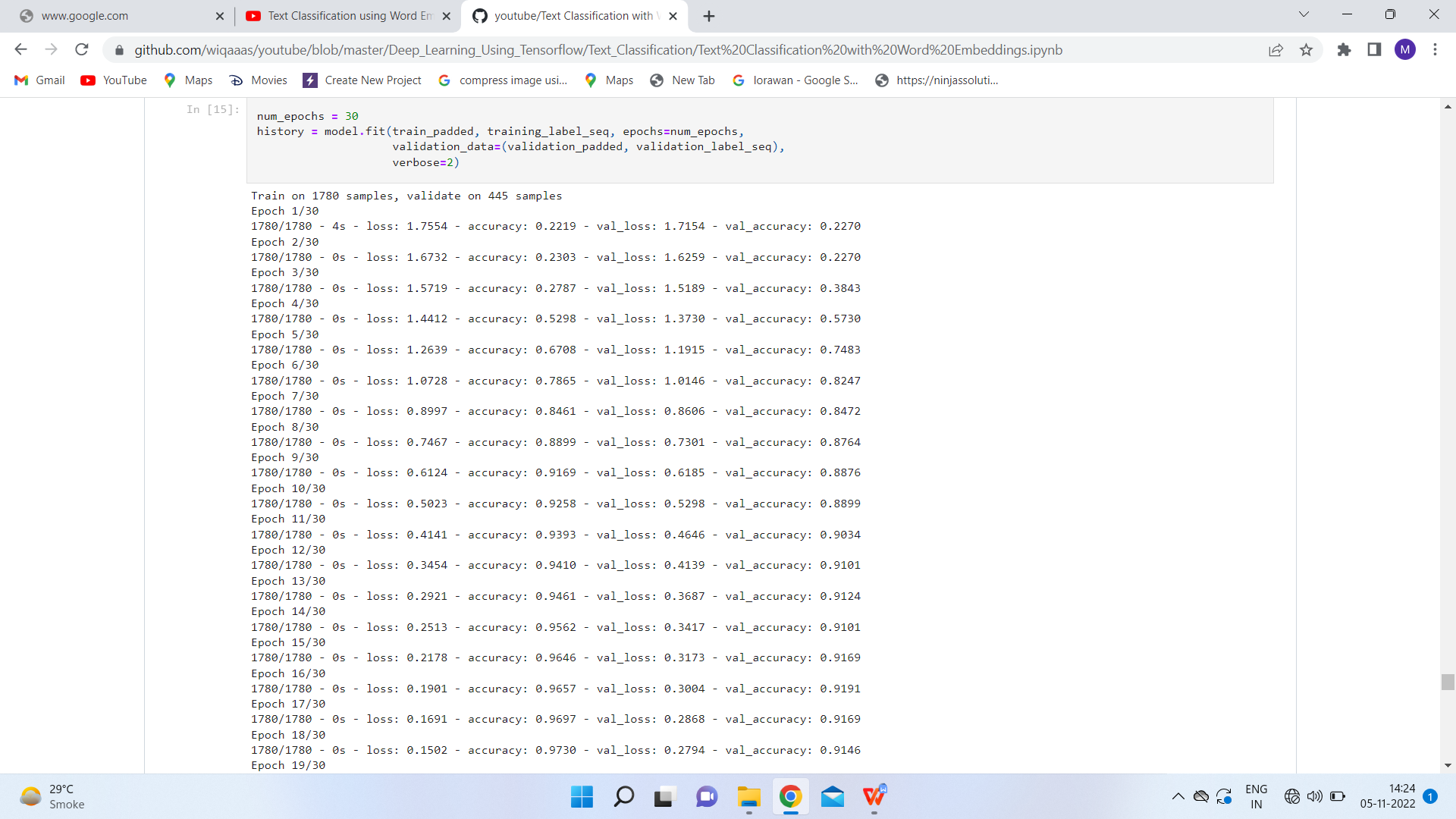


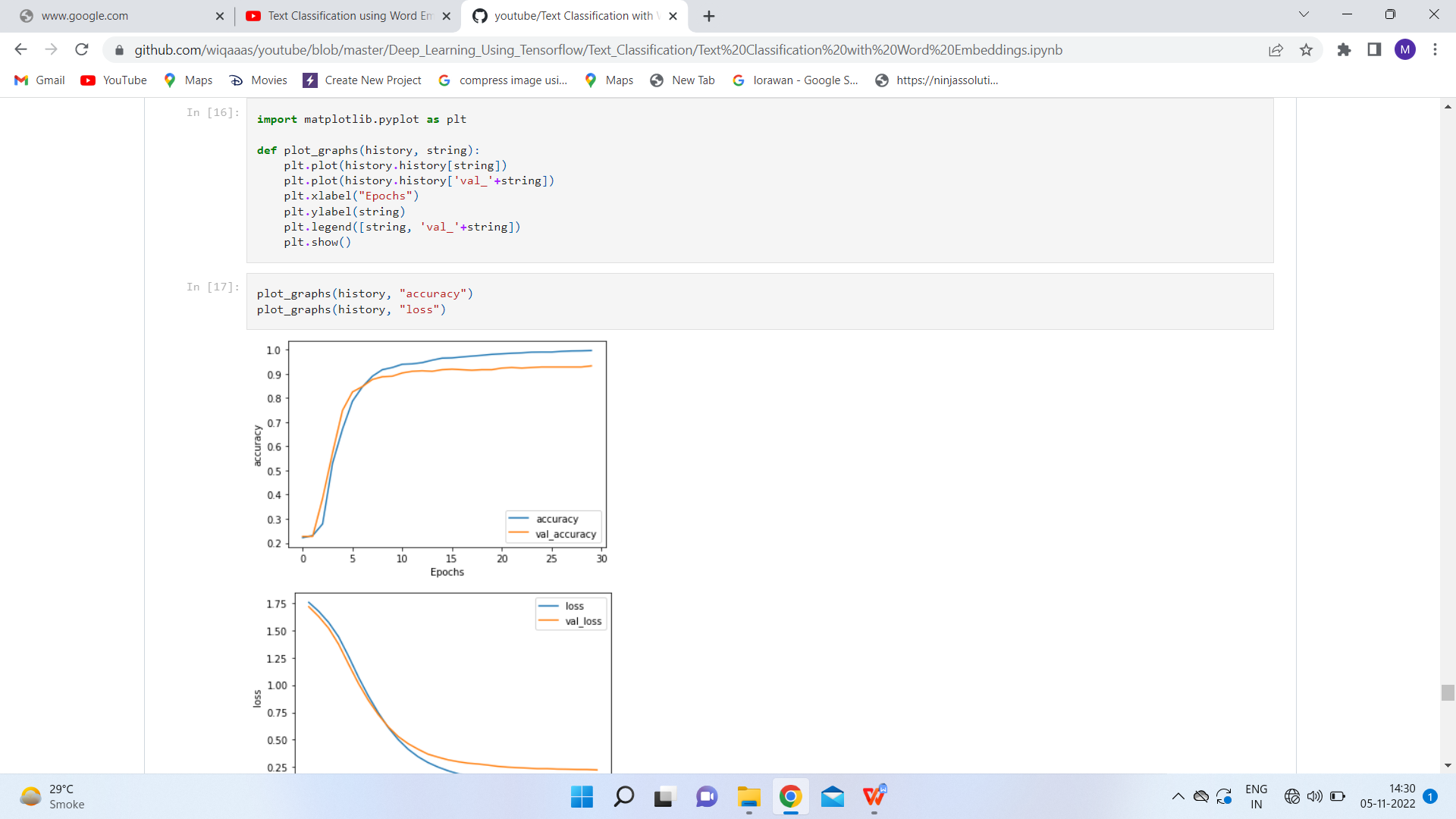




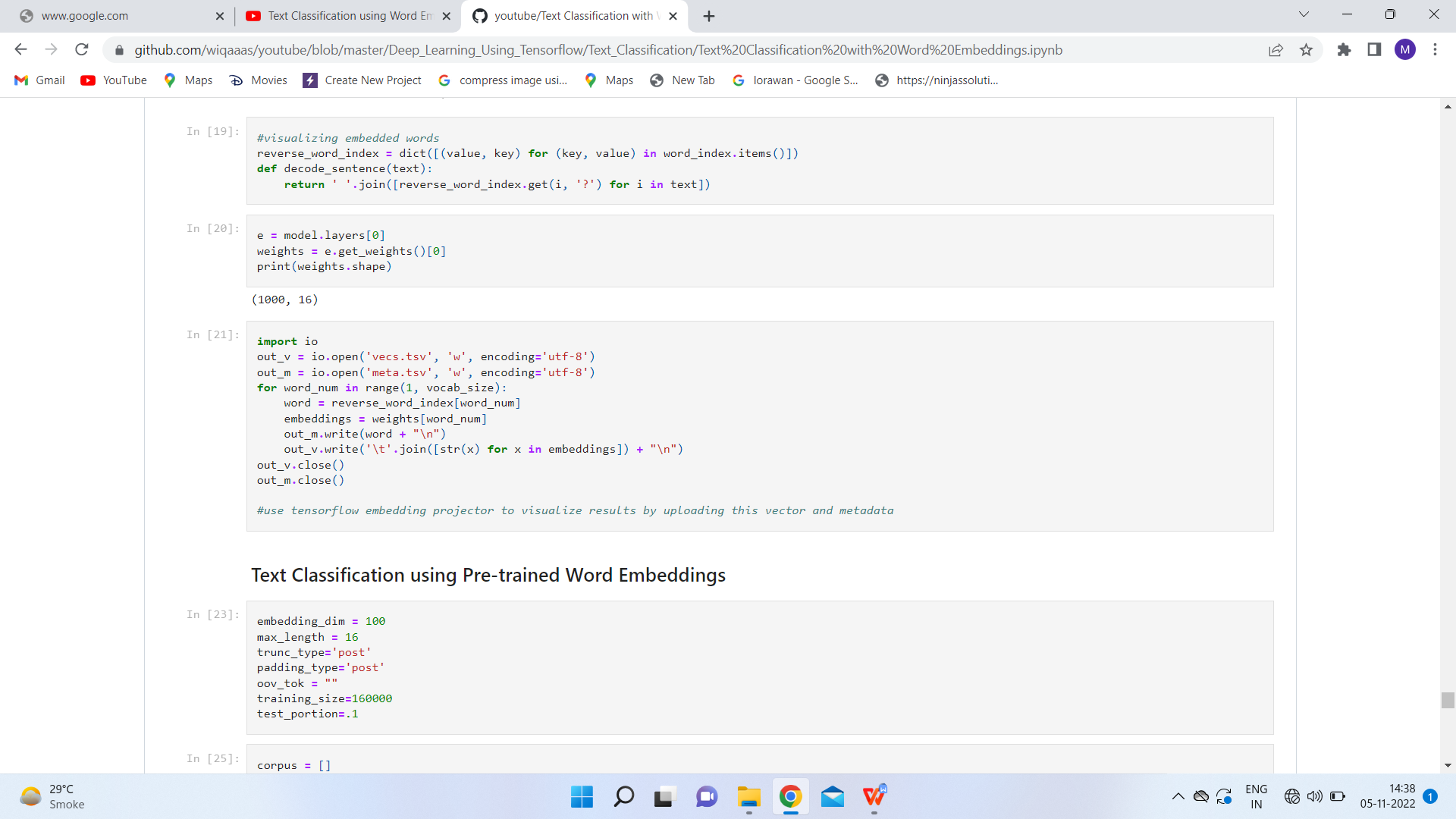


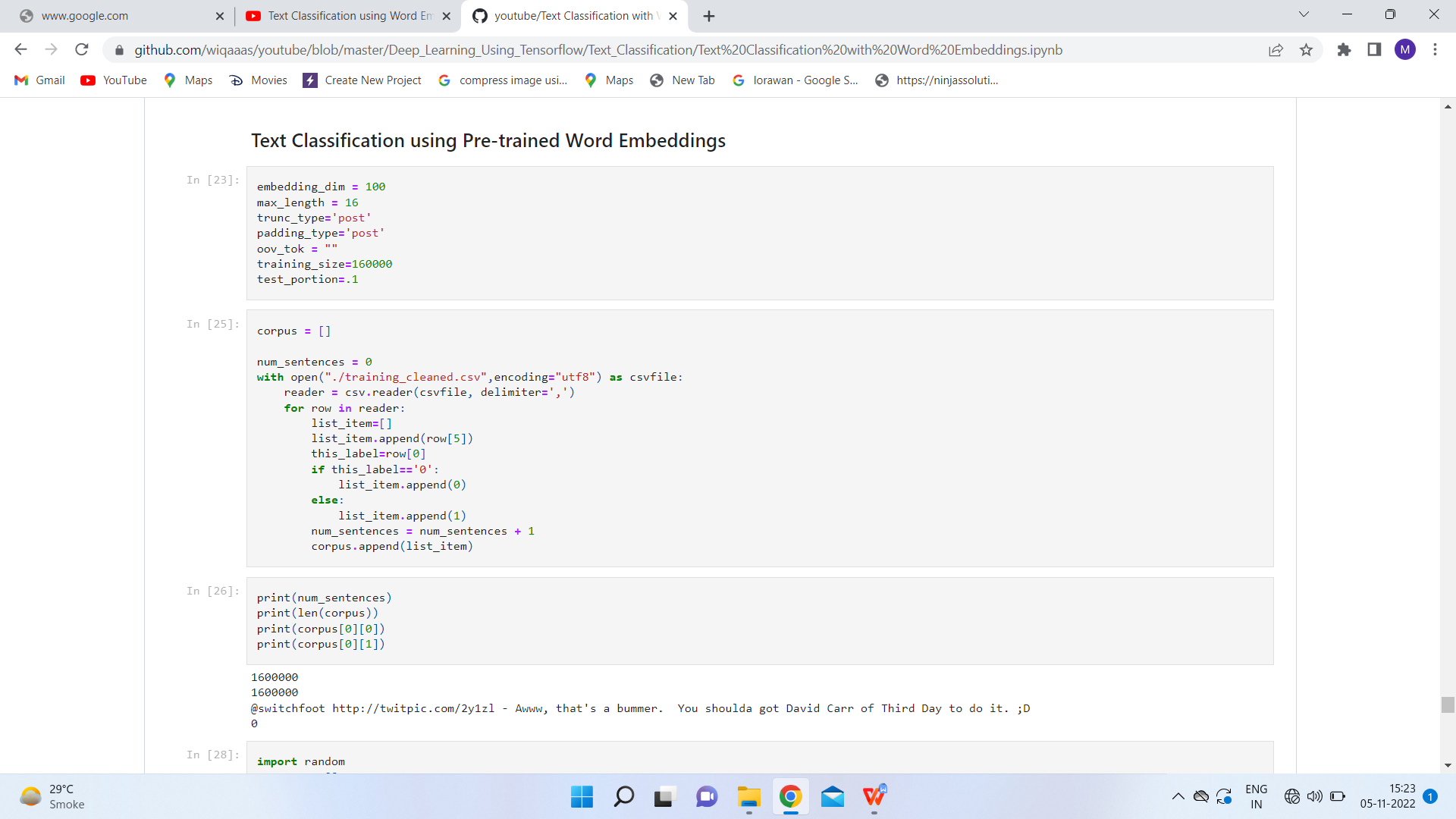


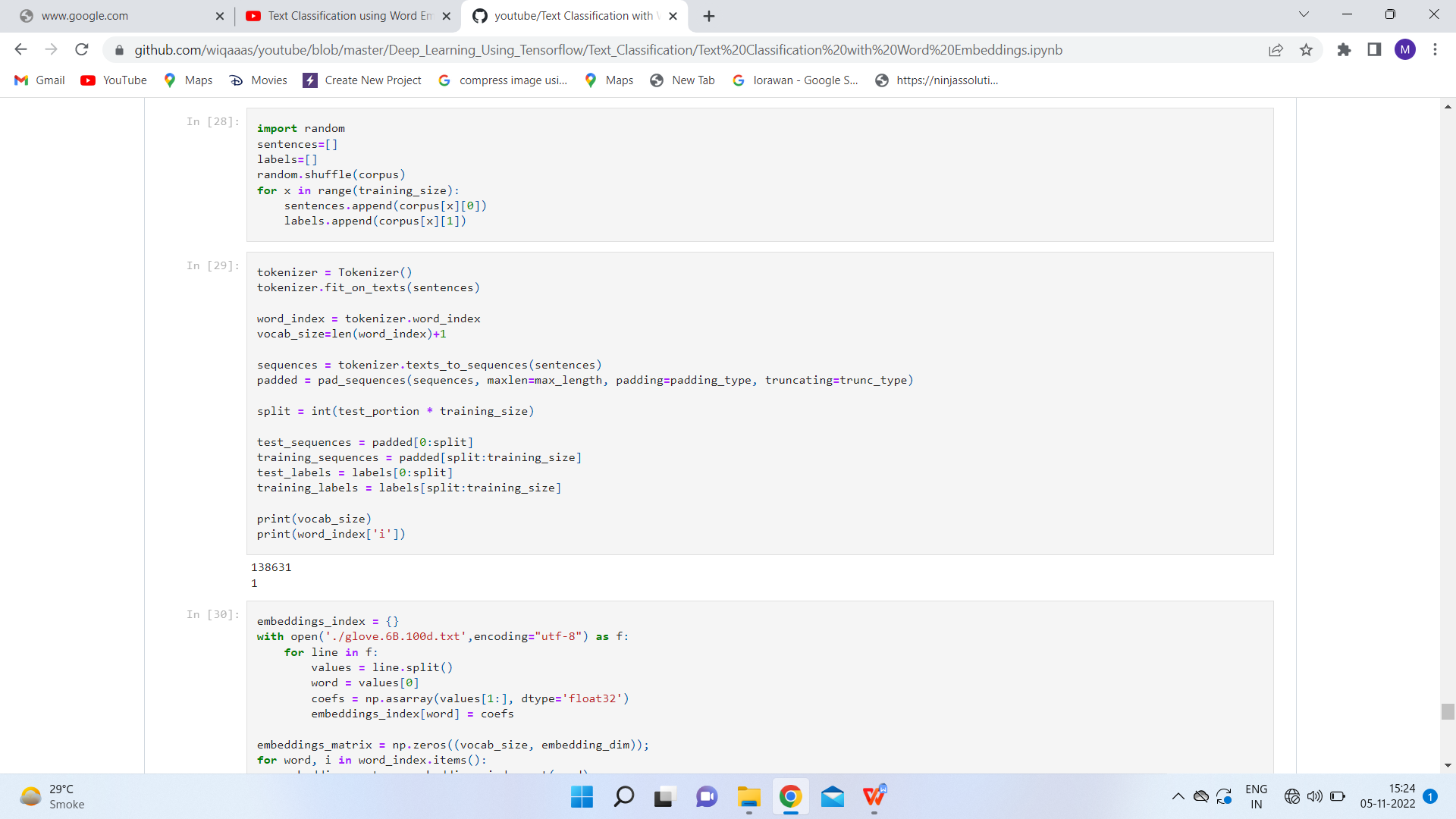


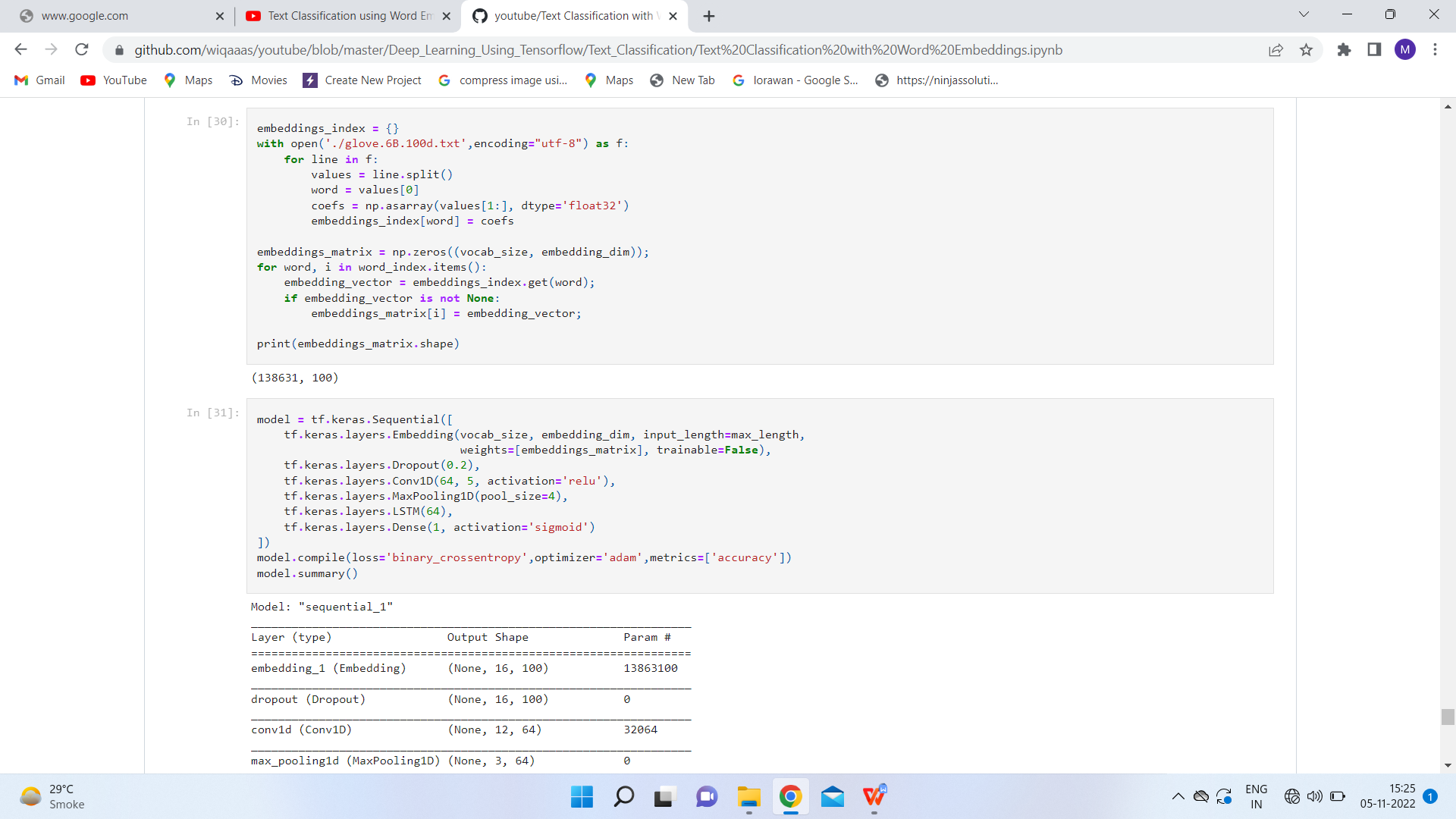


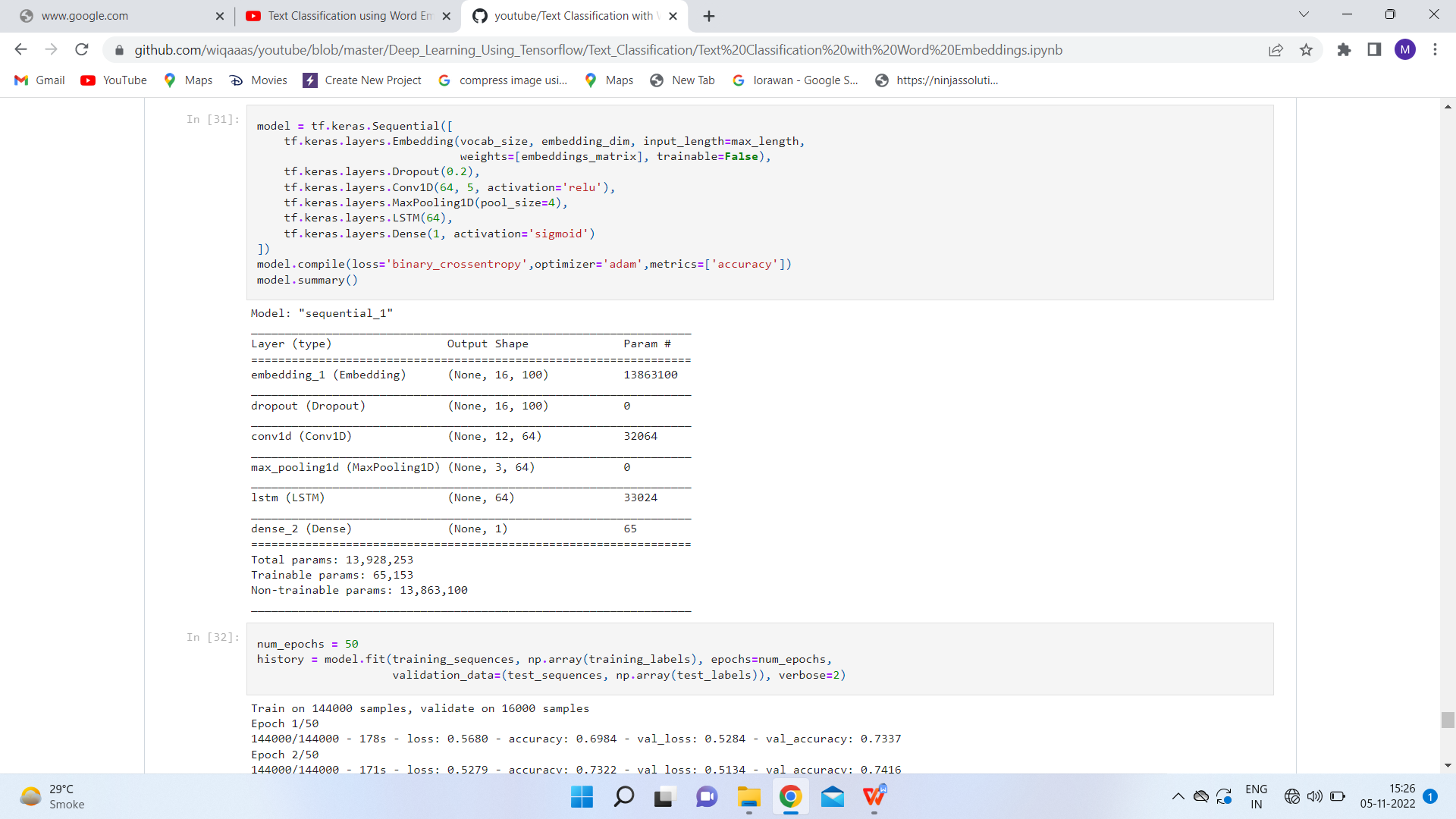


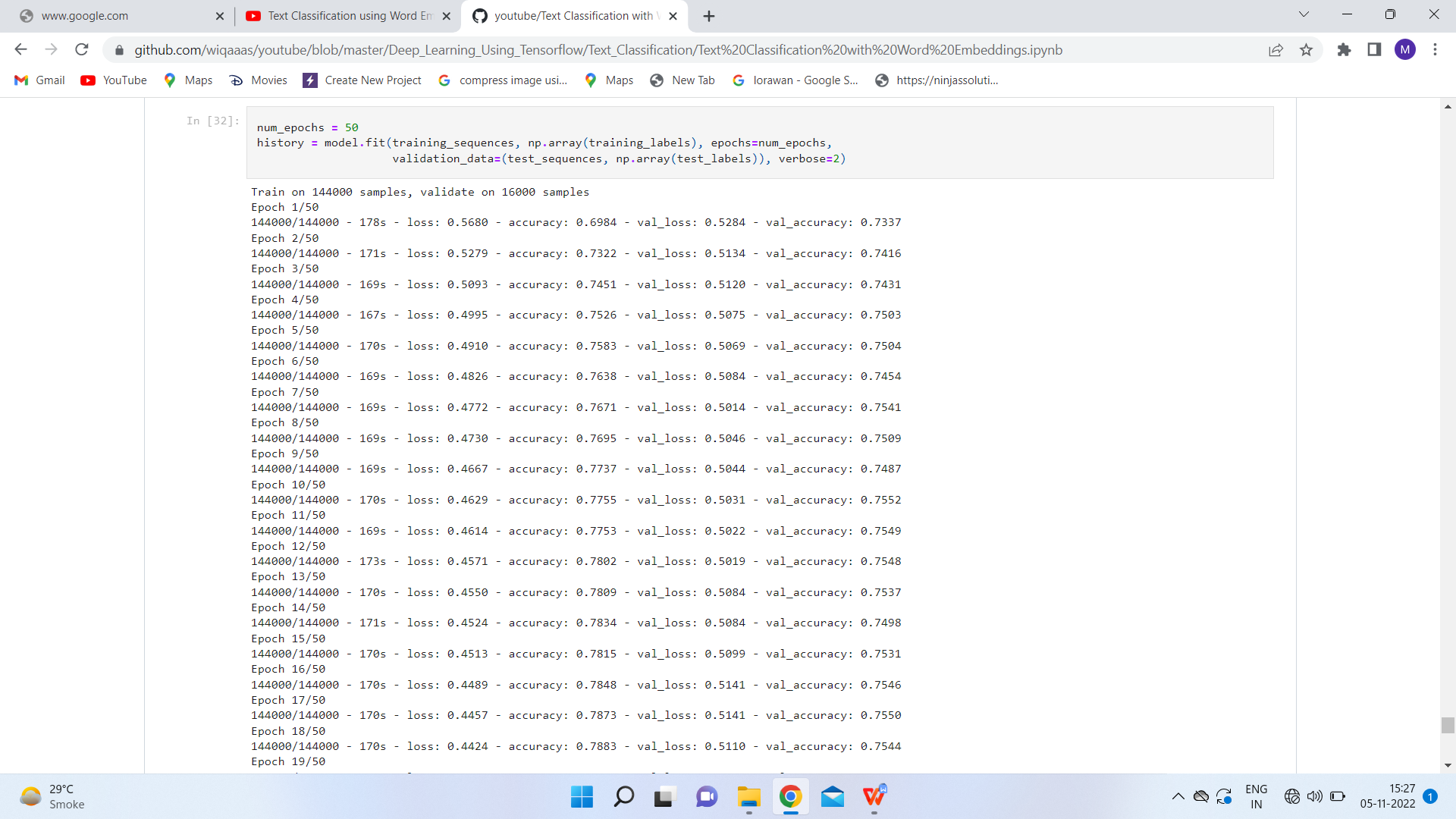


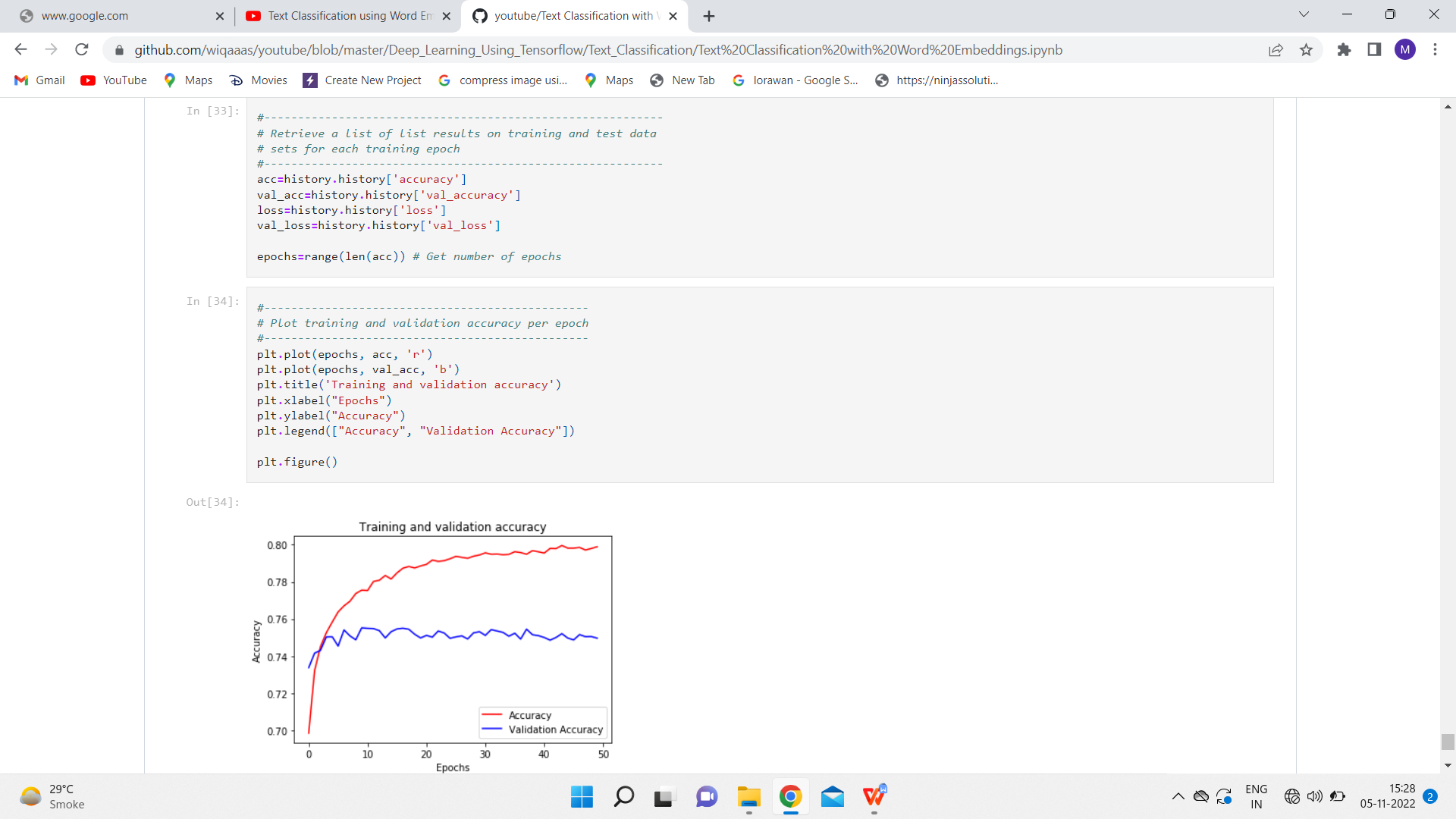


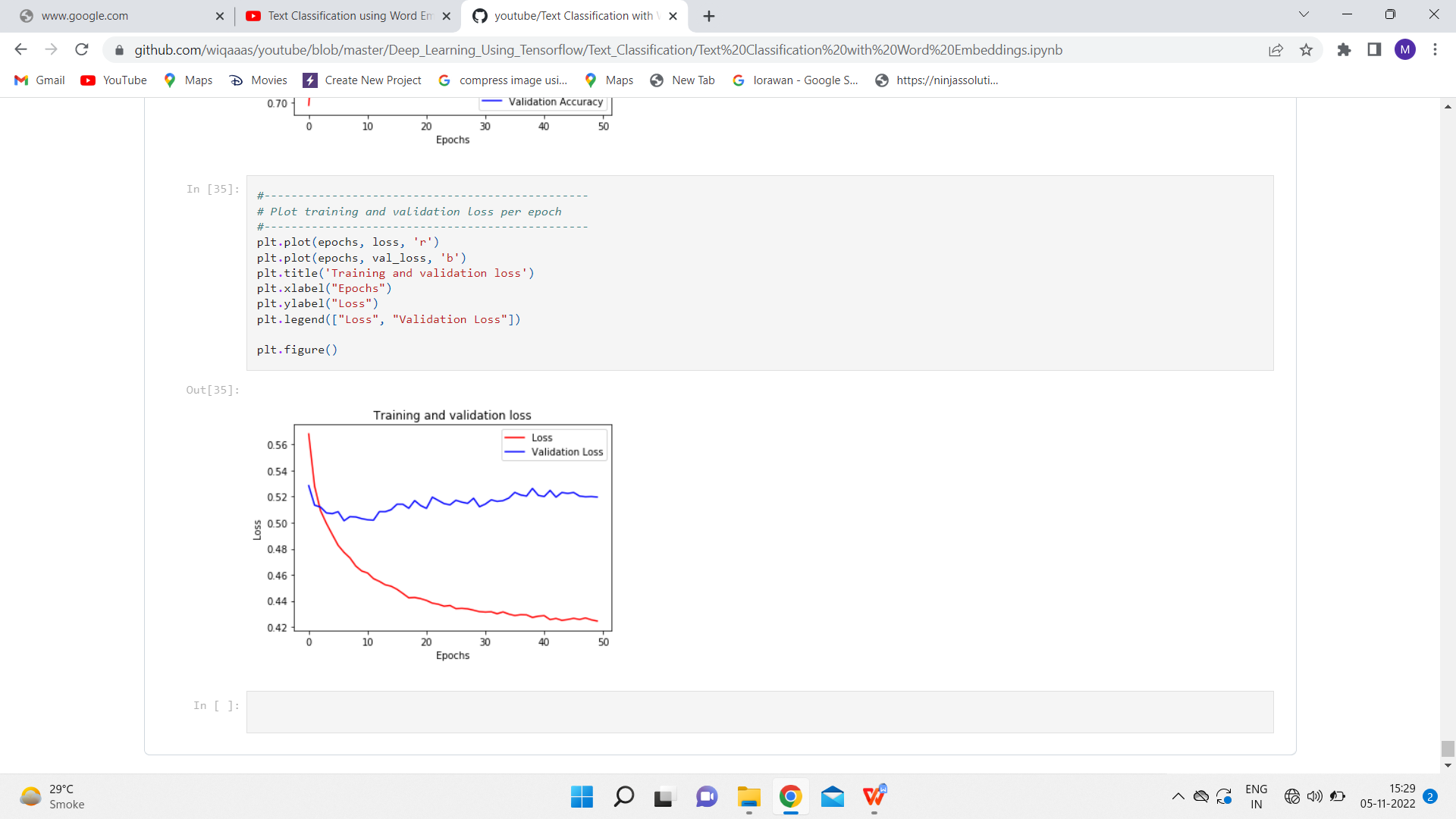












**Result**- All the concepts related to the aim are fully understood and applied correctly.