**Experiment 1**

**AIM**

To perform classification with word vectors.

**Software used**

Jupyter Notebook

**Dataset Used**

IMDB dataset from Hugging Face's datasets library

**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**

|  |
| --- |
| from datasets import load\_dataset  from sklearn.feature\_extraction.text import TfidfVectorizer  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn.metrics import accuracy\_score, confusion\_matrix  import matplotlib.pyplot as plt  import random  # Load the IMDB dataset from Hugging Face's datasets library  dataset = load\_dataset("imdb")  # Get the total number of examples in the dataset  total\_examples = len(dataset["train"]["text"])  # Randomly pick 5 indices  random\_indices = random.sample(range(total\_examples), 5)  # Extract random reviews and labels  random\_reviews = [dataset["train"]["text"][i] for i in random\_indices]  random\_labels = [dataset["train"]["label"][i] for i in random\_indices]  # Display the randomly picked samples  print(f"Sample Dataset:")  for review, label in zip(random\_reviews, random\_labels):      print(f"Label: {label}")      print(f"Review: {review}")      print("=" \* 50)  print()  # Extract text and labels  texts = dataset["train"]["text"]  labels = dataset["train"]["label"]  # Split the data into training and testing sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(texts, labels, test\_size=0.2, random\_state=42)  # Convert text data to feature vectors using TfidfVectorizer  vectorizer = TfidfVectorizer()  X\_train\_vectors = vectorizer.fit\_transform(X\_train)  X\_test\_vectors = vectorizer.transform(X\_test)  # Build and train a logistic regression classifier with increased max\_iter  classifier = LogisticRegression(max\_iter=1000)  # Increase max\_iter  classifier.fit(X\_train\_vectors, y\_train)  # Make predictions on the test set  predictions = classifier.predict(X\_test\_vectors)  # Display a few samples with predictions and actual labels  print(f"Some Predictions:")  for i in range(5):      print(f"Text: {X\_test[i]}")      print(f"Prediction: {predictions[i]}, Actual: {y\_test[i]}")      print()  # Plot confusion matrix  cm = confusion\_matrix(y\_test, predictions)  plt.imshow(cm, interpolation="nearest", cmap=plt.cm.Blues)  plt.title("Confusion Matrix")  plt.colorbar()  plt.xlabel("Predicted Label")  plt.ylabel("True Label")  plt.show()  # Plot loss curve  plt.plot(range(len(classifier.coef\_[0])), classifier.coef\_[0])  plt.title("Coefficient Curve")  plt.xlabel("Iteration")  plt.ylabel("Coefficient Value")  plt.show()  # Calculate and plot accuracy  train\_predictions = classifier.predict(X\_train\_vectors)  test\_predictions = classifier.predict(X\_test\_vectors)  train\_accuracy = accuracy\_score(y\_train, train\_predictions)  test\_accuracy = accuracy\_score(y\_test, test\_predictions)  # Plot accuracy curves  plt.plot(range(len(classifier.coef\_[0])), [train\_accuracy] \* len(classifier.coef\_[0]), label='Train', linestyle='--')  plt.plot(range(len(classifier.coef\_[0])), [test\_accuracy] \* len(classifier.coef\_[0]), label='Test', linestyle='--')  plt.title("Accuracy Curves")  plt.xlabel("Iteration")  plt.ylabel("Accuracy")  plt.legend()  plt.show()  # Print accuracy values  print(f'train\_accuracy: {train\_accuracy}')  print(f'test\_accuracy: {test\_accuracy}') |

**OUTPUT**

A close up of text

Description automatically generated

A close up of text

Description automatically generated

A blue squares with white squares

Description automatically generated A blue graph with numbers and lines

Description automatically generated

A graph with numbers and lines

Description automatically generated A close up of words

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 2**

**AIM**

To implement Neural Network Bigram Model.

**Software used**

Jupyter Notebook

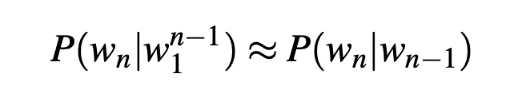
**Dataset Used**

NLTK's Reuters corpus

**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.

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**

A screenshot of a computer

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 3**

**AIM**

Implementation of word2vec using NumPy.

**Software used**

Jupyter Notebook

**Dataset/Corpus Used**

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

**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. Embeddings 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.

A diagram of a diagram

Description automatically generated

**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 and OUTPUT**

|  |
| --- |
| 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) |



A screen shot of a computer

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 4**

**AIM**

Implementation of word2vec using tensorflow.

**Software used**

Jupyter Notebook

**Dataset/Corpus Used**

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

**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**

A screenshot of a computer

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 5**

**AIM**

To implement GLOVE using numpy gradient descent.

**Software used**

Jupyter Notebook

**Dataset/Corpus Used**

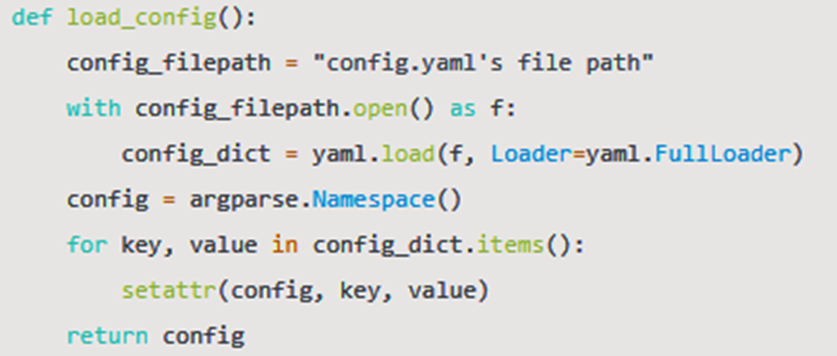
Genism Data – “text8”

**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**

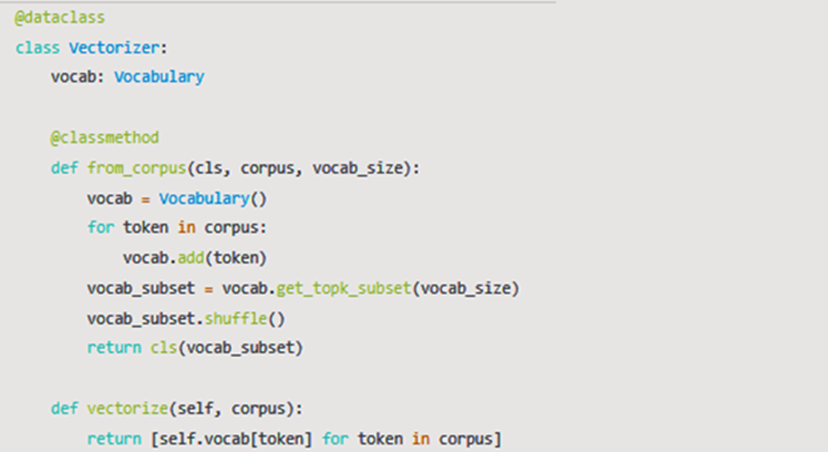
|  |
| --- |
| import gensim.downloader as api  dataset = api.load("text8")  import itertools  corpus = list(itertools.chain.from\_iterable(dataset)) |



A screenshot of a computer code

Description automatically generated



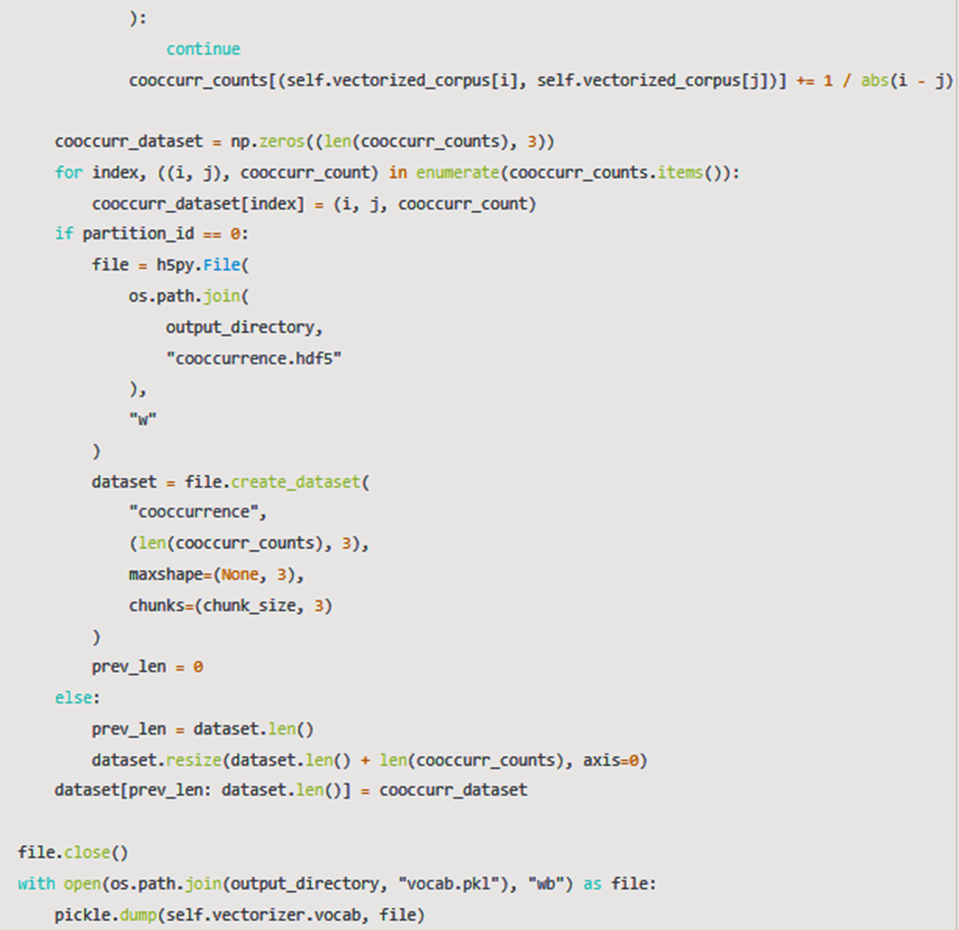


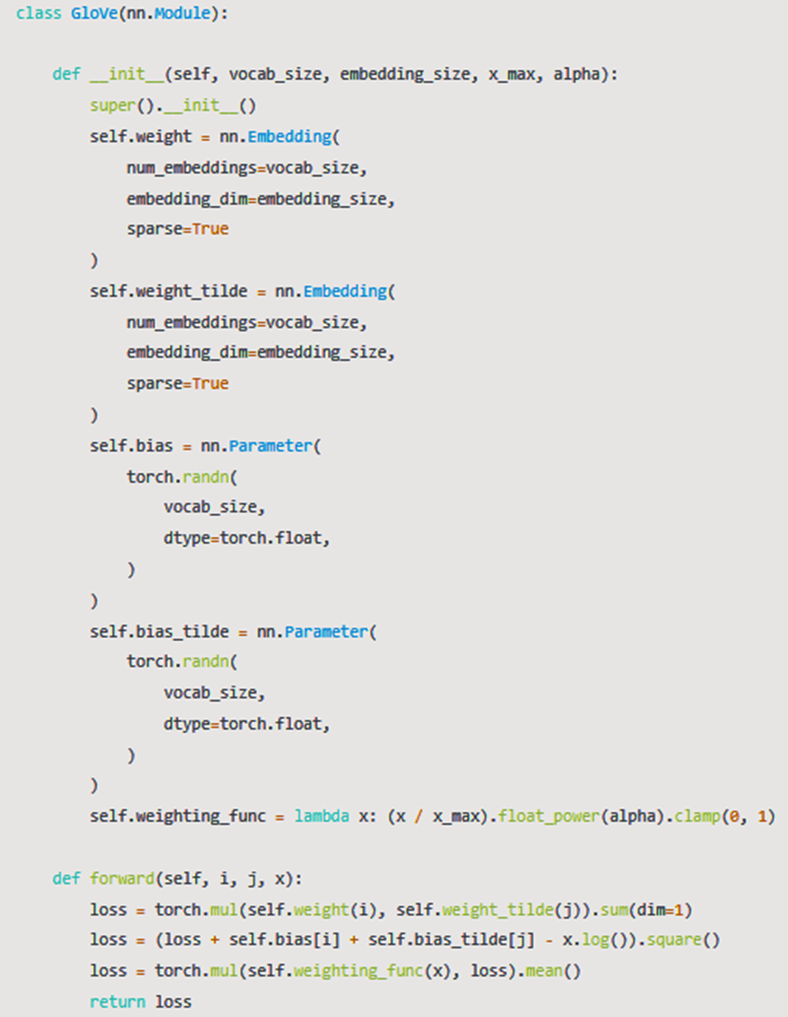




A screenshot of a computer code

Description automatically generated



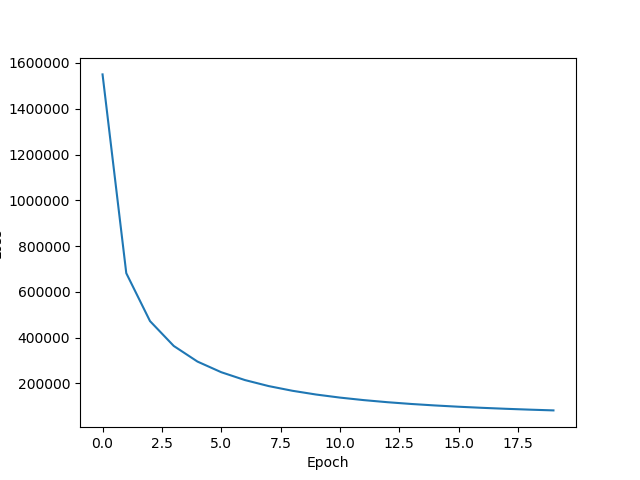


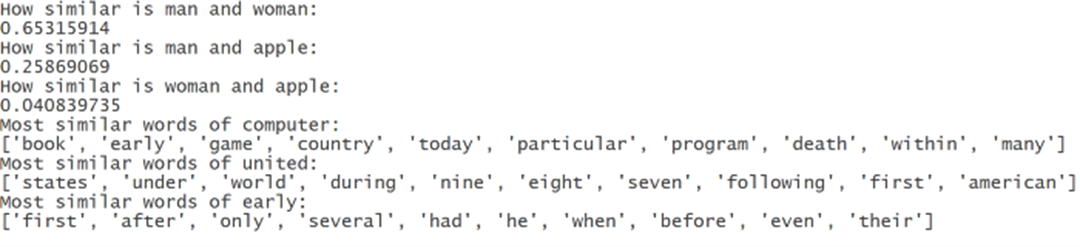
A screenshot of a computer program

Description automatically generated



**OUTPUT**





**RESULT**

The implementation was successful.

**Experiment 6**

**AIM**

To implement GLOVE using Alternative Least Squares.

**Software used**

Google Colaboratory

**Dataset/Corpus Used**

Kaggle Retail Rocket recommender system dataset – events.csv

**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**

|  |
| --- |
| !pip install implicit  #import libraries  import sys  import pandas as pd  import numpy as np  import scipy.sparse as sparse  from scipy.sparse.linalg import spsolve  import random  from sklearn.preprocessing import MinMaxScaler  import implicit  from datetime import datetime, timedelta  #Data Preprocessing  def create\_data(datapath,start\_date,end\_date):      df=pd.read\_csv(datapath)      df=df.assign(date=pd.Series(datetime.fromtimestamp(a/1000).date() for a in df.timestamp))      df=df.sort\_values(by='date').reset\_index(drop=True) # for some reasons RetailRocket did NOT sort data by date      df=df[(df.date>=datetime.strptime(start\_date,'%Y-%m-%d').date())&(df.date<=datetime.strptime(end\_date,'%Y-%m-%d').date())]      df=df[['visitorid','itemid','event']]      return df  #Download the kaggle RetailRocket data and give the events.csv file path  datapath= '/content/drive/MyDrive/dataset/events.csv'  data=create\_data(datapath,'2015-5-3','2015-5-18')  data['visitorid'] = data['visitorid'].astype("category")  data['itemid'] = data['itemid'].astype("category")  data['visitor\_id'] = data['visitorid'].cat.codes  data['item\_id'] = data['itemid'].cat.codes  data['event']=data['event'].astype('category')  data['event']=data['event'].cat.codes  sparse\_item\_user = sparse.csr\_matrix((data['event'].astype(float), (data['item\_id'], data['visitor\_id'])))  sparse\_user\_item = sparse.csr\_matrix((data['event'].astype(float), (data['visitor\_id'], data['item\_id'])))  #Building the model  model = implicit.als.AlternatingLeastSquares(factors=20, regularization=0.1, iterations=20)  alpha\_val = 40  data\_conf = (sparse\_item\_user \* alpha\_val).astype('double')  model.fit(data\_conf)  ###USING THE MODEL  #Get Recommendations  user\_id =   14  recommended = model.recommend(user\_id, sparse\_user\_item[user\_id])  recommended = pd.DataFrame(recommended)  recommended = recommended.T  recommended.columns = ['items', 'scores']  print(recommended)  #Get similar items  item\_id = 7  n\_similar = 4  similar = model.similar\_items(item\_id, n\_similar)  similar = pd.DataFrame(similar)  similar = similar.T  similar.columns = ['items', 'scores']  print(similar) |

**OUTPUT**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 7**

**AIM**

Visualizing data with analogies with t-SNE.

**Software used**

Jupyter Notebook

**Dataset/Corpus Used**

MNIST iris dataset

**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**

A screenshot of a computer program

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 8**

**AIM**

To visualize the data analogies using embedding projectors.

**Software used**

Google Colaboratory

**Dataset Used**

IMDB reviews dataset – subwords8k

**Theory**

The TensorBoard embedding projector is a very powerful tool in data analysis, specifically for interpreting and visualizing low-dimensional embeddings. In order to do so, first, it applies a dimensionality reduction algorithm to the input embeddings, 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**

A screenshot of a computer

Description automatically generated

A black and white image of a cloud of small dots

Description automatically generated with medium confidence

A black and white image of a heart

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 9**

**AIM**

Implement GloVe using tensorflow gradient descent.

**Software used**

Google Colaboratory

**Dataset/Corpus Used**

|  |
| --- |
| corpus = [      "the cat in the hat",      "the quick brown fox",      "the lazy dog",  ]  sample\_words = ["the", "cat", "in", "hat", "quick", "brown", "fox", "lazy", "dog"] |

**Theory**

The experiment involves implementing the Global Vectors for Word Representation (GloVe) algorithm using TensorFlow and gradient descent. GloVe is an unsupervised learning technique for generating word embeddings, representing words in a continuous vector space based on their co-occurrence statistics in a corpus. TensorFlow, an open-source machine learning library, is utilized to define and train the GloVe model. The model is constructed with embedding layers for target and context words, and gradient descent is employed to optimize the model parameters, minimizing the difference between predicted and actual co-occurrence probabilities. The resulting word embeddings capture semantic relationships between words, and their interpretation can provide insights into the underlying linguistic structure of the corpus. This experiment provides a practical understanding of the GloVe algorithm, TensorFlow implementation, and the role of gradient descent in training word embeddings for natural language processing tasks.

**Code**

|  |
| --- |
| import tensorflow as tf  import numpy as np  from keras.models import Model  from keras.layers import Input, Embedding, Dot, Reshape  # Sample corpus  corpus = [      "the cat in the hat",      "the quick brown fox",      "the lazy dog",      # Add more sentences as needed  ]  # Tokenize words  tokenizer = tf.keras.preprocessing.text.Tokenizer()  tokenizer.fit\_on\_texts(corpus)  total\_words = len(tokenizer.word\_index) + 1  # Generate word pairs for context and target words  def generate\_word\_pairs(corpus, window\_size=1):      word\_pairs = []      for sentence in corpus:          words = tokenizer.texts\_to\_sequences([sentence])[0]          for i, target\_word in enumerate(words):              for context\_word in words[max(0, i - window\_size) : i + window\_size]:                  if context\_word != target\_word:                      word\_pairs.append([target\_word, context\_word])      return np.array(word\_pairs)  # Build the GloVe model  embedding\_size = 50  # Choose an appropriate size for your embeddings  context\_size = 2  # Context window size  input\_target = Input(shape=(1,))  input\_context = Input(shape=(1,))  embedding = Embedding(total\_words, embedding\_size, input\_length=1)(input\_target)  context\_embedding = Embedding(total\_words, embedding\_size, input\_length=1)(input\_context)  dot\_product = Dot(axes=2)([embedding, context\_embedding])  dot\_product = Reshape((1,))(dot\_product)  # Define the GloVe model  glove\_model = Model(inputs=[input\_target, input\_context], outputs=dot\_product)  glove\_model.compile(optimizer="adam", loss="mean\_squared\_error")  # Generate training data  word\_pairs = generate\_word\_pairs(corpus, window\_size=context\_size)  target = np.array([pair[0] for pair in word\_pairs], dtype="int32")  context = np.array([pair[1] for pair in word\_pairs], dtype="int32")  labels = np.array([1.0] \* len(word\_pairs))  # Train the model  glove\_model.fit([target, context], labels, epochs=100, batch\_size=32)  # Extract word embeddings  word\_embeddings = glove\_model.get\_layer("embedding").get\_weights()[0]  # Now, word\_embeddings contains the trained GloVe embeddings  word\_embeddings  # Sample words for interpretation  sample\_words = ["the", "cat", "in", "hat", "quick", "brown", "fox", "lazy", "dog"]  # Create a dictionary to store word embeddings  word\_embedding\_dict = {}  for word in sample\_words:      word\_index = tokenizer.word\_index[word]      word\_embedding = word\_embeddings[word\_index]      word\_embedding\_dict[word] = word\_embedding  # Print the word embeddings  for word, embedding in word\_embedding\_dict.items():      print(f"{word}: {embedding}") |

**OUTPUT**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 10**

**AIM**

To perform point wise Mutual Information.

**Software used**

Google Colaboratory

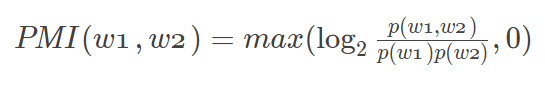
**Dataset/Corpus Used**

|  |
| --- |
| corpus = ['this is a foo bar bar black sheep foo bar bar black sheep foo bar bar black sheep shep bar bar black sentence'] |

**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 collections import Counter, OrderedDict  import numpy as np  import pandas as pd  def calculate\_pmi(word1\_count, word2\_count, cooccur\_count, total\_count):      p\_word1 = word1\_count / total\_count      p\_word2 = word2\_count / total\_count      p\_cooccur = cooccur\_count / total\_count      # Check for non-positive values to avoid math domain error      if p\_word1 \* p\_word2 == 0 or p\_cooccur == 0:          return 0  # Cap negative PMI values to 0      else:          pmi = max(0, np.log2(p\_cooccur / (p\_word1 \* p\_word2)))  # Use numpy for log2          return pmi  # Given corpus  corpus = ['this is a foo bar bar black sheep foo bar bar black sheep foo bar bar black sheep shep bar bar black sentence']  # Tokenize the corpus  tokenized\_corpus = corpus[0].split()  print(f"Tokens : {len(tokenized\_corpus)}\n")  # Calculate word counts  word\_counts = Counter(tokenized\_corpus)  # Convert word\_counts to DataFrame  word\_counts\_df = pd.DataFrame.from\_dict(word\_counts, orient='index', columns=['Count'])  word\_counts\_df.index.name = 'Word'  # Print word counts DataFrame  print("Word Counts:")  print(word\_counts\_df.T)  print()  # Create a co-occurrence matrix with headers  unique\_words = list(OrderedDict.fromkeys(tokenized\_corpus))  # Preserve order of appearance  cooccur\_matrix = np.zeros((len(unique\_words), len(unique\_words)), dtype=int)  # Populate the co-occurrence matrix  for i in range(len(tokenized\_corpus) - 1):      row\_index = unique\_words.index(tokenized\_corpus[i])      col\_index = unique\_words.index(tokenized\_corpus[i + 1])      cooccur\_matrix[row\_index][col\_index] += 1  # Convert co-occurrence matrix to DataFrame  cooccur\_df = pd.DataFrame(cooccur\_matrix, index=unique\_words, columns=unique\_words)  # Print the co-occurrence matrix DataFrame  print("Co-occurrence Matrix:")  print(cooccur\_df)  print()  # Calculate PMI for all possible word pairs  word\_counts = Counter(tokenized\_corpus)  cooccur\_counts = Counter(zip(tokenized\_corpus, tokenized\_corpus[1:]))  word\_pairs = [(word1, word2) for word1 in word\_counts.keys() for word2 in word\_counts.keys() if word1 != word2]  for word1, word2 in word\_pairs:      cooccur\_pair = (word1, word2)      pmi\_value = calculate\_pmi(word\_counts[word1], word\_counts[word2], cooccur\_counts[cooccur\_pair], len(tokenized\_corpus))      if(pmi\_value != 0):        print(f'PMI between {word1} and {word2}: {pmi\_value}') |

**OUTPUT**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**RESULT**

The implementation was successful.

**Experiment 11**

**AIM**

Implement Recursive Neural Tensor Network using tensorflow.

**Software used**

Google Colaboratory

**Theory**

The Recursive Neural Tensor Network (RNTN) is a neural network architecture designed for structured data, particularly hierarchical structures like parse trees or sequences. It extends recursive neural networks by incorporating a tensor layer, allowing the model to capture intricate interactions between pairs of words. RNTN starts with word vectors, computes tensor-based representations to capture higher-order relationships, and recursively combines these representations to form higher-level structures. The resulting vectors are processed through fully connected layers, introducing non-linearities. In this experiment, the RNTN is implemented using TensorFlow, with key components defined through the Keras API. The objective is to provide hands-on experience in leveraging tensor-based operations to enhance neural network architectures for tasks involving hierarchical relationships in structured data.

**Code**

|  |
| --- |
| import pandas as pd  import matplotlib.pyplot as plt  import warnings  warnings.filterwarnings("ignore")  import tensorflow as tf  import pandas as pd  import matplotlib.pyplot as plt  from keras.models import Model  from keras.layers import Input, Dense, Concatenate, Flatten, Dropout  from keras.initializers import RandomNormal  # Define Recursive Neural Tensor Network (RNTN) model with increased complexity  def build\_rntn(vocab\_size, embedding\_size):      input\_layer = Input(shape=(vocab\_size, embedding\_size))      # Word vectors      word\_vectors = Flatten()(input\_layer)      # Tensor layer      tensor\_layer = tf.linalg.matmul(tf.expand\_dims(word\_vectors, axis=-1), tf.expand\_dims(word\_vectors, axis=1))      tensor\_layer = Flatten()(tensor\_layer)      # Concatenate word vectors and tensor layer      combined\_layer = Concatenate()([word\_vectors, tensor\_layer])      # Fully connected layers with increased complexity      fc1 = Dense(50, activation='relu', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.1))(combined\_layer)      fc1 = Dropout(0.5)(fc1)  # Adding dropout for regularization      fc2 = Dense(25, activation='relu', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.1))(fc1)      fc2 = Dropout(0.5)(fc2)      output\_layer = Dense(1, activation='sigmoid', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.1))(fc2)      model = Model(inputs=input\_layer, outputs=output\_layer)      return model  # Example usage with increased batch size for dummy data  vocab\_size = 100  # Increased vocabulary size  embedding\_size = 20  # Increased embedding size  rntn\_model = build\_rntn(vocab\_size, embedding\_size)  optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001, clipvalue=1.0)  # Adjusted learning rate  rntn\_model.compile(optimizer=optimizer, loss='binary\_crossentropy', metrics=['accuracy'])  # Generate dummy training data with increased batch size  train\_batch\_size = 64  train\_input\_data = tf.random.normal((train\_batch\_size, vocab\_size, embedding\_size))  train\_labels = tf.random.uniform((train\_batch\_size, 1), minval=0, maxval=2, dtype=tf.int32)  # Generate dummy test data with increased batch size  test\_batch\_size = 32  test\_input\_data = tf.random.normal((test\_batch\_size, vocab\_size, embedding\_size))  test\_labels = tf.random.uniform((test\_batch\_size, 1), minval=0, maxval=2, dtype=tf.int32)  # Train the model with increased epochs  history = rntn\_model.fit(train\_input\_data, train\_labels, epochs=20, batch\_size=train\_batch\_size, validation\_data=(test\_input\_data, test\_labels), verbose=1)  # Predictions on test data  predictions = rntn\_model.predict(test\_input\_data)  # Create a table of predicted values against test labels  table = pd.DataFrame({'Test Labels': test\_labels.numpy().flatten(), 'Predicted Values': predictions.flatten()})  print("Table of Predicted Values against Test Labels:")  print(table)  # Plot accuracy and loss curves  plt.figure(figsize=(12, 4))  # Plot Training Accuracy  plt.subplot(1, 2, 1)  plt.plot(history.history['accuracy'], label='Training Accuracy')  plt.plot(history.history['val\_accuracy'], label='Test Accuracy')  plt.title('Training and Test Accuracy')  plt.xlabel('Epoch')  plt.ylabel('Accuracy')  plt.legend()  # Plot Training Loss  plt.subplot(1, 2, 2)  plt.plot(history.history['loss'], label='Training Loss')  plt.plot(history.history['val\_loss'], label='Test Loss')  plt.title('Training and Test Loss')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.legend()  plt.tight\_layout()  plt.show() |

**OUTPUT**

A screenshot of a computer

Description automatically generated

A graph of a line and a line

Description automatically generated with medium confidence

**RESULT**

The implementation was successful.