B.TECH. (2020-24)
Artificial Intelligence

NATURAL LANGUAGE PROCESSING WITH DEEP LEARNING

[CSE468]

Submitted To
Prof. (Dr.) Archana Singh

Submitted By
NIKITA
A023119820034
7AI 1

DEPARTMENT OF ARTIFICIAL INTELLIGENCE
AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY UTTAR PRADESH
NOIDA (U.P)

INDEX

Exp. No.	Name of Experiment	Date of Allotment of experiment	Date of Evaluation	Remarks	Signature of Faculty
1	To perform classification with word vectors.	18/07/2023	01/08/2023		
2	To implement Neural Network Bigram Model.	01/08/2023	08/08/2023		
3	Implementation of word2vec using NumPy.	08/08/2023	22/08/2023		
4	Implementation of word2vec using tensorflow.	22/08/2023	29/08/2023		
5	To implement GLOVE using numpy gradient descent.	29/08/2023	05/09/2023		
6	To implement GLOVE using Alternative Least Squares.	05/09/2023	12/09/2023		
7	Visualizing data with analogies with t-SNE.	12/09/2023	19/09/2023		
8	Visualizing data with analogies with PCA.	19/09/2023	26/09/2023		
9	To visualize the data analogies using embedding projectors.	26/09/2023	03/10/2023		
10	To perform point wise Mutual Information.	03/10/2023	10/10/2023		

Experiment 1

AIM
To perform classification with word vectors.

SOFTWARE USED
Jupyter Notebook

DATASET USED
BBC-text.csv

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 AND OUTPUT

Text Classification along with training Word Embeddings

In [1]:
```
import csv
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

In [2]:
```
vocab_size = 1000
embedding_dim = 16
max_length = 120
trunc_type='post'
padding_type='post'
oov_tok = ""
training_portion = .8
```

In [3]:
```
sentences = []
labels = []
stopwords = [ "a", "about", "above", "after", "again", "against", "all", "am", "an",
              "and", "any", "are", "as", "at", "be", "because", "been", "before",
              "being", "below", "between", "both", "but", "by", "could", "did", "do",
              "does", "doing", "down", "during", "each", "few", "for", "from",
              "further", "had", "has", "have", "having", "he", "he'd", "he'll",
              "he's", "her", "here", "here's", "hers", "herself", "him", "himself",
              "his", "how", "how's", "i", "i'd", "i'll", "i'm", "i've", "if", "in",
              "into", "is", "it", "it's", "its", "itself", "let's", "me", "more",
              "most", "my", "myself", "nor", "of", "on", "once", "only", "or",
              "other", "ought", "our", "ours", "ourselves", "out", "over", "own",
              "same", "she", "she'd", "she'll", "she's", "should", "so", "some",
              "such", "than", "that", "that's", "the", "their", "theirs", "them",
              "themselves", "then", "there", "there's", "these", "they", "they'd",
              "they'll", "they're", "they've", "this", "those", "through", "to",
              "too", "under", "until", "up", "very", "was", "we", "we'd", "we'll",
              "we're", "we've", "were", "what", "what's", "when", "when's", "where",
              "where's", "which", "while", "who", "who's", "whom", "why", "why's",
              "with", "would", "you", "you'd", "you'll", "you're", "you've", "your",
              "yours", "yourself", "yourselves" ]
print(len(stopwords))
```

```
#parsing text file
with open("./bbc-text.csv", 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    next(reader)
    for row in reader:
        labels.append(row[0])
        sentence = row[1]
        for word in stopwords:
            token = " " + word + " "
            sentence = sentence.replace(token, " ")
        sentences.append(sentence)
print(len(labels))
print(len(sentences))
print(sentences[0])
```

```
2225
2225
tv future hands viewers home theatre systems  plasma high-definition tvs  digital video recorders moving living room  way people watch tv will radicall
y different five years  time.  according expert panel gathered annual consumer electronics show las vegas discuss new technologies will impact one favo
urite pastimes. us leading trend  programmes content will delivered viewers via home networks  cable  satellite  telecoms companies  broadband service
providers front rooms portable devices.  one talked-about technologies ces digital personal video recorders (dvr pvr). set-top boxes  like us s tivo uk
s sky+ system  allow people record  store  play  pause forward wind tv programmes want.  essentially  technology allows much personalised tv. also buil
t-in high-definition tv sets  big business japan us  slower take off europe lack high-definition programming. not can people forward wind adverts  can
also forget abiding network channel schedules  putting together a-la-carte entertainment. us networks cable satellite companies worried means terms adv
ertising revenues well  brand identity  viewer loyalty channels. although us leads technology moment  also concern raised europe  particularly growing
uptake services like sky+.  happens today  will see nine months years  time uk  adam hume  bbc broadcast s futurologist told bbc news website. likes b
bc  no issues lost advertising revenue yet. pressing issue moment commercial uk broadcasters  brand loyalty important everyone.  will talking content b
rands rather network brands  said tim hanlon  brand communications firm starcom mediavest.  reality broadband connections  anybody can producer conten
t.  added:  challenge now hard promote programme much choice.  means  said stacey jolna  senior vice president tv guide tv group  way people find cont
ent want watch simplified tv viewers. means networks  us terms  channels take leaf google s book search engine future  instead scheduler help people fi
nd want watch. kind channel model might work younger ipod generation used taking control gadgets play them. might not suit everyone  panel recognised.
older generations comfortable familiar schedules channel brands know getting. perhaps not want much choice put hands  mr hanlon suggested.  end  kids j
ust diapers pushing buttons already - everything possible available  said mr hanlon.  ultimately  consumer will tell market want.  50 000 new gadgets
technologies showcased ces  many enhancing tv-watching experience. high-definition tv sets everywhere many new models lcd (liquid crystal display) tvs
launched dvr capability built  instead external boxes. one example launched show humax s 26-inch lcd tv 80-hour tivo dvr dvd recorder. one us s biggest
satellite tv companies  directtv  even launched branded dvr show 100-hours recording capability  instant replay  search function. set can pause rewind
tv 90 hours. microsoft chief bill gates announced pre-show keynote speech partnership tivo  called tivotogo  means people can play recorded programmes
windows pcs mobile devices. reflect increasing trend freeing multimedia people can watch want  want.
```

In [5]:
```
#splitting dataset
train_size = int(len(sentences) * training_portion)
train_sentences = sentences[:train_size]
train_labels = labels[:train_size]
validation_sentences = sentences[train_size:]
validation_labels = labels[train_size:]
print(len(sentences))
print(len(train_sentences))
print(len(validation_sentences))
```

```
2225
1780
445
```

In [6]:
```
#tokenizing sentences
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_sentences)
```

In [7]:
```
word_index = tokenizer.word_index
total_words = len(word_index)+1 #1 for oov word
print(total_words)
print(word_index)
```

```
27286
{'': 1, 's': 2, 'said': 3, 'will': 4, 'not': 5, 'mr': 6, 'year': 7, 'also': 8, 'people': 9, 'new': 10, 'us': 11, 'one': 12, 'can': 13, 'last': 14, 't':
15, 'first': 16, 'time': 17, 'two': 18, 'government': 19, 'world': 20, 'now': 21, 'uk': 22, 'best': 23, 'years': 24, 'no': 25, 'make': 26, 'just': 27,
'film': 28, 'told': 29, 'made': 30, 'get': 31, 'music': 32, 'game': 33, 'like': 34, 'back': 35, 'many': 36, '000': 37, 'labour': 38, 'three': 39, 'wel
l': 40, '1': 41, 'next': 42, 'bbc': 43, 'take': 44, 'set': 45, 'number': 46, 'added': 47, 'way': 48, 'market': 49, '2': 50, 'company': 51, 'may': 52,
'says': 53, 'election': 54, 'home': 55, 'off': 56, 'party': 57, 'good': 58, 'going': 59, 'much': 60, 'work': 61, '2004': 62, 'still': 63, 'win': 64, 's
how': 65, 'think': 66, 'games': 67, 'go': 68, 'top': 69, 'second': 70, 'won': 71, 'million': 72, '6': 73, 'england': 74, 'firm': 75, 'since': 76, 'wee
k': 77, 'say': 78, 'play': 79, 'part': 80, 'public': 81, 'use': 82, 'blair': 83, '3': 84, 'want': 85, 'minister': 86, 'however': 87, '10': 88, 'countr
y': 89, 'technology': 90, 'see': 91, '4': 92, 'five': 93, 'british': 94, 'news': 95, 'european': 96, 'high': 97, 'group': 98, 'tv': 99, 'used': 100, 'e
nd': 101, 'expected': 102, 'even': 103, 'players': 104, 'm': 105, 'brown': 106, '5': 107, 'six': 108, 'old': 109, 'net': 110, 'already': 111, 'four': 1
12, 'plans': 113, 'put': 114, 'come': 115, 'half': 116, 'london': 117, 'sales': 118, 'growth': 119, 'don': 120, 'long': 121, 'economy': 122, 'service':
123, 'right': 124, 'months': 125, 'chief': 126, 'day': 127, 'mobile': 128, 'former': 129, 'money': 130, 'britain': 131, 'director': 132, 'tax': 133, 's
ervices': 134, '2005': 135, 'deal': 136, 'need': 137, 'help': 138, 'digital': 139, 'according': 140, 'big': 141, 'industry': 142, 'place': 143, 'compan
ies': 144, 'users': 145, 'system': 146, 'business': 147, 'including': 148, 'team': 149, 'final': 150, 'based': 151, 'hit': 152, 'record': 153, 'repor
t': 154, 'third': 155, 'called': 156, 'really': 157, 'international': 158, 'month': 159, 'move': 160, 'wales': 161, 'europe': 162, 'another': 163, '7':
164, 'life': 165, 'around': 166, 'economic': 167, 'start': 168, 'great': 169, 'future': 170, '2003': 171, 'firms': 172, 'came': 173, 'france': 174, 'op
en': 175, 'got': 176, 'spokesman': 177, 'software': 178, 're': 179, 'without': 180, 'general': 181, 'club': 182, 'up': 183, 'took': 184, 'ireland': 18
```

In [10]:
```
validation_sequences = tokenizer.texts_to_sequences(validation_sentences)
validation_padded = pad_sequences(validation_sequences, padding=padding_type, maxlen=max_length)
```

In [11]:
```
print(len(validation_sentences[0]))
print(len(train_sequences[0]))
print(len(validation_padded[0]))
print()
print(validation_sentences[0])
print(validation_sequences[0])
print(validation_padded[0])
```

```
1100
449
120

hobbit picture  four years away  lord rings director peter jackson said will four years starts work film version hobbit.  oscar winner said visit sydne
y  desire  make  not lengthy negotiations.  think s gonna lot lawyers sitting room trying thrash deal will ever happen  said new zealander. rights jrr
tolkien s book split two major film studios. jackson  currently filming remake hollywood classic king kong  said thought sale mgm studios sony corporat
ion cast uncertainty project. 43-year-old australian city visit lord rings exhibition  attracted 140 000 visitors since opened december.  film-maker re
cently sued film company new line cinema undisclosed damages alleged withheld profits lost revenue first part middle earth trilogy. fellowship ring 200
1 went make worldwide profits $291 million (Â£152 million). jackson thought secured lucrative film directing deal history remake king kong  currently p
roduction wellington. picture  stars naomi watts oscar winner adrien brody  due released december. jackson also committed making film version lovely bo
nes  based best-selling book alice sebold.
[1, 1, 112, 24, 226, 276, 1, 132, 1, 1, 3, 4, 112, 24, 1, 61, 28, 493, 1, 674, 903, 3, 1, 1, 1, 26, 5, 1, 1, 66, 2, 1, 219, 1, 1, 1, 498, 1, 136, 4, 36
4, 1, 3, 10, 1, 317, 1, 1, 2, 528, 1, 18, 328, 28, 1, 1, 326, 1, 1, 891, 1, 1, 1, 3, 301, 353, 1, 1, 508, 1, 1, 1, 569, 1, 7, 109, 642, 558, 1, 276, 1,
1, 1, 1, 37, 1, 76, 1, 289, 28, 1, 505, 1, 28, 51, 10, 222, 1, 1, 1, 1, 1, 628, 268, 1, 16, 80, 1, 1, 1, 1, 1, 499, 311, 26, 1, 628, 1, 72, 1, 72, 1, 3
01, 1, 1, 28, 1, 136, 776, 1, 1, 1, 326, 455, 1, 1, 681, 1, 1, 674, 903, 1, 1, 271, 422, 289, 1, 8, 1, 202, 28, 493, 1, 1, 151, 23, 786, 528, 1, 1]
[364   1   3  10   1 317   1   1   2 528   1  18 328  28   1   1 326   1
   1 891   1   1   1   3 301 353   1   1 508   1   1   1 569   1   7 109
 642 558   1 276   1   1   1   1  37   1  76   1 289  28   1 505   1  28
  51  10 222   1   1   1   1   1 628 268   1  16  80   1   1   1   1   1
 499 311  26   1 628   1  72   1  72   1 301   1   1  28   1 136 776   1
   1   1 326 455   1   1 681   1   1 674 903   1   1 271 422 289   1   8
   1 202  28 493   1   1 151  23 786 528   1   1]
```

In [12]:
```
#tokenizing labels
label_tokenizer = Tokenizer()
label_tokenizer.fit_on_texts(labels)

training_label_seq = np.array(label_tokenizer.texts_to_sequences(train_labels))
validation_label_seq = np.array(label_tokenizer.texts_to_sequences(validation_labels))
```

In [13]:
```
print(training_label_seq)
print(training_label_seq.shape)
```

```
[[4]
 [2]
 [1]
 ...
 [1]
 [2]
 [5]]
(1780, 1)
```

```
#creating model
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(6, activation='softmax')
])
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

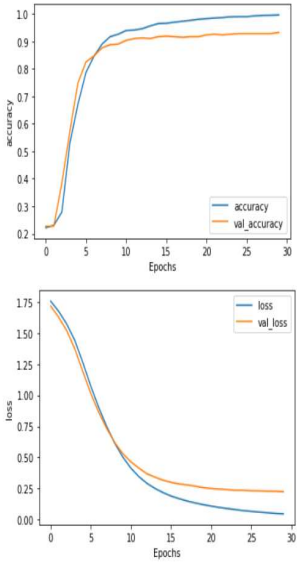
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 120, 16)           16000
_________________________________________________________________
global_average_pooling1d (Gl (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 24)                408
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 150
=================================================================
Total params: 16,558
Trainable params: 16,558
Non-trainable params: 0
_________________________________________________________________
```

In [15]:
```
num_epochs = 30
history = model.fit(train_padded, training_label_seq, epochs=num_epochs,
                    validation_data=(validation_padded, validation_label_seq),
                    verbose=2)
```

```
Train on 1780 samples, validate on 445 samples
Epoch 1/30
1780/1780 - 4s - loss: 1.7554 - accuracy: 0.2219 - val_loss: 1.7154 - val_accuracy: 0.2270
Epoch 2/30
1780/1780 - 0s - loss: 1.6732 - accuracy: 0.2303 - val_loss: 1.6259 - val_accuracy: 0.2270
Epoch 3/30
1780/1780 - 0s - loss: 1.5719 - accuracy: 0.2787 - val_loss: 1.5189 - val_accuracy: 0.3843
Epoch 4/30
1780/1780 - 0s - loss: 1.4412 - accuracy: 0.5298 - val_loss: 1.3730 - val_accuracy: 0.5730
Epoch 5/30
1780/1780 - 0s - loss: 1.2639 - accuracy: 0.6708 - val_loss: 1.1915 - val_accuracy: 0.7483
Epoch 6/30
1780/1780 - 0s - loss: 1.0728 - accuracy: 0.7865 - val_loss: 1.0146 - val_accuracy: 0.8247
Epoch 7/30
1780/1780 - 0s - loss: 0.8997 - accuracy: 0.8461 - val_loss: 0.8606 - val_accuracy: 0.8472
Epoch 8/30
1780/1780 - 0s - loss: 0.7467 - accuracy: 0.8899 - val_loss: 0.7301 - val_accuracy: 0.8764
Epoch 9/30
1780/1780 - 0s - loss: 0.6124 - accuracy: 0.9169 - val_loss: 0.6185 - val_accuracy: 0.8876
Epoch 10/30
1780/1780 - 0s - loss: 0.5023 - accuracy: 0.9258 - val_loss: 0.5298 - val_accuracy: 0.8899
Epoch 11/30
1780/1780 - 0s - loss: 0.4141 - accuracy: 0.9393 - val_loss: 0.4646 - val_accuracy: 0.9034
Epoch 12/30
1780/1780 - 0s - loss: 0.3454 - accuracy: 0.9410 - val_loss: 0.4139 - val_accuracy: 0.9101
Epoch 13/30
1780/1780 - 0s - loss: 0.2921 - accuracy: 0.9461 - val_loss: 0.3687 - val_accuracy: 0.9124
Epoch 14/30
1780/1780 - 0s - loss: 0.2513 - accuracy: 0.9562 - val_loss: 0.3417 - val_accuracy: 0.9101
Epoch 15/30
1780/1780 - 0s - loss: 0.2178 - accuracy: 0.9646 - val_loss: 0.3173 - val_accuracy: 0.9169
Epoch 16/30
1780/1780 - 0s - loss: 0.1901 - accuracy: 0.9657 - val_loss: 0.3004 - val_accuracy: 0.9191
Epoch 17/30
1780/1780 - 0s - loss: 0.1691 - accuracy: 0.9697 - val_loss: 0.2868 - val_accuracy: 0.9169
Epoch 18/30
1780/1780 - 0s - loss: 0.1502 - accuracy: 0.9730 - val_loss: 0.2794 - val_accuracy: 0.9146
Epoch 19/30
```

In [16]:
```
import matplotlib.pyplot as plt

def plot_graphs(history, string):
    plt.plot(history.history[string])
    plt.plot(history.history['val_'+string])
    plt.xlabel("Epochs")
    plt.ylabel(string)
    plt.legend([string, 'val_'+string])
    plt.show()
```

In [17]:
```
plot_graphs(history, "accuracy")
plot_graphs(history, "loss")
```



In [19]:
```
#visualizing embedded words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
def decode_sentence(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])
```

In [20]:
```
e = model.layers[0]
weights = e.get_weights()[0]
print(weights.shape)
```

```
(1000, 16)
```

In [21]:
```
import io
out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')
for word_num in range(1, vocab_size):
    word = reverse_word_index[word_num]
    embeddings = weights[word_num]
    out_m.write(word + "\n")
    out_v.write('\t'.join([str(x) for x in embeddings]) + "\n")
out_v.close()
out_m.close()

#use tensorflow embedding projector to visualize results by uploading this vector and metadata
```

Text Classification using Pre-trained Word Embeddings

In [23]:
```
embedding_dim = 100
max_length = 16
trunc_type='post'
padding_type='post'
oov_tok = ""
training_size=160000
test_portion=.1
```

In [25]:
```
corpus = []

num_sentences = 0
with open("./training_cleaned.csv",encoding="utf8") as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    for row in reader:
        list_item=[]
        list_item.append(row[5])
        this_label=row[0]
        if this_label=='0':
            list_item.append(0)
        else:
            list_item.append(1)
        num_sentences = num_sentences + 1
        corpus.append(list_item)
```

In [26]:
```
print(num_sentences)
print(len(corpus))
print(corpus[0][0])
print(corpus[0][1])
```

```
1600000
1600000
@switchfoot http://twitpic.com/2y1zl - Awww, that's a bummer.  You shoulda got David Carr of Third Day to do it. ;D
0
```

In [28]:
```
import random
sentences=[]
labels=[]
random.shuffle(corpus)
for x in range(training_size):
    sentences.append(corpus[x][0])
    labels.append(corpus[x][1])
```

In [29]:
```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(sentences)

word_index = tokenizer.word_index
vocab_size=len(word_index)+1

sequences = tokenizer.texts_to_sequences(sentences)
padded = pad_sequences(sequences, maxlen=max_length, padding=padding_type, truncating=trunc_type)

split = int(test_portion * training_size)

test_sequences = padded[0:split]
training_sequences = padded[split:training_size]
test_labels = labels[0:split]
training_labels = labels[split:training_size]

print(vocab_size)
print(word_index['i'])
```

```
138631
1
```

In [30]:
```
embeddings_index = {}
with open('./glove.6B.100d.txt',encoding="utf-8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

embeddings_matrix = np.zeros((vocab_size, embedding_dim));
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word);
    if embedding_vector is not None:
        embeddings_matrix[i] = embedding_vector;

print(embeddings_matrix.shape)
```

```
(138631, 100)
```

In [31]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length,
                              weights=[embeddings_matrix], trainable=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Conv1D(64, 5, activation='relu'),
    tf.keras.layers.MaxPooling1D(pool_size=4),
    tf.keras.layers.LSTM(64),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 16, 100)           13863100
_________________________________________________________________
dropout (Dropout)            (None, 16, 100)           0
_________________________________________________________________
conv1d (Conv1D)              (None, 12, 64)            32064
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 3, 64)             0
_________________________________________________________________
lstm (LSTM)                  (None, 64)                33024
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 13,928,253
Trainable params: 65,153
Non-trainable params: 13,863,100
_________________________________________________________________
```

In [32]:
```
num_epochs = 50
history = model.fit(training_sequences, np.array(training_labels), epochs=num_epochs,
                    validation_data=(test_sequences, np.array(test_labels)), verbose=2)
```

```
Train on 144000 samples, validate on 16000 samples
Epoch 1/50
144000/144000 - 178s - loss: 0.5680 - accuracy: 0.6984 - val_loss: 0.5284 - val_accuracy: 0.7337
Epoch 2/50
144000/144000 - 171s - loss: 0.5279 - accuracy: 0.7322 - val_loss: 0.5134 - val_accuracy: 0.7416
Epoch 3/50
144000/144000 - 169s - loss: 0.5093 - accuracy: 0.7451 - val_loss: 0.5120 - val_accuracy: 0.7431
Epoch 4/50
144000/144000 - 167s - loss: 0.4995 - accuracy: 0.7526 - val_loss: 0.5075 - val_accuracy: 0.7503
Epoch 5/50
144000/144000 - 170s - loss: 0.4910 - accuracy: 0.7583 - val_loss: 0.5069 - val_accuracy: 0.7504
Epoch 6/50
144000/144000 - 169s - loss: 0.4826 - accuracy: 0.7638 - val_loss: 0.5084 - val_accuracy: 0.7454
Epoch 7/50
144000/144000 - 169s - loss: 0.4772 - accuracy: 0.7671 - val_loss: 0.5014 - val_accuracy: 0.7541
Epoch 8/50
144000/144000 - 169s - loss: 0.4730 - accuracy: 0.7695 - val_loss: 0.5046 - val_accuracy: 0.7509
Epoch 9/50
144000/144000 - 169s - loss: 0.4667 - accuracy: 0.7737 - val_loss: 0.5044 - val_accuracy: 0.7487
Epoch 10/50
144000/144000 - 170s - loss: 0.4629 - accuracy: 0.7755 - val_loss: 0.5031 - val_accuracy: 0.7552
Epoch 11/50
144000/144000 - 169s - loss: 0.4614 - accuracy: 0.7753 - val_loss: 0.5022 - val_accuracy: 0.7549
Epoch 12/50
144000/144000 - 173s - loss: 0.4571 - accuracy: 0.7802 - val_loss: 0.5019 - val_accuracy: 0.7548
Epoch 13/50
144000/144000 - 170s - loss: 0.4550 - accuracy: 0.7809 - val_loss: 0.5084 - val_accuracy: 0.7537
Epoch 14/50
144000/144000 - 171s - loss: 0.4524 - accuracy: 0.7834 - val_loss: 0.5084 - val_accuracy: 0.7498
Epoch 15/50
144000/144000 - 170s - loss: 0.4513 - accuracy: 0.7815 - val_loss: 0.5099 - val_accuracy: 0.7531
Epoch 16/50
144000/144000 - 170s - loss: 0.4489 - accuracy: 0.7848 - val_loss: 0.5141 - val_accuracy: 0.7546
Epoch 17/50
144000/144000 - 170s - loss: 0.4457 - accuracy: 0.7873 - val_loss: 0.5141 - val_accuracy: 0.7550
Epoch 18/50
144000/144000 - 170s - loss: 0.4424 - accuracy: 0.7883 - val_loss: 0.5110 - val_accuracy: 0.7544
Epoch 19/50
```

In [33]:
```
#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc=history.history['accuracy']
val_acc=history.history['val_accuracy']
loss=history.history['loss']
val_loss=history.history['val_loss']

epochs=range(len(acc)) # Get number of epochs
```

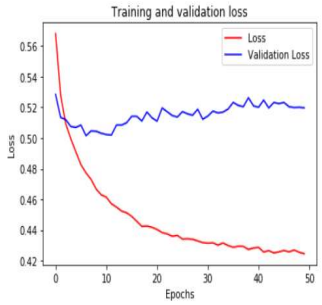
In [34]:
```
#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot(epochs, acc, 'r')
plt.plot(epochs, val_acc, 'b')
plt.title('Training and validation accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Accuracy", "Validation Accuracy"])

plt.figure()
```

Out[34]:

In [35]:

```
#-----------------------------------------------
# Plot training and validation loss per epoch
#-----------------------------------------------
plt.plot(epochs, loss, 'r')
plt.plot(epochs, val_loss, 'b')
plt.title('Training and validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Loss", "Validation Loss"])

plt.figure()
```

Out[35]:



In [ ]:

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:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1})$$

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

```
he said
he said kotc
he said kotc made
he said kotc made profits
he said kotc made profits of
he said kotc made profits of 265
he said kotc made profits of 265 ,
he said kotc made profits of 265 , 457
he said kotc made profits of 265 , 457 vs
he said kotc made profits of 265 , 457 vs loss
he said kotc made profits of 265 , 457 vs loss eight
he said kotc made profits of 265 , 457 vs loss eight cts
he said kotc made profits of 265 , 457 vs loss eight cts net
he said kotc made profits of 265 , 457 vs loss eight cts net loss
he said kotc made profits of 265 , 457 vs loss eight cts net loss 343
he said kotc made profits of 265 , 457 vs loss eight cts net loss 343 ,
he said kotc made profits of 265 , 457 vs loss eight cts net loss 343 , 266
he said kotc made profits of 265 , 457 vs loss eight cts net loss 343 , 266 ,
he said kotc made profits of 265 , 457 vs loss eight cts net loss 343 , 266 , 000
he said kotc made profits of 265 , 457 vs loss eight cts net loss 343 , 266 , 000 shares
```

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.

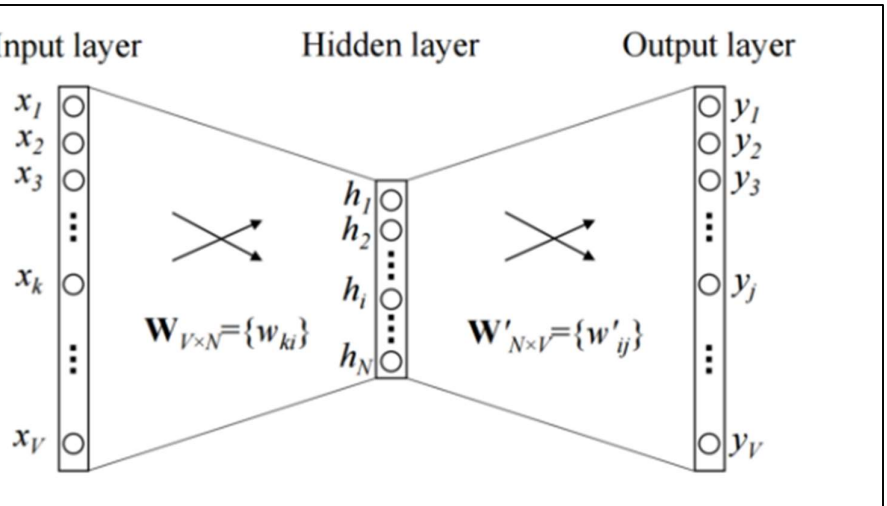


Figure 1: A simple CBOW model with only one word in the context

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)
```

```
array([ 0.62697258, -0.8435938 ,  2.10190231,  0.26548012,  0.2082425 ])
```

```
EPOCH: 584 LOSS: 20.631999407572565
EPOCH: 585 LOSS: 20.631520106423608
EPOCH: 586 LOSS: 20.631043111386806
EPOCH: 587 LOSS: 20.63056840670071
EPOCH: 588 LOSS: 20.630095976742535
EPOCH: 589 LOSS: 20.629625806026706
EPOCH: 590 LOSS: 20.629157879203383
EPOCH: 591 LOSS: 20.628692181056984
EPOCH: 592 LOSS: 20.628228696504777
EPOCH: 593 LOSS: 20.62776741059548
EPOCH: 594 LOSS: 20.627308308507835
EPOCH: 595 LOSS: 20.626851375549258
EPOCH: 596 LOSS: 20.626396597154457
EPOCH: 597 LOSS: 20.625943958884108
EPOCH: 598 LOSS: 20.625493446423523
EPOCH: 599 LOSS: 20.625045045581 32
```

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.

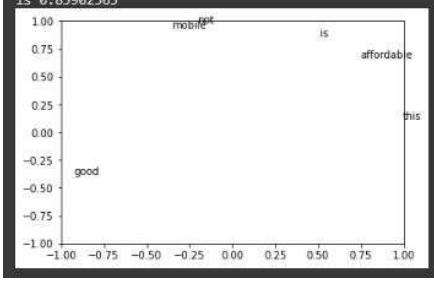
CODE

```
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

```
tf.Tensor(3.3884168, shape=(), dtype=float32)
tf.Tensor(1.4622096, shape=(), dtype=float32)
tf.Tensor(1.4024959, shape=(), dtype=float32)
tf.Tensor(1.376369, shape=(), dtype=float32)
tf.Tensor(1.3706317, shape=(), dtype=float32)
tf.Tensor(1.3691655, shape=(), dtype=float32)
tf.Tensor(1.3686327, shape=(), dtype=float32)
tf.Tensor(1.3684037, shape=(), dtype=float32)
tf.Tensor(1.3682947, shape=(), dtype=float32)
tf.Tensor(1.3682394, shape=(), dtype=float32)
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: FutureWarning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  FutureWarning,
mobile 0.9364252
affordable 0.6647415
good -0.38220012
not 0.9795167
this 0.11703096
is 0.85902363
```



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))
```

```
def load_config():
    config_filepath = "config.yaml's file path"
    with config_filepath.open() as f:
        config_dict = yaml.load(f, Loader=yaml.FullLoader)
    config = argparse.Namespace()
    for key, value in config_dict.items():
        setattr(config, key, value)
    return config
```

```
@dataclass
class Vocabulary:
    token2index: dict = field(default_factory=dict)
    index2token: dict = field(default_factory=dict)
    token_counts: list = field(default_factory=list)
    _unk_token: int = field(init=False, default=-1)

    def add(self, token):
        if token not in self.token2index:
            index = len(self)
            self.token2index[token] = index
            self.index2token[index] = token
            self.token_counts.append(0)
        self.token_counts[self.token2index[token]] += 1

    def get_topk_subset(self, k):
        tokens = sorted(
            list(self.token2index.keys()),
            key=lambda token: self.token_counts[self[token]],
            reverse=True
        )
        return type(self)(
            token2index={token: index for index, token in enumerate(tokens[:k])},
            index2token={index: token for index, token in enumerate(tokens[:k])},
            token_counts=[
                self.token_counts[self.token2index[token]] for token in tokens[:k]
            ]
        )
```

```
    def shuffle(self):
        new_index = [_ for _ in range(len(self))]
        random.shuffle(new_index)
        new_token_counts = [None] * len(self)
        for token, index in zip(list(self.token2index.keys()), new_index):
            new_token_counts[index] = self.token_counts[self[token]]
            self.token2index[token] = index
            self.index2token[index] = token
        self.token_counts = new_token_counts

    def get_index(self, token):
        return self[token]

    def get_token(self, index):
        if not index in self.index2token:
            raise Exception("Invalid index.")
        return self.index2token[index]

    @property
    def unk_token(self):
        return self._unk_token

    def __getitem__(self, token):
        if token not in self.token2index:
            return self._unk_token
        return self.token2index[token]

    def __len__(self):
        return len(self.token2index)
```

```
@dataclass
class Vectorizer:
    vocab: Vocabulary

    @classmethod
    def from_corpus(cls, corpus, vocab_size):
        vocab = Vocabulary()
        for token in corpus:
            vocab.add(token)
        vocab_subset = vocab.get_topk_subset(vocab_size)
        vocab_subset.shuffle()
        return cls(vocab_subset)

    def vectorize(self, corpus):
        return [self.vocab[token] for token in corpus]
```

```
@dataclass
class CooccurrenceEntries:
    vectorized_corpus: list
    vectorizer: Vectorizer

    @classmethod
    def setup(cls, corpus, vectorizer):
        return cls(
            vectorized_corpus=vectorizer.vectorize(corpus),
            vectorizer=vectorizer
        )
```

```
    def validate_index(self, index, lower, upper):
        is_unk = index == self.vectorizer.vocab.unk_token
        if lower < 0:
            return not is_unk
        return not is_unk and index >= lower and index <= upper

    def build(
        self,
        window_size,
        num_partitions,
        chunk_size,
        output_directory="."
    ):
        partition_step = len(self.vectorizer.vocab) // num_partitions
        split_points = [0]
        while split_points[-1] + partition_step <= len(self.vectorizer.vocab):
            split_points.append(split_points[-1] + partition_step)
        split_points[-1] = len(self.vectorizer.vocab)

        for partition_id in tqdm(range(len(split_points) - 1)):
            index_lower = split_points[partition_id]
            index_upper = split_points[partition_id + 1] - 1
            cooccurr_counts = Counter()
            for i in tqdm(range(len(self.vectorized_corpus))):
                if not self.validate_index(
                    self.vectorized_corpus[i],
                    index_lower,
                    index_upper
                ):
                    continue

                context_lower = max(i - window_size, 0)
                context_upper = min(i + window_size + 1, len(self.vectorized_corpus))
                for j in range(context_lower, context_upper):
                    if i == j or not self.validate_index(
                        self.vectorized_corpus[j],
                        -1,
                        -1
            ):
                        continue
                    cooccurr_counts[(self.vectorized_corpus[i], self.vectorized_corpus[j])] += 1 / abs(i - j)

    cooccurr_dataset = np.zeros((len(cooccurr_counts), 3))
    for index, ((i, j), cooccurr_count) in enumerate(cooccurr_counts.items()):
        cooccurr_dataset[index] = (i, j, cooccurr_count)
    if partition_id == 0:
        file = h5py.File(
            os.path.join(
                output_directory,
                "cooccurrence.hdf5"
            ),
            "w"
        )
        dataset = file.create_dataset(
            "cooccurrence",
            (len(cooccurr_counts), 3),
            maxshape=(None, 3),
            chunks=(chunk_size, 3)
        )
        prev_len = 0
    else:
        prev_len = dataset.len()
        dataset.resize(dataset.len() + len(cooccurr_counts), axis=0)
    dataset[prev_len: dataset.len()] = cooccurr_dataset
```

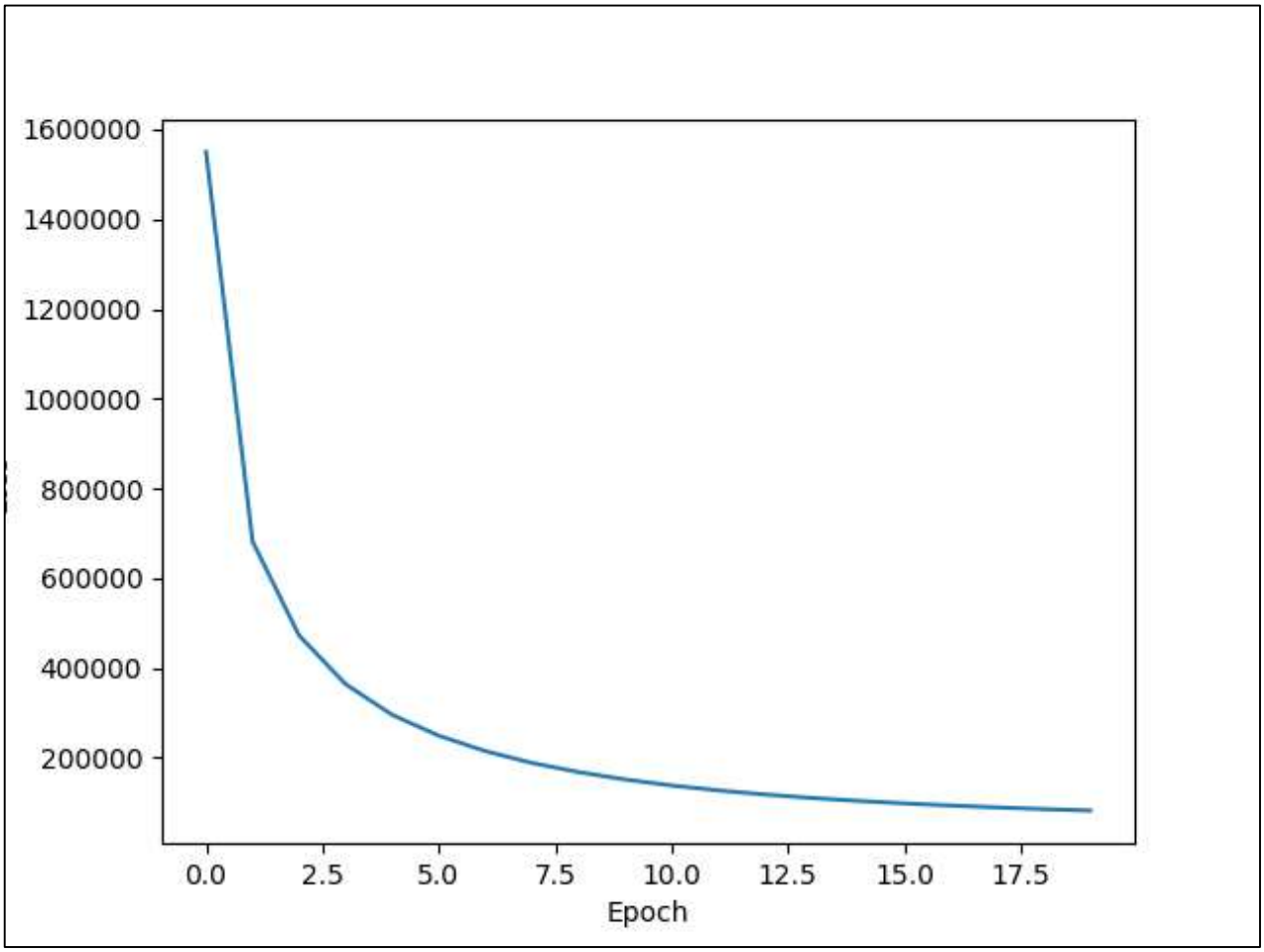
```
file.close()
with open(os.path.join(output_directory, "vocab.pkl"), "wb") as file:
    pickle.dump(self.vectorizer.vocab, file)
```

```
class GloVe(nn.Module):

    def __init__(self, vocab_size, embedding_size, x_max, alpha):
        super().__init__()
        self.weight = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_size,
            sparse=True
        )
        self.weight_tilde = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_size,
            sparse=True
        )
        self.bias = nn.Parameter(
            torch.randn(
                vocab_size,
                dtype=torch.float,
            )
        )
        self.bias_tilde = nn.Parameter(
            torch.randn(
                vocab_size,
                dtype=torch.float,
            )
        )
        self.weighting_func = lambda x: (x / x_max).float_power(alpha).clamp(0, 1)

    def forward(self, i, j, x):
        loss = torch.mul(self.weight(i), self.weight_tilde(j)).sum(dim=1)
        loss = (loss + self.bias[i] + self.bias_tilde[j] - x.log()).square()
        loss = torch.mul(self.weighting_func(x), loss).mean()
        return loss
```

```
dataloader = HDF5DataLoader(
    filepath=os.path.join(config.cooccurrence_dir, "cooccurrence.hdf5"),
    dataset_name="cooccurrence",
    batch_size=config.batch_size,
    device=config.device
)
model = GloVe(
    vocab_size=config.vocab_size,
    embedding_size=config.embedding_size,
    x_max=config.x_max,
    alpha=config.alpha
)
model.to(config.device)
optimizer = torch.optim.Adagrad(
    model.parameters(),
    lr=config.learning_rate
)
```

```
with dataloader.open():
    model.train()
    losses = []
    for epoch in tqdm(range(config.num_epochs)):
        epoch_loss = 0
        for batch in tqdm(dataloader.iter_batches()):
            loss = model(
                batch[0][:, 0],
                batch[0][:, 1],
                batch[1]
            )
            epoch_loss += loss.detach().item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        losses.append(epoch_loss)
        print(f"Epoch {epoch}: loss = {epoch_loss}")
        torch.save(model.state_dict(), config.output_filepath)
```

OUTPUT



```
How similar is man and woman:
0.65315914
How similar is man and apple:
0.25869069
How similar is woman and apple:
0.040839735
Most similar words of computer:
['book', 'early', 'game', 'country', 'today', 'particular', 'program', 'death', 'within', 'many']
Most similar words of united:
['states', 'under', 'world', 'during', 'nine', 'eight', 'seven', 'following', 'first', 'american']
Most similar words of early:
['first', 'after', 'only', 'several', 'had', 'he', 'when', 'before', 'even', 'their']
```

RESULT

The implementation was successful.

Experiment 6

AIM
To implement GLOVE using Alternative Least Squares.

SOFTWARE USED
Jupyter Notebook

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

```
0s
1 #Get Recommendations
2 user_id =   14
3 recommended = model.recommend(user_id, sparse_user_item[user_id])
4
5 recommended = pd.DataFrame(recommended)
6 recommended = recommended.T
7 recommended.columns = ['items', 'scores']
8
9 print(recommended)
```

```
      items       scores
0   27681.0  1.220773e-12
1   42185.0  1.044047e-12
2  115528.0  9.106946e-13
3    7842.0  8.595597e-13
4   26717.0  7.493854e-13
5   19762.0  7.466673e-13
6   30619.0  7.413609e-13
7  103203.0  7.381930e-13
8   41991.0  7.261463e-13
9   44878.0  7.224949e-13
```

```
#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)
```

```
      items    scores
0       7.0  1.000000
1  164779.0  0.999999
2   80629.0  0.999999
3  130467.0  0.999992
```

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.

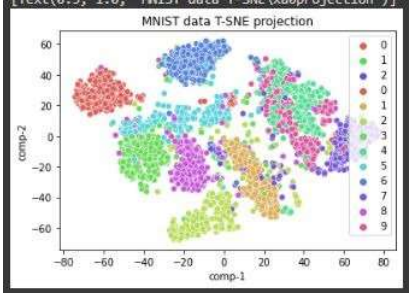
CODE

```
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

```
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: FutureWarning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  FutureWarning,
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 150 samples in 0.000s...
[t-SNE] Computed neighbors for 150 samples in 0.013s...
[t-SNE] Computed conditional probabilities for sample 150 / 150
[t-SNE] Mean sigma: 0.509910
[t-SNE] KL divergence after 250 iterations with early exaggeration: 50.387669
[t-SNE] KL divergence after 1000 iterations: 0.129141
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
(3000, 28, 28)
(3000, 784)
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 3000 samples in 0.001s...
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: FutureWarning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  FutureWarning,
[t-SNE] Computed neighbors for 3000 samples in 0.413s...
[t-SNE] Computed conditional probabilities for sample 1000 / 3000
[t-SNE] Computed conditional probabilities for sample 2000 / 3000
[t-SNE] Computed conditional probabilities for sample 3000 / 3000
[t-SNE] Mean sigma: 607.882413
[t-SNE] KL divergence after 250 iterations with early exaggeration: 78.365952
[t-SNE] KL divergence after 1000 iterations: 1.271625
[Text(0.5, 1.0, 'MNIST data T-SNE\xa0projection')]
```



RESULT

The implementation was successful.

Experiment 8

AIM
Visualizing data with analogies with PCA.

SOFTWARE USED
Jupyter Notebook

DATASET USED
Car price dataset

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.

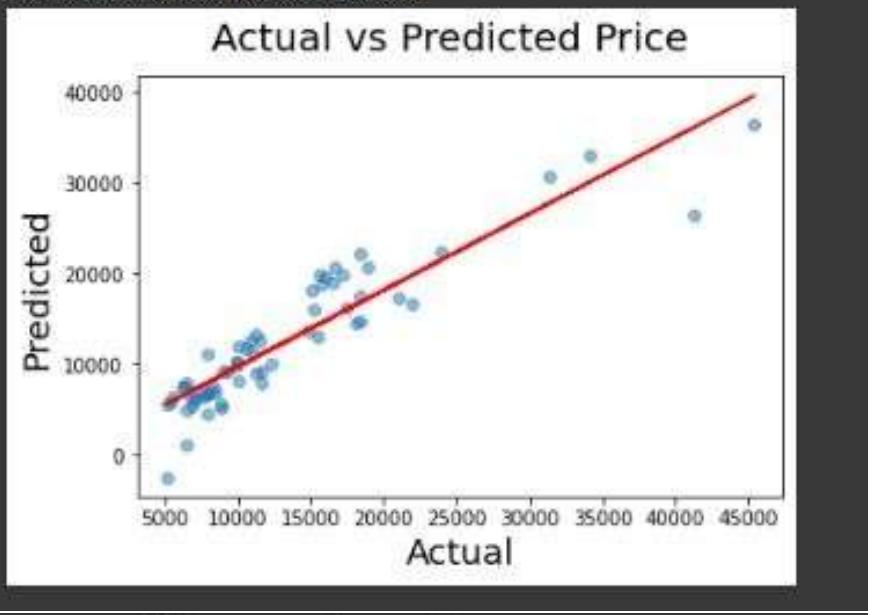
CODE

```
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

```
R2 Score: 0.8315252431797894
```




```
R2 Score (without using PCA): 0.7961110382079503
```

RESULT

The implementation was successful.

Experiment 9

AIM
To visualize the data analogies using embedding projectors.

SOFTWARE USED
Jupyter Notebook

DATASET USED
IMDB reviews dataset – subwords8k

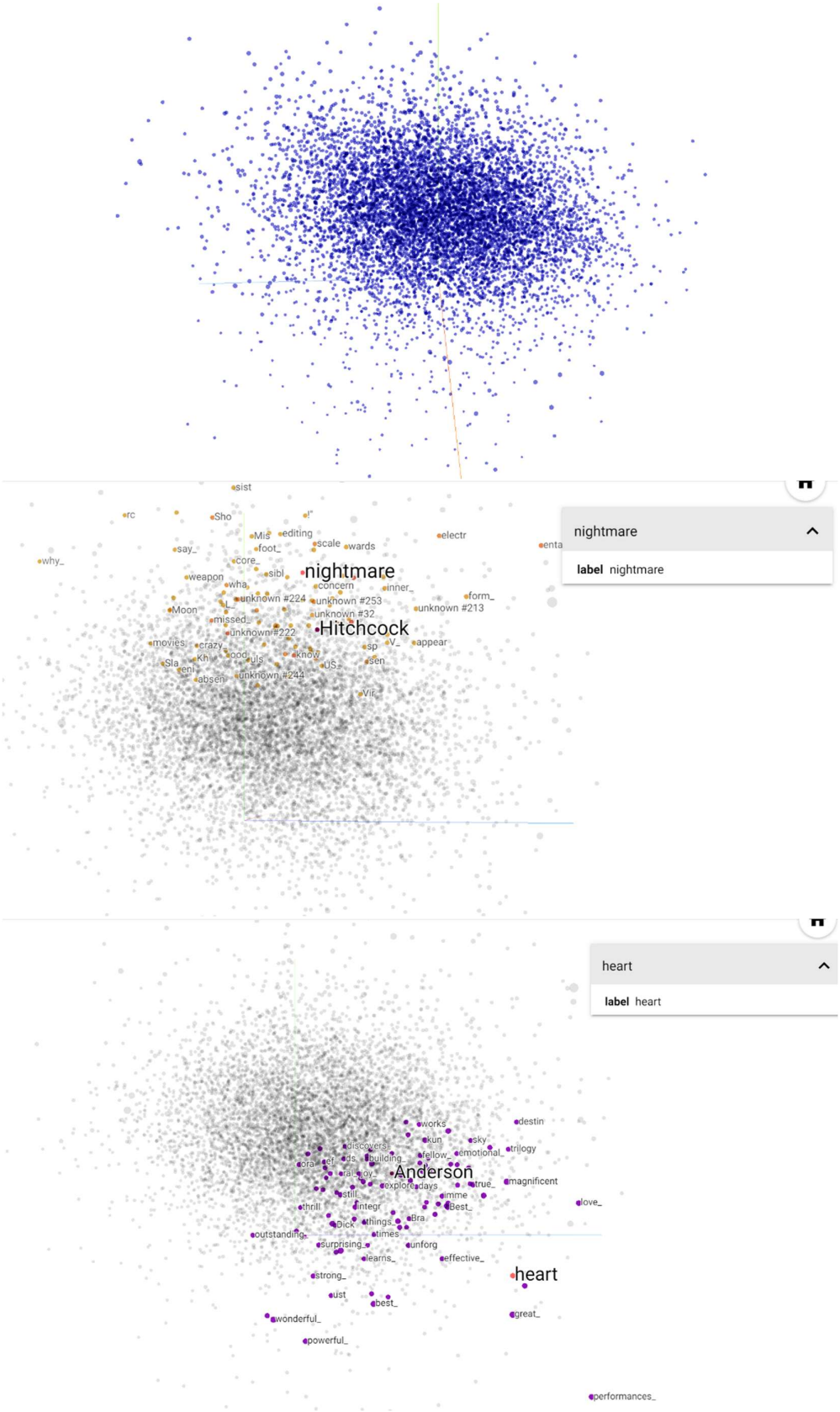
THEORY
The TensorBoard embedding projector is a very powerful tool in data analysis, specifically for interpreting and visualizing low-dimensional embeddings. In order to do so, first, it applies a dimensionality reduction algorithm to the input embeddings, between UMAP, T-SNE, PCA, or a custom one, 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



RESULT

The implementation was successful.

Experiment 10

AIM

To perform point wise Mutual Information.

SOFTWARE USED

Jupyter Notebook

DATASET/CORPUS USED

```
s1="The world knows it has lost a heroic champion of justice and freedom"
s2="The earth recognizes the loss of a valliant champoin of independence and justice"
```

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.

$$\operatorname{pmi}(x;y) \equiv \log_2 \frac{p(x,y)}{p(x)p(y)} = \log_2 \frac{p(x|y)}{p(x)} = \log_2 \frac{p(y|x)}{p(y)}$$

CODE and OUTPUT

```
s1="The world knows it has lost a heroic champion of justice and freedom"
s2="The earth recognizes the loss of a valliant champoin of independence and justice"

from Cleaner import clean
#Removing punctuations, stopwords and storing important words in a list
sent1=clean(s1)
sent2=clean(s2)
```

```
['The', 'world', 'know', 'lost', 'heroic', 'champion', 'justice', 'freedom']
['The', 'earth', 'recognizes', 'loss', 'valliant', 'champoin', 'independence', 'justice']
```

```
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)
```

```
Common Words:  ['The', 'justice']
Renewed List 1:  ['know', 'world', 'heroic', 'lost', 'champion', 'freedom']
Renewed List 2:  ['earth', 'recognizes', 'loss', 'valliant', 'champoin', 'independence']
```

```
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)
```

```
       loss   earth  recognizes     champoin      independence   valliant
know   0.04125       0.0    0.0165        0.020625       0.01375       0.020625
freedom        0.023571428571428571    0.018857142857142857    0.061285714285714284    0.011785714285714285    0.039285714285714285    0.0
world  0.033  0.026400000000000003    0.013200000000000002    0.0165        0.011000000000000001    0.0165
heroic 0.0275        0.055000000000000001    0.055  0.089375       0.022916666666666667    0.01375
lost   0.556875      0.033  0.04125       0.020625       0.0    0.051562500000000004
champion       0.020625       0.0165        0.041250000000000001    0.51046875     0.0171875      0.051562500000000004
```

```
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)
```

```
       valliant       recognizes     earth  independence    loss    champoin
champion       0       0.0    0.0    0.0    0.0    0
heroic 0       0.0    0.0    0.0    0.0    0
freedom        0       0.0    0.1765514388426884      0.3508157211181437      0.25294188175021576
lost   0       0.0    0.0    0.0    0.9972145802928659      0
world  0       0.0    0.0    0.0    0.0    0
know   0       0.0    0.0    0.0    0.13367263367077936     0
Sumattion of maximum terms:  1.3691062089524246
```

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

```
Sentence Similarity Measure:  0.4211382761190531
```

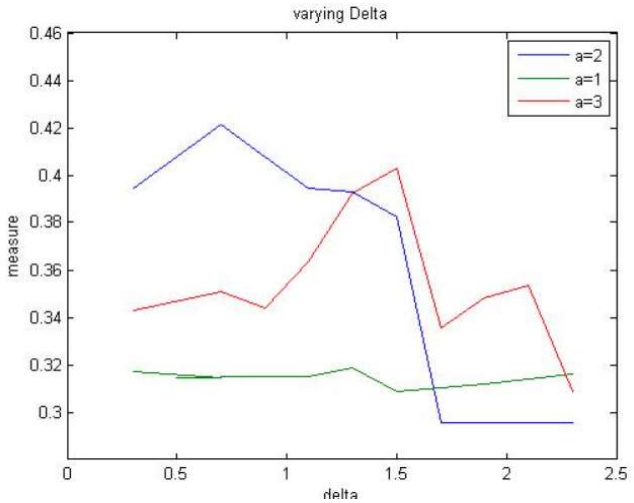
```
#Lemmatizing words and adding to the final array if they are not stopwordsfor w in words:
    if w not in stop_words:
           w=lemmatizer.lemmatize(w)
           filtered_text.append(w)
```

```
Corpus Words:  201206
Unique Words:  12907
```

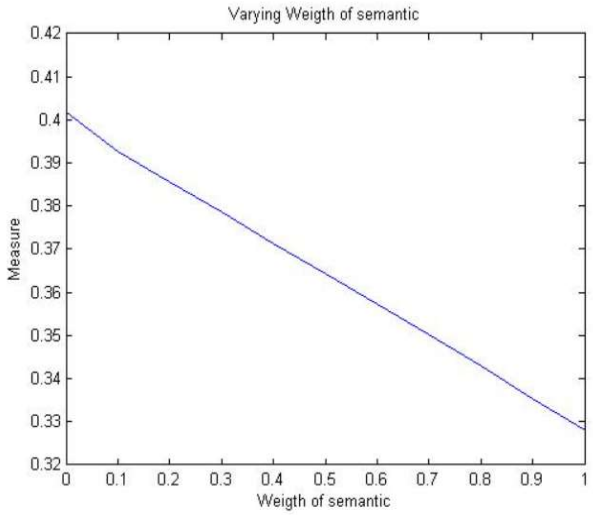
```
typefr=collections.Counter()
       for w in filtered_text:
              typefr[w]+=1
neighboursw1=collections.Counter()
       n2w1=[]
       for i in range(len(filtered_text)):
              if w1==filtered_text[i]:
                     neighboursw1[filtered_text[i]]+=1
                     for 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()
```

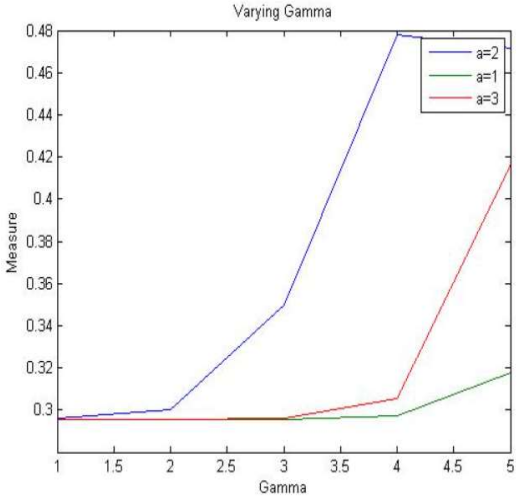
```
similarity,lmbda= normalized_similarity(similarity,lmbda)

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
```

Varying Delta



Varying Semantic Similarity Weight measure





RESULT

The implementation was successful.