Lab 6 - Performing/Showing Different Word Embeddings

- CountVectorizing (One-Hot Encoding)
- TF-IDF Encoding
- Word2Vec
- GloVe
- FastText
- ELMo
- Transformers

Import Basic Libraries

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Here we perform Word Embedding with CountVectorizing (and One-Hot Encoding).

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = ['Hello my name is Hamid and I will be using google colab.',
          'I am working on lab six module and then I will be working on lab seven module.',
          'We will look at different word embeddings.',
          'Hamid is enjoying the NLP course and google colab is so much fun.'
          ]

coun_vect = CountVectorizer()
count_matrix = coun_vect.fit_transform(corpus)

count_array = count_matrix.toarray()
vocab = coun_vect.get_feature_names_out()

# Use pandas to make a table with columns being the vocabs and rows with number of occurences of the words
pd.set_option('max_columns', None)
df = pd.DataFrame(data=count_array,columns = vocab)
print("Output table of vocabs and its recurrence: \n")
print(df)
print()
print("Output of numpy array: \n")
count_array
```

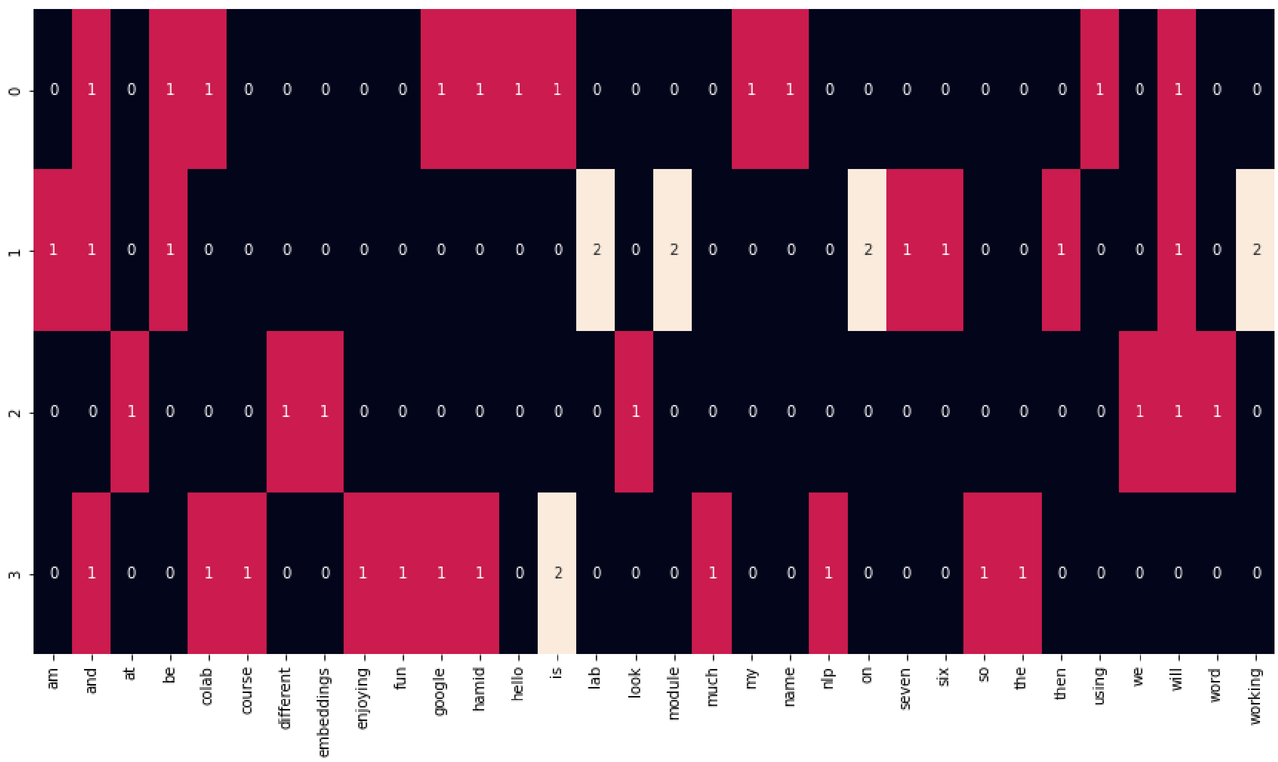
```
Output table of vocabs and its recurrence: 

   am  and  at  be  colab  course  different  embeddings  enjoying  fun  \
0   0    1   0   1      1       0          0           0         0    0   
1   1    1   0   1      0       0          0           0         0    0   
2   0    0   1   0      0       0          1           1         0    0   
3   0    1   0   0      1       1          0           0         1    1   

   google  hamid  hello  is  lab  look  module  much  my  name  nlp  on  \
0       1      1      1   1    0     0       0     0   1     1    0   0   
1       0      0      0   0    2     0       2     0   0     0    0   2   
2       0      0      0   0    0     1       0     0   0     0    0   0   
3       1      1      0   2    0     0       0     1   0     0    1   0   

   seven  six  so  the  then  using  we  will  word  working  
0      0    0   0    0     0      1   0     1     0        0  
```

```
1      1    1   0    0     1      0   0     1     0        2
2      0    0   0    0     0      0   1     1     1        0
3      0    0   1    1     0      0   0     0     0        0
```

Output of numpy array:

```
array([[0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0,
        0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
       [1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 2,
        1, 1, 0, 0, 1, 0, 0, 1, 0, 2],
       [0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 2, 0, 0, 0, 1, 0, 0, 1, 0,
        0, 0, 1, 1, 0, 0, 0, 0, 0, 0]])
```

```
fig, ax = plt.subplots(figsize=(15, 8))

sns.heatmap(count_array, annot=True, cbar = False, xticklabels = vocab);
```

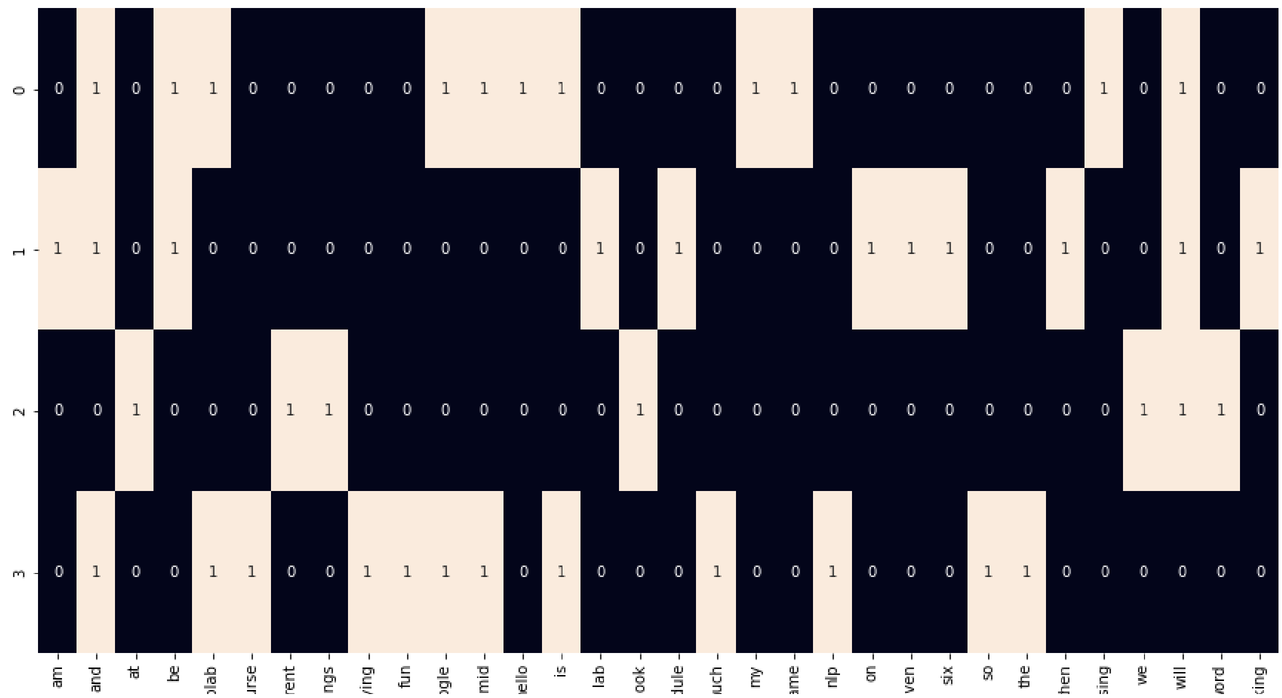



```
# One-hot encoding approach (place 1 no matter the reccurence)

one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()
one_hot
```

```
array([[0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0,
        0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
       [1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1,
        1, 1, 0, 0, 1, 0, 0, 1, 0, 1],
       [0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0,
        0, 0, 1, 1, 0, 0, 0, 0, 0, 0]])
```

```
fig, ax = plt.subplots(figsize=(15, 8))

sns.heatmap(one_hot, annot=True, cbar = False, xticklabels = vocab)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f96d8dc6fd0>
```



Here we perform Word Embedding with TF-IDF encoding.

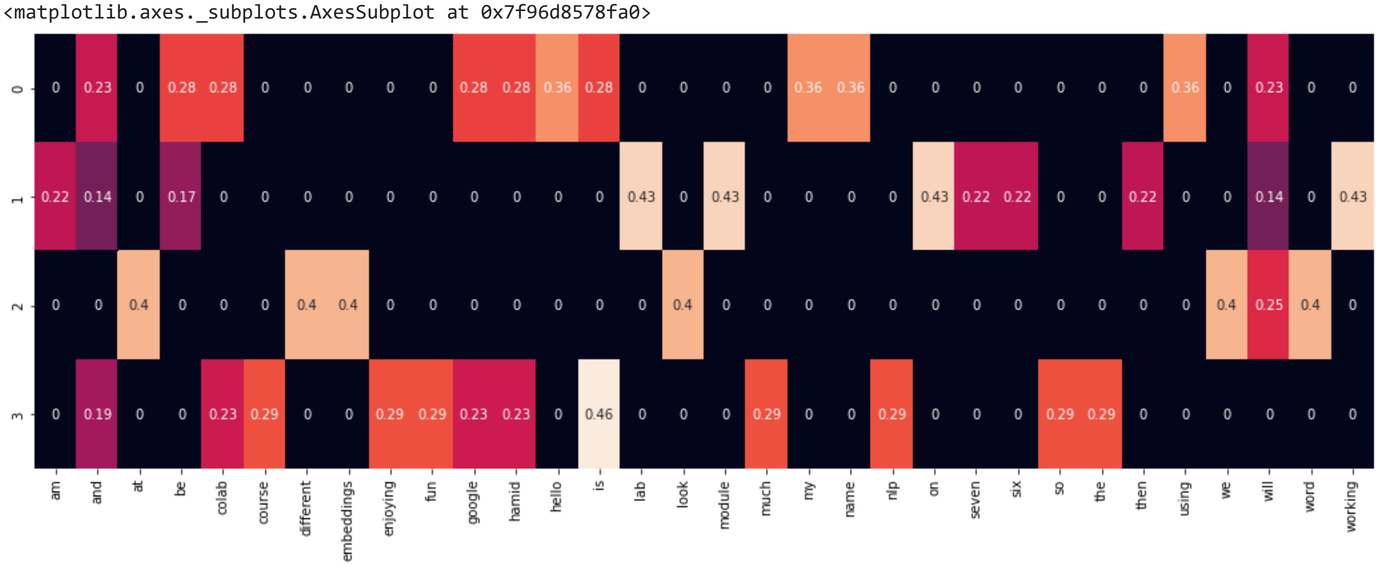
```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vec = TfidfVectorizer()
tf_idf = tfidf_vec.fit_transform(corpus).toarray()
vocab_2 = tfidf_vec.get_feature_names_out()

pd.set_option('max_columns', None)
df_2 = pd.DataFrame(data=tf_idf,columns = vocab_2)
print("Output table of vocabs and its weighted calculations: \n")
print(df_2)
print()
print("Output of numpy array: \n")
tf_idf
```

```
1  0.215985  0.137861  0.000000  0.170285  0.000000  0.000000   0.000000
2  0.000000  0.000000  0.395056  0.000000  0.000000  0.000000   0.395056
3  0.000000  0.186140  0.000000  0.000000  0.229920  0.291624   0.000000

   embeddings  enjoying       fun    google     hamid     hello        is  \
0    0.000000  0.000000  0.000000  0.280101  0.280101  0.355272  0.280101
1    0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
2    0.395056  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
3    0.000000  0.291624  0.291624  0.229920  0.229920  0.000000  0.459839

       lab      look   module      much        my      name       nlp  \
0  0.00000  0.000000  0.00000  0.000000  0.355272  0.355272  0.000000
1  0.43197  0.000000  0.43197  0.000000  0.000000  0.000000  0.000000
2  0.00000  0.395056  0.00000  0.000000  0.000000  0.000000  0.000000
3  0.00000  0.000000  0.00000  0.291624  0.000000  0.000000  0.291624

        on     seven       six        so       the      then     using  \
0  0.00000  0.000000  0.000000  0.000000  0.000000  0.000000  0.355272
1  0.43197  0.215985  0.215985  0.000000  0.000000  0.215985  0.000000
2  0.00000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
3  0.00000  0.000000  0.000000  0.291624  0.291624  0.000000  0.000000
```

```
3  0.000000  0.000000  0.000000  0.00000

Output of numpy array:

array([[0.        , 0.22676557, 0.        , 0.2801006 , 0.2801006 ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.2801006 , 0.2801006 , 0.35527209, 0.2801006 , 0.        ,
        0.        , 0.        , 0.        , 0.35527209, 0.35527209,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.35527209, 0.        , 0.22676557,
        0.        , 0.        ],
       [0.21598517, 0.13786054, 0.        , 0.17028519, 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.43197035,
        0.        , 0.43197035, 0.        , 0.        , 0.        ,
        0.        , 0.43197035, 0.21598517, 0.21598517, 0.        ,
        0.        , 0.21598517, 0.        , 0.        , 0.13786054,
        0.        , 0.43197035],
       [0.        , 0.        , 0.39505606, 0.        , 0.        ,
        0.        , 0.39505606, 0.39505606, 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.39505606, 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.39505606, 0.25215917,
        0.39505606, 0.        ],
       [0.        , 0.18613969, 0.        , 0.        , 0.22991954,
        0.29162379, 0.        , 0.        , 0.29162379, 0.29162379,
        0.22991954, 0.22991954, 0.        , 0.45983909, 0.        ,
        0.        , 0.        , 0.29162379, 0.        , 0.        ,
        0.29162379, 0.        , 0.        , 0.        , 0.29162379,
        0.29162379, 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        ]])
```

```
fig, ax = plt.subplots(figsize=(18, 6))
sns.heatmap(tf_idf, annot=True, cbar = False, xticklabels = vocab_2)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f96d8578fa0>
```



Here we perform Word Embedding with Word2Vec (CBOW and Skip Gram) on Alice In Wonderland Ebook.

```
# Read file and clean up the data text
df = pd.read_fwf('/content/Alice_Adventures_Book.txt', encoding='utf-8', names=["Text"])
```

```
df.shape
```

```
(2493, 1)
```

```
df.head()
```

	Text
0	Alice’s Adventures in Wonderland
1	by Lewis Carroll
2	THE MILLENNIUM FULCRUM EDITION 3.0
3	Contents
4	CHAPTER I. Down the Rabbit-Hole

```
df["Text"] = df["Text"].replace("\n", " ")
```

```
# Lower all text
df["Text"] = df["Text"].str.lower()
```

```
df.head()
```

	Text
0	alice’s adventures in wonderland
1	by lewis carroll
2	the millennium fulcrum edition 3.0
3	contents
4	chapter i. down the rabbit-hole

```
from nltk.tokenize import word_tokenize, sent_tokenize

import gensim, nltk
from gensim.models import Word2Vec
```

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

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
True
```

```
# Perform word tokenization on all text
df['Word Tokenized'] = df["Text"].apply(word_tokenize)
```

```
df.head()
```

	Text	Word Tokenized
0	alice’s adventures in wonderland	[alice, ’, s, adventures, in, wonderland]
1	by lewis carroll	[by, lewis, carroll]
2	the millennium fulcrum edition 3.0	[the, millennium, fulcrum, edition, 3.0]
3	contents	[contents]
4	chapter i. down the rabbit-hole	[chapter, i., down, the, rabbit-hole]

```
# Create CBOW model
model1 = gensim.models.Word2Vec(df["Word Tokenized"], min_count = 1, size = 125, window = 5)
```

```
# Print results
print("Cosine similarity between 'alice' " +
               "and 'wonderland' - CBOW : ",
    model1.wv.similarity('alice', 'wonderland'))

print("Cosine similarity between 'alice' " +
                 "and 'machines' - CBOW : ",
      model1.wv.similarity('alice', 'machines'))
```

```
Cosine similarity between 'alice' and 'wonderland' - CBOW :  0.99247503
Cosine similarity between 'alice' and 'machines' - CBOW :  0.8256512
```

```
# Create Skip Gram model
model2 = gensim.models.Word2Vec(df["Word Tokenized"], min_count = 1, size = 125, window = 5, sg = 1)
```

```
# Print results
print("Cosine similarity between 'alice' " +
          "and 'wonderland' - Skip Gram : ",
    model2.wv.similarity('alice', 'wonderland'))

print("Cosine similarity between 'alice' " +
            "and 'machines' - Skip Gram : ",
      model2.wv.similarity('alice', 'machines'))
```

```
Cosine similarity between 'alice' and 'wonderland' - Skip Gram :  0.94219637
Cosine similarity between 'alice' and 'machines' - Skip Gram :  0.92287004
```

Observations: We see close similarities from both models, another way to lower the errors, is to better clean the data, and perform extra preprocessing. Also playing around the size and window parameter can improve the results.

Here we perform Word Embedding with GloVe.

```
! python -m spacy download en_core_web_lg
```

```
# Glove
import spacy
# Load the spacy model that you have installed
import en_core_web_lg
nlp = en_core_web_lg.load()
```

```
# Few examples to illustrate Glove's model performance
doc1 = nlp("man king stands on the carpet and sees woman queen")
doc2 = nlp("man king sits on the throne and watches woman queen")
doc3 = nlp("bird stands on the tree and sees worm")
```

```
print("Similarity between doc1 and doc2: ")
print(doc1.similarity(doc2))
print()
print("Similarity between two doc1 and doc3: ")
print(doc1.similarity(doc3))
```

```
Similarity between doc1 and doc2:
0.9727433593222148

Similarity between two doc1 and doc3:
0.8084091351904159
```

```
print("Similarity between King and Queen: ")
print(doc1[1].similarity(doc1[9]))
print()
```

```
print("Similarity between King and Bird: ")
print(doc1[1].similarity(doc3[0]))
```

```
Similarity between King and Queen:
0.6108841896057129

Similarity between King and Bird:
0.1945231854915619
```

Observations:

Doc1 and Doc2 have high percentage of similarities, since those two sentences are very close to one another. However, Doc1 and Doc3 have lower percentage of similarities as it has a significant difference than Doc2.

Also, testing it between King and Bird vectors results in accurate percentage, as they are not similar.

Here we perform Word Embedding with FastText on Alice in Wonderland Ebook.

```
from gensim.models import FastText

ft_model = FastText(df["Word Tokenized"], min_count = 5, size = 200, window = 5, sg = 1, seed=42, iter=50)
```

```
# Save our model
ft_model.save("ft_model_alice")
```

```
# Load out model
ft_model_test = FastText.load('ft_model_alice')
```

```
ft_model_test.wv.get_vector('king').shape
```

```
(200,)
```

```
'king' in ft_model_test.wv.vocab
```

```
True
```

```
'burgerking' in ft_model_test.wv.vocab
```

```
False
```

```
ft_model_test.wv.similarity('alice', 'wonderland')
```

```
0.25769755
```

```
ft_model_test.wv.most_similar("king-warrior", topn=5)
```

```
[('king', 0.8777227401733398),
 ('executioner', 0.60203617811203),
 ('kind', 0.5669031143188477),
 ('walking', 0.5426331162452698),
 ('asking', 0.5243608951568604)]
```

Observations: Compared to Word2Vec, FastText does a better job of showing the correct percentage of similarities between alice and wonderland. The less percentage in this case is the better.

We also see the most similar text to "king-warrior", king being the highest or closest in similarity than the other words.

Here we perform Word Embedding with ElMo on Alice in Wonderland dataset (only the first chapter).

```
import tensorflow_hub as hub
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()
```

```
elmo = hub.Module("https://tfhub.dev/google/elmo/3", trainable=True)
```

```
sentences = df["Text"].tolist()
```

```
# From the book, we will only use chapter 1's content, instead of all chapters in order to reduce time.
elmo_input = sentences[16:196]
```

```
embeddings = elmo(
    elmo_input,
    signature="default",
    as_dict=True)["elmo"]
```

```
embeddings
```

```
<tf.Tensor 'module_apply_default/aggregation/mul_3:0' shape=(180, 17, 1024) dtype=float32>
```

```
%%time
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  sess.run(tf.tables_initializer())
  x = sess.run(embeddings)
```

```
CPU times: user 33.5 s, sys: 4.26 s, total: 37.7 s
Wall time: 26 s
```

```
x.shape
```

```
(180, 17, 1024)
```

```
embs = x.reshape(-1, 1024)
embs.shape
```

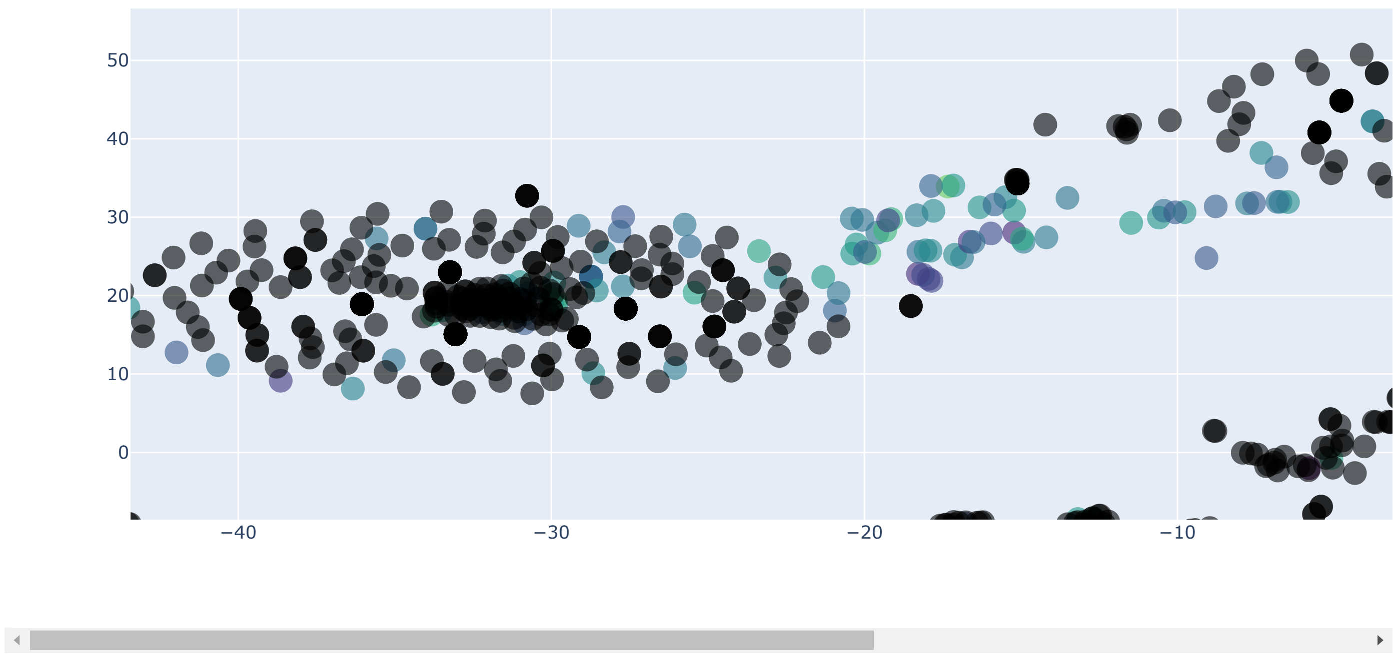
```
(3060, 1024)
```

```
from sklearn.decomposition import PCA

pca = PCA(n_components=100)
y = pca.fit_transform(embs)

from sklearn.manifold import TSNE

y = TSNE(n_components=2).fit_transform(y)
```

```
import plotly as py
import plotly.graph_objs as go
import numpy as np

data = [
    go.Scatter(
        x=[i[0] for i in y],
        y=[i[1] for i in y],
        mode='markers',
        text=[i for i in elmo_input],
```

```
    marker=dict(
        size=16,
        color = np.random.randn(500), #set color equal to a variable
        opacity= 0.6,
        colorscale='Viridis',
        showscale=False
    )
    )
]
layout = go.Layout()
layout = dict(
              yaxis = dict(zeroline = False),
              xaxis = dict(zeroline = False)
             )
fig = go.Figure(data=data, layout=layout)
```

```
fig.show()
```



Here we perform Word Embedding with Transformers.

```
!pip install transformers
```

```
import torch
torch.manual_seed(0)
from transformers import BertTokenizer, BertModel

import logging
import matplotlib.pyplot as plt
%matplotlib inline

# Load pre-trained model tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
```

```
Downloading (…)solve/main/vocab.txt: 100%                232k/232k [00:00<00:00, 1.64MB/s]
```

```
# Example sentence: Superman's movie plot

sentences = \
["superman father sends him off to earth as an infant to save him from the planet impending destruction.",
"the boy grows up in small town america, and eventually becomes a mild mannered reporter named clark kent.",
"once he arrives in metropolis, clark transforms himself into superman saving the city from crime.",
"however, when he unknowingly threatens the criminal genius, lex luthor plans to take over the world, trouble comes looking for

sentences
```

```
['superman father sends him off to earth as an infant to save him from the planet impending destruction.',
 'the boy grows up in small town america, and eventually becomes a mild mannered reporter named clark kent.',
 'once he arrives in metropolis, clark transforms himself into superman saving the city from crime.',
 'however, when he unknowingly threatens the criminal genius, lex luthor plans to take over the world, trouble comes
looking for our hero.']
```

```
# Print the original sentence.
print(' Original: ', sentences[0][:101])

#Print the sentence split into tokens.
print('Tokenized: ', tokenizer.tokenize(sentences[0])[:18])

#Print the sentence mapped to token ids.
print('Token IDs: ', tokenizer.convert_tokens_to_ids(tokenizer.tokenize(sentences[0]))[:18])
```

```
 Original:  superman father sends him off to earth as an infant to save him from the planet impending destruction
Tokenized:  ['superman', 'father', 'sends', 'him', 'off', 'to', 'earth', 'as', 'an', 'infant', 'to', 'save', 'him', 'from
Token IDs:  [10646, 2269, 10255, 2032, 2125, 2000, 3011, 2004, 2019, 10527, 2000, 3828, 2032, 2013, 1996, 4774, 17945, 62
```

```
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []
tokenized_texts = []


for sent in sentences:
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        truncation=True,
                        max_length = 48,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Save tokens from sentence as a separate array. We will use it later to explore and compare embeddings.
    marked_text = "[CLS] " + sent + " [SEP]"
    tokenized_texts.append(tokenizer.tokenize(marked_text))

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

# Convert the list into tensor.
input_ids = torch.cat(input_ids, dim=0)

# Print sentence 0, now as a list of IDs.
print('Original: ', sentences[0])
print('Token IDs:', input_ids[0])
```

```
Original:  superman father sends him off to earth as an infant to save him from the planet impending destruction.
Token IDs: tensor([  101, 10646,  2269, 10255,  2032,  2125,  2000,  3011,  2004,  2019,
        10527,  2000,  3828,  2032,  2013,  1996,  4774, 17945,  6215,  1012,
          102,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0])
```

```
/usr/local/lib/python3.8/dist-packages/transformers/tokenization_utils_base.py:2339: FutureWarning:

The `pad_to_max_length` argument is deprecated and will be removed in a future version, use `padding=True` or `padding='l
```

```
segments_ids = torch.ones_like(input_ids)
segments_ids.shape
```

```
torch.Size([4, 48])
```

```
model = BertModel.from_pretrained('bert-base-uncased',
          output_hidden_states = True, # Whether the model returns all hidden-states.
                                   )
model.eval();
```

Downloading (…)"pytorch_model.bin";: 100% 440M/440M [00:04<00:00, 93.8MB/s]

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['cls.predictions.tr
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with anoth
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly ide
```

```
with torch.no_grad():

    outputs = model(input_ids, segments_ids)

    hidden_states = outputs[2]
```

```
print ("Number of layers:", len(hidden_states), "  (initial embeddings + 12 BERT layers)")
print ("Number of batches:", len(hidden_states[0]))
print ("Number of tokens:", len(hidden_states[0][0]))
print ("Number of hidden units:", len(hidden_states[0][0][0]))
```

```
Number of layers: 13   (initial embeddings + 12 BERT layers)
Number of batches: 4
Number of tokens: 48
Number of hidden units: 768
```

```
# Concatenate the tensors for all layers. We use `stack` here to
# create a new dimension in the tensor.
token_embeddings = torch.stack(hidden_states, dim=0)

token_embeddings.size()
```

```
torch.Size([13, 4, 48, 768])
```

```
# Swap dimensions, so we get tensors in format: [sentence, tokens, hidden layes, features]
token_embeddings = token_embeddings.permute(1,2,0,3)

token_embeddings.size()
```

```
torch.Size([4, 48, 13, 768])
```

```
# we will use last four hidden layers to create each word embedding

processed_embeddings = token_embeddings[:, :, 9:, :]
processed_embeddings.shape
```

```
torch.Size([4, 48, 4, 768])
```

```
# Concatenate four layers for each token to create embeddings

embeddings = torch.reshape(processed_embeddings, (4, 48, -1))
embeddings.shape
```

```
torch.Size([4, 48, 3072])
```

```
for i, token_str in enumerate(tokenized_texts[0]):
  print (i, token_str)
```

```
0 [CLS]
1 superman
2 father
3 sends
4 him
5 off
6 to
7 earth
8 as
9 an
10 infant
11 to
12 save
13 him
14 from
15 the
16 planet
17 impending
18 destruction
19 .
20 [SEP]
```

```
from scipy.spatial.distance import cosine

superman_infant = cosine(embeddings[0][1], embeddings[0][10])
superman_earth = cosine(embeddings[0][1], embeddings[0][7])
father_infant = cosine(embeddings[0][2], embeddings[0][10])

print('Distance between superman and infant:  %.2f' % superman_infant)
print('Distance from superman to earth:  %.2f' % superman_earth)
print('Distance from father to infant:  %.2f' % father_infant)
```

```
Distance between superman and infant:  0.79
Distance from superman to earth:  0.57
Distance from father to infant:  0.66
```

Observations:

The distance outcome between the desired embeddings are pretty good. Here we see that superman is closer to infant, while superman is not that close to earth. Also there is a quite connection between father and infant.

References Used:

https://colab.research.google.com/drive/1N7HELWImK9xCYheyozVP3C_McbiRo1nb#scrollTo=1T01bAY75Mcv

https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/

https://www.gutenberg.org/files/11/11-0.txt
