In [3]:
```
import requests
from bs4 import BeautifulSoup
import pickle
```

In [4]:
```
from datetime import datetime as dt
Name = 'ATI Tesakulsiri'
ID = 'St123009'
print(f'This is the time I stamp and make the pdf -> {dt.now()}')
```

```
This is the time I stamp and make the pdf -> 2023-01-18 19:35:40.351714
```

- ## Copus consist of
  - how to webscraping from toward datascience
  - methamphetamine wikipedia
- with bs4 and requests library we can get text from internet.

In [4]:
```
load_D =False
if load_D:
    URL = 'https://towardsdatascience.com/web-scraping-basics-82f8b5acd45c'
    response = requests.get(URL)
    website_html = response.text
    soup = BeautifulSoup(website_html, "html.parser")
    all_paragraph = soup.find_all(name="p", class_="pw-post-body-paragraph")
    get_only_text = [para.getText() for para in all_paragraph]
    # print(len(get_only_text)) # 40 quite enough
    my_corpus = [text.lower() for text in get_only_text if len(text) > 40 ]
    with open('hijackdata.atikeep','wb') as tostore:
        pickle.dump(my_corpus,tostore)
```

In [5]:
```
UpdateSecCorpus = False
if UpdateSecCorpus:
    URL_meth = 'https://en.wikipedia.org/wiki/Methamphetamine'
    # URL = 'https://towardsdatascience.com/web-scraping-basics-82f8b5acd45c
    # response = requests.get(URL)
    # website_html = response.text
    soup = BeautifulSoup(website_html, "html.parser")
    # all_paragraph = soup.find_all(name="p", class_="pw-post-body-paragraph
    response_meth = requests.get(URL_meth)
    webmeth = response_meth.text
    ice = BeautifulSoup(webmeth,'html.parser')
    all_meth = [sen.getText() for sen in ice.findAll(name = 'p')]
    # print(all_meth)
    # get_only_text = [para.getText() for para in all_paragraph]
    # print(len(get_only_text)) # 40 quite enough
    my_corpus2 = [text.lower() for text in (all_meth) if len(text) > 40 ]
    with open('methjack2.atikeep','wb') as store2:
        pickle.dump(my_corpus2,store2)
    # with open('hijackdata.atikeep','wb') as tostore:
    #     pickle.dump(my_corpus,tostore)
```

Question 1 Try with new corpus

In [6]:
```
with open('hijackdata.atikeep','rb') as readed:
    my_corpus = pickle.load(readed)
with open('methjack2.atikeep','rb') as readed:
```

```
    my_corpus2 = pickle.load(readed)
print('all document = ',len(my_corpus),'doc, and each document have around',
print('all document = ',len(my_corpus2),'doc, and each document have around'
```

```
all document =  38 doc, and each document have around 113 characters
all document =  61 doc, and each document have around 225 characters
```

In [7]:

```
# for i in my_corpus:
#     print(len(i),'==',i)
```

In [8]:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
```

0. Init new corpus

In [9]:

```
import re
```

In [10]:

```
def remove_specials(wo_stops):
    clean_text = re.sub('[^A-Za-z]+', ' ', wo_stops)
    clean_text = " ".join([text for text in clean_text.split(' ') if len(tex
    return clean_text
    # src https://www.kaggle.com/code/wickkiey/spacy-text-preprocessing
```

In [11]:

```
corpus = [remove_specials(sen) for sen in my_corpus] +  [remove_specials(ser
```

In [12]:

```
#1. tokenize
#usually you use spaCy / NLTK to tokenize (but we gonna do this later on, we
corpus_tokenized = [sent.split(" ") for sent in corpus]
# corpus_tokenized  #we called each of this as "tokens", NOT words
```

In [13]:

```
#2. numericalize

#2.1 get all the unique words
#we want to flatten this (basically merge all list)
flatten = lambda l: [item for sublist in l for item in sublist]
vocabs  = list(set(flatten(corpus_tokenized)))  #vocabs is a term defining a
```

In [14]:

```
#2.2 assign id to all these vocabs
word2index = {v: idx for idx, v in enumerate(vocabs)}
```

In [15]:

```
#add <UNK>, which is a very normal token exists in the world
vocabs.append('<UNK>') #chaky, can it be ##UNK, or UNKKKKKK, or anything
word2index['<UNK>'] = len(vocabs) -1  #usually <UNK> is 0
```

In [16]:

```
#create index2word dictionary
#2 min
index2word = {v:k for k, v in word2index.items()}

# index2word
```

In [17]:

```
print(len(vocabs))
```

```
1810
```

2. Prepare train data

You move the window along, and create those tuples as we said in class

In [18]:
```
#move along the corpus
#to fit with our corpus, we gonna use window_size = 1

skipgrams = []

#for each corpus
for sent in corpus_tokenized:
    #for each sent ["apple", "banana", "fruit"]
    for i in range(1, len(sent) - 1): #start from 1 to second last
        center_word = sent[i]
        outside_words = [sent[i-1], sent[i+1]]  #window_size = 1
        for o in outside_words:
            skipgrams.append([center_word, o])

# skipgrams

#here we want to create (banana, apple), (banana, fruit) append to some list
```

In [19]:
```
#let's make what we have made into a function (batch function)
#return a batches of data, e.g., =2 --> ['banana', 'apple'], ['banana', 'fru
#also i want these batches to be id, NOT token   --> [5, 4]

def random_batch(batch_size, corpus):

    skipgrams = []

    #for each corpus
    for sent in corpus_tokenized:
        #for each sent ["apple", "banana", "fruit"]
        for i in range(1, len(sent) - 1): #start from 1 to second last
            center_word = word2index[sent[i]]
            outside_words = [word2index[sent[i-1]], word2index[sent[i+1]]]
            for o in outside_words:
                skipgrams.append([center_word, o])

    #only get a batch, not the entire list
    random_index = np.random.choice(range(len(skipgrams)), batch_size, repla

    #appending some list of inputs and labels
    random_inputs, random_labels = [], []
    for index in random_index:
        random_inputs.append([skipgrams[index][0]])  #center words, this wil
        random_labels.append([skipgrams[index][1]])

    return np.array(random_inputs), np.array(random_labels)
```

In [20]:
```
input, label = random_batch(10, corpus_tokenized)

print(f"{input.shape}")
print(f"{label=}")
```

```
(10, 1)
label=array([[ 115],
       [1424],
       [1667],
       [ 678],
       [1533],
       [   0],
       [ 460],
       [ 115],
       [ 560],
       [1200]])
```

3. Model

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m \leq j \leq m \\ j \neq 0}}\log P(w_{t+j} | w_t; \theta)$$

where $P(w_{t+j} | w_t; \theta) = $

$$P(o|c)=\frac{\exp(\mathbf{u_o^{\top}v_c})}{\sum_{w=1}^V\exp(\mathbf{u_w^{\top}v_c})}$$

where o is the outside words and c is the center word

In [21]:
```
voc_size = len(vocabs)
voc_size
```

Out[21]: 1810

In [22]:
```
#the model will accept three vectors - u_o, v_c, u_w
#u_o - vector for outside words
#v_c - vector for center word
#u_w - vectors of all vocabs

class Skipgram(nn.Module):

    def __init__(self, voc_size, emb_size):
        super(Skipgram, self).__init__()
        self.embedding_center_word  = nn.Embedding(voc_size, emb_size)  #is
        self.embedding_outside_word = nn.Embedding(voc_size, emb_size)

    def forward(self, center_word, outside_word, all_vocabs):
        #center_word, outside_word: (batch_size, 1)
        #all_vocabs: (batch_size, voc_size)

        #convert them into embedding
        center_word_embed  = self.embedding_center_word(center_word)     #(b
        outside_word_embed = self.embedding_outside_word(outside_word)   #(b
        all_vocabs_embed   = self.embedding_outside_word(all_vocabs)     #(b

        #bmm is basically @ or .dot , but across batches (i.e., ignore the b
        top_term = outside_word_embed.bmm(center_word_embed.transpose(1, 2))
        #(batch_size, 1, emb_size) @ (batch_size, emb_size, 1) = (batch_size

        top_term_exp = torch.exp(top_term)  #exp(uo vc)
        #(batch_size, 1)

        lower_term = all_vocabs_embed.bmm(center_word_embed.transpose(1, 2))
         #(batch_size, voc_size, emb_size) @ (batch_size, emb_size, 1) = (ba
```

```
        lower_term_sum = torch.sum(torch.exp(lower_term), 1) #sum exp(uw vc)
        #(batch_size, 1)

        loss_fn = -torch.mean(torch.log(top_term_exp / lower_term_sum))
        #(batch_size, 1) / (batch_size, 1) ==mean==> scalar

        return loss_fn
```

In [23]:
```
#preparing all_vocabs

batch_size = 2

def prepare_sequence(seq, word2index):
    #map(function, list of something)
    #map will look at each of element in this list, and apply this function
    idxs = list(map(lambda w: word2index[w] if word2index.get(w) is not None
    return torch.LongTensor(idxs)

all_vocabs = prepare_sequence(list(vocabs), word2index).expand(batch_size, v
all_vocabs.shape
```

Out[23]:
```
torch.Size([2, 1810])
```

testing the model

In [24]:
```
input, label = random_batch(batch_size, corpus_tokenized)
input #center word
```

Out[24]:
```
array([[1099],
       [1680]])
```

In [25]:
```
emb_size = 2 #usually, this can be 50, 100, or 300
model = Skipgram(voc_size, emb_size)
```

In [26]:
```
input_tensor = torch.LongTensor(input)
label_tensor = torch.LongTensor(label)  #LongTensor basically means integer.
loss = model(input_tensor, label_tensor, all_vocabs)
```

In [27]:
```
def get_embed(word):
    try:
        index = word2index[word]
    except:
        index = word2index['<UNK>']

    word = torch.LongTensor([index])

    center_embed  = model.embedding_center_word(word)
    outside_embed = model.embedding_outside_word(word)

    embed = (center_embed + outside_embed) / 2

    return  embed[0][0].item(), embed[0][1].item()
```

Training

In [28]:
```
batch_size = 2 #why?  no reason;
emb_size   = 2 #why?  no reason; usually 50, 100, 300, but 2 so we can plot
model      = Skipgram(voc_size, emb_size)
```

```
optimizer  = optim.Adam(model.parameters(), lr=0.001)
```

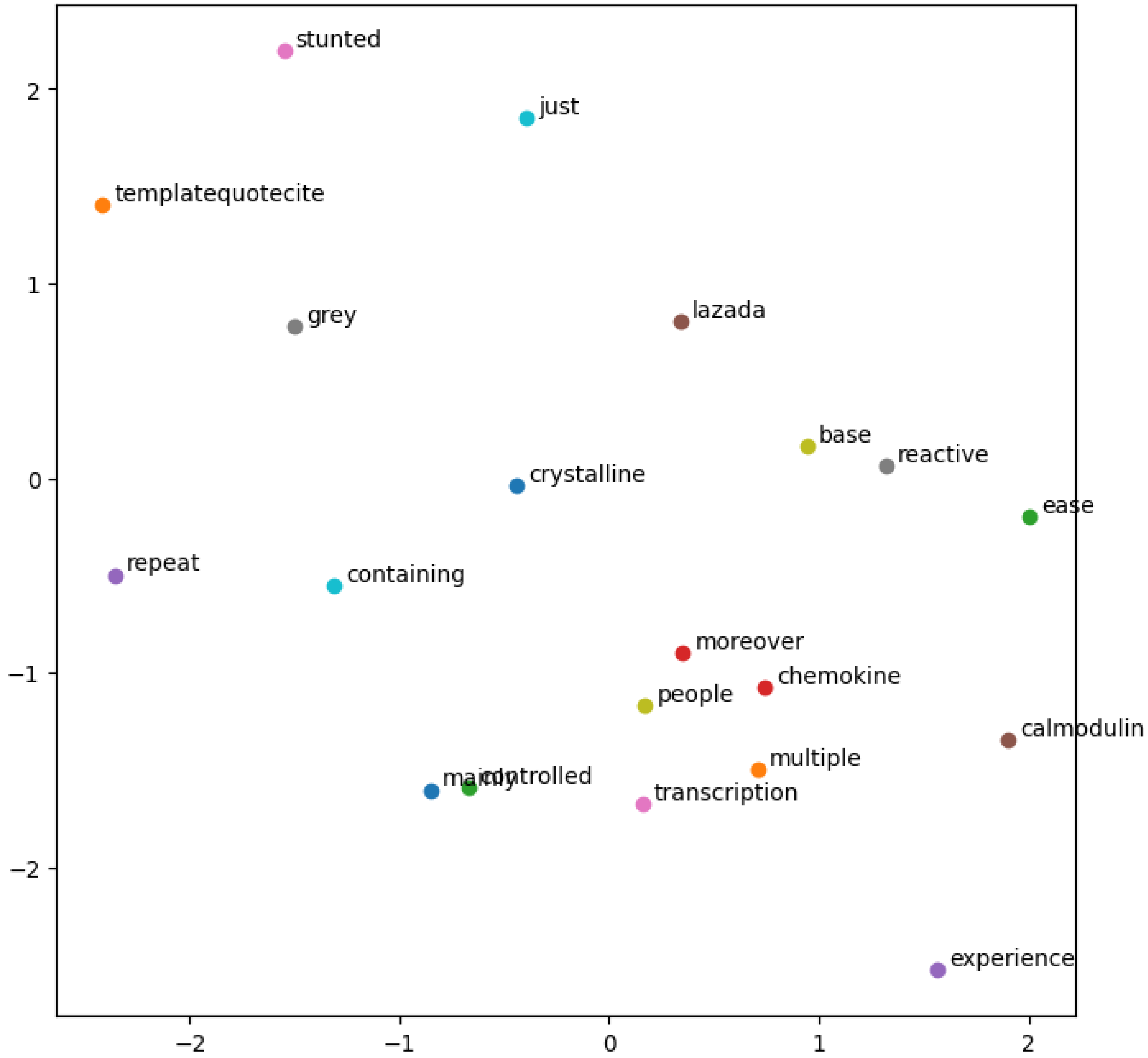
In [27]:
```
num_epochs = 5000
import time
start_time = time.time()
#for epoch
for epoch in range(num_epochs):

    #get random batch
    input_batch, label_batch = random_batch(batch_size, corpus)
    input_batch = torch.LongTensor(input_batch)
    label_batch = torch.LongTensor(label_batch)

    # print(input_batch.shape, label_batch.shape, all_vocabs.shape)

    #loss = model
    loss = model(input_batch, label_batch, all_vocabs)

    #backpropagate
    loss.backward()

    #update alpha
    optimizer.step()

    #print epoch loss
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1} | Loss: {loss:.6f} |", "--- %s seconds ---"
```

```
Epoch 1000 | Loss: 7.799082 | --- 9.038713216781616 seconds ---
Epoch 2000 | Loss: 6.630791 | --- 17.91659903526306 seconds ---
Epoch 3000 | Loss: 6.219728 | --- 27.36138129234314 seconds ---
Epoch 4000 | Loss: 8.737371 | --- 36.891706228256226 seconds ---
Epoch 5000 | Loss: 18.116714 | --- 46.38030409812927 seconds ---
```

---

result from normal skipgram

In [28]:
```
#help me plot fruit cat banana on matplotlib
plt.figure(figsize=(8,8))
for i, word in enumerate(vocabs[:20]): #loop each unique vocab
    x, y = get_embed(word)
    plt.scatter(x, y)
    plt.annotate(word, xy=(x, y), xytext=(5, 2), textcoords='offset points')
plt.show()
```



Conclude the result of question 1

-

Question 2 Try with window size = 2

- we need to fix certain part in this function maybe with one more loop
- and change a start and end to be more dynamic

In [48]:
```
#let's make what we have made into a function (batch function)
#return a batches of data, e.g., =2 --> ['banana', 'apple'], ['banana', 'fru
#also i want these batches to be id, NOT token   --> [5, 4]

def random_batch(batch_size, corpus,windowsize = 1):

    skipgrams = []

    #for each corpus
    for sent in corpus_tokenized:
        #for each sent ["apple", "banana", "fruit"]
        for i in range(windowsize, len(sent) - windowsize): #start from winc
            center_word = word2index[sent[i]]
            '''Change mostly work here both in upper loop and lower loop of
```

```
            for jk in range(1,windowsize+1):
                outside_words = [word2index[sent[i-jk]], word2index[sent[i+j
                for o in outside_words:
                    skipgrams.append([center_word, o])

    #only get a batch, not the entire list
    random_index = np.random.choice(range(len(skipgrams)), batch_size, repla

    #appending some list of inputs and labels
    random_inputs, random_labels = [], []
    for index in random_index:
        random_inputs.append([skipgrams[index][0]])  #center words, this wil
        random_labels.append([skipgrams[index][1]])

    return np.array(random_inputs), np.array(random_labels)
```

In [49]:
```
input, label = random_batch(1, corpus_tokenized,2)

print(f"{input.shape}")
print(f"{label.shape}")
```

```
(1, 1)
(1, 1)
```

The window size do increase but when we getting the batch it still return 1-1 relation

Training step

In [50]:
```
voc_size = len(vocabs)
# voc_size

batch_size = 2 #why?  no reason;
emb_size   = 2 #why?  no reason; usually 50, 100, 300, but 2 so we can plot
model      = Skipgram(voc_size, emb_size)

optimizer  = optim.Adam(model.parameters(), lr=0.001)
```

In [51]:
```
num_epochs = 5000
#for epoch
start_time = time.time()
for epoch in range(num_epochs):

    #get random batch
    input_batch, label_batch = random_batch(batch_size, corpus,2)
    input_batch = torch.LongTensor(input_batch)
    label_batch = torch.LongTensor(label_batch)

    # print(input_batch.shape, label_batch.shape, all_vocabs.shape)

    #loss = model
    loss = model(input_batch, label_batch, all_vocabs)

    #backpropagate
    loss.backward()

    #update alpha
```

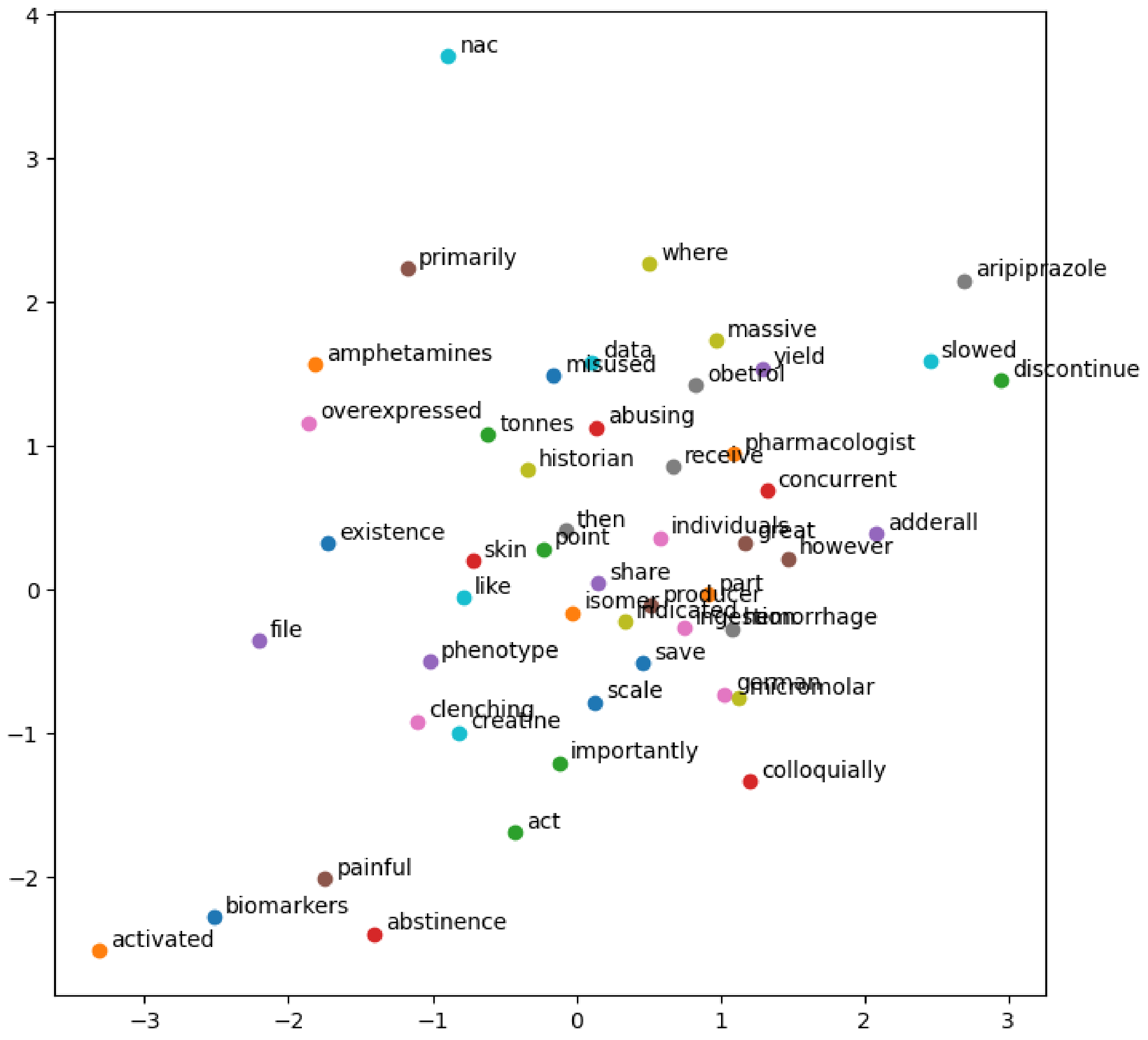
```
    optimizer.step()

    #print epoch loss
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1} | Loss: {loss:.6f} |" ,"--- %s seconds ---"
```

```
Epoch 1000 | Loss: 7.799690 | --- 21.23382878303528 seconds ---
Epoch 2000 | Loss: 7.830691 | --- 41.06342077255249 seconds ---
Epoch 3000 | Loss: 9.453474 | --- 61.18405890464783 seconds ---
Epoch 4000 | Loss: 11.351488 | --- 86.18988800048828 seconds ---
Epoch 5000 | Loss: 7.150196 | --- 108.89053583145142 seconds ---
```

---

result from using windowsize = 2

In [33]:
```
#help me plot fruit cat banana on matplotlib
plt.figure(figsize=(8,8))
for i, word in enumerate(vocabs[:50]): #loop each unique vocab
    x, y = get_embed(word)
    plt.scatter(x, y)
    plt.annotate(word, xy=(x, y), xytext=(5, 2), textcoords='offset points')
plt.show()
```



Conclude the result of question 2

-
-

Question 3 Im plement CBOW

In [1]:
```
def random_batch_for_cbow(batch_size, word_sequence ):

    cbow = []

    for sent in corpus_tokenized:
        for i in range(1, len(sent) - 1): # So we can modify the window size
            target = word2index[sent[i]]
            context = []




            context.append(word2index[sent[i - 1]])


            context.append(word2index[sent[i +  1]])

            # This part is different from skipgram
            # Now we use all context as input and target as label
            for w in context:
                cbow.append([context, target])

    random_inputs = []
    random_labels = []
    random_index = np.random.choice(range(len(cbow)), batch_size, replace=Fa

    for i in random_index:
        random_inputs.append(cbow[i][0])  # Context word that we want as inp
        random_labels.append([cbow[i][1]])  # Target word that we want as la

    return np.array(random_inputs), np.array(random_labels)
```

In [29]:
```
## my idea was to simply swap the center_word and outside_word so they dont
## but it will confuse since we just swap the name but the thing still be th
class CBow(nn.Module):

    def __init__(self, voc_size, emb_size):
        super(CBow, self).__init__()
        self.embedding_center_word  = nn.Embedding(voc_size, emb_size)  #is
        self.embedding_outside_word = nn.Embedding(voc_size, emb_size)

    def forward(self, center_word, outside_word, all_vocabs):
        #center_word, outside_word: (batch_size, 1)
        #all_vocabs: (batch_size, voc_size)

        #convert them into embedding
        center_word_embed  = self.embedding_center_word(center_word)     #(b
        outside_word_embed = self.embedding_outside_word(outside_word)   #(b
        all_vocabs_embed   = self.embedding_outside_word(all_vocabs)     #(b

        #bmm is basically @ or .dot , but across batches (i.e., ignore the b
        top_term = outside_word_embed.bmm(center_word_embed.transpose(1, 2))
```

```
        #(batch_size, 1, emb_size) @ (batch_size, emb_size, 1) = (batch_size

        top_term_exp = torch.exp(top_term)  #exp(uo vc)
        #(batch_size, 1)

        lower_term = all_vocabs_embed.bmm(center_word_embed.transpose(1, 2))
         #(batch_size, voc_size, emb_size) @ (batch_size, emb_size, 1) = (ba

        lower_term_sum = torch.sum(torch.exp(lower_term), 1) #sum exp(uw vc)
        #(batch_size, 1)

        loss_fn = -torch.mean(torch.log(top_term_exp / lower_term_sum))
        #(batch_size, 1) / (batch_size, 1) ==mean==> scalar

        return loss_fn
```

In [30]:
```
voc_size = len(vocabs)
# voc_size

batch_size = 2 #why?  no reason;
emb_size   = 2 #why?  no reason; usually 50, 100, 300, but 2 so we can plot
model      = CBow(voc_size, emb_size)

optimizer  = optim.Adam(model.parameters(), lr=0.001)
```

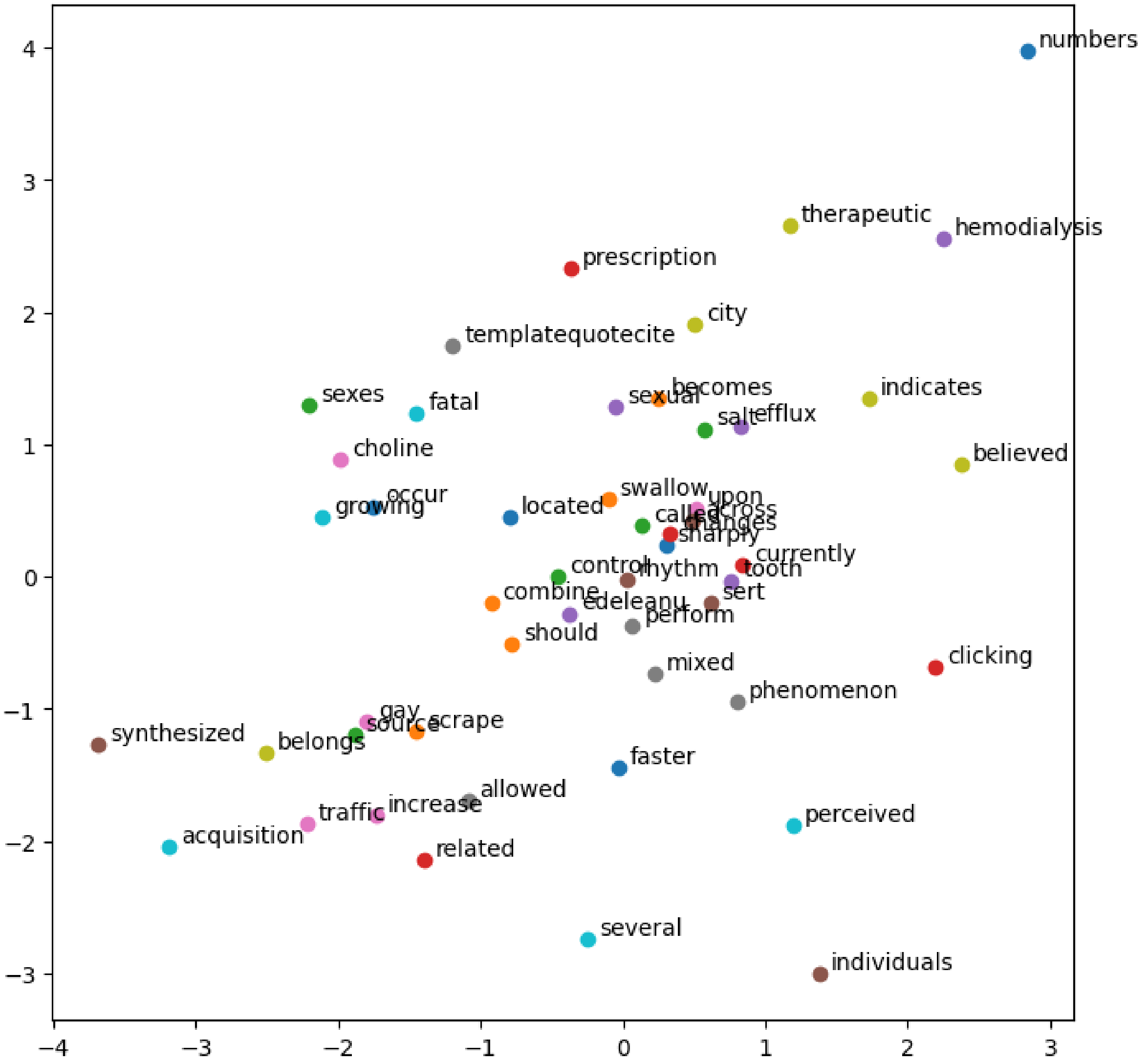
In [32]:
```
import time
```

In [34]:
```
# TODO
num_epochs = 5000
#for epoch
start_time = time.time()
for epoch in range(num_epochs):

    #get random batch
    input_batch, label_batch = random_batch(batch_size, corpus)
    input_batch = torch.LongTensor(input_batch)
    label_batch = torch.LongTensor(label_batch)

    # print(input_batch.shape, label_batch.shape, all_vocabs.shape)

    #loss = model
    loss = model(input_batch, label_batch, all_vocabs)

    #backpropagate
    loss.backward()

    #update alpha
    optimizer.step()

    #print epoch loss
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1} | Loss: {loss:.6f} |" ,"--- %s seconds ---"
```

```
Epoch 1000 | Loss: 7.662072 | --- 10.304527997970581 seconds ---
Epoch 2000 | Loss: 5.936458 | --- 19.786322116851807 seconds ---
Epoch 3000 | Loss: 7.765900 | --- 30.794956922531128 seconds ---
Epoch 4000 | Loss: 7.407135 | --- 40.7482202053070l seconds ---
Epoch 5000 | Loss: 10.823975 | --- 51.35486817359924 seconds ---
```

In [35]:
```
#help me plot fruit cat banana on matplotlib
plt.figure(figsize=(8,8))
```

```
for i, word in enumerate(vocabs[:50]): #loop each unique vocab
    x, y = get_embed(word)
    plt.scatter(x, y)
    plt.annotate(word, xy=(x, y), xytext=(5, 2), textcoords='offset points')
plt.show()
```



Question 4 compare time skipgrams vs. negative sampling version of skipgrams

- this question I do still using the window size of 1 to reduce the training time and not hurt my pc too much

In [ ]:
```
#let's make what we have made into a function (batch function)
#return a batches of data, e.g., =2 --> ['banana', 'apple'], ['banana', 'fru
#also i want these batches to be id, NOT token   --> [5, 4]

def random_batch(batch_size, corpus):

    skipgrams = []

    #for each corpus
    for sent in corpus_tokenized:
        #for each sent ["apple", "banana", "fruit"]
        for i in range(1, len(sent) - 1): #start from 1 to second last
```

```
            center_word = word2index[sent[i]]
            outside_words = [word2index[sent[i-1]], word2index[sent[i+1]]]
            for o in outside_words:
                skipgrams.append([center_word, o])

    #only get a batch, not the entire list
    random_index = np.random.choice(range(len(skipgrams)), batch_size, repla

    #appending some list of inputs and labels
    random_inputs, random_labels = [], []
    for index in random_index:
        random_inputs.append([skipgrams[index][0]])  #center words, this wil
        random_labels.append([skipgrams[index][1]])

    return np.array(random_inputs), np.array(random_labels)
```

In [48]:
```
# input, label = random_batch(10, corpus_tokenized,2)

# print(f"{input.shape}")
# print(f"{label=}")
```

```
(10, 1)
label=array([[ 717],
       [1349],
       [ 598],
       [ 318],
       [ 403],
       [ 250],
       [ 119],
       [1721],
       [ 376],
       [1089]])
```

3. Unigram distribution

$$P(w)=U(w)^{3/4}/Z$$
Defining the probability of sampling negative words``

In [49]:
```
#count all the occurrences of vocabs
from collections import Counter
z = 0.001

word_count = Counter(flatten(corpus_tokenized))
# word_count
```

In [50]:
```
num_total_words = sum([c for w, c in word_count.items()])
# num_total_words
unigram_table = []

for v in vocabs:
    uw = word_count[v]/num_total_words
    uw_alpha = uw ** 0.75
    uw_alpha_dividebyz = int(uw_alpha / z)
    # print("vocab: ", v)
    # print("distribution: ", uw_alpha_dividebyz)
    unigram_table.extend([v] * uw_alpha_dividebyz)

# Counter(unigram_table)
```

4. Negative sampling

A function to get negative samples, based on the current center and outside words in the batch

In [51]:
```
def prepare_sequence(seq, word2index):
    #map(function, list of something)
    #map will look at each of element in this list, and apply this function
    idxs = list(map(lambda w: word2index[w] if word2index.get(w) is not None
    return torch.LongTensor(idxs)

import random
#you don't want to pick samples = targets, basically negative samples
#k = number of negative samples - how many? they found 10 is the best
#will be run during training
#after random_batch,
def negative_sampling(targets, unigram_table, k):
    #targets is already in id.....
    #but the unigram_table is in word....
    #1. get the batch size of this targets
    batch_size = targets.shape[0]
    neg_samples = []
    #2. for each batch
    for i in range(batch_size):
        #randomly pick k negative words from unigram_table
        target_index = targets[i].item()  #looping each of the batch....
        nsample = []
        while len(nsample) < k:
            neg = random.choice(unigram_table)
            #if this word == target, skip this word
            if word2index[neg] == target_index:
                continue
            nsample.append(neg)
        #append this word to some list
        neg_samples.append(prepare_sequence(nsample, word2index).reshape(1,
    return torch.cat(neg_samples)  #tensor[[], []]
```

5. Model and test our method

$$\mathbf{J}_{\text{neg-sample}}(\mathbf{v}_c,o,\mathbf{U})=-\log(\sigma(\mathbf{u}_o^T\mathbf{v}_c))-\sum_{k=1}^K\log(\sigma(-\mathbf{u}_k^T\mathbf{v}_c))$$

In [52]:
```
class SkipgramNeg(nn.Module):

    def __init__(self, voc_size, emb_size):
        super(SkipgramNeg, self).__init__()
        self.embedding_center_word  = nn.Embedding(voc_size, emb_size)
        self.embedding_outside_word = nn.Embedding(voc_size, emb_size)
        self.logsigmoid = nn.LogSigmoid()

    def forward(self, center_words, outside_words, negative_words):
        #center_words, outside_words: (batch_size, 1)
        #negative_words:  (batch_size, k)

        center_embed  = self.embedding_center_word(center_words)    #(batch_
        outside_embed = self.embedding_outside_word(outside_words)  #(batch_
        neg_embed     = self.embedding_outside_word(negative_words) #(batch_

        uovc          =  outside_embed.bmm(center_embed.transpose(1, 2)).squ
        ukvc          = -neg_embed.bmm(center_embed.transpose(1, 2)).squeeze
        ukvc_sum      =  torch.sum(ukvc, 1).view(-1, 1) #(batch_size, 1)

        loss = self.logsigmoid(uovc) + self.logsigmoid(ukvc_sum)  #(batch_si

        return -torch.mean(loss)  #scalar, loss should be scalar, to call ba
```

In [53]:
```
input, label = random_batch(batch_size, corpus_tokenized)
input_tensor = torch.LongTensor(input)
label_tensor = torch.LongTensor(label)
```

In [54]:
```
emb_size = 2 #usually, this can be 50, 100, or 300
voc_size = len(vocabs)
model = SkipgramNeg(voc_size, emb_size)
neg_tensor = negative_sampling(label_tensor, unigram_table, 5)
#this should give one number
loss = model(input_tensor, label_tensor, neg_tensor)
```

4. Training

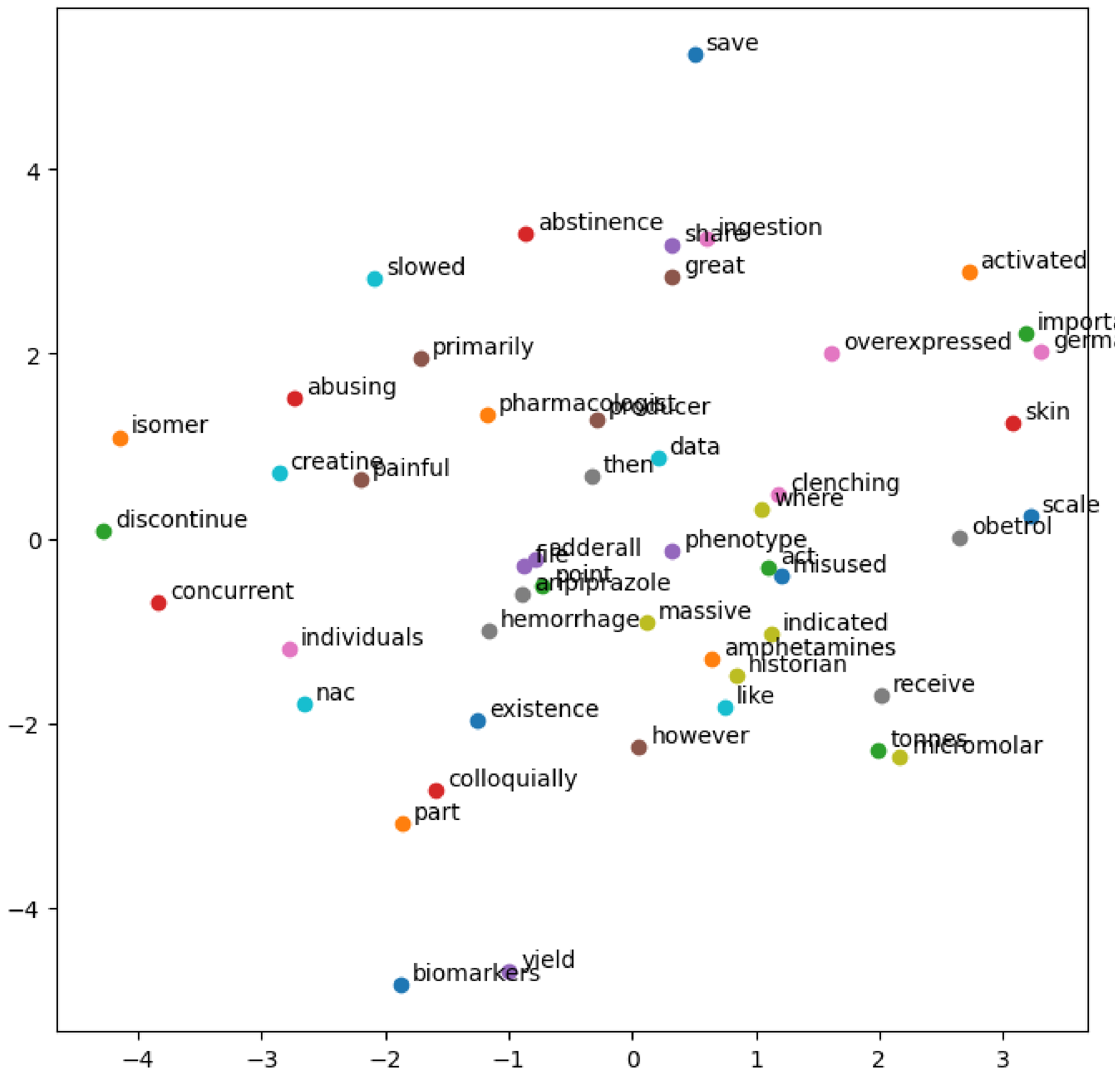
In [55]:
```
voc_size   = len(vocabs)
batch_size = 2 #why?  no reason;
emb_size   = 2 #why?  no reason; usually 50, 100, 300, but 2 so we can plot
model      = SkipgramNeg(voc_size, emb_size)

optimizer  = optim.Adam(model.parameters(), lr=0.001)
```

In [56]:
```
num_epochs = 5000
#for epoch
start_time = time.time()
for epoch in range(num_epochs):

    #get random batch
    input_batch, label_batch = random_batch(batch_size, corpus)
    input_batch = torch.LongTensor(input_batch)
    label_batch = torch.LongTensor(label_batch)
    neg_batch   = negative_sampling(label_batch, unigram_table, 5)

    #loss = model
    loss = model(input_batch, label_batch, neg_batch)
```

```
    #backpropagate
    loss.backward()

    #update alpha
    optimizer.step()

    #print epoch loss
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1} | Loss: {loss:.6f} |", "--- %s seconds ---"
```

```
Epoch 1000 | Loss: 2.569600 | --- 10.493175029754639 seconds ---
Epoch 2000 | Loss: 1.266238 | --- 20.70341920852661 seconds ---
Epoch 3000 | Loss: 2.680556 | --- 29.979011058807373 seconds ---
Epoch 4000 | Loss: 10.724624 | --- 40.1818060874939 seconds ---
Epoch 5000 | Loss: 4.843665 | --- 49.31004619598389 seconds ---
```

In [57]:
```
#help me plot fruit cat banana on matplotlib
plt.figure(figsize=(8,8))
for i, word in enumerate(vocabs[:50]): #loop each unique vocab
    x, y = get_embed(word)
    plt.scatter(x, y)
    plt.annotate(word, xy=(x, y), xytext=(5, 2), textcoords='offset points')
plt.show()
```



result from negative sampling skg

Question conclusion

result of normal skipgram

result of neg sampling skipgram