Introductory Machine Learning: Assignment 5

Deadline:

Assignment 5 is due Thursday, November 11 at 11:59pm. Late work will not be accepted as per the course policies (see the Syllabus and Course policies on Canvas.

Directly sharing answers is not okay, but discussing problems with the course staff or with other students is encouraged.

You should start early so that you have time to get help if you're stuck. The drop-in office hours schedule can be found on Canvas. You can also post questions or start discussions on Ed Discussion. The problems are broken up into steps that should help you to make steady progress.

Submission:

Submit your assignment as a .pdf on Gradescope, and as a .ipynb on Canvas. You can access Gradescope through Canvas on the left-side of the class home page. The problems in each homework assignment are numbered. Note: When submitting on Gradescope, please select the correct pages of your pdf that correspond to each problem. This will allow graders to find your complete solution to each problem.

To produce the .pdf, please do the following in order to preserve the cell structure of the notebook:

1. Go to "File" at the top-left of your Jupyter Notebook
2. Under "Download as", select "HTML (.html)"
3. After the .html has downloaded, open it and then select "File" and "Print" (note you will not actually be printing)
4. From the print window, select the option to save as a .pdf

Topics

1. Language models
2. Work embeddings

Problem 1: Gutenberg Books Language Models (15 points)

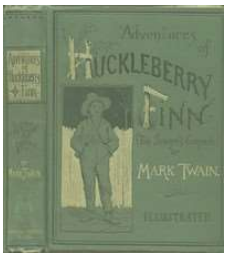
For this problem you will process books from the Project Gutenberg site which is a public respository of large numbers of books that are in the public domain. You'll build *character-based* (as opposed to word-based) language models on one book, and predict the letters of the other book using the model.

In [ ]:
```
import numpy as np
from collections import Counter
import matplotlib.pyplot as plots
%matplotlib inline
```

The following helper function `read_url` reads in the text at the given url, and then uses some

regular expressions to process the book, removing everything but the letters a-z, space and period.

In [ ]:
```
from urllib.request import urlopen
import re
def read_url(url):
    return re.sub('\\s+', ' ', urlopen(url).read().decode())

def process_text(text):
    text = re.sub('[^a-zA-z .]', '', text.lower())
    return re.sub('[\[\]\_]', '', text)
```



The online book for "Adventures of Huckleberry Finn," by Mark Twain, is here. From this web site you can see various metadata for the book as well as the link the text itself, which is https://www.gutenberg.org/files/76/76-0.txt

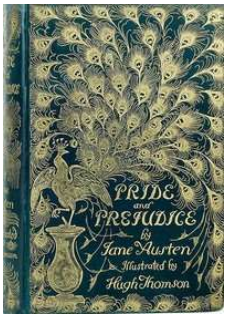
The book for Mark Twain's "A Connecticut Yankee in King Arthur's Court" is here. In the following cell we read in both of these books, and remove all characters except a-z, space, and period.

In [ ]:
```
huck_finn_url = 'https://www.gutenberg.org/files/76/76-0.txt'
huck_finn_text_raw = read_url(huck_finn_url)
huck_finn_text = process_text(huck_finn_text_raw)

ct_yankee_url = 'https://www.gutenberg.org/files/86/86-0.txt'
ct_yankee_text_raw = read_url(ct_yankee_url)
ct_yankee_text = process_text(ct_yankee_text_raw)
```

In [ ]:
```
print("\nSample of raw text:\n")
print(huck_finn_text_raw[10000:11000])

print("\nSample of processed text:\n")
print(huck_finn_text[10000:11000])
```

```
Sample of raw text:

my old rags and my sugar-hogshead again, and was free and satisfied. But Tom Sawyer he h
unted me up and said he was going to start a band of robbers, and I might join if I woul
d go back to the widow and be respectable. So I went back. The widow she cried over me,
and called me a poor lost lamb, and she called me a lot of other names, too, but she nev
er meant no harm by it. She put me in them new clothes again, and I couldn't do nothing
but sweat and sweat, and feel all cramped up. Well, then, the old thing commenced again.
The widow rung a bell for supper, and you had to come to time. When you got to the table
you couldn't go right to eating, but you had to wait for the widow to tuck down her head
and grumble a little over the victuals, though there warn't really anything the matter w
ith them,--that is, nothing only everything was cooked by itself. In a barrel of odds an
d ends it is different; things get mixed up, and the juice kind of swaps around, and the
things go better. After supp

Sample of processed text:
```

```
he widow rung a bell for supper and you had to come to time. when you got to the table y
ou couldnt go right to eating but you had to wait for the widow to tuck down her head an
d grumble a little over the victuals though there warnt really anything the matter with
themthat is nothing only everything was cooked by itself. in a barrel of odds and ends i
t is different things get mixed up and the juice kind of swaps around and the things go
better. after supper she got out her book and learned me about moses and the bulrushers
and i was in a sweat to find out all about him but by and by she let it out that moses h
ad been dead a considerable long time so then i didnt care no more about him because i d
ont take no stock in dead people. pretty soon i wanted to smoke and asked the widow to l
et me. but she wouldnt. she said it was a mean practice and wasnt clean and i must try t
o not do it any more. that is just the way with some people. they get down on a thing wh
en they dont know nothing about
```



The online book for "Pride and Prejudice", by Jane Austen, is here. And here is the online book for Jane Austen's "Emma". In the following cell we read in both of these books, and remove all characters except a-z, space, and period.

In [ ]:
```
pride_and_prejudice_url = 'https://www.gutenberg.org/files/1342/1342-0.txt'
pride_and_prejudice_text_raw = read_url(pride_and_prejudice_url)
pride_and_prejudice_text = process_text(pride_and_prejudice_text_raw)


emma_url = 'https://www.gutenberg.org/files/158/158-0.txt'
emma_text_raw = read_url(emma_url)
emma_text = process_text(emma_text_raw)
```

In [ ]:
```
print("\nSample of raw text:\n")
print(emma_text_raw[10000:11000])

print("\nSample of processed text:\n")
print(emma_text[10000:11000])
```

```
Sample of raw text:

all are.” Emma spared no exertions to maintain this happier flow of ideas, and hoped, by
the help of backgammon, to get her father tolerably through the evening, and be attacked
by no regrets but her own. The backgammon-table was placed; but a visitor immediately af
terwards walked in and made it unnecessary. Mr. Knightley, a sensible man about seven or
eight-and-thirty, was not only a very old and intimate friend of the family, but particu
larly connected with it, as the elder brother of Isabella’s husband. He lived about a mi
le from Highbury, was a frequent visitor, and always welcome, and at this time more welc
ome than usual, as coming directly from their mutual connexions in London. He had return
ed to a late dinner, after some days’ absence, and now walked up to Hartfield to say tha
t all were well in Brunswick Square. It was a happy circumstance, and animated Mr. Woodh
ouse for some time. Mr. Knightley had a cheerful manner, which always did him good; and
his many inquiries after “poo

Sample of processed text:
```

```
diately afterwards walked in and made it unnecessary. mr. knightley a sensible man about
seven or eightandthirty was not only a very old and intimate friend of the family but pa
rticularly connected with it as the elder brother of isabellas husband. he lived about a
mile from highbury was a frequent visitor and always welcome and at this time more welco
me than usual as coming directly from their mutual connexions in london. he had returned
to a late dinner after some days absence and now walked up to hartfield to say that all
were well in brunswick square. it was a happy circumstance and animated mr. woodhouse fo
r some time. mr. knightley had a cheerful manner which always did him good and his many
inquiries after poor isabella and her children were answered most satisfactorily. when t
his was over mr. woodhouse gratefully observed it is very kind of you mr. knightley to c
ome out at this late hour to call upon us. i am afraid you must have had a shocking wal
k. not at all sir. it is a bea
```

The following cell defines some helper code. You should just run this cell; do not change any of the code.

The first function, `ngrams`, takes some input text and a value of `n`. The function then iterates over the string and counts the number of occurrences of each substring of `n` characters. This is done with the very handy `Counter` class.

We then define a class `language_model` that is a 4-gram character-based language model. The probability of the "next character" is computed using linear interpolation, as described in class. A weight is assigned to unigrams, bigrams, trigrams, and four-grams (quadgrams?). The bigram probability that, for example, the letter `t` follows the letter `h` is the count of the bigram `ht` divided by the count of the unigram `h`. We add a little bit (1e-10) to the denominator to avoid dividing by zero.

We return the logarithm of the probability, because this will be convenient when computing perplexities.

In [ ]:

```
def ngrams(text, n=2):
    return Counter([text[(i-n):i] for i in np.arange(n, len(text)+1)])

class language_model:

    def __init__(self, text):
        self.one = ngrams(text, 1)
        self.two = ngrams(text, 2)
        self.three = ngrams(text, 3)
        self.four = ngrams(text, 4)
        self.weight = [0.1, 0.2, 0.3, 0.4]

    def set_weights(self, weights):
        self.weight = weights / np.sum(weights)

    def log_probability(self, gram):
        numer = [self.one[gram[3:]], self.two[gram[2:]], self.three[gram[1:]], self.fou
        denom = [sum(self.one[g] for g in self.one), self.one[gram[2:3]], self.two[gram
        prob = 0
        for i in np.arange(4):
            prob += self.weight[i] * numer[i] / (denom[i]+1e-10)
        return np.log(prob)
```

Problem 1.1

Just to be sure we understand what a character-based language model is, let's write an expression for the probability in an example. Suppose the language model assigns weight $w_1 = 0.1$ to the unigram model, weight $w_2 = 0.2$ to the bigram model, weight $w_3 = 0.3$ to the trigram model, and weight $w_4 = .4$ to the four-gram model. Note that we must have $w_1 + w_2 + w_3 + w_4 = 1$.

Write an expression for the probability $p(\mathrm{z} \,|\, \mathrm{qui})$ that the letter z follows the three letters qui. Assume that the unigram, bigram, trigram, and four-gram components are given by ratios of counts in the training data, as in the code above. For example, the bigram probability would be written as

$$\frac{\mathrm{count}(iz)}{\mathrm{count}(i)}$$

$$p(\mathrm{z} \,|\, \mathrm{qui}) = w1 * \frac{\mathrm{count}(z)}{\mathrm{count}(allletters)} w_2 * \frac{\mathrm{count}(iz)}{\mathrm{count}(i)} + w_3 * \frac{\mathrm{count}(uiz)}{\mathrm{count}(ui)} + w_4 * \frac{\mathrm{count}(quiz)}{\mathrm{count}(qui)}$$

Now, the cell below constructs two language models, one on the text of Jane Austen's "Emma", the other on the text of Mark Twain's "Huckleberry Finn".

In [ ]:
```
emma_lm = language_model(emma_text)
huck_finn_lm = language_model(huck_finn_text)
ct_lm = language_model(ct_yankee_text)
temp_text = ngrams(emma_text, n=4)
len(temp_text)
temp_text
```

Out[ ]:
```
Counter({'the ': 5397,
         'he p': 533,
         'e pr': 283,
         ' pro': 649,
         'proj': 93,
         'roje': 93,
         'ojec': 93,
         'ject': 288,
         'ect ': 396,
         'ct g': 91,
         't gu': 93,
         ' gut': 88,
         'gute': 97,
         'uten': 98,
         'tenb': 98,
         'enbe': 97,
         'nber': 97,
         'berg': 97,
         'erg ': 31,
         'rg e': 4,
         'g eb': 4,
         ' ebo': 20,
         'eboo': 21,
         'book': 37,
         'ook ': 192,
         'ok o': 19,
         'k of': 150,
         ' of ': 4364,
         'of e': 130,
         'f em': 21,
         ' emm': 845,
         'emma': 870,
         'mma ': 720,
```

```
'ma b': 27,
'a by': 3,
' by ': 591,
'by j': 4,
'y ja': 7,
' jan': 299,
'jane': 303,
'ane ': 278,
'ne a': 120,
'e au': 19,
' aus': 4,
'aust': 5,
'uste': 20,
'sten': 71,
'ten ': 187,
'en t': 359,
'n th': 1317,
' thi': 1509,
'this': 574,
'his ': 1697,
'is e': 81,
's eb': 8,
'ok i': 14,
'k is': 11,
' is ': 1236,
'is f': 148,
's fo': 187,
' for': 1715,
'for ': 1347,
'or t': 389,
'r th': 769,
' the': 7424,
'he u': 82,
'e us': 72,
' use': 130,
'use ': 527,
'se o': 144,
'e of': 1094,
'of a': 403,
'f an': 206,
' any': 716,
'anyo': 5,
'nyon': 5,
'yone': 6,
'one ': 693,
'e an': 1002,
'anyw': 19,
'nywh': 19,
'ywhe': 23,
'wher': 129,
'here': 915,
'ere ': 1401,
're i': 386,
'e in': 662,
' in ': 2183,
'in t': 714,
'e un': 119,
' uni': 43,
'unit': 51,
'nite': 47,
'ited': 95,
'ted ': 792,
'ed s': 191,
'd st': 81,
' sta': 244,
```

```
'stat': 100,
'tate': 117,
'ates': 223,
'tes ': 203,
'es a': 291,
's an': 683,
' and': 4605,
'and ': 5165,
'nd m': 456,
'd mo': 98,
' mos': 253,
'most': 362,
'ost ': 385,
'st o': 144,
't ot': 15,
' oth': 319,
'othe': 588,
'ther': 1927,
'her ': 3298,
'er p': 200,
'r pa': 86,
' par': 431,
'part': 376,
'arts': 9,
'rts ': 24,
'ts o': 80,
's of': 605,
'of t': 945,
'f th': 992,
'he w': 1337,
'e wo': 477,
' wor': 445,
'worl': 87,
'orld': 87,
'rld ': 64,
'ld a': 140,
'd at': 169,
' at ': 1029,
'at n': 54,
't no': 277,
' no ': 680,
'no c': 37,
'o co': 216,
' cos': 6,
'cost': 8,
'st a': 236,
't an': 561,
'nd w': 441,
'd wi': 388,
' wit': 1536,
'with': 1526,
'ith ': 1315,
'th a': 315,
'h al': 79,
' alm': 89,
'almo': 91,
'lmos': 91,
'st n': 92,
'no r': 37,
'o re': 168,
' res': 340,
'rest': 217,
'estr': 22,
'stri': 57,
'tric': 20,
```

```
'rict': 9,
'icti': 28,
'ctio': 293,
'tion': 1572,
'ions': 335,
'ons ': 474,
'ns w': 80,
's wh': 234,
' wha': 514,
'what': 563,
'hats': 21,
'atso': 2,
'tsoe': 2,
'soev': 2,
'oeve': 8,
'ever': 1271,
'ver.': 46,
'er. ': 424,
'r. y': 22,
'. yo': 246,
' you': 2655,
'you ': 1933,
'ou m': 212,
'u ma': 63,
' may': 236,
'may ': 235,
'ay c': 28,
'y co': 236,
' cop': 50,
'copy': 38,
'opy ': 13,
'py i': 14,
'y it': 77,
' it ': 2175,
'it g': 9,
't gi': 38,
' giv': 304,
'give': 271,
'ive ': 375,
've i': 113,
'e it': 305,
'it a': 250,
't aw': 23,
' awa': 169,
'away': 141,
'way ': 239,
'ay o': 66,
'y or': 59,
' or ': 548,
'or r': 37,
'r re': 126,
' reu': 3,
'reus': 2,
'euse': 2,
'se i': 118,
'it u': 22,
't un': 80,
' und': 231,
'unde': 244,
'nder': 361,
'der ': 221,
'er t': 751,
'he t': 398,
'e te': 69,
' ter': 44,
```

```
'term': 82,
'erms': 31,
'rms ': 33,
'ms o': 36,
'rg l': 16,
'g li': 44,
' lic': 19,
'lice': 24,
'icen': 23,
'cens': 27,
'ense': 123,
'nse ': 101,
' inc': 152,
'incl': 58,
'nclu': 44,
'clud': 42,
'lude': 28,
'uded': 18,
'ded ': 285,
'ed w': 338,
'th t': 270,
'h th': 401,
'k or': 19,
'or o': 92,
'r on': 78,
' onl': 334,
'onli': 6,
'nlin': 5,
'line': 76,
'ine ': 182,
'e at': 198,
'at w': 225,
't ww': 7,
' www': 9,
'www.': 9,
'ww.g': 9,
'w.gu': 9,
'.gut': 9,
'erg.': 9,
'rg.o': 9,
'g.or': 9,
'.org': 9,
'org.': 2,
'rg. ': 2,
'g. i': 50,
'. if': 101,
' if ': 481,
'if y': 107,
'f yo': 205,
'ou a': 290,
'u ar': 153,
' are': 476,
'are ': 662,
're n': 129,
'e no': 412,
' not': 2473,
'not ': 2290,
'ot l': 78,
't lo': 53,
' loc': 13,
'loca': 11,
'ocat': 16,
'cate': 43,
'ated': 155,
'ed i': 439,
```

```
'd in': 430,
'es y': 31,
's yo': 171,
'ou w': 265,
'u wi': 122,
' wil': 608,
'will': 635,
'ill ': 1022,
'll h': 212,
'l ha': 92,
' hav': 1472,
'have': 1343,
'ave ': 1440,
've t': 232,
'e to': 992,
' to ': 5206,
'to c': 219,
'o ch': 48,
' che': 79,
'chec': 19,
'heck': 19,
'eck ': 9,
'ck t': 23,
'k th': 79,
'he l': 343,
'e la': 151,
' law': 31,
'laws': 13,
'aws ': 15,
'ws o': 19,
'he c': 932,
'e co': 805,
' cou': 1017,
'coun': 142,
'ount': 142,
'untr': 30,
'ntry': 43,
'try ': 59,
'ry w': 171,
'y wh': 155,
' whe': 538,
're y': 146,
'e yo': 463,
're l': 44,
'e lo': 144,
'ed b': 310,
'd be': 913,
' bef': 258,
'befo': 259,
'efor': 333,
'fore': 350,
'ore ': 740,
're u': 21,
' usi': 8,
'usin': 86,
'sing': 240,
'ing ': 4560,
'ng t': 939,
'g th': 441,
'ook.': 6,
'ok. ': 6,
'k. t': 11,
'. ti': 9,
' tit': 2,
'titl': 6,
```

```
'itle': 6,
'tle ': 395,
'le e': 57,
'e em': 54,
'ma a': 71,
'a au': 1,
' aut': 17,
'auth': 10,
'utho': 10,
'thor': 42,
'hor ': 3,
'or j': 19,
'r ja': 38,
'en r': 26,
'n re': 89,
' rel': 54,
'rele': 11,
'elea': 3,
'leas': 406,
'ease': 171,
'ase ': 108,
'se d': 20,
'e da': 126,
' dat': 8,
'date': 13,
'ate ': 302,
'te a': 159,
' aug': 11,
'augu': 10,
'ugus': 8,
'gust': 20,
'ust ': 768,
'st  ': 1,
't  e': 1,
'  eb': 1,
'ok  ': 1,
'k  m': 1,
'  mo': 1,
'st r': 61,
't re': 208,
' rec': 227,
'rece': 86,
'ecen': 7,
'cent': 26,
'entl': 185,
'ntly': 137,
'tly ': 350,
'ly u': 39,
'y up': 10,
' upd': 2,
'upda': 2,
'pdat': 2,
'ed j': 16,
'd ju': 42,
' jun': 8,
'june': 8,
'une ': 44,
'ne  ': 1,
'e   ': 1,
'   l': 1,
'  la': 1,
' lan': 30,
'lang': 15,
'angu': 22,
'ngua': 10,
```

```
'guag': 10,
'uage': 10,
'age ': 210,
'ge e': 5,
'e en': 165,
' eng': 118,
'engl': 19,
'ngli': 9,
'glis': 9,
'lish': 52,
'ish ': 181,
'sh c': 6,
'h ch': 4,
' cha': 396,
'char': 165,
'hara': 72,
'arac': 47,
'ract': 87,
'acte': 54,
'cter': 47,
'ter ': 793,
'er s': 461,
'r se': 89,
' set': 137,
'set ': 68,
'et e': 21,
't en': 62,
' enc': 57,
'enco': 53,
'ncod': 1,
'codi': 1,
'odin': 3,
'ding': 284,
'ng u': 85,
'g ut': 1,
' utf': 1,
'utf ': 1,
'tf p': 1,
'f pr': 47,
'prod': 37,
'rodu': 66,
'oduc': 66,
'duce': 72,
'uced': 37,
'ced ': 138,
'd by': 232,
'by a': 84,
'y an': 515,
' an ': 465,
'an a': 282,
'n an': 597,
' ano': 114,
'anon': 2,
'nony': 2,
'onym': 2,
'nymo': 2,
'ymou': 19,
'mous': 7,
'ous ': 228,
'us v': 4,
's vo': 10,
' vol': 22,
'volu': 25,
'olun': 18,
'lunt': 20,
```

```
'unte': 42,
'ntee': 12,
'teer': 7,
'eer ': 9,
'er a': 787,
'r an': 503,
'nd d': 133,
'd da': 17,
' dav': 1,
'davi': 1,
'avid': 1,
'vid ': 1,
'id w': 19,
' wid': 10,
'widg': 1,
'idge': 19,
'dger': 1,
'ger ': 131,
'er  ': 1,
'r  s': 1,
'  st': 2,
'star': 20,
'tart': 19,
'art ': 132,
'rt o': 161,
't of': 736,
'ok e': 6,
'k em': 6,
'ma  ': 2,
'a  e': 1,
'  em': 1,
'en c': 49,
'n co': 158,
' con': 946,
'cont': 160,
'onte': 22,
'nten': 85,
'tent': 154,
'ents': 116,
'nts ': 158,
'ts v': 8,
'olum': 7,
'lume': 6,
'ume ': 18,
'me i': 171,
'e i.': 1,
' i. ': 4,
'i. c': 37,
'. ch': 170,
'chap': 113,
'hapt': 112,
'apte': 112,
'pter': 112,
'er i': 407,
'r i.': 3,
'r ii': 12,
' ii.': 4,
'ii. ': 26,
' iii': 8,
'iii.': 13,
'r iv': 6,
' iv.': 3,
'iv. ': 6,
'v. c': 12,
'er v': 94,
```

```
'r v.': 3,
' v. ': 3,
'r vi': 60,
' vi.': 3,
'vi. ': 6,
' vii': 12,
'vii.': 6,
'viii': 12,
'r ix': 6,
' ix.': 3,
'ix. ': 5,
'x. c': 6,
'er x': 56,
'r x.': 3,
' x. ': 3,
'r xi': 26,
' xi.': 3,
'xi. ': 3,
' xii': 12,
'xii.': 3,
'xiii': 6,
' xiv': 6,
'xiv.': 3,
'r xv': 24,
' xv.': 3,
'xv. ': 3,
' xvi': 18,
'xvi.': 3,
'xvii': 12,
'i. v': 2,
'. vo': 7,
'e ii': 4,
' xix': 2,
'xix.': 1,
'x. v': 1,
'e i ': 306,
' i c': 267,
'i ch': 6,
'r i ': 115,
' i e': 23,
'i em': 7,
'ma w': 134,
'a wo': 91,
' woo': 323,
'wood': 323,
'oodh': 332,
'odho': 318,
'dhou': 318,
'hous': 457,
'ouse': 440,
'se h': 104,
'e ha': 1036,
' han': 135,
'hand': 144,
'ands': 92,
'ndso': 52,
'dsom': 46,
'some': 523,
'ome ': 626,
'me c': 47,
'e cl': 37,
' cle': 67,
'clev': 32,
'leve': 39,
'ver ': 889,
```

```
'nd r': 107,
'd ri': 7,
' ric': 34,
'rich': 35,
'ich ': 578,
'ch w': 78,
'h wi': 23,
'h a ': 357,
' a c': 241,
'a co': 189,
' com': 868,
'comf': 127,
'omfo': 127,
'mfor': 128,
'fort': 300,
'orta': 87,
'rtab': 51,
'tabl': 166,
'able': 593,
'ble ': 707,
'le h': 88,
'e ho': 242,
' hom': 137,
'home': 138,
'me a': 213,
'nd h': 579,
'd ha': 537,
' hap': 281,
'happ': 303,
'appy': 144,
'ppy ': 124,
'py d': 3,
'y di': 117,
' dis': 479,
'disp': 106,
'ispo': 53,
'spos': 57,
'posi': 78,
'osit': 90,
'siti': 63,
'itio': 91,
'ion ': 1334,
'on s': 149,
'n se': 80,
' see': 571,
'seem': 215,
'eeme': 171,
'emed': 151,
'med ': 258,
'ed t': 1149,
'd to': 1003,
'to u': 47,
'o un': 40,
'ite ': 403,
'te s': 57,
'e so': 378,
' som': 436,
'me o': 144,
'he b': 307,
'e be': 598,
' bes': 105,
'best': 90,
'est ': 396,
'st b': 244,
't bl': 14,
```

```
' ble': 31,
'bles': 49,
'less': 178,
'essi': 154,
'ssin': 93,
'ings': 268,
'ngs ': 228,
'gs o': 28,
'f ex': 26,
' exi': 11,
'exis': 12,
'xist': 12,
'iste': 128,
'tenc': 31,
'ence': 389,
'nce ': 810,
'ce a': 200,
' had': 1616,
'had ': 1619,
'ad l': 38,
'd li': 65,
' liv': 84,
'live': 78,
'ived': 124,
'ved ': 340,
'ed n': 111,
'd ne': 119,
' nea': 78,
'near': 72,
'earl': 75,
'arly': 125,
'rly ': 171,
'ly t': 308,
'y tw': 15,
' twe': 25,
'twen': 31,
'went': 77,
'enty': 35,
'ntyo': 1,
'tyon': 1,
'ne y': 23,
'e ye': 44,
' yea': 85,
'year': 86,
'ears': 104,
'ars ': 107,
'rs i': 32,
's in': 402,
'ld w': 68,
'th v': 8,
'h ve': 19,
' ver': 1188,
'very': 1663,
'ery ': 1674,
'ry l': 101,
'y li': 117,
' lit': 379,
'litt': 364,
'ittl': 365,
'ttle': 420,
'le t': 280,
'to d': 206,
'o di': 62,
'dist': 154,
'istr': 78,
```

```
'stre': 79,
'tres': 44,
'ress': 209,
'ess ': 687,
'ss o': 153,
's or': 75,
'or v': 9,
'r ve': 39,
' vex': 13,
'vex ': 1,
'ex h': 1,
'x he': 4,
' her': 2929,
'her.': 254,
'r. s': 78,
'. sh': 468,
' she': 2325,
'she ': 2348,
'e wa': 1008,
' was': 2393,
'was ': 2379,
'as t': 591,
's th': 751,
'he y': 66,
'youn': 201,
'oung': 198,
'unge': 8,
'nges': 14,
'gest': 30,
'e tw': 68,
' two': 180,
'two ': 165,
'wo d': 11,
'o da': 28,
' dau': 69,
'daug': 69,
'augh': 154,
'ught': 543,
'ghte': 144,
'hter': 73,
'ters': 110,
'ers ': 365,
'rs o': 70,
'f a ': 208,
' a m': 323,
'a mo': 138,
't af': 29,
' aff': 127,
'affe': 93,
'ffec': 120,
'fect': 247,
'ecti': 228,
'iona': 87,
'onat': 39,
'nate': 70,
'te i': 53,
' ind': 356,
'indu': 57,
'ndul': 25,
'dulg': 25,
'ulge': 21,
'lgen': 10,
'gent': 109,
'ent ': 955,
'nt f': 72,
```

```
't fa': 59,
' fat': 226,
'fath': 210,
'athe': 407,
'ad i': 78,
'in c': 67,
'cons': 311,
'onse': 76,
'nseq': 35,
'sequ': 40,
'eque': 64,
'quen': 56,
'uenc': 55,
'ce o': 260,
'of h': 715,
'f he': 394,
'r si': 107,
' sis': 45,
'sist': 108,
'ster': 122,
'rs m': 23,
's ma': 132,
' mar': 289,
'marr': 177,
'arri': 721,
'rria': 116,
'riag': 116,
'iage': 116,
'ge b': 24,
' bee': 761,
'been': 760,
'een ': 924,
'en m': 133,
'n mi': 79,
' mis': 714,
'mist': 51,
'f hi': 285,
' his': 1150,
'is h': 123,
's ho': 73,
' hou': 218,
'se f': 54,
'e fr': 158,
' fro': 571,
'from': 561,
'rom ': 554,
'om a': 94,
'm a ': 54,
' a v': 229,
'a ve': 207,
'ry e': 53,
'y ea': 25,
' ear': 87,
'ly p': 74,
'y pe': 45,
' per': 564,
'peri': 104,
'erio': 165,
'riod': 20,
'iod.': 3,
'od. ': 28,
'd. h': 73,
'. he': 465,
'er m': 266,
'r mo': 67,
```

```
' mot': 96,
'moth': 83,
'er h': 420,
'r ha': 268,
'ad d': 53,
'd di': 72,
' die': 12,
'died': 8,
'ied ': 372,
' too': 301,
'too ': 229,
'oo l': 16,
'o lo': 115,
' lon': 265,
'long': 252,
'ong ': 283,
'ng a': 729,
'g ag': 24,
' ago': 34,
'ago ': 20,
'go f': 3,
'o fo': 60,
'or h': 339,
'r he': 339,
'r to': 391,
'to h': 737,
'o ha': 366,
've m': 129,
'e mo': 309,
' mor': 619,
'more': 475,
're t': 476,
'e th': 1355,
' tha': 2278,
'than': 492,
'han ': 420,
'an i': 221,
'n in': 300,
'indi': 94,
'ndis': 17,
'isti': 57,
'stin': 113,
'tinc': 31,
'inct': 31,
'nct ': 9,
'ct r': 9,
' rem': 173,
'reme': 156,
'emem': 66,
'memb': 66,
'embr': 14,
'mbra': 14,
'bran': 15,
'ranc': 94,
'ance': 508,
'er c': 192,
'r ca': 48,
' car': 217,
'care': 87,
'ares': 35,
'esse': 120,
'sses': 43,
'ses ': 139,
'd he': 713,
'r pl': 42,
```

```
' pla': 248,
'plac': 138,
'lace': 141,
'ace ': 140,
'ce h': 65,
'ad b': 379,
'en s': 209,
'n su': 89,
' sup': 249,
'supp': 176,
'uppl': 22,
'ppli': 19,
'plie': 97,
'lied': 97,
'an e': 144,
'n ex': 70,
' exc': 214,
'exce': 138,
'xcel': 40,
'cell': 40,
'elle': 57,
'llen': 50,
'lent': 99,
'nt w': 110,
't wo': 220,
' wom': 153,
'woma': 136,
'oman': 142,
'man ': 334,
'n as': 143,
' as ': 1439,
'as g': 89,
's go': 92,
' gov': 12,
'gove': 12,
'over': 267,
'vern': 13,
'erne': 52,
'rnes': 56,
'ness': 498,
'ss w': 249,
' who': 499,
'who ': 298,
'ho h': 73,
'ad f': 46,
'd fa': 45,
' fal': 33,
'fall': 27,
'alle': 84,
'len ': 8,
'en l': 33,
'n li': 37,
'le s': 108,
'e sh': 443,
' sho': 495,
'shor': 78,
'hort': 78,
...})
```

Problem 1.2

In this sub-problem, your job is to write a function that takes a language model `lm`, and a text string `text`, and computes the perplexity of the language model on the text.

Hints:

- Your function can ignore the first three characters of the text. Thus, you can begin predicting the fourth character from the first three.
- Either extract the sequence of 4-character substrings, or make a call to `ngrams(text, n=4)` to get a set of 4-grams and their counts on the text.
- Compute the logarithm of probability of the text. If you compute the probability, you will get a very tiny number and numerical "underflow".
- Use the function `lm.log_probability` where `lm` is a instance of the class `language_model`. For example, `emma_lm.log_probability('emma')` will compute the logarithm of the probability that the character "a" follows the three characters "emm" using the language model computed on Jane Austen's "Emma".
- Once you have the logarithm of the probability of the entire text, you'll need to scale appropriately and then take the exponential, using `np.exp`.
- Work out the formula by "pencil and paper" before trying to write the function.

In [ ]:
```
def compute_perplexity(text, lm):
    temp_text = ngrams(text, n=4)
    perp = 0
    for gram in temp_text:
        perp += lm.log_probability(gram) * temp_text[gram]
    perp = np.exp(perp * (-1 / len(text)))
    return perp# replace with appropriate value
```

Problem 1.3

To test your implementation of the perplexity function, evaluate the followign cell. This computes the perplexity of the "Emma" language model on all four of the books: "Emma", "Pride and Prejudice", "Huckleberry Finn", and "Connecticut Yankee". For this problem, you will be graded on whether or not you get the correct four numbers for each of these perplexities.

Just run the following cell, which will evaluate the perplexities and print them out. No need to modify the code.

In [ ]:
```
hf_perp = compute_perplexity(huck_finn_text, emma_lm)
ct_perp = compute_perplexity(ct_yankee_text, emma_lm)
pp_perp = compute_perplexity(pride_and_prejudice_text, emma_lm)
em_perp = compute_perplexity(emma_text, emma_lm)

print("Perplexity on Huckleberry Finn: %.2f" % hf_perp)
print("Perplexity on Connecticut Yankee: %.2f" % ct_perp)
print("Perplexity on Pride and Prejudice: %.2f" % pp_perp)
print("Perplexity on Emma: %.2f" % em_perp)
```

```
Perplexity on Huckleberry Finn: 6.42
Perplexity on Connecticut Yankee: 6.18
Perplexity on Pride and Prejudice: 5.49
Perplexity on Emma: 5.11
```

Problem 1.4

Now, interpret your results above. Explain the meaning of perplexity for a character-based language model. Which book has the lowest perpexity? Why is this? Which book has the second smallest perplexity? Does this make sense? Explain.

If the perplexity for the model is k, the model predicts on average as if there were k equally likely words to follow. If a model assigns a high probability to the test set, it means that it is not surprised to see it (it's not perplexed by it), which means that it has a good understanding of how the language works. *Emma* has the lowest perplexity, because it is the training set of emma_lm model and the model assigns high probability on this book. *Pride and Prejudice* has the second smallest perplexity. It makes sense because these two books written by the same author. Hence, they have the same language style and perfence, and similar themes and contents, which are quite different from the other two books written by Mark Twain.

Problem 1.5

Next, mix it up by computing the perplexity of the "Huckleberry Finn" language model on each of the four books. Comment on your findings.

In [ ]:
```
hf_perp = compute_perplexity(huck_finn_text, huck_finn_lm)
ct_perp = compute_perplexity(ct_yankee_text, huck_finn_lm)
pp_perp = compute_perplexity(pride_and_prejudice_text, huck_finn_lm)
em_perp = compute_perplexity(emma_text, huck_finn_lm)
print("Perplexity on Huckleberry Finn: %.2f" % hf_perp)
print("Perplexity on Connecticut Yankee: %.2f" % ct_perp)
print("Perplexity on Pride and Prejudice: %.2f" % pp_perp)
print("Perplexity on Emma: %.2f" % em_perp)
```

```
Perplexity on Huckleberry Finn: 5.12
Perplexity on Connecticut Yankee: 6.17
Perplexity on Pride and Prejudice: 6.46
Perplexity on Emma: 6.44
```

Huck Finn has the lowest perplexity, because it is the training set of huck_finn_lm model and the model assigns high probability on this book. *Conn Yankee* has the second smallest perplexity. It makes sense because these two books written by the same author. Hence, they have the same language style and perfence, and similar themes and contents, which are quite different from the other two books written by Jane Austen.

Problem 1.6

Finally, in this problem you should explore the choice of the weights assigned to unigrams, bigrams, trigrams, and four-grams. Recall that to set the weights on the language model `lm` you can use a function call like `lm.set_weights([.25, .25, .25, .25])`

1. Try to find weights for the "Emma" model so that the perplexity of "Pride and Prejudice" is as small as possible. What weights do you find? Do these weights make sense to you?

1. Try to find weights for the "Emma" model so that the perplexity of "Huckleberry Finn" is as small as possible. What happens to the perplexity for "Pride and Prejudice"? Does this perplexity exceed that of "Huck Finn"? How do the weights you find compare to those you found above? Can you explain intuitively why they are different?

[your code and markdown here]

In [ ]:
```
total = 0
pp_final_perp = 100000
while total < 500:
    w4 = np.random.uniform(0.9, 1, 1)
    w3 = np.random.uniform(0, 1-w4, 1)
    w2 = np.random.uniform(0, 1-w4-w3, 1)
    w1 =  1-w4-w3-w2
    emma_lm = language_model(emma_text)
    emma_lm.set_weights([w1, w2, w3, w4])
    pp_perp = compute_perplexity(pride_and_prejudice_text, emma_lm)
    if pp_final_perp > pp_perp:
        pp_final_perp = pp_perp
        w1_final = w1
        w2_final = w2
        w3_final = w3
        w4_final = w4
    total = total + 1
```

In [ ]:
```
emma_lm = language_model(emma_text)
emma_lm.set_weights([w1_final, w2_final, w3_final, w4_final])
pp_perp = compute_perplexity(pride_and_prejudice_text, emma_lm)
print("Perplexity on Pride and Prejudice: %.2f" % pp_perp)
```

```
Perplexity on Pride and Prejudice: 4.59
```

The weights is [0.01, 0.01, 0.05, 0.93]. It makes sense because they are written by the same author and they have the same language style and perfence. Hence, more weights on 4-gram or 3-gram can improve the perplexity of Pride and Prejudice.

In [ ]:
```
total = 0
pp_final_perp = 100000
while total < 500:
    w4 = np.random.uniform(0.5, 1, 1)
    w3 = np.random.uniform(0, 1-w4, 1)
    w2 = np.random.uniform(0, 1-w4-w3, 1)
    w1 =  1-w4-w3-w2
    emma_lm = language_model(emma_text)
    emma_lm.set_weights([w1, w2, w3, w4])
    pp_perp = compute_perplexity(huck_finn_text, emma_lm)
    if pp_final_perp > pp_perp:
        pp_final_perp = pp_perp
        w1_final = w1
        w2_final = w2
        w3_final = w3
        w4_final = w4
    total = total + 1
```

In [ ]:
```
emma_lm = language_model(emma_text)
emma_lm.set_weights([w1_final, w2_final, w3_final, w4_final])
huck_perp = compute_perplexity(huck_finn_text, emma_lm)
print("Perplexity on Huck Finn: %.2f" % huck_perp)
pp_perp = compute_perplexity(pride_and_prejudice_text, emma_lm)
print("Perplexity on P&P: %.2f" % pp_perp)
print([w1_final, w2_final, w3_final, w4_final])
```

```
Perplexity on Huck Finn: 6.02
Perplexity on P&P: 4.69
[array([0.01584392]), array([0.06461364]), array([0.15300654]), array([0.7665359])]
```

The perplexity on P&P increases, but this perplexity doesn't exceed that of "Huck Finn". The weights on 4-gram and 3-gram decrease and the weights on unigram and bigram increase. To decrease the perplexity of Emma model on Huck Finn, we should increase the prior probability intuitively because these two books are very different. However, Emma and P&P are written by the same author and they have the same language style and perfence. Hence, more weights on 4-gram or 3-gram can improve the perplexity of Pride and Prejudice.

Problem 1.7 (Extra credit: 2 points)

Choose two books from the Gutenberg collection that are by the same author (different from Twain and Austen). Build a language model on one of the books, and test, by evaluating perplexity, on all of the other books, five books in all. Where does the second book of the new author rank? Do the result make sense? Comment on your findings.

In [ ]:

```
hamlet_url = 'https://www.gutenberg.org/files/27761/27761-0.txt'
hamlet_text_raw = read_url(hamlet_url)
hamlet_text = process_text(hamlet_text_raw)

king_lear_url = "https://www.gutenberg.org/files/1532/1532-0.txt"
king_lear_text_raw = read_url(king_lear_url)
king_lear_text = process_text(king_lear_text_raw)

hamlet_lm = language_model(hamlet_text)
king_lear_lm = language_model(king_lear_text)

hf_perp = compute_perplexity(huck_finn_text, hamlet_lm)
ct_perp = compute_perplexity(ct_yankee_text, hamlet_lm)
pp_perp = compute_perplexity(pride_and_prejudice_text, hamlet_lm)
em_perp = compute_perplexity(emma_text, hamlet_lm)
kl_perp = compute_perplexity(king_lear_text, hamlet_lm)
print("Perplexity on Huckleberry Finn: %.2f" % hf_perp)
print("Perplexity on Connecticut Yankee: %.2f" % ct_perp)
print("Perplexity on Pride and Prejudice: %.2f" % pp_perp)
print("Perplexity on Emma: %.2f" % em_perp)
print("Perplexity on King Lear: %.2f" % kl_perp)
```

```
Perplexity on Huckleberry Finn: 6.72
Perplexity on Connecticut Yankee: 6.43
Perplexity on Pride and Prejudice: 6.26
Perplexity on Emma: 6.41
Perplexity on King Lear: 6.38
```

King Lear ranks the second among the five books. The result makes sense because the two books written by Mark Twain rank the last and there is no big gap between the perplexity of the other three books.

Problem 2: Word embedding experiments (15 points)

In this problem you will run experiments on word embeddings using two different algorithms or configurations: (1) word2vec embeddings trained on the "text8" Wikipedia corpus (2) GloVe

embeddings pre-trained on a much larger corpus.

The text8 data are described here: http://mattmahoney.net/dc/textdata.html. The text8 file is a 100MB excerpt of Wikipedia. This small dataset is sufficient for our exploratory purposes, but note that it is far too small for any real application. In the next few parts of this problem, you will construct word embeddings from the Wikipedia data.

`word2Vec` is a popular word embedding method. The following code will construct 100 dimensional embeddings on the text8 data.

In [ ]:
```
import gensim
from gensim.models import word2vec
import logging

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
                    level=logging.INFO)
sentences = word2vec.Text8Corpus('https://raw.githubusercontent.com/YData123/sds265-fa2
model = word2vec.Word2Vec(sentences, vector_size=100, window=10, min_count=10)
```

```
2021-11-11 23:40:40,912 : INFO : collecting all words and their counts
2021-11-11 23:40:41,115 : INFO : PROGRESS: at sentence #0, processed 0 words, keeping 0
word types
2021-11-11 23:40:53,912 : INFO : collected 253854 word types from a corpus of 17005207 r
aw words and 1701 sentences
2021-11-11 23:40:53,913 : INFO : Creating a fresh vocabulary
2021-11-11 23:40:54,245 : INFO : Word2Vec lifecycle event {'msg': 'effective_min_count=1
0 retains 47134 unique words (18.56736549355141%% of original 253854, drops 206720)', 'd
atetime': '2021-11-11T23:40:54.245988', 'gensim': '4.1.2', 'python': '3.8.8 (default, Ap
r 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP
0', 'event': 'prepare_vocab'}
2021-11-11 23:40:54,246 : INFO : Word2Vec lifecycle event {'msg': 'effective_min_count=1
0 leaves 16561031 word corpus (97.38800004022298%% of original 17005207, drops 444176)',
'datetime': '2021-11-11T23:40:54.246948', 'gensim': '4.1.2', 'python': '3.8.8 (default,
Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-
SP0', 'event': 'prepare_vocab'}
2021-11-11 23:40:54,630 : INFO : deleting the raw counts dictionary of 253854 items
2021-11-11 23:40:54,640 : INFO : sample=0.001 downsamples 38 most-common words
2021-11-11 23:40:54,642 : INFO : Word2Vec lifecycle event {'msg': 'downsampling leaves e
stimated 12333563.370024087 word corpus (74.5%% of prior 16561031)', 'datetime': '2021-1
1-11T23:40:54.642903', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:
03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0', 'event': 'pre
pare_vocab'}
2021-11-11 23:40:55,253 : INFO : estimated required memory for 47134 words and 100 dimen
sions: 61274200 bytes
2021-11-11 23:40:55,254 : INFO : resetting layer weights
2021-11-11 23:40:55,288 : INFO : Word2Vec lifecycle event {'update': False, 'trim_rule':
'None', 'datetime': '2021-11-11T23:40:55.288168', 'gensim': '4.1.2', 'python': '3.8.8 (d
efault, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.
0.22494-SP0', 'event': 'build_vocab'}
2021-11-11 23:40:55,289 : INFO : Word2Vec lifecycle event {'msg': 'training model with 3
workers on 47134 vocabulary and 100 features, using sg=0 hs=0 sample=0.001 negative=5 wi
ndow=10 shrink_windows=True', 'datetime': '2021-11-11T23:40:55.289165', 'gensim': '4.1.
2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'pla
tform': 'Windows-10-10.0.22494-SP0', 'event': 'train'}
2021-11-11 23:40:56,297 : INFO : EPOCH 1 - PROGRESS: at 4.70% examples, 578602 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:40:57,308 : INFO : EPOCH 1 - PROGRESS: at 10.05% examples, 610742 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:40:58,311 : INFO : EPOCH 1 - PROGRESS: at 14.99% examples, 609108 words/s,
in_qsize 5, out_qsize 0
```

```
2021-11-11 23:40:59,311 : INFO : EPOCH 1 - PROGRESS: at 19.87% examples, 607138 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:00,320 : INFO : EPOCH 1 - PROGRESS: at 24.69% examples, 604457 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:01,328 : INFO : EPOCH 1 - PROGRESS: at 29.69% examples, 607340 words/s,
in_qsize 6, out_qsize 1
2021-11-11 23:41:02,331 : INFO : EPOCH 1 - PROGRESS: at 34.74% examples, 610268 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:03,347 : INFO : EPOCH 1 - PROGRESS: at 39.56% examples, 607533 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:04,362 : INFO : EPOCH 1 - PROGRESS: at 44.33% examples, 604510 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:05,370 : INFO : EPOCH 1 - PROGRESS: at 49.85% examples, 611997 words/s,
in_qsize 5, out_qsize 1
2021-11-11 23:41:06,374 : INFO : EPOCH 1 - PROGRESS: at 55.56% examples, 620413 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:07,374 : INFO : EPOCH 1 - PROGRESS: at 61.43% examples, 629227 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:08,385 : INFO : EPOCH 1 - PROGRESS: at 66.84% examples, 631608 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:09,394 : INFO : EPOCH 1 - PROGRESS: at 72.25% examples, 634047 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:10,404 : INFO : EPOCH 1 - PROGRESS: at 77.78% examples, 635718 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:11,405 : INFO : EPOCH 1 - PROGRESS: at 84.01% examples, 643847 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:12,413 : INFO : EPOCH 1 - PROGRESS: at 89.07% examples, 642292 words/s,
in_qsize 3, out_qsize 0
2021-11-11 23:41:13,418 : INFO : EPOCH 1 - PROGRESS: at 93.65% examples, 637684 words/s,
in_qsize 4, out_qsize 0
2021-11-11 23:41:14,425 : INFO : EPOCH 1 - PROGRESS: at 99.24% examples, 639927 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:14,573 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:41:14,577 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:41:14,587 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:41:14,589 : INFO : EPOCH - 1 : training on 17005207 raw words (12334407 ef
fective words) took 19.3s, 639302 effective words/s
2021-11-11 23:41:15,597 : INFO : EPOCH 2 - PROGRESS: at 4.35% examples, 535422 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:16,603 : INFO : EPOCH 2 - PROGRESS: at 9.52% examples, 579799 words/s,
in_qsize 4, out_qsize 0
2021-11-11 23:41:17,607 : INFO : EPOCH 2 - PROGRESS: at 14.99% examples, 609430 words/s,
in_qsize 4, out_qsize 0
2021-11-11 23:41:18,626 : INFO : EPOCH 2 - PROGRESS: at 20.69% examples, 629945 words/s,
in_qsize 1, out_qsize 1
2021-11-11 23:41:19,647 : INFO : EPOCH 2 - PROGRESS: at 25.87% examples, 630302 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:20,654 : INFO : EPOCH 2 - PROGRESS: at 31.45% examples, 640581 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:21,655 : INFO : EPOCH 2 - PROGRESS: at 37.33% examples, 653424 words/s,
in_qsize 5, out_qsize 1
2021-11-11 23:41:22,656 : INFO : EPOCH 2 - PROGRESS: at 42.97% examples, 659146 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:23,666 : INFO : EPOCH 2 - PROGRESS: at 48.68% examples, 663474 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:24,667 : INFO : EPOCH 2 - PROGRESS: at 54.56% examples, 669991 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:25,687 : INFO : EPOCH 2 - PROGRESS: at 60.02% examples, 669396 words/s,
in_qsize 5, out_qsize 1
2021-11-11 23:41:26,690 : INFO : EPOCH 2 - PROGRESS: at 65.20% examples, 666722 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:27,732 : INFO : EPOCH 2 - PROGRESS: at 68.55% examples, 645433 words/s,
```

```
in_qsize 0, out_qsize 0
2021-11-11 23:41:28,739 : INFO : EPOCH 2 - PROGRESS: at 71.90% examples, 628825 words/s,
in_qsize 1, out_qsize 0
2021-11-11 23:41:29,754 : INFO : EPOCH 2 - PROGRESS: at 77.13% examples, 628443 words/s,
in_qsize 2, out_qsize 0
2021-11-11 23:41:30,757 : INFO : EPOCH 2 - PROGRESS: at 81.54% examples, 622854 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:31,771 : INFO : EPOCH 2 - PROGRESS: at 86.18% examples, 619386 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:32,783 : INFO : EPOCH 2 - PROGRESS: at 91.95% examples, 623848 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:33,791 : INFO : EPOCH 2 - PROGRESS: at 96.59% examples, 620803 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:41:34,550 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:41:34,551 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:41:34,554 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:41:34,556 : INFO : EPOCH - 2 : training on 17005207 raw words (12334811 ef
fective words) took 20.0s, 617865 effective words/s
2021-11-11 23:41:35,573 : INFO : EPOCH 3 - PROGRESS: at 4.82% examples, 588017 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:36,577 : INFO : EPOCH 3 - PROGRESS: at 10.58% examples, 642059 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:37,580 : INFO : EPOCH 3 - PROGRESS: at 16.40% examples, 666202 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:38,594 : INFO : EPOCH 3 - PROGRESS: at 21.22% examples, 646117 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:39,607 : INFO : EPOCH 3 - PROGRESS: at 26.16% examples, 638282 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:40,608 : INFO : EPOCH 3 - PROGRESS: at 30.92% examples, 631036 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:41,612 : INFO : EPOCH 3 - PROGRESS: at 36.21% examples, 634875 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:42,618 : INFO : EPOCH 3 - PROGRESS: at 42.33% examples, 649408 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:43,624 : INFO : EPOCH 3 - PROGRESS: at 47.85% examples, 652888 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:44,628 : INFO : EPOCH 3 - PROGRESS: at 52.97% examples, 650680 words/s,
in_qsize 4, out_qsize 1
2021-11-11 23:41:45,628 : INFO : EPOCH 3 - PROGRESS: at 58.61% examples, 655043 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:46,644 : INFO : EPOCH 3 - PROGRESS: at 64.67% examples, 661873 words/s,
in_qsize 5, out_qsize 1
2021-11-11 23:41:47,652 : INFO : EPOCH 3 - PROGRESS: at 69.90% examples, 660434 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:48,658 : INFO : EPOCH 3 - PROGRESS: at 75.49% examples, 661865 words/s,
in_qsize 4, out_qsize 2
2021-11-11 23:41:49,667 : INFO : EPOCH 3 - PROGRESS: at 81.48% examples, 665880 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:50,681 : INFO : EPOCH 3 - PROGRESS: at 87.48% examples, 669796 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:51,684 : INFO : EPOCH 3 - PROGRESS: at 93.65% examples, 674834 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:52,689 : INFO : EPOCH 3 - PROGRESS: at 99.53% examples, 677291 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:41:52,738 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:41:52,744 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:41:52,747 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:41:52,749 : INFO : EPOCH - 3 : training on 17005207 raw words (12333211 ef
fective words) took 18.2s, 678092 effective words/s
```

```
2021-11-11 23:41:53,768 : INFO : EPOCH 4 - PROGRESS: at 3.53% examples, 431206 words/s,
in_qsize 1, out_qsize 1
2021-11-11 23:41:54,770 : INFO : EPOCH 4 - PROGRESS: at 9.52% examples, 578377 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:55,773 : INFO : EPOCH 4 - PROGRESS: at 15.23% examples, 617650 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:56,782 : INFO : EPOCH 4 - PROGRESS: at 20.81% examples, 634316 words/s,
in_qsize 2, out_qsize 0
2021-11-11 23:41:57,797 : INFO : EPOCH 4 - PROGRESS: at 26.87% examples, 656379 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:58,807 : INFO : EPOCH 4 - PROGRESS: at 32.92% examples, 671874 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:41:59,829 : INFO : EPOCH 4 - PROGRESS: at 38.74% examples, 676988 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:00,844 : INFO : EPOCH 4 - PROGRESS: at 43.03% examples, 657830 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:01,849 : INFO : EPOCH 4 - PROGRESS: at 47.44% examples, 645171 words/s,
in_qsize 1, out_qsize 0
2021-11-11 23:42:02,851 : INFO : EPOCH 4 - PROGRESS: at 53.44% examples, 654547 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:03,864 : INFO : EPOCH 4 - PROGRESS: at 59.55% examples, 663094 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:42:04,877 : INFO : EPOCH 4 - PROGRESS: at 65.37% examples, 667033 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:05,883 : INFO : EPOCH 4 - PROGRESS: at 70.72% examples, 666370 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:06,886 : INFO : EPOCH 4 - PROGRESS: at 75.84% examples, 663215 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:07,892 : INFO : EPOCH 4 - PROGRESS: at 79.89% examples, 651672 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:08,902 : INFO : EPOCH 4 - PROGRESS: at 85.60% examples, 654401 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:09,905 : INFO : EPOCH 4 - PROGRESS: at 90.65% examples, 652532 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:10,914 : INFO : EPOCH 4 - PROGRESS: at 96.18% examples, 653641 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:11,661 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:42:11,666 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:42:11,672 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:42:11,674 : INFO : EPOCH - 4 : training on 17005207 raw words (12334424 ef
fective words) took 18.9s, 651960 effective words/s
2021-11-11 23:42:12,688 : INFO : EPOCH 5 - PROGRESS: at 4.94% examples, 604001 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:42:13,703 : INFO : EPOCH 5 - PROGRESS: at 10.29% examples, 621740 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:14,732 : INFO : EPOCH 5 - PROGRESS: at 15.70% examples, 629906 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:15,840 : INFO : EPOCH 5 - PROGRESS: at 17.81% examples, 525186 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:16,854 : INFO : EPOCH 5 - PROGRESS: at 19.81% examples, 469710 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:17,910 : INFO : EPOCH 5 - PROGRESS: at 23.34% examples, 460624 words/s,
in_qsize 1, out_qsize 0
2021-11-11 23:42:18,910 : INFO : EPOCH 5 - PROGRESS: at 28.57% examples, 487138 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:19,940 : INFO : EPOCH 5 - PROGRESS: at 32.57% examples, 486833 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:20,952 : INFO : EPOCH 5 - PROGRESS: at 37.51% examples, 500035 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:21,956 : INFO : EPOCH 5 - PROGRESS: at 39.51% examples, 475204 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:42:22,968 : INFO : EPOCH 5 - PROGRESS: at 43.21% examples, 473287 words/s,
```

```
in_qsize 2, out_qsize 0
2021-11-11 23:42:23,985 : INFO : EPOCH 5 - PROGRESS: at 48.38% examples, 486177 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:24,989 : INFO : EPOCH 5 - PROGRESS: at 53.20% examples, 494357 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:25,999 : INFO : EPOCH 5 - PROGRESS: at 58.20% examples, 502781 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:27,004 : INFO : EPOCH 5 - PROGRESS: at 63.73% examples, 514469 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:42:28,009 : INFO : EPOCH 5 - PROGRESS: at 68.25% examples, 517056 words/s,
in_qsize 5, out_qsize 0
2021-11-11 23:42:29,010 : INFO : EPOCH 5 - PROGRESS: at 73.43% examples, 524261 words/s,
in_qsize 4, out_qsize 0
2021-11-11 23:42:30,012 : INFO : EPOCH 5 - PROGRESS: at 79.37% examples, 534549 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:42:31,021 : INFO : EPOCH 5 - PROGRESS: at 85.42% examples, 545180 words/s,
in_qsize 6, out_qsize 0
2021-11-11 23:42:32,027 : INFO : EPOCH 5 - PROGRESS: at 91.42% examples, 554499 words/s,
in_qsize 2, out_qsize 0
2021-11-11 23:42:33,039 : INFO : EPOCH 5 - PROGRESS: at 97.53% examples, 563272 words/s,
in_qsize 5, out_qsize 1
2021-11-11 23:42:33,409 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:42:33,419 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:42:33,424 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:42:33,425 : INFO : EPOCH - 5 : training on 17005207 raw words (12333907 ef
fective words) took 21.7s, 567172 effective words/s
2021-11-11 23:42:33,427 : INFO : Word2Vec lifecycle event {'msg': 'training on 85026035
raw words (61670760 effective words) took 98.1s, 628416 effective words/s', 'datetime':
'2021-11-11T23:42:33.427632', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 13 2021,
15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0', 'even
t': 'train'}
2021-11-11 23:42:33,429 : INFO : Word2Vec lifecycle event {'params': 'Word2Vec(vocab=471
34, vector_size=100, alpha=0.025)', 'datetime': '2021-11-11T23:42:33.429626', 'gensim':
'4.1.2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]',
'platform': 'Windows-10-10.0.22494-SP0', 'event': 'created'}
```

The dictionary `model.wv`, keyed by words (as strings), has values which are the word embeddings (as numpy arrays).

We will also work with pre-trained GloVe embeddings. These embeddings were trained on a large corpus containing 6 billion tokens. You can load these embedding vectors using this code:

In [ ]:
```
import gensim.downloader as gdl
glove = gdl.load("glove-wiki-gigaword-100")
```

```
2021-11-11 23:42:37,702 : INFO : loading projection weights from C:\Users\Ji Qi/gensim-d
ata\glove-wiki-gigaword-100\glove-wiki-gigaword-100.gz
2021-11-11 23:43:34,943 : INFO : KeyedVectors lifecycle event {'msg': 'loaded (400000, 1
00) matrix of type float32 from C:\\Users\\Ji Qi/gensim-data\\glove-wiki-gigaword-100\\g
love-wiki-gigaword-100.gz', 'binary': False, 'encoding': 'utf8', 'datetime': '2021-11-11
T23:43:34.943199', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:03)
[MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0', 'event': 'load_wo
rd2vec_format'}
```

Here is a sample evaluation: puppy is to dog as what is to kitten?

In [ ]:
```
glove.most_similar(positive = ['dog', 'kitten'], negative = ['puppy'])[0]
```

```
2021-11-08 14:56:51,891 : INFO : precomputing L2-norms of word weight vectors
```

Out[ ]:
```
('cat', 0.6935379505157471)
```

Experiments

Conduct the following experiments with **both sets of embeddings**: the local word2vec embeddings, and the pre-trained GloVe embeddings. Based on the available memory on your computer, you may need to perform the experiments for each set of embeddings in a fresh Python session. Comment on the qualitative differences in the results for each of the embeddings.

2.1 Find the closest words

For each of the following words, find the 5 closest words in the embedding space, and report your results:

```
yale, physics, dessert, einstein, algebra, fish
```

Here, "closest" means in terms of cosine similarity. See the gensim documentation; you might want to use the most similar function, or a related function. Choose five other query words yourself, and for each of them show the closest words in the embedding space. Comment on your findings.

In [ ]:
```
# your code and markdown here
print(
glove.most_similar('yale', topn=5),
glove.most_similar('physics', topn=5),
glove.most_similar('dessert', topn=5),
glove.most_similar('einstein', topn=5),
glove.most_similar('algebra', topn=5),
glove.most_similar('fish', topn=5))
```

```
[('harvard', 0.9161344766616821), ('princeton', 0.867539644241333), ('university', 0.811
3802671432495), ('cornell', 0.8014455437660217), ('stanford', 0.787754476070404)] [('che
mistry', 0.8498000502586365), ('mathematics', 0.834094762802124), ('science', 0.79146987
19978333), ('biology', 0.7894973158836365), ('theoretical', 0.7342938780784607)] [('dess
erts', 0.8192153573036194), ('cake', 0.7649705410003662), ('delicious', 0.76006871461868
29), ('chocolate', 0.738676905632019), ('cakes', 0.7226456999778748)] [('relativity', 0.
6908349394798279), ('freud', 0.6417257785797119), ('physics', 0.6145339012145996), ('boh
r', 0.6144247055053711), ('theory', 0.6042598485946655)] [('algebras', 0.753978371620178
2), ('geometry', 0.6790961027145386), ('algebraic', 0.6474162340164185), ('boolean', 0.6
418250799179077), ('commutative', 0.6322143077850342)] [('shrimp', 0.7793381214141846),
('salmon', 0.7608143091201782), ('tuna', 0.7485246658325195), ('meat', 0.711925506591796
9), ('seafood', 0.6940563321113586)]
```

In [ ]:
```
print(
    model.wv.most_similar('yale', topn=5),
model.wv.most_similar('physics', topn=5),
model.wv.most_similar('dessert', topn=5),
model.wv.most_similar('einstein', topn=5),
model.wv.most_similar('algebra', topn=5),
model.wv.most_similar('fish', topn=5),
)
```

```
[('harvard', 0.8473474979400635), ('princeton', 0.8101245760917664), ('cornell', 0.80319
9827671051), ('rutgers', 0.7824962139129639), ('stanford', 0.7587567567825317)] [('elect
rodynamics', 0.7818374633789062), ('mechanics', 0.7781468033790588), ('theoretical', 0.7
```

```
464304566383362), ('astronomy', 0.7415562272071838), ('electromagnetism', 0.739653885364
5325)] [('stew', 0.8733224868774414), ('pork', 0.8663265109062195), ('grilled', 0.856291
7709350586), ('steamed', 0.8535162210464478), ('tomato', 0.8523162603378296)] [('fermi',
0.8070806264877319), ('dinger', 0.7915767431259155), ('planck', 0.7912207245826721), ('h
eisenberg', 0.780948281288147), ('maxwell', 0.7670904994010925)] [('algebraic', 0.869365
8113479614), ('topology', 0.838798999786377), ('associative', 0.8241924047470093), ('boo
lean', 0.8075669407844543), ('commutative', 0.7994086742401123)] [('foxes', 0.8467469811
439514), ('cattle', 0.8179076313972473), ('shrimp', 0.8156296610832214), ('whales', 0.80
89657425880432), ('sheep', 0.806541919708252)]
```

most of the top-5 closest words are similar words both in the same category and in the part of speech. For example, "yale", "harvard",and "princeton" are all ivy league universities and nouns; "physics", "mathematics", and "chemistry" are all natural science subjects and nouns. The other closest words of the word show some fundemental properties of the word. For example, "yale" is a "university"; "physics" is "theoretical" "science"; "fish" is a kind of "seafood".

In [ ]:

```
print(
    model.wv.most_similar('harvard', topn=5),
model.wv.most_similar('mathematics', topn=5),
model.wv.most_similar('planck', topn=5),
model.wv.most_similar('geometry', topn=5),
model.wv.most_similar('shrimp', topn=5),
)
```

```
[('yale', 0.847323477268219), ('faculty', 0.7951253652572632), ('cornell', 0.78379362821
57898), ('graduate', 0.7583586573600769), ('princeton', 0.7446129322052002)] [('mathemat
ical', 0.8498882055282593), ('geometry', 0.738296627998352), ('algebra', 0.7314383387565
613), ('calculus', 0.7209789156913757), ('logic', 0.7008056640625)] [('dinger', 0.789884
4480514526), ('heisenberg', 0.7720593214035034), ('einstein', 0.7584236860275269), ('fer
mi', 0.75608891248703), ('physics', 0.7519253492355347)] [('geometric', 0.82178103923797
61), ('trigonometry', 0.8076184988021851), ('calculus', 0.803760290145874), ('algebrai
c', 0.8016397953033447), ('topology', 0.7947611212730408)] [('citrus', 0.909737467765808
1), ('seafood', 0.8997024893760681), ('beef', 0.8798601627349854), ('pork', 0.8787608742
713928), ('potatoes', 0.8779131174087524)]
```

In [ ]:

```
print(
    glove.most_similar('harvard', topn=5),
glove.most_similar('mathematics', topn=5),
glove.most_similar('planck', topn=5),
glove.most_similar('geometry', topn=5),
glove.most_similar('shrimp', topn=5),
)
```

```
[('yale', 0.9161344766616821), ('princeton', 0.8444410562515259), ('graduate', 0.8181679
248809814), ('university', 0.8156033754348755), ('stanford', 0.810197114944458)] [('phys
ics', 0.834094762802124), ('mathematical', 0.7631471157073975), ('science', 0.7607673406
600952), ('biology', 0.7574610114097595), ('chemistry', 0.7529067397117615)] [('particl
e', 0.5575568675994873), ('astrophysics', 0.5189620852470398), ('cyclotron', 0.511197388
1721497), ('boltzmann', 0.5060967206954956), ('entropy', 0.505074679851532)] [('algebrai
c', 0.757987380027771), ('differential', 0.7062228918075562), ('trigonometry', 0.7045199
275016785), ('geometric', 0.6944104433059692), ('analytic', 0.6942931413650513)] [('lobs
ter', 0.8110523819923401), ('prawns', 0.7906647324562073), ('crab', 0.7873328328132629),
('fish', 0.7793381214141846), ('scallops', 0.7699012756347656)]
```

We can find that the GloVe model is doing a better job than word2vec model, since GloVe model was trained on a much larger dataset based on Wikipedia. For example, the word "fish", 5 closest words by using word2vec model are: 'sheep', 'shrimp', 'cattle', 'foxes' and 'seafood'. However, 5 closest words by using GloVe model are: 'shrimp', 'salmon', 'tuna', 'meat' and 'seafood'. It is obvious

that 'shrimp', 'salmon', 'tuna', 'meat' and 'seafood' are have something to do with 'fish'; however, 'cattle' and 'foxes' selected by word2vec doesn't have much thing to do with 'fish'.

2.2 Complete analogies

A surprising consequence of some word embedding methods is that they can be used to resolve analogies, like

```
france :  paris ::  england :  ?
```

You can "solve" this analogy by computing the nearest embedding vector to v where, $v = v_{paris} - v_{france} + v_{england}$.

Solve the following analogies with both sets of word embeddings and report your results:

```
france :  paris ::  england :  ?
france :  paris ::  germany :  ?
  queen :  woman ::  king :  ?
```

Choose five other analogies yourself, and report the results.

In [ ]:
```
glove.most_similar(positive=['paris', 'england'], negative=['france'], topn=10)
```

Out[ ]:
```
[('london', 0.8123227953910828),
 ('melbourne', 0.7388997077941895),
 ('sydney', 0.714728832244873),
 ('birmingham', 0.696514904499054),
 ('perth', 0.6886942386627197),
 ('cambridge', 0.6872051954269409),
 ('oxford', 0.6779206991195679),
 ('leeds', 0.6770237684249878),
 ('dublin', 0.6747875213623047),
 ('manchester', 0.6740053296089172)]
```

In [ ]:
```
model.wv.most_similar(positive=['paris', 'england'], negative=['france'], topn=10)
```

Out[ ]:
```
[('edinburgh', 0.7130708694458008),
 ('london', 0.676858127117157),
 ('glasgow', 0.6337019801139832),
 ('abbey', 0.6323848366737366),
 ('westminster', 0.6295573115348816),
 ('concord', 0.6265250444412231),
 ('worcester', 0.608669638633728),
 ('birmingham', 0.5902668237686157),
 ('surrey', 0.5862783193588257),
 ('nottingham', 0.5813407897949219)]
```

In [ ]:
```
glove.most_similar(positive=['paris', 'germany'], negative=['france'], topn=10)
```

Out[ ]:
```
[('berlin', 0.8846380710601807),
 ('frankfurt', 0.7985543608665466),
 ('vienna', 0.7675994038581848),
```

```
 ('munich', 0.7542588114738464),
 ('hamburg', 0.7182371616363525),
 ('bonn', 0.6890878081321716),
 ('prague', 0.6842440962791443),
 ('cologne', 0.6762093305587769),
 ('zurich', 0.6653268933296204),
 ('leipzig', 0.6619253754615784)]
```

In [ ]:
```
model.wv.most_similar(positive=['paris', 'germany'], negative=['france'], topn=10)
```

Out[ ]:
```
[('berlin', 0.8328025937080383),
 ('munich', 0.7696191668510437),
 ('vienna', 0.7063215374946594),
 ('leipzig', 0.6916210055351257),
 ('frankfurt', 0.6666767001152039),
 ('freiburg', 0.6657721996307373),
 ('speer', 0.6483389735221863),
 ('heidelberg', 0.6460251212120056),
 ('dresden', 0.6415418982505798),
 ('bonn', 0.6391190886497498)]
```

In [ ]:
```
glove.most_similar(positive=['king', 'woman'], negative=['queen'], topn=10)
```

Out[ ]:
```
[('man', 0.790776789188385),
 ('father', 0.7224542498588562),
 ('son', 0.7012037038803101),
 ('boy', 0.6988654732704163),
 ('another', 0.6853840351104736),
 ('person', 0.6754911541938782),
 ('who', 0.6728038191795349),
 ('brother', 0.6645877957344055),
 ('mother', 0.6613163948059082),
 ('one', 0.6554158329963684)]
```

In [ ]:
```
model.wv.most_similar(positive=['king', 'woman'], negative=['queen'], topn=10)
```

Out[ ]:
```
[('man', 0.6344357132911682),
 ('jew', 0.6257914304733276),
 ('son', 0.5733802914619446),
 ('child', 0.5629153251647949),
 ('prostitute', 0.5578898787498474),
 ('prophet', 0.5521579384803772),
 ('jacob', 0.5425581336021423),
 ('minotaur', 0.5401904582977295),
 ('descendant', 0.5394933819770813),
 ('stranger', 0.5387341976165771)]
```

In [ ]:
```
glove.most_similar(positive=['harvard', 'cambridge'], negative=['yale'], topn=10)
```

Out[ ]:
```
[('oxford', 0.8065130710601807),
 ('trinity', 0.6591581702232361),
 ('college', 0.6458384394645691),
 ('berkeley', 0.6405155658721924),
 ('university', 0.633739709854126),
 ('research', 0.6237657070159912),
 ('massachusetts', 0.6177628636360168),
 ('exeter', 0.6092889308929443),
 ('study', 0.6052048206329346),
 ('studies', 0.601348876953125)]
```

In [ ]:
```
model.wv.most_similar(positive=['harvard', 'cambridge'], negative=['yale'], topn=10)
```

Out[ ]:
```
[('oxford', 0.7378795146942139),
 ('princeton', 0.6621752381324768),
 ('oup', 0.6253624558448792),
 ('phd', 0.6068180203437805),
 ('humanities', 0.5995826721191406),
 ('univ', 0.5791908502578735),
 ('lectures', 0.5747931599617004),
 ('stanford', 0.5738669037818909),
 ('marischal', 0.5729469060897827),
 ('academic', 0.5621044635772705)]
```

In [ ]:
```
glove.most_similar(positive=['trump', 'italy'], negative=['usa'], topn=10)
```

Out[ ]:
```
[('berlusconi', 0.6134708523750305),
 ('silvio', 0.5475398898124695),
 ('dini', 0.5409714579582214),
 ('scalfaro', 0.5239711403846741),
 ('stefano', 0.4917651414871216),
 ('lamberto', 0.490522176027298),
 ('prodi', 0.4878629446029663),
 ('domenico', 0.4548022150993347),
 ('juventus', 0.447188138961792),
 ('parma', 0.44440269470214844)]
```

In [ ]:
```
model.wv.most_similar(positive=['trump', 'italy'], negative=['usa'], topn=10)
```

Out[ ]:
```
[('caligula', 0.528862714767456),
 ('tiberius', 0.5222833156585693),
 ('claudius', 0.522044837474823),
 ('soga', 0.5200117230415344),
 ('caesar', 0.5144539475440979),
 ('octavian', 0.5117034912109375),
 ('heraclius', 0.5025618672370911),
 ('usurper', 0.5015037059783936),
 ('agrippina', 0.49293211102485657),
 ('lepidus', 0.48556119203567505)]
```

In [ ]:
```
glove.most_similar(positive=['trump', 'japan'], negative=['usa'], topn=10)
```

Out[ ]:
```
[('abe', 0.5799363851547241),
 ('hashimoto', 0.5672397017478943),
 ('junichiro', 0.5445129871368408),
 ('koizumi', 0.5363299250602722),
 ('fukuda', 0.5132630467414856),
 ('yoshiro', 0.5123278498649597),
 ('hatoyama', 0.5096796751022339),
 ('shinzo', 0.5081068277359009),
 ('obuchi', 0.500024676322937),
 ('ivanka', 0.49894893169403076)]
```

In [ ]:
```
model.wv.most_similar(positive=['trump', 'japan'], negative=['usa'], topn=10)
```

Out[ ]:
```
[('shogun', 0.5167736411094666),
 ('japanese', 0.48669618368148804),
 ('melee', 0.4746353328227997),
 ('character', 0.4620440602302551),
 ('token', 0.4617423415184021),
```

```
 ('emperor', 0.45105770230293274),
 ('blackjack', 0.4462856352329254),
 ('swaps', 0.44136255979537964),
 ('goldeneye', 0.4401916265487671),
 ('vlad', 0.43126559257507324)]
```

In [ ]:
```
glove.most_similar(positive=['usa', 'france'], negative=['canada'], topn=10)
```

Out[ ]:
```
[('germany', 0.5512969493865967),
 ('french', 0.5501134395599365),
 ('spain', 0.5493493676185608),
 ('paris', 0.5301828384399414),
 ('italy', 0.517480194568634),
 ('de', 0.511626660823822),
 ('america', 0.5081153512001038),
 ('v', 0.5019479393959045),
 ('lyon', 0.4998127222061157),
 ('belgium', 0.49166321754455566)]
```

In [ ]:
```
model.wv.most_similar(positive=['usa', 'france'], negative=['canada'], topn=10)
```

Out[ ]:
```
[('italy', 0.6130130290985107),
 ('paris', 0.5933648347854614),
 ('switzerland', 0.5915034413337708),
 ('italiana', 0.5670886635780334),
 ('montreux', 0.5614999532699585),
 ('munich', 0.5507471561431885),
 ('commune', 0.5470203757286072),
 ('belgium', 0.5303712487220764),
 ('alsace', 0.5283372402191162),
 ('ukraine', 0.5259213447570801)]
```

In general, based on the above results, we can see that the GloVe model gave more reasonable answers. For example, for France to Paris , it makes sense to pick London for GloVe but not Edinghburg which is not an England city in Word2Vec.

2.3 Visualize embeddings

Use the t-SNE dimensionality reduction technique to visualize the embeddings in two dimensions. The sample code below will perform the t-SNE method on a subset of the vocabulary. You can start with this code and modify it to construct visualizations of the embeddings, or start with your own version of t-SNE.

The code provided shows the relative positions of the words conservative, liberal, elephant, and donkey based on GloVe embeddings. For your local embeddings, you may use similar code to visualize the embeddings.

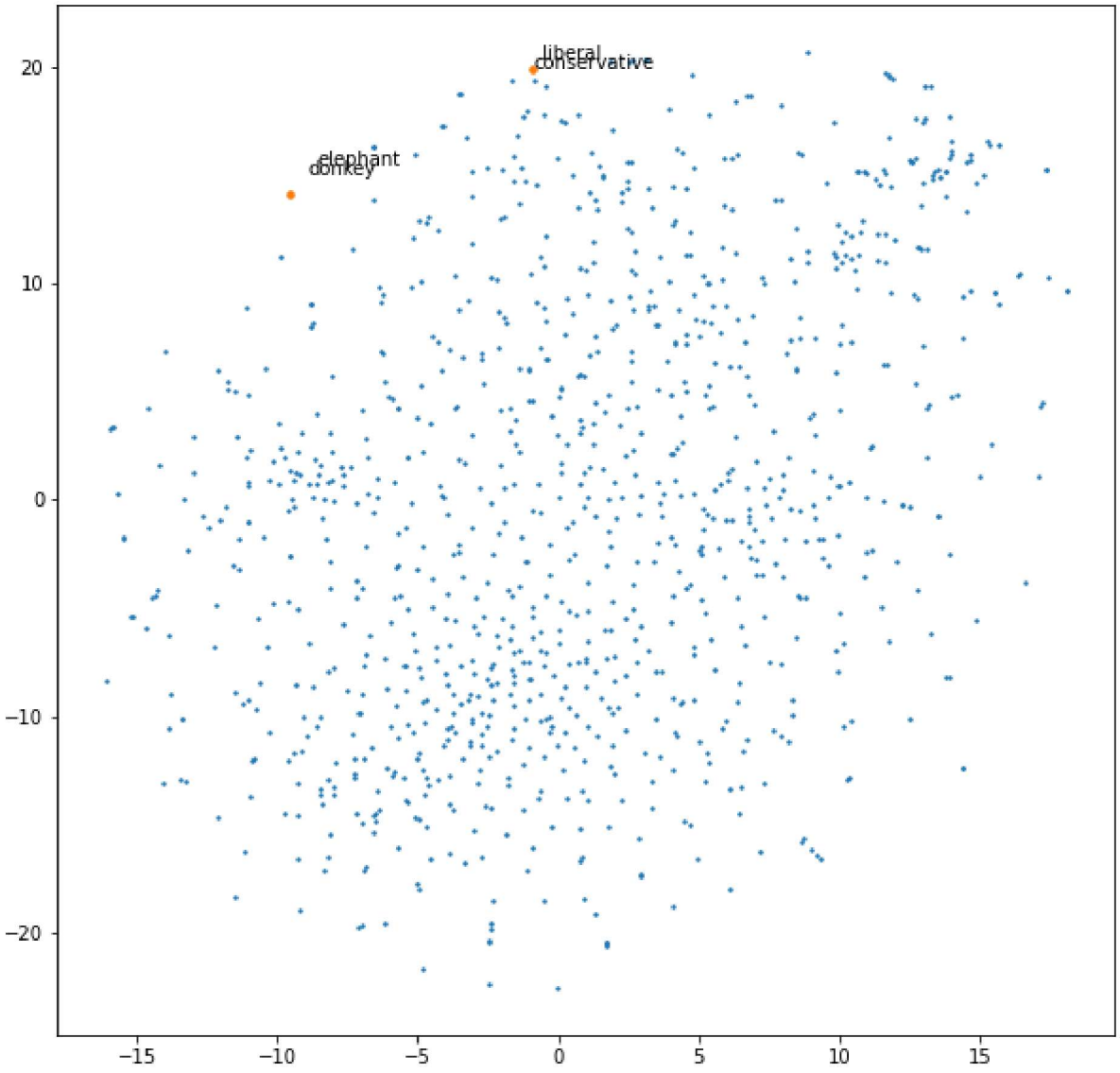
Find at least three more examples that produce expected results and three examples that produce surprising results. Include the plots in your write-up. Give reasons why you might see surprising behavior here.

In [ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
```

```
# this functions computes and displays the 2-d t-SNE maps for a subset of the embedding
# and displays them together with the points for a set of input words.

def display_tsne_neighborhood(model, input_word, nsample=1000, size1=2, size2=10, offse

    arr = np.empty((0,100), dtype='f')
    word_label = input_word

    # add the vector for each of the closest words to the array
    for w in range(len(input_word)):
        arr = np.append(arr, np.array([model[input_word[w]]]), axis=0)

    voc = [w for w in model.key_to_index]
    wrds = np.random.choice(range(len(voc)), size=nsample, replace=False)
    for w in wrds:
        wrd_vector = model[voc[w]]
        arr = np.append(arr, np.array([wrd_vector]), axis=0)

    # find tsne coords for 2 dimensions
    tsne = TSNE(n_components=2, random_state=0)
    np.set_printoptions(suppress=True)
    Y = tsne.fit_transform(arr)

    x_coord = Y[:, 0]
    y_coord = Y[:, 1]
    # display scatter plot
    size=2
    plt.scatter(x_coord, y_coord, s=size1)
    plt.scatter(x_coord[0:len(input_word)], y_coord[0:len(input_word)],s=size2)

    # label the input words
    for w in range(len(input_word)):
        plt.annotate(input_word[w], xy=(x_coord[w],y_coord[w]),
                     xytext=(w*offset,w*offset), textcoords='offset points')
    plt.show()
```

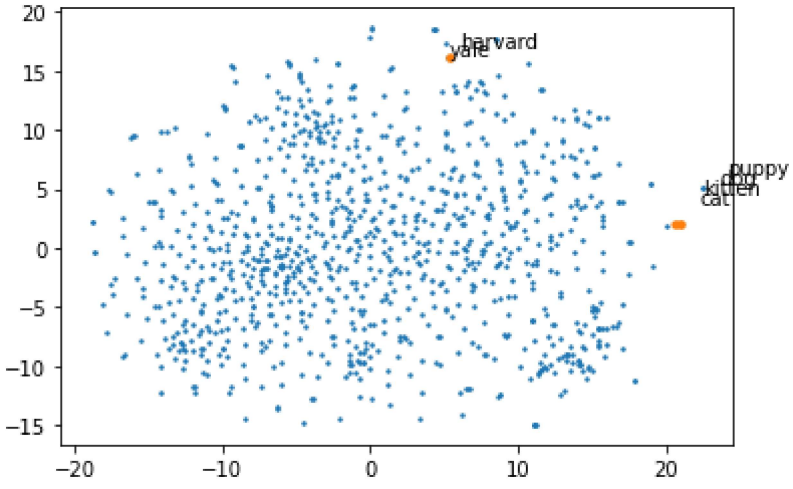
In [ ]:

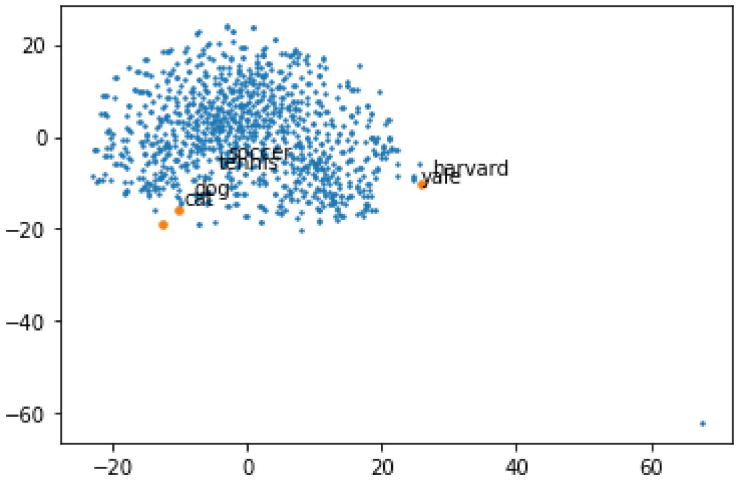
```
%matplotlib inline
plt.figure(figsize=(10,10))
display_tsne_neighborhood(glove, input_word = ['conservative', 'liberal', 'donkey', 'el
```



In [ ]:

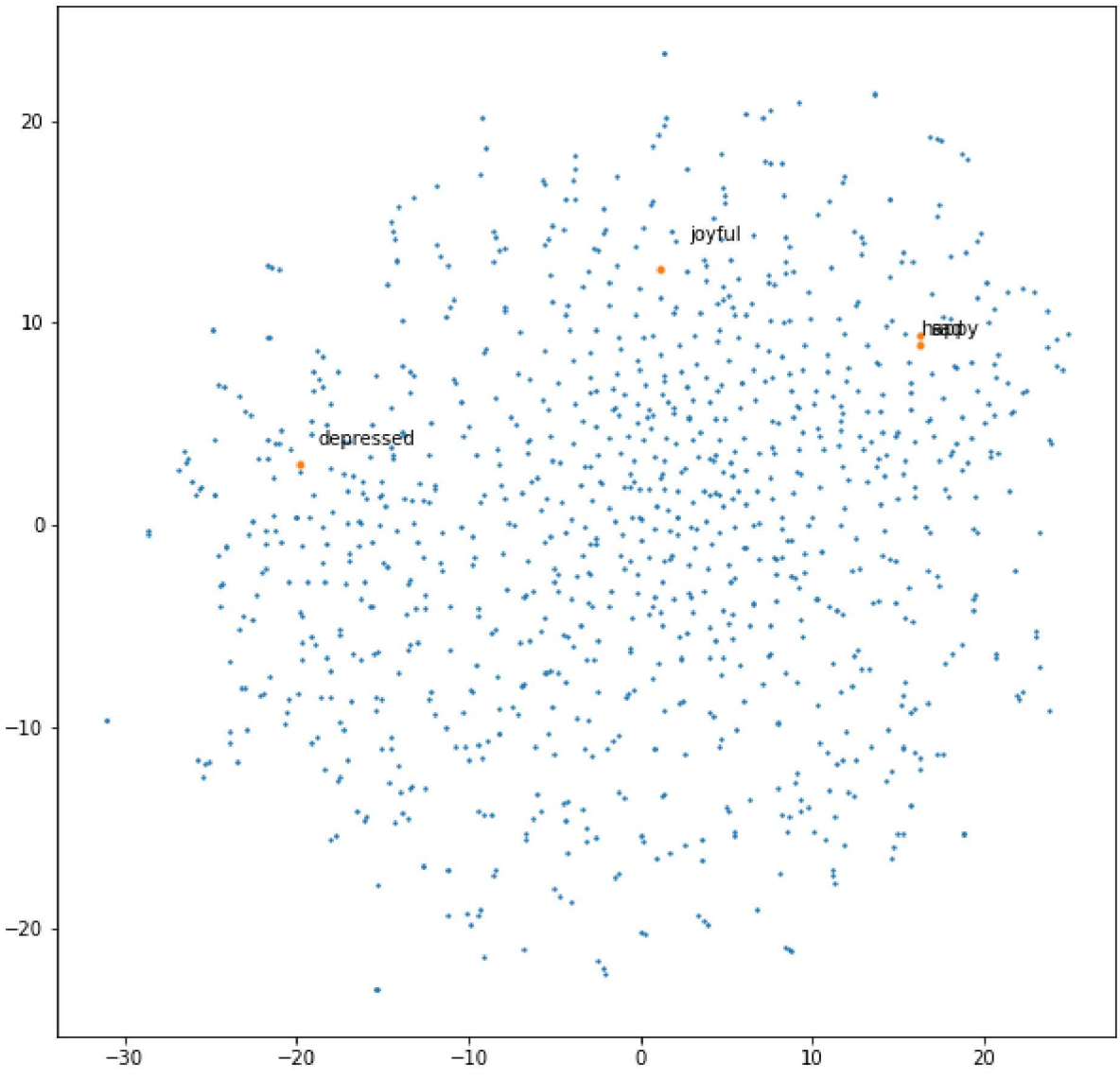
```
# expected results
display_tsne_neighborhood(glove, input_word = ['yale', 'harvard', 'cat', 'kitten', 'dog
display_tsne_neighborhood(model.wv, input_word =['yale', 'harvard', 'cat', 'dog', 'tenn
```





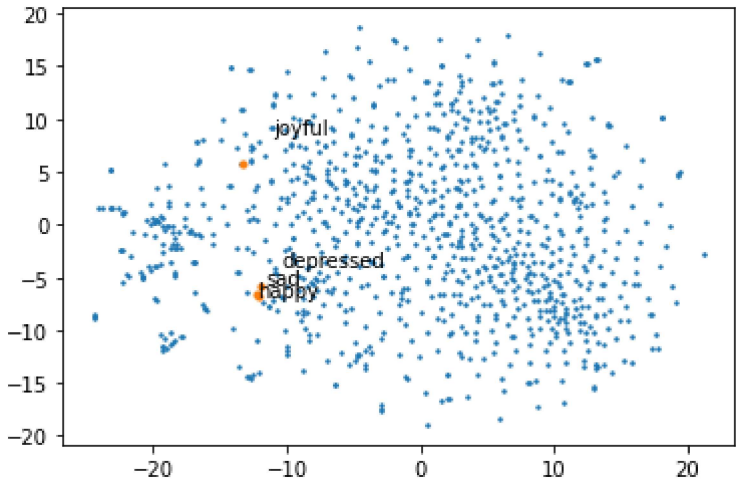
In [ ]:

```
# Surprising Examples
%matplotlib inline
plt.figure(figsize=(10, 10))
display_tsne_neighborhood(model.wv, input_word =['happy', 'sad', 'depressed', 'joyful']
```

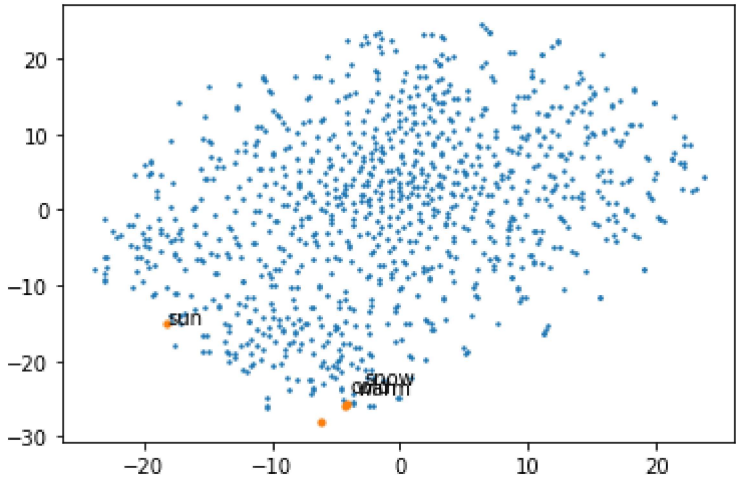


In [ ]:

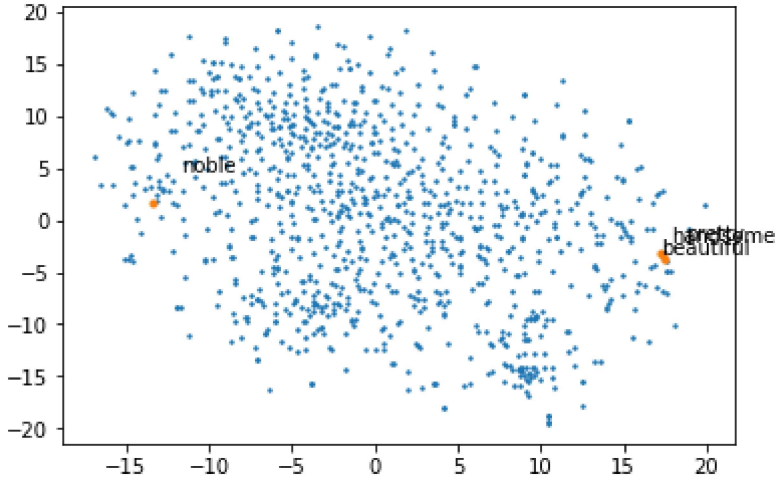
```
display_tsne_neighborhood(glove, input_word =['happy', 'sad', 'depressed', 'joyful'])
```



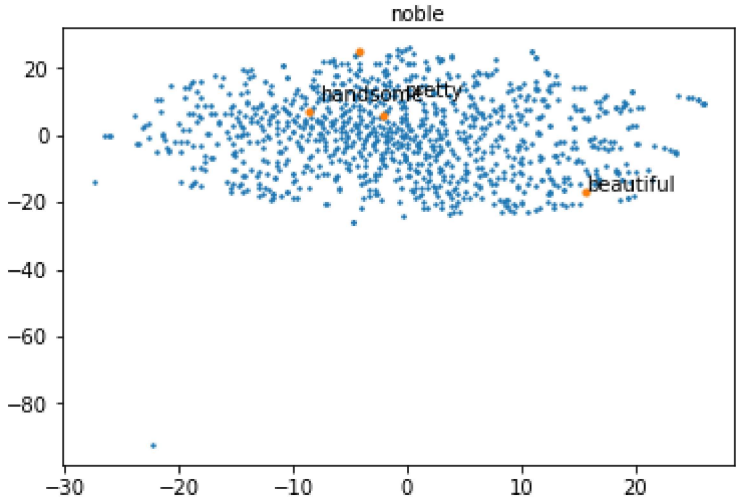
In [ ]:
```
display_tsne_neighborhood(model.wv, input_word =['sun', 'warm', 'snow', 'cold'])
```



In [ ]:
```
display_tsne_neighborhood(glove, input_word =['beautiful', 'handsome', 'pretty', 'noble
```



In [ ]:
```
display_tsne_neighborhood(model.wv, input_word =['beautiful', 'handsome', 'pretty', 'no
```



The model we are using are not that advance or complicated, so it make simple decisions. Sometimes the word could have multiple meanings so our model confuses it. For example, 'sun' has many aspects and attributes, hence it is far away from warm and snow.

Problem 3: Experiments with Musician Embeddings (15 points)

In this problem, we will use a collection of playlists obtained from last.fm. We treat each playlist as a document, and each artist in the playlist as a word. By feeding this dataset to word2vec, we will be able to learn artist embeddings.

Artist Embeddings

The following experiments will be done with the playlist data file `playlists.txt` . Each line in this file is a playlist. The integers on each line are unique artist identifiers, indicating which artists were in each playlist. The artists are in `artists.txt` .

The code below constructs artist embeddings with word2vec. The artist names are mapped to id numbers in the playlists; the code maps them back to display the names.

In [ ]:
```
import gensim
from gensim.models import word2vec
import logging

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.I
```

In [ ]:
```
playlists = word2vec.LineSentence('https://raw.githubusercontent.com/YData123/sds265-fa
music_model = word2vec.Word2Vec(playlists, vector_size=64, window=100, min_count=10)
```

```
2021-11-11 23:21:11,512 : INFO : collecting all words and their counts
2021-11-11 23:21:11,802 : INFO : PROGRESS: at sentence #0, processed 0 words, keeping 0
word types
2021-11-11 23:21:13,264 : INFO : PROGRESS: at sentence #10000, processed 107891 words, k
eeping 3250 word types
2021-11-11 23:21:14,273 : INFO : collected 3292 word types from a corpus of 189900 raw w
ords and 18111 sentences
2021-11-11 23:21:14,274 : INFO : Creating a fresh vocabulary
2021-11-11 23:21:14,286 : INFO : Word2Vec lifecycle event {'msg': 'effective_min_count=1
0 retains 2228 unique words (67.67922235722965%% of original 3292, drops 1064)', 'dateti
```

```
me': '2021-11-11T23:21:14.284913', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 13
2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0',
'event': 'prepare_vocab'}
2021-11-11 23:21:14,288 : INFO : Word2Vec lifecycle event {'msg': 'effective_min_count=1
0 leaves 183255 word corpus (96.50078988941549%% of original 189900, drops 6645)', 'date
time': '2021-11-11T23:21:14.288660', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 1
3 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0',
'event': 'prepare_vocab'}
2021-11-11 23:21:14,310 : INFO : deleting the raw counts dictionary of 3292 items
2021-11-11 23:21:14,311 : INFO : sample=0.001 downsamples 59 most-common words
2021-11-11 23:21:14,312 : INFO : Word2Vec lifecycle event {'msg': 'downsampling leaves e
stimated 173511.13228209765 word corpus (94.7%% of prior 183255)', 'datetime': '2021-11-
11T23:21:14.312217', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:0
3) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0', 'event': 'prep
are_vocab'}
2021-11-11 23:21:14,350 : INFO : estimated required memory for 2228 words and 64 dimensi
ons: 2254736 bytes
2021-11-11 23:21:14,351 : INFO : resetting layer weights
2021-11-11 23:21:14,356 : INFO : Word2Vec lifecycle event {'update': False, 'trim_rule':
'None', 'datetime': '2021-11-11T23:21:14.356665', 'gensim': '4.1.2', 'python': '3.8.8 (d
efault, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.
0.22494-SP0', 'event': 'build_vocab'}
2021-11-11 23:21:14,358 : INFO : Word2Vec lifecycle event {'msg': 'training model with 3
workers on 2228 vocabulary and 64 features, using sg=0 hs=0 sample=0.001 negative=5 wind
ow=100 shrink_windows=True', 'datetime': '2021-11-11T23:21:14.358655', 'gensim': '4.1.
2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]', 'pla
tform': 'Windows-10-10.0.22494-SP0', 'event': 'train'}
2021-11-11 23:21:15,464 : INFO : EPOCH 1 - PROGRESS: at 34.79% examples, 58060 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:16,548 : INFO : EPOCH 1 - PROGRESS: at 77.88% examples, 62563 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:17,130 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:21:17,133 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:21:17,166 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:21:17,171 : INFO : EPOCH - 1 : training on 189900 raw words (173691 effect
ive words) took 2.8s, 61907 effective words/s
2021-11-11 23:21:18,276 : INFO : EPOCH 2 - PROGRESS: at 34.79% examples, 58086 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:19,287 : INFO : EPOCH 2 - PROGRESS: at 77.88% examples, 64706 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:19,783 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:21:19,789 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:21:19,815 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:21:19,815 : INFO : EPOCH - 2 : training on 189900 raw words (173574 effect
ive words) took 2.6s, 65796 effective words/s
2021-11-11 23:21:20,934 : INFO : EPOCH 3 - PROGRESS: at 34.79% examples, 57274 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:21,954 : INFO : EPOCH 3 - PROGRESS: at 73.91% examples, 59675 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:22,648 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:21:22,663 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:21:22,689 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:21:22,691 : INFO : EPOCH - 3 : training on 189900 raw words (173475 effect
ive words) took 2.9s, 60478 effective words/s
2021-11-11 23:21:23,718 : INFO : EPOCH 4 - PROGRESS: at 30.04% examples, 53458 words/s,
in_qsize 0, out_qsize 0
```

```
2021-11-11 23:21:24,780 : INFO : EPOCH 4 - PROGRESS: at 77.88% examples, 65453 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:25,187 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:21:25,202 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:21:25,218 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:21:25,218 : INFO : EPOCH - 4 : training on 189900 raw words (173463 effect
ive words) took 2.5s, 68612 effective words/s
2021-11-11 23:21:26,313 : INFO : EPOCH 5 - PROGRESS: at 45.07% examples, 75503 words/s,
in_qsize 0, out_qsize 0
2021-11-11 23:21:27,292 : INFO : worker thread finished; awaiting finish of 2 more threa
ds
2021-11-11 23:21:27,292 : INFO : worker thread finished; awaiting finish of 1 more threa
ds
2021-11-11 23:21:27,308 : INFO : EPOCH 5 - PROGRESS: at 100.00% examples, 83000 words/s,
in_qsize 0, out_qsize 1
2021-11-11 23:21:27,308 : INFO : worker thread finished; awaiting finish of 0 more threa
ds
2021-11-11 23:21:27,308 : INFO : EPOCH - 5 : training on 189900 raw words (173391 effect
ive words) took 2.1s, 82889 effective words/s
2021-11-11 23:21:27,324 : INFO : Word2Vec lifecycle event {'msg': 'training on 949500 ra
w words (867594 effective words) took 13.0s, 66928 effective words/s', 'datetime': '2021
-11-11T23:21:27.324102', 'gensim': '4.1.2', 'python': '3.8.8 (default, Apr 13 2021, 15:0
8:03) [MSC v.1916 64 bit (AMD64)]', 'platform': 'Windows-10-10.0.22494-SP0', 'event': 't
rain'}
2021-11-11 23:21:27,324 : INFO : Word2Vec lifecycle event {'params': 'Word2Vec(vocab=222
8, vector_size=64, alpha=0.025)', 'datetime': '2021-11-11T23:21:27.324102', 'gensim':
'4.1.2', 'python': '3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]',
'platform': 'Windows-10-10.0.22494-SP0', 'event': 'created'}
```

In [ ]:
```
music_model.wv['299']
```

Out[ ]:
```
array([ 0.43221834, -0.719046  ,  0.35036913, -0.14231592,  0.4971208 ,
       -0.1797617 , -0.5566266 ,  0.20567587,  0.13281594,  1.0846256 ,
       -1.0277593 ,  0.03148408, -1.1114904 ,  1.0139874 , -0.58704954,
        0.71923184, -1.2146462 , -0.4928999 , -0.17486212, -0.00302529,
        0.3356248 , -0.1338086 ,  0.8927364 , -0.5659078 ,  0.14080319,
        0.07290081, -0.62687695, -0.37941518, -0.2949403 ,  0.6088427 ,
        0.12690884, -0.73043376, -0.4217455 , -0.5017378 ,  0.22015284,
       -0.79450095, -0.03438405,  0.62053007,  0.2710968 , -0.21670812,
       -0.3212371 ,  0.4089281 ,  0.47807458, -0.04247258, -0.73664457,
       -0.24361713, -0.14736517, -0.265629  ,  0.39810064,  0.8482604 ,
       -0.18930028, -0.9559344 ,  0.5377905 ,  0.99594253,  0.24159366,
        0.8123202 ,  0.6239836 , -0.84368503, -0.44289273,  1.108842  ,
       -0.2063995 , -0.8246551 , -0.18020152,  0.48357093], dtype=float32)
```

In [ ]:
```
from urllib.request import urlopen

artist = []
file = urlopen('https://raw.githubusercontent.com/YData123/sds265-fa21/main/assignments
for line in file:
    art = line.decode("utf-8")
    artist.append(art.strip())

artist[0:10]
```

Out[ ]:
```
['Everette Harp',
 'Bishop Paul S. Morton & Aretha Franklin',
 'Frankie Ballard',
 'Herb Alpert',
```

```
'Rod Stewart & Chaka Khan',
'Scars On 45',
'New Radicals',
'Crosby, Stills & Nash',
'Ledisi',
'La Quinta Estacion']
```

In [ ]:
```
id2name = {}
name2id = {}
for w in range(len(artist)):
    id2name["%s" % w] = artist[w]
    name2id[artist[w]] = "%s" % w

id2name[name2id['Elton John']]
```

Out[ ]:
```
'Elton John'
```

3.1 Similar artists

Find the 5 closest artist embedding vectors to the artists "The Beatles", "Lady Gaga", and "Nirvana". Comment on the results.

In [ ]:
```
# Don't change this function
def similar_artists(model, artist, n=5):
    id = name2id[artist]
    out = model.wv.most_similar(id, topn=n)

    print("artists similar to '%s'\n" % artist)
    for i in range(n) :
        name = id2name[out[i][0]]
        print("\t%s" % name)

similar_artists(music_model, 'Aerosmith')
```

```
artists similar to 'Aerosmith'

        Billy Idol
        The Jimi Hendrix Experience
        Cheap Trick
        Def Leppard
        George Thorogood & The Destroyers
```

In [ ]:
```
similar_artists(music_model, 'The Beatles')
```

```
artists similar to 'The Beatles'

        Steppenwolf
        Grand Funk Railroad
        Creedence Clearwater Revival
        The Electric Light Orchestra
        Warren Zevon
```

In [ ]:
```
similar_artists(music_model, 'Lady Gaga')
```

```
artists similar to 'Lady Gaga'

        Bruno Mars
        Katy Perry
```

```
        Black Eyed Peas
        Christina Perri
        Pink
```

In [ ]:
```
similar_artists(music_model, 'Nirvana')
```

```
artists similar to 'Nirvana'

        Pearl Jam
        Temple Of The Dog
        Faith No More
        Stone Temple Pilots
        Bush
```

For each artist, the music style and active time period of the 5 closest artists is similar to him/her/themselves. For example, The Beatles were an English rock band active between 1960 and 1970, while Steppenwolf was an American rock band that was prominent from 1968 to 1972. The Electric Light Orchestra (ELO) are an English rock band formed in Birmingham in 1970. Also, Nirvana and Beatles are bands so the closest artists are mostly bands. Lady Gaga is single singer so Bruno Mars, Katy Perry, etc. are single popular singers.

3.3 Visualize embeddings

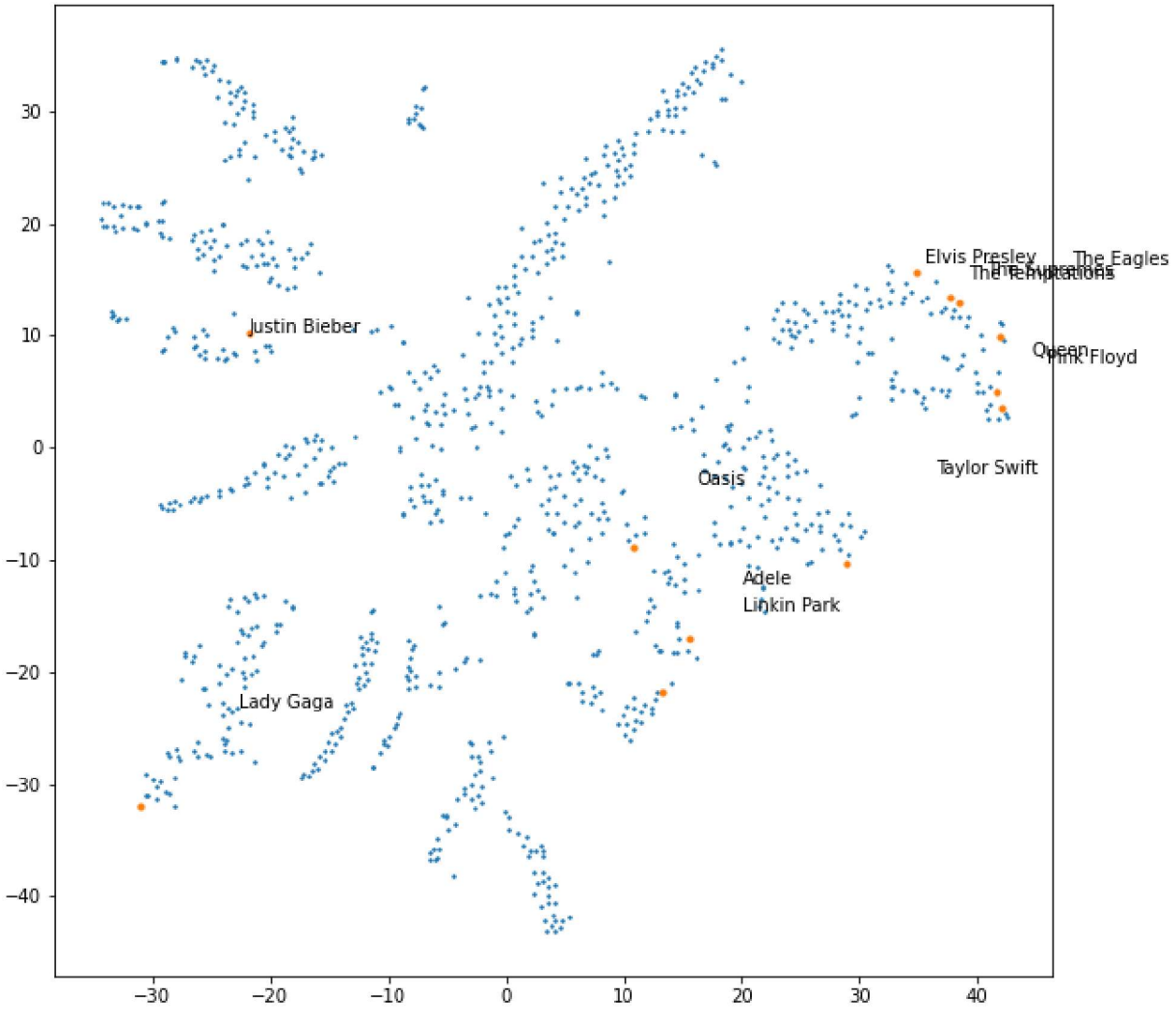
Use the t-SNE dimensionality reduction technique to visualize the artist embeddings. After running t-SNE on the artist embeddings, try visualizing "The Temptations" and "The Supremes" together. Find a few more examples that you think are interesting and include the plots in your write-up. Comment on your findings.

In [ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE


# this functions computes and displays the 2-d t-SNE maps for a subset of the embedding
# and displays them together with the points for a set of input words.

def display_tsne_artists(model, artists, nsample=1000, size1=2, size2=10, offset=5):

    arr = np.empty((0,64), dtype='f')

    # add the vector for each of the closest words to the array
    for a in range(len(artists)):
        id = name2id[artists[a]]
        arr = np.append(arr, np.array([model[id]]), axis=0)

    voc = [w for w in model.key_to_index]
    ids = np.random.choice(range(len(voc)), size=nsample, replace=False)
    for w in ids:
        wrd_vector = model[voc[w]]
        arr = np.append(arr, np.array([wrd_vector]), axis=0)

    # find tsne coords for 2 dimensions
    tsne = TSNE(n_components=2, random_state=0)
    np.set_printoptions(suppress=True)
    Y = tsne.fit_transform(arr)
```

```
    x_coord = Y[:, 0]
    y_coord = Y[:, 1]
    # display scatter plot
    size=2
    plt.scatter(x_coord, y_coord, s=size1)
    plt.scatter(x_coord[0:len(artists)], y_coord[0:len(artists)],s=size2)

    # label the input words
    for w in range(len(artists)):
        plt.annotate(artists[w], xy=(x_coord[w],y_coord[w]), \
                     xytext=(w*offset,w*offset), textcoords='offset points')
    plt.show()
```

In [ ]:

```
%matplotlib inline
plt.figure(figsize=(10, 10))
display_tsne_artists(music_model.wv, artists=["Justin Bieber", "Elvis Presley", "The Te
```



The music style and active time period are the most important variables for clustering. For example, "Pink Floyd" and "Queen" belong to the same music genre "rock" and they are active at the same time period. We can see that generally the lefter the samples are in the plot, the later their active years are. Also, the up and right clustering is the rock genre artists active on 1960-1990s, and the down cluster is the popular genre artists active on 21st century.