به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

تمرین ۲ بخش دوم

نام و نام خانودگی: علی عطاءاللهی

شماره دانشجویی: ۸۱۰۱۹۹۴۶۱

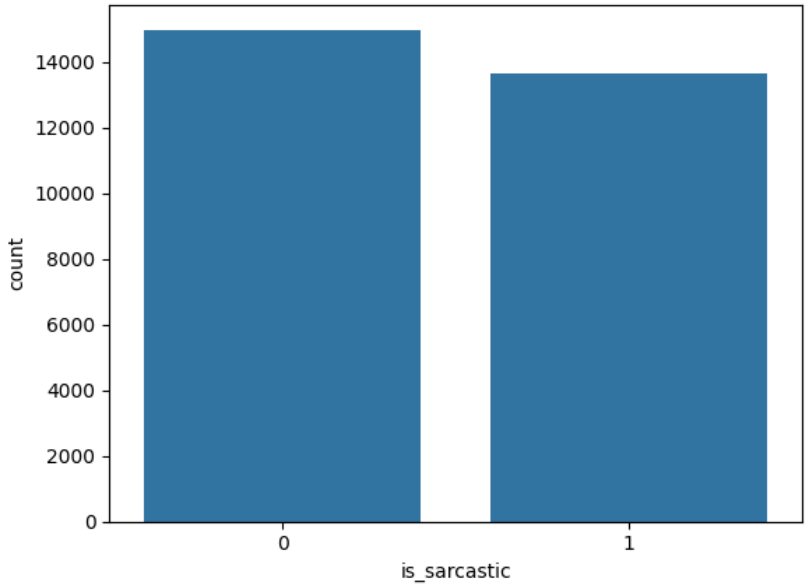
فروردین ماه ۱۴۰۳

فهرست

پاسخ سوال دوم

پاسخ بخش اول – پیش پردازش مجموعه داده

ابتدا داده را با کمک کتابخانه pandas می‌خوانیم.سپس با استفاده از کتابخانه seaborn نمودار لیبل‌ها را رسم می کنیم.



در تابع clean_headline_text، متن را به حالت استاندار و تمیز تبدیل می‌کنیم. این کار شامل حذف علائم غیرالفبا، تبدیل کاراکترها به شکل کوچک، جدا کردن کلمات، حذف کلمات stopword و متصل کردن کلمات باقیمانده به هم می‌شود.

```python
def clean_headline_text(text_str):
    text_str = re.sub('[^a-zA-Z]', ' ', text_str)
    text_str = text_str.lower()
    text_str = text_str.split(' ')
    text_str = [w for w in text_str if not w in set(stopwords.words('english'))]
    text_str = ' '.join(text_str)
    return text_str
```

پاسخ بخش دوم – بارگذاری GLOVE

Glove : از یک روش تعبیه سازی کلمات استفاده می‌کنیم که آن را از قبل دانلود کرده و کده را از حالت zip خارج کرده و از glove.6B.100d.txt به عنوان منبع استفاده می‌کنیم. سپس یک دیکشنری embedding از کلمات ایجاد می‌کنیم. حالا جملات را ابتدا توکنایز می‌کنیم و سپس برای تمام جملات دیتاست، بردار glove مربوط به کلمات آن‌ها را بدست می‌آوریم. برای این کار کافی است کلمات جمله را در دیکشنری embedding بخوانیم و مقدار آن‌ها را ذخیره کنیم. بعد از اینکه بردار glove را برای همه کلمات جمله بدست آوردیم، از آن‌ها میانگین می‌گیریم و آن را برمی‌گردانیم تا به عنوان ویژگی برای classification استفاده کنیم.

```
embeddings_dict = {}
file = open('data/glove.6B.100d.txt', encoding='utf-8')
for line in file:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_dict[word] = coefs
file.close()

print('Found %s word vectors.' % len(embeddings_dict))
```

```python
def compute_embedding_glove(sentence_list):
    vectors_list = []
    for word in sentence_list:
        try:
            word_vec = embeddings_dict[word]
            vectors_list.append(word_vec)
        except:
            continue
    return np.mean(vectors_list, axis=0)
```

پاسخ بخش سوم – آموزش مدل

در مرحله بعد، داده‌ها را با استفاده از ماژول train_test_split از کتابخانه Sklearn به دو بخش train و test تقسیم می‌کنیم.
این کار را با شافل کردن داده‌ها انجام می‌دهیم و نسبت را طبق صورت پروژه به نسبت ۸۰ به ۲۰ درصد قرار می‌دهیم.

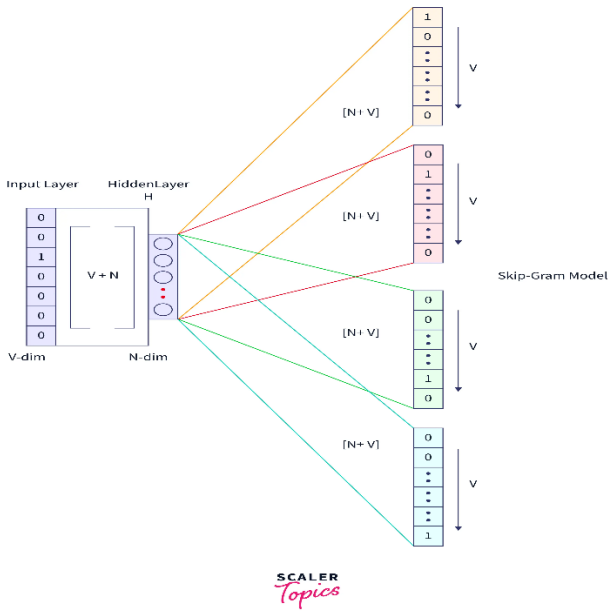
```python
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, shuffle=True, test_size=0.2, random_state=60)
```

پاسخ سوال سوم

پاسخ بخش اول

preprocess

ما در این تمرین از معماری skip-gram با استفاده از negative sampling برای پیاده‌سازی word2vec استفاده می‌کنیم.



در این روش، حسن آموزش این روش نسبت به CBOW را به دلیل سرعت بالاتر آموزش آن برجسته می‌کند. در این روش، مدل یک کلمه را دریافت می‌کند و باید کلمات اطراف آن را پیش‌بینی کند. این امکان وجود دارد که شبکه را برای نمایش کلمات در آموزش قرار دهیم، به گونه‌ای که کلمات با یک context مشابه، بردار embedding مشابهی داشته باشند.

create_lookup_tables

در این تابع، دو دیکشنری پیاده‌سازی شده است. یکی از این دیکشنری‌ها کلمه را دریافت می‌کند و ایندکس آن را برمی‌گرداند، و دیکشنری دیگر برعکس عمل می‌کند. از این تابع برای ساخت این دو دیکشنری برای کلمات corpus استفاده می‌کنیم.

```python
def create_lookup_tables(words):
    word_counts = Counter(words)
    sorted_vocab = sorted(word_counts, key=word_counts.get, reverse=True)
    int_to_vocab = {ii: word for ii, word in enumerate(sorted_vocab)}
    vocab_to_int = {word: ii for ii, word in int_to_vocab.items()}
    return vocab_to_int, int_to_vocab

def convert_words_to_ints(words, vocab_to_int):
    return [vocab_to_int[word] for word in words]

vocab_to_int, int_to_vocab = create_lookup_tables(words)
int_words = convert_words_to_ints(words, vocab_to_int)
```

برخی کلمات مانند "for" یا "a" برای کلمات همسایه، معمولاً context خوبی فراهم نمی‌کنند. بنابراین، حذف آن‌ها به بهبود کیفیت تعبیه سازی کلمات کمک می‌کند و سرعت یادگیری را افزایش می‌دهد. این فرآیند به زیرمجموعه‌گیری اطلاق می‌شود. برای هر کلمه w_i، ما آن را با احتمال زیر حذف می‌کنیم:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

در اینجا، $f(w_i)$ تعداد تکرار کلمه در دیتاست است و t پارامتر حد آستانه است. پس در پیاده‌سازی برای تمام کلمات، تعداد تکرار هر کلمه را حساب می‌کنیم و از فرمول بالا احتمال حذف آن را محاسبه می‌کنیم. سپس با کمک تابع random، اگر عدد تصادفی تولید شده کمتر از این احتمال باشد، آن کلمه را حذف می‌کنیم.

```
threshold = 1e-5
word_counts = Counter(int_words)
#print(list(word_counts.items())[0])  # dictionary of int_words, how many times they appear

total_count = len(int_words)
freqs = {word: count/total_count for word, count in word_counts.items()}
p_drop = {word: 1 - np.sqrt(threshold/freqs[word]) for word in word_counts}
# discard some frequent words, according to the subsampling equation
# create a new list of words for training
train_words = [word for word in int_words if random.random() < (1 - p_drop[word])]

```

get_target

دیتای ما حالا مناسب برای دادن به مدل شده است. در معماری skip-gram، برای هر کلمه در متن، باید context اطراف آن را تعریف کنیم و تمام کلماتی که در پنجره به اندازه C اطراف آن قرار دارند را بدست آوریم. به عبارت دیگر، C/2 کلمه قبلی و C/2 کلمه بعدی را به عنوان همسایگان کلمه داده شده در نظر می‌گیریم. در اینجا، سایز C را ۴ در نظر گرفتیم. بنابراین، دو کلمه قبل و بعد جزو context حساب می‌شوند.

```python
def get_target(words, idx, window_size=2):
    ''' Get a list of words in a window around an index. '''

    R = np.random.randint(1, window_size+1)
    start = idx - R if (idx - R) > 0 else 0
    stop = idx + R
    target_words = words[start:idx] + words[idx+1:stop+1]

    return list(target_words)
```

get_batches

حالا باید یک تابع بسازیم که برای مدل، batch از ورودی و تارگت بسازد. ورودی‌ها کلمات داخل corpus هستند و تارگت کلمات همسایه آن‌ها هستند که می‌توان برای بدست آوردن آن‌ها از تابع get_target کمک گرفت. برای این کار، اولاً فقط batchهای کامل را در نظر می‌گیریم، یعنی batch آخر اگر سایز کافی را نداشته باشد، حذف می‌شود. سپس برای هر batch به ازای تمام کلمات آن، همسایگان آن را بدست می‌آوریم.

```python
def get_batches(words, batch_size, window_size=2):
    ''' Create a generator of word batches as a tuple (inputs, targets) '''

    n_batches = len(words)//batch_size

    # only full batches
    words = words[:n_batches*batch_size]

    for idx in range(0, len(words), batch_size):
        x, y = [], []
        batch = words[idx:idx+batch_size]
        for ii in range(len(batch)):
            batch_x = batch[ii]
            batch_y = get_target(batch, ii, window_size)
            y.extend(batch_y)
            x.extend([batch_x]*len(batch_y))
        yield x, y
```

cosine_similarity

برای اینکه ببینیم مدل ما در هر مرحله چقدر پیشرفت داشته، علاوه بر loss می‌توانیم در هر مرحله شباهت کلمات corpus را با یک سری از کلمات سنجید و چند تای نزدیک آن را چاپ کنیم. برای اندازه‌گیری نزدیکی، از کسینوس دو بردار استفاده می‌کنیم. این یک راه خوب و جانبی برای ارزیابی مدل ما است.

$$similarity = \cos(\theta) = \frac{\vec{a}.\vec{b}}{|\vec{a}|.|\vec{b}|}$$

```python
def cosine_similarity(embedding, valid_size=16, valid_window=100, device='cpu'):
    """ Returns the cosine similarity of validation words with words in the embedding matrix.
        Here, embedding should be a PyTorch embedding module.
    """

    # Here we're calculating the cosine similarity between some random words and
    # our embedding vectors. With the similarities, we can look at what words are
    # close to our random words.
    # sim = (a . b) / |a||b|
    embedding_vectors = embedding.weight

    # Magnitude of embedding vectors, |b|
    magnitudes = embedding_vectors.pow(2).sum(dim=1).sqrt().unsqueeze(0)

    # Pick N words from our ranges (0,window) and (1000,1000+window). Lower id implies more frequent
    valid_examples = np.array(random.sample(range(valid_window), valid_size//2))
    valid_examples = np.append(valid_examples,
                               random.sample(range(1000,1000+valid_window), valid_size//2))
    valid_examples = torch.LongTensor(valid_examples).to(device)
    valid_vectors = embedding(valid_examples)
    similarities = torch.mm(valid_vectors, embedding_vectors.t())/magnitudes

    return valid_examples, similarities
```

تابع forward_noise در کلاس SkipGramNeg

در skip-gram، که فقط از کلمات context استفاده می‌کنیم و بقیه کلمات را non-context در نظر می‌گیریم، دو مشکل وجود دارد:

- برای هر نمونه training، فقط وزن‌های مربوط به کلمات context ممکن است تغییر معناداری کنند، در حالی که در فرآیند back-propagation ما تلاش می‌کنیم تمام وزن‌های لایه‌های مخفی را آپدیت کنیم. وزن‌های مربوط به کلمات non-context بسیار اندک یا اصلاً تغییر نمی‌کنند، یعنی در هر مرحله آپدیت وزن‌ها اسپارس است.
- برای هر نمونه training، محاسبه احتمالات نهایی با استفاده از softmax هزینه بر است، زیرا مخرج کسر شامل جمع امتیازات کل کلمات vocabulary برای نرمالیز کردن می‌شود.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} for\ j = 1,2,\dots,k$$

```python
def forward_noise(self, batch_size, num_samples):
    """ Generate noise vectors with shape (batch_size, num_samples, embedding_size)"""
    if self.noise_dist is None:
        # Sample words uniformly
        noise_dist = torch.ones(self.vocab_size)
    else:
        noise_dist = self.noise_dist

    # Sample words from our noise distribution
    noise_words = torch.multinomial(noise_dist,
                                    batch_size * num_samples,
                                    replacement=True)
    device = "cuda" if model.output_embedding.weight.is_cuda else "cpu"
    noise_words = noise_words.to(device)
    noise_vectors = self.output_embedding(noise_words).view(batch_size, num_samples, self.embedding_size)
    return noise_vectors
```

برای غلبه بر این دو مشکل، به جای اینکه brute force عمل کنیم، یعنی همه وزن‌ها را بخواهیم آپدیت کنیم، باید سعی کنیم تعداد وزن‌های آپدیت شده برای هر نمونه آموزشی را کاهش دهیم. تکنیک مورد نظر برای این کار negative sampling است. به جای اینکه احتمال همسایگی را پیش‌بینی کنیم، سعی می‌کنیم احتمال اینکه دو کلمه همسایه هستند یا نیستند را پیش‌بینی کنیم. به عنوان مثال، در روش عادی، احتمال P(word1| word2) را پیش‌بینی می‌کردیم، اما در روش negative sampling احتمال P(1|<word1, word2>) را حساب می‌کنیم. همچنین برای ساده‌تر شدن مسئله، به طور تصادفی تعداد محدودی از کلمات را به عنوان کلمات negative برای آپدیت وزن‌ها انتخاب می‌کنیم که k برابر با تعداد کلمات positive است. بنابراین، loss فقط برای کلمات context و negative عمل backpropagation را انجام می‌دهد.

Loss برای کلاس NegativeSamplingLoss

تابع هدف:

$$-\log\sigma\left(u_{w_0}^T v_{w_1}\right) - \sum_{i}^{N} E_{w_i \sim P_n(w)} log\sigma(-u_{w_i}^T v_{w_I})$$

در اینجا، u_{w_0} همان بردار جانمایی است و v_{w_I} بردار زمینه است. عبارت اول تارگت‌های درست است و عبارت دوم تارگت‌های نویزی یا negative است. پس مدل سعی می‌کند برای کلماتی که context هستند ۱ را پیش‌بینی کند و برای کلمات negative که non-context هستند ۰ را پیش‌بینی کند.

```python
class NegativeSamplingLoss(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, input_vectors, output_vectors, noise_vectors):

        batch_size, embed_size = input_vectors.shape

        # Input vectors should be a batch of column vectors
        input_vectors = input_vectors.view(batch_size, embed_size, 1)

        # Output vectors should be a batch of row vectors
        output_vectors = output_vectors.view(batch_size, 1, embed_size)

        # bmm = batch matrix multiplication
        # correct log-sigmoid loss
        out_loss = torch.bmm(output_vectors, input_vectors).sigmoid().log()
        out_loss = out_loss.squeeze()

        # incorrect log-sigmoid loss
        noise_loss = torch.bmm(noise_vectors.neg(), input_vectors).sigmoid().log()
        noise_loss = noise_loss.squeeze().sum(1)  # sum the losses over the sample of noise vectors

        # negate and sum correct and noisy log-sigmoid losses
        # return average batch loss
        return -(out_loss + noise_loss).mean()
```

Trainng

ما ابتدا کورپوس را به تابع get_batches منتقل می‌کنیم تا برایمان داده‌های آموزشی مناسب را مطابق با توضیحات پیشین تولید کند، و سپس حلقه را روی آن اجرا می‌کنیم. هر بار محاسبات را روی یک batch انجام می‌دهیم.

```python
for input_words, target_words in get_batches(train_words, 512):
    steps += 1
    inputs, targets = torch.LongTensor(input_words), torch.LongTensor(target_words)
    inputs, targets = inputs.to(device), targets.to(device)
```

ابتدا با استفاده از تابع forward_input در کلاس SkipgramNeg، بردار ورودی embedding را محاسبه می‌کنیم.

```python
input_vectors = model.forward_input(inputs)
```

بعد برای بردار خروجی embedding را با کمک تابع forward_output از همان کلاس حساب می کنیم.

```
output_vectors = model.forward_output(targets)
```

آخر بردار کلمات negative را به تابع forward_noise می دهیم تا embedding آن را برگرداند.

```
noise_vectors = model.forward_noise(inputs.shape[0], 4)
```

حالا با دادن embedding به تابع forward کلاس NegativeSamplingLoss، loss را محاسبه می‌کنیم. سپس با انجام عمل backward propagation و optimization، وزن‌ها را به‌روزرسانی می‌کنیم.

```
# Negative sampling loss
loss = loss_fn(input_vectors, output_vectors, noise_vectors)

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

هایپرپارامترها و سایر تنظیمات :

epochs	25
optimizer	optim.Adam
embedding_dim	100
Learning rate	0.003
Neighborhood window size	4
number of negative samples for each sample	4

توجه: برای تولید کلمات negative از توزیع unigram استفاده می‌کنیم، اما برای جلوگیری از اینکه تاثیر کلمات نادر ناچیز نباشد، از فرمول زیر برای smoothing احتمالات استفاده می‌کنیم تا شانس انتخاب کلمات نادر کمی بیشتر شود:

$$P_n(w) = \left(\frac{U(w)}{Z}\right)^{3/4}$$

که U(w) توزیع یکنواخت کلمات است.

همان طور که ملاحظه می شود ایپاک‌های نهایی نسبت Loss کمی دارند و در مقایسه با ایپاک‌های قبلی به طور قابل ملاحظه ای کاهش یافته و این یعنی یادگیری اتفاق افتاده است.

```
Epoch: 6/25
Loss: 5.983024597167969
if | strangest, abode, resentment, sound, severed
all | children, repaid, jail, overpowering, sought
<QUOTATION_MARK> | heads, readers, eggs, soldier, brains
he | 'some, our, slashed, border, sutherland's
were | doors, firmness, snakish, inquirer, left-hand
in | irene, incarnate, mccauley, far-gone, horrify
our | although, he, urgency, pride, hit
him | assistants, that, fled, presents, averse
data | arthur, puzzling, delicate, eustace, false
solution | aloud, blue, chalk-pit, fulfilled, standi
'well | receive, crumbly, grosvenor, bustling, youth
intention | rambling, tenant, piping, turns, address
soul | villain, roofs, frantically, throughout, jowl
bar | miners', border, bachelor, harm, disturbing
knees | then, retained, pretended, showed, sign
grounds | shuddered, withdraw, wandering, voraciously, scaffolding
...

Epoch: 11/25
Loss: 4.536405086517334
or | superb, move, guess, sailing, tired
about | balancing, 'jumping, re-marriage, roar, chairs
of | fire, laugh, assizes, proposed, conscious
...
knees | then, pretended, epistle, belief, details
vanished | disqualify, wound, melon, sensitive, exclude
...
```

پس از آموزش مدل، دو ماتریس جانمایی و زمینه با هم جمع می‌شوند تا بردار embedding نهایی را بسازند.

```
embeddings = model.input_embedding.weight.to('cpu').data.numpy() + model.output_embedding.weight.to('cpu').data.numpy()
```

پاسخ بخش دوم

در اینجا، برای ارزیابی نحوه جاسازی کلمات از پیش آموزش دیده، یک کار قیاس کلمه انجام می‌دهیم. با گرفتن بردار برای "پادشاه"، کم کردن بردار برای "مرد" و اضافه کردن بردار برای "زن"، بر اساس فرمول مشخص بردار قیاسی را ایجاد می‌کنیم. سپس شباهت کسینوس بین این بردار قیاس و بردار "ملکه" واقعی محاسبه می‌شود و امتیازی بین ۰ و ۱ ارائه می‌شود. امتیاز بالا نشان می‌دهد که

جاسازی‌ها با موفقیت رابطه جنسیتی را نشان می‌دهند، در حالی که نمره پایین نشان‌دهنده محدودیت‌ها در جاسازی‌ها یا چالش‌هایی با آن کار قیاسی خاص است.

```python
king_idx = find_word_index('king', vocabulary_list)
man_idx = find_word_index('man', vocabulary_list)
woman_idx = find_word_index('woman', vocabulary_list)
queen_idx = find_word_index('queen', vocabulary_list)

king_vector = embeddings[king_idx]
man_vector = embeddings[man_idx]
woman_vector = embeddings[woman_idx]
queen_vector = embeddings[queen_idx]

# Compute the analogy vector
analogy_vector = king_vector - man_vector + woman_vector

# Calculate the cosine similarity between the analogy vector and the queen vector
similarity_score = 1 - cosine(analogy_vector, queen_vector)

print(f"Similarity between 'king - man + woman' and 'queen': {similarity_score}")
```

امتیاز حاصل در اجراهای مختلف تغییر می‌کند، اما حدود ۰.۲۵ می‌باشد. این عدد حاصل کسینوس بین بردار قیاسی که از "پادشاه - مرد + زن" ساخته شده است و بردار تعبیه شده واقعی برای "ملکه" می‌باشد. این امتیاز نسبتاً پایین است و هر چقدر پایین‌تر باشد، نشان می‌دهد که شباهت کمتری وجود دارد. دلیل این شباهت کم ممکن است از محدودیت‌های موجود در خود embedding ها باشد که نشان می‌دهد آنها ممکن است این الگوی معنایی خاص یا رابطه جنسیتی را به درستی رمزگذاری نکنند. همچنین می‌تواند دلیل آن کم بودن اطلاعات نسبت به این کلمات باشد که کلمه queen تکرار کمی در کورپوس دارد.

پاسخ بخش سوم

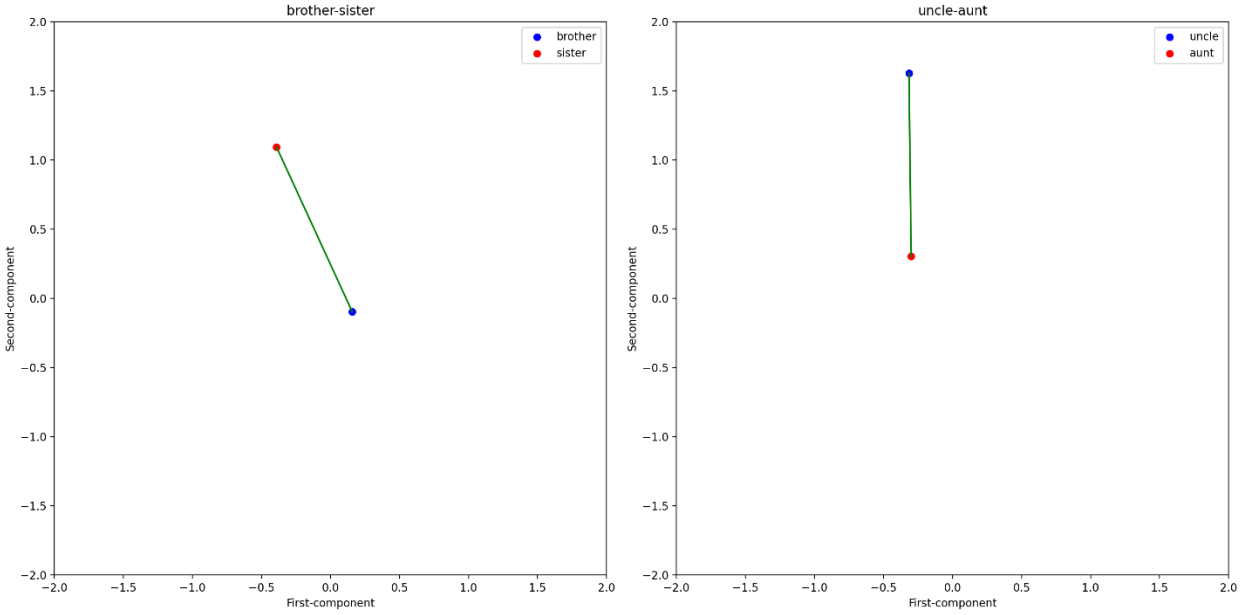
در این مرحله، طبق خواسته سوال، قصد داریم بردار تفاضل جفت کلمات مشخص شده را در یک نمودار دو بعدی نمایش دهیم.

```python
# Create a PCA object with 2 components
pca_model = PCA(n_components=2)

# Fit the word vectors to the PCA object
word_vectors_pca = pca_model.fit_transform(embeddings)

# Create a dictionary of the four vector pairs
word_pairs = {'brother-sister': ('brother', 'sister'),
              'uncle-aunt': ('uncle', 'aunt')}
```

خب، در حال حاضر هر کلمه شامل ۱۰۰ مقدار است. بنابراین، آن را نمی‌توان در یک نمودار دو بعدی نمایش داد. بنابراین، لازم است که برای هر کلمه، عمل کاهش ابعاد را انجام دهیم. یکی از محبوب‌ترین روش‌ها برای این کار روش PCA است. رویکرد کلی آن این است که ترکیب‌های خطی از ویژگی‌های مختلف داده را جستجو می‌کند و ویژگی‌هایی را انتخاب می‌کند که بیشترین واریانس را در داده‌ها پوشش دهند. برای استفاده از PCA، از ماژولی با همین نام در کتابخانه Sklearn استفاده می‌کنیم و به عنوان ورودی تعداد کامپونت‌ها را ۲ می‌دهیم و سپس داده‌ها را تبدیل به فضای دو بعدی می‌کنیم. حالا برای هر جفت کلمه، مقدار embedding کاهش‌یافته آن‌ها را حساب کرده و از هم کم می‌کنیم تا بردارهای تفاضل آن‌ها بدست آید. سپس بردار تفاضل‌ها را همراه با خود کلمات در فضای دو بعدی جانمایی می‌کنیم، که در شکل زیر نتیجه آورده شده است:



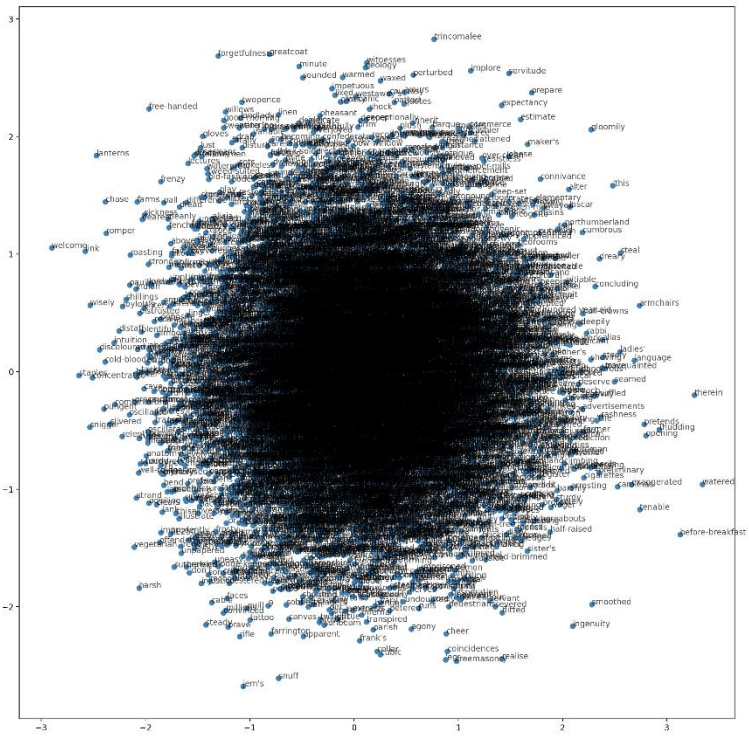
تحلیل این تصاویر:

تحلیل این تصاویر:

احتمالاً این نمودار دقیقاً شبیه چیزی که انتظار داریم نخواهد بود، به این معنا که مثلاً فاصله پادشاه با مرد دقیقاً اندازه فاصله ملکه با زن نباشد. دو عامل می‌توانند علت این امر باشند:

- اندازه کورپوس کم است. در حالی که مدل‌های embedding که استفاده می‌شوند روی چندین میلیون کلمه آموزش می‌بینند و نباید انتظار داشته باشیم که embedding ما بر روی دیتاست خیلی کوچک، کلمات را به صورت عالی تفکیک کند.
- علاوه بر کوچک بودن کورپوس، نوع متن شکسپیر نیز ادبی است. این به این معناست که کلمات خیلی کم تکرار می‌شوند نسبت به متن‌های رایج. این باعث می‌شود مدل سخت‌ترین context مشابه را برای کلمات پیدا کند، و این مشکل کوچک بودن داده را وخیم‌تر می‌کند.

در نهایت، embedding کل کلمات را در فضای دو بعدی نمایش داده‌ایم که در ادامه آورده شده است:



از این نمودار نیز می‌توان استنباط کرد که مدل به خوبی کلمات را تفکیک نکرده و کلمات بیشتر در گوشه‌ها متمرکز شده‌اند. اگر مدل بهتر عمل می‌کرد، احتمالاً پراکندگی بیشتری داشتیم، که علت ضعف آن در موارد قبلی ذکر شده است.

```python
# Create a figure and axis object for plotting
fig, ax = plt.subplots(figsize=(16, 16))

# Print the number of embeddings
print(len(embeddings))

# Scatter plot each word vector
for index in range(len(embeddings)):
    plt.scatter(*word_vectors_pca[index, :], color='steelblue')  # Scatter plot the word vector
    plt.annotate(int_to_vocab[index], (word_vectors_pca[index, 0], word_vectors_pca[index, 1]), alpha=0.7)  # Annotate each point with its corresponding word

# Display the plot
plt.show()
```