NLP Assignment_1_Report

20201036 Dohoon Kim(김도훈)

Problem 1. Simple Mathematics with Word2Vec

#The point of this problem is to complete the functions 'word_analogy_with_vector' and 'get_cosine_similarity'.

#Before doing this, we should normalize each vector. To do this, I define new function, 'normalize_vector'

```
def normalize_vector(v):
    norm = np.sqrt(np.sum(v**2))
    return v / norm
```

#This is a structure of the function. To use L2 norm, I define variable 'norm' as square root of sum of square of every value in the vector. In this case 'numpy' library is used, to operate arithmetic such as 'sqrt' and 'sum'. As a result, the size of the normalized vector becomes 1.

```
x_1_vector = model['king']
print(np.sum(x_1_vector**2))
x_1_vector_normalized = normalize_vector(model['king'])
np.sum(x_1_vector_normalized**2)
```

```
49.112144
1.0
```

#There are several reasons for normalization.

#First, in the process of getting similarity, we need to focus direction of vectors, not size. By unifying the sizes of all vectors, we can easily handle the direction.

#Second, using normalized vectors simplifies the calculation of cosine similarity since dot product of normalized vectors has same value as cosine similarity of it.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

#It makes sense when looking at this formula because sum of square in the vectors is 1, we can ignore a denominator.

#Word Analogy with Vector

```
def word_analogy_with_vector(model, x_1, x_2, y_1):

    x_1_vector = normalize_vector(model[x_1])
    x_2_vector = normalize_vector(model[x_2])
    y_1_vector = normalize_vector(model[y_1])
    y_2_vector = (x_2_vector - x_1_vector) + y_1_vector
    y_2_vector_normalized = normalize_vector(y_2_vector)

    return y_2_vector_normalized
```

#The objective of this function is to get normalized vector of y_2 which corresponds to x_1 → x_2 == y_1 → y_2.

It means that: x_2 – x_1 == y_2 – y_1 // y_2 == (x_2 – x_1) + y_1

#First, normalize word vectors in model. In model, there are 300-dimensional word vectors from 'glove-wiki-gigaword-300'.

#Second, get y_2 vector and normalize it and return.

#Next step is checking whether function works well.

```
result_vector = word_analogy_with_vector(model, 'korea', 'seoul', 'japan')
print('result vector is ', result_vector)
assert isinstance(result_vector, np.ndarray), "Output of the function has to be np.ndarray"
a = normalize_vector(model['tokyo'])
print(np.dot(result_vector, a))
b = model.most_similar(result_vector)
print(b)
c = model.most_similar(['seoul', 'japan'], negative=['korea'])
print(c)
```

#In variable 'result_vector', there is y_2 vector when x_1('korea'), x_2('seoul') and y_1('japan') is given. When we print 'result_vector', 300-dimensional vector will be shown.

#The third line serves two purposes. The first is to check if 'result_vector' is of type 'np.ndarray'. The 'isinstance' function checks if the object given as the first argument is an instance of the type provided as the second argument. The second purpose is for debugging. The 'assert' statement causes an 'AssertionError' to be raised when 'isinstance' returns False. This allows type-related issues to be easily identified and errors to be quickly corrected.

#After verifying 'result_vector' is appropriate type for arithmetic, now it is time to check the function returns reasonable output.

#At first, consider likely most similar word to result vector. By human's nature, we can easily infer that it is 'tokyo'. Then get the vector of 'tokyo' in model and normalize it.

#Second, operate dot product of result vector and 'tokyo' and compare the output of operation: model.most_similar(result_vector).

#Third, to check result_vector well reflects the y_2 vector, put the words corresponding to x_1, x_2, and y_1 directly into the most_similar function.

#The result is as follows.

```
0.82680357
[('tokyo', 0.8268035650253296), ('japan', 0.7188869118690491), ('seoul', 0.6656908988952637), ('
[('tokyo', 0.8268035650253296), ('osaka', 0.6480785012245178), ('japanese', 0.6473483443260193),
```

#Cosine similarity of two vector is equal with dot product of it, after we normalize each vector.

```
from numpy import dot

def get_cosine_similarity(model, x, y):
  vector_x = normalize_vector(model[x])
  vector_y = normalize_vector(model[y])

  similarity = dot(vector_x, vector_y)
  return similarity
word_a = 'seoul'
word_b = 'korea'

similarity = get_cosine_similarity(model, word_a, word_b)
print(similarity)
assert -1 <= similarity <= 1, "Similarity has to be between -1 and 1"

print('gensim library result:', model.similarity(word_a, word_b))
```

```
0.69181466
gensim library result: 0.69181466
```

#In this case, when seeing result of dot product and output of function: similarity (which is in 'gensim' library) it has exactly same value.

Problem 2. Find Most Similar Words

#Word2Vec is a model used to learn vector representations of words. The model maps the meanings of words to vector spaces, allowing them to numerically represent semantic relationships between words. The Word2Vec model can be learned in two main ways: Continuous Bag of Words (CBOW) and Skip-Gram models (but mainly using Skip-Gram).

#The CBOW model predicts the target word based on the context words. For example, the CBOW model's goal is to predict the words that will go into the blank in the sentence "The cats on the ____." The model averages out the vectors of the surrounding words to form a single hidden layer, through which it predicts the target word.

#The Skip-Gram model predicts surrounding words based on a target word. For example, given the word "cat," the goal is to predict surrounding words such as "The," "sits," "on," and "the." The Skip-Gram model predicts the likelihood of existence of surrounding words using vectors in the target word.

#The Word2Vec model's learning process is as follows:

#1. Initialize the vector of the word at random.

#2. Select context and target words from the training dataset.

#3. Use the current word vector to predict the target word and calculate the difference (error) from the actual target word.

#4. Update word vectors to reduce errors. This process usually uses backpropagation and Gradient Descent.

#5. Repeat this process over the entire dataset to gradually optimize the word vector.

#The following illustrates some interesting examples of similar words found using the Word2Vec model.

#1. football - soccer

```
[('soccer', 0.7682591676712036),
 ('basketball', 0.7341024279594421
 ('league', 0.6599653959274292),
 ('baseball', 0.6479504704475403),
 ('rugby', 0.6429778933525085),
 ('hockey', 0.6413834095001221),
 ('club', 0.6326252222061157),
 ('team', 0.6183146238327026),
 ('teams', 0.5992820858955383),
 ('player', 0.5946673154830933)]
```

#"Football" and "soccer" are regarded as similar in Word2Vec because they both refer to the same sport but are used in different regions. In the United States and Canada, it's commonly referred to as "soccer," while in most other parts of the world, it's known as "football." The model learns from the contexts in which these terms appear, recognizing their semantic relationship within the sports domain.

#2. car - vehicle

```
[('cars', 0.7827162146568298),
 ('vehicle', 0.7655367851257324),
 ('truck', 0.7350621819496155),
 ('driver', 0.7114784717559814),
 ('driving', 0.6442225575447083),
 ('vehicles', 0.6328005194664001),
 ('motorcycle', 0.6022513508796692)
 ('automobile', 0.595572829246521),
 ('parked', 0.5910030603408813),
 ('drivers', 0.5778359770774841)]
```

Word2Vec considers "car" and "vehicle" similar because a car is a type of vehicle. During training, the model learns that these words share a broader category and are often mentioned together in discussions about transportation, automotive industry, or vehicle engineering.

#3. sushi - sashimi

```
[('sashimi', 0.7076607346534729),
 ('restaurant', 0.5439533591270447),
 ('restaurants', 0.5290504097938538),
 ('chefs', 0.525417685508728),
 ('diners', 0.5239505171775818),
 ('seafood', 0.5159598588943481),
 ('chef', 0.514190673828125),
 ('steak', 0.5131548047065735),
 ('sandwich', 0.5088735222816467),
 ('tempura', 0.5042536854743958)]
```

Word2Vec considers "sushi" and "sashimi" similar because they both belong to Japanese cuisine and are popular food items. During training, the model learns that these words often co-occur in discussions about Japanese food, restaurants, or recipes, leading to their semantic similarity.

#4. computer - laptop

```
[('computers', 0.8248152732849121),
 ('software', 0.7334420084953308),
 ('pc', 0.6240139603614807),
 ('technology', 0.6198545098304749),
 ('computing', 0.6178765296936035),
 ('laptop', 0.5955509543418884),
 ('internet', 0.5857782363891602),
 ('ibm', 0.5825320482254028),
 ('systems', 0.5744993686676025),
 ('hardware', 0.5728795528411865)]
```

Word2Vec considers "computer" and "laptop" similar because they both belong to the category of computing devices. During training, the model learns that these words often co-occur in contexts discussing technology, computing, or discussions about different types of personal computing devices.

#5. korea - seoul

```
[('korean', 0.7692005038261414),
 ('pyongyang', 0.7274450659751892)
 ('seoul', 0.691814661026001),
 ('dprk', 0.6857666373252869),
 ('koreans', 0.6707963943481445),
 ('south', 0.6654611825942993),
 ('north', 0.6483291387557983),
```

#Words that are semantically related or have similar contexts tend to have similar vector representations in Word2Vec space. Since "Korea" and "Seoul" frequently appear together with similar surrounding words like "country," "capital," "East Asia," or "culture," Word2Vec assigns them similar vector representations.

#6. coffee - espresso

```
[('tea', 0.6692111492156982),
 ('espresso', 0.5902243852615356),
 ('cocoa', 0.5677695274353027),
 ('starbucks', 0.5497761964797974),
 ('drinks', 0.5467038154602051),
 ('beans', 0.5411592125892639),
 ('drink', 0.540280818939209),
 ('beer', 0.528264582157135),
 ('sipping', 0.504896879196167),
 ('sugar', 0.5025252103805542)]
```

In Word2Vec, "coffee" and "espresso" are often considered similar because they both belong to the category of caffeinated beverages. During training, the model learns that these words frequently co-occur in contexts related to coffee shops.

Problem 3. Word Analogy

#I selected some interesting examples.

```
def analogy(model, x1, x2, y1):
  pp.pprint(model.most_similar([x2, y1], negative=[x1]))

analogy(model, 'english', 'milk', 'spanish')
analogy(model, 'milk', 'english', 'leche')
analogy(model, 'korea', 'dokdo', 'japan')
analogy(model, 'doctor', 'stethoscope', 'chef')
analogy(model, 'apple', 'fruit', 'carrot')
analogy(model, 'fruit', 'apple', 'vegetable')
analogy(model, 'fruit', 'grape', 'vegetable')
```

#First case is about translation. I wondered that ‘analogy’ function can work as translator of dictionary itself. So, I put some translate problem in it: What’s milk in Spanish? The result is.

```
[('dairy', 0.49474677443504333),
 ('soy', 0.4567534625530243),
 ('tainted', 0.43346109986305237),
 ('yogurt', 0.43176794052124023),
 ('chocolate', 0.4305379092693329),
 ('meat', 0.42809098958969116),
 ('cheese', 0.4228046238422394),
 ('butter', 0.41646599769592285),
 ('cream', 0.39644482731819153),
 ('sugar', 0.3922494649887085)]
```

#I expected result contains word 'leche', which is milk in Spanish, but it doesn't work.

#In second case, I modified first case, to make the function infer which linguistic domain 'leche' belongs. The result is.

```
[('fluent', 0.3713277280330658),
 ('clergyman', 0.342074990272522),
 ('linguist', 0.3387399911880493),
 ('welsh', 0.3386596143245697),
 ('gaelic', 0.3386145830154419),
 ('spanish', 0.3369142413139343),
 ('esperanto', 0.3337211608886719),
 ('arabic', 0.3316267132759094),
 ('tagalog', 0.32203537225723267),
 ('dictionary', 0.32192251086235046)]
```

The result shows us that it focuses on languages but expect result: 'spanish' is not most similar and has a few similarities with inputs.

#After experimenting translation problem, the third case is about controversial global relationship. The reason this case was interesting was that I was wondering if the function would output 'takeshima' as a result. The result is.

```
[('takeshima', 0.6005429029464722),
 ('senkaku', 0.5756986737251282),
 ('diaoyu', 0.5645158290863037),
 ('islets', 0.5059949159622192),
 ('tiaoyutai', 0.4854239523410797),
 ('kuril', 0.4674299359321594),
 ('tiaoyutais', 0.4620523750782013),
 ('kurils', 0.439883828163147),
 ('senkakus', 0.4196641743183136),
 ('hokkaidō', 0.4099167585372925)]
```

#The most similar word is ‘takeshima’ as I expected. In this case, it is more like a translation case. It is an analysis of how an object called Dokdo in Korea is called in Japan. Maybe lots of Korean will be angry if this function outputs 'takeshima'. On the other hand, second similar word ‘senkaku’ is a place that where Japan is experiencing territorial disputes. It reads the relationship between a country and the place where it suffers territorial disputes.

#The fourth case is asking function which tool chef uses as doctor uses stethoscope. We anticipate Word2Vec to understand the relationship between a doctor and their tool and extend it to a chef and their tool. The result is.

```
[('bistro', 0.4800702631473541),
 ('restaurateur', 0.42922040820121765),
 ('chefs', 0.4275852143764496),
 ('brasserie', 0.41301387548446655),
 ('boulud', 0.4107045531272888),
 ('menteur', 0.4000392556190491),
 ('pastry', 0.3974584639072418),
 ('batali', 0.39573559165000916),
 ('cutlery', 0.3818184733390808),
 ('ducasse', 0.37873271107673645)]
```

#The words ‘brasserie’ or ‘cutlery’ only have meaning of tool but are not exactly corresponding to tool that chef uses. It seems difficult to specify a word because the range of x1, y1 is larger than that of x2.

#Looking at the above results, I wondered if x2 is a subcategory of x1, then the function could properly output subcategories of y1.

#To confirm, experiments were conducted from the opposite case: "apple" is to "fruit" as "carrot" is to what? The result is.

```
[('carrots', 0.5681754946708679),
 ('fruits', 0.5261087417602539),
 ('vegetables', 0.5018566250801086),
 ('celery', 0.4958705008029938),
 ('berries', 0.48848971724510193),
 ('vegetable', 0.47271883487701416),
 ('watermelon', 0.4597003757953644),
 ('cucumber', 0.4533902704715729),
 ('edible', 0.4419475495815277),
 ('mushrooms', 0.4363740086555481)]
```

#’Vegetable’, expected result is on the third place. It makes sense.

#Then the next case is opposite of it. The result is.

```
[('macintosh', 0.47241920232772827),
 ('iphone', 0.4642904996871948),
 ('ipad', 0.4421779215335846),
 ('microsoft', 0.4420732259750366),
 ('intel', 0.43717560172080994),
 ('ipod', 0.42306920886039734),
 ('google', 0.4204891324043274),
 ('pc', 0.4045141637325287),
 ('ibm', 0.39418795704841614),
 ('software', 0.3887002170085907)]
```

#Totally different from what I expected. It recognized 'Apple' as an IT company, not a fruit, and listed all the words related to Apple Inc. There are cases where words are used in completely different meanings depending on the situation, and this case was interesting because it seemed to show such an example accurately.

#Returning to the original problem, I changed the type of fruit from apple to grape and printed out the results.

```
[('tomato', 0.49749475717544556),
 ('plantings', 0.4648124275917053),
 ('canola', 0.46154969930648804),
 ('grapes', 0.4306953549385071),
 ('varieties', 0.42661282420158386),
 ('beet', 0.4265314042568207),
 ('soy', 0.4253773093223572),
 ('cabbage', 0.4128795564174652),
 ('varietals', 0.41218307614326477),
 ('soybean', 0.4089605510234833)]
```

#Although the similarity is low, it was significant because subcategory words of vegetables were listed. Since the process is inferring individual items from top categories, it is expected that similarity is lower than opposite case which is to deduce a higher category based on it when a lower category is given.

Problem 4. Visualize Word Vectors

#In this problem, I selected words indicating perfume ingredients since perfume is my personal interest.

#At first, I got information about ingredients from article:

https://www.fragrancex.com/blog/perfume-ingredients/

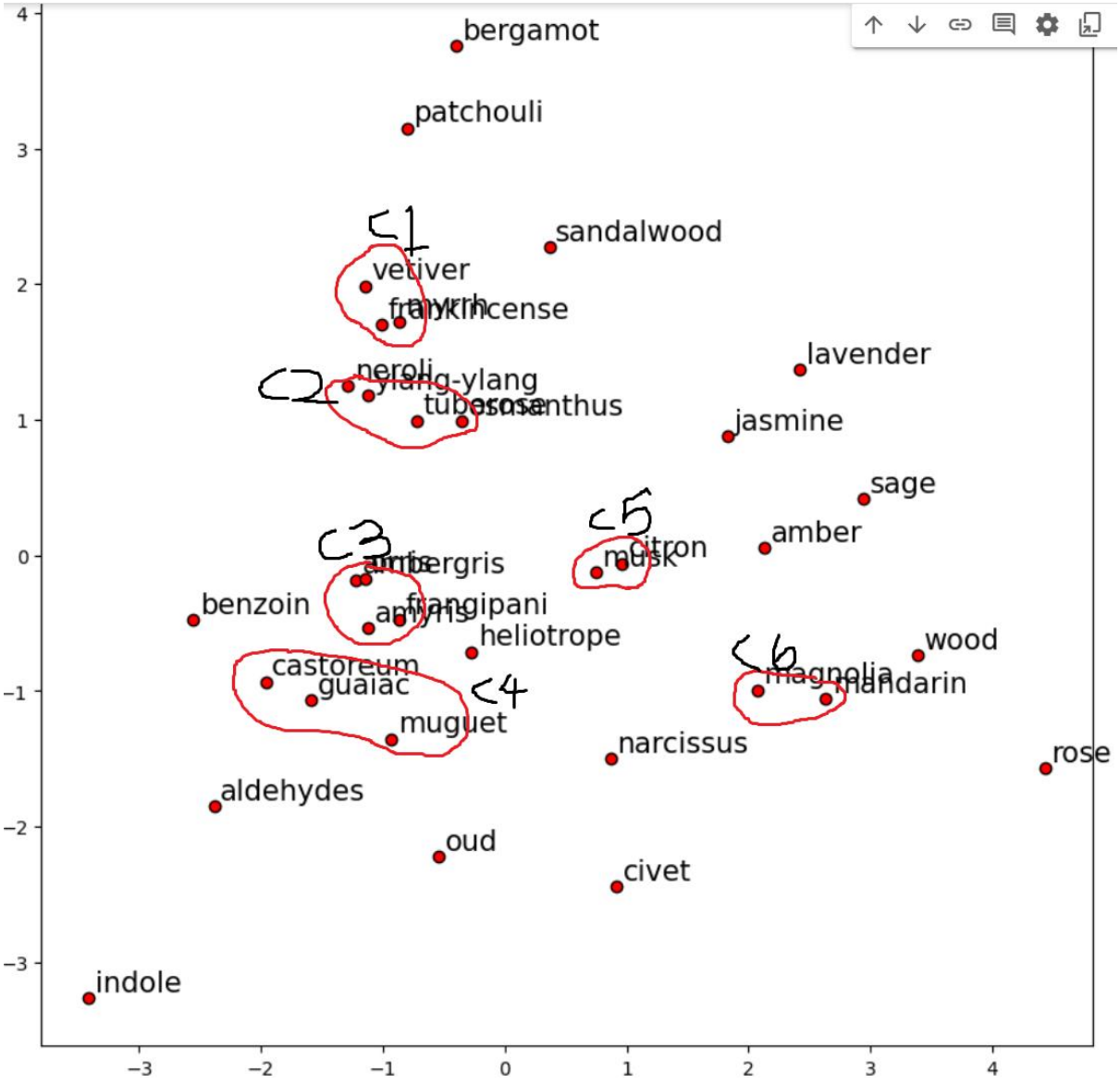
and made list of names of each ingredient.

#However, there are some rarely used ingredients that the model doesn't include

(e.g. agrumen, ambrette, labdanum, etc.). Therefore, I modified list by eliminating them and 34 left.

```
# Select word list of your own interests
word_list = ['aldehydes','amber','ambergris', 'amyris',
             'benzoin','bergamot','castoreum','citron',
             'civet','sage','frangipani','frankincense',
             'guaiac','wood','heliotrope','indole','jasmine',
             'lavender','magnolia','mandarin','muguet','musk',
             'myrrh','narcissus','neroli','orris','osmanthus',
             'oud','patchouli','rose','sandalwood','tuberose',
             'vetiver','ylang-ylang'
]

print(len(word_list))
display_pca_scatterplot(model, word_list)
```

```
34
```



#This is the result of mapping words.

#How Words are Located in 2D Space and Clustered

#PCA is a way to reduce dimensions while maximizing the variance of high-dimensional data. Through this process, similar words can be placed in similar location in 2D space. There are mainly six clusters in 2D space. All of clusters have less elements than four. Words in each cluster have more to do with than words from outside. For example, in C1, "vetiver", "frankincense", "myrrh" are all woody, earthy, and intense. While in C2, words "neroli," "ylang-ylang", "tuberose", and "osmanthus" are likely placed in close to each other, since they are words that indicate ingredients that perform floral, sweet, and fresh scent.

#Suitability of Word Clustering and Unexpected Examples

#There is reasonable relationship between each word in same cluster, but this clustering still has unexpected drawbacks. For example, though "bergamot" and "mandarin" are all citruses and similar scent, the model did not catch their similarities. Another case is remoted location of 'rose'. It was expected that the 'rose' assigned near of floral ingredients (e.g. neroli, tuberose, etc.). Since the pre-trained model 'glove-wiki-gigaword-300' is not especially trained to cover some fragrance problem, it is likely that detailed analysis is not performed.

Problem 5. Train New Word2Vec

#To handle drawbacks found previous problem and make more precise classification of ingredients, I will use another model which is trained by another text file.

#At first, to accomplish the purpose, I found text file related to perfume in 'kaggle'.

The url is. https://www.kaggle.com/datasets/nandini1999/perfume-recommendation-dataset

#This is dataset of information of more than 2000 perfumes. It was originally csv file, which has columns such as:

1. Name = Name of the perfume
2. Brand = Brand or Company to which the perfume belongs to
3. Description = Some text describing the feel and features of the perfume.
4. Notes = A list of fragrance notes present in the perfume
5. Image URL = URL of the perfume image

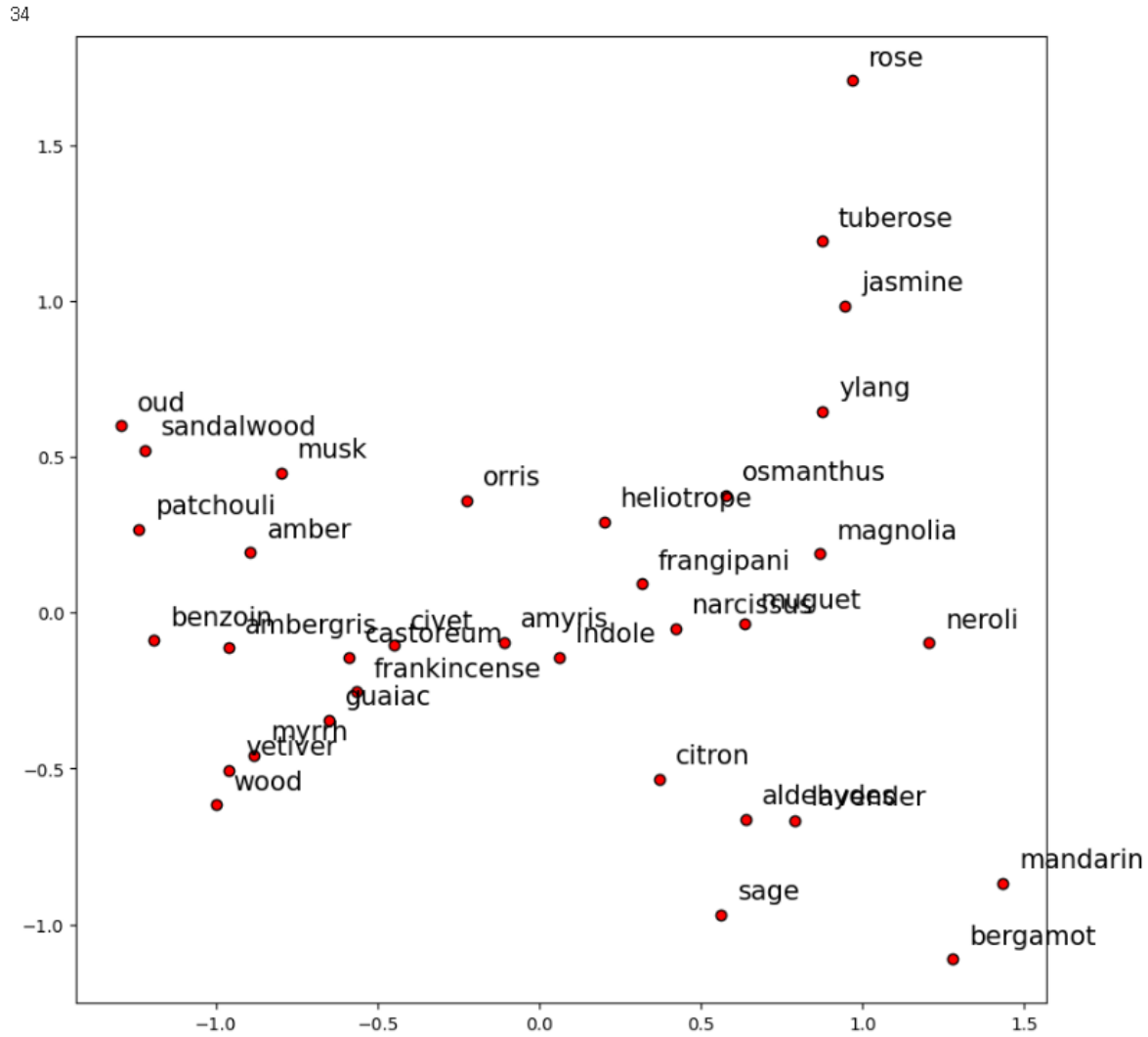
#However, to train the model with text file, which is composed by sentences, I only extracted the information from the third column: description. Then I add the article that I used in previous problem and saved it as 'perfume_decription.txt'.

```
your_text_fn = '/content/perfume_description.txt' # Enter your text file name here

#with open(your_text_fn, 'r') as f:
with open(your_text_fn, 'r', encoding='utf-8', errors='ignore') as f:
  strings = f.readlines()
corpus = make_tokenized_corpus(strings)
'''
```

```
model1 = Word2Vec(sentences=corpus,
                  vector_size=100,
                  window=5,
                  min_count=2,
                  sg=1,
                  negative=5,
                  workers=4)

model1 = model1.wv # To match with pr
```

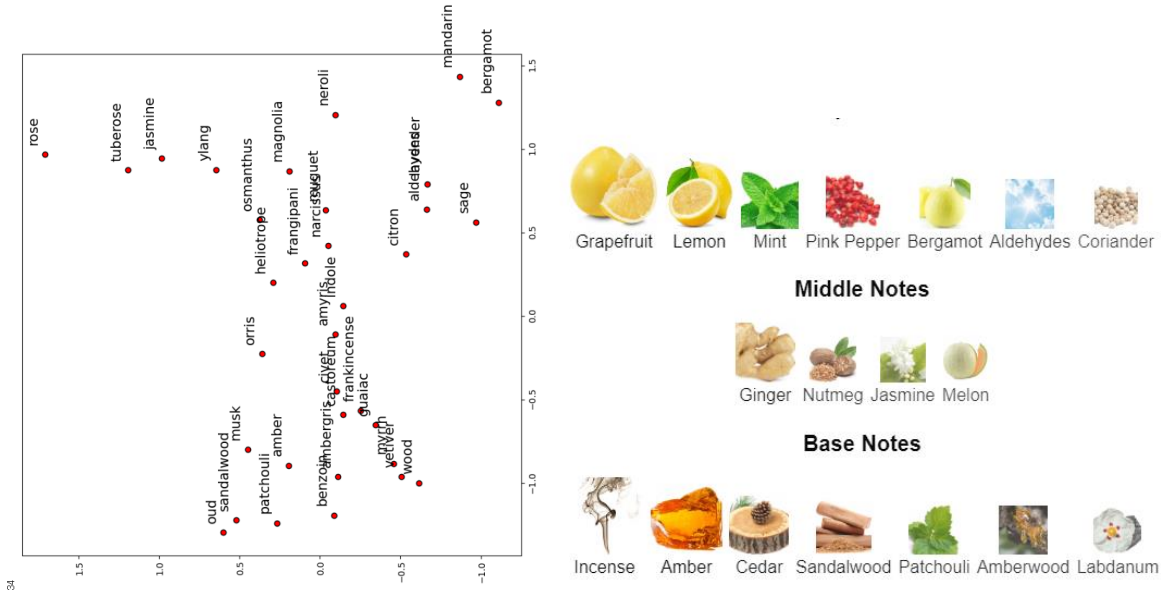
#Since amount of word from dataset is much smaller than "glove-wiki-gigaword-300", I reduced some parameters of Word2ved such as 'min_count' or 'negative'.

#The work in question 4 was carried out in the same way. The result is



#The first interesting difference derived from using another dataset is that it seems to understand each feature of ingredients. Looking at x-axis, the aroma of the raw material

becomes lighter as the x value increases. On the left side of the graph, some heavy material such as 'oud', 'amber', 'sandalwood' are located while on the right side, there are light ingredients which perform floral or citrus scent. Usually, heavy materials are used in base note and light materials are used in top note. So, If you rotate the graph counterclockwise 90 degree, you will find a similarity to the picture on the right below.



#The second interesting thing about new model is that it prints out more similar words when typing word of an ingredient in function 'most_similar'.

```
target_word = 'bergamot' # Enter your word string here
# check the word is in the vocabulary of the model
assert model.has_index_for(target_word), f"The selecte
model.most_similar(target_word)
```

```
[('chamomile', 0.4372253417968875),
 ('patchouli', 0.4317514896392822),
 ('lavender', 0.405318021774292),
 ('neroli', 0.40015318989753723),
 ('spearmint', 0.3657273650169372 6),
 ('corallina', 0.3653071820735931 4),
 ('horndon', 0.3649813234806061),
 ('lidcombe', 0.364951103925704 96),
 ('broadbeach', 0.3599428236484527 6),
 ('thymol', 0.3595306873321533)]
```

```
target_word = 'bergamot' # Enter your word string here
# check the word is in the vocabulary of the model
assert model1.has_index_for(target_word), f"The selecte
model1.most_similar(target_word)
```

```
[('grapefruit', 0.9601817727088928),
 ('lemon', 0.9556978940963745),
 ('mandarin', 0.9434173703193665),
 ('lime', 0.942919135093689),
 ('petitgrain', 0.9335858225822449),
 ('zesty', 0.9315468072891235),
 ('blackcurrant', 0.9287005066871643),
 ('tangy', 0.9284946322441101),
 ('pink', 0.9220525026321411),
 ('basil', 0.9212995171546936)]
```

#Left image is output of the function when using previous model and left is that of new model. In this case, since 'grapefruit', 'lemon', 'mandarin', and 'lime' is all orange-shaped fruits, it seems that new model better understands characteristics of specific words.

#The third interesting thing is that cosine similarity is much higher than existing model. When comparing result, the similarity was higher in left case. To confirm, the similarity of words not related to perfume was obtained and the results were compared.

```
target_word = 'king' # Enter your word string here
# check the word is in the vocabulary of the model
assert model.has_index_for(target_word), f"The sele
model.most_similar(target_word)
```

```
[('queen', 0.6336469054222107),
 ('prince', 0.6196622848510742),
 ('monarch', 0.5899620652198792),
 ('kingdom', 0.5791266560554504),
 ('throne', 0.5606487989425659),
 ('ii', 0.5562329292297363),
 ('iii', 0.5503199100494385),
 ('crown', 0.5224862694740295),
 ('reign', 0.5217353701591492),
 ('kings', 0.5066401958465576)]
```

```
target_word = 'king' # Enter your word s
# check the word is in the vocabulary of
assert model1.has_index_for(target_word)
model1.most_similar(target_word)
```

```
[('child', 0.9845563769340515),
 ('1920s', 0.9834492206573486),
 ('isabey', 0.9826719164848328),
 ('angels', 0.9824622869491577),
 ('wrote', 0.9824082255363464),
 ('success', 0.9820895195007324),
 ('londons', 0.9816744923591614),
 ('mother', 0.9813142418861389),
 ('novels', 0.9811524748802185),
 ('grew', 0.9810875654220581)]
```

#When putting 'king' to each function, the result is same. It means that no matter which word is used, the function outputs high similarity.

#The fourth interesting example is related to the case of word 'king'. When word has nothing to do with perfume, the model outputs less accurate result. We can easily judge that 'king' is not like 'child' or '1920s' as having more than 98% similarity. Maybe this happens because of lack of words in dataset.

```
print(len(model1))
```

```
12313
```

#Only 12313 words are in the model, and it is much smaller than original. That is reason why this kind of problem occurs. To improve it, fine tuning the pre-existing model with given text is needed.

#The last interesting thing is that it performs better in term of estimating scent.

```
print(analogy(model, 'neroli', 'floral', 'sandalwood'))
print(analogy(model1, 'neroli', 'floral', 'sandalwood'))
```

```
[('flowers', 0.4777246415615082),
 ('flower', 0.461261510848999),
 ('adorned', 0.45864105224609375),
 ('ornaments', 0.4481959342956543),
 ('incense', 0.4429246783256531),
 ('decorative', 0.4398514926433563),
 ('bouquet', 0.4307063817977905),
 ('embroidered', 0.4271883070468902 6),
 ('lace', 0.42621755599975586),
 ('carvings', 0.4248176813125610 4)]
None
[('patchouli', 0.69670569896698),
 ('vanilla', 0.6926760673522949),
 ('amber', 0.6913019418716431),
 ('benzoin', 0.691146969795227),
 ('tonka', 0.6843622326850891),
 ('cedarwood', 0.6829566359519958),
 ('animalic', 0.6752651929855347),
 ('base', 0.6719072461128235),
 ('labdanum', 0.6650844216346741),
 ('bean', 0.6644149422645569)]
None
```

#From the bottom results, we see that this model can better extract scent information. Just as 'floral' refers to the scent of 'neroli', 'animalic' refers to the scent of 'sandalwood'. Other examples are also related to the scent of 'sandalwood'.