Hee Ji Park
CSCI544 (NLP)
10.01.2021

Report – HW2 (CSCI544)

0. Information

Python version	3.6.12
Jupyter notebook version	6.1.4
Package and libraries	import pandas as pd import numpy as np import nltk import warnings from sklearn.model_selection import train_test_split from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score from sklearn.feature_extraction.text import TfidfVectorizer from sklearn.linear_model import Perceptron from sklearn.svm import LinearSVC from nltk.corpus import stopwords # remove the stop words using NLTK package import contractions from bs4 import BeautifulSoup import rec from gensim.test.utils import common_texts from gensim.models import Word2Vec from nltk.tokenize import word_tokenize, sent_tokenize from nltk.stem import WordNetLemmatizer import torch from torch import nn from torch import optim from torch.utils.data import TensorDataset, DataLoader from collections import Counter import itertools

1. Compare vectors generated by myself and the pretrained model. Check the semantic similarities for some examples. – 2 models

	My own Word2Vec	google-news-300
most similar (positive(cat + dog), topn=1)	[('swear', 0.7196011543273926)] [('swear', 0.6660094261169434)]	[('puppy', 0.8089798092842102)] [('puppy', 0.7729379534721375)]
excellent | outstanding?	0.6124733	0.55674857
most_similar('excellent')	[('exceptional', 0.6363705992698669), ('outstanding', 0.6124733090400696), ('affordable', 0.6097689270973206), ('superb', 0.5697858333587646), ('acceptable', 0.5670941472053528), ('wonderful', 0.5664124488830566), ('adequate', 0.539593517780304), ('A+', 0.5354660749435425), ('awesome', 0.5353838205337524), ('amazingly', 0.5322791337966919)]	[('terrific', 0.7409726977348328), ('superb', 0.7062715888023376), ('exceptional', 0.681470513343811), ('fantastic', 0.6802847981452942), ('good', 0.6442928910255432), ('great', 0.6124600172042847), ('Excellent', 0.6091997623443604), ('impeccable', 0.5980967283248901), ('exemplary', 0.5959650278091431), ('marvelous', 0.582928478717804)]
most_similar('outstanding')	[('speedy', 0.723319947719574), ('responsive', 0.7164787650108337), ('confirmed', 0.702231764793396), ('Seller', 0.7006707191467285), ('A+', 0.6945520043373108), ('exceptional', 0.6799882054328918), ('unacceptable', 0.6744035482406616), ('speaking', 0.6743783950805664), ('Company', 0.6710595488548279), ('Europe', 0.6699382066726685)]	[('oustanding', 0.8012188673019409), ('Outstanding', 0.6041857600212097), ('exceptional', 0.6031844615936279), ('anchorman_Jason_Lezak', 0.5947381258010864), ('outsanding', 0.566262423992157), ('Stock_HEI', 0.5573362708091736), ('excellent', 0.556748628616333), ('Synplicity_FPGA_implementation', 0.5520347356796265), ('exemplary', 0.5467386245727539), ('W3_Awards_honors', 0.5172522068023682)]
Similarity (chair & desk?)	0.48516065	0.3149568
Similarity soap ?	[('soapy', 0.7130864858627319), ('detergent', 0.6908825635910034), ('sponge', 0.6703446507453918), ('dishwashing', 0.6624318361282349), ('wiping', 0.6428627967834473), ('bleach', 0.6349363327026367), ('rinsed', 0.6319710612297058), ('rinsing', 0.6148936748504639), ('scrubbed', 0.6108755469322205), ('deposits', 0.6097919344902039)]	[('soaps', 0.7613304257392883), ('Soap', 0.6950218677520752), ('Colgate_Palmolive_toothpaste', 0.6457595229148865), ('detergent', 0.5981624126434326), ('Colgate_toothpaste_Palmolive', 0.5842004418373108), ('antiseptic_soaps', 0.5792285203933716), ('soapy', 0.5768905878067017), ('Laundry_detergent', 0.5718933939933777), ('Tide_detergent_Ivory', 0.5658919215202332), ('Unilever_ULVR_LN', 0.5604559779

2. What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

[Report] => The Word2Vec model I trained uses Amazon review data, so there are many terms related to the product. Objectively, the Word2Vec model using Google News data seems better to encode semantic similarities between words better.

3. Using two Word2Vec features, train a perceptron and SVM. Use the average Word2Vec vectors for each review as the input feature. – 4 models (+2 models)

Perceptron	Accuracy	Precision	Recall	F1-score
My Own Word2Vec	0.750125	0.685045	0.938765	0.792083
google-news-300	0.776527	0.852174	0.676862	0.754468
TF-IDF	0.800000	0.804757	0.800000	0.802372

SVM	Accuracy	Precision	Recall	F1-score
My Own Word2Vec	0.817226	0.829517	0.804938	0.817043
google-news-300	0.807808	0.824987	0.788357	0.806256
TF-IDF	0.846000	0.855634	0.837931	0.846690

[Report] => Among the three feature types, the model using TF-IDF as a feature showed the best performance. Comparing my own word2vec and word2vec-google-news, the performance difference between the two is very small, so the performance effect will vary depending on what kind of data the training data is.

4-a. [Feedforward Neural Networks] Using the average Word2Vec vectors, report accuracy values on the testing split for MLP model for each of the binary and ternary classification cases. – 4 models

- Use 'nn.CrossEntropyLoss()' as a loss function
- Use 'optim.Adam(model.parameters(), lr=0.001) as a optimizer

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	82%	65%
google-news-300	78%	60%

4-b. [Feedforward Neural Networks] Using the concatenated the first 10 Word2Vec vectors for each review as the input feature, report accuracy values on the testing split for MLP model for each of the binary and ternary classification cases. – 4 models

- Use 'nn.CrossEntropyLoss()' as a loss function
- Use 'optim.Adam(model.parameters(), lr=0.001) as a optimizer

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	73%	54%
google-news-300	72%	52%

[Report] => Looking at the results of Feedforward Neural Networks, overall, the model with the features that used my own word2vec vectors as an input has better performance.

4-c. Compare the accuracy values for binary classification.

Binary Classification	TF-IDF	the average Word2Vec vectors (My own word2vec)	the average Word2Vec vectors (google-news-300)	the concatenated the first 10 Word2Vec vectors (My own word2vec)	the concatenated the first 10 Word2Vec vectors (google-news-300)
Perceptron	80%	75%	77%	x	x
SVM	84%	80%	80%	x	x
Feedforward Neural Networks	x	82%	78%	73%	72%

[Report] => The SVM model using TF-IDF features has the best performance compared to the other models. The Feedforward Neural Networks model using the average my own word2vec vectors has the second-best performance.

5. I have computational resource limitations, especially memory issue, so I had to set epoch=3.

5-a. [Simple Recurrent Neural Networks] (Limit the maximum length to 50) – 4 models

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	66%	52%
google-news-300	64%	53%

5-b. [A gated recurrent unit cell (GRU)] (Limit the maximum length to 50) – 4 models

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	64%	53%
google-news-300	66%	51%

[Report] => The performance of RNN and GRU is not significantly different. However, I have computational resource limitations, especially memory issue, so I had to set epoch to 3. (epoch=3). If I have resources, I would like to increase the amount of epoch, which might get different results.

Hee Ji Park (4090715830)

- CSCI544 : Homewoek Assignment 2
- Python version: 3.6.12
- Jupyter notebook version : 6.1.4

1. Dataset Generation

(0) import package and libraries

In [1]:

```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score # For the result
from sklearn.feature_extraction.text import TfidfVectorizer # For TF-IDF
from sklearn.linear_model import Perceptron # For perceptron
from sklearn.svm import LinearSVC # For SVM
# For data cleaning and preprocessing
from nltk.corpus import stopwords # remove the stop words using NLTK package
import contractions
from bs4 import BeautifulSoup
import re
# For Word2Vec
from gensim.test.utils import common_texts
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.stem import WordNetLemmatizer
# For FNN and RNN
import torch
from torch import nn
from torch import optim
from torch.utils.data import TensorDataset, DataLoader
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:₩Users₩gmlwl₩AppData₩Roaming₩nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

(1) Load the dataset

In [2]:

```
df = pd.read_csv('data.tsv', sep='₩t', error_bad_lines=False) # read data
df.head()
```

```
b'Skipping line 16148: expected 15 fields, saw 22₩nSkipping line 20100: expected 15 fields, saw 22₩nSkipping line 45178: exp
ected 15 fields, saw 22₩nSkipping line 48700: expected 15 fields, saw 22₩nSkipping line 63331: expected 15 fields, saw 22₩n'
b'Skipping line 86053: expected 15 fields, saw 22₩nSkipping line 88858: expected 15 fields, saw 22₩nSkipping line 115017: ex
pected 15 fields, saw 22₩n'
b'Skipping line 137366: expected 15 fields, saw 22₩nSkipping line 139110: expected 15 fields, saw 22₩nSkipping line 165540:
expected 15 fields, saw 22₩nSkipping line 171813: expected 15 fields, saw 22₩n'
b'Skipping line 203723: expected 15 fields, saw 22₩nSkipping line 209366: expected 15 fields, saw 22₩nSkipping line 211310:
expected 15 fields, saw 22₩nSkipping line 246351: expected 15 fields, saw 22₩nSkipping line 252364: expected 15 fields, saw
22₩n'
b'Skipping line 267003: expected 15 fields, saw 22₩nSkipping line 268957: expected 15 fields, saw 22₩nSkipping line 303336:
expected 15 fields, saw 22₩nSkipping line 306021: expected 15 fields, saw 22₩nSkipping line 311569: expected 15 fields, saw
22₩nSkipping line 316767: expected 15 fields, saw 22₩nSkipping line 324009: expected 15 fields, saw 22₩n'
b'Skipping line 359107: expected 15 fields, saw 22₩nSkipping line 368367: expected 15 fields, saw 22₩nSkipping line 381180:
expected 15 fields, saw 22₩nSkipping line 390453: expected 15 fields, saw 22₩n'
b'Skipping line 412243: expected 15 fields, saw 22₩nSkipping line 419342: expected 15 fields, saw 22₩nSkipping line 457388:
expected 15 fields, saw 22₩n'
b'Skipping line 459935: expected 15 fields, saw 22₩nSkipping line 460167: expected 15 fields, saw 22₩nSkipping line 466460:
expected 15 fields, saw 22₩nSkipping line 500314: expected 15 fields, saw 22₩nSkipping line 500339: expected 15 fields, saw
22₩nSkipping line 505396: expected 15 fields, saw 22₩nSkipping line 507760: expected 15 fields, saw 22₩nSkipping line 51362
6: expected 15 fields, saw 22₩n'
b'Skipping line 527638: expected 15 fields, saw 22₩nSkipping line 534209: expected 15 fields, saw 22₩nSkipping line 535687:
expected 15 fields, saw 22₩nSkipping line 547671: expected 15 fields, saw 22₩nSkipping line 549054: expected 15 fields, saw
22₩n'
b'Skipping line 599929: expected 15 fields, saw 22₩nSkipping line 604776: expected 15 fields, saw 22₩nSkipping line 609937:
expected 15 fields, saw 22₩nSkipping line 632059: expected 15 fields, saw 22₩nSkipping line 638546: expected 15 fields, saw
22₩n'
b'Skipping line 665017: expected 15 fields, saw 22₩nSkipping line 677680: expected 15 fields, saw 22₩nSkipping line 684370:
expected 15 fields, saw 22₩nSkipping line 720217: expected 15 fields, saw 29₩n'
b'Skipping line 723240: expected 15 fields, saw 22₩nSkipping line 723433: expected 15 fields, saw 22₩nSkipping line 763891:
expected 15 fields, saw 22₩n'
b'Skipping line 800288: expected 15 fields, saw 22₩nSkipping line 802942: expected 15 fields, saw 22₩nSkipping line 803379:
expected 15 fields, saw 22₩nSkipping line 805122: expected 15 fields, saw 22₩nSkipping line 821899: expected 15 fields, saw
22₩nSkipping line 831707: expected 15 fields, saw 22₩nSkipping line 842829: expected 15 fields, saw 22₩nSkipping line 84360
4: expected 15 fields, saw 22₩n'
b'Skipping line 863904: expected 15 fields, saw 22₩nSkipping line 875655: expected 15 fields, saw 22₩nSkipping line 886796:
expected 15 fields, saw 22₩nSkipping line 892299: expected 15 fields, saw 22₩nSkipping line 902518: expected 15 fields, saw
22₩nSkipping line 903079: expected 15 fields, saw 22₩nSkipping line 912678: expected 15 fields, saw 22₩n'
b'Skipping line 932953: expected 15 fields, saw 22₩nSkipping line 936838: expected 15 fields, saw 22₩nSkipping line 937177:
expected 15 fields, saw 22₩nSkipping line 947695: expected 15 fields, saw 22₩nSkipping line 960713: expected 15 fields, saw
22₩nSkipping line 965225: expected 15 fields, saw 22₩nSkipping line 980776: expected 15 fields, saw 22₩n'
b'Skipping line 999318: expected 15 fields, saw 22₩nSkipping line 1007247: expected 15 fields, saw 22₩nSkipping line 101598
7: expected 15 fields, saw 22₩nSkipping line 1018984: expected 15 fields, saw 22₩nSkipping line 1028671: expected 15 fields,
saw 22₩n'
```

```
b'Skipping line 1063360: expected 15 fields, saw 22₩nSkipping line 1066195: expected 15 fields, saw 22₩nSkipping line 106657
8: expected 15 fields, saw 22₩nSkipping line 1066869: expected 15 fields, saw 22₩nSkipping line 1068809: expected 15 fields,
saw 22₩nSkipping line 1069505: expected 15 fields, saw 22₩nSkipping line 1087983: expected 15 fields, saw 22₩nSkipping line
1108184: expected 15 fields, saw 22₩n'
b'Skipping line 1118137: expected 15 fields, saw 22₩nSkipping line 1142723: expected 15 fields, saw 22₩nSkipping line 115249
2: expected 15 fields, saw 22₩nSkipping line 1156947: expected 15 fields, saw 22₩nSkipping line 1172563: expected 15 fields,
saw 22₩n'
b'Skipping line 1209254: expected 15 fields, saw 22₩nSkipping line 1212966: expected 15 fields, saw 22₩nSkipping line 123653
3: expected 15 fields, saw 22₩nSkipping line 1237598: expected 15 fields, saw 22₩n'
b'Skipping line 1273825: expected 15 fields, saw 22₩nSkipping line 1277898: expected 15 fields, saw 22₩nSkipping line 128365
4: expected 15 fields, saw 22₩nSkipping line 1286023: expected 15 fields, saw 22₩nSkipping line 1302038: expected 15 fields,
saw 22₩nSkipping line 1305179: expected 15 fields, saw 22₩n'
b'Skipping line 1326022: expected 15 fields, saw 22₩nSkipping line 1338120: expected 15 fields, saw 22₩nSkipping line 133850
3: expected 15 fields, saw 22₩nSkipping line 1338849: expected 15 fields, saw 22₩nSkipping line 1341513: expected 15 fields,
saw 22₩nSkipping line 1346493: expected 15 fields, saw 22₩nSkipping line 1373127: expected 15 fields, saw 22₩n'
b'Skipping line 1389508: expected 15 fields, saw 22₩nSkipping line 1413951: expected 15 fields, saw 22₩nSkipping line 143362
6: expected 15 fields, saw 22₩n'
b'Skipping line 1442698: expected 15 fields, saw 22₩nSkipping line 1472982: expected 15 fields, saw 22₩nSkipping line 148228
2: expected 15 fields, saw 22₩nSkipping line 1487808: expected 15 fields, saw 22₩nSkipping line 1500636: expected 15 fields,
saw 22₩n'
b'Skipping line 1511479: expected 15 fields, saw 22₩nSkipping line 1532302: expected 15 fields, saw 22₩nSkipping line 153795
2: expected 15 fields, saw 22₩nSkipping line 1539951: expected 15 fields, saw 22₩nSkipping line 1541020: expected 15 fields,
saw 22₩n'
b'Skipping line 1594217: expected 15 fields, saw 22₩nSkipping line 1612264: expected 15 fields, saw 22₩nSkipping line 161590
7: expected 15 fields, saw 22₩nSkipping line 1621859: expected 15 fields, saw 22₩n'
b'Skipping line 1653542: expected 15 fields, saw 22₩nSkipping line 1671537: expected 15 fields, saw 22₩nSkipping line 167287
9: expected 15 fields, saw 22₩nSkipping line 1674523: expected 15 fields, saw 22₩nSkipping line 1677355: expected 15 fields,
saw 22₩nSkipping line 1703907: expected 15 fields, saw 22₩n'
b'Skipping line 1713046: expected 15 fields, saw 22₩nSkipping line 1722982: expected 15 fields, saw 22₩nSkipping line 172729
0: expected 15 fields, saw 22₩nSkipping line 1744482: expected 15 fields, saw 22₩n'
b'Skipping line 1803858: expected 15 fields, saw 22₩nSkipping line 1810069: expected 15 fields, saw 22₩nSkipping line 182975
1: expected 15 fields, saw 22₩nSkipping line 1831699: expected 15 fields, saw 22₩n'
b'Skipping line 1863131: expected 15 fields, saw 22₩nSkipping line 1867917: expected 15 fields, saw 22₩nSkipping line 187479
0: expected 15 fields, saw 22₩nSkipping line 1879952: expected 15 fields, saw 22₩nSkipping line 1880501: expected 15 fields,
saw 22₩nSkipping line 1886655: expected 15 fields, saw 22₩nSkipping line 1887888: expected 15 fields, saw 22₩nSkipping line
1894286: expected 15 fields, saw 22₩nSkipping line 1895400: expected 15 fields, saw 22₩n'
b'Skipping line 1904040: expected 15 fields, saw 22₩nSkipping line 1907604: expected 15 fields, saw 22₩nSkipping line 191573
9: expected 15 fields, saw 22₩nSkipping line 1921514: expected 15 fields, saw 22₩nSkipping line 1939428: expected 15 fields,
saw 22₩nSkipping line 1944342: expected 15 fields, saw 22₩nSkipping line 1949699: expected 15 fields, saw 22₩nSkipping line
1961872: expected 15 fields, saw 22₩n'
b'Skipping line 1968846: expected 15 fields, saw 22₩nSkipping line 1999941: expected 15 fields, saw 22₩nSkipping line 200149
2: expected 15 fields, saw 22₩nSkipping line 2011204: expected 15 fields, saw 22₩nSkipping line 2025295: expected 15 fields,
saw 22₩n'
b'Skipping line 2041266: expected 15 fields, saw 22₩nSkipping line 2073314: expected 15 fields, saw 22₩nSkipping line 208013
3: expected 15 fields, saw 22₩nSkipping line 2088521: expected 15 fields, saw 22₩n'
b'Skipping line 2103490: expected 15 fields, saw 22₩nSkipping line 2115278: expected 15 fields, saw 22₩nSkipping line 215317
4: expected 15 fields, saw 22₩nSkipping line 2161731: expected 15 fields, saw 22₩n'
b'Skipping line 2165250: expected 15 fields, saw 22₩nSkipping line 2175132: expected 15 fields, saw 22₩nSkipping line 220681
7: expected 15 fields, saw 22₩nSkipping line 2215848: expected 15 fields, saw 22₩nSkipping line 2223811: expected 15 fields,
```

```
saw 22₩n'
b'Skipping line 2257265: expected 15 fields, saw 22₩nSkipping line 2259163: expected 15 fields, saw 22₩nSkipping line 226329
1: expected 15 fields, saw 22₩n'
b'Skipping line 2301943: expected 15 fields, saw 22₩nSkipping line 2304371: expected 15 fields, saw 22₩nSkipping line 230601
5: expected 15 fields, saw 22₩nSkipping line 2312186: expected 15 fields, saw 22₩nSkipping line 2314740: expected 15 fields,
saw 22₩nSkipping line 2317754: expected 15 fields, saw 22₩n'
b'Skipping line 2383514: expected 15 fields, saw 22₩n'
b'Skipping line 2449763: expected 15 fields, saw 22₩n'
b'Skipping line 2589323: expected 15 fields, saw 22₩n'
b'Skipping line 2775036: expected 15 fields, saw 22₩n'
b'Skipping line 2935174: expected 15 fields, saw 22₩n'
b'Skipping line 3078830: expected 15 fields, saw 22₩n'
b'Skipping line 3123091: expected 15 fields, saw 22₩n'
b'Skipping line 3185533: expected 15 fields, saw 22₩n'
b'Skipping line 4150395: expected 15 fields, saw 22₩n'
b'Skipping line 4748401: expected 15 fields, saw 22₩n'
```

Out[2]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	total_
0	US	37000337	R3DT59XH7HXR9K	B00303FI0G	529320574	Arthur Court Paper Towel Holder	Kitchen	5.0	0.0	
1	US	15272914	R1LFS11BNASSU8	B00JCZKZN6	274237558	Olde Thompson Bavaria Glass Salt and Pepper Mi...	Kitchen	5.0	0.0	
2	US	36137863	R296RT05AG0AF6	B00JLIKA5C	544675303	Progressive International PL8 Professional Man...	Kitchen	5.0	0.0	
3	US	43311049	R3V37XDZ7ZCI3L	B000GBNB8G	491599489	Zyliss Jumbo Garlic Press	Kitchen	5.0	0.0	
4	US	13763148	R14GU232NQFYX2	B00VJ5KX9S	353790155	1 X Premier Pizza Cutter - Stainless Steel 14"...	Kitchen	5.0	0.0	

(2) Build a bananced dataset of 250k reviews along with their rationg(50K per instances per each rating score) through random selection

In [3]:
```
# check the value of the star_rating
df['star_rating'].unique()
```

Out[3]:
```
array([ 5.,  1.,  3.,  4.,  2., nan])
```

In [4]:
```
# remove strange star_rating values like 'nan'
df.drop(df[df['star_rating'].isna()].index, inplace = True)

# change float type to int type
df['star_rating'] = df['star_rating'].astype(int)
```

In [5]:
```
# Select dataset along with their rationg(50K per instances per each rating score) through random selection
sample1 = df[df.star_rating == 1].sample(n=5000, random_state=2)
sample2 = df[df.star_rating == 2].sample(n=5000, random_state=2)
sample3 = df[df.star_rating == 3].sample(n=5000, random_state=2)
sample4 = df[df.star_rating == 4].sample(n=5000, random_state=2)
sample5 = df[df.star_rating == 5].sample(n=5000, random_state=2)

# concatenate sample 1-5 as a new dataframe called 'new'
new = sample1.append([sample2,sample3,sample4,sample5]).reset_index(drop=True)
```

In [6]:
```
# Check if each grade(star_rating) has 50K instances
rating_count = {k: v for k, v in zip(new['star_rating'].value_counts().index, new['star_rating'].value_counts())}
rating_count
```

Out[6]:
```
{5: 5000, 4: 5000, 3: 5000, 2: 5000, 1: 5000}
```

(3) Create ternary labels using the ratings

- class 1 : rating = 4 or 5
- class 2 : rating = 1 or 2
- class 3 : rating = 3

In [7]:
```
# To create ternary labels, mapping the ratings.
new.loc[(new['star_rating'] > 3), 'class'] = 1
new.loc[(new['star_rating'] < 3), 'class'] = 2
new.loc[(new['star_rating'] == 3), 'class'] = 3
```

In [8]:
```
new.head()
```

Out[8]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	tc
0	US	11903534	R3NBYY1PF9MXRT	B00FRMV38Y	872686823	Decodyne™ Morning Mug, Heat Sensitive Co...	Kitchen	1	0.0	
1	US	27885863	R1A10GP8CPG1A2	B003YFI0O6	111524501	Oster Electric Wine-Bottle Opener	Kitchen	1	1.0	
2	US	15355157	R147NFWDLR0AT9	B0002T4ZL4	978772977	Oggi 5355 4-Piece Acrylic Canister Set with Ai...	Kitchen	1	2.0	
3	US	15647704	R1GQQLPV9LCY1T	B0034J6QIY	591197834	Cuisinart SS-700 Single Serve Brewing System -...	Kitchen	1	0.0	
4	US	26424346	R1X5BB0UPZ4IWT	B000AXQA8I	330600737	Kuhn Rikon Twist and Chop, Artichoke	Kitchen	1	3.0	

(4) Store dataset after generation and reuse ot to reduce the computational load

In [9]:

```
new.to_csv('ternary_data.csv') # ternary dataset
new = pd.read_csv('ternary_data.csv', index_col=0)
new.head()
```

Out[9]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	tc

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	tc
0	US	11903534	R3NBYY1PF9MXRT	B00FRMV38Y	872686823	Decodyne™ Morning Mug, Heat Sensitive Co...	Kitchen	1	0.0	
1	US	27885863	R1A10GP8CPG1A2	B003YFI0O6	111524501	Oster Electric Wine-Bottle Opener	Kitchen	1	1.0	
2	US	15355157	R147NFWDLR0AT9	B0002T4ZL4	978772977	Oggi 5355 4-Piece Acrylic Canister Set with Ai...	Kitchen	1	2.0	
3	US	15647704	R1GQQLPV9LCY1T	B0034J6QIY	591197834	Cuisinart SS-700 Single Serve Brewing System -...	Kitchen	1	0.0	
4	US	26424346	R1X5BB0UPZ4IWT	B000AXQA8I	330600737	Kuhn Rikon Twist and Chop, Artichoke	Kitchen	1	3.0	

(5) Split the dataset : 80% traning set and 20% testing set

In [10]:
```
x_train, x_test, y_train, y_test = train_test_split(new['review_body'], new['class'], test_size=0.2, random_state = 2)
```

---

2. Word Embedding

- reference source : https://radimrehurek.com/gensim/models/word2vec.html

(a)

(1) Load the pretrained "word2vec-google-news-300" Word2Vec model and learn how to extract word embeddings for your datasets.

In [11]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

(2) Check semantic similarities of the generated vecotrs using two examples

- (1) cat + dog = ?

In [12]:
```
# calculate the similarity using the default "cosine similarity" measure.
print(wv.most_similar(positive=['cat','dog'], topn=1))

# calculate the similarity using different measure "cosmul"
print(wv.most_similar_cosmul(positive=['cat','dog'], topn=1))
```

```
[('puppy', 0.8089798092842102)]
[('puppy', 0.7729379534721375)]
```

- (2) Similarity between 'excellent' and 'outstanding'

In [13]:
```
print(wv.similarity('excellent','outstanding'))
```

```
0.55674857
```

=> Similarity between 'excellent' and 'outstanding' is lower than I expected. So I'll check similar words

In [14]:
```
print(wv.most_similar('excellent'))
```

```
[('terrific', 0.7409726977348328), ('superb', 0.7062715888023376), ('exceptional', 0.681470513343811), ('fantastic', 0.68028
47981452942), ('good', 0.6442928910255432), ('great', 0.6124600172042847), ('Excellent', 0.6091997623443604), ('impeccable',
0.5980967283248901), ('exemplary', 0.5959650278091431), ('marvelous', 0.582928478717804)]
```

In [15]:
```
print(wv.most_similar('outstanding'))
```

```
[('oustanding', 0.8012188673019409), ('Outstanding', 0.6041857600212097), ('exceptional', 0.6031844615936279), ('anchorman_J
ason_Lezak', 0.5947381258010864), ('outsanding', 0.566262423992157), ('Stock_HEI', 0.5573362708091736), ('excellent', 0.5567
48628616333), ('Synplicity_FPGA_implementation', 0.5520347356796265), ('exemplary', 0.5467386245727539), ('W3_Awards_honor
s', 0.5172522068023682)]
```

- (3) Similarity between 'chair' and 'desk'

In [16]:
```
print(wv.similarity('chair','desk'))
```

```
0.3149568
```

- (4) Similar things with soap?

In [17]:
```
print(wv.most_similar('soap'))
```

```
[('soaps', 0.7613304257392883), ('Soap', 0.6950218677520752), ('Colgate_Palmolive_toothpaste', 0.6457595229148865), ('deterg
ent', 0.5981624126434326), ('Colgate_toothpaste_Palmolive', 0.5842004418373108), ('antiseptic_soaps', 0.5792285203933716),
('soapy', 0.5768905878067017), ('Laundry_detergent', 0.5718933939933777), ('Tide_detergent_Ivory', 0.5658919215202332), ('Un
ilever_ULVR_LN', 0.5604559779167175)]
```

(b)

(1) Train a Word2Vec model using my dataset

- embedding size = 300
- window size = 11
- minimun word count = 10
- sg = 1 : skip-gram

In [18]:
```
import nltk

nltk.download('punkt') # For tokenizer
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:₩Users₩gmlwl₩AppData₩Roaming₩nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[18]:
```
True
```

In [19]:
```
def tokenize(temp):
    # For each sentence, word tokenization is performed using NLTK
    result = [word_tokenize(sentence) for sentence in temp]
    return result
```

In [20]:
```
# Train Word2Vec model using my own dataset
# sg = 1 : skip-gram
model = Word2Vec(sentences=tokenize(new['review_body']), vector_size=300, window=11, min_count=10, workers=4, sg=1)
model.save("word2vec.model")
```

(2) Check the semantic similarities using my own model

- (1) cat + dog = ?

In [21]:
```
# calculate the similarity using the default "cosine similarity" measure.
print(model.wv.most_similar(positive=['cat','dog'], topn=1))

# calculate the similarity using different measure "cosmul"
print(model.wv.most_similar_cosmul(positive=['cat','dog'], topn=1))
```

```
[('swear', 0.7196011543273926)]
[('swear', 0.6660094261169434)]
```

- (2) Similarity between 'excellent' and 'outstanding'

In [22]:
```
print(model.wv.similarity('excellent','outstanding'))
```

```
0.6124733
```

In [23]:
```
print(model.wv.most_similar('excellent'))
```

```
[('exceptional', 0.6363705992698669), ('outstanding', 0.6124733090400696), ('affordable', 0.6097689270973206), ('superb', 0.5697858333587646), ('acceptable', 0.5670941472053528), ('wonderful', 0.5664124488830566), ('adequate', 0.539593517780304), ('A+', 0.5354660749435425), ('awesome', 0.5353838205337524), ('amazingly', 0.5322791337966919)]
```

In [24]:
```
print(model.wv.most_similar('outstanding'))
```

```
[('speedy', 0.723319947719574), ('responsive', 0.7164787650108337), ('confirmed', 0.702231764793396), ('Seller', 0.7006707191467285), ('A+', 0.6945520043373108), ('exceptional', 0.6799882054328918), ('unacceptable', 0.6744035482406616), ('speaking', 0.6743783950805664), ('Company', 0.6710595488548279), ('Europe', 0.6699382066726685)]
```

- (3) Similarity between 'chair' and 'desk'

In [25]:
```
print(model.wv.similarity('chair','desk'))
```

```
0.48516065
```

- (4) Similar things with soap?

In [26]:
```
print(model.wv.most_similar('soap'))
```

```
[('soapy', 0.7130864858627319), ('detergent', 0.6908825635910034), ('sponge', 0.6703446507453918), ('dishwashing', 0.6624318361282349), ('wiping', 0.6428627967834473), ('bleach', 0.6349363327026367), ('rinsed', 0.6319710612297058), ('rinsing', 0.6148936748504639), ('scrubbed', 0.6108755469322205), ('deposits', 0.6097919344902039)]
```

(Conclude) Comparing vectors generated by myself and the pretrained model?

	My own Word2Vec	google-news-300

	My own Word2Vec	google news 300
most similar (positive(cat + dog), topn=1)	[('swear', 0.7196011543273926)] [('swear', 0.6660094261169434)]	[('puppy', 0.8089798092842102)] [('puppy', 0.7729379534721375)]
excellent | outstanding?	0.6124733	0.55674857
most_similar('excellent')	[('exceptional', 0.6363705992698669), ('outstanding', 0.6124733090400696), ('affordable', 0.6097689270973206), ('superb', 0.5697858333587646), ('acceptable', 0.5670941472053528), ('wonderful', 0.5664124488830566), ('adequate', 0.539593517780304), ('A+', 0.5354660749435425), ('awesome', 0.5353838205337524), ('amazingly', 0.5322791337966919)]	[('terrific', 0.7409726977348328), ('superb', 0.7062715888023376), ('exceptional', 0.681470513343811), ('fantastic', 0.6802847981452942), ('good', 0.6442928910255432), ('great', 0.6124600172042847), ('Excellent', 0.6091997623443604), ('impeccable', 0.5980967283248901), ('exemplary', 0.5959650278091431), ('marvelous', 0.582928478717804)]
most_similar('outstanding')	[('speedy', 0.723319947719574), ('responsive', 0.7164787650108337), ('confirmed', 0.702231764793396), ('Seller', 0.7006707191467285), ('A+', 0.6945520043373108), ('exceptional', 0.6799882054328918), ('unacceptable', 0.6744035482406616), ('speaking', 0.6743783950805664), ('Company', 0.6710595488548279), ('Europe', 0.6699382066726685)]	[('oustanding', 0.8012188673019409), ('Outstanding', 0.6041857600212097), ('exceptional', 0.6031844615936279), ('anchorman_Jason_Lezak', 0.5947381258010864), ('outsanding', 0.566262423992157), ('Stock_HEI', 0.5573362708091736), ('excellent', 0.556748628616333), ('Synplicity_FPGA_implementation', 0.5520347356796265), ('exemplary', 0.5467386245727539), ('W3_Awards_honors', 0.5172522068023682)]
Similarity (chair & desk?)	0.48516065	0.3149568
Similarity soap ?	[('soapy', 0.7130864858627319), ('detergent', 0.6908825635910034), ('sponge', 0.6703446507453918), ('dishwashing', 0.6624318361282349), ('wiping', 0.6428627967834473), ('bleach', 0.6349363327026367), ('rinsed', 0.6319710612297058), ('rinsing', 0.6148936748504639), ('scrubbed', 0.6108755469322205), ('deposits', 0.6097919344902039)]	[('soaps', 0.7613304257392883), ('Soap', 0.6950218677520752), ('Colgate_Palmolive_toothpaste', 0.6457595229148865), ('detergent', 0.5981624126434326), ('Colgate_toothpaste_Palmolive', 0.5842004418373108), ('antiseptic_soaps', 0.5792285203933716), ('soapy', 0.5768905878067017), ('Laundry_detergent', 0.5718933939933777), ('Tide_detergent_Ivory', 0.5658919215202332), ('Unilever_ULVR_LN', 0.5604559779

[Conclude] What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

=> The Word2Vec model I trained uses Amazon review data, so there are many terms related to the product. Objectively, the Word2Vec model using Google News data is better to encode semantic similarities between words.

---

3. Simple models

(0) First of all, we have to discard class 3(=rating 3). Because we will only use class 1 and class 2 data

In [27]:

```
# Make a new dataframe with class 1 and class 2.
new = pd.read_csv('ternary_data.csv', index_col=0)
new2 = new.loc[new['class'] < 3].reset_index(drop=True)
new2.to_csv('binary_data.csv')
new2.head()
```

Out[27]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	tc
0	US	11903534	R3NBYY1PF9MXRT	B00FRMV38Y	872686823	Decodyne™ Morning Mug, Heat Sensitive Co...	Kitchen	1	0.0	
1	US	27885863	R1A10GP8CPG1A2	B003YFI0O6	111524501	Oster Electric Wine-Bottle Opener	Kitchen	1	1.0	
2	US	15355157	R147NFWDLR0AT9	B0002T4ZL4	978772977	Oggi 5355 4-Piece Acrylic Canister Set with Ai...	Kitchen	1	2.0	

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	tc
3	US	15647704	R1GQQLPV9LCY1T	B0034J6QIY	591197834	Cuisinart SS-700 Single Serve Brewing System -...	Kitchen	1	0.0	
4	US	26424346	R1X5BB0UPZ4IWT	B000AXQA8I	330600737	Kuhn Rikon Twist and Chop, Artichoke	Kitchen	1	3.0	

In [28]:

```
# To save memory, only keep three columns ['star_rating','class','review_body']
review = new2[['star_rating','class','review_body']]
review
```

Out[28]:

	star_rating	class	review_body
0	1	2.0	Really didn't work as well as the pictures. Wh...
1	1	2.0	brand new out of the box and it wouldn't even ...
2	1	2.0	pure junk, lowest quality you could possibly g...
3	1	2.0	Really bummed! Machine worked great until a fe...
4	1	2.0	Sure, it cuts things, but the blades don't hav...
5	1	2.0	I bought a couple bambu bowls 3 months ago, an...
6	1	2.0	I bought this for my son. It arrived cracked, ...
7	1	2.0	was impossible to use because did not fit any ...
8	1	2.0	Flavors aren't good.
9	1	2.0	The bristles are way too soft to move any silk...
10	1	2.0	After two months (and just after the return wi...
11	1	2.0	Was shipped fast and we received it on time. N...
12	1	2.0	I absolutely would not buy or recommend this p...

	star_rating	class	review_body
13	1	2.0	This is the most poorly made item I have ever ...
14	1	2.0	I bought the cookware set through Groupon and ...
15	1	2.0	OK - I loved this machine at first but by opin...
16	1	2.0	Like others who have the identical experience,...
17	1	2.0	I received the package and when I opened it, i...
18	1	2.0	While the picture and the description were a p...
19	1	2.0	We ordered two sets of these glasses and they ...
20	1	2.0	Never got it to work. Had to return it. The Ke...
21	1	2.0	The scale did not work when I received it. In...
22	1	2.0	Wanted to save a couple of dollars and bought ...
23	1	2.0	I bought this slow cooker after reading the Co...
24	1	2.0	I returned it. It was very poorly made.
25	1	2.0	Makes very, very weak coffee when used, even w...
26	1	2.0	Pure crap. Poorly made. Not worth the price by...
27	1	2.0	Warning is on the packaging but nowhere on Ama...
28	1	2.0	Barely gets warm. Hub part works fine.
29	1	2.0	It did great for a few weeks. The straw fell o...
...	...	...	...
19970	5	1.0	LOVE this gel paste! I will be buying this fo...
19971	5	1.0	Very happy with my purchase!
19972	5	1.0	Makes an excellent, hot cup of coffee! Nice to...
19973	5	1.0	My husband had a curved paring knife in his ki...
19974	5	1.0	Beautiful and practical!
19975	5	1.0	exactly what I needed to make fresh squeezed j...
19976	5	1.0	Brews coffee in minutes. Good size for two people

	star_rating	class	review_body
19977	5	1.0	Love Mine so bought it for a friend as a gift.
19978	5	1.0	OXO makes two different versions of this...ide...
19979	5	1.0	looks great and like that its a 16oz
19980	5	1.0	I was a skeptic at first but this is one of th...
19981	5	1.0	This is a nice set of two salt or pepper mills...
19982	5	1.0	Ease of use, speed of cooking, (be sure to pre...
19983	5	1.0	I've used this pan for years and could not fin...
19984	5	1.0	Used this once, and it worked much better than...
19985	5	1.0	Theses favors are great quality and packaged v...
19986	5	1.0	I've been making my pies in my old pyrex dish....
19987	5	1.0	I love these colorful mugs...it is exactly as ...
19988	5	1.0	Good container Perfect Size Ordered ...
19989	5	1.0	HAVE NOT HAD A CHANCE TO USE IT YET BUT IM SUR...
19990	5	1.0	perfect, best iv'e had yet.
19991	5	1.0	The coffee grinder is very lightweight, extre...
19992	5	1.0	This is a great cheese cutter. Smooth....
19993	5	1.0	Great gift idea for alcoholic women. Xmas is c...
19994	5	1.0	It is what it is... A awesome spill proof coff...
19995	5	1.0	Just what i needed
19996	5	1.0	We have used several times - Love it!
19997	5	1.0	Good items
19998	5	1.0	After less than nine months the thermal decant...
19999	5	1.0	I love these; I've bought expensive versions a...

20000 rows × 3 columns

(1) Use data cleaning and preprocessing in order to include only important words from each review and improve performance

In [29]:
```
def data_cleaning(review):
    # convert the all reviews into the lower case
    review["preprocess_review"] = review["review_body"].str.lower()
    # remove HTML from the reviews
    review["preprocess_review"] = review["preprocess_review"].apply(lambda x: BeautifulSoup(x).get_text())# remove URLs fr
   # remove URLs from the reviews
    review["preprocess_review"] = review["preprocess_review"].str.replace('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*₩(₩),]
    # remove non-alphabetical characters
    review["preprocess_review"] = review["preprocess_review"].str.replace('[^a-zA-Z₩'+₩'+]', ' ')
    # remove the extra spaces between the words
    review["preprocess_review"] = review["preprocess_review"].replace('₩s+', ' ', regex=True)
    # perform contractions on the reviews
    review["preprocess_review"] = review["preprocess_review"].apply(lambda x: contractions.fix(x))
```

In [30]:
```
nltk.download('stopwords')
stop_words = (set(stopwords.words("english")) )

# remore stop_words
def removeStop(s):
    s_list = s.split()
    final_list = [word for word in s_list if word not in stop_words]
    final_string = ' '.join(final_list)
    return final_string

# Perform lemmatizer
def lemmatization_function(s):
    s_list = s.split()
    wordnet_lemmatizer = WordNetLemmatizer()
    final_list = [wordnet_lemmatizer.lemmatize(x) for x in s_list]
    final_string = ' '.join(final_list)
    return final_string
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:₩Users₩gmlwl₩AppData₩Roaming₩nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [31]:
```
def data_preprocessing(review):
    # remove the stop words
    review["preprocess_review"] = review["preprocess_review"].apply(lambda x: removeStop(x))

    # perform lemmatization
    review["preprocess_review"] = review["preprocess_review"].apply(lambda x: lemmatization_function(x))
```

In [32]:
```
# implement data cleaning and preprocessing
data_cleaning(review)
data_preprocessing(review)
```

In [33]:
```
review
```

Out[33]:

	star_rating	class	review_body	preprocess_review
0	1	2.0	Really didn't work as well as the pictures. Wh...	really work well picture mug awake still see c...
1	1	2.0	brand new out of the box and it wouldn't even ...	brand new box would even dig cork let alone pu...
2	1	2.0	pure junk, lowest quality you could possibly g...	pure junk lowest quality could possibly get gi...
3	1	2.0	Really bummed! Machine worked great until a fe...	really bummed machine worked great month ago t...
4	1	2.0	Sure, it cuts things, but the blades don't hav...	sure cut thing blade enough force behind cut d...
5	1	2.0	I bought a couple bambu bowls 3 months ago, an...	bought couple bambu bowl month ago bamboo laye...
6	1	2.0	I bought this for my son. It arrived cracked, ...	bought son arrived cracked returned replacement
7	1	2.0	was impossible to use because did not fit any ...	impossible use fit pot wound donating goodwill
8	1	2.0	Flavors aren't good.	flavor good
9	1	2.0	The bristles are way too soft to move any silk...	bristle way soft move silk save money
10	1	2.0	After two months (and just after the return wi...	two month return window closed kettle suddenly...
11	1	2.0	Was shipped fast and we received it on time. N...	shipped fast received time good buy zipper bro...
12	1	2.0	I absolutely would not buy or recommend this p...	absolutely would buy recommend productst two ye...
13	1	2.0	This is the most poorly made item I have ever ...	poorly made item ever bought online even thoug...
14	1	2.0	I bought the cookware set through Groupon and ...	bought cookware set groupon dissapointed poor ...
15	1	2.0	OK - I loved this machine at first but by opin...	ok loved machine first opinion changed purchas...
16	1	2.0	Like others who have the identical experience,...	like others identical experience unit simply f...
17	1	2.0	I received the package and when I opened it, i...	received package opened noticed one eight squa...
18	1	2.0	While the picture and the description were a p...	picture description perfect match replacement ...
19	1	2.0	We ordered two sets of these glasses and they ...	ordered two set glass arrived today reading de...
20	1	2.0	Never got it to work. Had to return it. The Ke...	never got work return keurig cuisinart s terri...

	star_rating	class	review_body	preprocess_review
21	1	2.0	The scale did not work when I received it. In...	scale work received installed battery got err ...
22	1	2.0	Wanted to save a couple of dollars and bought ...	wanted save couple dollar bought used kettle d...
23	1	2.0	I bought this slow cooker after reading the Co...	bought slow cooker reading cook illustrated gl...
24	1	2.0	I returned it. It was very poorly made.	returned poorly made
25	1	2.0	Makes very, very weak coffee when used, even w...	make weak coffee used even expresso much plast...
26	1	2.0	Pure crap. Poorly made. Not worth the price by...	pure crap poorly made worth price stretch imag...
27	1	2.0	Warning is on the packaging but nowhere on Ama...	warning packaging nowhere amazon's product des...
28	1	2.0	Barely gets warm. Hub part works fine.	barely get warm hub part work fine
29	1	2.0	It did great for a few weeks. The straw fell o...	great week straw fell lot put far would work f...
...	...	...	...	...
19970	5	1.0	LOVE this gel paste! I will be buying this fo...	love gel paste buying formula different color ...
19971	5	1.0	Very happy with my purchase!	happy purchase
19972	5	1.0	Makes an excellent, hot cup of coffee! Nice to...	make excellent hot cup coffee nice rid carafe ...
19973	5	1.0	My husband had a curved paring knife in his ki...	husband curved paring knife kitchen met I neve...
19974	5	1.0	Beautiful and practical!	beautiful practical
19975	5	1.0	exactly what I needed to make fresh squeezed j...	exactly needed make fresh squeezed juice
19976	5	1.0	Brews coffee in minutes. Good size for two people	brew coffee minute good size two people
19977	5	1.0	Love Mine so bought it for a friend as a gift.	love mine bought friend gift
19978	5	1.0	OXO makes two different versions of this...ide...	oxo make two different version identical lid a...
19979	5	1.0	looks great and like that its a 16oz	look great like oz
19980	5	1.0	I was a skeptic at first but this is one of th...	skeptic first one product make happy purchase ...
19981	5	1.0	This is a nice set of two salt or pepper mills...	nice set two salt pepper mill enjoy fresh crac...
19982	5	1.0	Ease of use, speed of cooking, (be sure to pre...	ease use speed cooking sure preheat full minut...
19983	5	1.0	I've used this pan for years and could not fin...	I used pan year could find local retail store ...
19984	5	1.0	Used this once, and it worked much better than...	used worked much better previous silicone mat ...

	star_rating	class	review_body	preprocess_review
19985	5	1.0	Theses favors are great quality and packaged v...	thesis favor great quality packaged neatly wou...
19986	5	1.0	I've been making my pies in my old pyrex dish....	I making pie old pyrex dish cleanup never easy...
19987	5	1.0	I love these colorful mugs...it is exactly as ...	love colorful mug exactly saw online would rec...
19988	5	1.0	Good container Perfect Size Ordered ...	good containerperfect sizeordered two moreperf...
19989	5	1.0	HAVE NOT HAD A CHANCE TO USE IT YET BUT IM SUR...	chance use yet I sure awesome sorry cannot com...
19990	5	1.0	perfect, best iv'e had yet.	perfect best iv'e yet
19991	5	1.0	The coffee grinder is very lightweight, extre...	coffee grinder lightweight extremely easy use ...
19992	5	1.0	This is a great cheese cutter. Smooth....	great cheese cutter smooth
19993	5	1.0	Great gift idea for alcoholic women. Xmas is c...	great gift idea alcoholic woman xmas coming gr...
19994	5	1.0	It is what it is... A awesome spill proof coff...	awesome spill proof coffee mug first one dent ...
19995	5	1.0	Just what i needed	needed
19996	5	1.0	We have used several times - Love it!	used several time love
19997	5	1.0	Good items	good item
19998	5	1.0	After less than nine months the thermal decant...	le nine month thermal decanter get hot touch n...
19999	5	1.0	I love these; I've bought expensive versions a...	love I bought expensive version funky angle st...

20000 rows × 4 columns

In [34]:

```
# split train data and test data (80%:20%)
X_train, X_test, Y_train, Y_test = train_test_split(review['preprocess_review'], review['class'], test_size=0.2, random_sta
```

(2) Make the Word2Vec features as a input using my own Word2Vec

In [35]:

```
# sg = 1 : skip-gram
wm_model = Word2Vec(sentences=tokenize(review['preprocess_review']), vector_size=300, window=11, min_count=10, workers=4, s
wm_model.save("new_word2vec.model")
```

In [36]:

```
def change_to_vector(X, Y):

    total_vector = tokenize(X)
    new_X = []
```

```
    remove_index = []
    for idx,sentence in zip(X.index,total_vector):
        average = [0,]
        words = list(filter(lambda x: x in wm_model.wv.index_to_key, sentence)) # Filtering. only keep existed words
        if len(words) == 0 : # If list 'words' is empty, we have to remove it, So keep the index value.
            remove_index.append(idx)
            continue
        else:
            for word in words:
                average += wm_model.wv[word]
            new_X.append(average / len(words))

    # Remove the Y_train value paired with the removed X_train
    new_Y = Y.drop(labels=remove_index)

    return new_X, new_Y
```

In [37]:

```
wm_X_train, wm_Y_train = change_to_vector(X_train, Y_train)
```

In [38]:

```
wm_X_test, wm_Y_test = change_to_vector(X_test, Y_test)
```

(3) Train and test perceptron with my own Word2Vec features

In [39]:

```
# Train Perceptron and test data
pct = Perceptron(tol=1e-3, random_state=2)
pct.fit(wm_X_train, wm_Y_train)
pct_y_test_pred = pct.predict(wm_X_test)
```

In [40]:

```
print('[My own Word2Vec] Perceptron_Test_Accuracy: %f' % accuracy_score(wm_Y_test, pct_y_test_pred))
print('[My own Word2Vec] Perceptron_Test_Precision: %f' % precision_score(wm_Y_test, pct_y_test_pred))
print('[My own Word2Vec] Perceptron_Test_Recall: %f' % recall_score(wm_Y_test, pct_y_test_pred))
print('[My own Word2Vec] Perceptron_Test_F1 Score: %f' % f1_score(wm_Y_test, pct_y_test_pred))
```

```
[My own Word2Vec] Perceptron_Test_Accuracy: 0.750125
[My own Word2Vec] Perceptron_Test_Precision: 0.685045
[My own Word2Vec] Perceptron_Test_Recall: 0.938765
[My own Word2Vec] Perceptron_Test_F1 Score: 0.792083
```

(4) Train and Test SVM with my own Word2Vec features

In [41]:

```
# Train SVM and test data
svm = LinearSVC(random_state=2)
svm.fit(wm_X_train, wm_Y_train)
svm_y_test_pred = svm.predict(wm_X_test)
```

In [42]:
```
print('[My own Word2Vec] SVM_Test_Accuracy: %f' % accuracy_score(wm_Y_test, svm_y_test_pred))
print('[My own Word2Vec] SVM_Test_Precision: %f' % precision_score(wm_Y_test, svm_y_test_pred))
print('[My own Word2Vec] SVM_Test_Recall: %f' % recall_score(wm_Y_test, svm_y_test_pred))
print('[My own Word2Vec] SVM_Test_F1 Score: %f' % f1_score(wm_Y_test, svm_y_test_pred))
```

```
[My own Word2Vec] SVM_Test_Accuracy: 0.817226
[My own Word2Vec] SVM_Test_Precision: 0.829517
[My own Word2Vec] SVM_Test_Recall: 0.804938
[My own Word2Vec] SVM_Test_F1 Score: 0.817043
```

(5) Make the Word2Vec features as a input using "word2vec-google-news-300."

In [43]:
```
def change_to_wv_google_news(X, Y):

    total_vector = tokenize(X)
    new_X = []
    remove_index = []
    for idx,sentence in zip(X.index,total_vector):
        average = [0,]
        words = list(filter(lambda x: x in wv.index_to_key, sentence)) # Filtering. Only keep existed words
        if len(words) == 0 : # If list 'words' is empty, we have to remove it, So keep the index value.
            remove_index.append(idx)
            continue
        else:
            for word in words:
                average += wv[word]
            new_X.append(average / len(words))

    # Remove the Y_train value paired with the removed X_train
    new_Y = Y.drop(labels=remove_index)

    return new_X, new_Y
```

In [44]:
```
gn_X_train, gn_Y_train = change_to_wv_google_news(X_train, Y_train)
```

In [45]:
```
gn_X_test, gn_Y_test = change_to_wv_google_news(X_test, Y_test)
```

(6) Train and test perceptron with "word2vec-google-news-300" model

In [46]:
```
# Train Perceptron and test data
pct = Perceptron(tol=1e-3, random_state=2)
pct.fit(gn_X_train, gn_Y_train)
gn_pct_y_test_pred = pct.predict(gn_X_test)
```

In [47]:
```
print('[google-news-300] Perceptron_Test_Accuracy: %f' % accuracy_score(gn_Y_test, gn_pct_y_test_pred))
print('[google-news-300] Perceptron_Test_Precision: %f' % precision_score(gn_Y_test, gn_pct_y_test_pred))
print('[google-news-300] Perceptron_Test_Recall: %f' % recall_score(gn_Y_test, gn_pct_y_test_pred))
print('[google-news-300] Perceptron_Test_F1 Score: %f' % f1_score(gn_Y_test, gn_pct_y_test_pred))
```

```
[google-news-300] Perceptron_Test_Accuracy: 0.776527
[google-news-300] Perceptron_Test_Precision: 0.852174
[google-news-300] Perceptron_Test_Recall: 0.676862
[google-news-300] Perceptron_Test_F1 Score: 0.754468
```

(7) Train and Test SVM with "word2vec-google-news-300" model

In [48]:
```
# Train SVM and test data
svm = LinearSVC(random_state=2)
svm.fit(gn_X_train, gn_Y_train)
gn_svm_y_test_pred = svm.predict(gn_X_test)
```

In [49]:
```
print('[google-news-300] SVM_Test_Accuracy: %f' % accuracy_score(gn_Y_test, gn_svm_y_test_pred))
print('[google-news-300] SVM_Test_Precision: %f' % precision_score(gn_Y_test, gn_svm_y_test_pred))
print('[google-news-300] SVM_Test_Recall: %f' % recall_score(gn_Y_test, gn_svm_y_test_pred))
print('[google-news-300] SVM_Test_F1 Score: %f' % f1_score(gn_Y_test, gn_svm_y_test_pred))
```

```
[google-news-300] SVM_Test_Accuracy: 0.807808
[google-news-300] SVM_Test_Precision: 0.824987
[google-news-300] SVM_Test_Recall: 0.788357
[google-news-300] SVM_Test_F1 Score: 0.806256
```

(8) Create Tf-Idf

In [50]:
```
tfvector = TfidfVectorizer()
tf_x_train = tfvector.fit_transform(X_train)
tf_x_test = tfvector.transform(X_test)
```

In [51]:
```
print(len(X_train),len(Y_train))
```

```
16000 16000
```

(9) Train and test perceptron with tf-idf

In [52]:
```
pct = Perceptron(tol=1e-3, random_state=2)
pct.fit(tf_x_train, Y_train)
pct_y_train_pred = pct.predict(tf_x_train)
tf_pct_y_test_pred = pct.predict(tf_x_test)
```

In [53]:
```
print('[TF-IDF] Perceptron_Test_Accuracy: %f' % accuracy_score(Y_test, tf_pct_y_test_pred))
print('[TF-IDF] Perceptron_Test_Precision: %f' % precision_score(Y_test, tf_pct_y_test_pred))
```

```
print('[TF-IDF] Perceptron_Test_Recall: %f' % recall_score(Y_test, tf_pct_y_test_pred))
print('[TF-IDF] Perceptron_Test_F1 Score: %f' % f1_score(Y_test, tf_pct_y_test_pred))
```

```
[TF-IDF] Perceptron_Test_Accuracy: 0.800000
[TF-IDF] Perceptron_Test_Precision: 0.804757
[TF-IDF] Perceptron_Test_Recall: 0.800000
[TF-IDF] Perceptron_Test_F1 Score: 0.802372
```

(10) Train and test SVM with tf-idf

In [54]:

```
svm = LinearSVC(random_state=2)
svm.fit(tf_x_train, Y_train)
svm_y_train_pred = svm.predict(tf_x_train)
tf_svm_y_test_pred = svm.predict(tf_x_test)
```

In [55]:

```
print('[TF-IDF] SVM_Test_Accuracy: %f' % accuracy_score(Y_test, tf_svm_y_test_pred))
print('[TF-IDF] SVM_Test_Precision: %f' % precision_score(Y_test, tf_svm_y_test_pred))
print('[TF-IDF] SVM_Test_Recall: %f' % recall_score(Y_test, tf_svm_y_test_pred))
print('[TF-IDF] SVM_Test_F1 Score: %f' % f1_score(Y_test, tf_svm_y_test_pred))
```

```
[TF-IDF] SVM_Test_Accuracy: 0.846000
[TF-IDF] SVM_Test_Precision: 0.855634
[TF-IDF] SVM_Test_Recall: 0.837931
[TF-IDF] SVM_Test_F1 Score: 0.846690
```

Report

"word2Vec-google-news-300" VS "my own Word2Vec model" VS "TF-IDF"

Perceptron	Accuracy	Precision	Recall	F1-score
My Own Word2Vec	0.750125	0.685045	0.938765	0.792083
google-news-300	0.776527	0.852174	0.676862	0.754468
TF-IDF	0.800000	0.804757	0.800000	0.802372
SVM	**Accuracy**	**Precision**	**Recall**	**F1-score**
My Own Word2Vec	0.817226	0.829517	0.804938	0.817043
google-news-300	0.807808	0.824987	0.788357	0.806256
TF-IDF	0.846000	0.855634	0.837931	0.846690

-> Among the three feature types, the model using TF-IDF as a feature showed the best performance. Comparing my own word2vec and word2vec-google-news, the performance difference between the two is very small, so the performance effect will vary depending on what kind of data the training data is.

---

4. Feedforward Neural Networks

reference : https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist

(a) Using the average Word2Vec vectors

- binary classification using class 1 and class 2
- ternary model for the three class

(1) Make binary classification datasets (class 1 = rating 4, 5) (class 2 = rating 1, 2)

In [57]:
```
# Make a binary classification dataset
new = pd.read_csv('binary_data.csv', index_col=0)
bi = new.loc[new['class'] < 3].reset_index(drop=True)
bi = bi[['star_rating','class','review_body']]

# implement data cleaning and preprocessing
```

```
data_cleaning(bi)
data_preprocessing(bi)

# For split train data and test data
from sklearn.model_selection import train_test_split

# using my own Word2Vec model
temp_X, temp_Y = change_to_vector(bi['preprocess_review'], bi['class'])
bi_mw_X_train, bi_mw_X_test, bi_mw_Y_train, bi_mw_Y_test = train_test_split(temp_X, temp_Y, test_size=0.2, random_state = 2

# using "word2vec-google-news-300" modoel
temp_X2, temp_Y2 = change_to_wv_google_news(bi['preprocess_review'], bi['class'])
bi_gn_X_train, bi_gn_X_test, bi_gn_Y_train, bi_gn_Y_test = train_test_split(temp_X2, temp_Y2, test_size=0.2, random_state =
```

(2) Make ternary classification datasets (class 1 = rating 4, 5) (class 2 = rating 1, 2) (class 3 = rating 3)

In [58]:

```
# Make a ternary model for three class
new = pd.read_csv('ternary_data.csv', index_col=0)

# To create ternary labels, mapping the ratings.
new.loc[(new['star_rating'] > 3), 'class'] = 1
new.loc[(new['star_rating'] < 3), 'class'] = 2
new.loc[(new['star_rating'] == 3), 'class'] = 3

ten = new[['star_rating','class','review_body']]

# implement data cleaning and preprocessing
data_cleaning(ten)
data_preprocessing(ten)

# split train data and test data
from sklearn.model_selection import train_test_split

# using my own Word2Vec model
temp_X3, temp_Y3 = change_to_vector(ten['preprocess_review'], ten['class'])
ten_wm_X_train, ten_wm_X_test, ten_wm_Y_train, ten_wm_Y_test = train_test_split(temp_X3, temp_Y3, test_size=0.2, random_sta

# using "word2vec-google-news-300" modoel
temp_X4, temp_Y4 = change_to_wv_google_news(ten['preprocess_review'], ten['class'])
ten_gn_X_train, ten_gn_X_test, ten_gn_Y_train, ten_gn_Y_test = train_test_split(temp_X4, temp_Y4, test_size=0.2, random_sta
```

(4) Make a Feedforward model

In [59]:

```
# Feedforward model
model = nn.Sequential(
```

```python
    nn.Linear(300, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.ReLU(),
    nn.Linear(10, 2))

print(model)
```

```
Sequential(
  (0): Linear(in_features=300, out_features=50, bias=True)
  (1): ReLU()
  (2): Linear(in_features=50, out_features=10, bias=True)
  (3): ReLU()
  (4): Linear(in_features=10, out_features=2, bias=True)
)
```

In [60]:

```python
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [61]:

```python
def train(epoch):
    model.train()  # Model train

    # Train model with mini batch
    for data, targets in loader_train:

        optimizer.zero_grad()  # Initiate
        outputs = model(data)  # model train and output
        loss = loss_fn(outputs, targets)  # Calculate loss values (real value - predicted value)
        loss.backward()  # Backpropagation
        optimizer.step()  # Edit weights

    if epoch % 10 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(epoch, 100, loss.item()))
```

In [62]:

```python
def test():
    model.eval()  # Test Model
    correct = 0

    # Create minibatch
    with torch.no_grad():
        for data, targets in loader_test:

            outputs = model(data)  # Put input data and get output data
```

```
            # Calculate correct case
            _, predicted = torch.max(outputs.data, 1)  # Calculate which label has the highest probability
            correct += predicted.eq(targets.data.view_as(predicted)).sum()  # If it matches the answer, increase the count

    # Print accuracy
    data_num = len(loader_test.dataset)
    print('\nAccuracy with test data: {}/{} ({:.0f}%)\n'.format(correct,
                                                               data_num, 100. * correct / data_num))
```

Binary classification

(5-1) Binary classification with average Word2Vec vectors that are made by my own Word2Vec model

In [63]:

```
# Set data (chage target data. If the class is '2', change to '1') and change data type
# Change datatype to tensor
X_train = torch.Tensor(bi_mw_X_train)
X_test = torch.Tensor(bi_mw_X_test)
y_train = torch.LongTensor(bi_mw_Y_train-1)
y_test = torch.LongTensor(bi_mw_Y_test.values-1)

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [64]:

```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.578847
Epoch   10/100 Cost: 0.349494
Epoch   20/100 Cost: 0.488557
Epoch   30/100 Cost: 0.304918
Epoch   40/100 Cost: 0.306718
Epoch   50/100 Cost: 0.303817
Epoch   60/100 Cost: 0.171023
Epoch   70/100 Cost: 0.174204
Epoch   80/100 Cost: 0.297377
Epoch   90/100 Cost: 0.203899
```

```
Accuracy with test data: 3269/3994 (82%)
```

(5-2) Binary classification with average Word2Vec vectors that are made by word2vec-google-news-300 model

In [65]:
```python
# Set data (chage target data. If the class is '2', change to '1') and change data type
# Change datatype to tensor
X_train = torch.Tensor(bi_gn_X_train)
X_test = torch.Tensor(bi_gn_X_test)
y_train = torch.LongTensor(bi_gn_Y_train-1)
y_test = torch.LongTensor(bi_gn_Y_test.values-1)

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [66]:
```python
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [67]:
```python
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.537454
Epoch   10/100 Cost: 0.213030
Epoch   20/100 Cost: 0.310074
Epoch   30/100 Cost: 0.192528
Epoch   40/100 Cost: 0.224727
Epoch   50/100 Cost: 0.133237
Epoch   60/100 Cost: 0.131064
Epoch   70/100 Cost: 0.118380
Epoch   80/100 Cost: 0.103861
Epoch   90/100 Cost: 0.156226

Accuracy with test data: 3122/3997 (78%)
```

Tenery classification

(6-1) Tenerary classification with average Word2Vec vectors that are made by my own Word2Vec model

In [68]:
```
model = nn.Sequential(
    nn.Linear(300, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.ReLU(),
    nn.Linear(10, 3))

print(model)
```

```
Sequential(
  (0): Linear(in_features=300, out_features=50, bias=True)
  (1): ReLU()
  (2): Linear(in_features=50, out_features=10, bias=True)
  (3): ReLU()
  (4): Linear(in_features=10, out_features=3, bias=True)
)
```

In [69]:
```
# Set data
X_train = torch.Tensor(ten_wm_X_train)
X_test = torch.Tensor(ten_wm_X_test)
y_train = torch.LongTensor(ten_wm_Y_train-1) # Reduce each class number by 1
y_test = torch.LongTensor(ten_wm_Y_test.values-1) # Reduce each class number by 1

ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [70]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [71]:
```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.901392
Epoch   10/100 Cost: 0.692493
Epoch   20/100 Cost: 0.836561
Epoch   30/100 Cost: 0.581890
```

```
Epoch  40/100 Cost: 0.621298
Epoch  50/100 Cost: 0.622448
Epoch  60/100 Cost: 0.671561
Epoch  70/100 Cost: 0.554857
Epoch  80/100 Cost: 0.586218
Epoch  90/100 Cost: 0.582200

Accuracy with test data: 3245/4992 (65%)
```

(6-2) Ternary classification with average Word2Vec vectors that are made by word2vec-google-news-300 model

In [72]:
```
# Set data
X_train = torch.Tensor(ten_gn_X_train)
X_test = torch.Tensor(ten_gn_X_test)
y_train = torch.LongTensor(ten_gn_Y_train-1) # Reduce each class number by 1
y_test = torch.LongTensor(ten_gn_Y_test.values-1) # Reduce each class number by 1

ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [73]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [74]:
```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch   0/100 Cost: 0.829969
Epoch  10/100 Cost: 0.794858
Epoch  20/100 Cost: 0.601884
Epoch  30/100 Cost: 0.560058
Epoch  40/100 Cost: 0.367260
Epoch  50/100 Cost: 0.484547
Epoch  60/100 Cost: 0.845158
Epoch  70/100 Cost: 0.536882
Epoch  80/100 Cost: 0.200163
Epoch  90/100 Cost: 0.314929
```

```
Accuracy with test data: 3017/4996 (60%)
```

Report - with the average Word2Vec vectors

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	82%	65%
google-news-300	78%	60%

(b) Concatenate the first 10 Word2Vec vectors

(1) To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature

In [75]:
```
# Calculate concatenate the first 10 word2vec vectors with my own Word2Vec model
def concatenate_word2vec(X, Y):
    total_vector = tokenize(X)
    new_X = []
    remove_index = []
    for idx,sentence in zip(X.index,total_vector):
        temp_X = []
        words = list(filter(lambda x: x in wm_model.wv.index_to_key, sentence)) # Only keep existed words.
        if len(words) == 0 : # If list 'words' is empty, we have to remove it, So keep the index value.
            remove_index.append(idx)
            continue
        else:
            for word in words[:10]: # first 10 Word3Vec vectors
                temp_X = np.concatenate((temp_X, wm_model.wv[word]))
        if len(temp_X) != 3000:
            temp_X = np.pad(temp_X, (0,3000-len(temp_X)), 'constant', constant_values=0)
        new_X.append(temp_X)

    # Remove the Y_train value paired with the removed X_train
    new_Y = Y.drop(labels=remove_index)
    return new_X, new_Y
```

In [76]:
```
# Calculate concatenate the first 10 word2vec vectors with Word2Vec-google-news model
```

```python
def concatenate_word2vec_google_news(X, Y):
    total_vector = tokenize(X)
    new_X = []
    remove_index = []
    for idx,sentence in zip(X.index,total_vector):
        temp_X = []
        words = list(filter(lambda x: x in wv.index_to_key, sentence)) # Only keep existed words.
        if len(words) == 0 : # If list 'words' is empty, we have to remove it, So keep the index value.
            remove_index.append(idx)
            continue
        else:
            for word in words[:10]: # first 10 Word3Vec vectors
                temp_X = np.concatenate((temp_X, wv[word]))
        if len(temp_X) != 3000:
            temp_X = np.pad(temp_X, (0,3000-len(temp_X)), 'constant', constant_values=0)
        new_X.append(temp_X)

    # Remove the Y_train value paired with the removed X_train
    new_Y = Y.drop(labels=remove_index)

    return new_X, new_Y
```

(2) Set data (concatenate the first 10 Word2Vec vectors for each review as the input feature)

In [77]:

```python
# using my own Word2Vec model
temp_X5, temp_Y5 = concatenate_word2vec(bi['preprocess_review'], bi['class'])
bi_mw10_X_train, bi_mw10_X_test, bi_mw10_Y_train, bi_mw10_Y_test = train_test_split(temp_X5, temp_Y5, test_size=0.2, random

# using "word2vec-google-news-300" modoel
temp_X6, temp_Y6 = concatenate_word2vec_google_news(bi['preprocess_review'], bi['class'])
bi_gn10_X_train, bi_gn10_X_test, bi_gn10_Y_train, bi_gn10_Y_test = train_test_split(temp_X6, temp_Y6, test_size=0.2, random
```

In [78]:

```python
# using my own Word2Vec model
temp_X7, temp_Y7 = concatenate_word2vec(ten['preprocess_review'], ten['class'])
ten_wm10_X_train, ten_wm10_X_test, ten_wm10_Y_train, ten_wm10_Y_test = train_test_split(temp_X7, temp_Y7, test_size=0.2, ra

# using "word2vec-google-news-300" modoel
temp_X8, temp_Y8 = concatenate_word2vec_google_news(ten['preprocess_review'], ten['class'])
ten_gn10_X_train, ten_gn10_X_test, ten_gn10_Y_train, ten_gn10_Y_test = train_test_split(temp_X8, temp_Y8, test_size=0.2, ra
```

Binary classification

(3) Set model for binary classification with concatenate the first 10 Word2Vec voctors

In [79]:
```
model = nn.Sequential(
    nn.Linear(3000, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.ReLU(),
    nn.Linear(10, 2))

print(model)
```

```
Sequential(
  (0): Linear(in_features=3000, out_features=50, bias=True)
  (1): ReLU()
  (2): Linear(in_features=50, out_features=10, bias=True)
  (3): ReLU()
  (4): Linear(in_features=10, out_features=2, bias=True)
)
```

(4-1) With my own word2vec vector

In [80]:
```
# Set data (chage target data. If the class is '2', change to '1') and change data type
# Change datatype to tensor
X_train = torch.Tensor(bi_mw10_X_train)
X_test = torch.Tensor(bi_mw10_X_test)
y_train = torch.LongTensor(bi_mw10_Y_train-1) # Reduce each class number by 1
y_test = torch.LongTensor(bi_mw10_Y_test.values-1) # Reduce each class number by 1

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [81]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [82]:
```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.408697
Epoch   10/100 Cost: 0.070430
```

```
Epoch   20/100 Cost: 0.004547
Epoch   30/100 Cost: 0.064171
Epoch   40/100 Cost: 0.000299
Epoch   50/100 Cost: 0.000494
Epoch   60/100 Cost: 0.002284
Epoch   70/100 Cost: 0.000079
Epoch   80/100 Cost: 0.050645
Epoch   90/100 Cost: 0.050819

Accuracy with test data: 2908/3994 (73%)
```

(4-2) With google news word2vec

In [83]:

```
# Set data (chage target data. If the class is '2', change to '1') and change data type
# Change datatype to tensor
X_train = torch.Tensor(bi_gn10_X_train)
X_test = torch.Tensor(bi_gn10_X_test)
y_train = torch.LongTensor(bi_gn10_Y_train-1) # Reduce each class number by 1
y_test = torch.LongTensor(bi_gn10_Y_test.values-1) # Reduce each class number by 1

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [84]:

```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [85]:

```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.648548
Epoch   10/100 Cost: 0.123390
Epoch   20/100 Cost: 0.009467
Epoch   30/100 Cost: 0.005616
Epoch   40/100 Cost: 0.001202
Epoch   50/100 Cost: 0.003593
Epoch   60/100 Cost: 0.025437
```

```
Epoch   70/100 Cost: 0.035908
Epoch   80/100 Cost: 0.000258
Epoch   90/100 Cost: 0.000211

Accuracy with test data: 2883/3997 (72%)
```

Ternary classification

(5) Set model for binary classification with concatenate the first 10 Word2Vec voctors

In [86]:
```
model = nn.Sequential(
    nn.Linear(3000, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.ReLU(),
    nn.Linear(10, 3))

print(model)
```

```
Sequential(
  (0): Linear(in_features=3000, out_features=50, bias=True)
  (1): ReLU()
  (2): Linear(in_features=50, out_features=10, bias=True)
  (3): ReLU()
  (4): Linear(in_features=10, out_features=3, bias=True)
)
```

(6-1) With my own word2vec vectors

In [87]:
```
# Set data
X_train = torch.Tensor(ten_wm10_X_train)
X_test = torch.Tensor(ten_wm10_X_test)
y_train = torch.LongTensor(ten_wm10_Y_train-1) # Reduce each class number by 1
y_test = torch.LongTensor(ten_wm10_Y_test.values-1) # Reduce each class number by 1

ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [88]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
```

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

In [89]:
```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.873941
Epoch   10/100 Cost: 0.465283
Epoch   20/100 Cost: 0.099524
Epoch   30/100 Cost: 0.142683
Epoch   40/100 Cost: 0.060731
Epoch   50/100 Cost: 0.011700
Epoch   60/100 Cost: 0.015981
Epoch   70/100 Cost: 0.013891
Epoch   80/100 Cost: 0.005859
Epoch   90/100 Cost: 0.030007

Accuracy with test data: 2707/4992 (54%)
```

(6-2) With google news word2vec vectors

In [90]:
```
# Set data r
X_train = torch.Tensor(ten_gn10_X_train)
X_test = torch.Tensor(ten_gn10_X_test)
y_train = torch.LongTensor(ten_gn10_Y_train-1) # Reduce each class number by 1
y_test = torch.LongTensor(ten_gn10_Y_test.values-1) # Reduce each class number by 1

ds_train = TensorDataset(X_train, y_train)
ds_test = TensorDataset(X_test, y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [91]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()

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

In [92]:
```
# Train with epoch = 100, and test data
for epoch in range(100):
    train(epoch)
test()
```

```
Epoch    0/100 Cost: 0.665067
Epoch   10/100 Cost: 0.385254
Epoch   20/100 Cost: 0.025130
Epoch   30/100 Cost: 0.001866
Epoch   40/100 Cost: 0.000823
Epoch   50/100 Cost: 0.042108
Epoch   60/100 Cost: 0.005129
Epoch   70/100 Cost: 0.002946
Epoch   80/100 Cost: 0.000190
Epoch   90/100 Cost: 0.000655

Accuracy with test data: 2611/4996 (52%)
```

Report - with the concatenated the first 10 Word2Vec vectors

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	73%	54%
google-news-300	72%	52%

[Report] => Looking at the results of Feedforward Neural Networks, overall, the model with the features that used my own word2vec vectors as an input has better performance.

[Conclude] - Total Report

Binary Classification	TF-IDF	the average Word2Vec vectors (My own word2vec)	the average Word2Vec vectors (google-news-300)	the concatenated the first 10 Word2Vec vectors (My own word2vec)	the concatenated the first 10 Word2Vec vectors (google-news-300)
Perceptron	80%	75%	77%	x	x
SVM	84%	80%	80%	x	x
Feedforward Neural Networks	x	82%	78%	73%	72%

[Report] => The SVM model using TF-IDF features has the best performance compared to the other models. The Feedforward Neural Networks model using the average my own word2vec vectors has the second-best performance.

---

5. Recurrent Neural Networks

※[Important] Order :

1. (Using my word2vec model) Binary RNN
2. (Using my word2vec model) Binary GRU
3. (Using google-news-300 model) Binary RNN
4. (Using google-news-300 model) Binary GRU
5. (Using my word2vec model) Ternary RNN
6. (Using my word2vec model) Ternary GRU

7. (Using google-news-300 model) Ternary RNN
8. (Using google-news-300 model) Ternary GRU

Unfortunately, I have computational resource limitations, especially memory issue, so I had to set epoch to 3. (epoch=3)

(a-1-1) Binary model with RNN using my word2vec model

(0) Using the features that I generated using the models I prepared in the "Word Embedding" section.

In [2]:
```
new = pd.read_csv('binary_data.csv', index_col=0)
new.head()
```

Out[2]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	tc
0	US	11903534	R3NBYY1PF9MXRT	B00FRMV38Y	872686823	Decodyne™ Morning Mug, Heat Sensitive Co...	Kitchen	1	0.0	
1	US	27885863	R1A10GP8CPG1A2	B003YFI0O6	111524501	Oster Electric Wine-Bottle Opener	Kitchen	1	1.0	
2	US	15355157	R147NFWDLR0AT9	B0002T4ZL4	978772977	Oggi 5355 4-Piece Acrylic Canister Set with Ai...	Kitchen	1	2.0	
3	US	15647704	R1GQQLPV9LCY1T	B0034J6QIY	591197834	Cuisinart SS-700 Single Serve Brewing System -...	Kitchen	1	0.0	
4	US	26424346	R1X5BB0UPZ4IWT	B000AXQA8I	330600737	Kuhn Rikon Twist and Chop, Artichoke	Kitchen	1	3.0	

(1) Ready to dataset. We should change string(word) to number to feed tada into RNN.

In [3]:
```
def tokenize(temp):
    # For each sentence, word tokenization is performed using NLTK
    result = [word_tokenize(sentence) for sentence in temp]
    return result
```

In [4]:
```
# load my word2vec model
word2vec = Word2Vec.load("word2vec.model")
```

In [5]:
```
# Change the tokenized reviews to int type(using my Word2vec model)
def review_to_int(reviews):
    reviews_ints = []
    for review in reviews:
        # if specific word is in my word2vec model -> use index number. If not, put 0 instead of the words' index.
        reviews_ints.append([word2vec.wv.key_to_index[word] if word in  word2vec.wv.key_to_index else 0 for word in revie

    return reviews_ints
```

In [6]:
```
# Limit the maximum review length to 50
def pad_features(x, desired_len):
    for i, row in enumerate(x):
        if len(row) > desired_len: # Turncate longer reviews
            x[i] = row[:desired_len]
        elif len(row) < desired_len: # Padding shorter reviews with a '0'
            x[i] = row[:len(row)] + [0]*(desired_len-len(row))

    return x
```

In [7]:
```
# Split train data and test data
x_train, x_test, y_train, y_test = train_test_split(new['review_body'], new['class'].values, test_size=0.2, random_state =
```

In [8]:
```
# Change words to number values using my own word2vec-news model
new_x_train = review_to_int(tokenize(x_train))
new_x_train = np.array(pad_features(new_x_train, 50))

new_x_test = review_to_int(tokenize(x_test))
new_x_test = np.array(pad_features(new_x_test, 50))
```

In [9]:
```
from torch.utils.data import TensorDataset, DataLoader
# change data type to tensor
```

```python
X_train = torch.LongTensor(new_x_train)
X_test = torch.LongTensor(new_x_test)
Y_train = torch.LongTensor(y_train-1)
Y_test = torch.LongTensor(y_test-1)

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, Y_train)
ds_test = TensorDataset(X_test, Y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [10]:

```python
class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_size, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_size, batch_first=True, nonlinearity='relu')
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        out, _ = self.rnn(embedded)
        out = self.fc(out)
        return out
```

In [11]:

```python
INPUT_DIM = len(word2vec.wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [12]:

```python
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [13]:

```python
def train(epoch, batch_size):
    model.train()  # Model train
    epoch_loss = 0

    # Train model with mini batch
    for data, targets in loader_train:
```

```
        optimizer.zero_grad()  # Initiate
        outputs = model(data)  # model train and output
        loss = loss_fn(outputs, targets.reshape(1,batch_size).t())  # Calculate loss values (real value - predicted value)
        loss.backward()  # Backpropagation
        optimizer.step()  # Edit weights
        epoch_loss += loss.item()


    print('Cost: {:.6f}'.format(loss.item()))
```

In [14]:

```
def test(model, data_loader):
    model.eval()  # Test Model
    correct = 0

    # Create minibatch
    with torch.no_grad():
        for data, targets in loader_test:
            outputs = model(data)  # Put input data and get output data

            # Calculate correct case
            _, predicted = torch.max(outputs.data, 1)  # Calculate which label has the highest probability
            correct += predicted.eq(targets.data.view_as(predicted)).sum()  # If it matches the answer, increase the count

    # Print accuracy
    data_num = len(loader_test.dataset)
    print('₩nAccuracy with test data: {}/{} ({:.0f}%)₩n'.format(correct,data_num, 100. * correct / data_num))
```

In [16]:

```
for epoch in range(3): # I have computational resource limitations, especially memory issue, so I had to set epoch to 3. (e
    train(epoch, 64)
test(model, loader_test)
```

```
Cost: 0.617980
Cost: 0.593134
Cost: 0.516981

Accuracy with test data: 2630/4000 (66%)
```

(b-1-1) Binary model with GRU (a gated recurrent unit cell) using my word2vec model

In [17]:

```
class GRU(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_size, num_classes):
        super(GRU, self).__init__()
```

```python
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.gru = nn.GRU(embedding_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        out, _ = self.gru(embedded)
        out = self.fc(out)
        return out
```

In [18]:
```python
INPUT_DIM = len(word2vec.wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = GRU(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [19]:
```python
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [20]:
```python
for epoch in range(3):
    train(epoch, 64)
test(model, loader_test)
```

```
Cost: 0.777114
Cost: 0.658260
Cost: 0.614778

Accuracy with test data: 2562/4000 (64%)
```

In [ ]:
```python

```

(a-1-2) Binary model with RNN using my google-word2vec-model

In [21]:
```python
# load word2vec-google-news model
import gensim.downloader as api
google_wv = api.load('word2vec-google-news-300')
```

In [22]:
```python
# Change the tokenized reviews to int type(using my Word2vec model)
```

```python
def google_review_to_int(reviews):
    reviews_ints = []
    for review in reviews:
        # if specific word is in my word2vec model -> use index number. If not, put 0 instead of the words' index.
        reviews_ints.append([google_wv.key_to_index[word] if word in  google_wv.key_to_index else 0 for word in review])

    return reviews_ints
```

In [23]:
```python
# Change words to number values using google-word2vec-news model
new_x_train = google_review_to_int(tokenize(x_train))
new_x_train = np.array(pad_features(new_x_train, 50))

new_x_test = google_review_to_int(tokenize(x_test))
new_x_test = np.array(pad_features(new_x_test, 50))
```

In [24]:
```python
from torch.utils.data import TensorDataset, DataLoader
# change data type to tensor
X_train = torch.LongTensor(new_x_train)
X_test = torch.LongTensor(new_x_test)
Y_train = torch.LongTensor(y_train-1)
Y_test = torch.LongTensor(y_test-1)

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, Y_train)
ds_test = TensorDataset(X_test, Y_test)

loader_train = DataLoader(ds_train, batch_size=64, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=64, shuffle=False)
```

In [25]:
```python
INPUT_DIM = len(google_wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [26]:
```python
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [27]:
```python
for epoch in range(3):
    train(epoch, 64)
test(model, loader_test)
```

```
Cost: 0.693178
Cost: 0.511630
Cost: 0.574840

Accuracy with test data: 2565/4000 (64%)
```

(b-1-2) Binary model with GRU (a gated recurrent unit cell) using google-word2vec-model

In [28]:
```
INPUT_DIM = len(google_wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = GRU(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [29]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [30]:
```
for epoch in range(3):
    train(epoch, 64)
test(model, loader_test)
```

```
Cost: 0.742343
Cost: 0.570386
Cost: 0.552954

Accuracy with test data: 2625/4000 (66%)
```

(a-2-1) Ternary model with RNN using my own word2vec model

In [31]:
```
import pandas as pd
new = pd.read_csv('ternary_data.csv', index_col=0)
new.head()
```

Out[31]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	to

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	to
0	US	11903534	R3NBYY1PF9MXRT	B00FRMV38Y	872686823	Decodyne™ Morning Mug, Heat Sensitive Co...	Kitchen	1	0.0	
1	US	27885863	R1A10GP8CPG1A2	B003YFI0O6	111524501	Oster Electric Wine-Bottle Opener	Kitchen	1	1.0	
2	US	15355157	R147NFWDLR0AT9	B0002T4ZL4	978772977	Oggi 5355 4-Piece Acrylic Canister Set with Ai...	Kitchen	1	2.0	
3	US	15647704	R1GQQLPV9LCY1T	B0034J6QIY	591197834	Cuisinart SS-700 Single Serve Brewing System -...	Kitchen	1	0.0	
4	US	26424346	R1X5BB0UPZ4IWT	B000AXQA8I	330600737	Kuhn Rikon Twist and Chop, Artichoke	Kitchen	1	3.0	

In [32]:
```
# Split train data and test data
x_train, x_test, y_train, y_test = train_test_split(new['review_body'], new['class'].values, test_size=0.2, random_state =
```

In [33]:
```
# Change words to number values using my own word2vec-news model
new_x_train = review_to_int(tokenize(x_train))
new_x_train = np.array(pad_features(new_x_train, 50))

new_x_test = review_to_int(tokenize(x_test))
new_x_test = np.array(pad_features(new_x_test, 50))
```

In [34]:
```
from torch.utils.data import TensorDataset, DataLoader
```

```python
# change data type to tensor
X_train = torch.LongTensor(new_x_train)
X_test = torch.LongTensor(new_x_test)
Y_train = torch.LongTensor(y_train-1)
Y_test = torch.LongTensor(y_test-1)

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, Y_train)
ds_test = TensorDataset(X_test, Y_test)

loader_train = DataLoader(ds_train, batch_size=32, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=32, shuffle=False)
```

In [35]:

```python
INPUT_DIM = len(word2vec.wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [36]:

```python
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [37]:

```python
for epoch in range(3):
    train(epoch, 32)
test(model, loader_test)
```

```
Cost: 1.055819
Cost: 0.997521
Cost: 0.919905

Accuracy with test data: 2624/5000 (52%)
```

(b-2-1) Ternary model with GRU (a gated recurrent unit cell) using my own word2vec model

In [38]:

```python
INPUT_DIM = len(word2vec.wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1
```

```
model = GRU(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [39]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [41]:
```
for epoch in range(3):
    train(epoch, 32)
test(model, loader_test)
```

```
Cost: 0.891429
Cost: 1.189794
Cost: 0.805755

Accuracy with test data: 2627/5000 (53%)
```

(a-2-2) Ternary model with RNN using google-word2vec model

In [42]:
```
import pandas as pd
new = pd.read_csv('ternary_data.csv', index_col=0)
new.head()
```

Out[42]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	to
0	US	11903534	R3NBYY1PF9MXRT	B00FRMV38Y	872686823	Decodyne™ Morning Mug, Heat Sensitive Co...	Kitchen	1	0.0	
1	US	27885863	R1A10GP8CPG1A2	B003YFI0O6	111524501	Oster Electric Wine-Bottle Opener	Kitchen	1	1.0	
2	US	15355157	R147NFWDLR0AT9	B0002T4ZL4	978772977	Oggi 5355 4-Piece Acrylic Canister Set with Ai...	Kitchen	1	2.0	

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpful_votes	to
3	US	15647704	R1GQQLPV9LCY1T	B0034J6QIY	591197834	Cuisinart SS-700 Single Serve Brewing System -...	Kitchen	1	0.0	
4	US	26424346	R1X5BB0UPZ4IWT	B000AXQA8I	330600737	Kuhn Rikon Twist and Chop, Artichoke	Kitchen	1	3.0	

In [43]:

```
# Change words to number values using google-word2vec-news model
new_x_train = google_review_to_int(tokenize(x_train))
new_x_train = np.array(pad_features(new_x_train, 50))

new_x_test = google_review_to_int(tokenize(x_test))
new_x_test = np.array(pad_features(new_x_test, 50))
```

In [44]:

```
from torch.utils.data import TensorDataset, DataLoader
# change data type to tensor
X_train = torch.LongTensor(new_x_train)
X_test = torch.LongTensor(new_x_test)
Y_train = torch.LongTensor(y_train-1)
Y_test = torch.LongTensor(y_test-1)

# Make a dataset and dataloader
ds_train = TensorDataset(X_train, Y_train)
ds_test = TensorDataset(X_test, Y_test)

loader_train = DataLoader(ds_train, batch_size=32, shuffle=True)
loader_test = DataLoader(ds_test, batch_size=32, shuffle=False)
```

In [45]:

```
INPUT_DIM = len(google_wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [46]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [47]:
```
for epoch in range(3):
    train(epoch, 32)
test(model, loader_test)
```

```
Cost: 1.071386
Cost: 0.833799
Cost: 1.027805

Accuracy with test data: 2630/5000 (53%)
```

(b-2-2) Ternary model with GRU (a gated recurrent unit cell) using google-word2vec model

In [48]:
```
INPUT_DIM = len(google_wv)+1
EMBEDDING_DIM = 50
HIDDEN_DIM = 50
OUTPUT_DIM = 1

model = GRU(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

In [49]:
```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [50]:
```
for epoch in range(3):
    train(epoch, 32)
test(model, loader_test)
```

```
Cost: 1.103006
Cost: 0.943143
Cost: 1.102362

Accuracy with test data: 2563/5000 (51%)
```

Report

5. I have computational resource limitations, especially memory issue, so I had to set epoch=3.

5-a. [Simple Recurrent Neural Networks] (Limit the maximum length to 50) – 4 models

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	66%	52%
google-news-300	64%	53%

5-b. [A gated recurrent unit cell (GRU)] (Limit the maximum length to 50) – 4 models

Accuracy	Binary Classification	Ternary Classification
My Own Word2Vec	64%	53%
google-news-300	66%	51%

: The performance of RNN and GRU is not significantly different. However, I have computational resource limitations, especially memory issue, so I had to set epoch to 3. (epoch=3). If I have resources, I would like to increase the amount of epoch, which might get different results.