Python Version 3.7.0

Packages version

1. PyTorch - 1.12.1
2. Gensim - 4.2.0
3. Pandas - 1.1.5
4. Numpy - 1.21.6
5. Emoji - 2.0.0
6. Textblob - 0.15.3
7. NLTK - 3.7
8. Sklearn - 1.0.2

Packages required for executing code

In [1]:
```
import pandas as pd
import numpy as np
from numpy.linalg import norm
import re
from unicodedata import normalize
import emoji
import contractions
from textblob import TextBlob
from textblob import Word

import gensim.downloader as api
from gensim.test.utils import datapath
from gensim import utils
import gensim.models
import nltk

from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report
from sklearn.linear_model import Perceptron
from sklearn.pipeline import make_pipeline
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import MaxAbsScaler

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torch.utils.data import Dataset

import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
wv = api.load('word2vec-google-news-300')
```

1. Dataset Generation

This step includes Downloading dataset and Dataset cleaning. Dataset cleaning techniques used are similar to the ones used in HW1. This is to ensure effective comparison of model performances. Cleaning techniques used -

1. Drop nan values
2. Convert column data types to string(reviews) and integer(rating)
3. Select 100k samples such that 20k of the samples belong to each rating class
4. Remove html tags
5. Remove external links
6. Remove extra white spaces
7. Remove accents
8. Replace emojis with text
9. Replace contractions and Spell correction

Read data from data.tsv and drop columns that are not necessary

In [3]:
```
filepath =  "data.tsv"

amazon_reviews = pd.read_csv(filepath, sep = "\t",
                             skip_blank_lines = True,
                             error_bad_lines=False,
                             warn_bad_lines = False)
```

In [4]:
```
amazon_reviews.drop(
    columns=['marketplace',
             'customer_id',
             'review_id',
             'product_id',
             'product_title',
             'product_category',
             'helpful_votes',
             'total_votes',
             'vine',
             'verified_purchase',
             'review_headline',
             'review_date',
             'product_parent'], inplace = True)


amazon_reviews.rename(columns=
                      {'review_body': 'reviews',
                       'star_rating': 'star_rating'},
                      inplace=True)
```

Drop nan values

In [5]:
```
amazon_reviews = amazon_reviews[amazon_reviews['star_rating'].notna()]

amazon_reviews = amazon_reviews[amazon_reviews['reviews'].notna()]
```

Convert column data types to string(reviews) and integer(rating)

In [6]:
```
amazon_reviews['star_rating'] = pd.to_numeric(
    amazon_reviews['star_rating']).astype('Int64')

amazon_reviews.star_rating.unique()
```

Out[6]:
```
<IntegerArray>
[5, 1, 4, 3, 2]
Length: 5, dtype: Int64
```

Select 100k samples such that 20k of the samples belong to each rating class

In [7]:
```
sample_count = 20000

sampled_reviews = amazon_reviews.groupby('star_rating').apply(
    lambda x: x.sample(n=sample_count,
                       random_state=42)).reset_index(drop = True)

sampled_reviews.groupby(['star_rating'])['star_rating'].count()
```

Out[7]:
```
star_rating
1    20000
2    20000
3    20000
4    20000
5    20000
Name: star_rating, dtype: int64
```

Remove html tags

In [8]:
```
sampled_reviews['reviews'] = sampled_reviews['reviews'].str.replace(
    r'<[^<>]*>', ' ', regex=True)
```

Remove external links

In [9]:
```
sampled_reviews['reviews'] = sampled_reviews['reviews'].apply(
    lambda x: re.sub(r'http\S+', ' ', str(x)))
```

Remove extra white spaces

In [10]:
```
sampled_reviews['reviews'] = sampled_reviews['reviews'].str.strip()

sampled_reviews['reviews'] = sampled_reviews['reviews'].replace(
    r'\s+', ' ', regex=True)
```

Remove accents

In [11]:
```
remove_accent = lambda text: normalize("NFKD", text).encode(
    "ascii", "ignore").decode(
    "utf-8", "ignore")

sampled_reviews["reviews"] = sampled_reviews["reviews"].apply(
    remove_accent)
```

Replace emojis with text

In [12]:
```
sampled_reviews['reviews'] = sampled_reviews['reviews'].apply(
    lambda x: emoji.demojize(x))

sampled_reviews["reviews"].replace(to_replace=r';\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';-\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':-\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':\'\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';\'\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';*\(',
                                   value='sad, angry',
                                   regex=True, inplace = True)


sampled_reviews["reviews"].replace(to_replace=r':\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':\'\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':-\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';-\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';-\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':o\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';o\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';0\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';O\)',
                                   value='Happy', regex=True,
                                   inplace = True)
```

```
sampled_reviews["reviews"].replace(to_replace=r':=\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';=\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':^\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':d\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':0\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r':~\)',
                                   value='Happy', regex=True,
                                   inplace = True)
sampled_reviews["reviews"].replace(to_replace=r';*\)',
                                   value='Happy', regex=True,
                                   inplace = True)
```

Replace contractions and Spell correction

In [13]:
```
sampled_reviews['reviews'] = sampled_reviews['reviews'].apply(
    lambda x: contractions.fix(x))

def remove_contractions(phrase):
    phrase = re.sub(r"won\'t", "will not", phrase)
    phrase = re.sub(r"can\'t", "can not", phrase)
    phrase = re.sub(r"n\'t", " not", phrase)
    phrase = re.sub(r"\'re", " are", phrase)
    phrase = re.sub(r"\'s", " is", phrase)
    phrase = re.sub(r"\'d", " would", phrase)
    phrase = re.sub(r"\'ll", " will", phrase)
    phrase = re.sub(r"\'t", " not", phrase)
    phrase = re.sub(r"\'ve", " have", phrase)
    phrase = re.sub(r"\'m", " am", phrase)
    return phrase


sampled_reviews['reviews'] = sampled_reviews['reviews'].apply(
    remove_contractions)


sampled_reviews['reviews'] = sampled_reviews['reviews'].str.replace(
    '[^a-zA-Z ]', ' ')
sampled_reviews['reviews'] = sampled_reviews['reviews'].str.strip()
sampled_reviews['reviews'] = sampled_reviews['reviews'].replace(
    r'\s+', ' ', regex=True)

rx = re.compile(r'([^\W\d_])\1{2,}')

def word_correction(text):
    correct = re.sub(r'[^\W\d_]+',
                     lambda x: Word(
                         rx.sub(r'\1\1', x.group())).correct()
                     if rx.search(
                         x.group()) else x.group(), text)
    return correct

sampled_reviews['reviews'] = sampled_reviews['reviews'].apply(
    word_correction)
```

2. Word Embedding

a) “word2vec-google-news-300” Word2Vec model

Following are tried to check semantic similarities

1. weak + break ~ flimsy
2. terrific ~ superb
3. woman + royal ~ princess
4. king - man + woman ~ queen
5. execellent ~ outstanding

weak + break ~ flimsy

In [14]:
```
vec_weak = wv['weak']
vec_break = wv['break']
vec_flimsy = wv['flimsy']

result = vec_weak + vec_break

print(np.dot(result,vec_flimsy)/(norm(result)*norm(vec_flimsy)))
print(wv.most_similar(positive=[result], topn=5))
```

```
0.31923553
[('weak', 0.8075981736183167), ('break', 0.7346563339233398), ('weaker',
0.6097529530525208), ('Weak', 0.5651276707649231), ('weakening', 0.561063
2300376892)]
```

terrific ~ superb

In [15]:
```
vec_terrific = wv["terrific"]
vec_superb = wv["superb"]

print(np.dot(vec_terrific,vec_superb)/(norm(vec_terrific)*norm(vec_superb))
```

```
0.7476986
```

woman + royal ~ princess

In [16]:
```
vec_woman = wv["woman"]
vec_royal = wv["royal"]
vec_princess = wv["princess"]

result = vec_woman + vec_royal

print(np.dot(result,vec_princess)/(norm(result)*norm(vec_princess)))
```

```
0.63185287
```

king - man + woman ~ queen

In [17]:
```
vec_king = wv["king"]
vec_man = wv["man"]
vec_queen = wv["queen"]
vec_woman = wv["woman"]

result = vec_king - vec_man + vec_woman

print(np.dot(result,vec_queen)/(norm(result)*norm(vec_queen)))
```

```
0.73005176
```

execellent ~ outstanding

In [18]:
```
vec_excellent = wv["excellent"]
vec_outstanding = wv["outstanding"]

print(np.dot(vec_excellent,vec_outstanding)/(norm(vec_excellent)*norm(vec_o
```

```
0.5567486
```

b) Train a Word2Vec model using your own dataset. Set the embedding size to be 300 and the window size to be 11. You can also consider a minimum word count of 10. Check the semantic similarities for the same two examples in part (a). 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? For the rest of this assignment, use the pretrained “word2vec-googlenews-300” Word2Ve features.

In [19]:
```
class JewellryReviewCorpus:
    def __iter__(self):
        for index, row in sampled_reviews.iterrows():
            yield utils.simple_preprocess(row['reviews'])
```

In [20]:
```
sentences = JewellryReviewCorpus()
jewellry_review_model = gensim.models.Word2Vec(
    sentences=sentences, window=11, vector_size=300, min_count = 10)
```

weak + break ~ flimsy

In [21]:
```
vec_weak = jewellry_review_model.wv['weak']
vec_break = jewellry_review_model.wv['break']
vec_flimsy = jewellry_review_model.wv['flimsy']
result = vec_weak + vec_break
print(np.dot(result,vec_flimsy)/(norm(result)*norm(vec_flimsy)))
print(jewellry_review_model.wv.most_similar(positive=[result], topn=5))
```

```
0.64387095
[('break', 0.9472134709358215), ('bend', 0.7155886292457581), ('weak', 0.
6939854621887207), ('snap', 0.6832017302513123), ('breaks', 0.68019121885
29968)]
```

terrific ~ superb

In [22]:
```
vec_terrific = jewellry_review_model.wv["terrific"]
vec_superb = jewellry_review_model.wv["superb"]

print(np.dot(vec_terrific,vec_superb)/(norm(
    vec_terrific)*norm(vec_superb)))
```

```
0.41385764
```

woman + royal ~ princess

In [23]:
```
vec_woman = jewellry_review_model.wv["woman"]
vec_royal = jewellry_review_model.wv["royal"]
vec_princess = jewellry_review_model.wv["princess"]

result = vec_woman + vec_royal

print(np.dot(result,vec_princess)/(norm(result)*norm(vec_princess)))
```

```
0.31565464
```

king - man + woman ~ queen

In [24]:
```
vec_king = jewellry_review_model.wv["king"]
vec_man = jewellry_review_model.wv["man"]
vec_queen = jewellry_review_model.wv["queen"]
vec_woman = jewellry_review_model.wv["woman"]

result = vec_king - vec_man + vec_woman

print(np.dot(result,vec_queen)/(norm(result)*norm(vec_queen)))
```

```
0.276741
```

execellent ~ outstanding

In [25]:
```
vec_excellent = jewellry_review_model.wv["excellent"]
vec_outstanding = jewellry_review_model.wv["outstanding"]

print(np.dot(vec_excellent,vec_outstanding)/(
    norm(vec_excellent)*norm(vec_outstanding)))
```

```
0.8027048
```

Summary and Observations -

Following is summary of various semantic similarities that were tested with Google Word2Vec and Amazon Reviews Dataset Word2Vec.

Various words were tried to check if the models were able to capture semantic similarities. Cosine similarity was used

Semantics	Google Word2Vec	Amazon Review Dataset Word2Vec
weak + break ~ flimsy	0.31923553	0.64406204
terrific ~ superb	0.7476986	0.41486466
woman + royal ~ princess	0.63185287	0.31566978
king - man + woman ~ queen	0.73005176	0.27788362
execellent ~ outstanding	0.73005176	0.8023542

weak + break ~ flimsy

These 3 words occur a lot in Amazon review dataset and mostly co-occur in several reviews. Because of this, we can see that Amazon Review Dataset Word2Vec returned vectors that have high similarity score(0.64) for weak + break and flimsy. But when the same is sent through Google Word2Vec, it has lower score of 0.31.

When we call most_similar function for Google Word2Vec for weak + break, the words returned were weak, Weak, breaks etc. But when same was tried with Amazon Review Dataset Word2Vec, it returned break, bend, snap etc. The word and its context play major role in identifying word semantic similarities.

terrific ~ superb, woman + royal ~ princess, king - man + woman ~ queen

Amazon Review Dataset Word2Vec is trained on limited vocabulary related to reviews regarding Jewellery products. Hence its ability to perform well on generic words is less. For the above examples which include generic words(words that are not specific to jewellery in general), Google Word2Vec was able to capture the word similarity for the words and returned vectors that had high cosine similarity whereas Amazon Review Dataset Word2Vec was not able to capture the word similarity or semantic similarity and gave low scores

execellent ~ outstanding

This set of words are present in both the vocabularies and it can be seen that both Google Word2Vec and Amazon Reviews Dataset Word2Vec have been able to capture the semantic similarities of the words. A cosine similarity of 0.73 and 0.80 was obtained showing high similarities in the words

Train and Test Split

1. We first tokenize the reviews and store it in a new dataframe column tokenized_reviews.
2. gensim.utils.simple_preprocess function is used to data clean and tokenize reviews. This also converts words to lower case
3. The data is split in 80% train and 20% test

Train and Test split is done at the beginning so that all the models can be tested against same train samples and test samples. This will also help us understand and compare model accuracies easily.

In [26]:
```
sampled_reviews['reviews'] = sampled_reviews['reviews'].astype(str)
sampled_reviews['tokenized_reviews'] = sampled_reviews['reviews'].apply(
    lambda x: gensim.utils.simple_preprocess(x))
sampled_reviews.head()
```

Out[26]:

	star_rating	reviews	tokenized_reviews
0	1	These are horrible Very cloudy no shine at all...	[these, are, horrible, very, cloudy, no, shine...
1	1	I returned it for it was too thin and the clos...	[returned, it, for, it, was, too, thin, and, t...
2	1	Nothing like the picture studs were as big as ...	[nothing, like, the, picture, studs, were, as,...
3	1	I ordered this item paying extra for Guarantee...	[ordered, this, item, paying, extra, for, guar...
4	1	I had seen something very similar to this item...	[had, seen, something, very, similar, to, this...

In [27]:
```
train_count = 16000
train_data = sampled_reviews.groupby('star_rating').apply(
    lambda x: x.sample(n=train_count, random_state=42))
train_index_tuple_list = train_data.index.values.tolist()
train_index_list = [x[1] for x in train_index_tuple_list]
test_data = sampled_reviews[~sampled_reviews.index.isin(train_index_list)]
train_data.reset_index(drop=True, inplace = True)
test_data.reset_index(drop=True, inplace = True)
```

In [28]:
```
words_google = set(wv.index_to_key)
```

3. Simple models

Input word2Vec for SVM and Perceptron

1. First convert each word in the tokenized_reviews column to word vector using Google Word2Vec pretrained model for both test and train datasets. Each word here is now represented by an array of size 300
2. We then take average of the word vectors. The averaged vector is the representation of entire review. Each review now is represented by an array of size 300. This is performed for both test and train datasets

In [29]:
```
X_train_vect = np.array(
    [np.array(
        [wv[i] for i in ls if i in words_google]
    ) for ls in train_data["tokenized_reviews"]])
X_test_vect = np.array(
    [np.array(
        [wv[i] for i in ls if i in words_google]
    ) for ls in test_data["tokenized_reviews"]])
```

In [30]:
```
X_train_vect_avg = []
for v in X_train_vect:
    if v.size:
        X_train_vect_avg.append(v.mean(axis=0))
    else:
        X_train_vect_avg.append(np.zeros(300, dtype=float))

X_test_vect_avg = []
for v in X_test_vect:
    if v.size:
        X_test_vect_avg.append(v.mean(axis=0))
    else:
        X_test_vect_avg.append(np.zeros(300, dtype=float))
```

In [31]:
```
y_train = train_data["star_rating"].astype('int').to_numpy()
y_test = test_data["star_rating"].astype('int').to_numpy()
```

SVM

LinearSVC function from sklearn python package is used. Input to SVM is the averaged word2Vec that was generated in previous step Parameters set for the model are -

1. random_state=12
2. tol=1e-3
3. class_weight = 'balanced'

In [32]:
```
svm_linear_clf = LinearSVC(
    random_state=12,
    tol=1e-3,
    class_weight = 'balanced')
svm_linear_clf.fit(X_train_vect_avg, y_train)

y_test_pred = svm_linear_clf.predict(X_test_vect_avg)
report = classification_report(y_test, y_test_pred, output_dict=True)
svm_wordvec_accuracy = report["accuracy"]
report
```

Out[32]:
```
{'1': {'precision': 0.5040181691125087,
  'recall': 0.72125,
  'f1-score': 0.5933772110242698,
  'support': 4000},
 '2': {'precision': 0.3791291291291291,
  'recall': 0.2525,
  'f1-score': 0.30312124849939975,
  'support': 4000},
 '3': {'precision': 0.39994532531437943,
  'recall': 0.36575,
  'f1-score': 0.3820840950639854,
  'support': 4000},
 '4': {'precision': 0.4425026214610276,
  'recall': 0.3165,
  'f1-score': 0.3690424136423262,
  'support': 4000},
 '5': {'precision': 0.5949342234439426,
  'recall': 0.7575,
  'f1-score': 0.6664467172550314,
  'support': 4000},
 'accuracy': 0.4827,
 'macro avg': {'precision': 0.46410589369219757,
  'recall': 0.4827,
  'f1-score': 0.4628143370970025,
  'support': 20000},
 'weighted avg': {'precision': 0.4641058936921975,
  'recall': 0.4827,
  'f1-score': 0.4628143370970025,
  'support': 20000}}
```

Perceptron

Perceptron function from sklearn python package is used. Input to Perceptron is the averaged word2Vec that was generated in previous step Parameters set for the model are -

1. random_state=1
2. tol=1e-5
3. max_iter = 30000
4. validation_fraction = 0.1

In [78]:
```
perceptron_clf = Perceptron(
    random_state=1,
    tol=1e-5,
    max_iter = 30000,
    validation_fraction = 0.1
)
perceptron_clf.fit(X_train_vect_avg, y_train)

y_test_pred = perceptron_clf.predict(X_test_vect_avg)
report = classification_report(
    y_test,
    y_test_pred,
    output_dict = True)
perceptron_wordvec_accuracy = report["accuracy"]
report
```

Out[78]:
```
{'1': {'precision': 0.5212617288880016,
  'recall': 0.65275,
  'f1-score': 0.5796425796425797,
  'support': 4000},
 '2': {'precision': 0.3736434108527132,
  'recall': 0.1205,
  'f1-score': 0.1822306238185255,
  'support': 4000},
 '3': {'precision': 0.32136268601648016,
  'recall': 0.65325,
  'f1-score': 0.430797131316462,
  'support': 4000},
 '4': {'precision': 0.47985347985347987,
  'recall': 0.03275,
  'f1-score': 0.06131523519775334,
  'support': 4000},
 '5': {'precision': 0.5586180857088918,
  'recall': 0.73975,
  'f1-score': 0.6365494245455523,
  'support': 4000},
 'accuracy': 0.4398,
 'macro avg': {'precision': 0.45094787826391336,
  'recall': 0.43979999999999997,
  'f1-score': 0.37810699890417454,
  'support': 20000},
 'weighted avg': {'precision': 0.4509478782639133,
  'recall': 0.4398,
  'f1-score': 0.37810699890417454,
  'support': 20000}}
```

What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features)

Both the models were run on same training and testing dataset. Also, same preprocessing techniques were applied on both dataset. This was to ensure that both model performances can be compared directly with each other.

Details regarding input features that were compared against in the Experiment -

TF-IDF - Term frequency Inverse Document Frequency is numerical statistic that provides us with measure of how important a word is to a document given a corpus. It builds on a simple idea which is bag of words.

Word2Vec - Word2Vec is implementation of advanced techniques such Continuous bag of words and Skipgram. It takes in dataset as input. It first creates vocabulary and then generates vectors to represent them. In this process, it also encodes word similarities, semantic similarities and word co-occurences.

SVM Model

Following is the summary of the SVM model performance with TF-IDF and Word2Vec as input features

Measure	TF-IDF	Word2Vec
Accuracy	0.50	0.4827
Precision	0.4870	0.4641
Recall	0.501	0.4827
F1 Score	0.4895	0.4628

It can be seen that irrespective of whether the input features were generated through simple techniques or advanced techniques, Support Vector Machine model has performed more or less the same. Infact, the accuracy, precision, recall and F1-Score obtained with simple technique TF-IDF is slightly greater than the one that was tried with Word2vec.

Perceptron Model

Following is the summary of the Perceptron model performance with TF-IDF and Word2Vec as input features

Measure	TF-IDF	Word2Vec
Accuracy	0.4232	0.4398
Precision	0.4232	0.4509
Recall	0.4257	0.4397
F1 Score	0.4238	0.3781

Similar results as SVM are seen in Perceptron as well. Irrespective of technique used for input feature word representation, Perceptron model has performed similar in both cases. We can notice that for tf-idf input features, the values of accuracy, precision, recall and f1 scores are consistent. For word2Vec input, the f1 score is on lower end.

The overall learning from this -

1. TF-IDF features were extracted based on Amazon Review Dataset and was catered to pick work context, co-occurences and similarities from it. And probably this could be one of the reasons why it performed better when compared to Google word2Vec input features which is generic and not catered for Amazon dataset specifically.

2. Advanced or complex techniques don't really mean better performance or outcomes.

4. Feedforward Neural Networks

a) To generate the input features, use the average Word2Vec vectors similar to the “Simple models” section and train the neural network. Report accuracy values on the testing split for your MLP.

The following steps are performed on both train dataset and test dataset

1. Tokenized reviews are converted to word2Vectors first. Each word is a vector of size 300
2. The vectors for each review are averaged. After this step, each review is represented by a single vector of size 300
3. If there are any samples with review vectors as nan/null, these are identified and dropped

In [34]:
```
fnn_X_train = train_data.copy()
fnn_X_test = test_data.copy()

fnn_X_train["fnn_vectors"] = fnn_X_train['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
fnn_X_train['averaged_vectors'] = fnn_X_train.loc[:, 'fnn_vectors']
indexlist = []

for index, row in fnn_X_train.iterrows():
    count = 0
    vectors = np.zeros(300)
    vectors_list = row['fnn_vectors']
    if vectors_list is not None and len(vectors_list) >= 1:
        for v in vectors_list:
            count = count + 1
            vectors = np.add(vectors, v)
        average_vector = np.divide(vectors, len(vectors_list))
        fnn_X_train.at[index, "averaged_vectors"] = average_vector
    else:
        indexlist.append(index)

fnn_X_train = fnn_X_train[~fnn_X_train.index.isin(indexlist)]

fnn_X_test["fnn_vectors"] = fnn_X_test['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
fnn_X_test['averaged_vectors'] = fnn_X_test.loc[:, 'fnn_vectors']
indexlist = []

for index, row in fnn_X_test.iterrows():
    count = 0
    vectors = np.zeros(300)
    vectors_list = row['fnn_vectors']
    if vectors_list is not None and len(vectors_list) >= 1:
        for v in vectors_list:
            count = count + 1
            vectors = np.add(vectors, v)
        average_vector = np.divide(vectors, len(vectors_list))
        fnn_X_test.at[index, "averaged_vectors"] = average_vector
    else:
        indexlist.append(index)

fnn_X_test = fnn_X_test[~fnn_X_test.index.isin(indexlist)]

fnn_X_train.reset_index(drop=True, inplace = True)
fnn_X_test.reset_index(drop=True, inplace = True)
```

FNNTensorDataset and Dataloader

1. It has helper function **getitem** that fetches item sample from dataframe based on index value.
2. Both train and test dataset are casted to FNNTensorDataset class
3. This is done so as to be able to define DataLoader
4. A dataloader takes in FNNTensorDataset class type and generates batches (batch size is set to 100) of data that are selected randomly (SubsetRandomSampler) and returns it.

In [35]:
```
class FNNTensorDataset(Dataset):
    def __init__(self, dataframe):
        self.data = dataframe

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        features = self.data.loc[index, 'averaged_vectors']
        label = self.data.loc[index, 'star_rating']
        return torch.from_numpy(features).float(), label - 1

    def __getindexlist__(self):
        return list(self.data.index.values)

fnn_X_train_tensor = FNNTensorDataset(fnn_X_train)
fnn_X_test_tensor = FNNTensorDataset(fnn_X_test)

num_of_workers = 0
batch_size = 100
valid_size = 0.2

indices = list(range(len(fnn_X_train_tensor)))
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(
    fnn_X_train_tensor,
    batch_size=batch_size,
    sampler=SubsetRandomSampler(indices)
)
```

FNNNet Module

1. We have defined a Feedforward Neural Network model with 2 hidden layers of size 50 and 10
2. As the input is averaged vector, the size of input layer is 300
3. As the output is classification of reviews into 5 ratings, the output layer size is 5

Cross Entropy loss is used.

For Optimizer, Stochastic Gradient Descent and Adam were tried with different learning rates (0.01, 0.001, 0.005, 0.0001, 0.0005). The best accuracy was achieved with Adam optimizer with a learning rate of 0.0005

In [36]:
```
class FNNNet(nn.Module):
    def __init__(self):
        super(FNNNet, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        self.fc1 = nn.Linear(1*300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 5)

    def forward(self, x):
        x = x.view(-1, 1*300)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x))
        return x

fnn = FNNNet()
print(fnn)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(fnn.parameters(), lr=0.0005)
```

```
FNNNet(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=5, bias=True)
)
```

Training on the dataset is done for 100 epochs

In [37]:
```
n_epochs = 100

for epoch in range(n_epochs):
    torch.manual_seed(42)
    train_loss = 0.0
    valid_loss = 0.0

    fnn.train()

    for data, target in train_loader:
        optimizer.zero_grad()
        output = fnn(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)

    fnn.eval()

    if(epoch != 0 and epoch%20 == 0):
        train_loss = train_loss/len(train_loader.dataset)
        print("Completed: " + str(epoch))
```

```
Completed: 20
Completed: 40
Completed: 60
Completed: 80
```

Predict function

1. The predict function takes in the model and dataloader.
2. The function iterates through every sample in dataloader and makes prediction using the model
3. For each prediction, the output is vector of size 5. The index of the highest value in the vector is selected
4. A list of predictions and actual data values is created and returned back

In [38]:
```
def predict_fnn(fnn_model, dataloader):
    prediction_list = []
    actual_list = []
    for data, target in dataloader:
        outputs = fnn_model(data)
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted.cpu())
        actual_list.append(target)
    return prediction_list, actual_list
```

Test Accuracy

1. Test Loader is created with test dataset
2. A function call to predict_fnn is made with model and test loader

3. The predictions and actual labels obtained are compared with each other using sklearn classification_report
4. Accuracy for the model is reported

In [39]:
```
test_loader = torch.utils.data.DataLoader(
    fnn_X_test_tensor,
    batch_size=fnn_X_test_tensor.__len__())
predictions, actuals = predict_fnn(fnn, test_loader)
predictions = predictions[0].numpy()
actuals = actuals[0].numpy()
report = classification_report(
    actuals,
    predictions,
    output_dict=True)
print("Accuracy of FNN model with Average Vectors : " + str(
    report["accuracy"]*100) + "%\n")
print("Classification Report")
fnna_wordvec_average_accuracy = report["accuracy"]
report
```

```
Accuracy of FNN model with Average Vectors : 49.76732549412059%

Classification Report
```

Out[39]:
```
{'0': {'precision': 0.5589732711073399,
  'recall': 0.65875,
  'f1-score': 0.6047739270140005,
  'support': 4000},
 '1': {'precision': 0.38153930430019173,
  'recall': 0.3485985985985986,
  'f1-score': 0.3643258794298418,
  'support': 3996},
 '2': {'precision': 0.40625776011919545,
  'recall': 0.4092046023011506,
  'f1-score': 0.4077258566978193,
  'support': 3998},
 '3': {'precision': 0.4492753623188406,
  'recall': 0.41116116116116114,
  'f1-score': 0.42937410165947987,
  'support': 3996},
 '4': {'precision': 0.6704776422764228,
  'recall': 0.660575719649562,
  'f1-score': 0.6654898499558694,
  'support': 3995},
 'accuracy': 0.4976732549412059,
 'macro avg': {'precision': 0.4933046680243981,
  'recall': 0.49765801634209444,
  'f1-score': 0.4943379229514021,
  'support': 19985},
 'weighted avg': {'precision': 0.49330023508080384,
  'recall': 0.4976732549412059,
  'f1-score': 0.494342795003278,
  'support': 19985}}
```

b) To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature (x = [WT1, ..., WT10]) and train the neural network. Report the accuracy value on the testing split for your MLP model. What do you conclude by comparing accuracy values you obtain with those obtained in the “’Simple Models” section.

The following steps are performed on both train dataset and test dataset

1. Tokenized reviews are converted to word2Vectors first. Each word is a vector of size 300
2. Each review is truncated to have only 10 words
3. The vectors are then concatenated. If the size of vectors is less than 300*10, then zeros are appended at the end
4. After this step, each review is represented by a single vector of size 300*10

In [40]:
```
fnn_X_train = train_data.copy()
fnn_X_test = test_data.copy()

fnn_X_train["fnn_vectors"] = fnn_X_train['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
fnn_X_train["reduced_reviews"] = fnn_X_train['fnn_vectors'].apply(
    lambda x: x[:10])

for index, row in fnn_X_train.iterrows():
    data_list = row['reduced_reviews']
    if data_list is None:
        fnn_X_train.at[index, "reduced_reviews"] = np.asarray(
            [np.zeros(300)] * 10)
    elif len(data_list) < 10:
        data_list.extend([np.zeros(300)] * (10-len(data_list)))
        fnn_X_train.at[index, "reduced_reviews"] = np.asarray(
            data_list)
    else:
        fnn_X_train.at[index, "reduced_reviews"] = np.asarray(
            data_list)
fnn_X_train["star_rating"] = fnn_X_train["star_rating"].astype(
    'int').to_numpy()


fnn_X_test["fnn_vectors"] = fnn_X_test['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
fnn_X_test["reduced_reviews"] = fnn_X_test['fnn_vectors'].apply(
    lambda x: x[:10])

for index, row in fnn_X_test.iterrows():
    data_list = row['reduced_reviews']
    if data_list is None:
        fnn_X_test.at[index, "reduced_reviews"] = np.asarray(
            [np.zeros(300)] * 10)
    elif len(data_list) < 10:
        data_list.extend([np.zeros(300)] * (10-len(data_list)))
        fnn_X_test.at[index, "reduced_reviews"] = np.asarray(
            data_list)
    else:
        fnn_X_test.at[index, "reduced_reviews"] = np.asarray(
            data_list)
fnn_X_test["star_rating"] = fnn_X_test["star_rating"].astype(
    'int').to_numpy()
```

FNNBTensorDataset and Dataloader

1. It has helper function **getitem** that fetches item sample from dataframe based on index value.
2. Both train and test dataset are casted to FNNBTensorDataset class
3. This is done so as to be able to define DataLoader
4. A dataloader takes in FNNBTensorDataset class type and generates batches (batch size is set to 100) of data that are selected randomly (SubsetRandomSampler) and returns it.

In [41]:
```
class FNNBTensorDataset(Dataset):
    def __init__(self, dataframe):
        self.data = dataframe

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        features = self.data.loc[index, 'reduced_reviews']
        label = self.data.loc[index, 'star_rating']
        return torch.from_numpy(features).float(), label - 1

    def __getindexlist__(self):
        return list(self.data.index.values)

fnn_X_train_tensor = FNNBTensorDataset(fnn_X_train)
fnn_X_test_tensor = FNNBTensorDataset(fnn_X_test)

num_of_workers = 0
batch_size = 100
valid_size = 0.2

indices = list(range(len(fnn_X_train_tensor)))
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(
    fnn_X_train_tensor,
    batch_size=batch_size,
    sampler=SubsetRandomSampler(indices)
)
```

FNNBNet Module

1. We have defined a Feedforward Neural Network model with 2 hidden layers of size 50 and 10
2. As the input is concatenated vectors, the size of input layer is 3000
3. As the output is classification of reviews into 5 ratings, the output layer size is 5

Cross Entropy loss is used.

For Optimizer, Stochastic Gradient Descent and Adam were tried with different learning rates (0.01, 0.001, 0.005, 0.0001, 0.0005). The best accuracy was achieved with Adam optimizer with a learning rate of 0.0005

In [42]:
```
class FNNBNet(nn.Module):
    def __init__(self):
        super(FNNBNet, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        self.fc1 = nn.Linear(10*300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 10)

    def forward(self, x):
        x = x.view(-1, 10*300)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# initialize the NN
fnnb = FNNBNet()
print(fnnb)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(fnnb.parameters(), lr=0.0005)
```

```
FNNBNet(
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=10, bias=True)
)
```

Training on the dataset is done for 4 epochs

In [43]:
```
n_epochs = 4

for epoch in range(n_epochs):
    torch.manual_seed(42)
    train_loss = 0.0

    fnnb.train()

    for data, target in train_loader:
        optimizer.zero_grad()
        output = fnnb(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)

    fnnb.eval()

    train_loss = train_loss/len(train_loader.dataset)

    if(epoch != 0 and epoch%2 == 0):
        train_loss = train_loss/len(train_loader.dataset)
        print("Completed: " + str(epoch))
```

```
Completed: 2
```

Test Accuracy

1. Test Loader is created with test dataset
2. A function call to predict_fnn is made with model and test loader
3. The predictions and actual labels obtained are compared with each other using sklearn classification_report
4. Accuracy for the model is reported

In [44]:
```
test_loader = torch.utils.data.DataLoader(
    fnn_X_test_tensor,
    batch_size=fnn_X_test_tensor.__len__())
predictions, actuals = predict_fnn(fnnb, test_loader)
predictions = predictions[0].numpy()
actuals = actuals[0].numpy()
report = classification_report(
    actuals,
    predictions,
    output_dict=True)
print("Accuracy of FNN model with Concatenated Vectors : " + str(
    report["accuracy"]*100) + "%\n")
fnnb_wordvec_cat_accuracy = report["accuracy"]
print("Classification Report")
report
```

```
Accuracy of FNN model with Concatenated Vectors : 43.26%

Classification Report
```

Out[44]:
```
{'0': {'precision': 0.4684126349460216,
  'recall': 0.58575,
  'f1-score': 0.5205509886691846,
  'support': 4000},
 '1': {'precision': 0.33868894601542415,
  'recall': 0.2635,
  'f1-score': 0.296400449943757,
  'support': 4000},
 '2': {'precision': 0.3593204561321853,
  'recall': 0.386,
  'f1-score': 0.3721827166445703,
  'support': 4000},
 '3': {'precision': 0.4092773745661092,
  'recall': 0.32425,
  'f1-score': 0.3618356814060538,
  'support': 4000},
 '4': {'precision': 0.5461538461538461,
  'recall': 0.6035,
  'f1-score': 0.573396674584323,
  'support': 4000},
 'accuracy': 0.4326,
 'macro avg': {'precision': 0.42437065156271725,
  'recall': 0.43260000000000004,
  'f1-score': 0.42487330224957776,
  'support': 20000},
 'weighted avg': {'precision': 0.4243706515627173,
  'recall': 0.4326,
  'f1-score': 0.42487330224957776,
  'support': 20000}}
```

What do you conclude by comparing accuracy values you obtained with FeedForward Neural Networks model with those obtained in the "'Simple Models" section.

Both the models were run on same training and testing dataset. Also, same preprocessing techniques were applied on both dataset. This was to ensure that all model performances can be compared directly with each other.

Following is the summary of the SVM, Perceptron and FNN model performances with TF-IDF and Word2Vec as input features

Model	Accuracy
SVM	0.4827
Perceptron	0.4398
FNN with average word vectors	0.4976
FNN with concatenated word vectors	0.4326

SVM, Perceptron and FNN models were tried with average word2Vec input features. Among these, SVM and FNN have performed well and have almost same accuracy with slight difference. FNN has been able to perform 2% better when compared to SVM. But given the computation resources that FNN might require for larger dataset, its a trade off between accuracy and computational resources that one has to make.

FNN was also tried by considering 10 words from each review concatenated together. The first 10 words were considered by checking if they existed in the Google Word2Vec dictionary. If the word didn't exist, it was discarded. All reviews that were less than 10 word length were padded with zeros vectors. The performance of this model was very low compared to SVM and FNN that were tried with average word2Vec.

Another observation is that models with average word vectors as input have performed better compared to models which had concatenated word vectors as input. The intuition behind average word vectors for review representation is that, if most words in the review are similar or co-occur, then their vectors are closer to each other in space. Taking average of these vectors, results in single vector that would be mostly at the center of all these vectors. This single vector will be good representation of the entire review.

Overall, Perceptron and FNN (with concatenated word vectors) performed poorly. SVM and FNN (with average word vectors) have performed better.

Recurrent Neural Network

(a) Train a simple RNN for sentiment analysis. You can consider an RNN cell with the hidden state size of 20. To feed your data into our RNN, limit the maximum review length to 20 by truncating longer reviews and padding shorter reviews with a null value (0). Report accuracy values on the testing split for your RNN model. What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

The following steps are performed on both train dataset and test dataset

1. Tokenized reviews are converted to word2Vectors first. Each word is a vector of size 300
2. The vectors for each review reduced to 20 words. After this step, each review is represented by a vector of 20 vectors each of size 300 (300*20)
3. If there are any samples with review vectors size lesser than 300*20, these are padded with zeros

In [45]:
```
rnn_X_train = train_data.copy()
rnn_X_test = test_data.copy()

rnn_X_train["rnn_vectors"] = rnn_X_train['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
rnn_X_train["reduced_rnn_vectors"] = rnn_X_train['rnn_vectors'].apply(
    lambda x: x[:20])

rnn_X_train["star_rating"] = rnn_X_train["star_rating"].astype(
    'int').to_numpy()

for index, row in rnn_X_train.iterrows():
    data_list = row['reduced_rnn_vectors']
    if data_list is None:
        rnn_X_train.at[index, "reduced_rnn_vectors"] = [np.zeros(
            300)] * 20
    elif len(data_list) < 20:
        data_list.extend([np.zeros(300)] * (20-len(data_list)))
        rnn_X_train.at[index, "reduced_rnn_vectors"] = data_list


rnn_X_test["rnn_vectors"] = rnn_X_test['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
rnn_X_test["reduced_rnn_vectors"] = rnn_X_test['rnn_vectors'].apply(
    lambda x: x[:20])

rnn_X_test["star_rating"] = rnn_X_test["star_rating"].astype(
    'int').to_numpy()

for index, row in rnn_X_test.iterrows():
    data_list = row['reduced_rnn_vectors']
    if data_list is None:
        rnn_X_test.at[index, "reduced_rnn_vectors"] = [np.zeros(
            300)] * 20
    elif len(data_list) < 20:
        data_list.extend([np.zeros(300)] * (20-len(data_list)))
        rnn_X_test.at[index, "reduced_rnn_vectors"] = data_list
```

RNNTensorDataset and Dataloader

1. It has helper function **getitem** that fetches item sample from dataframe based on index value.
2. Both train and test dataset are casted to RNNTensorDataset class
3. This is done so as to be able to define DataLoader
4. A dataloader takes in RNNTensorDataset class type and generates batches (batch size is set to 100) of data that are selected randomly (SubsetRandomSampler) and returns it.

In [46]:
```
class RNNTensorDataset(Dataset):
    def __init__(self, dataframe):
        self.data = dataframe

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        features = self.data.loc[index, 'reduced_rnn_vectors']
        label = self.data.loc[index, 'star_rating']
        return features, label, label-1

    def __getindexlist__(self):
        return list(self.data.index.values)

rnn_X_train_tensor = RNNTensorDataset(rnn_X_train)
rnn_X_test_tensor = RNNTensorDataset(rnn_X_test)

num_of_workers = 0
batch_size = 100
valid_size = 0.2

indices = list(range(len(rnn_X_train_tensor)))
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(
    rnn_X_train_tensor,
    batch_size=batch_size,
    sampler=SubsetRandomSampler(indices)
)
```

RNN Module

1. We have defined a Recurrent Neural Network model with hidden size of 20
2. Size of input layer is input size + hidden size
3. As the output is classification of reviews into 5 ratings, the output layer size is 5

Negative Log Likelihood loss is used.

For Optimizer, Stochastic Gradient Descent and Adam were tried with different learning rates (0.01, 0.001, 0.005, 0.0001, 0.0005). The best accuracy was achieved with Adam optimizer with a learning rate of 0.0001

In [47]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self, batch_size):
        return torch.zeros(batch_size, self.hidden_size)


hidden_size = 20
output_size = 5
input_size = 300
rnn = RNN(input_size, hidden_size, output_size)

all_categories = [1, 2, 3, 4, 5]
def categoryFromOutput(output):
    top_n, top_i = torch.max(output,dim=1)
    return top_i


criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=0.0001)
```

Training on the dataset is done for 120 epochs

Each input is fed in sequence to the model. With every input, output and hidden state is generated. This hidden state is then again fed with next input word in the sequence. The output of final word in the review is used to find the classification done by the model

In [48]:
```
def train(class_data, class_data_index, review):
    hidden = rnn.initHidden(batch_size)
    optimizer.zero_grad()
    for i in range(len(review)):
        review_tensor = torch.from_numpy(
            np.asarray(review[i])).float()
        output, hidden = rnn(review_tensor, hidden)

    loss = criterion(output, class_data_index)
    loss.backward()
    optimizer.step()

    return output, loss.item()

n_iters = 120
print_every = 10

for iter in range(1, n_iters + 1):
    torch.manual_seed(42)
    for data, target, target_index in train_loader:
        output, loss = train(target, target_index, data)
    if (iter%print_every == 0):
        print("Completed epoch " + str(iter))
```

```
Completed epoch 10
Completed epoch 20
Completed epoch 30
Completed epoch 40
Completed epoch 50
Completed epoch 60
Completed epoch 70
Completed epoch 80
Completed epoch 90
Completed epoch 100
Completed epoch 110
Completed epoch 120
```

Test Accuracy

1. Test Loader is created with test dataset
2. A function call to test_accuracy is made with loader and batch siexe
3. The predictions and actual labels obtained are compared with each other using custom written function categoryFromOutput and torch.eq
4. Accuracy for the model is reported

In [49]:
```
def evaluateRNN(review, size):
    hidden = rnn.initHidden(size)

    for i in range(len(review)):
        review_tensor = torch.from_numpy(
            np.asarray(review[i])).float()
        output, hidden = rnn(review_tensor, hidden)
    return output

def test_accuracy(loader, size):
    compare_list = []
    for data, target, target_index in loader:
        output = evaluateRNN(data, size)
        prediction_index = categoryFromOutput(output)
        compare = torch.eq(prediction_index, target_index)
        compare_list = compare.tolist()
    return compare_list

test_loader = torch.utils.data.DataLoader(
    rnn_X_test_tensor,
    batch_size=rnn_X_test_tensor.__len__())
compare_list = test_accuracy(
    test_loader, rnn_X_test_tensor.__len__())
rnn_accuracy = sum(compare_list)/len(compare_list)
print("accuracy: " + str(rnn_accuracy))
```

```
accuracy: 0.45405
```

What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

All the models were run on same training and testing dataset. Also, same preprocessing techniques were applied on both dataset. This was to ensure that all model performances can be compared directly with each other.

Following is the summary of the RNN and FNN model performance with TF-IDF and Word2Vec as input features

Model	Accuracy
RNN with 20 words	0.4540
FNN with average word vectors	0.4976
FNN with concatenated word vectors	0.4326

RNN has performed better compared to FNN with concatenated words. This could be because

1. FNN made use of 10 words while RNN is considering 20 words. This leads to larger features and more data to work with for the RNN model and hence find better classification boundaries
2. RNN works in a sequence and is capable of maintaining historical data in form of hidden states for predictions and is good at handling shorter sequences

Though RNN has performed better compared to FNN with concatenated words, it has not performed so well when compared to FNN with average word vectors. The reason for this could include

1. FNN worked with all words in the review and took average of it whereas RNN worked with just 20 words from each review. A lot of important information can be lost leading to lower accuracy for RNN
2. RNN has issue of vanishing or exploding gradients that might cause significant issues in the learning of the model during back propagation

The conclusion is that FNN with average word vectors performs best till now among all the models that have been compared.

By varying hyperparameters of the models such as number of hidden layers, hidden layer size, activation functions used etc., it is probably possible to gain better accuracy. Accuracy values also depend on the data cleaning steps that have been included.

(b) Repeat part (a) by considering a gated recurrent unit cell. What do you conclude by comparing accuracy values you obtain with those obtained using simple RNN.

The following steps are performed on both train dataset and test dataset

1. Tokenized reviews are converted to word2Vectors first. Each word is a vector of size 300
2. The vectors for each review reduced to 20 words. After this step, each review is represented by a vector of 20 vectors each of size 300 (300*20)
3. If there are any samples with review vectors size lesser than 300*20, these are padded with zeros

In [50]:
```
rnn_X_train = train_data.copy()
rnn_X_test = test_data.copy()

rnn_X_train["gru_vectors"] = rnn_X_train['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
rnn_X_train["reduced_gru_vectors"] = rnn_X_train['gru_vectors'].apply(
    lambda x: x[:20])
rnn_X_train["star_rating"] = rnn_X_train["star_rating"].astype(
    'int').to_numpy()

for index, row in rnn_X_train.iterrows():
    data_list = row['reduced_gru_vectors']
    if data_list is None:
        rnn_X_train.at[index, "reduced_gru_vectors"] = [np.zeros(
            300)] * 20
    elif len(data_list) < 20:
        data_list.extend([np.zeros(300)] * (20-len(data_list)))
        rnn_X_train.at[index, "reduced_gru_vectors"] = data_list


rnn_X_test["gru_vectors"] = rnn_X_test['tokenized_reviews'].apply(
    lambda wordlist: [wv[i] for i in wordlist if i in words_google])
rnn_X_test["reduced_gru_vectors"] = rnn_X_test['gru_vectors'].apply(
    lambda x: x[:20])
rnn_X_test["star_rating"] = rnn_X_test["star_rating"].astype(
    'int').to_numpy()

for index, row in rnn_X_test.iterrows():
    data_list = row['reduced_gru_vectors']
    if data_list is None:
        rnn_X_test.at[index, "reduced_gru_vectors"] = [np.zeros(
            300)] * 20
    elif len(data_list) < 20:
        data_list.extend([np.zeros(300)] * (20-len(data_list)))
        rnn_X_test.at[index, "reduced_gru_vectors"] = data_list
```

GRUTensorDataset and Dataloader

1. It has helper function **getitem** that fetches item sample from dataframe based on index value.
2. Both train and test dataset are casted to GRUTensorDataset class
3. This is done so as to be able to define DataLoader
4. A dataloader takes in GRUTensorDataset class type and generates batches (batch size is set to 100) of data that are selected randomly (SubsetRandomSampler) and returns it.

In [51]:
```
class GRUTensorDataset(Dataset):
    def __init__(self, dataframe):
        self.data = dataframe

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        features = self.data.loc[index, 'reduced_gru_vectors']
        label = self.data.loc[index, 'star_rating']
        return torch.from_numpy(
            np.asarray(features)).float(), label, label - 1

    def __getindexlist__(self):
        return list(self.data.index.values)

rnn_X_train_tensor = GRUTensorDataset(rnn_X_train)
rnn_X_test_tensor = GRUTensorDataset(rnn_X_test)

num_of_workers = 0
batch_size = 100
valid_size = 0.2

indices = list(range(len(rnn_X_train_tensor)))
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(
    rnn_X_train_tensor,
    batch_size=batch_size,
    sampler=SubsetRandomSampler(indices)
)
```

GRUNet Module

1. We have defined a Gated Recurrent Neural Network model
2. As the output is classification of reviews into 5 ratings, the output layer size is 5
3. Softmax is used for last layer

Negative Log Likelihood loss is used.

For Optimizer, Stochastic Gradient Descent and Adam were tried with different learning rates (0.01, 0.001, 0.005, 0.0001, 0.0005). The best accuracy was achieved with Adam optimizer with a learning rate of 0.0001

In [52]:
```
class GRUNet(nn.Module):
    def __init__(self, input_dim, hidden_dim,
                 output_dim, n_layers):
        super(GRUNet, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        self.gru = nn.GRU(input_dim, hidden_dim,
                          n_layers, batch_first = True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = self.softmax(self.fc(out[:,-1]))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = weight.new(self.n_layers, batch_size,
                            self.hidden_dim).zero_()
        return hidden

hidden_size = 20
output_size = 5
input_size = 300
n_layers = 1
gru = GRUNet(input_size, hidden_size, output_size, n_layers)
print(gru)
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(gru.parameters(), lr=0.0005)
```

```
GRUNet(
  (gru): GRU(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=5, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

Training on the dataset is done for 50 epochs

In [53]:
```
n_epochs = 50

for epoch in range(n_epochs):
    torch.manual_seed(42)
    train_loss = 0.0
    gru.train()
    for data, target, target_index in train_loader:
        h = gru.init_hidden(batch_size)
        optimizer.zero_grad()
        output, h = gru(data, h.data)
        loss = criterion(output, target_index)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)

    gru.eval()
```

Test Accuracy

1. Test Loader is created with test dataset
2. A function call to test_accuracy is made with loader and batch siexe
3. The predictions and actual labels obtained are compared with each other using custom written function categoryFromOutput and torch.eq
4. Accuracy for the model is reported

In [54]:
```
def evaluateGRU(review, size):
    hidden = gru.init_hidden(size)
    output, hidden = gru(review, hidden)
    return output

def test_accuracy(loader, size):
    compare_list = []
    for data, target, target_index in loader:
        output = evaluateGRU(data, size)
        prediction_index = categoryFromOutput(output)
        compare = torch.eq(prediction_index, target_index)
        compare_list = compare.tolist()
    return compare_list

test_loader = torch.utils.data.DataLoader(
    rnn_X_test_tensor,
    batch_size=rnn_X_test_tensor.__len__())
compare_list = test_accuracy(
    test_loader,
    rnn_X_test_tensor.__len__())
gru_accuracy = sum(compare_list)/len(compare_list)
print("accuracy: " + str(gru_accuracy))
```

```
accuracy: 0.51105
```

What do you conclude by comparing accuracy values you obtain with those obtained using simple RNN.

All the models were run on same training and testing dataset. Also, same preprocessing techniques were applied on both dataset. This was to ensure that all model performances can be compared directly with each other.

Following is the summary of the model performances

Model	Accuracy
RNN with 20 words	0.4540
Gated RNN with 20 words.	0.51105

We can conclude that Gated Recurrent Unit has performed better than RNN and other models (FNN, SVM, Perceptron) that were tried.

Some of the possible reasons for this -

1. RNN has issue of vanishing or exploding gradients. This can hinder the learning during back propagation in training phase
2. Also, RNN has issue of short term memory problem for long sequences.

Gated Recurrent Unit overcomes these issues and has capability of handling long term memory dependencies with control mechanisms for information flow.

Note:

All of the Neural Networks models that were tried were with limited hidden layers and nodes in hidden layers with some restrictions on the input size for each review. There are possibilites of obtaining better accuracy with deeper networks and thorough hyper paramater tuning.

Report All Model Accuracies

Simple Models with TF-IDF

In [55]:
```
print("Accuracy of SVM Model with TF-IDF inputs : " + "50%")
print("Accuracy of Perceptron Model with TF-IDF inputs : " + "43%")
```

```
Accuracy of SVM Model with TF-IDF inputs : 50%
Accuracy of Perceptron Model with TF-IDF inputs : 43%
```

Simple Models with Word2Vec

In [79]:
```
print("Accuracy of SVM Model with word2Vec inputs : " + str(
    svm_wordvec_accuracy*100) + "%")
print("Accuracy of Perceptron Model with word2Vec inputs : " + str(
    perceptron_wordvec_accuracy*100) + "%")
```

```
Accuracy of SVM Model with word2Vec inputs : 48.27%
Accuracy of Perceptron Model with word2Vec inputs : 43.980000000000004%
```

FeedForward Neural Networks

In [57]:
```
print("Accuracy of FNN Model with averaged word2Vec inputs : " + str(
    fnna_wordvec_average_accuracy*100) + "%")
print("Accuracy of FNN Model with concatenated word2Vec inputs : " + str(
    fnnb_wordvec_cat_accuracy*100) + "%")
```

```
Accuracy of FNN Model with averaged word2Vec inputs : 49.76732549412059%
Accuracy of FNN Model with concatenated word2Vec inputs : 43.26%
```

Recurrent Neural Networks

In [58]:
```
print("Accuracy of RNN Model with word2Vec inputs : " + str(
    rnn_accuracy*100) + "%")
print("Accuracy of GRU Model with word2Vec inputs : " + str(
    gru_accuracy*100) + "%")
```

```
Accuracy of RNN Model with word2Vec inputs : 45.405%
Accuracy of GRU Model with word2Vec inputs : 51.105000000000004%
```

REFERENCES

1. https://blog.floydhub.com/gru-with-pytorch/ (https://blog.floydhub.com/gru-with-pytorch/)
2. https://github.com/georgeyiasemis/Recurrent-Neural-Networks-from-scratch-using-PyTorch (https://github.com/georgeyiasemis/Recurrent-Neural-Networks-from-scratch-using-PyTorch)
3. https://towardsdatascience.com/implementation-of-rnn-lstm-and-gru-a4250bf6c090 (https://towardsdatascience.com/implementation-of-rnn-lstm-and-gru-a4250bf6c090)
4. https://github.com/georgeyiasemis/Recurrent-Neural-Networks-from-scratch-using-PyTorch/blob/main/rnnmodels.py (https://github.com/georgeyiasemis/Recurrent-Neural-Networks-from-scratch-using-PyTorch/blob/main/rnnmodels.py)
5. https://github.com/kaustubhhiware/LSTM-GRU-from-scratch (https://github.com/kaustubhhiware/LSTM-GRU-from-scratch)
6. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)
7. https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook (https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook)
8. https://www.youtube.com/watch?v=WEV61GmmPrk (https://www.youtube.com/watch?v=WEV61GmmPrk)