I summarized everything at the beginning of the document since the code for this HW is long and might be hard to read.
After the written report, there are pdf versions of the ipynb file.

2. Word Embedding

(a) In this section, I used three examples

(i) Cat + Kid: this example is to compare very basic words between the two models. From the similarity result, we can see that Google's model does a better job on this matter.
Top words from Google's model: puppy, kitten, pup, beagle, maine_coon_cat
Top words from our model contains nothing related to kiddy cat or other animal like kitten

(ii) Ring - Accessory: this example is to compare if each model can produce other ring related words that are not accessory. Google's model is still a better model here since it can produce words like 'squared_circle''. However, our model keeps producing the word ring, rings since the domain of our dataset is not wide enough.

(iii) Dog + Wild: this is also another example of common knowledge. We expect words like wolf from this combination.
Top words from Google's model: pitbulls_rottweilers, pitbull, wolfdogs, etc.
Top words from our model has no word related to wolf or other kinds of dogs.

Results (comparison of the two models)

```
ring - accessory Example
Google W2V: [('ring', 0.6125255227088928), ('rings', 0.3762699067592621),
('squared_circle', 0.3308883011341095), ('TitanTron', 0.326825827360153
2), ('Ring', 0.32146134972572327), ('WBA_heavyweight_titlist', 0.32121133
80432129), ('bell', 0.3198775053024292), ('Cattle_Mutilation', 0.31495627
760887146), ('Evander_Holyfield_Riddick_Bowe', 0.3141948878765106), ('bel
l_clanged', 0.30424806475639343)]
Dataset W2V: [('ring', 0.20062153041362762), ('ring,', 0.1682205200195312
5), ('ring.', 0.16415229439735413), ('rings', 0.15541096031665802), ('edg
es', 0.15373587608337402), ('prongs', 0.15153872966766357), ("couldn't",
0.14867763221263885), ('reading', 0.14384742081165314), ('thumb', 0.14196
433126926422), ('diamonds', 0.1401032954454422)]
```

```
dog + wild Example
Google W2V: [('dog', 0.8025956153869629), ('wild', 0.744804859161377),
('dogs', 0.7257054448127747), ('cat', 0.6927908658981323), ('pit_bulls_ro
ttweilers', 0.6573203802108765), ('puppy', 0.6570946574211121), ('pit_bul
l', 0.653598427772522), ('wolfdogs', 0.6503258347511292), ('cats', 0.6465
378999710083), ('Rottweiller', 0.6457170248031616)]
Dataset W2V: [('wild', 0.8480218052864075), ('dog', 0.5792834758758545),
('coworker', 0.3584851920604706), ('remained', 0.3199407160282135), ('lov
elinks®', 0.31843301653862), ('aagaard', 0.3175673484802246), ('/>affenpi
nscher', 0.3067035377025604), ('cat:', 0.2988812029361725), ('inspiration
al', 0.2903487980365753), ('hospital', 0.28840193152427673)]
```

```
----------------------------------------
cat + kid Example
Google W2V: [('cat', 0.8128764629364014), ('kid', 0.7883383631706238),
('puppy', 0.6957499384880066), ('kitten', 0.6867817044258118), ('pup', 0.
6848490834236145), ('dog', 0.680992841720581), ('beagle', 0.6532338857650
757), ('Maine_coon_cat', 0.6322909593582153), ('cats', 0.630014896392822
3), ('pooch', 0.6185204386711121)]
Dataset W2V: [('cat', 0.7570907473564148), ('kid', 0.7343471050262451),
('lover', 0.3123132884502411), ('cat.', 0.3069145977497101), (',i', 0.275
1149833202362), ('section.', 0.2650524973869324), ('teacher.', 0.25451567
7690506), ('car.', 0.2537133991718292), ('tastes.', 0.2512478530406952),
('evenly.', 0.2472926527261734)]
----------------------------------------
```

(b) Word2Vec trained using google dataset seems to embed semantic similarity better than our model since they cover very large words from the corpus. Our Word2Vec model however, seems to encode good semantic similarity in jewelry type of words since our dataset is a jewelry dataset

```
Similarity between pairs
Google pairs
'cat'   'cute'  0.31
'neckless'      'accessory'     0.10
'ring'  'accessory'     0.12
'cat'   'kid'   0.28
'girl'  'woman' 0.75
Our model pairs
'cat'   'cute'  0.16
'neckless'      'accessory'     0.07
'ring'  'accessory'     0.05
'cat'   'kid'   0.11
'girl'  'woman' 0.32
----------------------------------------
```

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

3. Simple Model.

Simple Model: Perceptron IF-IDF

	precision	recall	f1-score	support
1	0.57	0.73	0.64	4000
2	0.44	0.52	0.48	4000
3	0.64	0.26	0.37	4000
4	0.49	0.55	0.52	4000
5	0.68	0.70	0.69	4000
accuracy			0.55	20000
macro avg	0.57	0.55	0.54	20000
weighted avg	0.57	0.55	0.54	20000

Simple Model: Perceptron Word2Vec

	precision	recall	f1-score	support
1	0.92	0.24	0.38	4000
2	0.66	0.16	0.26	4000
3	0.29	0.83	0.43	4000
4	0.43	0.37	0.40	3999
5	0.73	0.56	0.64	3999
accuracy			0.43	19998
macro avg	0.61	0.43	0.42	19998
weighted avg	0.61	0.43	0.42	19998

Simple Model: SVM (IF-IDF)

	precision	recall	f1-score	support
1	0.64	0.71	0.68	4000
2	0.52	0.49	0.50	4000
3	0.51	0.44	0.47	4000
4	0.57	0.54	0.55	4000
5	0.69	0.79	0.74	4000
accuracy			0.59	20000
macro avg	0.59	0.59	0.59	20000
weighted avg	0.59	0.59	0.59	20000

Simple Model: SVM (Word2Vec)

	precision	recall	f1-score	support
1	0.61	0.75	0.68	4000
2	0.49	0.43	0.46	4000
3	0.47	0.44	0.46	4000
4	0.52	0.39	0.45	3999
5	0.65	0.79	0.71	3999
accuracy			0.56	19998
macro avg	0.55	0.56	0.55	19998
weighted avg	0.55	0.56	0.55	19998

For Perceptron, Word2Vec vectors seem to encode better semantic similarity. However for SVM model, IF-IDF performs better than Word2Vec model

4. Feedforward Neural Network

(a) Report accuracy of average Word2Vec

```
Accuracy FNN (Average):  0.6062106210621062
```

(b) Report accuracy of concatenated 10 words

```
Accuracy (Concat. 10 words):  0.5134257123489022
```

Conclusion: when using the average of the full Word2Vec vectors, we get better results than only selecting 10 vectors from the review. I think even when averaging, Word2Vec results better when having more data behind the scenes and cutting them into 10 words reduces the accuracy.

Feedforward Neural Network gives about the same accuracy as the simple model. It outperformed when using 10 words concatenation which makes sense since our input got cut significantly.

5. Recurrent Neural Network

(a) Report accuracy of RNN

```
RNN Accuracy: 0.5123768565284793
```

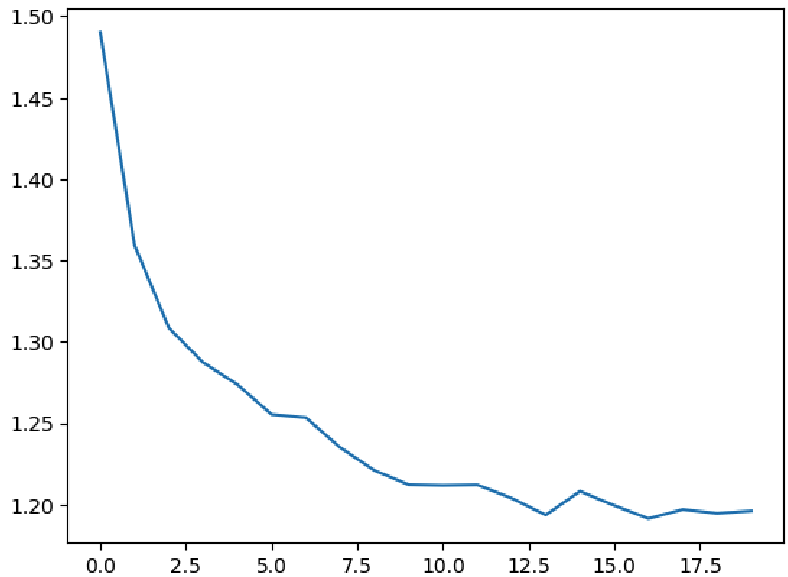
The accuracy of RNN is about the same as FNN, the difference between them is the run-time. I found that FNN trains much faster than RNN

(b) Report accuracy of GRU

```
GRU Accuracy: 0.4123429958432859
```

The accuracy of GRU reduces significantly when shifting from RNN to GRU, this model also trains longer since we have a long sequence of reviews.

Example of RNN Loss



In [106]:
```
# import
import pandas as pd
import numpy as np
import re
from bs4 import BeautifulSoup
from sklearn.utils import shuffle
import contractions
import warnings
warnings.filterwarnings('ignore')
```

In [107]:
```
#! pip install bs4 # in case you don't have it installed
#!pip install gensim

# Dataset: https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_u
```

Read Data

In [108]:
```
data = pd.read_table('https://s3.amazonaws.com/amazon-reviews-pds/tsv/amaz
```

In [109]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

Keep Reviews and Ratings

We select 20000 reviews randomly from each rating class.

In [111]:
```
# skip to do after step 4
review_rating = data[["review_headline", "review_body","star_rating"]]
#star_rating = data[["star_rating"]]
```

In [112]:
```
five_rating = review_rating[review_rating['star_rating'] == '5']
four_rating = review_rating[review_rating['star_rating'] == '4']
three_rating = review_rating[review_rating['star_rating'] == '3']
two_rating = review_rating[review_rating['star_rating'] == '2']
one_rating = review_rating[review_rating['star_rating'] == '1']

#Shuffle the data (seed = 42)
five_rating = shuffle(five_rating, random_state = 42)
four_rating = shuffle(four_rating, random_state = 42)
three_rating = shuffle(three_rating, random_state = 42)
two_rating = shuffle(two_rating, random_state = 42)
one_rating = shuffle(one_rating, random_state = 42)
```

In [113]:
```
#Select only 20000 rows of reviews from each class

five_rating,four_rating,three_rating,two_rating,one_rating = five_rating.he
```

In [114]:
```
#split train/test dataset as 80%/20% for each class

five_rating_train,four_rating_train,three_rating_train,two_rating_train,one
five_rating_test,four_rating_test,three_rating_test,two_rating_test,one_rat

# concat to get full train set and test set

train_set = pd.concat([five_rating_train,four_rating_train,three_rating_tra
test_set = pd.concat([five_rating_test,four_rating_test,three_rating_test,t
# Data Cleaning
# combine review_headline and review_body into review
# convert to lowercase before combining
train_set['review'] = train_set['review_headline'].str.lower() + ' ' + trai
test_set['review'] = test_set['review_headline'].str.lower() + ' ' + test_s

# Create a list of strings, one for each title
train_list = [train for train in train_set['review'].dropna()]
test_list = [test for test in test_set['review'].dropna()]

# Collapse the list of strings into a single long string for processing
string = ' '.join(train_list)
string = ' '.join(test_list)

from nltk.tokenize import word_tokenize

# Tokenize the string into words
tokens = word_tokenize(string)

# Print first 10 words
tokens[:10]
```

Out[114]:
```
['son', 'loves', 'them', 'my', 'son', 'loves', 'these', '.', 'they', 'loo
k']
```

In [115]:
```
print("Shape of new dataframes - {} , {}".format(train_set.shape, test_set.
```

```
Shape of new dataframes - (80000, 4) , (20000, 4)
```

In [116]:
```
train_set.head()
```

Out[116]:

	review_headline	review_body	star_rating	review
0	Five Stars	Love it!	5	five stars love it!
1	Love it....	This idem is exactly how it's describe, "E...	5	love it.... this idem is exactly how it's desc...
2	Beautiful Mala, great purchase	Highly recommended - arrived on time, well mad...	5	beautiful mala, great purchase highly recommen...
3	Gorgeous Earrings	Got these for my wife as a birthday present. S...	5	gorgeous earrings got these for my wife as a b...
4	GREAT Necklace!	I have worn this necklace every day since I pu...	5	great necklace! i have worn this necklace ever...

2. Word Embedding

In [117]:
```
# 2(a) print similarity between pairs

# pairs = [
#     ('car', 'minivan'),   # a minivan is a kind of car
#     ('car', 'bicycle'),   # still a wheeled vehicle
#     ('car', 'airplane'),  # ok, no wheels, but still a vehicle
#     ('car', 'cereal'),    # ... and so on
#     ('car', 'communism'),
# ]
pairs = [
    ('cat', 'cute'),
    ('cat', 'kitten'),
    ('cute', 'kitten'),
    ('queen', 'woman'),
    ('queen', 'girl'),
]
for w1, w2 in pairs:
    print('%r\t%r\t%.2f' % (w1, w2, wv.similarity(w1, w2)))
vec_cat = wv['cat']
vec_cute = wv['cute']
vec_kitten = wv['kitten']
```

```
'cat'	'cute'	0.31
'cat'	'kitten'	0.75
'cute'	'kitten'	0.39
'queen'	'woman'	0.32
'queen'	'girl'	0.35
```

In [118]:
```
# 2(b)
sentences = train_set['review']
print(len(sentences))
print(sentences[1])
```

```
80000
love it.... this idem is exactly how it's describe, &#34;eye-catching&#3
4; i love it!  if you're looking for something to make a statement, this
is the ring for you. very classily and elegent, a nice piece.
```

In [153]:
```
import gensim
from gensim.models.callbacks import CallbackAny2Vec
from gensim.models import Word2Vec


# initialize callback class
class callback(CallbackAny2Vec):

    def __init__(self):
        self.epoch = 0

    def on_epoch_end(self, model):
        loss = model.get_latest_training_loss() # this is a cumulative loss

        if self.epoch == 0:
            print('Cumulative Loss after epoch {}: {}'.format(self.epoch, l
        if self.epoch % 100 == 0:
            print('Cumulative Loss after epoch {}: {}'.format(self.epoch, l
        self.epoch += 1
```

In [156]:
```
import time

# prepare input of word2vec class (list of strings)
sentences_list = []
for i in range (0,len(sentences)):
    sen = str(sentences[i])
    temp = sen.split()
    sentences_list.append(temp)

# initialize word2vec class

w2v_model = Word2Vec(vector_size = 300, window = 11, min_count = 10, worker

#build vocab

w2v_model.build_vocab(sentences_list)

#train the w2v model
start = time.time()
w2v_model.train(sentences_list, total_examples = w2v_model.corpus_count,epo
               , callbacks = [callback()])
end = time.time()

print("Elaspsed time in seconds: " + str(end - start))
# save the word2vec model
w2v_model.save('word2vec.model')
```

```
Cumulative Loss after epoch 0: 576335.5625
Cumulative Loss after epoch 0: 576335.5625
Cumulative Loss after epoch 100: 32014854.0
Cumulative Loss after epoch 200: 53230776.0
Cumulative Loss after epoch 300: 67636440.0
Cumulative Loss after epoch 400: 69157144.0
Cumulative Loss after epoch 500: 70507424.0
Cumulative Loss after epoch 600: 71697520.0
Cumulative Loss after epoch 700: 72679864.0
Cumulative Loss after epoch 800: 73486240.0
Cumulative Loss after epoch 900: 74082568.0
Cumulative Loss after epoch 1000: 74469432.0
Elaspsed time in seconds: 1284.7435319423676
```

In [157]:
```
tokens = []
labels = []

for word in w2v_model.wv.key_to_index:
    tokens.append(w2v_model.wv[word])
    labels.append(word)
```

In [172]:
```
# w2v_model.wv['cat']
# w2vmodel.wv.similarity('france', 'spain')

# similarity between pairs
pairs = [
    ('cat', 'cute'),
    ('neckless', 'accessory'),
    ('ring', 'accessory'),
    ('cat', 'kid'),
    ('girl', 'woman')
]

print('Similarity between pairs')
print('Google pairs')
for w1, w2 in pairs:
    print('%r\t%r\t%.2f' % (w1, w2, wv.similarity(w1, w2)))
print('Our model pairs')
for w1, w2 in pairs:
    print('%r\t%r\t%.2f' % (w1, w2, w2v_model.wv.similarity(w1, w2)))
print('----------------------------------------')
# similarity between each example
diff_vector_google1 = wv['cat'] + wv['kid']
diff_vector_dataset1 = w2v_model.wv['cat'] + w2v_model.wv['kid']
diff_vector_google2 = wv['ring'] - wv['accessory']
diff_vector_dataset2 = w2v_model.wv['ring'] - w2v_model.wv['accessory']
diff_vector_google3 = wv['dog'] + wv['wild']
diff_vector_dataset3 = w2v_model.wv['dog'] + w2v_model.wv['wild']
print('cat + kid Example')
print('Google W2V:', wv.most_similar(positive=[diff_vector_google1]))
print('Dataset W2V:', w2v_model.wv.most_similar(positive = [diff_vector_dat
print('----------------------------------------')
print('ring - accessory Example')
print('Google W2V:', wv.most_similar(positive=[diff_vector_google2]))
print('Dataset W2V:',w2v_model.wv.most_similar(positive = [diff_vector_data
print('----------------------------------------')
print('dog + wild Example')
print('Google W2V:', wv.most_similar(positive=[diff_vector_google3]))
print('Dataset W2V:',w2v_model.wv.most_similar(positive = [diff_vector_data
```

```
Similarity between pairs
Google pairs
'cat'   'cute'  0.31
'neckless'      'accessory'     0.10
'ring'  'accessory'     0.12
'cat'   'kid'   0.28
'girl'  'woman' 0.75
Our model pairs
'cat'   'cute'  0.16
'neckless'      'accessory'     0.07
'ring'  'accessory'     0.05
'cat'   'kid'   0.11
'girl'  'woman' 0.32
----------------------------------------
cat + kid Example
Google W2V: [('cat', 0.8128764629364014), ('kid', 0.7883383631706238),
('puppy', 0.6957499384880066), ('kitten', 0.6867817044258118), ('pup', 0.
6848490834236145), ('dog', 0.680992841720581), ('beagle', 0.6532338857650
```

```
757), ('Maine_coon_cat', 0.6322909593582153), ('cats', 0.630014896392822
3), ('pooch', 0.6185204386711121)]
Dataset W2V: [('cat', 0.7570907473564148), ('kid', 0.7343471050262451),
('lover', 0.3123132884502411), ('cat.', 0.3069145977497101), (',i', 0.275
1149833202362), ('section.', 0.2650524973869324), ('teacher.', 0.25451567
7690506), ('car.', 0.2537133991718292), ('tastes.', 0.2512478530406952),
('evenly.', 0.2472926527261734)]
----------------------------------------
ring - accessory Example
Google W2V: [('ring', 0.6125255227088928), ('rings', 0.3762699067592621),
('squared_circle', 0.3308883011341095), ('TitanTron', 0.326825827360153
2), ('Ring', 0.32146134972572327), ('WBA_heavyweight_titlist', 0.32121133
80432129), ('bell', 0.3198775053024292), ('Cattle_Mutilation', 0.31495627
760887146), ('Evander_Holyfield_Riddick_Bowe', 0.3141948878765106), ('bel
l_clanged', 0.30424806475639343)]
Dataset W2V: [('ring', 0.20062153041362762), ('ring,', 0.1682205200195312
5), ('ring.', 0.16415229439735413), ('rings', 0.15541096031665802), ('edg
es', 0.15373587608337402), ('prongs', 0.15153872966766357), ("couldn't",
0.14867763221263885), ('reading', 0.14384742081165314), ('thumb', 0.14196
433126926422), ('diamonds', 0.1401032954454422)]
----------------------------------------
dog + wild Example
Google W2V: [('dog', 0.8025956153869629), ('wild', 0.744804859161377),
('dogs', 0.7257054448127747), ('cat', 0.6927908658981323), ('pit_bulls_ro
ttweilers', 0.6573203802108765), ('puppy', 0.6570946574211121), ('pit_bul
l', 0.653598427772522), ('wolfdogs', 0.6503258347511292), ('cats', 0.6465
378999710083), ('Rottweiller', 0.6457170248031616)]
Dataset W2V: [('wild', 0.8480218052864075), ('dog', 0.5792834758758545),
('coworker', 0.3584851920604706), ('remained', 0.3199407160282135), ('lov
elinks®', 0.31843301653862), ('aagaard', 0.3175673484802246), ('/>affenpi
nscher', 0.3067035377025604), ('cat:', 0.2988812029361725), ('inspiration
al', 0.2903487980365753), ('hospital', 0.28840193152427673)]
```

In [ ]:

3. Simple Models

In [177]:
```
# Define functions that are needed for vectorization

def document_vector(word2vec_model, doc):
    # remove out-of-vocabulary words
    doc = [word for word in doc if word in wv.index_to_key]
    return np.mean(wv[doc], axis=0)

# Our earlier preprocessing was done when we were dealing only with word ve
# Here, we need each document to remain a document
def preprocess(text):
    text = text.lower()
    doc = word_tokenize(text)
    #doc = [word for word in doc if word not in stop_words]
    #doc = [word for word in doc if word.isalpha()]
    return doc

# Function that will help us drop documents that have no word vectors in wo
def has_vector_representation(word2vec_model, doc):
    """check if at least one word of the document is in the
    word2vec dictionary"""
    return not all(word not in word2vec_model.index_to_key for word in doc)

# Filter out documents
def filter_docs(corpus, texts, condition_on_doc):
    """
    Filter corpus and texts given the function condition_on_doc which takes
    """
    number_of_docs = len(corpus)

    if texts is not None:
        texts = [text for (text, doc) in zip(texts, corpus)
                 if condition_on_doc(doc)]

    corpus = [doc for doc in corpus if condition_on_doc(doc)]

    print("{} docs removed".format(number_of_docs - len(corpus)))

    return (corpus, texts)
```

In [178]:
```
# tokenize our document
corpus_train = [preprocess(train) for train in train_list]
corpus_test = [preprocess(test) for test in test_list]

# Remove docs that don't include any words in W2V's vocab
corpus_train, train_list = filter_docs(corpus_train, train_list, lambda doc
corpus_test, test_list = filter_docs(corpus_test, test_list, lambda doc: ha

# Filter out any empty docs
corpus_train, train_list = filter_docs(corpus_train, train_list, lambda doc
x = []
for doc in corpus_train: # append the vector for each document
    x.append(document_vector(wv, doc))

X = np.array(x) # list to array
```

```
0 docs removed
0 docs removed
0 docs removed

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call las
t)
Input In [178], in <cell line: 12>()
     11 x = []
     12 for doc in corpus_train: # append the vector for each document
---> 13     x.append(document_vector(wv, doc))
     15 X = np.array(x)

Input In [177], in document_vector(word2vec_model, doc)
      3 def document_vector(word2vec_model, doc):
      4     # remove out-of-vocabulary words
----> 5     doc = [word for word in doc if word in wv.index_to_key]
      6     return np.mean(wv[doc], axis=0)

Input In [177], in <listcomp>(.0)
      3 def document_vector(word2vec_model, doc):
      4     # remove out-of-vocabulary words
----> 5     doc = [word for word in doc if word in wv.index_to_key]
      6     return np.mean(wv[doc], axis=0)

KeyboardInterrupt: 
```

In [ ]:
```
# re-create x_test
x_test = []
corpus_test, test_list = filter_docs(corpus_test, test_list, lambda doc: (l
for doc in corpus_test: # append the vector for each document
    x_test.append(document_vector(wv, doc))
```

In [188]:
```
# code for load data from local csv file to avoid processing time
from numpy import genfromtxt
my_data_xtrain = pd.read_csv('x_train.csv', sep=',', header=None)
my_data_xtest = pd.read_csv('x_test.csv', sep=',', header=None)

X = my_data_xtrain.to_numpy()
x_test = my_data_xtest.to_numpy()
```

In [189]:
```
# re-create y_test from the new x_train
c = np.in1d(test_set['review'],test_list)
y = test_set['star_rating'].to_numpy()
x_removed_indices = [i for i, x in enumerate(c) if not x]
y_test = np.delete(y, x_removed_indices)
```

In [190]:
```
# re-create y_train from the new x_train
c = np.in1d(train_set['review'],train_list)
y = train_set['star_rating'].to_numpy()
x_removed_indices = [i for i, x in enumerate(c) if not x]
y_train = np.delete(y, x_removed_indices)
```

In [ ]:
```
# re-build y of the remaining docs
#y = []
# for i in range(0,len(train_list)):
#     for j in range(0,len(train_set)):
#         if train_list[i] == train_set['review'][j]:
#             y.append(train_set['star_rating'][j])
#             break
```

In [ ]:
```
import pandas as pd
#pd.DataFrame(X).to_csv("x_train.csv", header=None, index=None)
#pd.DataFrame(x_test).to_csv("x_test.csv", header=None, index=None)
# save as csv file so I dont have to re-process them every time.
```

In [ ]:
```
# from sklearn import datasets
# from sklearn.preprocessing import StandardScaler
# from sklearn.linear_model import Perceptron
# from sklearn.metrics import accuracy_score


# ppn = Perceptron(max_iter=1000, eta0=0.1, random_state=0)
# ppn.fit(X, y_train)
# y_pred = ppn.predict(x_test)
```

In [ ]:
```
# from sklearn import metrics
# from sklearn.metrics import precision_score
# from sklearn.metrics import recall_score
# from sklearn.metrics import f1_score


# recall = recall_score(y_test, y_pred, average='weighted')
# precision = precision_score(y_test, y_pred, average='weighted')
# f1score = f1_score(y_test, y_pred, average='weighted')
# # Model Accuracy: how often is the classifier correct?
# print(metrics.classification_report(y_test,y_pred))
```

In [ ]:
```
# from sklearn.svm import SVC
# from sklearn.svm import LinearSVC
# from sklearn.multiclass import OneVsOneClassifier

# model = LinearSVC(random_state=0, multi_class = 'ovr')
# # define ovo strategy
# #ovo = OneVsOneClassifier(model)
# # fit model
# #ovo.fit(x_train, y_train)
# model.fit(X, y_train)
# # make predictions
# ypredsvm = model.predict(x_test)
# recallsvm = recall_score(y_test, ypredsvm, average='weighted')
# precisionsvm = precision_score(ypredsvm, y_pred, average='weighted')
# f1scoresvm = f1_score(y_test, ypredsvm, average='weighted')
# print(metrics.classification_report(y_test,ypredsvm))
```

In [191]:

```
# start of Question 4
# convert numpy array to pandas (preprocessing for tensor)
import torch
from torch.utils.data import DataLoader, Dataset
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


#train dataset
pd_xtrain = pd.DataFrame(X)
pd_ytrain = pd.DataFrame(y_train, columns = ['label'])
pd_train_data = pd.concat([pd_ytrain,pd_xtrain],axis=1)

#test dataset
pd_xtest = pd.DataFrame(x_test)
pd_ytest = pd.DataFrame(y_test, columns = ['label'])
pd_test_data = pd.concat([pd_ytest,pd_xtest],axis=1)

# np array to Tensor

# Passing to DataLoader
X=np.vstack(X).astype(np.float)
y_train=np.vstack(y_train).astype(np.float)
x_test=np.vstack(x_test).astype(np.float)
y_test=np.vstack(y_test).astype(np.float)


tensor_train_features = torch.Tensor(X)
tensor_train_labels = torch.Tensor(y_train)
tensor_test_features= torch.Tensor(x_test)
tensor_test_labels = torch.Tensor(y_test)

tensor_train_data = torch.utils.data.TensorDataset(tensor_train_features, t
tensor_test_data = torch.utils.data.TensorDataset(tensor_test_features, ten
```

In [187]:

```
len(tensor_test_labels)
```

Out[187]: 19998

In [ ]:

```
len(X[0])
```

In [192]:
```
# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 32
# percentage of training set to use as validation
valid_size = 0.2

# convert data to torch.FloatTensor
#transform = transforms.ToTensor()

# obtain training indices that will be used for validation
num_train = len(pd_train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = torch.utils.data.DataLoader(tensor_train_data, batch_size=ba
    sampler=train_sampler, num_workers=num_workers,)
valid_loader = torch.utils.data.DataLoader(tensor_train_data, batch_size=ba
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(tensor_test_data, batch_size=batc
    num_workers=num_workers)
```

In [ ]:

In [193]:
```
#define our NN model and initialize the model

import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # number of hidden nodes in each layer (512)
        hidden_1 = 50
        hidden_2 = 10
        # linear layer (300 -> hidden_1)
        self.fc1 = nn.Linear(300, hidden_1)
        # linear layer (n_hidden -> hidden_2)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(hidden_2, 5)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # flatten image input
        #x = x.view(-1, 300)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x

# initialize the NN
model = Net()
```

In [194]:
```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

In [195]:
```
#number of epochs to train the model
n_epochs = 150

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    print("training. . .")
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the
        output = model(data)
        # calculate the loss
        target = target.squeeze(1)
        target = target.type(torch.LongTensor)
        target = target - 1
        #target = torch.argmax(target, dim=1)
        #print(target)
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    print("validating. . .")
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the
        output = model(data)
        # calculate the loss
        target = target.squeeze(1)
        target = target.type(torch.LongTensor)
        target = target - 1
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)
```

```
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.for
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```
63
training. . .
validating. . .
Epoch: 106      Training Loss: 0.749434         Validation Loss: 0.1848
30
training. . .
validating. . .
Epoch: 107      Training Loss: 0.747336         Validation Loss: 0.1843
53
training. . .
validating. . .
Epoch: 108      Training Loss: 0.750398         Validation Loss: 0.1877
19
training. . .
validating. . .
Epoch: 109      Training Loss: 0.746402         Validation Loss: 0.1854
03
training. . .
validating. . .
Epoch: 110      Training Loss: 0.747180         Validation Loss: 0.1859
```

In [196]:
```
model.load_state_dict(torch.load('model.pt'))
```

Out[196]: <All keys matched successfully>

In [197]:
```
test_loader = torch.utils.data.DataLoader(tensor_test_data, batch_size=1, n
# Test the model
def predict(model, dataloader):
    prediction_list = []
    for i, batch in enumerate(dataloader):
        #print(batch)
        outputs = model(batch[0])
        #print(outputs.data)
        _, predicted = torch.max(outputs.data, 1)
        #print(predicted)
        #print((torch.max(outputs.data, 1)))
        prediction_list.append(predicted.cpu())
    return prediction_list

predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [198]:
```
pred= pd.DataFrame()
cor = 0
for i in range (0,len(predictions)):
    if y_test[i] == predictions[i].item() + 1:
        cor = cor + 1
print('Accuracy FNN (Average): ', cor/len(predictions))
```

```
Accuracy FNN (Average):  0.6062106210621062
```

In [213]:
```
# 4(b)
def document_vector_first10(word2vec_model, doc):
    top10 = []
    list_of_zeros = list(0 for i in range (0,300))
    for word in doc:
        if len(top10) == 10:
            break
        if word in wv.index_to_key and len(top10) < 10:
            top10.append(wv[word])

    top10_data = []
    #print(len(top10))
    #print(top10[0])
    for i in range(0,10):
        if i >= len(top10):
            top10_data = list(itertools.chain(top10_data,list_of_zeros))
        elif i ==0:
            top10_data = top10[0]
        else:
            top10_data = list(itertools.chain(top10_data,top10[i]))
    return top10_data
```

In [214]:
```
my_data = document_vector_first10(wv, corpus_train[0])
```

In [215]:
```
#4 (b) Select 10 vectors

# Create a list of strings, one for each title
train_list = [train for train in train_set['review'].dropna()]
test_list = [test for test in test_set['review'].dropna()]
import itertools

corpus_train = [preprocess(train) for train in train_list]
corpus_test = [preprocess(test) for test in test_list]

# Remove docs that don't include any words in W2V's vocab
corpus_train, train_list = filter_docs(corpus_train, train_list, lambda doc
corpus_test, test_list = filter_docs(corpus_test, test_list, lambda doc: ha

# Filter out any empty docs
corpus_train, train_list = filter_docs(corpus_train, train_list, lambda doc

print('##### preparing vectors')
x_2 = []
for doc in corpus_train: # append the vector for each document
    x_2.append(document_vector_first10(wv, doc))
X_2 = np.array(x_2) # list to array


x_test_2 = []
corpus_test, test_list = filter_docs(corpus_test, test_list, lambda doc: (l
for doc in corpus_test: # append the vector for each document
    x_test_2.append(document_vector_first10(wv, doc))
```

```
1 docs removed
0 docs removed
0 docs removed
##### preparing vectors
0 docs removed
```

In [216]:
```
# my_data_xtrain = pd.read_csv('x_train_2.csv', sep=',', header=None)
# my_data_xtest = pd.read_csv('x_test_2.csv', sep=',', header=None)

# X_2 = my_data_xtrain.to_numpy()
# x_test_2 = my_data_xtest.to_numpy()
```

In [217]:
```
# re-create y_train from the new x_train
c = np.in1d(test_set['review'],test_list)
y = test_set['star_rating'].to_numpy()
x_removed_indices = [i for i, x in enumerate(c) if not x]
y_test = np.delete(y, x_removed_indices)

# re-create y_train from the new x_train
c = np.in1d(train_set['review'],train_list)
y = train_set['star_rating'].to_numpy()
x_removed_indices = [i for i, x in enumerate(c) if not x]
y_train = np.delete(y, x_removed_indices)
```

In [218]:
```
# pd.DataFrame(X_2).to_csv("x_train_2.csv")
# pd.DataFrame(x_test_2).to_csv("x_train_2.csv")
```

In [219]:
```
#train dataset
pd_xtrain = pd.DataFrame(X_2)
pd_ytrain = pd.DataFrame(y_train, columns = ['label'])
pd_train_data = pd.concat([pd_ytrain,pd_xtrain],axis=1)

#test dataset
pd_xtest = pd.DataFrame(x_test_2)
pd_ytest = pd.DataFrame(y_test, columns = ['label'])
pd_test_data = pd.concat([pd_ytest,pd_xtest],axis=1)

# np array to Tensor

# Passing to DataLoader
X=np.vstack(X_2).astype(np.float)
y_train=np.vstack(y_train).astype(np.float)
x_test=np.vstack(x_test_2).astype(np.float)
y_test=np.vstack(y_test).astype(np.float)


tensor_train_features = torch.Tensor(X)
tensor_train_labels = torch.Tensor(y_train)
tensor_test_features= torch.Tensor(x_test)
tensor_test_labels = torch.Tensor(y_test)

tensor_train_data = torch.utils.data.TensorDataset(tensor_train_features, t
tensor_test_data = torch.utils.data.TensorDataset(tensor_test_features, ten

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 32
# percentage of training set to use as validation
valid_size = 0.2

# convert data to torch.FloatTensor
#transform = transforms.ToTensor()

# obtain training indices that will be used for validation
num_train = len(pd_train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = torch.utils.data.DataLoader(tensor_train_data, batch_size=ba
    sampler=train_sampler, num_workers=num_workers,)
valid_loader = torch.utils.data.DataLoader(tensor_train_data, batch_size=ba
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(tensor_test_data, batch_size=batc
    num_workers=num_workers)
```

```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

In [220]:

```
#define our NN model and initialize the model

import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # number of hidden nodes in each layer (512)
        hidden_1 = 50
        hidden_2 = 10
        # linear layer (300 -> hidden_1)
        self.fc1 = nn.Linear(3000, hidden_1)
        # linear layer (n_hidden -> hidden_2)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(hidden_2, 5)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # flatten image input
        #x = x.view(-1, 300)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x

# initialize the NN
model = Net()
```

In [221]:
```
# 4(b) training and testing

#number of epochs to train the model
n_epochs = 150

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    print("training. . .")
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the
        output = model(data)
        # calculate the loss
        target = target.squeeze(1)
        target = target.type(torch.LongTensor)
        target = target - 1
        #target = torch.argmax(target, dim=1)
        #print(target)
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    print("validating. . .")
    for data, target in valid_loader:
        output = model(data)
        # calculate the loss
        target = target.squeeze(1)
        target = target.type(torch.LongTensor)
        target = target - 1
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)
```

```
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.for
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```
training. . .
validating. . .
Epoch: 1        Training Loss: 1.294407         Validation Loss: 0.3238
34
Validation loss decreased (inf --> 0.323834).  Saving model ...
training. . .
validating. . .
Epoch: 2        Training Loss: 1.294376         Validation Loss: 0.3238
34
training. . .
validating. . .
Epoch: 3        Training Loss: 1.294339         Validation Loss: 0.3238
34
training. . .
validating. . .
Epoch: 4        Training Loss: 1.294411         Validation Loss: 0.3238
34
training. . .
validating. . .
```

In [222]:

```
model.load_state_dict(torch.load('model.pt'))
```

Out[222]:

```
<All keys matched successfully>
```

In [223]:
```
#4 (b) result
test_loader = torch.utils.data.DataLoader(tensor_test_data, batch_size=1, n
# Test the model
def predict(model, dataloader):
    prediction_list = []
    for i, batch in enumerate(dataloader):
        #print(batch)
        outputs = model(batch[0])
        #print(outputs.data)
        _, predicted = torch.max(outputs.data, 1)
        #print(predicted)
        #print((torch.max(outputs.data, 1)))
        prediction_list.append(predicted.cpu())
    return prediction_list

predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [248]:
```
pred= pd.DataFrame()
cor = 0
for i in range (0,len(predictions)):
    if y_test[i] == predictions[i].item() + 1:
        cor = cor + 1
print('Accuracy (Concat. 10 words): ', cor/len(predictions))
```

```
Accuracy (Concat. 10 words):  0.5134257123489022
```

In [225]:
```
# Question 5 - RNN

# 4(b)
def document_vector_first20(word2vec_model, doc):
    top20 = []
    list_of_zeros = list(0 for i in range (0,300))
    for word in doc:
        if len(top20) == 20:
            break
        if word in wv.index_to_key and len(top20) < 20:
            top20.append(wv[word])

    top20_data = []
    #print(len(top10))
    #print(top10[0])
    for i in range(0,20):
        if i >= len(top20):
            top20_data = list(itertools.chain(top20_data,list_of_zeros))
        elif i ==0:
            top20_data = top20[0]
        else:
            top20_data = list(itertools.chain(top20_data,top20[i]))
    return top20_data
```

In [226]:
```
#5 Select 20 vectors

# Create a list of strings, one for each title
train_list = [train for train in train_set['review'].dropna()]
test_list = [test for test in test_set['review'].dropna()]
import itertools

corpus_train = [preprocess(train) for train in train_list]
corpus_test = [preprocess(test) for test in test_list]

# Remove docs that don't include any words in W2V's vocab
corpus_train, train_list = filter_docs(corpus_train, train_list, lambda doc
corpus_test, test_list = filter_docs(corpus_test, test_list, lambda doc: ha

# Filter out any empty docs
corpus_train, train_list = filter_docs(corpus_train, train_list, lambda doc

print('##### preparing vectors')
x_3 = []
for doc in corpus_train: # append the vector for each document
    x_3.append(document_vector_first20(wv, doc))
X_3 = np.array(x_3) # list to array


x_test_3 = []
corpus_test, test_list = filter_docs(corpus_test, test_list, lambda doc: (l
for doc in corpus_test: # append the vector for each document
    x_test_3.append(document_vector_first20(wv, doc))
```

```
1 docs removed
0 docs removed
0 docs removed
##### preparing vectors
0 docs removed
```

In [227]:
```
# re-create y_train from the new x_train
c = np.in1d(test_set['review'],test_list)
y = test_set['star_rating'].to_numpy()
x_removed_indices = [i for i, x in enumerate(c) if not x]
y_test_3 = np.delete(y, x_removed_indices)


# re-create y_train from the new x_train
c = np.in1d(train_set['review'],train_list)
y = train_set['star_rating'].to_numpy()
x_removed_indices = [i for i, x in enumerate(c) if not x]
y_train_3 = np.delete(y, x_removed_indices)
```

In [228]:
```
import torch
def lineToTensor(subList):
    #print(len(subList))
    tensor = torch.zeros(len(subList), 300) # initialize tensor size
    subList=np.vstack(subList).astype(np.float)
    for i, review in  enumerate(subList):
        for j in range (0, len(subList[i])):
            tensor[i][j] = subList[i][j]
    return tensor

def get20vector(X):
    x_20vec = []
    sub_list = [X[n:n+300] for n in range(0, len(X), 300)]
    return sub_list
```

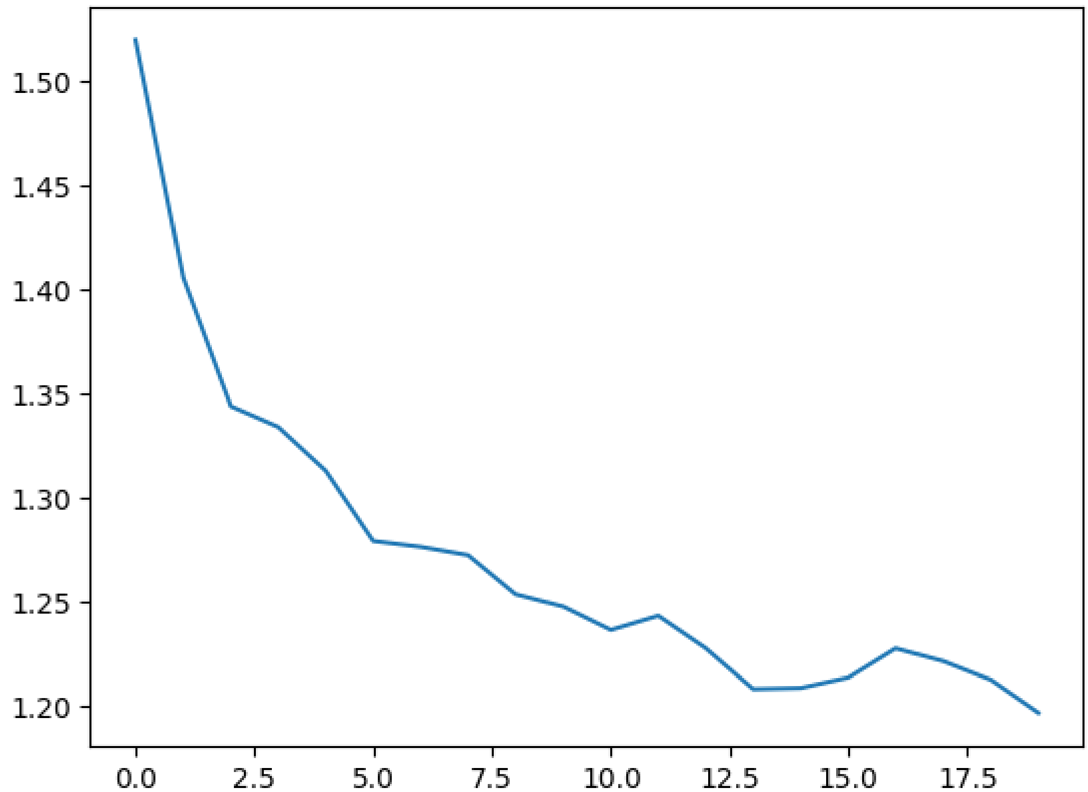
In [229]:
```
# Declare RNN class
import torch.nn as nn

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):
        return torch.zeros(20,self.hidden_size)

n_hidden = 20
output_size = 5
input_size = 300
rnn = RNN(input_size, n_hidden, output_size)
optimizer2 = torch.optim.SGD(rnn.parameters(), lr=0.05)
criterion = nn.CrossEntropyLoss()
```

In [230]:
```
learning_rate = 0.1 # If you set this too high, it might explode. If too lo

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()
    rnn.zero_grad()
    output, hidden = rnn(line_tensor, hidden)
    #output.retain_grad()
    o = output.mean(dim=0)#print(output)
    result = torch.argmax(o) + 1
    loss = criterion(o, category_tensor[0][0]-1)
    loss.backward()
    #print(loss.item())
    # perform a single optimization step (parameter update)
    optimizer2.step()

    return output, loss.item()
```

In [231]:
```
import time
import math

n_iters = 100000 # default = 100000
#print_every = 5000
plot_every = 5000



# Keep track of losses for plotting
current_loss = 0
all_losses = []

def randomTrainingExample():
    idx = np.random.choice(np.arange(len(X_3)-1), 1, replace=False) # rando
    subList = get20vector(X_3[idx[0]])
    x_sample = lineToTensor(subList)
    y_sample = y_train_3[idx]
    y_sample=np.vstack(y_sample).astype(np.float)
    category_tensor = torch.tensor(y_sample, dtype=torch.long)
    line_tensor = lineToTensor(x_sample)
    return category_tensor, line_tensor

def realTrainingExample(i):
    #idx = np.random.choice(np.arange(len(X_3)), 1, replace=False) # random
    subList = get20vector(X_3[i])
    x_sample = lineToTensor(subList)
    y_sample = y_train_3[i]
    y_sample=np.vstack(y_sample).astype(np.float)
    category_tensor = torch.tensor(y_sample, dtype=torch.long)
    line_tensor = lineToTensor(x_sample)
    return category_tensor, line_tensor

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)


#start = time.time()

# Train from random generated examples
for iter in range(1, n_iters + 1):
    category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        #print('currently at:', iter)
        #print(current_loss)
        all_losses.append(current_loss / plot_every)
        current_loss = 0
```

In [232]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
```

Out[232]: [<matplotlib.lines.Line2D at 0x7fbfea9ca4c0>]



In [233]:
```
def evaluate(line_tensor):
    hidden = rnn.initHidden()
    #print(line_tensor[0])
    output, hidden = rnn(line_tensor, hidden)

    return output
```

In [234]:
```
def predict(input_line, n_predictions=1):
    #print('\n> %s' % input_line)
    with torch.no_grad():
        output = evaluate(lineToTensor(input_line))
        o = output.mean(dim=0)
        result = torch.argmax(o) + 1
        return result
```

In [235]:
```
y_pred = []
for i in range(0,len(x_test_3)- 1):
    #print(predict(get20vector(x_test_3[i])))
    y_pred.append(predict(get20vector(x_test_3[i])))
```

In [236]:
```
correct = 0
predicted = 0
for i in range(0,len(y_test_3)-1):
    #print(y_pred[i].item(), y_test_3[i])
    #print (int(y_pred[i].item()) == int(y_test_3[i]))
    if int(y_pred[i].item()) == int(y_test_3[i]):
        correct = correct + 1
    predicted = predicted + 1
```

In [243]:
```
print('RNN Accuracy:', correct/predicted)
```

```
RNN Accuracy: 0.5123768565284793
```

In [238]:
```
# 5(b) Declare GRU, we can use the same input as 5(a) for this one

class GRUNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers, drop_pr
        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.relu = nn.ReLU()

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

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

    def initHidden(self):
        return torch.zeros(20,self.hidden_dim)

n_hidden = 20
output_size = 5
input_size = 300
gru = GRUNet(input_size, n_hidden, output_size,20)
optimizer3 = torch.optim.SGD(gru.parameters(), lr=0.001)
```

In [246]:
```
def train(category_tensor, line_tensor):
    hidden = gru.initHidden()
    gru.zero_grad()
    #for i in range(line_tensor.size()[0]):
       # hidden = rnn(line_tensor[i], hidden)
    #print(line_tensor)
    output, hidden = gru(line_tensor, hidden)
    #output.retain_grad()
    #print(output)
    o = output.mean(dim=0)#print(output)
    #print(o)
    #o.retain_grad()
    result = torch.argmax(o) + 1
    o.reshape([1])
    loss = criterion(output, category_tensor[0][0]-1)
    #loss = criterion(o.reshape([1]), category_tensor[0][0].reshape([1])-1)
    loss.backward()
    #print(loss.item())
    # perform a single optimization step (parameter update)
    optimizer3.step()

    return output, loss.item()
```

In [247]:
```
n_iters = 100000
#print_every = 5000
plot_every = 5000
for iter in range(1, n_iters-1):
    category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor)
    #print(loss)
    current_loss += loss
```

In [241]:
```
def evaluate(line_tensor):
    hidden = rnn.initHidden()
    #print(line_tensor[0])
    output, hidden = rnn(line_tensor, hidden)

    return output

def predict(input_line, n_predictions=1):
    #print('\n> %s' % input_line)
    with torch.no_grad():
        output = evaluate(lineToTensor(input_line))
        o = output.mean(dim=0)
        #print(output)
        result = torch.argmax(o) + 1
        # Get top N categories
        #topv, topi = o.topk(n_predictions, 1, True)
        #predictions = []
        #print(result)
        return result
#         for i in range(n_predictions):
#             value = topv[0][i].item()
#             category_index = topi[0][i].item()
#             print('(%.2f) %s' % (value, all_categories[category_index]))
#             predictions.append([value, all_categories[category_index]])




y_pred = []
for i in range(0,len(x_test_3)- 1):
    #print(predict(get20vector(x_test_3[i])))
    y_pred.append(predict(get20vector(x_test_3[i])))


correct = 0
predicted = 0
for i in range(0,len(y_test_3)-1):
    #print(y_pred[i].item(), y_test_3[i])
    #print (int(y_pred[i].item()) == int(y_test_3[i])
    if int(y_pred[i].item()) == int(y_test_3[i]):
        correct = correct + 1
    predicted = predicted + 1
```

In [244]:
```
print('GRU Accuracy: ', correct/predicted)
```

```
GRU Accuracy:  0.4123429958432859
```

In [ ]: