HW2-Abhay-Iyer-PT

September 27, 2022

1 Brief overview of approach for NLP HW2

Part 1

- We import the dataset and perform data cleaning as mentioned below.
- From the original dataset we keep only the 'star_rating' and 'review_body' columns to train/test multiple classifiers.
- We convert 'star_rating' to integer and 'review_body' to string for uniformity of datatypes across these columns.
- We sample 20,000 random reviews from each 'star_rating' class.
- Following data cleaning techniques have been performed on the dataset -
    1. convert all reviews to lowercase
    2. remove html tags and urls from reviews using BeautifulSoup
    3. remove extra spaces in the reviews
    4. remove punctuations
    5. remove non-alphabetical characters
- No data preprocessing techniques (like stopword removal/lemmatization) have been performed on the dataset.

Part 2

- First, we load a pretrained Word2Vec model and check for semantic similarities of various word combinations by generating their vectors. It was interesting to observe the results for a few cases. For instance, word embeddings of 'excellent' and 'outstanding' had a relatively low cosine similarity score. Additionally the combination 'germany' - 'berlin' + 'paris' had a low similarity score for 'france' and on using the most_similar() API, we notice that 'spain' has a higher cosine similarity than 'france' even though germany-berlin is a country-capital relationship so we would expect France to have the highest cosine similarity since France-paris is another country-capital relationship.
- Second, we train our own Word2Vec model on the Amazon reviews dataset and perform the same semantic similarity experiments to observe differences. More insights have been provided in the respective section.

Part 3

- In this part, we used the word embeddings generated by the pretrained Word2Vec model to train simple models like Perceptron and SVM. It was interesting to see the difference in results between these classifiers and classifiers trained on TF-IDF features. More insights have been provided in the respective section.

Part 4

- Here, we train two models which are feedforward neural networks but with different training data.
- The first model (a) - we utilize the input features generated in Part 3 to see how our model performs. We create an 80-20 split of training data like all other models and create their respective dataloaders and training loop.
- For part (b), we generate input features by concatenating the first 10 Word2Vec vectors for each review. More insights for both models are provided in their respective sections.

Part 5

- Here, we train two models with different architectures (however both are recurrent neural networks) with the same training data and compare their performances.
- For the model in part (a), we create a vanilla RNN with a hidden state size of 20 and pass it reviews which have a maximum length of 20 words. Reviews shorter than 20 words have been padded with tokens, while longer reviews have been truncated to 20 words.
- In part (b), we create a model which utilizes GRU instead of a vanilla RNN cell.
- More insights for both models are provided in their respective sections.

1.1 Imports

[5]:
```
# sklearn imports
from sklearn.metrics import classification_report
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC

# Gensim imports
import gensim.downloader as api
import gensim
from gensim.models import KeyedVectors
from gensim import models
from gensim import utils
from gensim.models import Word2Vec

# Torch imports
import torch
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F

# Utilities
import pandas as pd
import numpy as np
import nltk
import re
from bs4 import BeautifulSoup
```

```
import string
from tqdm import tqdm
from collections import Counter
```

[2]:
```
word2vec_path = '/home1/adiyer/gensim-data/word2vec-google-news-300/
↪word2vec-google-news-300.gz'
try:
    w2v_model = models.KeyedVectors.load_word2vec_format(word2vec_path,␣
↪binary=True)
except:
    w2v_model = api.load('word2vec-google-news-300')
```

2 Dataset generation

Note - 80-20 split of dataset is performed below in multiple sections of the notebook.

[3]:
```
filepath = './amazon_reviews_us_Jewelry_v1_00.tsv'
reviews_df = pd.read_csv(filepath, sep='\t', on_bad_lines='skip',␣
↪dtype='unicode')
```

[4]:
```
reviews_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1766992 entries, 0 to 1766991
Data columns (total 15 columns):
 #   Column             Dtype
---  ------             -----
 0   marketplace        object
 1   customer_id        object
 2   review_id          object
 3   product_id         object
 4   product_parent     object
 5   product_title      object
 6   product_category   object
 7   star_rating        object
 8   helpful_votes      object
 9   total_votes        object
 10  vine               object
 11  verified_purchase  object
 12  review_headline    object
 13  review_body        object
 14  review_date        object
dtypes: object(15)
memory usage: 202.2+ MB
```

2.1 Keep Reviews and Ratings

- Selecting only 'star_rating' and 'review_body'

- We use ‘review_body’ to develop the input features
- We use ‘star_rating’ as the target results which must be predicted

[6]:
```
reviews_df = reviews_df[['star_rating','review_body']]
```

[7]:
```
reviews_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1766992 entries, 0 to 1766991
Data columns (total 2 columns):
 #   Column       Dtype 
---  ------       ----- 
 0   star_rating  object
 1   review_body  object
dtypes: object(2)
memory usage: 27.0+ MB
```

2.2 Randomly selecting reviews from each star_rating_class

[8]:
```
# Converting all 'star_rating' to integer representations
# Select all rows which have 'star_rating' and 'review_body' as existing int/
 ↪string values for optimal training results of the models

reviews_df['star_rating'] = pd.
 ↪to_numeric(reviews_df['star_rating'],errors='coerce')
reviews_df = reviews_df[reviews_df['star_rating'].notna()]
reviews_df = reviews_df[reviews_df['review_body'].notna()]

#Convert all 'star_rating' to int
reviews_df['star_rating'] = reviews_df['star_rating'].astype(int)

#Convert all reviews to string
reviews_df['review_body'] = reviews_df['review_body'].astype(str)
```

[9]:
```
reviews_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1766748 entries, 0 to 1766991
Data columns (total 2 columns):
 #   Column       Dtype 
---  ------       ----- 
 0   star_rating  int64 
 1   review_body  object
dtypes: int64(1), object(1)
memory usage: 40.4+ MB
```

2.3 Sampling 20,000 samples per class

[10]:
```
rating_1 = reviews_df[reviews_df.star_rating.eq(1)].sample(20000,␣
 ↪random_state=1)
rating_2 = reviews_df[reviews_df.star_rating.eq(2)].sample(20000,␣
 ↪random_state=1)
rating_3 = reviews_df[reviews_df.star_rating.eq(3)].sample(20000,␣
 ↪random_state=1)
rating_4 = reviews_df[reviews_df.star_rating.eq(4)].sample(20000,␣
 ↪random_state=1)
rating_5 = reviews_df[reviews_df.star_rating.eq(5)].sample(20000,␣
 ↪random_state=1)

sampled_reviews_df_20000 = pd.concat([rating_1, rating_2, rating_3, rating_4,␣
 ↪rating_5])
```

[11]:
```
sampled_reviews_df_20000.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100000 entries, 344647 to 1478372
Data columns (total 2 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   star_rating  100000 non-null  int64
 1   review_body  100000 non-null  object
dtypes: int64(1), object(1)
memory usage: 2.3+ MB
```

3 Word Embedding

Answers 1. What do you conclude from comparing vectors generated by yourself and the pretrained model? The cosine similarity metric yields better results (i.e similar vectors have a higher cosine similarity score) for the pretrained model in comparison to the model trained by me. This could be attributed to multiple reasons. Firstly, the Google model has been trained on 3 billion words extracted from a news dataset collated by Google wherein the pretrained model's vocabulary size is about 3 million words. This would result in a model which is extremely efficient in understanding semantic similarities in most contexts (especially news). However the Word2Vec model trained on the Amazon reviews dataset has a vocabulary in the range of thousands which is miniscule in comparison to the pretrained model wherein it was trained on 100,000 reviews. Secondly, our Word2Vec model has been limited to an embedding size of 300 and window size of 11 with a minimum word count of 10. This limitation might impede the the performance of the model since hyperparameter tuning is restricted.

2. Which of the Word2Vec models seems to encode semantic similarities between words better? The pretrained 'word2vec-google-news-300' encodes semantic similarities better than the Word2Vec model which is trained on the Amazon reviews dataset. Overall, we could state that the pretrained Word2Vec model learned more meaningful relations between words in the corpus and encoded better semantic relationships of the words.

3.0.1 Word2Vec model semantic similarity examples

```
[12]: value1 = (w2v_model['king'] - w2v_model['man'] + w2v_model['woman']).reshape(1,␣
       ↪-1)
      value2 = (w2v_model['queen']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.73005176]]
```

```
[13]: value1 = (w2v_model['excellent']).reshape(1, -1)
      value2 = (w2v_model['outstanding']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.5567486]]
```

Other examples

```
[14]: value1 = (w2v_model['father'] - w2v_model['man'] + w2v_model['woman']).
       ↪reshape(1, -1)
      value2 = (w2v_model['mother']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.8671473]]
```

```
[15]: value1 = (w2v_model['man'] - w2v_model['woman'] + w2v_model['daughter']).
       ↪reshape(1, -1)
      value2 = (w2v_model['son']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.85979044]]
```

```
[16]: value1 = (w2v_model['excellent']).reshape(1, -1)
      value2 = (w2v_model['terrific']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.7409728]]
```

```
[17]: value1 = (w2v_model['germany'] - w2v_model['berlin'] + w2v_model['paris']).
       ↪reshape(1, -1)
      value2 = (w2v_model['france']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.54980403]]
```

Checking examples with most_similar()

```
[18]: w2v_model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
```

```
[18]: [('queen', 0.7118193507194519)]
```

```
[19]: w2v_model.most_similar(positive=['excellent'], topn=10)
```

```
[19]: [('terrific', 0.7409726977348328),
       ('superb', 0.7062715888023376),
       ('exceptional', 0.681470513343811),
       ('fantastic', 0.6802847981452942),
       ('good', 0.6442928910255432),
       ('great', 0.6124600172042847),
       ('Excellent', 0.6091997623443604),
       ('impeccable', 0.5980967283248901),
       ('exemplary', 0.5959650278091431),
       ('marvelous', 0.582928478717804)]
```

```
[20]: w2v_model.most_similar(positive=['woman', 'father'], negative=['man'], topn=1)
```

```
[20]: [('mother', 0.8462507128715515)]
```

```
[21]: w2v_model.most_similar(positive=['man', 'daughter'], negative=['woman'], topn=1)
```

```
[21]: [('son', 0.8490632772445679)]
```

```
[22]: # Interesting to note that Spain is more similar than France
      w2v_model.most_similar(positive=['germany', 'paris'], negative=['berlin'],␣
       ↪topn=10)
```

```
[22]: [('spain', 0.5414217114448547),
       ('france', 0.5339585542678833),
       ('europe', 0.5268116593360901),
       ('british', 0.5118182897567749),
       ('wiv', 0.5049731731414795),
       ('india', 0.5046922564506531),
       ('portugal', 0.5025326609611511),
       ('england', 0.49969735741615295),
       ('russia', 0.498568058013916),
       ('thailand', 0.49663135409355164)]
```

3.0.2 Training word2vec model on Amazon reviews

```
[23]: class AmazonReviewsCorpus:
          """An iterator that gives sentences as a list of strings."""
          def __iter__(self):
              for line in sampled_reviews_df_20000['review_body'].to_list():
                  yield utils.simple_preprocess(line)
```

```
[24]: reviews = AmazonReviewsCorpus()
      model = Word2Vec(sentences=reviews, vector_size=300, window=11, min_count=10)
```

```
[25]: value1 = (model.wv['king'] - model.wv['man'] + model.wv['woman']).reshape(1, -1)
      value2 = (model.wv['queen']).reshape(1, -1)
      print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.2853059]]
```

[26]:
```
value1 = (model.wv['excellent']).reshape(1, -1)
value2 = (model.wv['outstanding']).reshape(1, -1)
print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.8820023]]
```

Other examples

[27]:
```
value1 = (model.wv['father'] - model.wv['man'] + model.wv['woman']).reshape(1,␣
 ↪-1)
value2 = (model.wv['mother']).reshape(1, -1)
print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
Cosine similarity: [[0.54931146]]
```

Insights

- The error below has been left on purpose in the notebook. This shows us how powerful/huge the pretrained model is in comparison to the Word2Vec model trained on the Amazon reviews dataset. For instance, Germany is not present in the vocabulary of the model trained in part (b) which means that none of the reviews referenced this particular word. This could be improved by training our Word2Vec model on a huger dataset which could enrich its vocabulary and word embeddings.

[28]:
```
value1 = (model.wv['germany'] - model.wv['berlin'] + model.wv['paris']).
 ↪reshape(1, -1)
value2 = (model.wv['france']).reshape(1, -1)
print(f'Cosine similarity: {cosine_similarity(value1, value2)}')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/SLURM_11124997/ipykernel_69937/1438926971.py in <module>
----> 1 value1 = (model.wv['germany'] - model.wv['berlin'] + model.wv['paris'])
 ↪reshape(1, -1)
      2 value2 = (model.wv['france']).reshape(1, -1)
      3 print(f'Cosine similarity: {cosine_similarity(value1, value2)}')

~/.conda/envs/timesformer/lib/python3.7/site-packages/gensim/models/keyedvectors
 ↪py in __getitem__(self, key_or_keys)
    393         """
    394         if isinstance(key_or_keys, _KEY_TYPES):
--> 395             return self.get_vector(key_or_keys)
    396
    397         return vstack([self.get_vector(key) for key in key_or_keys])

~/.conda/envs/timesformer/lib/python3.7/site-packages/gensim/models/keyedvectors
 ↪py in get_vector(self, key, norm)
    436
    437         """
--> 438         index = self.get_index(key)
    439         if norm:
    440             self.fill_norms()

~/.conda/envs/timesformer/lib/python3.7/site-packages/gensim/models/keyedvectors.
 ↪py in get_index(self, key, default)
    410             return default
    411         else:
--> 412             raise KeyError(f"Key '{key}' not present")
    413
    414     def get_vector(self, key, norm=False):

KeyError: "Key 'germany' not present"
```

4 Simple Models

Answers 1. What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features)

- For the perceptron model we observe
  - Model trained on TFIDF Features - Overall accuracy - 50%
  - Model trained on Word2Vec Features - Overall accuracy - 39%
- For the SVM model we observe
  - Model trained on TFIDF Features - Overall accuracy - 51%
  - Model trained on Word2Vec Features - Overall accuracy - 49%

This goes to show that there is no single data preprocessing step which could lead to best results for all models. For example, the perceptron model has a significant difference in overall accuracy when trained on word2vec vs tfidf which could mean that the tfidf extracted features were better understood by the single layer perceptron in comparison to the features provided by the Google word2vec model. However for SVM, both techniques have a marginal difference and could mean that for the SVM model both techniques work and could be used in different situations based on the requirement.

Overall it looks like TFIDF features performed better for both models (When compared with only accuracy). Additionally, SVM with TFIDF features has the best accuracy on the test set.

4.1 Data cleaning and preparation

4.1.1 Utility functions

[29]:
```python
def URLRemoval(sentence):
    """Function to remove the HTML tags and URLs from reviews using␣
 ↪BeautifulSoup
```

```
    Args:
        sentence (string): Sentence from which we remove the HTML tags and URLs

    Returns:
        string: Sentence which does not contain any HTML tags and URLs
    """
    return BeautifulSoup(sentence, 'lxml').get_text()

def nonAlphabeticRemoval(sentence):
    """Function to remove non-alphabetic characters from the sentence.
    Note - We do not remove spaces from the sentence however extra spaces are␣
↪removed in a different function

    Args:
        sentence (string): Sentence from which we remove the non-alphabetic␣
↪characters

    Returns:
        string: Sentence from which non-alphabetic characters have been removed
    """
    return re.sub(r"[^a-zA-Z ]+", "", sentence)  #This will also remove numbers.

def removeExtraSpaces(sentence):
    """Remove extra spaces from the sentence

    Args:
        sentence (string): Sentence from which we remove extra spaces

    Returns:
        string: Sentence from which extra spaces have been removed
    """
    return ' '.join(sentence.split())

def removePunctuation(sentence):
    """Function to remove punctuations from a sentence

    Args:
        sentence (string): Sentence from which punctuations have to be removed

    Returns:
        string: Sentence from which punctuations have been removed
    """
    for value in string.punctuation:
        if value in sentence:
            sentence = sentence.replace(value, ' ')
    return sentence.strip()
```

```
[30]: def generateW2VFeatures(word):
          """Function to return word2vec word embeddings for a particular word

          Args:
              word (string): Word for which we need to generate embeddings

          Returns:
              np array: Vector of word embeddings which is extracted from the␣
       ↪pretrained word2vec model
          """
          return w2v_model[word]
```

```
[31]: def displayReport(actualLabels, predictedLabels, classifierName):
          """Function to display precision/recall/f1-score metrics for a classifier

          Args:
              actualLabels (_type_): True labels of the data
              predictedLabels (_type_): Labels predicted by the classifier
              classifierName (_type_): Name of the classifier which predicted the␣
       ↪labels
          """
          targetNames = ['Rating 1', 'Rating 2', 'Rating 3', 'Rating 4', 'Rating 5']
          report = classification_report(actualLabels, predictedLabels,␣
       ↪target_names=targetNames, output_dict=True)
          print(f'Accuracy: {report["accuracy"]}')
          print('============================================================')
          print(f'Precision, Recall, f1-score for Testing split for {classifierName}')
          print('============================================================')
          for targetClass in targetNames:
              print(f'{targetClass}: {report[targetClass]["precision"]},␣
       ↪{report[targetClass]["recall"]}, {report[targetClass]["f1-score"]}')
          print(f'Macro Average: {report["macro avg"]["precision"]}, {report["macro␣
       ↪avg"]["recall"]}, {report["macro avg"]["f1-score"]}')
          print('============================================================')
```

4.1.2 Data Cleaning

1. Convert all reviews into lower case
2. Remove HTML and URLs from the reviews
3. Remove punctuations
4. Remove extra spaces and non-alphabetic characters

Convert all reviews into the lower case

```
[32]: sampled_reviews_df_20000['review_body'] =␣
       ↪sampled_reviews_df_20000['review_body'].apply(lambda value:value.lower())
```

Remove the HTML and URLs from the reviews

```
[33]: sampled_reviews_df_20000['review_body'] =␣
       ↪sampled_reviews_df_20000['review_body'].apply(URLRemoval)
```

```
/home1/adiyer/.conda/envs/timesformer/lib/python3.7/site-
packages/bs4/__init__.py:439: MarkupResemblesLocatorWarning: The input looks
more like a filename than markup. You may want to open this file and pass the
filehandle into Beautiful Soup.
  MarkupResemblesLocatorWarning
```

Remove extra spaces

```
[34]: sampled_reviews_df_20000['review_body'] =␣
       ↪sampled_reviews_df_20000['review_body'].apply(removeExtraSpaces)
```

Remove punctuations

```
[35]: sampled_reviews_df_20000['review_body'] =␣
       ↪sampled_reviews_df_20000['review_body'].apply(removePunctuation)
```

Remove extra spaces - doing it again because removing punctuations creates extra spaces

```
[36]: sampled_reviews_df_20000['review_body'] =␣
       ↪sampled_reviews_df_20000['review_body'].apply(removeExtraSpaces)
```

Remove non-alphabetical characters (excluding spaces)

```
[37]: sampled_reviews_df_20000['review_body'] =␣
       ↪sampled_reviews_df_20000['review_body'].apply(nonAlphabeticRemoval)
```

```
[38]: sampled_reviews_df_20000.head(10)
```

```
[38]:          star_rating                                        review_body
      344647             1  bought two pairs one set for each twin one pai…
      903655             1  i liked the earring but the back didn t stay o…
      600617             1                             smaller than i expected
      1075405            1  when i saw the price for the ring pictured i j…
      1001976            1  piece of crap i cannot believe how bad this ri…
      281719             1  the balls wouldn t stay in the g already lost …
      1572384            1  it looks cheap and the eyes of hello kitty lef…
      499950             1               they broke the first day i wore them
      1283936            1  so i purchased this item thinking it d be a go…
      322181             1   tarnished after a few months would not recommend
```

4.1.3 Generate W2V Features

- Compute average Word2Vec vectors for each review as the input feature by averaging the vector embeddings of the reviews
- If the word does not exist in the pretrained Word2Vec model's vocabulary - we add a vector filled with 0s

```
[39]: unknownKeyValue, finalW2VFeatures = np.zeros((300,)), []

      finalW2VFeatures = [np.mean([generateW2VFeatures(word) if word in w2v_model␣
       ↪else unknownKeyValue for word in sentence.split(' ') ],axis=0) for sentence␣
       ↪in tqdm(sampled_reviews_df_20000['review_body'].tolist())]
      arrayW2VFeatures = np.array(finalW2VFeatures)
```

```
100%|      | 100000/100000 [00:08<00:00, 11619.94it/s]
```

```
[40]: arrayW2VFeatures.shape
```

```
[40]: (100000, 300)
```

80-20 train-test split is created in the next section with stratification being performed for equal representation of classes in the training and test sets.

```
[41]: trainData, testData, trainLabels, testLabels =␣
       ↪train_test_split(arrayW2VFeatures, sampled_reviews_df_20000['star_rating'].
       ↪to_list(), test_size=0.2, random_state=42,␣
       ↪stratify=sampled_reviews_df_20000['star_rating'].to_list())
```

```
[42]: print(len(trainData), len(testData))
```

```
80000 20000
```

4.2 Perceptron

```
[43]: perceptronModel = Perceptron(eta0=0.1, tol=1e-5, n_jobs=-1, max_iter=5000)
```

```
[44]: perceptronModel.fit(trainData, trainLabels)
```

```
[44]: Perceptron(eta0=0.1, max_iter=5000, n_jobs=-1, tol=1e-05)
```

```
[45]: predictedLabels = perceptronModel.predict(testData)
      displayReport(testLabels, predictedLabels, 'Perceptron')
```

```
Accuracy: 0.38985
============================================================
Precision, Recall, f1-score for Testing split for Perceptron
============================================================
Rating 1: 0.4558487113008998, 0.74725, 0.5662593539831392
Rating 2: 0.3593314763231198, 0.16125, 0.22260569456427956
Rating 3: 0.37103448275862067, 0.269, 0.3118840579710145
Rating 4: 0.33056158110388045, 0.68575, 0.4460887949260042
Rating 5: 0.7644444444444445, 0.086, 0.1546067415730337
Macro Average: 0.4562441391861931, 0.38985, 0.3402889286034942
============================================================
```

4.3 SVM

```
[46]: svmModel = LinearSVC().fit(trainData, trainLabels)
```

```
[47]: predictedLabels = svmModel.predict(testData)
      displayReport(testLabels, predictedLabels, 'SVM')
```

```
Accuracy: 0.48695
============================================================
Precision, Recall, f1-score for Testing split for SVM
============================================================
Rating 1: 0.5092035093755376, 0.74, 0.6032813614592887
Rating 2: 0.40763052208835343, 0.25375, 0.31278890600924497
Rating 3: 0.4087530966143683, 0.37125, 0.3890999606969737
Rating 4: 0.4299965600275198, 0.3125, 0.3619516432604604
Rating 5: 0.5873569904983518, 0.75725, 0.6615703833133122
Macro Average: 0.46858813572082614, 0.48695000000000005, 0.46573845094785593
============================================================
```

5 Feedforward Neural Networks

5.0.1 Answers

1. What do you conclude by comparing accuracy values you obtain with those obtained in the "Simple Models" section?

- From the accuracies achieved by the Feedforward neural networks we observe that there is not a huge difference/significant improvement in performance compared to the simple models. For instance, the SVM with TFIDF features outperforms all the models (simple and feedforward models). Limiting the scope of models to HW2 - we observe that the Feedforward neural network in 4A achieves the highest accuracy value of 0.5066 which marginally better when compared to the SVM model trained on Word2Vec features. This leads us to the conclusion that statistical models like the SVM/Single layer perceptron can still be leveraged for such scenarios and it is interesting to observe that more powerful models do not always lead to the best performances.

5.1 Dataset creation

- Creating a custom dataset class for Amazon reviews to enable the creation of the relevant dataloaders

```
[48]: class CustomAmazonDataset(Dataset):
          def __init__(self, data, labels, transform=None):
              self.data = data
              self.labels = labels
              self.transform = transform

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

```
    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

- We modify the train/test labels of this dataset since we utilize the cross-entropy loss function. Class labels must range from [0, C) where C is the number of classes.

[49]:
```
# Modification required for cross-entropy loss.
def modifyLabels(labels):
    """Function to modify labels and reduce them by 1 due to the requirements␣
 ↪of cross-entropy loss for target variables

    Args:
        labels (list): List of labels

    Returns:
        list: List of labels with every element reduced by 1 since our ratings␣
 ↪are in the range of [1,5]. We return a list with values in range of [0,4]
    """
    for i in range(len(labels)):
        labels[i] -= 1
    return labels
```

[50]:
```
trainLabels = modifyLabels(trainLabels)
testLabels = modifyLabels(testLabels)
```

[51]:
```
dataTrain = CustomAmazonDataset(trainData, trainLabels, transform=transforms.
 ↪ToTensor())
dataTest = CustomAmazonDataset(testData, testLabels, transform=transforms.
 ↪ToTensor())
```

DataLoader

[52]:
```
## Hyperparameters
batch_size = 512
epochs = 30
```

[53]:
```
trainLoader = DataLoader(dataTrain, batch_size=batch_size, shuffle=True)
testLoader = DataLoader(dataTest, batch_size=batch_size, shuffle=True)
```

5.2 Part - 4a

- Accuracy achieved - 0.5066

5.2.1 Network Architecture

```
[54]: class MultiLayerPerceptron(nn.Module):
          def __init__(self, inputDim):
              super(MultiLayerPerceptron, self).__init__()
              hiddenLayer1 = 50
              hiddenLayer2 = 10
              self.fc1 = nn.Linear(inputDim, hiddenLayer1)
              self.fc2 = nn.Linear(hiddenLayer1, hiddenLayer2)
              self.fc3 = nn.Linear(hiddenLayer2, 5)
              # self.dropout = nn.Dropout(0.2)

          def forward(self, x):
              x = F.relu(self.fc1(x))
              x = F.relu(self.fc2(x))
              x = self.fc3(x)
              return x
```

5.2.2 Model Training

```
[55]: ######### MODEL CREATION ###########
      inputDim = 300
      model = MultiLayerPerceptron(inputDim)
      criterion = nn.CrossEntropyLoss()
      optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
      device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
      model = model.to(device)


      ######### MODEL TRAINING ###########
      for epoch in range(epochs):
          trainLoss = 0.0
          validationLoss = 0.0

          model.train()
          for data, label in trainLoader:
              data, label = data.to(device), label.to(device)
              optimizer.zero_grad()
              output = model(data.float())
              loss = criterion(output, label)
              loss.backward()
              optimizer.step()
              trainLoss += loss.item()
          trainLoss = trainLoss/len(trainLoader.dataset)
          print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, trainLoss))
```

```
Epoch: 1        Training Loss: 0.002582
Epoch: 2        Training Loss: 0.002321
```

```
Epoch: 3        Training Loss: 0.002277
Epoch: 4        Training Loss: 0.002258
Epoch: 5        Training Loss: 0.002235
Epoch: 6        Training Loss: 0.002225
Epoch: 7        Training Loss: 0.002212
Epoch: 8        Training Loss: 0.002195
Epoch: 9        Training Loss: 0.002189
Epoch: 10       Training Loss: 0.002184
Epoch: 11       Training Loss: 0.002179
Epoch: 12       Training Loss: 0.002163
Epoch: 13       Training Loss: 0.002157
Epoch: 14       Training Loss: 0.002156
Epoch: 15       Training Loss: 0.002148
Epoch: 16       Training Loss: 0.002141
Epoch: 17       Training Loss: 0.002137
Epoch: 18       Training Loss: 0.002137
Epoch: 19       Training Loss: 0.002128
Epoch: 20       Training Loss: 0.002119
Epoch: 21       Training Loss: 0.002121
Epoch: 22       Training Loss: 0.002117
Epoch: 23       Training Loss: 0.002112
Epoch: 24       Training Loss: 0.002108
Epoch: 25       Training Loss: 0.002107
Epoch: 26       Training Loss: 0.002104
Epoch: 27       Training Loss: 0.002095
Epoch: 28       Training Loss: 0.002091
Epoch: 29       Training Loss: 0.002088
Epoch: 30       Training Loss: 0.002094
```

[57]:
```
def predict(model, dataloader, modelName, device):
    """Function to utilize the trained model and predict on the samples in the␣
 ↪dataloader. Also prints the accuracy of the model.

    Args:
        model (MultiLayerPerceptron): Trained MLP model
        dataloader (Dataloader): Word for which we need to generate embeddings
        modelName (string): Name of model to differentiate between different␣
 ↪MLP models (For question 4a and 4b)
        device (string): 'cuda' or 'cpu' based on GPU availability
    """
    model.eval()
    correct = 0
    for inputs, targets in dataloader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs.float())
        value, predictions = torch.max(outputs.data, 1)
        correct += (predictions == targets).sum().item()
    print(f'Accuracy of {modelName} - {correct/len(dataloader.dataset)}')
```

[58]:
```
predict(model, testLoader, 'FFNN - Part A', device)
```

```
Accuracy of FFNN - Part A - 0.5066
```

5.3 Part 4b

- Accuracy achieved - 0.3945

5.3.1 Generate new W2V Features

- Iterate over the reviews and concatenate the first 10 word embeddings. If the review is shorter than 10 words, we add padding in the form of an unknownKeyValue which is an np array of 0s

[59]:
```
# Concatenate first 10 Word2Vec vectors for each review and train the neural␣
↪network
unknownKeyValue, newFinalW2VFeatures = np.zeros((300,)), []

for sentence in tqdm(sampled_reviews_df_20000['review_body'].tolist()):
    sentenceList, outputArr = sentence.split(' '), []

    for word in sentenceList:
        if word in w2v_model:
            outputArr.append(generateW2VFeatures(word))
        else:
            outputArr.append(unknownKeyValue)

    if len(sentenceList) < 10:
        for i in range(10 - len(sentenceList)):
            outputArr.append(unknownKeyValue)
    newFinalW2VFeatures.append(np.array(outputArr[:10]).reshape(3000,))

newArrayW2VFeatures = np.array(newFinalW2VFeatures)
newArrayW2VFeatures.shape
```

```
100%|      | 100000/100000 [00:07<00:00, 12940.98it/s]
```

[59]:
```
(100000, 3000)
```

[60]:
```
newTrainData, newTestData, newTrainLabels, newTestLabels =␣
↪train_test_split(newArrayW2VFeatures,␣
↪sampled_reviews_df_20000['star_rating'].to_list(), test_size=0.2,␣
↪random_state=42, stratify=sampled_reviews_df_20000['star_rating'].to_list())
```

[61]:
```
newTrainLabels = modifyLabels(newTrainLabels)
newTestLabels = modifyLabels(newTestLabels)
```

[62]:
```
newDataTrain = CustomAmazonDataset(newTrainData, newTrainLabels,␣
 ↪transform=transforms.ToTensor())
newDataTest = CustomAmazonDataset(newTestData, newTestLabels,␣
 ↪transform=transforms.ToTensor())
```

[63]:
```
newTrainLoader = DataLoader(newDataTrain, batch_size=batch_size, shuffle=True)
newTestLoader = DataLoader(newDataTest, batch_size=batch_size)
```

[66]:
```
######### MODEL CREATION ###########
inputDim = 3000
modelB = MultiLayerPerceptron(inputDim)
criterionB = nn.CrossEntropyLoss()
optimizerB = torch.optim.Adam(modelB.parameters(), lr=1e-3)
deviceB = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
modelB = modelB.to(deviceB)


######### MODEL TRAINING ###########
for epoch in range(epochs):
    trainLoss = 0.0
    modelB.train()
    for data, label in newTrainLoader:
        data, label = data.to(deviceB), label.to(deviceB)
        optimizerB.zero_grad()
        output = modelB(data.float())
        loss = criterionB(output, label)
        loss.backward()
        optimizerB.step()
        trainLoss += loss.item()
    trainLoss = trainLoss/len(newTrainLoader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, trainLoss))
```

```
Epoch: 1        Training Loss: 0.002847
Epoch: 2        Training Loss: 0.002579
Epoch: 3        Training Loss: 0.002509
Epoch: 4        Training Loss: 0.002458
Epoch: 5        Training Loss: 0.002406
Epoch: 6        Training Loss: 0.002357
Epoch: 7        Training Loss: 0.002309
Epoch: 8        Training Loss: 0.002259
Epoch: 9        Training Loss: 0.002214
Epoch: 10       Training Loss: 0.002169
Epoch: 11       Training Loss: 0.002123
Epoch: 12       Training Loss: 0.002082
Epoch: 13       Training Loss: 0.002044
Epoch: 14       Training Loss: 0.001999
Epoch: 15       Training Loss: 0.001962
Epoch: 16       Training Loss: 0.001926
```

```
Epoch: 17       Training Loss: 0.001886
Epoch: 18       Training Loss: 0.001854
Epoch: 19       Training Loss: 0.001816
Epoch: 20       Training Loss: 0.001782
Epoch: 21       Training Loss: 0.001749
Epoch: 22       Training Loss: 0.001724
Epoch: 23       Training Loss: 0.001690
Epoch: 24       Training Loss: 0.001652
Epoch: 25       Training Loss: 0.001630
Epoch: 26       Training Loss: 0.001594
Epoch: 27       Training Loss: 0.001567
Epoch: 28       Training Loss: 0.001544
Epoch: 29       Training Loss: 0.001515
Epoch: 30       Training Loss: 0.001489
```

[67]:
```
predict(modelB, newTestLoader, 'FFNN - Part B', deviceB)
```

```
Accuracy of FFNN - Part B - 0.3945
```

6 Recurrent Neural Networks

6.0.1 Answers

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

The accuracy value obtained by the simple RNN is 0.47445 which is better than the accuracy value obtained by the FFNN in 4b (0.3945) but less in comparison to the FFNN model trained in 4a (0.5066). This shows that even though we're utilizing RNNs which are supposedly better for the task of analyzing sequential data - our results indicate a mixed performance. This could be attributed to multiple reasons. Our simple RNN could be improved in architecture to outperform the FFNN models. For instance, we could enhance our model by utilizing regularization techniques like dropout which might show better generalization on the test dataset. Hyperparameters could be better tuned to improve the RNN/FFNN performances.

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

In 5b - our model is trained with a GRU layer instead of RNN. Here, we observe that the model with GRU performs marginally better (0.49195) in comparison to the model with RNN (0.47445). This could potentially be due to the structure of the GRU wherein the gates determines the amount of previous sequential information that must pass to the next state. This also helps reduce the chances of experiencing vanishing gradients which Vanilla RNNs are extremely prone to.

6.1 Part 5a

- Accuracy achieved - 0.47445

[68]:
```
class RecNeuralNet(nn.Module):
    def __init__(self, vocabSize, embeddingDim, hiddenDim, embeddings,␣
↪outputSize):
        super(RecNeuralNet, self).__init__()
        self.embeddings = nn.Embedding(vocabSize, embeddingDim, padding_idx=0)
        self.embeddings.weight.data.copy_(torch.from_numpy(embeddings))
        self.embeddings.weight.requires_grad = False
        self.rnn = nn.RNN(embeddingDim, hiddenDim, batch_first=True,␣
↪nonlinearity='relu')
        self.fc = nn.Linear(hiddenDim, outputSize)

    def forward(self, input):
        output = self.embeddings(input)
        rnnOutput, hidden = self.rnn(output)
        output = self.fc(hidden[-1])
        return output
```

6.1.1 Dataset creation

Generate W2V Features

Counting the occurrences of words in the Amazon reviews corpus

[69]:
```
counter = Counter()
for idx, rowValue in sampled_reviews_df_20000.iterrows():
    counter.update(rowValue['review_body'].split(' '))
```

Defining an embedding matrix

- We create a function to return the pretrained word embeddings for words which occur in the amazon reviews corpus.
- Additionally, we also return the vocabulary of the corpus i.e all the words which occur in the corpus.
- Moreover, we generate indices for every word in the corpus and add two more word embeddings. ”” - for padding with a word embedding equal to 0 and “unknown” for words which do not have word embeddings in the pretrained Word2Vec model. For words which are associated with the ‘unknown’ tag - we generate a word embedding by drawing samples from a uniform distribution.

[70]:
```
def computeEmbeddingMatrix(originalWordEmbeddings, words, embeddingDim = 300):
    """Function to compute the embedding matrix for the Amazon reviews dataset.␣
↪This embedding matrix is used in the RNN.

    Args:
        originalWordEmbeddings (KeyedVector): Pretrained word2vec model
        words (dictionary): Dictionary which stores all words which occur in␣
↪the corpus and their counts
        embeddingDim (int): Embedding dimension of the word vectors

    Returns:
        np.array: Stores the word embeddings at indices which match the indices␣
↪mapped in the dictionary returned to the user
        np.array: Array which contains all the words which occur in the corpus
        dict: Dictionary which maps all words in the vocabulary to indices

    """
    vocabSize, vocabIndicesDict, vocab = len(words) + 2, {}, ['', 'unknown']
    index = 2 # We initialize the first 2 elements of amazonWeightsMatrix below␣
↪manually. All other elements are set in the for loop.
    vocabIndicesDict[vocab[0]], vocabIndicesDict[vocab[1]] = 0, 1
    amazonWeightsMatrix = np.zeros((vocabSize, embeddingDim), dtype="float32")

    # amazonWeightsMatrix[0] - word embedding for padding token
    # amazonWeightsMatrix[1] - word embedding for unknown token

    amazonWeightsMatrix[0], amazonWeightsMatrix[1]= np.zeros(embeddingDim,␣
↪dtype='float32'), np.random.uniform(-0.25, 0.25, embeddingDim)

    for word in words:
        amazonWeightsMatrix[index] = originalWordEmbeddings[word] if word in␣
↪originalWordEmbeddings else np.random.uniform(-0.25,0.25, embeddingDim)
        vocabIndicesDict[word] = index
        vocab.append(word)
        index += 1
    return amazonWeightsMatrix, np.array(vocab), vocabIndicesDict
```

- Iterate over the reviews and limit the maximum review length to 20. If the review is shorter than 20 words, we add padding which is mapped to an np array of 0s. If it is greater than 20 words, we truncate the review and only take the first 20 words into consideration
- Here padding is 0 and unknown key value is 1. These are indices/locations of these tokens in the amazonWeightsMatrix
- In the inner for loop we create an array(outputArr) of indices which is then appended to the outer array newFinalW2VFeatures.
- Thus, we create a (100000,20) vector where every review has 20 tokens which are the indices of it's word embeddings in the amazonWeightsMatrix
- We load the amazonWeightsMatrix as pretrained weights for the Embeddings layer in the RNN architecture and pass it the indices to help with the lookup.

[73]:
```
padding, unknownKeyValue, newFinalW2VFeatures = 0, 1, []
amazonWeightsMatrix, vocab, vocabDict = computeEmbeddingMatrix(w2v_model,␣
↪counter)

for sentence in tqdm(sampled_reviews_df_20000['review_body'].tolist()):
    sentenceList, outputArr = sentence.split(' '), []

    for word in sentenceList:
        if word in vocabDict:
            outputArr.append(vocabDict[word])
        else:
            outputArr.append(unknownKeyValue)
    if len(sentenceList) < 20:                          # Limit the max review␣
 ↪length to 20
        for i in range(20 - len(sentenceList)):
            outputArr.append(padding)
    newFinalW2VFeatures.append(np.array(outputArr[:20]))

newFinalW2VFeatures = np.array(newFinalW2VFeatures)
newFinalW2VFeatures.shape
```

```
100%|      | 100000/100000 [00:01<00:00, 77315.30it/s]
```

[73]:
```
(100000, 20)
```

[74]:
```
trainData, testData, trainLabels, testLabels =␣
 ↪train_test_split(newFinalW2VFeatures,␣
 ↪sampled_reviews_df_20000['star_rating'].to_list(), test_size=0.2,␣
 ↪random_state=42, stratify=sampled_reviews_df_20000['star_rating'].to_list())
```

[75]:
```
trainLabels = modifyLabels(trainLabels)
testLabels = modifyLabels(testLabels)
```

[76]:
```
dataTrain = CustomAmazonDataset(trainData, trainLabels, transform=transforms.
 ↪ToTensor())
dataTest = CustomAmazonDataset(testData, testLabels, transform=transforms.
 ↪ToTensor())
```

[77]:
```
## Hyperparameters
batchSize = 512
epochs = 30
```

[78]:
```
trainLoader = DataLoader(dataTrain, batch_size=batchSize, shuffle=True)
testLoader = DataLoader(dataTest, batch_size=batchSize)
```

6.1.2 Model Training

[79]:
```
############## MODEL CREATION ##############
hiddenDim = 20
outputDim = 5
learningRate = 1e-3
numEpochs = 30
embeddingDim = 300

model = RecNeuralNet(vocab.shape[0], embeddingDim, hiddenDim,␣
 ↪amazonWeightsMatrix, outputDim)
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learningRate)

############## MODEL TRAINING ##############
for epoch in range(numEpochs):
    trainLoss = 0.0
    for data, labels in trainLoader:
        data, labels = data.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        trainLoss += loss.item()
    trainLoss = trainLoss/len(trainLoader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, trainLoss))
```

```
Epoch: 1 	Training Loss: 0.003137
Epoch: 2 	Training Loss: 0.002718
Epoch: 3 	Training Loss: 0.002579
Epoch: 4 	Training Loss: 0.002528
Epoch: 5 	Training Loss: 0.002494
Epoch: 6 	Training Loss: 0.002461
Epoch: 7 	Training Loss: 0.002431
Epoch: 8 	Training Loss: 0.002404
Epoch: 9 	Training Loss: 0.002383
Epoch: 10 	Training Loss: 0.002366
Epoch: 11 	Training Loss: 0.002354
Epoch: 12 	Training Loss: 0.002343
Epoch: 13 	Training Loss: 0.002332
Epoch: 14 	Training Loss: 0.002326
Epoch: 15 	Training Loss: 0.002318
Epoch: 16 	Training Loss: 0.002313
Epoch: 17 	Training Loss: 0.002306
Epoch: 18 	Training Loss: 0.002302
Epoch: 19 	Training Loss: 0.002293
Epoch: 20 	Training Loss: 0.002291
Epoch: 21 	Training Loss: 0.002284
Epoch: 22 	Training Loss: 0.002282
Epoch: 23 	Training Loss: 0.002278
Epoch: 24 	Training Loss: 0.002282
Epoch: 25 	Training Loss: 0.002274
Epoch: 26 	Training Loss: 0.002268
Epoch: 27 	Training Loss: 0.002262
Epoch: 28 	Training Loss: 0.002264
Epoch: 29 	Training Loss: 0.002260
```

```
Epoch: 30       Training Loss: 0.002257
```

[80]:
```
def predictRNN(model, dataloader, modelName, device):
    """Function to utilize the trained model and predict on the samples in the␣
 ↪dataloader. Also prints the accuracy of the model.

    Args:
        model (RecNeuralNet/RecNeuralNetGRU): Trained RNN/GRU model
        dataloader (Dataloader): Dataloader which contains the test data
        modelName (string): Name of model to differentiate between different␣
 ↪RNN/GRU models (For question 5a and 5b)
        device (string): 'cuda' or 'cpu' based on GPU availability
    """
    model.eval()
    correct = 0
    for inputs, targets in dataloader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        value, predictions = torch.max(outputs.data, 1)
        correct += (predictions == targets).sum().item()
    print(f'Accuracy of {modelName} - {correct/len(dataloader.dataset)}')
```

[81]:
```
predictRNN(model, testLoader, 'RNN', device)
```

```
Accuracy of RNN - 0.47445
```

6.2 Part 5b

- Accuracy achieved - 0.49195

[82]:
```
class RecNeuralNetGRU(nn.Module):
    def __init__(self, vocabSize, embeddingDim, hiddenDim, embeddings,␣
 ↪outputSize):
        super(RecNeuralNetGRU, self).__init__()
        self.embeddings = nn.Embedding(vocabSize, embeddingDim, padding_idx=0)
        self.embeddings.weight.data.copy_(torch.from_numpy(embeddings))
        self.embeddings.weight.requires_grad = False
        self.gru = nn.GRU(embeddingDim, hiddenDim, batch_first=True)
        self.fc = nn.Linear(hiddenDim, outputSize)

    def forward(self, input):
        output = self.embeddings(input)
        gruOutput, hidden = self.gru(output)
        output = self.fc(hidden[-1])
        return output
```

[84]:
```
############## MODEL CREATION ##############
hiddenDim = 20
```

```
outputDim = 5
learningRate = 1e-2
numEpochs = 15
embeddingDim = 300

model = RecNeuralNetGRU(vocab.shape[0], embeddingDim, hiddenDim,␣
 ↪amazonWeightsMatrix, outputDim)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learningRate)

############## MODEL TRAINING ##############
for epoch in range(numEpochs):
    trainLoss = 0.0
    for data, labels in trainLoader:
        data, labels = data.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        trainLoss += loss.item()
    trainLoss = trainLoss/len(trainLoader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, trainLoss))
```

```
Epoch: 1        Training Loss: 0.002520
Epoch: 2        Training Loss: 0.002263
Epoch: 3        Training Loss: 0.002204
Epoch: 4        Training Loss: 0.002168
Epoch: 5        Training Loss: 0.002139
Epoch: 6        Training Loss: 0.002120
Epoch: 7        Training Loss: 0.002098
Epoch: 8        Training Loss: 0.002079
Epoch: 9        Training Loss: 0.002064
Epoch: 10       Training Loss: 0.002049
Epoch: 11       Training Loss: 0.002028
Epoch: 12       Training Loss: 0.002020
Epoch: 13       Training Loss: 0.002011
Epoch: 14       Training Loss: 0.001996
Epoch: 15       Training Loss: 0.001985
```

[85]:
```
# 15 epochs
predictRNN(model, testLoader, 'GRU', device)
```

```
Accuracy of GRU - 0.49195
```