

▼ Dependencies

▼ Install

```
1 !pip install contractions
2 !pip install ipython-autotime
3 !pip install fastparquet
```

Requirement already satisfied: contractions in /usr/local/lib/python3.10/dist-packages (0.1.73)
Requirement already satisfied: textsearch>=0.0.21 in /usr/local/lib/python3.10/dist-packages (from contractions) (0.0.24)
Requirement already satisfied: anyascii in /usr/local/lib/python3.10/dist-packages (from textsearch>=0.0.21->contractions) (0.3.2)
Requirement already satisfied: pyahocorasick in /usr/local/lib/python3.10/dist-packages (from textsearch>=0.0.21->contractions) (2.0.0)
Requirement already satisfied: ipython-autotime in /usr/local/lib/python3.10/dist-packages (0.3.1)
Requirement already satisfied: ipython in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (7.34.0)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (67.7.2)
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.19.1)
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (4.4.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.7.5)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (5.7.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!<3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (3.0.39)
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (2.16.1)
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.2.0)
Requirement already satisfied: matplotlib-inline in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.1.6)
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (4.8.0)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ipython->ipython-autotime) (0.8.3)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.10/dist-packages (from pexpect>4.3->ipython->ipython-autotime) (0.7.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!<3.0.1,<3.1.0,>=2.0.0->ipython->ipython-autotime) (0.2.8)
Requirement already satisfied: fastparquet in /usr/local/lib/python3.10/dist-packages (2023.8.0)
Requirement already satisfied: pandas>=1.5.0 in /usr/local/lib/python3.10/dist-packages (from fastparquet) (1.5.3)
Requirement already satisfied: numpy>=1.20.3 in /usr/local/lib/python3.10/dist-packages (from fastparquet) (1.23.5)
Requirement already satisfied: cramjam>=2.3 in /usr/local/lib/python3.10/dist-packages (from fastparquet) (2.7.0)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from fastparquet) (2023.6.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from fastparquet) (23.2)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.5.0->fastparquet) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.5.0->fastparquet) (2023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas>=1.5.0->fastparquet) (1.16.0)

▼ Imports

```
1 import os
2 import re
3 import shutil
4 import unicodedata
5 import multiprocessing
6
7 import warnings
8 warnings.filterwarnings("ignore")
9
10 import numpy as np
11 import pandas as pd
12 import requests
13
14 import nltk
15 from nltk.corpus import stopwords, wordnet
16 from nltk.stem import WordNetLemmatizer
17 from nltk.tokenize import word_tokenize
18
19 nltk.download('punkt', quiet=True)
20 nltk.download('wordnet', quiet=True)
21 nltk.download('stopwords', quiet=True)
22 nltk.download('averaged_perceptron_tagger', quiet=True)
23
24 import contractions
25
26 import gensim
27 import gensim.downloader as api
28 from gensim.models import Word2Vec
29
30 from sklearn.model_selection import train_test_split
31 from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
32
33 from sklearn.linear_model import Perceptron
34 from sklearn.svm import LinearSVC
35
36 import torch
37 import torch.nn as nn
38 import torch.optim as optim
39 from torch.utils.data.sampler import RandomSampler, BatchSampler
40 from torch.utils.data import Dataset, DataLoader
41
42 from tqdm.notebook import tqdm
43
44 %load_ext autotime
```

time: 467 µs (started: 2023-10-19 22:59:24 +00:00)

▼ Config

Set up important configuration parameters and file paths for the project, making it easy to manage various settings and paths from one centralized location

```
1 os.chdir("/content/drive/MyDrive/Colab Notebooks/CSCI544/HW3")
2 os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
3
4 CURRENT_DIR = os.getcwd()
5
6
7 class DatasetConfig:
8     RANDOM_STATE = 34
9     TEST_SPLIT = 0.2
10    N_SAMPLES_EACH_CLASS = 50000
11    DATA_PATH = os.path.join(
12        CURRENT_DIR, "amazon_reviews_us_Office_Products_v1_00.tsv.gz"
13    )
14    PROCESSED_DATA_PATH = os.path.join(
15        CURRENT_DIR, "amazon_review_processed_sentiment_analysis.parquet"
16    )
17    PREPROCESSED_DATA_PATH = os.path.join(
18        CURRENT_DIR, "amazon_review_preprocessed_sentiment_analysis.parquet"
19    )
```

```
20 BUILD_NEW = True
21 if os.path.exists(PROCESSED_DATA_PATH) and os.path.exists(PREPROCESSED_DATA_PATH):
22     BUILD_NEW = False
23
24
25 class Word2VecConfig:
26     PRETRAINED_MODEL = "word2vec-google-news-300"
27     PRETRAINED_DEFAULT_SAVE_PATH = os.path.join(
28         gensim.downloader.BASE_DIR, PRETRAINED_MODEL, f"{PRETRAINED_MODEL}.gz"
29     )
30     PRETRAINED_MODEL_SAVE_PATH = os.path.join(
31         CURRENT_DIR, PRETRAINED_MODEL, f"{PRETRAINED_MODEL}.gz"
32     )
33     WINDOW_SIZE = 13
34     MAX_LENGTH = 300
35     MIN_WORD_COUNT = 9
36     CUSTOM_MODEL_PATH = os.path.join(CURRENT_DIR, "word2vec-custom.model")
```

time: 15.6 ms (started: 2023-10-19 22:59:24 +00:00)

▼ Helper Functions

▼ Download & Save Pretrained model

- Run the `api.load()` once and copied the model from temporary path to local drive for fast loading of model in memory.

References:

1. [Faster way to load word2vec model](#)
2. [Tutorial](#)

```
1 def load_pretrained_model():
2     if not os.path.exists(Word2VecConfig.PRETRAINED_MODEL_SAVE_PATH):
3         # Create a directory if it doesn't exist
4         os.makedirs(Word2VecConfig.PRETRAINED_MODEL, exist_ok=True)
5         # Download the model embeddings
6         pretrained_model = api.load(Word2VecConfig.PRETRAINED_MODEL, return_path=True)
7         # Copy & save the embeddings file
8         shutil.copyfile(
9             Word2VecConfig.PRETRAINED_DEFAULT_SAVE_PATH, Word2VecConfig.PRETRAINED_MODEL
10        )
11    else:
12        pretrained_model = gensim.models.keyedvectors.KeyedVectors.load_word2vec_format(
13            Word2VecConfig.PRETRAINED_MODEL_SAVE_PATH, binary=True
14        )
15    return pretrained_model
16
17
18 # Load the pretrained model
19 pretrained_model = load_pretrained_model()
```

time: 2min 1s (started: 2023-10-19 18:38:29 +00:00)

▼ Accelerator Configuration

```
1 def get_device():
2     if torch.cuda.is_available():
3         # Check if GPU is available
4         return torch.device("cuda")
5     else:
6         # Use CPU if no GPU or TPU is available
7         return torch.device("cpu")
8
9 device = get_device()
10 device
```

device(type='cpu')time: 11.8 ms (started: 2023-10-19 22:59:24 +00:00)

▼ Download Data

Checks if a file specified by `DatasetConfig.DATA_PATH` exists. If not, it downloads the file from a given URL and saves it with the same name. If the file already exists, it prints a message indicating so

```
1 if not os.path.exists(DatasetConfig.DATA_PATH):
2     url = (
3         "https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.com/amazon-reviews-pds"
4         "/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz"
5     )
6     file_name = DatasetConfig.DATA_PATH
7
8     with requests.get(url, stream=True) as response:
9         with open(file_name, "wb") as file:
10             for chunk in response.iter_content(chunk_size=8192):
11                 file.write(chunk)
12
13     print(f"Downloaded file '{os.path.relpath(file_name)}' successfully.")
14 else:
15     print(f"File '{DatasetConfig.DATA_PATH}' already exists.")
```

File '/content/drive/MyDrive/Colab Notebooks/CSCI544/HW3/amazon_reviews_us_Office_Products_v1_00.tsv.gz' already exists.
time: 4.21 ms (started: 2023-10-19 22:59:24 +00:00)

▼ Dataset Preparation

This code provides a pipeline for processing and preparing a dataset for sentiment analysis:

1. `LoadData` class loads a dataset from a specified path, keeping only relevant columns.
2. `ProcessData` class performs the following tasks:
 - Converts star ratings to numeric values.
 - Classifies sentiments based on star ratings (1 for negative, 2 for positive).
 - Balances the dataset by sampling an equal number of samples for both sentiments.
3. `CleanText` class defines various text cleaning operations:
 - Removing non-ASCII characters.

- Expanding contractions.
 - Removing email addresses, URLs, and HTML tags.
 - Lowercasing and stripping spaces.
4. `clean_and_process_data` function executes the entire data processing pipeline:
- Loads the data.
 - Applies basic processing.
 - Balances the dataset.
 - Cleans the text.
 - Tokenizes the reviews.
5. `preprocess_review_body` function generates word embeddings for each word in a review using a pre-trained Word2Vec model.
6. `get_reviews_dataset` function handles the entire data preprocessing and embedding generation process. It checks if the preprocessed data already exists, and if not, it performs the data preprocessing and saves the preprocessed data in Parquet format.

Overall, this pipeline ensures that the dataset is properly loaded, cleaned, processed, balanced, and transformed into embeddings suitable for sentiment analysis.

Note:

- Parquet format is efficient for storage.
- Storing data to avoid running the pipeline and embedding generation process all over again.
- Provides a ready-to-use dataset for sentiment analysis tasks, allowing for quicker experimentation and model training

▼ Read and Process

```
1 class LoadData:
2     @staticmethod
3     def load_data(path):
4         df = pd.read_csv(
5             path,
6             sep="\t",
7             usecols=["review_headline", "review_body", "star_rating"],
8             on_bad_lines="skip",
9             memory_map=True,
10        )
11        return df
12
13
14 class ProcessData:
15     @staticmethod
16     def filter_columns(df):
17         return df.loc[:, ["review_body", "star_rating"]]
18
19     @staticmethod
20     def convert_star_rating(df):
21         df["star_rating"] = pd.to_numeric(df["star_rating"], errors="coerce")
22         df.dropna(subset=["star_rating"], inplace=True)
23         return df
24
25     @staticmethod
26     def classify_sentiment(df):
27         df["sentiment"] = df["star_rating"].apply(lambda x: 1 if x <= 3 else 2)
28         return df
29
30     @staticmethod
31     def sample_data(df, n_samples, random_state):
32         sampled_df = pd.concat(
33             [
34                 df.query("sentiment==1").sample(n=n_samples, random_state=random_state),
35                 df.query("sentiment==2").sample(n=n_samples, random_state=random_state),
36             ],
37             ignore_index=True,
38         ).sample(frac=1, random_state=random_state, ignore_index=True)
39
40         sampled_df.drop(columns=["star_rating"], inplace=True)
41         return sampled_df
42
43
44 class CleanText:
45     @staticmethod
46     def unicode_to_ascii(s):
47         return "".join(
48             c for c in unicodedata.normalize("NFD", s) if unicodedata.category(c) != "Mn"
49         )
50
51     @staticmethod
52     def expand_contractions(text):
53         """Expand contraction for eg., wouldn't => would not"""
54         return contractions.fix(text)
55
56     @staticmethod
57     def remove_email_addresses(text):
58         return re.sub(r"[a-zA-Z0-9_\-\.]+\@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,5}", "", text)
59
60     @staticmethod
61     def remove_urls(text):
62         return re.sub(r"\bhttps?:\/\/\S+|www\.\S+", "", text)
63
64     @staticmethod
65     def remove_html_tags(text):
66         return re.sub(r"<.*?>", "", text)
67
68     @staticmethod
69     def clean_text(text):
70         text = text.lower().strip()
71         text = CleanText.unicode_to_ascii(text)
72         # text = CleanText.remove_email_addresses(text)
73         # text = CleanText.remove_urls(text)
74         text = CleanText.remove_html_tags(text)
75         text = CleanText.expand_contractions(text)
76
77         # creating a space between a word and the punctuation following it
78         # text = re.sub(r"([?.!,;])", r" \1 ", text)
79         # text = re.sub(r'[" "]+', " ", text)
80
81         # removes all non-alphabetical characters
82         # text = re.sub(r"^[a-zA-Z\s]+", "", text)
83
84         # remove extra spaces
85         # text = re.sub(" +", " ", text)
```

```
86         return text
87
88
89 def clean_and_process_data(path):
90     df = LoadData.load_data(path)
91
92     # Basic processing
93     df_filtered = ProcessData.filter_columns(df)
94     df_filtered = ProcessData.convert_star_rating(df_filtered)
95     df_filtered = ProcessData.classify_sentiment(df_filtered)
96
97     balanced_df = ProcessData.sample_data(
98         df_filtered, DatasetConfig.N_SAMPLES_EACH_CLASS, DatasetConfig.RANDOM_STATE
99     )
100
101     # Clean data
102     balanced_df.dropna(inplace=True)
103     balanced_df["review_body"] = balanced_df["review_body"].astype(str)
104     balanced_df["review_body"] = balanced_df["review_body"].apply(CleanText.clean_text)
105     # Drop reviews that are empty
106     balanced_df = balanced_df.loc[balanced_df["review_body"].str.strip() != ""]
107
108     # Tokenize Reviews
109     balanced_df["review_body"] = balanced_df["review_body"].apply(word_tokenize)
110     return balanced_df
111
112
113 def preprocess_review_body(text, word2vec_model, topn=None):
114     embeddings = [word2vec_model[word] for word in text if word in word2vec_model]
115
116     if topn is not None:
117         embeddings = np.concatenate(embeddings[:topn], axis=0)
118     else:
119         embeddings = np.mean(embeddings, axis=0)
120     return embeddings
121
122
123 def get_reviews_dataset(new=False):
124     if new or not os.path.exists(DatasetConfig.DATA_PATH):
125         balanced_df = clean_and_process_data(DatasetConfig.DATA_PATH)
126         balanced_df.to_parquet(DatasetConfig.PROCESSED_DATA_PATH, index=False)
127
128         # Preprocess data and generate word2vec embeddings Avg and top 10
129         balanced_df["embeddings"] = balanced_df["review_body"].apply(
130             lambda text: preprocess_review_body(text, pretrained_model, topn=None)
131         )
132         # Drop rows with NaN embeddings
133         balanced_df.dropna(subset=["embeddings"], inplace=True)
134
135         balanced_df["embeddings_top_10"] = balanced_df["review_body"].apply(
136             lambda text: preprocess_review_body(text, pretrained_model, topn=10)
137         )
138
139         balanced_df.to_parquet(DatasetConfig.PREPROCESSED_DATA_PATH, index=False)
140     else:
141         balanced_df = pd.read_parquet(
142             DatasetConfig.PREPROCESSED_DATA_PATH,
143             # engine="fastparquet"
144         )
145     return balanced_df
146
147
148 time: 4.81 ms (started: 2023-10-19 22:59:25 +00:00)
```

```
1 balanced_df = get_reviews_dataset(
2     new=DatasetConfig.BUILD_NEW
3 )
4 print("Total Records:", balanced_df.shape)
5 balanced_df.head(10)
```

Total Records: (99862, 4)

		review_body	sentiment	embeddings	embeddings_top_10
0	[i, set, up, a, photo, booth, at, my, sister, ...	2	[0.016994974, 0.024544675, -0.010975713, 0.093...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...	
1	[like, everyone, else, ,, i, like, saving, mon...	1	[0.044110615, 0.036876563, 0.0371785, 0.113560...	[0.103515625, 0.13769531, -0.0029754639, 0.181...	
2	[the, pen, is, perfect, what, i, want, !, howe...	2	[0.026102701, 0.029064532, 0.010800962, 0.0622...	[0.080078125, 0.10498047, 0.049804688, 0.05346...	
3	[i, think, they, are, too, expensive, for, the...	1	[-0.0039075767, 0.032967318, 0.02339106, 0.113...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...	
4	[black, is, working, wonderfully, ,, and, both...	1	[0.034285888, 0.013478661, 0.041618653, 0.1132...	[0.10498047, 0.018432617, 0.008972168, -0.0128...	
5	[i, have, problems, with, the, moveable, tab, ...	1	[0.010405041, 0.026173819, 0.03433373, 0.09698...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...	
6	[this, printer, sucks, !, it, started, out, wo...	1	[0.05581854, 0.035414256, 0.047512088, 0.09278...	[0.109375, 0.140625, -0.03173828, 0.16601562, ...	
7	[the, ink, on, these, cartridges, leak, ,, i, ...	1	[0.0037488434, 0.053543895, 0.038638465, 0.134...	[0.080078125, 0.10498047, 0.049804688, 0.05346...	
8	[it, gets, points, for, working, as, designed,...	2	[0.046220347, 0.029853666, 0.058699824, 0.0745...	[0.084472656, -0.0003528595, 0.053222656, 0.09...	
9	[i, ordered, these, and, they, work, just, fin...	1	[-0.0013514927, 0.016482098, 0.031290326, 0.07...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...	

time: 29.1 s (started: 2023-10-19 22:59:25 +00:00)

▼ Review Body stats

Mean number of words = 66

Median number of words = 37

Limiting sequence length for RNN based embeddings = 45

```
1 balanced_df["review_body"].apply(len).describe()
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
219
```

```
5 stratify=balanced_df["sentiment"]
6 )

time: 50.7 ms (started: 2023-10-19 23:00:43 +00:00)

Word Embedding

Semantic similarity examples with pretrained embeddings

1 # Example 1: King - Man + Woman = Queen
2 result = pretrained_model.most_similar(positive=['woman', 'king'], negative=['man'])
3 print(f"Semantic Similarity: {result[0][0]}")
4
5 # Example 2: excellent ~ outstanding
6 result = pretrained_model.similarity('excellent', 'outstanding')
7 print(f"Semantic Similarity: {result}")
8
9 # Example 3: Paris - France + Italy = Milan
10 result = pretrained_model.most_similar(positive=['Italy', 'Paris'], negative=['France'])
11 print(f"Semantic Similarity: {result[0][0]}")
12
13 # Example 4: Car - Wheel + Boat = Yacht
14 result = pretrained_model.most_similar(positive=['Boat', 'Car'], negative=['Wheel'])
15 print(f"Semantic Similarity: {result[0][0]}")
16
17 # Example 5: Delicious ~ Tasty
18 result = pretrained_model.similarity('Delicious', 'Tasty')
19 print(f"Semantic Similarity: {result}")
20
21 # Example 6: Computer ~ Plant
22 result = pretrained_model.similarity('Computer', 'Plant')
23 print(f"Semantic Similarity: {result}")
24
25 # Example 7: Cat ~ Dog
26 result = pretrained_model.similarity('Cat', 'Dog')
27 print(f"Semantic Similarity: {result}")

Semantic Similarity: queen
Semantic Similarity: 0.5567485690116882
Semantic Similarity: Milan
Semantic Similarity: Yacht
Semantic Similarity: 0.5718502402305603
Semantic Similarity: 0.04445184767246246
Semantic Similarity: 0.6061107516288757
time: 9.78 s (started: 2023-10-18 21:26:10 +00:00)

1 del pretrained_model

time: 776 µs (started: 2023-10-19 11:13:25 +00:00)
```

Custom Word2Vec Embeddings Generation

```
1 sentences=train_df["review_body"].apply(lambda x: x.tolist()).tolist()
2
3 # Train Word2Vec model
4 w2v_model_custom = Word2Vec(
5     sentences=sentences,
6     vector_size=Word2VecConfig.MAX_LENGTH,
7     window=Word2VecConfig.WINDOW_SIZE,
8     min_count=Word2VecConfig.MIN_WORD_COUNT,
9     workers=multiprocessing.cpu_count()
10 )
11
12 # Save the model
13 w2v_model_custom.save(Word2VecConfig.CUSTOM_MODEL_PATH)

time: 1min 30s (started: 2023-10-18 21:36:09 +00:00)
```

Test Custom Embeddings

```
1 # Load the custom model
2 w2v_model_custom = Word2Vec.load(Word2VecConfig.CUSTOM_MODEL_PATH)
3
4 # Example 1: King - Man + Woman = Queen
5 res = w2v_model_custom.wv.most_similar(positive=['woman', 'king'], negative=['man'])
6 print(f"Semantic Similarity (Custom Model): {res[0]}")
7
8 # Example 2: excellent ~ outstanding
9 res = w2v_model_custom.wv.similarity('excellent', 'outstanding')
10 print(f"Semantic Similarity (Custom Model): {res}")

Semantic Similarity (Custom Model): ('queen', 0.5723455548286438)
Semantic Similarity (Custom Model): 0.7957370281219482
time: 241 ms (started: 2023-10-18 21:37:47 +00:00)
```

Conclusion

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

1. Custom-trained Word2Vec Model:

- Strengths:
 - Captures domain-specific relationships and nuances as it trained on very specific dataset.
- Weaknesses:
 - It may not perform as well on tasks outside of its training domain.
 - The quality of embeddings heavily depends on the dataset used for training.
 - For example, if the dataset is small or not representative of the overall language, the embeddings may be less reliable.

2. Pretrained "word2vec-google-news-300" Model:

- Strengths:
 - This model has been pretrained on a massive corpus of text from various domains, making it highly versatile and capable of capturing a wide range of semantic relationships.
 - It can generalize well to different tasks and domains.

◦ **Weaknesses:**

- While it provides strong generalization, it may not capture domain-specific relationships as effectively as a model trained on domain-specific data.

- The semantic similarity score is higher for the pretrained model compared to the custom model. This indicates that the pretrained model is better at encoding semantic similarities between words.
- The custom Word2Vec model, which was trained on the provided dataset, may not have had access to as diverse and extensive a corpus as the pretrained model. This can lead to limitations in its ability to generalize and capture nuanced semantic relationships.

```
1 del w2v_model_custom, res, sentences

time: 319 µs (started: 2023-10-19 01:33:10 +00:00)
```

▼ Simple Models

```
1 def evaluate_model(model, X_test, y_test):
2     # Predict on the test set
3     y_pred = model.predict(X_test)
4
5     # Calculate evaluation metrics
6     precision = precision_score(y_test, y_pred, average="binary")
7     recall = recall_score(y_test, y_pred, average="binary")
8     f1 = f1_score(y_test, y_pred, average="binary")
9     accuracy = accuracy_score(y_test, y_pred)
10
11     return precision, recall, f1, accuracy
12
13
14 def train_and_evaluate_model(model_class, X_train, y_train, X_test, y_test, **model_params):
15     # Initialize model
16     model = model_class(**model_params)
17
18     # Train the model
19     model.fit(X_train, y_train)
20
21     # Evaluate model
22     precision, recall, f1, accuracy = evaluate_model(model, X_test, y_test)
23     return model, precision, recall, f1, accuracy

time: 1.08 ms (started: 2023-10-19 23:00:53 +00:00)
```

```
1 X_train = np.vstack(train_df["embeddings"])
2 y_train = train_df["sentiment"]
3 X_test = np.vstack(test_df["embeddings"])
4 y_test = test_df["sentiment"]

time: 270 ms (started: 2023-10-19 23:00:54 +00:00)
```

▼ SVM

Params	Precision	Recall	F1	Accuracy	Features Used
LinearSVC(C=0.1,max_iter=10000)	0.7997	0.8671	0.8320	0.8321	Word2Vec
LinearSVC(max_iter=10000)	0.8045	0.8623	0.8324	0.8262	Word2Vec
LinearSVC(C=0.01,max_iter=15000)	0.7836	0.8835	0.8305	0.8281	Word2Vec

```
1 # Train and evaluate LinearSVC model
2 (
3     _,
4     precision_svc,
5     recall_svc,
6     f1_svc,
7     acc_svc
8 ) = train_and_evaluate_model(
9     LinearSVC,
10    X_train, y_train, X_test, y_test,
11    max_iter=10000,
12    # C=0.1
13 )
14
15 print(f'Precision Recall F1 Accuracy (LinearSVC): {precision_svc:.4f} {recall_svc:.4f} {f1_svc:.4f} {acc_svc:.4f}')
```

Precision Recall F1 Accuracy (LinearSVC): 0.8045 0.8623 0.8324 0.8262

time: 28.2 s (started: 2023-10-19 01:31:55 +00:00)

▼ Perceptron

Params	Precision	Recall	F1	Accuracy	Features Used
Perceptron(eta0=0.01,max_iter=5000,penalty='elasticnet',warm_start=True)	0.7693	0.8778	0.8200	0.8071	Word2Vec
Perceptron(max_iter=5000)	0.7786	0.8613	0.8179	0.8110	Word2Vec
Perceptron()	0.7786	0.8613	0.8179	0.8110	Word2Vec
Perceptron(eta0=0.1,max_iter=5000,penalty='elasticnet',warm_start=True)	0.5977	0.9844	0.7438	0.6655	Word2Vec
Perceptron(eta0=0.001,max_iter=10000,penalty='l2')	0.7367	0.9114	0.8148	0.7849	Word2Vec
Perceptron(eta0=0.01,max_iter=10000,penalty='l2',warm_start=True)	0.7653	0.8789	0.8181	0.8002	Word2Vec
Perceptron(eta0=0.01,penalty='l1',warm_start=True)	0.6133	0.9813	0.7548	0.6832	Word2Vec

```
1 # Train and evaluate Perceptron model using Bow features
2 (
3     _,
4     precision_perceptron,
5     recall_perceptron,
6     f1_perceptron,
7     acc_perceptron
8 ) = train_and_evaluate_model(
9     Perceptron,
10    X_train, y_train, X_test, y_test,
11    max_iter=5000,
12    eta0=0.01,
13    warm_start=True,
14    penalty="elasticnet"
15 )
16
17 print(f'Precision Recall F1 (Perceptron): {precision_perceptron:.4f} {recall_perceptron:.4f} {f1_perceptron:.4f} {acc_perceptron:.4f}')
```

Precision Recall F1 (Perceptron): 0.7693 0.8778 0.8200 0.8071

time: 1.1 s (started: 2023-10-19 01:32:24 +00:00)

- ▼ With TFIDF Features

▼ Homework 1 Script Edited

```

1 # @title Homework 1 Script Edited
2
3 %writefile HW1-CSCI544-wo-neg-sw.py
4 # Python Version: 3.10.12
5
6 import re
7 import unicodedata
8
9 import warnings
10
11 warnings.filterwarnings("ignore")
12
13 import numpy as np
14 import pandas as pd
15
16 import nltk
17 from nltk.corpus import stopwords, wordnet
18 from nltk.stem import WordNetLemmatizer
19 from nltk.tokenize import word_tokenize
20
21 nltk.download("punkt", quiet=True)
22 nltk.download("wordnet", quiet=True)
23 nltk.download("stopwords", quiet=True)
24 nltk.download("averaged_perceptron_tagger", quiet=True)
25
26 import contractions
27
28 from sklearn.model_selection import train_test_split
29 from sklearn.feature_extraction.text import TfidfVectorizer
30 from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
31
32 from sklearn.linear_model import Perceptron
33 from sklearn.svm import LinearSVC
34
35
36 class Config:
37     RANDOM_STATE = 56
38     DATA_PATH = "amazon_reviews_us_Office_Products_v1_00.tsv.gz"
39     TEST_SPLIT = 0.2
40     N_SAMPLES_EACH_CLASS = 50000
41     NUM_TFIDF_FEATURES = 5000
42     NUM_BOW_FEATURES = 5000
43
44
45 class DataLoader:
46     @staticmethod
47     def load_data(path):
48         df = pd.read_csv(
49             path,
50             sep="\t",
51             usecols=["review_headline", "review_body", "star_rating"],
52             on_bad_lines="skip",
53             memory_map=True,
54         )
55         return df
56
57
58 class DataProcessor:
59     @staticmethod
60     def filter_columns(df):
61         return df.loc[:, ["review_body", "star_rating"]]
62
63     @staticmethod
64     def convert_star_rating(df):
65         df["star_rating"] = pd.to_numeric(df["star_rating"], errors="coerce")
66         df.dropna(subset=["star_rating"], inplace=True)
67         return df
68
69     @staticmethod
70     def classify_sentiment(df):
71         df["sentiment"] = df["star_rating"].apply(lambda x: 1 if x <= 3 else 2)
72         return df
73
74     @staticmethod
75     def sample_data(df, n_samples, random_state):
76         sampled_df = pd.concat(
77             [
78                 df.query("sentiment==1").sample(n=n_samples, random_state=random_state),
79                 df.query("sentiment==2").sample(n=n_samples, random_state=random_state),
80             ],
81             ignore_index=True,
82         ).sample(frac=1, random_state=random_state)
83
84         sampled_df.drop(columns=["star_rating"], inplace=True)
85         return sampled_df
86
87
88 class TextCleaner:
89     @staticmethod
90     def unicode_to_ascii(s):
91         return "".join(
92             c for c in unicodedata.normalize("NFD", s) if unicodedata.category(c) != "Mn"
93         )
94
95     @staticmethod
96     def expand_contractions(text):
97         return contractions.fix(text)
98
99     @staticmethod
100     def remove_email_addresses(text):
101         return re.sub(r"[a-zA-Z0-9_\-\.]+\@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,5}", " ", text)
102
103     @staticmethod
104     def remove_urls(text):
105         return re.sub(r"\bhttps?:\/\/\S+|www\.\S+", " ", text)
106
107     @staticmethod
108     def remove_html_tags(text):
109         return re.sub(r"<.*?>", "", text)
110
111     @staticmethod
112     def clean_text(text):

```


```

113     text = TextCleaner.unicode_to_ascii(text.lower().strip())
114     # replacing email addresses with empty string
115     text = TextCleaner.remove_email_addresses(text)
116     # replacing urls with empty string
117     text = TextCleaner.remove_urls(text)
118     # Remove HTML tags
119     text = TextCleaner.remove_html_tags(text)
120     # Expand contraction for eg., wouldn't => would not
121     text = TextCleaner.expand_contractions(text)
122     # creating a space between a word and the punctuation following it
123     text = re.sub(r"([?.!,;])", r" \1 ", text)
124     text = re.sub(r'[" "]+' , " ", text)
125     # removes all non-alphabetical characters
126     text = re.sub(r"[^a-zA-Z\s]+", "", text)
127     # remove extra spaces
128     text = re.sub(" +", " ", text)
129     text = text.strip()
130     return text
131
132
133 class TextPreprocessor:
134     lemmatizer = WordNetLemmatizer()
135
136     @staticmethod
137     def get_stopwords_pattern():
138         # Stopword list
139         og_stopwords = set(stopwords.words("english"))
140
141         # Define a list of negative words to remove
142         neg_words = ["no", "not", "nor", "neither", "none", "never", "nobody", "nowhere"]
143         custom_stopwords = [word for word in og_stopwords if word not in neg_words]
144         pattern = re.compile(r"\b(" + r"|".join(custom_stopwords) + r")\b\s*")
145         return pattern
146
147     @staticmethod
148     def pos_tagger(tag):
149         if tag.startswith("J"):
150             return wordnet.ADJ
151         elif tag.startswith("V"):
152             return wordnet.VERB
153         elif tag.startswith("N"):
154             return wordnet.NOUN
155         elif tag.startswith("R"):
156             return wordnet.ADV
157         else:
158             return None
159
160     @staticmethod
161     def lemmatize_text_using_pos_tags(text):
162         words = nltk.pos_tag(word_tokenize(text))
163         words = map(lambda x: (x[0], TextPreprocessor.pos_tagger(x[1])), words)
164         lemmatized_words = [
165             TextPreprocessor.lemmatizer.lemmatize(word, tag) if tag else word for word, tag in words
166         ]
167         return " ".join(lemmatized_words)
168
169     @staticmethod
170     def lemmatize_text(text):
171         words = word_tokenize(text)
172         lemmatized_words = [TextPreprocessor.lemmatizer.lemmatize(word) for word in words]
173         return " ".join(lemmatized_words)
174
175     pattern = get_stopwords_pattern()
176
177     @staticmethod
178     def preprocess_text(text):
179         # replacing all the stopwords
180         text = TextPreprocessor.pattern.sub("", text)
181         text = TextPreprocessor.lemmatize_text(text)
182         return text
183
184
185 clean_text_vect = np.vectorize(TextCleaner.clean_text)
186 preprocess_text_vect = np.vectorize(TextPreprocessor.preprocess_text)
187
188
189 def clean_and_process_data(path):
190     df = DataLoader.load_data(path)
191     df_filtered = DataProcessor.filter_columns(df)
192     df_filtered = DataProcessor.convert_star_rating(df_filtered)
193     df_filtered = DataProcessor.classify_sentiment(df_filtered)
194
195     balanced_df = DataProcessor.sample_data(
196         df_filtered, Config.N_SAMPLES_EACH_CLASS, Config.RANDOM_STATE
197     )
198
199     balanced_df["review_body"] = balanced_df["review_body"].astype(str)
200
201     # Clean data
202     # avg_len_before_clean = balanced_df["review_body"].apply(len).mean()
203     balanced_df["review_body"] = balanced_df["review_body"].apply(clean_text_vect)
204     # Drop reviews that are empty
205     balanced_df = balanced_df.loc[balanced_df["review_body"].str.strip() != ""]
206     # avg_len_after_clean = balanced_df["review_body"].apply(len).mean()
207
208     # Preprocess data
209     # avg_len_before_preprocess = avg_len_after_clean
210     balanced_df["review_body"] = balanced_df["review_body"].apply(preprocess_text_vect)
211     # avg_len_after_preprocess = balanced_df["review_body"].apply(len).mean()
212
213     # Print Results
214     # print(f"{avg_len_before_clean:.2f}, {avg_len_after_clean:.2f}")
215     # print(f"{avg_len_before_preprocess:.2f}, {avg_len_after_preprocess:.2f}")
216
217     return balanced_df
218
219
220 def evaluate_model(model, X_test, y_test):
221     # Predict on the test set
222     y_pred = model.predict(X_test)
223
224     # Calculate evaluation metrics
225     precision = precision_score(y_test, y_pred, average="binary")
226     recall = recall_score(y_test, y_pred, average="binary")
227     f1 = f1_score(y_test, y_pred, average="binary")
228     accuracy = accuracy_score(y_test, y_pred)
229
230     return precision, recall, f1, accuracy
231
232

```



```
233 def train_and_evaluate_model(model_class, X_train, y_train, X_test, y_test, **model_params):
234     # Initialize model
235     model = model_class(**model_params)
236
237     # Train the model
238     model.fit(X_train, y_train)
239
240     # Evaluate model
241     precision, recall, f1, accuracy = evaluate_model(model, X_test, y_test)
242     return model, precision, recall, f1, accuracy
243
244
245 def main():
246     balanced_df = clean_and_process_data(Config.DATA_PATH)
247
248     # Splitting the reviews dataset
249     X_train, X_test, y_train, y_test = train_test_split(
250         balanced_df["review_body"],
251         balanced_df["sentiment"],
252         test_size=Config.TEST_SPLIT,
253         random_state=Config.RANDOM_STATE,
254     )
255
256     # Feature Extraction
257     tfidf_vectorizer = TfidfVectorizer(max_features=Config.NUM_TFIDF_FEATURES)
258     X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
259     X_test_tfidf = tfidf_vectorizer.transform(X_test)
260
261     # Train and evaluate Perceptron model using TF-IDF features
262     (
263         _,
264         precision_perceptron_tfidf,
265         recall_perceptron_tfidf,
266         f1_perceptron_tfidf,
267         acc_perceptron_tfidf
268     ) = train_and_evaluate_model(
269         Perceptron, X_train_tfidf, y_train, X_test_tfidf, y_test, max_iter=4000
270     )
271
272     # Train and evaluate SVM model using TF-IDF features
273     (
274         _,
275         precision_svm_tfidf,
276         recall_svm_tfidf,
277         f1_svm_tfidf,
278         acc_svm_tfidf
279     ) = train_and_evaluate_model(
280         LinearSVC, X_train_tfidf, y_train, X_test_tfidf, y_test, max_iter=2500
281     )
282
283     # Print the results
284     print("Precision Recall F1-Score Accuracy")
285     print("Perceptron")
286     print(
287         f"{precision_perceptron_tfidf:.4f} {recall_perceptron_tfidf:.4f} {f1_perceptron_tfidf:.4f} {acc_perceptron_tfidf:.4f}"
288     )
289
290     print("SVM: LinearSVC")
291     print(f"{precision_svm_tfidf:.4f} {recall_svm_tfidf:.4f} {f1_svm_tfidf:.4f} {acc_svm_tfidf:.4f}")
292
293
294 if __name__ == "__main__":
295     main()
296
```



Overwriting HW1-CSCI544-wo-neg-sw.py
time: 50.9 ms (started: 2023-10-19 22:02:47 +00:00)

```
1 !python HW1-CSCI544-wo-neg-sw.py

Precision Recall F1-Score Accuracy
Perceptron
0.7637 0.8702 0.8135 0.7998
SVM: LinearSVC
0.8573 0.8602 0.8588 0.8581
time: 4min 1s (started: 2023-10-19 22:02:52 +00:00)
```

▼ Conclusion

Best Accuracies

Model	Accuracy	Features Used
Perceptron	0.8110	Word2Vec
LinearSVC	0.8321	Word2Vec
Perceptron	0.7998	TF-IDF
LinearSVC	0.8581	TF-IDF

- LinearSVC outperforms Perceptron for both feature types (Word2Vec and TF-IDF).
 - LinearSVC is better suited for this classification task compared to Perceptron.
- When using Word2Vec features, both Perceptron and LinearSVC achieve lower accuracy compared to when using TF-IDF features.
 - Word2Vec embeddings might not be as effective for this specific sentiment classification task as compared to TF-IDF vectors.
- The LinearSVC model performs particularly well with TF-IDF features, achieving an accuracy of 85.81%.
 - TF-IDF vectors are highly effective in capturing important information for sentiment classification in this dataset.

Overall, based on the provided performance metrics, it seems that TF-IDF features are more effective for this sentiment classification task compared to the Word2Vec embeddings. However, it's important to note that the effectiveness of features can vary depending on the specific dataset and task.

```
1 del balanced_df
2 del X_train, y_train, X_test, y_test

time: 557 μs (started: 2023-10-19 23:01:03 +00:00)
```

▼ Create Pytorch Dataset

- Custom pytorch dataset for on-the-fly processing an d efficient resource utilization
- Each sample in this dataset includes embeddings and their corresponding target label. The label is adjusted by subtracting 1 from the label value in the DataFrame

- Using DataLoader's
 - Used to load and manage batches of data during the training process.
 - Handle tasks like shuffling, batching, and parallel data loading, making it easier to feed data to the model.

```
1 class AmazonReviewsSentimentDataset(Dataset):
2     def __init__(self, df, embeddings_col_name, label_col_name):
3         self.data = df
4         self.embeddings_col_name = embeddings_col_name
5         self.label_col_name = label_col_name
6
7     def __len__(self):
8         return len(self.data)
9
10    def __getitem__(self, idx):
11        if idx >= self.__len__():
12            raise IndexError
13
14        label = self.data.iloc[idx][self.label_col_name] - 1
15        embeddings = self.data.iloc[idx][self.embeddings_col_name]
16
17        return {
18            "embeddings": torch.tensor(embeddings, dtype=torch.float32),
19            "target": torch.tensor(label, dtype=torch.long)
20        }
```

time: 1.19 ms (started: 2023-10-19 23:01:06 +00:00)

```
1 train_dataset = AmazonReviewsSentimentDataset(
2     train_df, embeddings_col_name="embeddings", label_col_name="sentiment"
3 )
4 valid_dataset = AmazonReviewsSentimentDataset(
5     test_df, embeddings_col_name="embeddings", label_col_name="sentiment"
6 )
```

time: 702 µs (started: 2023-10-19 23:01:08 +00:00)

▼ Loaders & Samplers

```
1 TRAIN_BATCH_SIZE = 128
2 VALID_BATCH_SIZE = 64
3 NUM_PARALLEL_WORKERS = multiprocessing.cpu_count()
4 EPOCHS = 10
```

time: 769 µs (started: 2023-10-19 23:01:12 +00:00)

```
1 # train_sampler = RandomSampler(train_dataset)
2
3 train_data_loader = DataLoader(
4     train_dataset,
5     batch_size=TRAIN_BATCH_SIZE,
6     # sampler=train_sampler,
7     drop_last=True,
8     shuffle=True,
9     # num_workers=NUM_PARALLEL_WORKERS
10 )
11
12 # valid_sampler = RandomSampler(valid_dataset)
13
14 valid_data_loader = DataLoader(
15     valid_dataset,
16     batch_size=VALID_BATCH_SIZE,
17     # sampler=valid_sampler,
18     drop_last=False,
19     shuffle=False,
20     # num_workers=NUM_PARALLEL_WORKERS
21 )
22
23 test_data_loader = DataLoader(
24     valid_dataset,
25     batch_size=valid_dataset.data.__len__(),
26     # sampler=valid_sampler,
27     drop_last=False,
28     shuffle=False,
29     # num_workers=NUM_PARALLEL_WORKERS
30 )
```

time: 7.5 ms (started: 2023-10-19 18:41:36 +00:00)

▼ Training & Evaluation Functions

- compute_accuracy calculates the accuracy of model predictions given true labels.
- train_loop_fn handles one training epoch, updating the model's weights based on computed gradients.
- eval_loop_fn handles one validation epoch, computing the model's performance on the validation set.
- train_and_evaluate orchestrates the training process, saving checkpoints if specified. It reports metrics after each epoch. If a final model path is provided, it saves the model at the end.

```
1 def compute_accuracy(outputs, labels):
2     predicted = torch.argmax(outputs.data, dim=1)
3
4     predicted = predicted.detach().cpu().numpy()
5     labels = labels.detach().cpu().numpy()
6
7     acc = accuracy_score(labels, predicted)
8     return acc
9
10 def train_loop_fn(data_loader, model, optimizer, loss_fn, device):
11     model.train()
12     train_loss = 0.0
13     acc = []
14
15     for batch in tqdm(data_loader):
16         embeddings = batch['embeddings'].to(device, dtype=torch.float32, non_blocking=True)
17         labels = batch['target'].to(device, dtype=torch.long, non_blocking=True)
18
19         optimizer.zero_grad()
20
21         outputs = model(embeddings.float())
22         loss = loss_fn(outputs, labels)
23
```

```

24     loss.backward()
25     optimizer.step()
26
27     train_loss += loss.item()*len(labels)
28     acc.append(compute_accuracy(outputs, labels))
29
30     acc = sum(acc)/len(acc)
31     return train_loss, acc
32
33 def eval_loop_fn(data_loader, model, device):
34     valid_loss = 0.0
35     acc = []
36     model.eval()
37
38     for batch in data_loader:
39         embeddings = batch['embeddings'].to(device, dtype=torch.float32, non_blocking=True)
40         labels = batch['target'].to(device, dtype=torch.long, non_blocking=True)
41
42         outputs = model(embeddings.float())
43
44         loss = criterion(outputs, labels)
45         valid_loss += loss.item()*len(labels)
46
47         acc.append(compute_accuracy(outputs, labels))
48
49     acc = sum(acc)/len(acc)
50
51     return valid_loss, acc
52
53
54 def train_and_evaluate(
55     model,
56     train_data_loader, valid_data_loader,
57     optimizer, loss_fn,
58     device,
59     num_epochs,
60     checkpoint=False,
61     path="model.pt"
62 ):
63     if checkpoint:
64         dirname = path.split(".")[0]
65         checkpoint_path = os.path.join(dirname)
66         if os.path.exists(checkpoint_path):
67             shutil.rmtree(checkpoint_path)
68         os.makedirs(dirname)
69
70     for epoch in range(num_epochs):
71         # Train Step
72         train_loss, train_acc = train_loop_fn(
73             train_data_loader, model, optimizer, loss_fn, device
74         )
75
76         # Validation Step
77         valid_loss, valid_acc = eval_loop_fn(valid_data_loader, model, device)
78
79         train_loss /= len(train_data_loader.dataset)
80         valid_loss /= len(valid_data_loader.dataset)
81
82         if checkpoint:
83             cp = os.path.join(checkpoint_path, f"{path}_{epoch}.pt")
84             torch.save(model.state_dict(), cp)
85             print(f"Saved Checkpoint to '{cp}'")
86
87         epoch_log = (
88             f"Epoch {epoch+1}/{num_epochs}, "
89             f" Train Accuracy={train_acc:.4f}, Validation Accuracy={valid_acc:.4f}, "
90             f" Train Loss={train_loss:.4f}, Validation Loss={valid_loss:.4f}"
91         )
92         print(epoch_log)
93
94     torch.save(model.state_dict(), path)
95     return model

```

time: 2.94 ms (started: 2023-10-19 23:01:15 +00:00)

▼ Feedforward Neural Networks

```

1 class MLP(nn.Module):
2     def __init__(self, num_input_features, num_classes):
3         super(MLP, self).__init__()
4         # Input size is 300 (Word2Vec dimensions)
5         self.fc1 = nn.Linear(num_input_features, 50)
6         self.fc2 = nn.Linear(50, 5)
7         # Output size is 2 for binary classification
8         self.fc3 = nn.Linear(5, num_classes)
9
10    def forward(self, x):
11        x = torch.relu(self.fc1(x))
12        x = torch.relu(self.fc2(x))
13        x = self.fc3(x)
14        return x
15
16
17 net = MLP(num_input_features=Word2VecConfig.MAX_LENGTH, num_classes=2)
18 criterion = nn.CrossEntropyLoss()
19 optimizer = optim.SGD(net.parameters(), lr=0.01)
20 net = net.to(device)

```

time: 4.36 ms (started: 2023-10-19 10:33:35 +00:00)

▼ With average Word2Vec features

```

1 model = train_and_evaluate(
2     model=net,
3     train_data_loader=train_data_loader,
4     valid_data_loader=valid_data_loader,
5     optimizer=optimizer,
6     loss_fn=criterion,
7     device=device,
8     num_epochs=20,
9     checkpoint=True,
10    path="mlp_w_avg_w2v_feat.pt"
11 )

```

100% 624/624 [00:21<00:00, 35.15it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_0.pt'
Epoch 1/20, Train Accuracy=0.7113, Validation Accuracy=0.7318, Train Loss=0.6679, Validation Loss=0.6583

100% 624/624 [00:21<00:00, 20.21it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_1.pt'
Epoch 2/20, Train Accuracy=0.7298, Validation Accuracy=0.7375, Train Loss=0.6438, Validation Loss=0.6256

100% 624/624 [00:19<00:00, 35.05it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_2.pt'
Epoch 3/20, Train Accuracy=0.7470, Validation Accuracy=0.7608, Train Loss=0.6036, Validation Loss=0.5779

100% 624/624 [00:19<00:00, 34.62it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_3.pt'
Epoch 4/20, Train Accuracy=0.7637, Validation Accuracy=0.7773, Train Loss=0.5553, Validation Loss=0.5302

100% 624/624 [00:21<00:00, 13.51it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_4.pt'
Epoch 5/20, Train Accuracy=0.7766, Validation Accuracy=0.7893, Train Loss=0.5137, Validation Loss=0.4946

100% 624/624 [00:20<00:00, 33.96it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_5.pt'
Epoch 6/20, Train Accuracy=0.7862, Validation Accuracy=0.7958, Train Loss=0.4848, Validation Loss=0.4710

100% 624/624 [00:21<00:00, 34.93it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_6.pt'
Epoch 7/20, Train Accuracy=0.7932, Validation Accuracy=0.7976, Train Loss=0.4662, Validation Loss=0.4568

100% 624/624 [00:27<00:00, 34.54it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_7.pt'
Epoch 8/20, Train Accuracy=0.7987, Validation Accuracy=0.8046, Train Loss=0.4536, Validation Loss=0.4458

100% 624/624 [00:35<00:00, 28.22it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_8.pt'
Epoch 9/20, Train Accuracy=0.8035, Validation Accuracy=0.8087, Train Loss=0.4446, Validation Loss=0.4378

100% 624/624 [00:23<00:00, 15.18it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_9.pt'
Epoch 10/20, Train Accuracy=0.8066, Validation Accuracy=0.8103, Train Loss=0.4374, Validation Loss=0.4322

100% 624/624 [00:29<00:00, 7.02it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_10.pt'
Epoch 11/20, Train Accuracy=0.8097, Validation Accuracy=0.8128, Train Loss=0.4314, Validation Loss=0.4266

100% 624/624 [00:21<00:00, 22.66it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_11.pt'
Epoch 12/20, Train Accuracy=0.8116, Validation Accuracy=0.8178, Train Loss=0.4263, Validation Loss=0.4221

100% 624/624 [00:25<00:00, 13.12it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_12.pt'
Epoch 13/20, Train Accuracy=0.8137, Validation Accuracy=0.8198, Train Loss=0.4220, Validation Loss=0.4186

100% 624/624 [00:32<00:00, 34.89it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_13.pt'
Epoch 14/20, Train Accuracy=0.8153, Validation Accuracy=0.8206, Train Loss=0.4178, Validation Loss=0.4147

100% 624/624 [00:19<00:00, 33.72it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_14.pt'
Epoch 15/20, Train Accuracy=0.8171, Validation Accuracy=0.8211, Train Loss=0.4143, Validation Loss=0.4115

100% 624/624 [00:21<00:00, 30.96it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_15.pt'
Epoch 16/20, Train Accuracy=0.8182, Validation Accuracy=0.8241, Train Loss=0.4112, Validation Loss=0.4102

100% 624/624 [00:21<00:00, 30.42it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_16.pt'
Epoch 17/20, Train Accuracy=0.8190, Validation Accuracy=0.8240, Train Loss=0.4083, Validation Loss=0.4066

100% 624/624 [00:21<00:00, 21.98it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_17.pt'
Epoch 18/20, Train Accuracy=0.8191, Validation Accuracy=0.8245, Train Loss=0.4058, Validation Loss=0.4044

100% 624/624 [00:19<00:00, 33.59it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_18.pt'
Epoch 19/20, Train Accuracy=0.8220, Validation Accuracy=0.8261, Train Loss=0.4035, Validation Loss=0.4025

100% 624/624 [00:19<00:00, 33.92it/s]
Saved Checkpoint to 'mlp_w_avg_w2v_feat/mlp_w_avg_w2v_feat.pt_19.pt'
Epoch 20/20, Train Accuracy=0.8228, Validation Accuracy=0.8267, Train Loss=0.4014, Validation Loss=0.4009
time: 9min 29s (started: 2023-10-19 07:40:53 +00:00)

Overall Accuracy on Test Set

```
1 path_to_saved_model = 'mlp_w_avg_w2v_feat.pt'
2 model = MLP(num_input_features=Word2VecConfig.MAX_LENGTH, num_classes=2)
3 model.load_state_dict(torch.load(path_to_saved_model, map_location=device))
4
5 for batch in test_data_loader:
6     embeddings = batch['embeddings'].to(device, dtype=torch.float32, non_blocking=True)
7     y_pred = model(embeddings)
8     y_true = batch["target"].to(device, dtype=torch.long, non_blocking=True)
9
10 acc = compute_accuracy(y_pred, y_true)
11 print("Accuracy (Test Dataset):", round(acc,4))
```

Accuracy (Test Dataset): 0.8262
time: 8.79 s (started: 2023-10-19 10:33:58 +00:00)

▼ With top 10 Word2Vec features

- Embeddings are padded for maintaining consistent input dimensions across different samples in a batch.

```
1 class ARDatasetWithTop10Embeddings(Dataset):
2     def __init__(self, df, embeddings_col_name, label_col_name, max_length):
3         self.data = df
4         self.embeddings_col_name = embeddings_col_name
5         self.label_col_name = label_col_name
6         self.max_length = max_length
7
8     def __len__(self):
9         return len(self.data)
10
11     def __getitem__(self, idx):
12         if idx >= self.__len__():
13             raise IndexError
14
15         label = self.data.iloc[idx][self.label_col_name] - 1
16         embeddings = self.data.iloc[idx][self.embeddings_col_name]
17
18         # Pad embeddings to max_length
19         if len(embeddings) < self.max_length:
20             padding = np.zeros(self.max_length - len(embeddings))
21             embeddings = np.concatenate((embeddings, padding))
22
```

```
23     return {
24         "embeddings": torch.tensor(embeddings, dtype=torch.float32),
25         "target": torch.tensor(label, dtype=torch.long)
26     }
27
28 train_dataset = ARDatasetWithTop10Embeddings(
29     train_df, embeddings_col_name="embeddings_top_10", label_col_name="sentiment", max_length=3000
30 )
31 valid_dataset = ARDatasetWithTop10Embeddings(
32     test_df, embeddings_col_name="embeddings_top_10", label_col_name="sentiment", max_length=3000
33 )
34
35 train_data_loader = DataLoader(
36     train_dataset,
37     batch_size=TRAIN_BATCH_SIZE,
38     drop_last=True,
39     shuffle=True,
40 )
41
42 valid_data_loader = DataLoader(
43     valid_dataset,
44     batch_size=VALID_BATCH_SIZE,
45     drop_last=False,
46     shuffle=False,
47 )
48
49 test_data_loader = DataLoader(
50     valid_dataset,
51     batch_size=valid_dataset.__len__(),
52     drop_last=False,
53     shuffle=False,
54 )
```

time: 12 ms (started: 2023-10-19 10:36:08 +00:00)

```
1 net2 = MLP(num_input_features=3000, num_classes=2)
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.SGD(net2.parameters(), lr=0.01)
4 net2 = net2.to(device)
5
6
7 model2 = train_and_evaluate(
8     model=net2,
9     train_data_loader=train_data_loader,
10    valid_data_loader=valid_data_loader,
11    optimizer=optimizer,
12    loss_fn=criterion,
13    device=device,
14    num_epochs=20,
15    checkpoint=True,
16    path="mlp_w_top10_w2v_feat.pt"
17 )
```

100%

624/624 [00:30<00:00, 25.53it/s]

Saved Checkpoint to 'mlp_w_top10_w2v_feat/mlp_w_top10_w2v_feat.pt_0.pt'

Epoch 1/20, Train Accuracy=0.4960, Validation Accuracy=0.4998, Train Loss=0.6930, Validation Loss=0.6932

100%

624/624 [00:31<00:00, 17.36it/s]

Saved Checkpoint to 'mlp w_top10_w2v_feat/mlp w_top10_w2v_feat.pt 1.pt'

Overall Accracy on Test Set

```
1 path_to_saved_model = 'mlp_w_top10_w2v_feat.pt'
2 model = MLP(num_input_features=3000, num_classes=2)
3 model.load_state_dict(torch.load(path_to_saved_model, map_location=device))
4
5 for batch in test_data_loader:
6     embeddings = batch['embeddings'].to(device, dtype=torch.float32, non_blocking=True)
7     y_pred = model(embeddings)
8     y_true = batch["target"].to(device, dtype=torch.long, non_blocking=True)
9
10 acc = compute_accuracy(y_pred, y_true)
11 print("Accuracy (Test Dataset):", round(acc,4))
```

Accuracy (Test Dataset): 0.7589

time: 5.48 s (started: 2023-10-19 10:36:17 +00:00)

Epoch 7/20. Train Accuracv=0.5746. Validation Accuracv=0.5004. Train Loss=0.6922. Validation Loss=0.6919

Comparision with Simple Model

The LinearSVC model trained on TF-IDF features was the most effective in this scenario, outperforming both simple models and MLP models trained with Word2Vec embeddings.

Conclusion

1. Feature Importance:

- The choice of features significantly impacts model performance.
- In this case, TF-IDF features proved to be the most informative for sentiment analysis, as evidenced by the high accuracy achieved by LinearSVC with TF-IDF.

2. Complexity vs. Performance:

- Simple models like Perceptron and LinearSVC can sometimes outperform more complex models.
- This is evident in the case where LinearSVC with TF-IDF outperformed the MLP models.

3. Embedding Selection:

- Not all embeddings are equally effective. The choice of Word2Vec embeddings, particularly using the average vectors, yielded competitive results, showcasing the importance of using quality word embeddings.

4. Dimensionality Matters:

- Using only the top 10 Word2Vec embeddings didn't capture enough information for sentiment analysis.
- It's important to consider the dimensionality of the embeddings and how well they represent the underlying semantics.

▼ Recurrent Neural Networks

```
Saved Checkpoint to 'map_rnn_top10_rnn_feat/map_rnn_top10_rnn_feat.pt_0.pt'

1 class ARDatasetFullEmb(Dataset):
2     def __init__(self, df, embeddings_col_name, label_col_name, max_length):
3         self.data = df
4         self.embeddings_col_name = embeddings_col_name
5         self.label_col_name = label_col_name
6         self.max_length = max_length
7
8     def __len__(self):
9         return len(self.data)
10
11     def __getitem__(self, idx):
12         if idx >= self.__len__():
13             raise IndexError
14
15         label = self.data.iloc[idx][self.label_col_name] - 1
16         embeddings = self.data.iloc[idx][self.embeddings_col_name]
17
18         # Pad embeddings to max_length
19         if len(embeddings) < self.max_length:
20             padding = np.zeros(self.max_length - len(embeddings))
21             embeddings = np.concatenate((embeddings, padding))
22
23         embeddings = embeddings.reshape(10, 300)
24
25         return {
26             "embeddings": torch.tensor(embeddings, dtype=torch.float32),
27             "target": torch.tensor(label, dtype=torch.long)
28         }
29
30 train_dataset = ARDatasetFullEmb(
31     train_df, embeddings_col_name="embeddings_top_10", label_col_name="sentiment", max_length=3000,
32 )
33 valid_dataset = ARDatasetFullEmb(
34     test_df, embeddings_col_name="embeddings_top_10", label_col_name="sentiment", max_length=3000,
35 )
36
37 train_data_loader = DataLoader(
38     train_dataset,
39     batch_size=TRAIN_BATCH_SIZE,
40     drop_last=True,
41     shuffle=True,
42 )
43
44 valid_data_loader = DataLoader(
45     valid_dataset,
46     batch_size=VALID_BATCH_SIZE,
47     drop_last=False,
48     shuffle=False,
49 )
50
51 test_data_loader = DataLoader(
52     valid_dataset,
53     batch_size=valid_dataset.__len__(),
54     drop_last=False,
55     shuffle=False,
56 )

time: 2.38 ms (started: 2023-10-19 23:51:17 +00:00)
```

```
1 def compute_accuracy(outputs, labels):
2     predicted = torch.argmax(outputs.data, dim=1)
3
4     predicted = predicted.detach().cpu().numpy()
5     labels = labels.detach().cpu().numpy()
6
7     acc = accuracy_score(labels, predicted)
8     return acc
9
10 def train_loop_fn(data_loader, model, optimizer, loss_fn, device):
11     model.train()
12     train_loss = 0.0
13     acc = []
14
15     for batch in tqdm(data_loader):
16         optimizer.zero_grad()
17
18         embeddings = batch['embeddings'].detach()
19         labels = batch['target'].detach()
20
21         all_emb = torch.stack(embeddings).to(device, dtype=torch.float32, non_blocking=True)
22         all_lb = torch.stack(labels).to(device, dtype=torch.long, non_blocking=True)
23
24         outputs, _ = model(all_emb.float())
25         loss = loss_fn(outputs, all_lb)
26
27         loss.backward()
28         optimizer.step()
29
30         train_loss += loss.item()*len(all_lb)
31         acc.append(compute_accuracy(outputs, all_lb))
32
33     acc = sum(acc)/len(acc)
34     return train_loss, acc
35
36 def eval_loop_fn(data_loader, model, device):
37     valid_loss = 0.0
38     acc = []
39     model.eval()
40
41     for batch in data_loader:
42         embeddings = batch['embeddings'].detach()
43         labels = batch['target'].detach()
44
45         all_emb = torch.stack(embeddings).to(device, dtype=torch.float32, non_blocking=True)
46         all_lb = torch.stack(labels).to(device, dtype=torch.long, non_blocking=True)
47
48         outputs, _ = model(all_emb.float())
49
50         loss = criterion(outputs, all_lb)
51         valid_loss += loss.item()*len(all_lb)
52
53         acc.append(compute_accuracy(outputs, all_lb))
54
55     acc = sum(acc)/len(acc)
56
57     return valid_loss, acc
58
59
60 def train_and_evaluate(
61     model,
62     train_data_loader, valid_data_loader,
63     optimizer, loss_fn,
64     device,
65     num_epochs,
66     checkpoint=False,
67     path="model.pt"
68 ):
69     if checkpoint:
70         dirname = path.split(".")[0]
71         checkpoint_path = os.path.join(dirname)
72         if os.path.exists(checkpoint_path):
73             shutil.rmtree(checkpoint_path)
74         os.makedirs(dirname)
75
76     for epoch in range(num_epochs):
77         # Train Step
78         train_loss, train_acc = train_loop_fn(
79             train_data_loader, model, optimizer, loss_fn, device
80         )
81
82         # Validation Step
83         valid_loss, valid_acc = eval_loop_fn(valid_data_loader, model, device)
84
85         train_loss /= len(train_data_loader.dataset)
86         valid_loss /= len(valid_data_loader.dataset)
87
88         if checkpoint:
89             cp = os.path.join(checkpoint_path, f"{path}_{epoch}.pt")
90             torch.save(model.state_dict(), cp)
91             print(f"Saved Checkpoint to '{cp}'")
92
93         epoch_log = (
94             f"Epoch {epoch+1}/{num_epochs},"
95             f" Train Accuracy={train_acc:.4f}, Validation Accuracy={valid_acc:.4f},"
96             f" Train Loss={train_loss:.4f}, Validation Loss={valid_loss:.4f}"
97         )
98         print(epoch_log)
99
100     torch.save(model.state_dict(), path)
101     return model
```

time: 3.38 ms (started: 2023-10-19 23:56:33 +00:00)

▼ Simple RNN

```
1 class RNNModel(nn.Module):
2     def __init__(
3         self, input_size, hidden_size, num_layers, output_size, model_type="rnn"
4     ):
5         super(RNNModel, self).__init__()
6
7         self.hidden_size = hidden_size
8         self.num_layers = num_layers
9         self.model_type = model_type
10
```

```
11     if model_type == "gru":
12         self.layer = nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
13     elif model_type == "lstm":
14         self.layer = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
15     else:
16         self.layer = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
17
18     #dropout layer
19     self.dropout = nn.Dropout(0.3)
20
21     # Fully connected layers
22     self.fc = nn.Linear(hidden_size, output_size)
23
24     def forward(self, x):
25         batch_size = x.size(0)
26         hidden = self.init_hidden(batch_size)
27
28         out, hidden = self.layer(x, hidden)
29         # Stack up the model output
30         out = out.contiguous().view(-1, self.hidden_size)
31
32         out = self.dropout(out)
33         # Only use the output from the last time step
34         out = self.fc(out)
35         return out, hidden
36
37     def init_hidden(self, batch_size):
38         hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)
39         return hidden
```

time: 1.53 ms (started: 2023-10-19 23:56:41 +00:00)

```
1 input_size = 300
2 hidden_size = 10
3 output_size = 2
4
5 net3 = RNNModel(input_size, hidden_size, 10, output_size, model_type="rnn").to(device)
6 criterion = nn.CrossEntropyLoss()
7 optimizer = torch.optim.Adam(net3.parameters(), lr=0.01)
8
9 model3 = train_and_evaluate(
10     model=net3,
11     train_data_loader=train_data_loader,
12     valid_data_loader=valid_data_loader,
13     optimizer=optimizer,
14     loss_fn=criterion,
15     device=device,
16     num_epochs=25,
17     checkpoint=True,
18     path="simple_rnn_w2v_feat.pt"
19 )
```

0% 0/624 [00:00<?, ?it/s]

TypeError Traceback (most recent call last)

[<ipython-input-65-6b1bcdac76e>](#) in <cell line: 9>()
7 optimizer = torch.optim.Adam(net3.parameters(), lr=0.01)
8

----> 9 model3 = train_and_evaluate(
10 model=net3,
11 train_data_loader=train_data_loader,

⬆ 1 frames

[<ipython-input-63-a604df9676f1>](#) in train_loop_fn(data_loader, model, optimizer, loss_fn, device)
19 labels = batch['target'].detach()
20
----> 21 all_emb = torch.stack(embeddings).to(device, dtype=torch.float32, non_blocking=True)
22 all_lb = torch.stack(labels).to(device, dtype=torch.long, non_blocking=True)
23

TypeError: stack(): argument 'tensors' (position 1) must be tuple of Tensors, not Tensor

SEARCH STACK OVERFLOW

time: 134 ms (started: 2023-10-19 23:56:42 +00:00)

▼ GRU

```
1 input_size = 300
2 hidden_size = 10
3 output_size = 2
4
5 net4 = RNNModel(input_size, hidden_size, 10, output_size, model_type="gru").to(device)
6 criterion = nn.CrossEntropyLoss()
7 optimizer = torch.optim.Adam(net4.parameters(), lr=0.01)
8
9 model4 = train_and_evaluate(
10     model=net4,
11     train_data_loader=train_data_loader,
12     valid_data_loader=valid_data_loader,
13     optimizer=optimizer,
14     loss_fn=criterion,
15     device=device,
16     num_epochs=25,
17     checkpoint=True,
18     path="gru_w2v_feat.pt"
19 )
```

▼ LSTM

```
1 input_size = 300
2 hidden_size = 10
3 output_size = 2
4
5 net5 = RNNModel(input_size, hidden_size, 10, output_size, model_type="lstm").to(device)
6 criterion = nn.CrossEntropyLoss()
7 optimizer = torch.optim.Adam(net5.parameters(), lr=0.01)
8
9 model5 = train_and_evaluate(
10     model=net5,
11     train_data_loader=train_data_loader,
12     valid_data_loader=valid_data_loader,
13     optimizer=optimizer,
14     loss_fn=criterion,
15     device=device,
```



```
16 num_epochs=25,
17 checkpoint=True,
18 path="lstm_w2v_feat.pt"
19 )
```

Conclusion

- 1. **Feature Representations:**
 - TF-IDF outperforms Word2Vec across all models.
 - Averaged Word2Vec is better than Concatenated Word2Vec.
- 2. **Model Comparisons:**
 - SVM outperforms Perceptron consistently.
 - MLP with averaged Word2Vec performs better than RNN, GRU, and LSTM with Word2Vec.
- 3. **Recurrent Models:**
 - RNN, GRU, and LSTM show similar performance with Word2Vec embeddings.
- 4. **Overall Performance:**
 - Highest accuracy (~87%) is achieved with SVM using TF-IDF.

- Other:
- Averaging Word2Vec embeddings seems a more effective representation
 - SVM model is better at capturing the non-linear relationships in the data compared to the Perceptron
 - TF-IDF may capture important information more effectively than Word2Vec embeddings

References

1. <https://www.kaggle.com/code/abhishek/bert-multi-lingual-tpu-training-8-cores>
2. <https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist>
3. <https://www.kaggle.com/code/arunmohan003/sentiment-analysis-using-lstm-pytorch>
4. <https://pytorch.org/docs/stable/generated/torch.nn.RNN.html>
5. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html
6. My own HW1 python file submission
7. <https://piazza.com/class/llm91seaknw3j6/post/408>

THE END