Amazon Food Review Analysis and Classification

Group Work

The pretrained GloVe models used to produce meaningful embeddings were obtained from https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/). (Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation.)

TABLE OF CONTENTS

1. Data Exploration, Visualization And Cleaning
    - 1.1. Get Data
    - 1.2. Checking for Null Values
    - 1.3. Inspecting Text
    - 1.4. Data Cleaning
    - 1.5. Feature Engineering
    - 1.6. Correlations
    - 1.7. Viewing Data Distribution
    - 1.8. Reducing Skewness
    - 1.9. Standardization
2. Text Processing and Normalization
    - 2.1. Stop Word Removal
    - 2.2. Stemming
    - 2.3. Lemmatization
3. Vector Space Model and Feature Representation
    - 3.1. Bag Of Words Representation (Term Frequency)
    - 3.2. Bag Of Words Representation (Term Frequency Inverse Document Frequency)
    - 3.3. Word Embedding Using GloVe Model
4. Model Selection, Training, Hyperparameter Tuning and Evaluation
    - 4.2.1. Linear Regression
    - 4.2.2. Naive Bayes
    - 4.2.3. Support Vector Machines
    - 4.2.4. Stochastic Gradient Descent
    - 4.2.5. Grid Search/Displaying performance measures of all best models
5. Modelling Text as a Sequence
    - 5.1. LSTM Model
    - 5.2. GRU Model
    - 5.3. LSTM Vs GRU
6. Topic Modelling of High and Low Ratings
    - 6.1. Investigating 5 Star Rated Reviews
    - 6.2. Investigating 1 Star Rated Reviews
    - 6.3. Topic Conclusions

References

1. Data Exploration, Visualization And Cleaning

In [9]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
plt.style.use("dark_background")
import seaborn as sns
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\matplotlib_in
ne\config.py:66: DeprecationWarning:

InlineBackend._figure_formats_changed is deprecated in traitlets 4.1: use
@observe and @unobserve instead.
```

1.1. Get Data

In [2]:
```
DF = pd.read_csv("./data/train.csv")
display(DF.head(3))
print(DF.shape)
```

	Score	Review_text
0	5	I received this product early from the seller!...
1	5	*****<br />Numi's Collection Assortment Melang...
2	5	I was very careful not to overcook this pasta,...

```
(426340, 2)
```

1.2. Checking for Null Values

In [3]:
```
DF.isnull().sum()
```

Out[3]:
```
Score          0
Review_text    0
dtype: int64
```

OBSERVATION: No columns in the dataset has null values.

1.3. Inspecting Text

In [4]:
```
with open("./data/review_text.txt", 'w', encoding = 'utf-8') as f:
    f.write(" ".join(DF["Review_text"][:1000]))
```

OBSERVATION: The text is relatively clean apart from html tags. Thus cleaning will involve removing html tags, punctuation, stop words and other special characters if any like brackets etc. Expansion of word contractions like say "that'll" to "that will", will have to be done prior to cleaning.

1.4. Data Cleaning

All of the patterns mentioned below are to be matched and substituted with either "" or " " to remove them. Regex explanation in detail:

`<[^<>]*>` -> The dataset was found to contain several html embeddings, for instance, "<br />". Getting rid of special characters alone would not help this case. Hence, the best solution was to disregard anything that comes in between triangle brackets. This substitution was done before the emoji translation because in the case that the emoji translation was done first, some of the brackets were considered to be part of emoticons when placed with other characters. For instance, in ":<br />", the ":<" would be considered as an emoji which was not desirable.

`emoticons` -> The emoticons were changed into their written form before removing special characters, capital letters and numbers because many emoticons make use of these. The initial strategy was to keep the emoticons intact using regex which would be done using (M3SOulu, 2018). But after further discussion, it was concluded that many emojis would lose their original meaning if the letters being used in caps were converted to lower case, for instance :D -> :d. To combat this, the emoticons were replaced with their textual description using a library called "emot".

`make lower case` -> Make full text lowercase. This was not done before expanding emoticons because many emoticons like ":D" are written using capital letters.

`-` -> Replace "-" with " ".

`expand contractions` -> Expand contractions like "don't" to "do not". To do so, although libraries like pycontractions and contractions exist, they proved to not be very effective. So, dictionary of english contractions to replace with expansions was compiled from https://stackoverflow.com/questions/19790188/expanding-english-language-contractions-in-python (https://stackoverflow.com/questions/19790188/expanding-english-language-contractions-in-python).

`'s` -> Remove "'s" that indicates belonging. This was not done before expanding contractions so as not to remove contractions like "she's".

`[^a-z ]*` -> Substitute out anything that is not a character from a-z or a space. The review is converted to lower case before being passed to this.

`(https?\w*)|(www\w*)` -> Substitute out URLs.

`\s\s+` -> Remove extra spaces.

At the end of regex substitution, if the review contains no letters, then empty string is returned. Later all empty reviews in the DF is dropped.

In [4]:
```python
import re
#!pip install emot
from emot.emo_unicode import EMOTICONS_EMO # for the emoticons
CONTRACTIONS = {
    "ain't": "is not",
    "aren't": "are not",
    "can't": "cannot",
    "can't've": "cannot have",
    "'cause": "because",
    "could've": "could have",
    "couldn't": "could not",
    "couldn't've": "could not have",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "hadn't": "had not",
    "hadn't've": "had not have",
    "hasn't": "has not",
    "haven't": "have not",
    "he'd": "he would",
    "he'd've": "he would have",
    "he'll": "he will",
    "he'll've": "he will have",
    "he's": "he is",
    "how'd": "how did",
    "how'd'y": "how do you",
    "how'll": "how will",
    "how's": "how is",
    "i'd": "i would",
    "i'd've": "i would have",
    "i'll": "i will",
    "i'll've": "i will have",
    "i'm": "i am",
    "i've": "i have",
    "isn't": "is not",
    "it'd": "it would",
    "it'd've": "it would have",
    "it'll": "it will",
    "it'll've": "it will have",
    "it's": "it is",
    "let's": "let us",
    "ma'am": "madam",
    "mayn't": "may not",
    "might've": "might have",
    "mightn't": "might not",
    "mightn't've": "might not have",
    "must've": "must have",
    "mustn't": "must not",
    "mustn't've": "must not have",
    "needn't": "need not",
    "needn't've": "need not have",
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
    "sha'n't": "shall not",
    "shan't've": "shall not have",
    "she'd": "she would",
    "she'd've": "she would have",
    "she'll": "she will",
    "she'll've": "she will have",
    "she's": "she is",
    "should've": "should have",
    "shouldn't": "should not",
    "shouldn't've": "should not have",
    "so've": "so have",
    "so's": "so as",
    "that'd": "that would",
    "that'd've": "that would have",
    "that's": "that is",
    "there'd": "there would",
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    "we'd": "we would",
    "we'd've": "we would have",
    "we'll": "we will",
    "we'll've": "we will have",
    "we're": "we are",
    "we've": "we have",
    "weren't": "were not",
    "what'll": "what will",
    "what'll've": "what will have",
    "what're": "what are",
    "what's": "what is",
    "what've": "what have",
    "when's": "when is",
    "when've": "when have",
    "where'd": "where did",
    "where's": "where is",
    "where've": "where have",
    "who'll": "who will",
    "who'll've": "who will have",
    "who's": "who is",
    "who've": "who have",
    "why's": "why is",
    "why've": "why have",
    "will've": "will have",
    "won't": "will not",
    "won't've": "will not have",
    "would've": "would have",
    "wouldn't": "would not",
    "wouldn't've": "would not have",
    "y'all": "you all",
    "y'all'd": "you all would",
    "y'all'd've": "you all would have",
    "y'all're": "you all are",
    "y'all've": "you all have",
    "you'd": "you would",
    "you'd've": "you would have",
    "you'll": "you will",
    "you'll've": "you will have",
    "you're": "you are",
    "you've": "you have"
}
```

In [5]:

```
def clean(text):
    ''' Function that returns a given piece of text after cleaning it.
    # check for html embeddings and sub them out with a space
    text = re.sub("<[^<>]*>"," ",text)

    # if an emoticon is encountered, replace with word equivalent
    for emote in EMOTICONS_EMO:
        text = text.replace(emote, " "+EMOTICONS_EMO[emote])

    # make lowercase
    text = text.lower().strip()

    # expand contractions
    for key, val in CONTRACTIONS.items():
        text = re.sub(key, val, text)

    # discard 's
    text = re.sub("'s","",text)

    # discard if it is not a-z or space
    text = re.sub('-'," ",text)

    # discard if it is not a-z or space
    text = re.sub('[^a-z ]*',"",text)

    # discard urls
    text = re.sub('(https?\w*)|(www\w*)',"",text)

    # discard extra spaces
    text = re.sub('\s\s+'," ",text)

    # check if final text is empty or not
    if re.search("[a-z]", text): return text
    else: return ""
```

WARNING: Time consuming cell ahead! (upto 3 mins)

```
In [6]:
DF["Review"] = DF["Review_text"].apply(lambda review: clean(review))
DF = DF.drop(["Review_text"], axis=1)
DF = DF[DF["Review"] != ""]
display(DF)
```

	Score	Review
0	5	i received this product early from the seller ...
1	5	numi collection assortment melange includes h...
2	5	i was very careful not to overcook this pasta ...
3	5	buying this multi pack i was misled by the pic...
4	5	these bars are so good i loved them warmed up ...
...	...	...
426335	5	i had been buying at a store but they had a ha...
426336	5	so glad that there are companies that are maki...
426337	4	i love real scottish haggis and this brand tho...
426338	5	we eat a lot of syrup in our house my three ye...
426339	5	i buy this to give to my dog he needs to lose ...

426339 rows × 2 columns

1.5. Feature Engineering

In this section, some additional columns shall be computed and added to the data frame which may be helpful attributes that can be fed into ML models to aid in classifying reviews as per score. 4 new features as listed below shall be added.

1. `Word_Count` -> No. of words in the review.
2. `Avg_Word_Length` -> Average length of words in the review.
3. `Polarity` -> Polarity of the word calculated using Vader library
4. `Longest_Word_Length` -> Length of longest word in the review.

```
In [7]:
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
# Load vader lexicon and sentiment analyzer
nltk.download('vader_lexicon')
sid = SentimentIntensityAnalyzer()
import math
```

```
[nltk_data] Downloading package vader_lexicon to C:\Users\Gayathri
[nltk_data]     Girish Nair\AppData\Roaming\nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
```

WARNING: Time consuming cell ahead! (upto 5 mins)

In [8]:
```
DF["Word_Count"] = DF["Review"].apply(lambda review: len(review.split('
DF["Avg_Word_Length"] = DF["Review"].apply(lambda review: np.mean([len(
DF["Longest_Word_Length"] = DF["Review"].apply(lambda review: max([len(
DF["Polarity"] = DF["Review"].apply(lambda review: sid.polarity_scores(
display(DF)
```

	Score	Review	Word_Count	Avg_Word_Length	Longest_Word_Length	Polarity
0	5	i received this product early from the seller ...	38	4.236842	8	0.9488
1	5	numi collection assortment melange includes h...	316	4.797468	12	0.9921
2	5	i was very careful not to overcook this pasta ...	205	4.307317	14	0.9980
3	5	buying this multi pack i was misled by the pic...	35	4.542857	9	0.4019
4	5	these bars are so good i loved them warmed up ...	29	4.379310	11	0.9644
...	...	...	...	...	...	...
426335	5	i had been buying at a store but they had a ha...	167	4.083832	11	0.9266
426336	5	so glad that there are companies that are maki...	56	4.732143	11	0.9780
426337	4	i love real scottish haggis and this brand tho...	93	3.967742	9	0.8370
426338	5	we eat a lot of syrup in our house my three ye...	115	3.956522	10	0.9814
426339	5	i buy this to give to my dog he needs to lose ...	46	3.434783	7	-0.1269

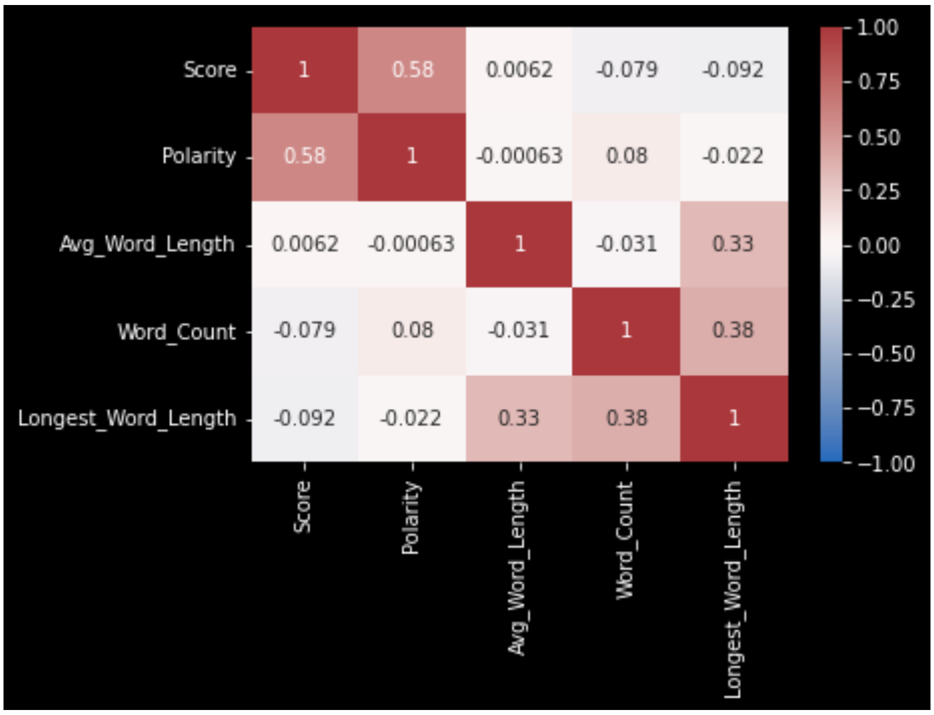
426339 rows × 6 columns

1.6. Correlations

This section aims to explore the correlation between "Score" and the newly created features.

In [10]:
```
sns.heatmap(DF[[
    "Score", "Polarity", "Avg_Word_Length", "Word_Count", "Longest_Word
]].corr(), cmap="vlag", vmin=-1, vmax=1, annot=True)
```

Out[10]:
```
<AxesSubplot:>
```



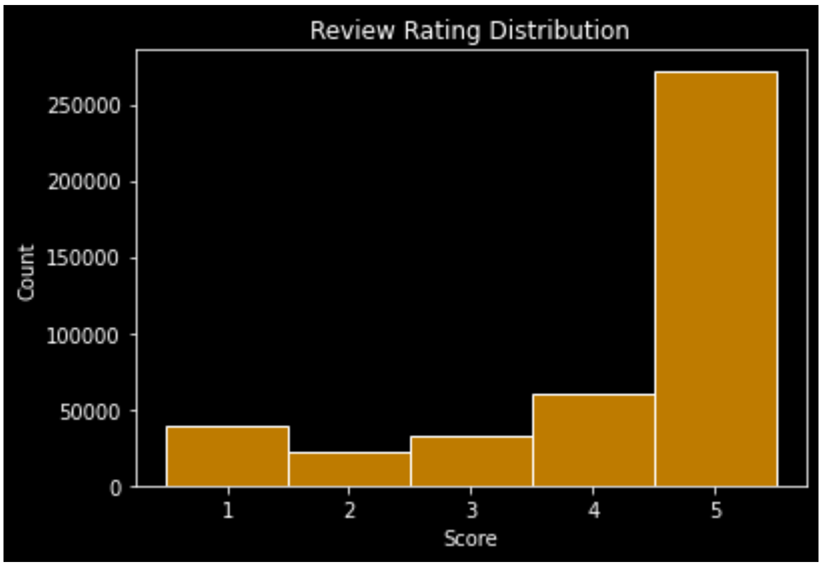
OBSERVATION: The feature polarity seems promising w.r.t. the score but none of the other features have much correlation with the score. Since "Avg_Word_Length" has the least amount of correlation with Score this feature shall be dropped while other features shall be retained as an experiment to see if this will make any difference in the classifier's performance.

In [9]:
```
DF = DF.drop(["Avg_Word_Length"], axis=1)
```

1.7. Viewing Data Distribution

In [12]:
```
sns.histplot(data=DF, x="Score", color="orange", discrete=True)
plt.title("Review Rating Distribution")
```

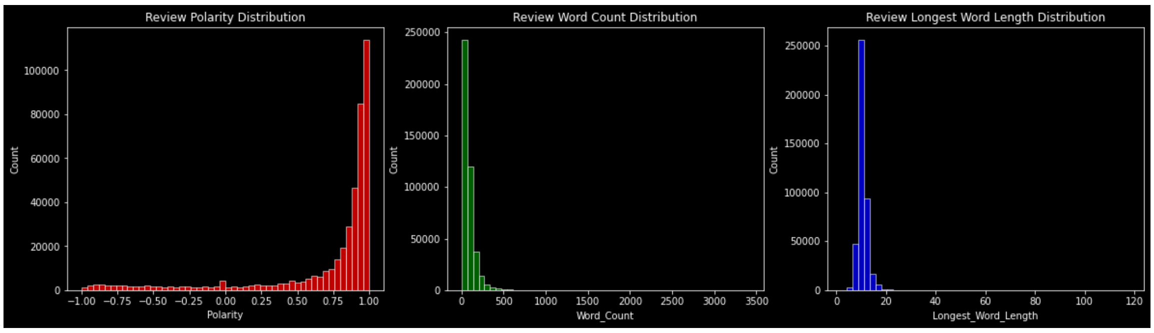
Out[12]:
```
Text(0.5, 1.0, 'Review Rating Distribution')
```



OBSERVATION: It can be observed here that there the distribution of ratings are very uneven with most ratings having score 5. This presents an issue w.r.t. machine learning models which won't fully learn to categorize text in terms of minority classes due to an associated lack in no. of training samples. Thus, upon splitting of data into train and test sets, an resampling techniques may be used to make training data more uniformly distributed.

In [13]:
```
plt.figure(figsize=(20,5))

plt.subplot(131)
sns.histplot(data=DF, x="Polarity", color="red", bins=50)
plt.title("Review Polarity Distribution")

plt.subplot(132)
sns.histplot(data=DF, x="Word_Count", color="green", bins=50)
plt.title("Review Word Count Distribution")

plt.subplot(133)
sns.histplot(data=DF, x="Longest_Word_Length", color="blue", bins=50)
plt.title("Review Longest Word Length Distribution")
```

Out[13]:
```
Text(0.5, 1.0, 'Review Longest Word Length Distribution')
```



OBSERVATION: Distribution of the 3 features "Polarity", "Word_Count" and "Longest_Word_Length" is heavily skewed. Polarity and Longest_Word_Length distributions show left/negative skewness while Word_Count distribution shows right/positive skewness. Skewness in input data can lower performance of many Machine Learning (ML) models. This is usually because the tail portion of a skewed distribution can often have the effect of outliers. This can throw off a model's calculations thereby confusing it. (R. Sharma, 2019) To avoid this, skewness is often handles using statistical techniques.

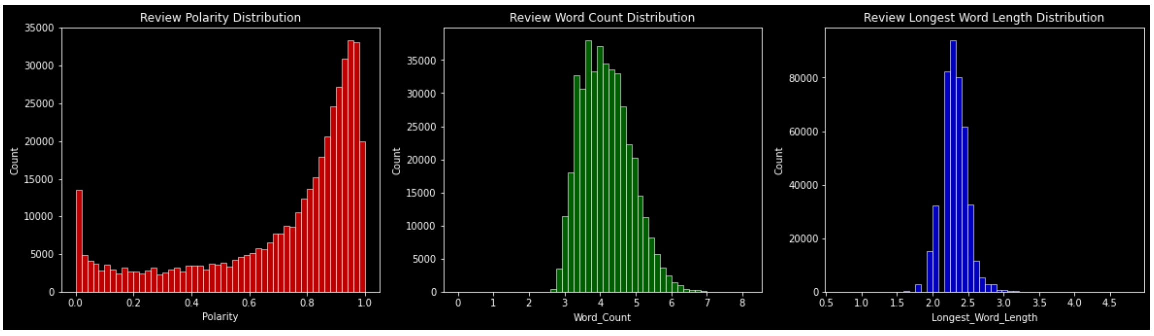
1.8. Reducing Skewness

Left skewness can be reduced by transforming the data using a square function (lambda x: x**2) and right skewness can be reduced by applying a log transformation to the data (lambda x: math.log(x)) (R. Vasudev, 2017). applying transformations that reduce skewness can sometimes reduce correlation, as demonstrated below.

In [11]:
```
DF_reduced_skewness = pd.DataFrame({
    "Score": DF["Score"],
    "Word_Count": DF["Word_Count"].apply(lambda x: math.log(x)),
    "Longest_Word_Length": DF["Longest_Word_Length"].apply(lambda x: ma
    "Polarity": DF["Polarity"].apply(lambda x: x**2)
})
```

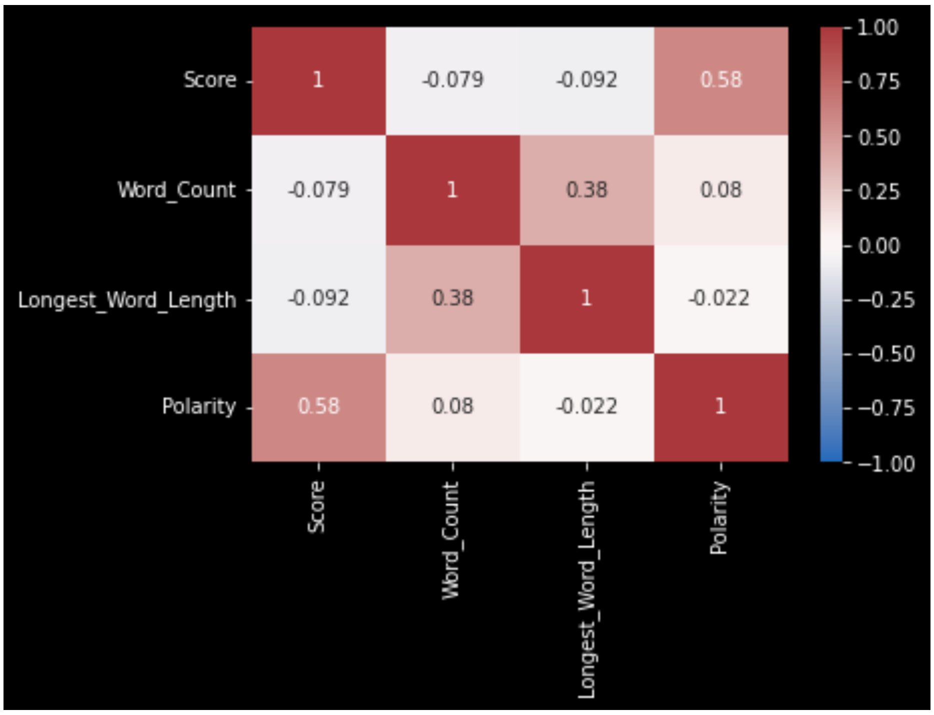
In [12]:
```
plt.figure(figsize=(20,5))

plt.subplot(131)
sns.histplot(data=DF_reduced_skewness, x="Polarity", color="red", bins=
plt.title("Review Polarity Distribution")

plt.subplot(132)
sns.histplot(data=DF_reduced_skewness, x="Word_Count", color="green", b
plt.title("Review Word Count Distribution")

plt.subplot(133)
sns.histplot(data=DF_reduced_skewness, x="Longest_Word_Length", color="
plt.title("Review Longest Word Length Distribution")
```

Out[12]:
```
Text(0.5, 1.0, 'Review Longest Word Length Distribution')
```



```
In [16]:  sns.heatmap(DF.corr(), cmap="vlag", vmin=-1, vmax=1, annot=True)
```

```
Out[16]: <AxesSubplot:>
```



OBSERVATION: It can be seen that the skewness across all 3 distributions has been reduced. However, while the correlation of Score with Word_Count and Longest_Word_Length has increased a little, that with Polarity has reduced as was predicted may happen in the previous observation. Thus, here, it is best to not apply square transformation to Polarity feature values. So, only transformed values of Word_Count and Longest_Word_Length shall be a part of DF.

In [12]:
```
DF["Word_Count"] = DF_reduced_skewness["Word_Count"]
DF["Longest_Word_Length"] = DF_reduced_skewness["Longest_Word_Length"]
display(DF)
```

	Score	Review	Word_Count	Longest_Word_Length	Polarity
0	5	i received this product early from the seller ...	3.637586	2.079442	0.9488
1	5	numi collection assortment melange includes h...	5.755742	2.484907	0.9921
2	5	i was very careful not to overcook this pasta ...	5.323010	2.639057	0.9980
3	5	buying this multi pack i was misled by the pic...	3.555348	2.197225	0.4019
4	5	these bars are so good i loved them warmed up ...	3.367296	2.397895	0.9644
...	...	...	...	...	...
426335	5	i had been buying at a store but they had a ha...	5.117994	2.397895	0.9266
426336	5	so glad that there are companies that are maki...	4.025352	2.397895	0.9780
426337	4	i love real scottish haggis and this brand tho...	4.532599	2.197225	0.8370
426338	5	we eat a lot of syrup in our house my three ye...	4.744932	2.302585	0.9814
426339	5	i buy this to give to my dog he needs to lose ...	3.828641	1.945910	-0.1269

426339 rows × 5 columns

1.9. Standardization

Often, features to be used as input to ML algorithms are normalized/standardized so that they can be compared better mathematically and the difference in units/scales among them do not negatively influence the model's learning. Here values of the features "Word_Count" and "Longest_Word_Length" are very different from that of polarity which is already in the range -1 to 1. If the features follow more of a Gaussian/Normal distribution (which is the case here for "Word_Count" and "Longest_Word_Length" after reducing skewness), then it is usually best practice to standard scale the data before using it as input to an ML model. (Manikanth, 2021).

OBSERVATION: Thus, Standard Scaling shall be applied after splitting the data into train and test sets. The data must first be split because test data is to be transformed using sklearn's standard scaler which must be fit on training data.

2. Text Processing and Normalization

In [8]:
```
from nltk.corpus import stopwords
import nltk
```

2.1. Stop Word Removal

In [1]:
```python
STOP_WORDS = set(stopwords.words('english'))

def remove_stopwords(text): # (M.D. Pietro, 2017)
    ''' Function that removes stopwords from a given list of word
        and returns this new possibly smaller list.
        @param text: The list of words from which to remove stopwords.
    '''
    return " ".join([word for word in text.split(" ") if not word in ST
```

In [2]:
```python
DF["Review_Processed"] = DF["Review"].apply(lambda review: remove_stopw
```

2.2. Stemming

OBSERVATION:

(Jabeen, 2018) (Srinidhi, 2020)

Between all the stemming algorithms available, the ones that retain the majority of the root of words were Porter's stemmer and the Snowball's stemmer in NLTK. The Lancaster stemmer would overstem—i.e. remove an even bigger length of the word ending to result in very short stems. This did not work well with our application as many of the words were reduced to semantically incorrect forms, whereas, it was ideal for us to maintain as much semantics of the words as possible. Thus, the Lancaster stemmer was deemed unfitting of use here.

Between the Porter stemmer and the Snowball stemmer, as stemming in general removes the affixes of words sequentially based on a set of rules rather than trying to find the root of the words by referring to a database or dictionary, it meant that the text of the reviews would after stemming be lexicographically invalid. Since our application deals with sentiment analysis, it was of great importance that words are post text normalization are valid words with their meanings intact as the aim of this project is to be able to assign a label/score to a review based on the sentiment of the review. Thus, lemmatization was the preferred method for normalization.

2.3. Lemmatization

OBSERVATION: To speed up the lemmatization process, some elements from the spaCy's nlp pipeline were disabled. The optimal results came from disabling `ner` and replacing `parser` with `sentencizer`. (TR517, 2018).

In [15]:
```python
import spacy
# spaCy has to be version 3.2.2 to run the following lemmatization code
# check spaCy version by typing spacy.__version.__
#!python -m spacy download en_core_web_sm
nlp = spacy.load('en_core_web_sm', disable=['ner', 'parser'])
nlp.add_pipe('sentencizer')
```

Out[15]:
```
<spacy.pipeline.sentencizer.Sentencizer at 0x7ffc5625c840>
```

OBSERVATION: For best results, the lemmatization was done on the original review prior to cleaning. This is because in the cleaning process the words are converted to lowercase and the lemmatizer treats upper case words as Nouns that can represent people, names, organisation etc and hence would lemmatize them correctly. spaCy, also, has an additional feature built in within the lemmatizer's process that recognises contractions. For formal contractions, i.e. contractions that contain apostrophe between the shortened two words e.g. `couldn't` , after the lemmatization process, that word becomes `could not` . For "informal" contractions, i.e. contracted words without an apostrophe between the abridged words e.g. `couldnt` , it would recognise that it's a contraction and would break the word into two: `could` and `nt` . Thus, lemmatization `couldnt` as `could nt` . To improve this process of expanding the contractions the third rule is added so that `nt` would be lemmatized to `not` .

However, after reviewing the lemmatized reviews, it did not expand all the contractions available within the text. Thus, it was concluded that it would be better to lemmatize the reviews after they were cleaned to ensure that the contractions are expanded correctly and any odd spacing between words can be removed to allow the lemmatizer to better recognise the given words.

In [15]:
```
nlp.get_pipe('attribute_ruler').add([[{"TEXT":"us"}]],{"LEMMA":"us"})
nlp.get_pipe('attribute_ruler').add([[{"TEXT":"them"}]],{"LEMMA":"them"
nlp.get_pipe('attribute_ruler').add([[{"TEXT":"nt"}]],{"LEMMA":"not"})
```

OBSERVATION: spaCy allows us to add rules for how to lemmatize specific words. By default, spaCy lemmatizes `us -> we` and `them -> they` , however, in our case we want these words to not change, hence the addition of these rules are to the maintain the original form of these specific words.

Explicit tokenization is not necessary in spaCy's implementation of lemmatization, therefore, it has not been done. (spaCy, 2022)

In [16]:
```
def lemmatize(text_list):
    doc = nlp(text_list)
    return " ".join([token.lemma_ for token in doc])
```

WARNING: Time consuming cell ahead! (upto 30 mins)

In [19]:
```
DF["Review_Processed"] = DF["Review_Processed"].apply(lambda review: le
```

In [22]:
```
# Save DF
DF.to_csv("data/train_processed.csv", index=False)
```

3. Vector Space Model and Feature Representation

In [6]:
```python
import gensim.downloader as gensim_api
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import BernoulliNB,GaussianNB
from sklearn.preprocessing import label_binarize
from imblearn.under_sampling import NearMiss
import json
```

In [4]:
```python
DF = pd.read_csv("data/train_processed.csv")
DF = DF.drop(["Review"], axis=1)
display(DF.head(3))
```

	Score	Word_Count	Longest_Word_Length	Polarity	Review_Processed
0	5	3.637586	2.079442	0.9488	receive product early seller tastey great mid ...
1	5	5.755742	2.484907	0.9921	numi collection assortment melange include h...
2	5	5.323010	2.639057	0.9980	careful overcook pasta make sure take bite eve...

In [5]:
```python
print("before dropping duplicates",len(DF))
DF = DF.drop_duplicates(subset=['Review_Processed']) # remove duplicate
DF = DF[DF["Review_Processed"]!=""]
DF.dropna(inplace=True)
print("after dropping duplicates",len(DF))
```

```
before dropping duplicates 426339
after dropping duplicates 308445
```

In [6]:
```python
X_train_text, X_test_text, y_train, y_test = train_test_split(
    DF.drop(["Score"], axis=1),
    DF["Score"],
    test_size=0.3,
    random_state=23,
    shuffle=True,
    stratify=DF["Score"]
)
X_train_text = X_train_text.reset_index(drop=True)
X_test_text = X_test_text.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)
y_test = y_test.reset_index(drop=True)
```

In [7]:
```python
# standardizing Word_Count and Longest_Word_Length
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train_text[["Word_Count", "Longest_Word_Length"]])
scaled_train = scaler.transform(X_train_text[["Word_Count", "Longest_Wo
scaled_test = scaler.transform(X_test_text[["Word_Count", "Longest_Word
X_train_text["Word_Count"] = scaled_train[:,0]
X_train_text["Longest_Word_Length"] = scaled_train[:,1]
X_test_text["Word_Count"] = scaled_test[:,0]
X_test_text["Longest_Word_Length"] = scaled_test[:,1]
```

In [8]:
```python
print(X_train_text.shape, y_train.shape)
print(X_test_text.shape, y_test.shape)
```

```
(215911, 4) (215911,)
(92534, 4) (92534,)
```

In [9]:
```python
# (Angelica Lo Duca, 2021)
def strategy(x, y, threshold, t='majority'):
    ''' Function that aids in doing over/under sampling. '''
    targets = ''
    if t == 'majority': targets = y.value_counts() > threshold
    elif t == 'minority': targets = y.value_counts() < threshold
    tc = targets[targets == True].index
    strategy = {}
    for target in tc: strategy[target] = threshold
    return strategy
```

OBSERVATION:

- Two different models have been used that stem form the bag of words model ie; the TFIDF (Term Frequency inverse document frequency) and the term frequency model. Due to the nature of the problem space (text based sentiment analysis), we have opted to use bigrams instead of unigrams. It was so chosen due to phrases such as "not like", "not interested" or more of the sort which expresses a positive sentiment with one word but a negative with another which might cause the final model to miscalculate if take separately.
- The features generated initially (Word_Count, Longest_Word_Length and Polarity) were appended to the feature list geenrated for the TF and TFIDF models and the classifier accept this final dataframe of features.

Resampling

This dataset contained data that was heavily imbalanced. To counter this issue resampling was taken advantage of. Just undersampling the data to the class that contained the least number of samples ended up taking out a lot of useful data and reduced the over number of training samples to around 60,000. Only using oversampling to reach the majority class was a little too heavy on the RAM and increased the number of training samples to around 700,000. The best bet was to combine both. The threshold was set to 30,000 and all the classes were forced to reach that threshold whether through oversampling or undersampling. This amounted to a training set of 150,000 samples. Although not ideal (loss of a lot of the samples in the majority class), due to the major difference in the number of samples per class, this seemed to be the most sensible way to get around the feat.

3.1. Bag Of Words Representation (Term Frequency)

In [34]:
```python
count_vect = CountVectorizer(ngram_range = (2, 2), max_features=1000)
count_vect.fit(X_train_text['Review_Processed'])
X_train_tfvec = count_vect.transform(X_train_text['Review_Processed'])

print(f"Text Vectors: {repr(X_train_tfvec)}")
print(f"\nno. of unique words = {len(count_vect.vocabulary_)}")
print(f"\nvocabulary = {list(count_vect.vocabulary_.items())[:10]} ..."
```

```
Text Vectors: <215911x1000 sparse matrix of type '<class 'numpy.int64'>'
	with 925055 stored elements in Compressed Sparse Row format>

no. of unique words = 1000

vocabulary = [('make good', 546), ('right amount', 735), ('melt mouth', 56
2), ('happy face', 409), ('face smiley', 218), ('good chocolate', 340), ('
think great', 858), ('great idea', 389), ('take time', 811), ('time get',
866)] ...
```

In [35]:
```python
TF_TRAIN_DF = pd.DataFrame(
    data = X_train_tfvec.toarray(),
    columns = count_vect.get_feature_names_out(),
)

display(TF_TRAIN_DF.head(3))
print("TF matrix shape = {}".format(TF_TRAIN_DF.shape))
```

	able find	able get	absolutely delicious	absolutely love	actually taste	add flavor	add little	add sugar	add water	almond butter	...	would think
0	0	0	0	0	0	0	0	0	0	0	...	0
1	0	0	0	0	0	0	0	0	0	0	...	0
2	0	0	0	0	0	0	0	0	0	0	...	0

3 rows × 1000 columns

```
TF matrix shape = (215911, 1000)
```

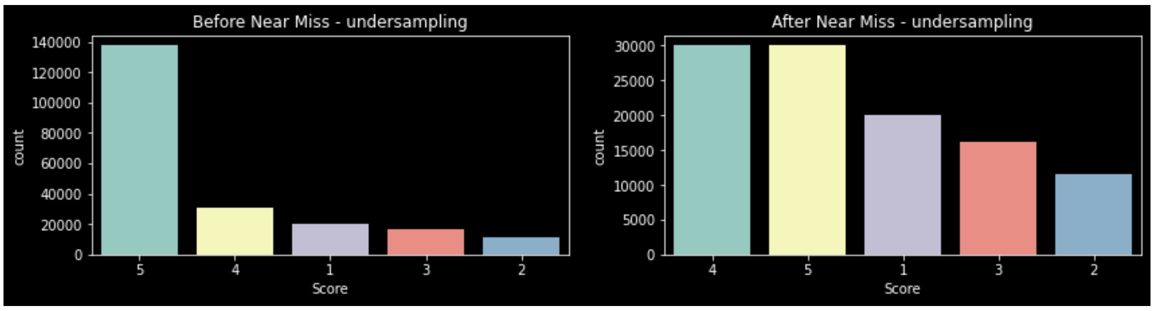
In [36]:
```python
X_test_tfvec = count_vect.transform(X_test_text['Review_Processed'])

TF_TEST_DF = pd.DataFrame (
    data = X_test_tfvec.toarray(),
    columns = count_vect.get_feature_names_out(),
)

print((TF_TEST_DF.columns == TF_TRAIN_DF.columns).all())
print(X_test_tfvec.shape)
```

```
True
(92534, 1000)
```

In [37]:
```python
X_train_tfbow = TF_TRAIN_DF
X_test_tfbow = TF_TEST_DF
y_train_tfbow = y_train
y_test_tfbow = y_test
```

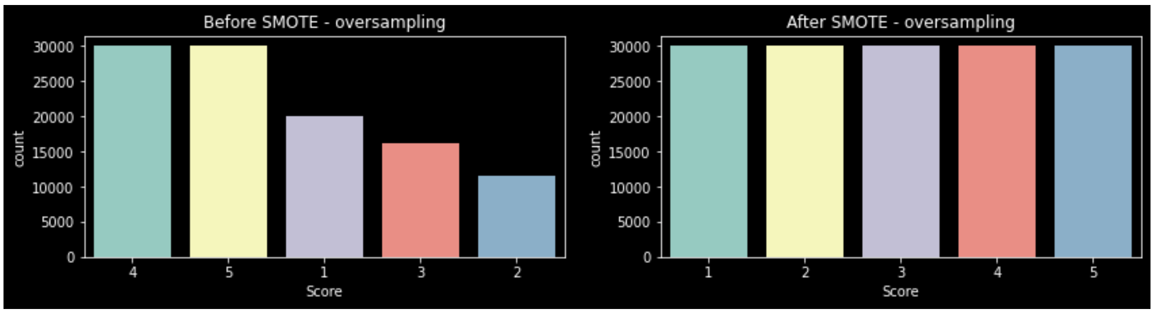
In [38]:
```
# UNDERSAMPLING
undersample_tf = NearMiss(
    version=1, n_neighbors=3,
    sampling_strategy=strategy(X_train_tfbow,y_train_tfbow,30000,t='maj
)

plt.figure(figsize=(14,3))
plt.subplot(121)
plt.title("Before Near Miss - undersampling")
sns.countplot(x=y_train_tfbow, order=y_train_tfbow.value_counts().index
plt.subplot(122)
X_train_tfbow, y_train_tfbow = undersample_tf.fit_resample(X_train_tfbo
plt.title("After Near Miss - undersampling")
sns.countplot(x=y_train_tfbow, order=y_train_tfbow.value_counts().index
```

Out[38]:
```
<AxesSubplot:title={'center':'After Near Miss - undersampling'}, xlabel=
core', ylabel='count'>
```



In [39]:
```
# OVERSAMPLING
oversample_tf = SMOTE(random_state=30,sampling_strategy=strategy(X_trai

plt.figure(figsize=(14,3))
plt.subplot(121)
plt.title("Before SMOTE - oversampling")
sns.countplot(x=y_train_tfbow, order=y_train_tfbow.value_counts().index
plt.subplot(122)
X_train_tfbow, y_train_tfbow = oversample_tf.fit_resample(X_train_tfbow
plt.title("After SMOTE - oversampling")
sns.countplot(x=y_train_tfbow, order=y_train_tfbow.value_counts().index
```

Out[39]:
```
<AxesSubplot:title={'center':'After SMOTE - oversampling'}, xlabel='Scor
', ylabel='count'>
```



In [40]:
```
X_train_tfbow = X_train_tfbow.to_numpy()
y_train_tfbow = y_train_tfbow.to_numpy()
X_test_tfbow = X_test_tfbow.to_numpy()
y_test_tfbow = y_test_tfbow.to_numpy()
```

WARNING: Time consuming cell ahead! (upto 2 mins)

In [41]:
```
# save BOW (TF) representation of documents as json
with open('./data/train_tf_bow.json', 'w') as outfile:
    json.dump({
        "X_train": X_train_tfbow.tolist(),
        "X_test": X_test_tfbow.tolist(),
        "y_train": y_train_tfbow.tolist(),
        "y_test": y_test_tfbow.tolist()
    }, outfile)
```

3.2. Bag Of Words Representation (Term Frequency Inverse Document Frequency)

In [9]:
```
tfidf_vectorizer = TfidfVectorizer(ngram_range=(2,2), max_features=1000
X_train_tfidfvec = tfidf_vectorizer.fit_transform(X_train_text['Review_

print(len(X_train_text))
print(f"Text Vectors: {repr(X_train_tfidfvec)}")
print(f"\nno. of unique words = {len(tfidf_vectorizer.vocabulary_)}")
print(f"\nvocabulary = {list(tfidf_vectorizer.vocabulary_.items())[:10]
```

```
215911
Text Vectors: <215911x1000 sparse matrix of type '<class 'numpy.float64'>'
        with 925055 stored elements in Compressed Sparse Row format>

no. of unique words = 1000

vocabulary = [('make good', 546), ('right amount', 735), ('melt mouth', 56
2), ('happy face', 409), ('face smiley', 218), ('good chocolate', 340), ('
think great', 858), ('great idea', 389), ('take time', 811), ('time get',
866)] ...
```

In [10]:
```
# storing TF-IDF matrix as a dataframe
TFIDF_TRAIN_DF = pd.DataFrame(
    data = X_train_tfidfvec.toarray(),
    columns = tfidf_vectorizer.get_feature_names_out(),
)
display(TFIDF_TRAIN_DF.head(3))
print("TFIDF matrix shape = {}".format(TFIDF_TRAIN_DF.shape))
```

	able find	able get	absolutely delicious	absolutely love	actually taste	add flavor	add little	add sugar	add water	almond butter	...	would think
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0

3 rows × 1000 columns

```
TFIDF matrix shape = (215911, 1000)
```

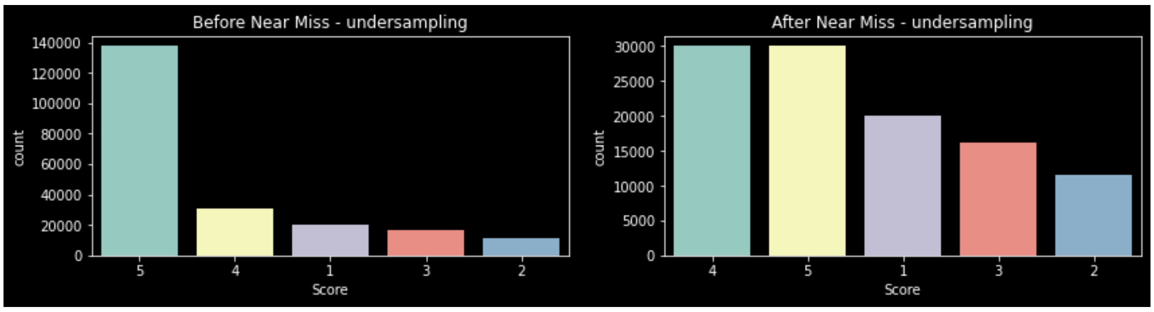
In [11]:
```python
X_test_tfidfvec = tfidf_vectorizer.transform(X_test_text['Review_Proces

TFIDF_TEST_DF = pd.DataFrame (
    data = X_test_tfidfvec.toarray(),
    columns = tfidf_vectorizer.get_feature_names_out(),
)

print((TFIDF_TEST_DF.columns == TFIDF_TRAIN_DF.columns).all())
print(X_test_tfidfvec.shape)
```

```
True
(92534, 1000)
```

In [12]:
```python
X_train_tfidfbow = pd.concat([
    TFIDF_TRAIN_DF,
    X_train_text[["Word_Count", "Longest_Word_Length", "Polarity"]]
], axis=1)

X_test_tfidfbow = pd.concat([
    TFIDF_TEST_DF,
    X_test_text[["Word_Count", "Longest_Word_Length", "Polarity"]]
], axis=1)

y_train_tfidfbow = y_train
y_test_tfidfbow = y_test
```

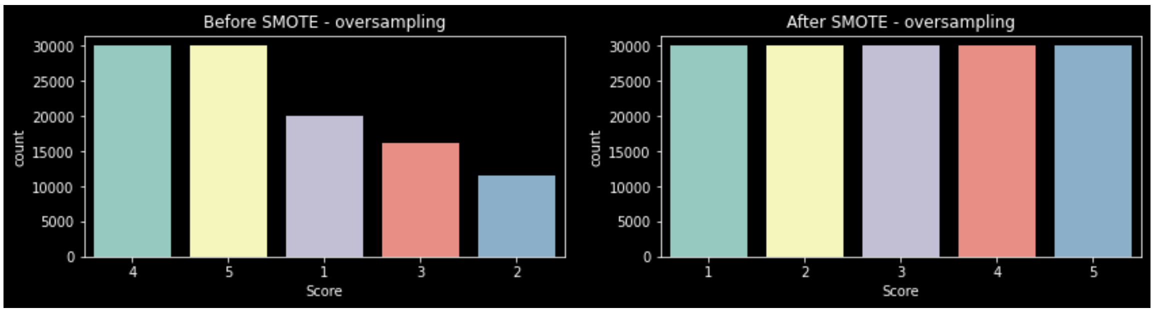
In [13]:
```python
# UNDERSAMPLING
undersample = NearMiss(
    version=1, n_neighbors=3,
    sampling_strategy=strategy(X_train_tfidfbow,y_train_tfidfbow,30000,
)

plt.figure(figsize=(14,3))
plt.subplot(121)
plt.title("Before Near Miss - undersampling")
sns.countplot(x=y_train_tfidfbow, order=y_train_tfidfbow.value_counts()
plt.subplot(122)
X_train_tfidfbow, y_train_tfidfbow = undersample.fit_resample(X_train_t
plt.title("After Near Miss - undersampling")
sns.countplot(x=y_train_tfidfbow, order=y_train_tfidfbow.value_counts()
```

Out[13]:
```
<AxesSubplot:title={'center':'After Near Miss - undersampling'}, xlabel=
core', ylabel='count'>
```



In [14]:
```
# OVERSAMPLING
oversample = SMOTE(
    random_state=30,
    sampling_strategy=strategy(X_train_tfidfbow,y_train_tfidfbow,30000,
)

plt.figure(figsize=(14,3))
plt.subplot(121)
plt.title("Before SMOTE - oversampling")
sns.countplot(x=y_train_tfidfbow, order=y_train_tfidfbow.value_counts()
plt.subplot(122)
X_train_tfidfbow, y_train_tfidfbow = oversample.fit_resample(X_train_tf
plt.title("After SMOTE - oversampling")
sns.countplot(x=y_train_tfidfbow, order=y_train_tfidfbow.value_counts()
```

Out[14]:
```
<AxesSubplot:title={'center':'After SMOTE - oversampling'}, xlabel='Scor
', ylabel='count'>
```



In [15]:
```
X_train_tfidfbow = X_train_tfidfbow.to_numpy()
y_train_tfidfbow = y_train_tfidfbow.to_numpy()
X_test_tfidfbow = X_test_tfidfbow.to_numpy()
y_test_tfidfbow = y_test_tfidfbow.to_numpy()

print(X_train_tfidfbow.shape, y_train_tfidfbow.shape)
print(X_test_tfidfbow.shape, y_test_tfidfbow.shape)
```

```
(150000, 1003) (150000,)
(92534, 1003) (92534,)
```

OBSERVATION: As observed in the `CountVectorizer()` and `TfidfVectorizer()` models, the `max_features` argument has been set to 1000 for each of the above mentioned models. Due to low RAM availability on Google Colab, the `CountVectorizer()` and `TfidfVectorizer()` models were limited to only the top 1000 features (ordered by term frequency) of the given text corpus.

WARNING: Time consuming cell ahead! (upto 3 mins)

In [16]:
```
# save BOW representation of documents as json
with open('./data/train_tfidf_bow.json', 'w') as outfile:
    json.dump({
        "X_train": X_train_tfidfbow.tolist(),
        "X_test": X_test_tfidfbow.tolist(),
        "y_train": y_train_tfidfbow.tolist(),
        "y_test": y_test_tfidfbow.tolist()
    }, outfile)
```

3.3. Word Embedding Using GloVe Model

In [10]:
```
import gensim.downloader as gensim_api
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import BernoulliNB,GaussianNB
from sklearn.preprocessing import label_binarize
from imblearn.under_sampling import NearMiss
from sklearn import svm
```

OBSERVATION: Initially, the `glove-wiki-gigaword-200` model from the Gensim API was chosen as the pre-trained GloVe vectors for the classifiers. However, after comprehensive research, it was found that the pre-trained GloVe vectors offered by the `glove-wiki-gigaword-200` model was trained on ~2 billion tweets. Since the **Amazon Food Reviews** dataset comprises of reviews of fine foods from Amazon which have been rated from a scale of 1 to 5, the `glove-wiki-gigaword-200` model would be unsuitable for the problem formulation. Moreover, instead of tweets bodies, the classifiers would be dealing with reviews and ratings. Therefore, it was decided to use Stanford University's GloVe 100-dimension pre-trained word vectors `(glove.6B.100d.txt)` as the GloVe model.

In [17]:
```
GLOVE_MODEL_100 = {}
with open("./data/glove.6B.100d.txt", "r", encoding="utf8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], 'float32')
        GLOVE_MODEL_100[word] = vector
f.close()
```

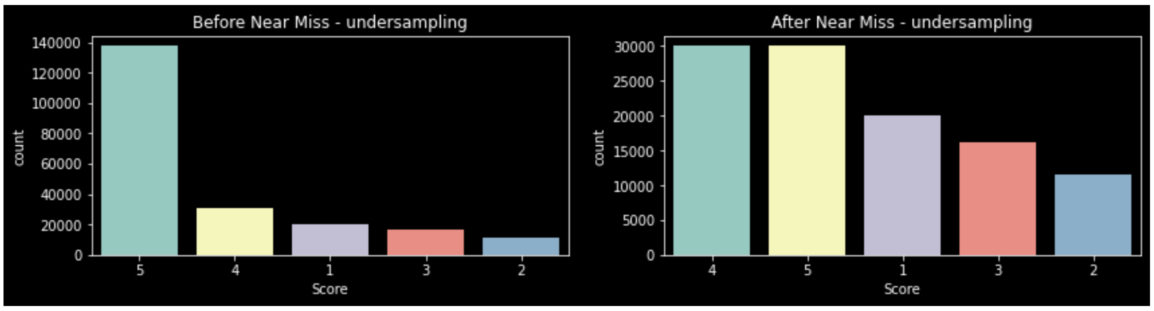
In [82]:
```
from sklearn import preprocessing

def glove_embed(docs):
    ''' Function that converts given documents
        into vectors using a GloVe model.
        @param docs: Documents to embed as an iterable.
        @return docs: Embedded documents as np array.
    '''
    docs_embedded = []
    for doc_words in [doc.split(" ") for doc in docs]:
        doc_vector = np.array([0.0]*100)
        for word in doc_words:
            try: word_vec = GLOVE_MODEL_100[word]
            except: word_vec = np.array([0.0]*100)
            doc_vector += word_vec
        # print(doc_vector.shape)
        # doc_vector = preprocessing.normalize([doc_vector], axis=1)[0]
        docs_embedded.append(doc_vector)
    docs_embedded = preprocessing.normalize(docs_embedded, axis=1)
    return np.array(docs_embedded)
```

In [83]:
```python
X_train_we = pd.concat([
    pd.DataFrame(glove_embed(X_train_text["Review_Processed"])),
    X_train_text[["Word_Count", "Longest_Word_Length", "Polarity"]]
], axis=1)
X_test_we = pd.concat([
    pd.DataFrame(glove_embed(X_test_text["Review_Processed"])),
    X_test_text[["Word_Count", "Longest_Word_Length", "Polarity"]]
], axis=1)
y_train_we = y_train
y_test_we = y_test
```

In [84]:
```python
# UNDERSAMPLING
undersample_glove = NearMiss(
    version=1, n_neighbors=3,
    sampling_strategy=strategy(X_train_we, y_train_we,30000, t='majorit
)

plt.figure(figsize=(14,3))
plt.subplot(121)
plt.title("Before Near Miss - undersampling")
sns.countplot(x=y_train_we, order=y_train_we.value_counts().index)
plt.subplot(122)
X_train_we, y_train_we = undersample_glove.fit_resample(X_train_we, y_t
plt.title("After Near Miss - undersampling")
sns.countplot(x=y_train_we, order=y_train_we.value_counts().index)
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\sklearn\utils
alidation.py:1688: FutureWarning: Feature names only support names that ar
e all strings. Got feature names with dtypes: ['int', 'str']. An error wil
l be raised in 1.2.
  warnings.warn(
```

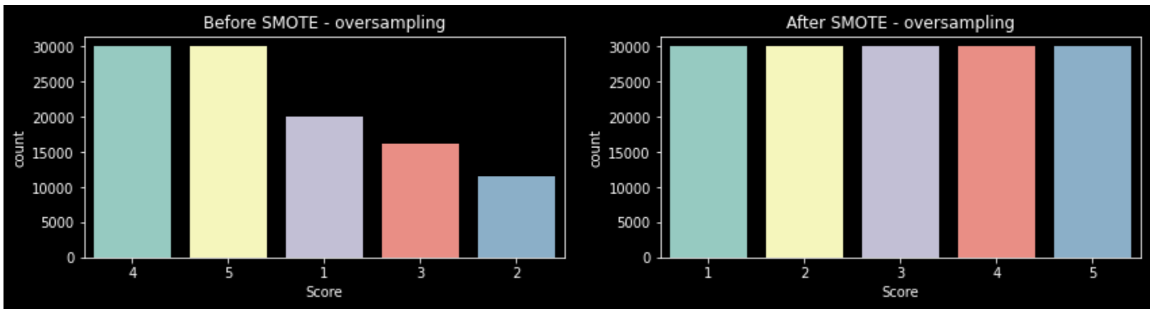
Out[84]:
```
<AxesSubplot:title={'center':'After Near Miss - undersampling'}, xlabel=
core', ylabel='count'>
```



In [85]:
```
# OVERSAMPLING
oversample_glove = SMOTE(
    random_state=30,
    sampling_strategy=strategy(X_train_we, y_train_we, 30000, t='minori
)

plt.figure(figsize=(14,3))
plt.subplot(121)
plt.title("Before SMOTE - oversampling")
sns.countplot(x=y_train_we, order=y_train_we.value_counts().index)
plt.subplot(122)
X_train_we, y_train_we = oversample_glove.fit_resample(X_train_we, y_tr
plt.title("After SMOTE - oversampling")
sns.countplot(x=y_train_we, order=y_train_we.value_counts().index)
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\sklearn\utils\v
alidation.py:1688: FutureWarning: Feature names only support names that ar
e all strings. Got feature names with dtypes: ['int', 'str']. An error wil
l be raised in 1.2.
  warnings.warn(
```

Out[85]:
```
<AxesSubplot:title={'center':'After SMOTE - oversampling'}, xlabel='Scor
', ylabel='count'>
```



In [86]:
```
X_train_we = X_train_we.to_numpy()
y_train_we = y_train_we.to_numpy()
X_test_we = X_test_we.to_numpy()
y_test_we = y_test_we.to_numpy()

print(X_train_we.shape, y_train_we.shape)
print(X_test_we.shape, y_test_we.shape)
```

```
(150000, 103) (150000,)
(92534, 103) (92534,)
```

WARNING: Time consuming cell ahead! (upto 1 min)

In [87]:
```
# save Word Embedding of documents as json
with open('./data/train_we.json', 'w') as outfile:
    json.dump({
        "X_train": X_train_we.tolist(),
        "X_test": X_test_we.tolist(),
        "y_train": y_train_we.tolist(),
        "y_test": y_test_we.tolist()
    }, outfile)
```

4. Model Selection, Training, Hyperparameter Tuning

and Evaluation

In [119]:
```python
import gc

def clean_memory():
    ''' Function that cleans unused memory. '''
    gc.collect()
    tf.keras.backend.clear_session()
```

4.1. Classifier Evaluation Functions

In [1]:
```python
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import label_binarize
```

OBSERVATION: In addition to accuracy, other evaluation metrics such as Precision, Recall, F Score, ROC curve AUC were also calculated to judge the performance of the model better. This was done because accuracy alone can be misleading especially in this case where there are 5 classes. Calculating advanced metrics such as ROC curves of each class (rating 1 to 5) can help determine if the model does well w.r.t. each of them.

In [2]:
```python
def calculate_performance_measures(cm_df):
    ''' Calculates and returns
        * TP, TN, FP, FN
        * Precision, Recall
        * F-Measure
        for every class in a confusion matrix given as a dataFrame.
        @param cm_df = Confusion matrix as a dataFrame.
        @return metrics: Dictionary with all calculated metrics.
    '''
    metrics = {}
    attributes = list(cm_df.columns)

    for attribute in attributes:
        tp = cm_df.loc[[attribute], [attribute]].values[0][0]
        fp = cm_df.loc[:, [attribute]].drop([attribute],axis=0).values.
        fn = cm_df.loc[[attribute], :].drop([attribute],axis=1).values.
        tn = sum([cm_df.loc[[label], [label]].values[0][0] for label in
        precision = tp/(tp+fn)
        recall = tp/(tp+fp)
        f = 2*((recall*precision)/(recall+precision))
        tp_rate = tp/(tp+fn)
        fp_rate = fp/(fp+tn)
        metrics[attribute] = {
            "TP": tp,
            "TN": tn,
            "FP": fp,
            "FN": fn,
            "Precision": precision,
            "Recall": recall,
            "F": f,
            "TP Rate": tp_rate,
            "FP Rate": fp_rate
        }

    return metrics
```

In [3]:
```python
def get_performance_measures(classifier_name, df_name, truth, predictio
    ''' Returns all performance measures of current model on given data
        all calculated metrics.
        @param classifier_name: A name for the classifier.
        @param df_name: A name for the dataframe.
        @param truth: Ground truth.
        @param prediction: Predicted values.
        @param labels: The labels corresponding to classes.
                       (for example: [-1, 0, 1] corresponds to ["negati
        @param prediction_prob: Probability of predictions
        @returns performance_measures: Dictionary with calculated
                                        * Accuracy
                                        * Confusion Matrix
                                        * Class Specific Performance
                                        * ROC
                                       values.
    '''

    # print(TextStyle.GREEN(TextStyle.BOLD("Performance of {} on {}.".f

    performance_measures = {}
    truth = np.array(truth)

    accuracy = accuracy_score(truth, prediction)
    performance_measures["Accuracy"] = accuracy

    # confusion matrix
    cm = pd.DataFrame(confusion_matrix(truth, prediction), index=labels
    performance_measures["Confusion Matrix"] = cm.values.tolist()

    # other performance measures for each attribute
    performance_measures["Class Specific Performance"] = calculate_perf

    plt.figure(figsize=(15,5))
    plt.subplot(121)
    sns.heatmap(cm, cmap="viridis", annot=True)  # plot confusion matri
    plt.title("Confusion Matrix")

    FPRateROC = dict()
    TPRateROC = dict()
    t = label_binarize(truth,classes=labels)
    AUC = dict()

    for i in range(5):
        FPRateROC[i], TPRateROC[i], _ = roc_curve(t[:,i], prediction_pr
        AUC[i] = auc(FPRateROC[i], TPRateROC[i])

    plt.subplot(122)
    plt.title("ROC AUC Curves")
    plt.plot(FPRateROC[0], TPRateROC[0], color='gray', label='1 star (a
    plt.plot(FPRateROC[1], TPRateROC[1], color='pink',label='2 star (ar
    plt.plot(FPRateROC[2], TPRateROC[2], color='purple', label='3 star 
    plt.plot(FPRateROC[3], TPRateROC[3], color='teal', label='4 star (a
    plt.plot(FPRateROC[4], TPRateROC[4], color='cyan',label='5 star (ar
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=12)
    plt.ylabel('True Positive Rate (Recall)', fontsize=12)
    plt.legend(loc="lower right")
    return performance_measures
```

4.2. Classifier Experiments

This section contains experiments conducted using different classifiers.

In [7]:
```python
with open('./data/train_we.json') as f:
  data = json.load(f)

X_train_we = data["X_train"]
X_test_we = data["X_test"]
y_train_we = data["y_train"]
y_test_we = data["y_test"]

del data
```

In [17]:
```python
with open('./data/train_tf_bow.json') as f:
  data = json.load(f)

X_train_tfbow = data["X_train"]
X_test_tfbow = data["X_test"]
y_train_tfbow = data["y_train"]
y_test_tfbow = data["y_test"]

del data
```

In [19]:
```python
with open('./data/train_tfidf_bow.json') as f:
  data = json.load(f)

X_train_bow = data["X_train"]
X_test_bow = data["X_test"]
y_train_bow = data["y_train"]
y_test_bow = data["y_test"]

del data
```

OBSERVATION: Initially the glove 200 dimension model was used but after better results were observed using the 100 dimensions model, the glove model was changed

OBSERVATION:

- Glove gave the best accuracy with the SGD model
- The TFIDF model also gave the best accuracy with the SGD model
- The model using just the term frequencies gave the best accuracy with the Multinomial Naive Bayes

```
In [28]:
from sklearn.multiclass import OneVsRestClassifier
from sklearn.linear_model import LogisticRegression
import sklearn.linear_model as linear_model
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import BernoulliNB, GaussianNB, MultinomialNB,
from sklearn.linear_model import SGDClassifier as SGD
from sklearn import svm
from sklearn.model_selection import GridSearchCV as GSCV
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from numpy import mean
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
```

4.2.1 Logistic Regression

OBSERVATION: To improve the results of the Logistic Regression models, many hyper parameters were tested. The `solvers` tested on were: liblinear, lbfgs, sag, saga and newton-cg. In addition, for each solver different penalties were tested. Many options for the `multi-class` parameter was used as "ovr" for one-vs-rest scheme, "multinomial" for cross-entropy loss. However, regardless of all these tweakings, the results did not improve across all inputs, i.e. no improvement for TF, TF-IDF or GloVe embeddings. Therefore, the default settings were used to run all Logistic Regression models.

Term Frequencies

```
In [ ]:
logreg = LogisticRegression(max_iter=10000)
logreg.fit(X_train_tfbow, y_train_tfbow)
predictions = logreg.predict(X_test_tfbow)
```

```
In [ ]:
print('Logistic Regression Train accuracy %s' % logreg.score(X_train_tf
print('Logistic Regression Test accuracy %s' % accuracy_score(y_test_tf
```

```
Logistic Regression Train accuracy 0.4492466666666667
Logistic Regression Test accuracy 0.3022780815700175
```

TFIDF

```
In [23]:
logreg = LogisticRegression(max_iter=10000)
logreg.fit(X_train_bow, y_train_bow)
predictions = logreg.predict(X_test_bow)
pred_prob = logreg.predict_proba(X_test_bow)
pred_p = np.array([np.argmax(p)+1 for p in pred_prob])
```

```
In [ ]:
print('Logistic Regression Train accuracy %s' % logreg.score(X_train_bo
print('Logistic Regression Test accuracy %s' % accuracy_score(predictio
```

```
Logistic Regression Train accuracy 0.50716
Logistic Regression Test accuracy 0.35618259234443556
```

GloVe

In [3]:
```python
logreg = LogisticRegression(max_iter=10000)
logreg.fit(X_train_we, y_train_we)
predictions = logreg.predict(X_test_we)
y_test = np.array([np.argmax(p)+1 for p in y_test_we])
pred_prob = logreg.predict_proba(X_test_we)
pred_p_max = np.array([np.argmax(p)+1 for p in pred_prob])
```

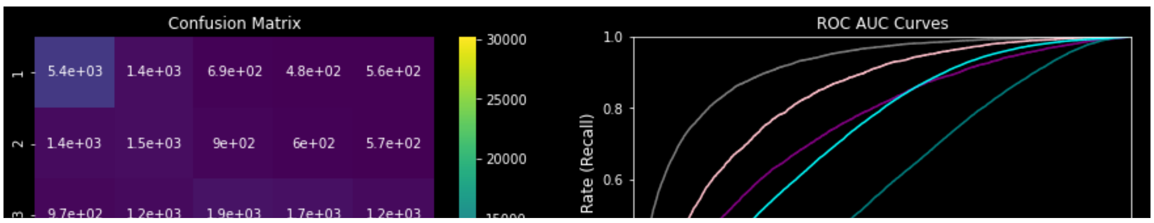
In [12]:
```python
print('Logistic Regression Train accuracy %s' % logreg.score(X_train_we
print('Logistic Regression Test accuracy %s' % accuracy_score(predictio
```

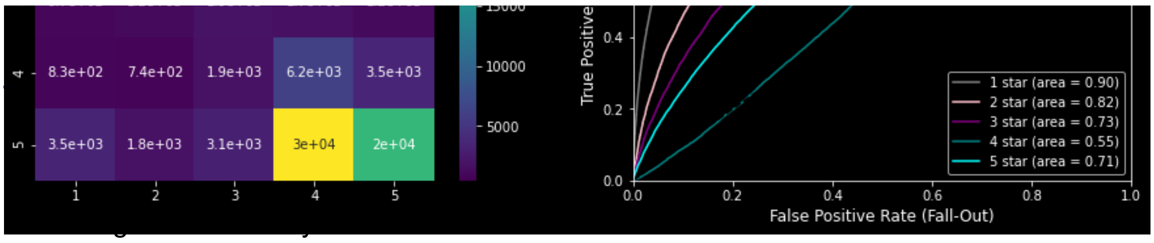
```
Logistic Regression Train accuracy 0.5065533333333333
Logistic Regression Test accuracy 0.3817299587178767
```

OBSERVATION: The results show that the model is overfitting when using the GloVe embeddings.

```
In [13]:  get_performance_measures(logreg, "reviews", y_test_we, pred_p_max, [1,2
```

```
Out[13]: {'Accuracy': 0.3817299587178767,
          'Confusion Matrix': [[5401, 1431, 689, 478, 558],
           [1352, 1490, 899, 601, 570],
           [968, 1193, 1932, 1664, 1219],
           [827, 740, 1864, 6236, 3502],
           [3495, 1833, 3143, 30185, 20264]],
          'Class Specific Performance': {1: {'TP': 5401,
            'TN': 29922,
            'FP': 6642,
            'FN': 3156,
            'Precision': 0.6311791515718126,
            'Recall': 0.44847629328240474,
            'F': 0.5243689320388351,
            'TP Rate': 0.6311791515718126,
            'FP Rate': 0.18165408598621596},
           2: {'TP': 1490,
            'TN': 33833,
            'FP': 5197,
            'FN': 3422,
            'Precision': 0.30333876221498374,
            'Recall': 0.22282039778675042,
            'F': 0.2569186998879214,
            'TP Rate': 0.30333876221498374,
            'FP Rate': 0.1331539841147835},
           3: {'TP': 1932,
            'TN': 33391,
            'FP': 6595,
            'FN': 5044,
            'Precision': 0.2769495412844037,
            'Recall': 0.22657441069543802,
            'F': 0.24924208217764307,
            'TP Rate': 0.2769495412844037,
            'FP Rate': 0.164932726454259},
           4: {'TP': 6236,
            'TN': 29087,
            'FP': 32928,
            'FN': 6933,
            'Precision': 0.47353633533297895,
            'Recall': 0.1592278623225411,
            'F': 0.23831998929929488,
            'TP Rate': 0.47353633533297895,
            'FP Rate': 0.5309683141175522},
           5: {'TP': 20264,
            'TN': 15059,
            'FP': 5849,
            'FN': 38656,
            'Precision': 0.34392396469789543,
            'Recall': 0.7760119480718416,
            'F': 0.47661496125033803,
            'TP Rate': 0.34392396469789543,
            'FP Rate': 0.2797493782284293}}}
```





4.2.2 Naive Bayes

OBSERVATION: Various models for Naive Bayes classification were used such as MultinomialNB, GaussianNB, ComplementNB and BernoulliNB. The MultinomialNB model performed the best and in second place came the GaussianNB in this section across all inputs from TF, TF-IDF and GloVe. These two models were then tested with OneVsRestClassifier as its a commonly used with multiclass classification. It splits the classifier problem into a binary classification of each class. In our case, we have 5 classes, so it would split the multiclassification problem in 5 binary classifications. A binary classification would then be trained on each of these classes and the model with most confidence would be the one making the predictions. (Brownlee, 2020)

Term Frequencies

In [ ]:
```
nb_model = BernoulliNB()
nb_model.fit(X_train_tfbow, y_train_tfbow)
pred = nb_model.predict(X_test_tfbow)
pred_prob = nb_model.predict_proba(X_test_tfbow)
print(nb_model.score(X_test_tfbow, y_test_tfbow))
```

```
0.32621522899690925
```

In [ ]:
```
nb_model_com = ComplementNB()
nb_model_com.fit(X_train_tfbow, y_train_tfbow)
pred_prob = nb_model_com.predict_proba(X_test_tfbow)
print(nb_model_com.score(X_test_tfbow, y_test_tfbow))
```

```
0.2712840685585839
```

In [ ]:
```
nb_model_multi = OneVsRestClassifier(BernoulliNB())
nb_model_multi.fit(X_train_tfbow, y_train_tfbow)
print(nb_model_multi.score(X_test_tfbow, y_test_tfbow))
```

```
0.32659346834676983
```

In [ ]:
```
nb_model_multi = MultinomialNB()
nb_model_multi.fit(X_train_tfbow, y_train_tfbow)
pred_prob = nb_model_multi.predict_proba(X_test_tfbow)
print(nb_model_multi.score(X_test_tfbow, y_test_tfbow))
```

```
0.41107052542849115
```

OBSERVATION: Although many different models were tested only one model performed the best and using the OneVsRestClassifier did not improve the accuracy. Thus, the best scoring model for TF is the MultinomialNB. This is because the MultinomialNB uses word frequency to classify data. (Great Learning Team, 2020). We can run cross-validation on this model and see the results below.

```
In [ ]:
ten_cv = KFold(n_splits=10, random_state = 0, shuffle=True)
ten_cv_scores = cross_val_score(nb_model_multi, X_test_bow, pred_b, cv
print("Accuracy of 10-fold cross validation on testing set: ", ten_cv_s
print("Mean of 10-fold CV accuracy: ", mean(ten_cv_scores))
```

```
Accuracy of 10-fold cross validation on testing set:  [0.64372163 0.6424
9  0.64988113 0.65366328 0.64616881 0.65178861
 0.64930293 0.64757376 0.65016751 0.65135632]
Mean of 10-fold CV accuracy:  0.6486048882422517
```

TFIDF

```
In [101]:
print(X_train_bow[1002][-3])
print(X_train_bow[2004][-2])
```

```
-0.8777470029260738
-0.2083351524776262
```

```
In [ ]:
#TF-IDF
model = Pipeline([('MinMaxScaler', MinMaxScaler()),('MultinomialNB', Mu
model.fit(X_train_bow,y_train_bow)
pred_prob_t = model.predict_proba(X_test_bow)
print(model.score(X_test_bow, y_test_bow))
```

```
0.35200034581883416
```

OBSERVATION: Because Multinomial NB does not take in negative values and as shown above the data for `X_train_bow` contain negative values, the data normalized were normalized using a min-max scaler. This is to allow the normalized values to be recognized/accepted by MultinomialNB.

```
In [ ]:
nb_model_g = OneVsRestClassifier(GaussianNB())
nb_model_g.fit(X_train_bow, y_train_bow)
pred_prob_t = nb_model_g.predict_proba(X_test_bow)
print(nb_model_g.score(X_test_bow, y_test_bow))
```

```
0.3719173492986362
```

GloVe

```
In [ ]:
#GLOVE
nb_model_multi_w = OneVsRestClassifier(GaussianNB()) #Increase by 1% us
nb_model_multi_w.fit(X_train_we, y_train_we)
pred_prob_w = nb_model_multi_w.predict_proba(X_test_we)
print(nb_model_multi_w.score(X_test_we, y_test_we))
```

```
0.3137225236129423
```

```
In [ ]:
#GLOVE
model = Pipeline([('MinMaxScaler',MinMaxScaler()),('MultinomialNB',Mult
y_train_w = np.array([np.argmax(p)+1 for p in y_train_we])
model.fit(X_train_we, y_train_w)
predictions = model.predict(X_test_we)
pred_prob_w = model.predict_proba(X_test_we)
pred_w = np.array([np.argmax(p)+1 for p in y_test_we]) #TRUTH
#print(model.score(X_test_we, pred_w))
print("MultinomialNB() Training Set Accuracy: ", model.score(X_train_we
print("MultinomialNB() Test Set Accuracy: ", model.score(X_test_we, pre
```

```
MultinomialNB() Training Set Accuracy:  0.4051666666666667
MultinomialNB() Test Set Accuracy:  0.37949294313441545
```

```
In [118]:
model = Pipeline([('MinMaxScaler',MinMaxScaler()),('MultinomialNB',Mult
model.fit(X_train_we, y_train_w)
predictions = model.predict(X_test_we)
```

```
In [ ]:
ten_cv = KFold(n_splits=10, random_state = 0, shuffle=True)
ten_cv_scores = cross_val_score(model, X_test_we, pred_w, cv = ten_cv)
print("Accuracy of 10-fold cross validation on testing set: ", ten_cv_s
print("Mean of 10-fold CV accuracy: ", mean(ten_cv_scores))
```

```
Accuracy of 10-fold cross validation on testing set:  [0.63183488 0.6305
15 0.63756214 0.64015561 0.63849562 0.63665838
 0.6369826  0.63492921 0.6397925  0.64044094]
Mean of 10-fold CV accuracy:  0.6367390025854149
```

OBSERVATION:

(Lyashenko and Jha, 2022)

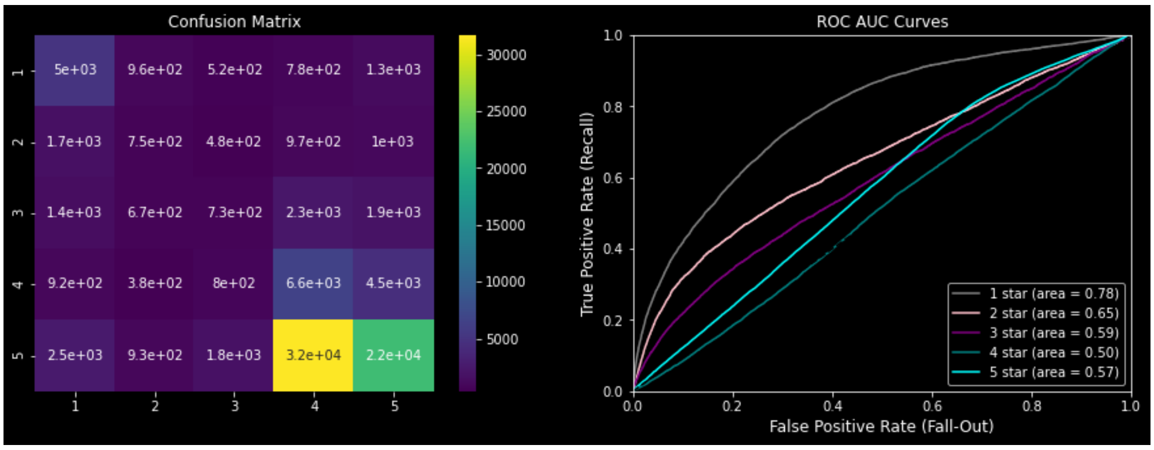
The reason to run cross-validation is to assess the performance of a machine learning model on unseen data; in this case the testing data. K Folds technique will be used as it ensures that every sample from the original data will appear in both the training and testing set.

The data will be split into K groups, and K-1 folds will be used to fit the model. In our case, we are splitting the data into 10 groups, where 9 groups will be used to fit/train model and the validation of the model will happen with the remaining 1 group which will undergo testing.

We can see that across different types of data, i.e. data from TF, TF-IDF and GloVe embeddings, the cross validation provided accuracy levels (for each fold) that are much higher than what the regular train/test has presented. This increase can be explained by the data set being shuffled in the beginning and by having every sample (i.e. sample ranging across reviews of all ratings) appear so that the model is training fairly.

```
In [ ]: get_performance_measures(
            classifier_name = "MultinomialNB",
            df_name = "Amazon Food Review Dataset",
            truth = pred_w,
            prediction = predictions,
            prediction_prob = pred_prob_w,
            labels = [1, 2, 3, 4, 5]
        )
```

```
{'Accuracy': 0.37949294313441545,
 'Confusion Matrix': [[5045, 962, 517, 781, 1252],
  [1706, 746, 481, 974, 1005],
  [1362, 668, 732, 2302, 1912],
  [920, 384, 797, 6607, 4461],
  [2506, 928, 1824, 31676, 21986]],
 'Class Specific Performance': {1: {'TP': 5045,
   'TN': 30071,
   'FP': 6494,
   'FN': 3512,
   'Precision': 0.5895757859062756,
   'Recall': 0.4372129300632637,
   'F': 0.5020899681528662,
   'TP Rate': 0.5895757859062756,
   'FP Rate': 0.17760153151921237},
  2: {'TP': 746,
   'TN': 34370,
   'FP': 2942,
   'FN': 4166,
   'Precision': 0.15187296416938112,
   'Recall': 0.20227765726681127,
   'F': 0.17348837209302326,
   'TP Rate': 0.15187296416938112,
   'FP Rate': 0.07884862778730703},
  3: {'TP': 732,
   'TN': 34384,
   'FP': 3619,
   'FN': 6244,
   'Precision': 0.10493119266055045,
   'Recall': 0.16823718685359687,
   'F': 0.12924869780171272,
   'TP Rate': 0.10493119266055045,
   'FP Rate': 0.09522932400073679},
  4: {'TP': 6607,
   'TN': 28509,
   'FP': 35733,
   'FN': 6562,
   'Precision': 0.5017085579770674,
   'Recall': 0.1560462919225319,
   'F': 0.23805148714622854,
   'TP Rate': 0.5017085579770674,
   'FP Rate': 0.5562248995983936},
  5: {'TP': 21986,
   'TN': 13130,
   'FP': 8630,
   'FN': 36934,
   'Precision': 0.37315003394433127,
   'Recall': 0.7181212437940946,
   'F': 0.49110972122944957,
   'TP Rate': 0.37315003394433127,
'FP Rate': 0.39659926470588236}}
```



4.2.3 Support Vector Machines (SVM)

OBSERVATION: the computing time required for an SVM is directly proportinal to the size of the dataset. Hence other algorithms such as SGD were looked into for dealing with this. Another option was to make use of the coordinate descent algorithm which was tested out but strangely gave all the predicted labels as only the majority class or few of the majority classes despite resampling being done.

TFIDF

In [ ]:
```
linearsvm_classifier_bow = svm.LinearSVC(max_iter=10000)
linearsvm_classifier_bow.fit(X_train_bow, y_train_bow)
prediction_linearsvm_bow = linearsvm_classifier_bow.predict(pd.DataFram
print(accuracy_score(prediction_linearsvm_bow ,y_test_bow))
```

```
0.36811334212289537
```

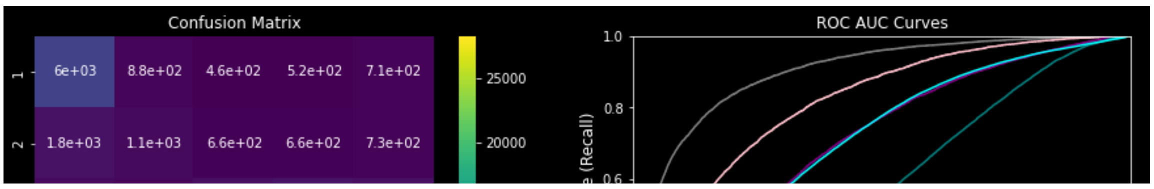
GloVe

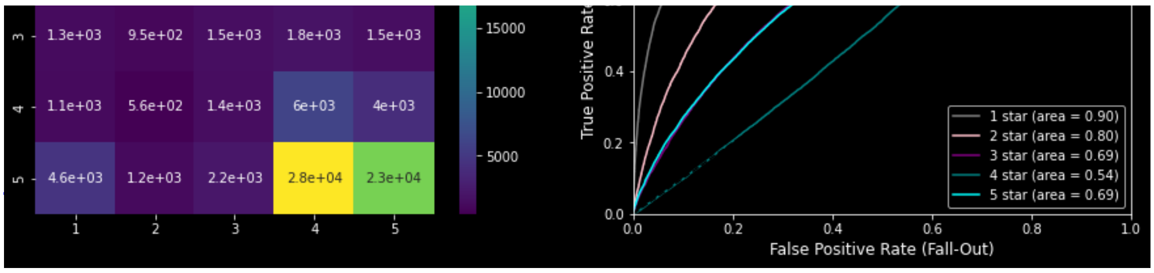
In [ ]:
```
linearsvc_classifier_we= svm.LinearSVC(max_iter=10000)
linearsvc_classifier_we.fit(X_train_we, y_train_we)
prediction_linearsvc_we = linearsvc_classifier_we.predict(pd.DataFrame(
print(accuracy_score(prediction_linearsvc_we ,y_test_we))
```

```
0.4021764972874835
```

In [ ]:
```
pred_prob = np.array(linearsvc_classifier_we.decision_function(X_test_w
get_performance_measures(linearsvc_classifier_we, "reviews", y_test_we,
```

```
{'Accuracy': 0.4021764972874835,
 'Class Specific Performance': {1: {'F': 0.512090916858925,
   'FN': 2564,
   'FP': 8856,
   'FP Rate': 0.22096911023504168,
   'Precision': 0.7003622764987729,
   'Recall': 0.40359620176442856,
   'TN': 31222,
   'TP': 5993,
   'TP Rate': 0.7003622764987729},
  2: {'F': 0.23226210350584306,
   'FN': 3799,
   'FP': 3559,
   'FP Rate': 0.08973550843397797,
   'Precision': 0.22658794788273615,
   'Recall': 0.2382277397260274,
   'TN': 36102,
   'TP': 1113,
   'TP Rate': 0.22658794788273615},
  3: {'F': 0.22215464152664793,
   'FN': 5515,
   'FP': 4716,
   'FP Rate': 0.11653076352853967,
   'Precision': 0.2094323394495413,
   'Recall': 0.23652258377853327,
   'TN': 35754,
   'TP': 1461,
   'TP Rate': 0.2094323394495413},
  4: {'F': 0.2393947795777716,
   'FN': 7125,
   'FP': 31281,
   'FP Rate': 0.500880676359444,
   'Precision': 0.45895664059533753,
   'Recall': 0.1619290020093771,
   'TN': 31171,
   'TP': 6044,
   'TP Rate': 0.45895664059533753},
  5: {'F': 0.5112234397439811,
   'FN': 36316,
   'FP': 6907,
   'FP Rate': 0.32098708058369735,
   'Precision': 0.3836388323150034,
   'Recall': 0.7659516790349361,
   'TN': 14611,
   'TP': 22604,
   'TP Rate': 0.3836388323150034}},
 'Confusion Matrix': [[5993, 875, 457, 523, 709],
  [1750, 1113, 656, 660, 733],
  [1318, 948, 1461, 1793, 1456],
  [1145, 563, 1408, 6044, 4009],
  [4643, 1173, 2195, 28305, 22604]]}
```





```
In [ ]:
linearsvm_classifier_bow_tf = svm.LinearSVC(max_iter=10000)
linearsvm_classifier_bow_tf.fit(X_train_tfbow, y_train_tfbow)
prediction_linearsvm_bow_tf = linearsvm_classifier_bow_tf.predict(pd.Da
print(accuracy_score(prediction_linearsvm_bow_tf ,y_test_tfbow))
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/svm/_base.py:1208: Conver
nceWarning: Liblinear failed to converge, increase the number of iteration
s.
  ConvergenceWarning,

0.31537596991376143
```

4.2.4 Stochastic Gradient Descent (SGD)

OBSERVATION: Almost a 20% increase in the TFIDF model, as well as the glove model, was observed in the accuracy after the involvement of SGD. There was also a significant reduction in the training time in all the models."

Term Frequencies

```
In [ ]:
sgd_classifier_bow_tf = SGD(loss="hinge", penalty="l2", max_iter=10000)
sgd_classifier_bow_tf.fit(X_train_tfbow, y_train_tfbow)
prediction_sgd_bow_tf = sgd_classifier_bow_tf.predict(pd.DataFrame(X_te
print(accuracy_score(prediction_sgd_bow_tf ,y_test_tfbow))
```

```
0.23320076944690601
```

```
In [ ]:
sgd_classifier_bow_tf = SGD(loss="hinge", penalty="l1", max_iter=10000)
sgd_classifier_bow_tf.fit(X_train_tfbow, y_train_tfbow)
prediction_sgd_bow_tf = sgd_classifier_bow_tf.predict(pd.DataFrame(X_te
print(accuracy_score(prediction_sgd_bow_tf ,y_test_tfbow))
```

```
0.33448246050100505
```

TFIDF

```
In [ ]:
sgd_classifier_bow=SGD(loss="hinge", penalty="l2", max_iter=10000)
sgd_classifier_bow.fit(X_train_bow, y_train_bow)
prediction_sgd_bow = sgd_classifier_bow.predict(pd.DataFrame(X_test_bow
print(accuracy_score(prediction_sgd_bow ,y_test_bow))
```

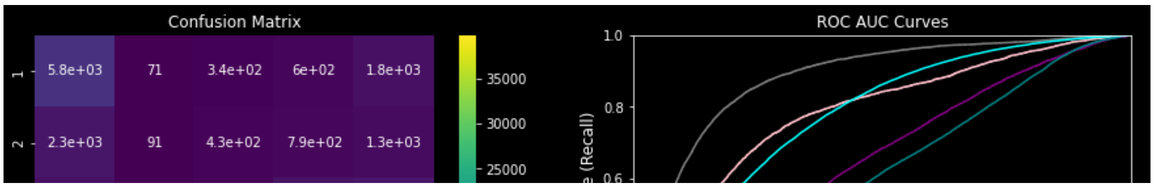
```
0.4526012060431841
```

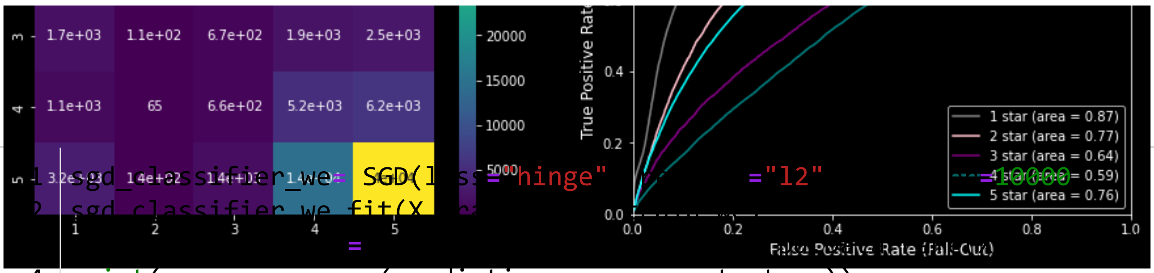
In [ ]:
```
sgd_classifier_bow=SGD(loss="hinge", penalty="l1", max_iter=10000)
sgd_classifier_bow.fit(X_train_bow, y_train_bow)
prediction_sgd_bow = sgd_classifier_bow.predict(pd.DataFrame(X_test_bow
print(accuracy_score(prediction_sgd_bow ,y_test_bow))
```

```
0.5574059264702704
```

```
In [ ]:  pred_prob = np.array(sgd_classifier_bow.decision_function(X_test_bow))
         get_performance_measures(sgd_classifier_bow, "reviews", y_test_bow, pre
```

```
{'Accuracy': 0.5574059264702704,
 'Class Specific Performance': {1: {'F': 0.5098730606488011,
   'FN': 2773,
   'FP': 8347,
   'FP Rate': 0.15416866757785083,
   'Precision': 0.6759378286782751,
   'Recall': 0.40931285825490055,
   'TN': 45795,
   'TP': 5784,
   'TP Rate': 0.6759378286782751},
  2: {'F': 0.03372868791697554,
   'FN': 4821,
   'FP': 393,
   'FP Rate': 0.007575027466702647,
   'Precision': 0.018526058631921825,
   'Recall': 0.18801652892561985,
   'TN': 51488,
   'TP': 91,
   'TP Rate': 0.018526058631921825},
  3: {'F': 0.12862595419847328,
   'FN': 6302,
   'FP': 2830,
   'FP Rate': 0.052665860240066996,
   'Precision': 0.09661697247706422,
   'Recall': 0.192351598173516,
   'TN': 50905,
   'TP': 674,
   'TP Rate': 0.09661697247706422},
  4: {'F': 0.28908783502868707,
   'FN': 7954,
   'FP': 17695,
   'FP Rate': 0.27622972572160037,
   'Precision': 0.39600577112916696,
   'Recall': 0.2276298559580969,
   'TN': 46364,
   'TP': 5215,
   'TP Rate': 0.39600577112916696},
  5: {'F': 0.7211229341181798,
   'FN': 19105,
   'FP': 11690,
   'FP Rate': 0.49842244393280466,
   'Precision': 0.6757467752885268,
   'Recall': 0.7730317444908261,
   'TN': 11764,
   'TP': 39815,
   'TP Rate': 0.6757467752885268}},
 'Confusion Matrix': [[5784, 71, 339, 598, 1765],
  [2336, 91, 426, 793, 1266],
  [1741, 114, 674, 1938, 2509],
  [1077, 65, 662, 5215, 6150],
  [3193, 143, 1403, 14366, 39815]]}
```





In [ ]:
```
sgd_classifier_we= SGD(loss="hinge", penalty="l2", max_iter=10000)
sgd_classifier_we.fit(X_
prediction_sgm_we =
print(accuracy_score(prediction_sgm_we ,y_test_we))
```

```
0.5060194090820671
```

In [ ]:
```
sgd_classifier_we= SGD(loss="hinge", penalty="l1", max_iter=10000)
sgd_classifier_we.fit(X_train_we, y_train_we)
prediction_sgm_we = sgd_classifier_we.predict(pd.DataFrame(X_test_we))
print(accuracy_score(prediction_sgm_we ,y_test_we))
```

```
0.4832710138975944
```

In [ ]:
```
pd.DataFrame(np.array(prediction_sgm_we)).value_counts()
```

```
5    39426
4    26995
1    15319
2     8915
3     1879
dtype: int64
```

grid Search

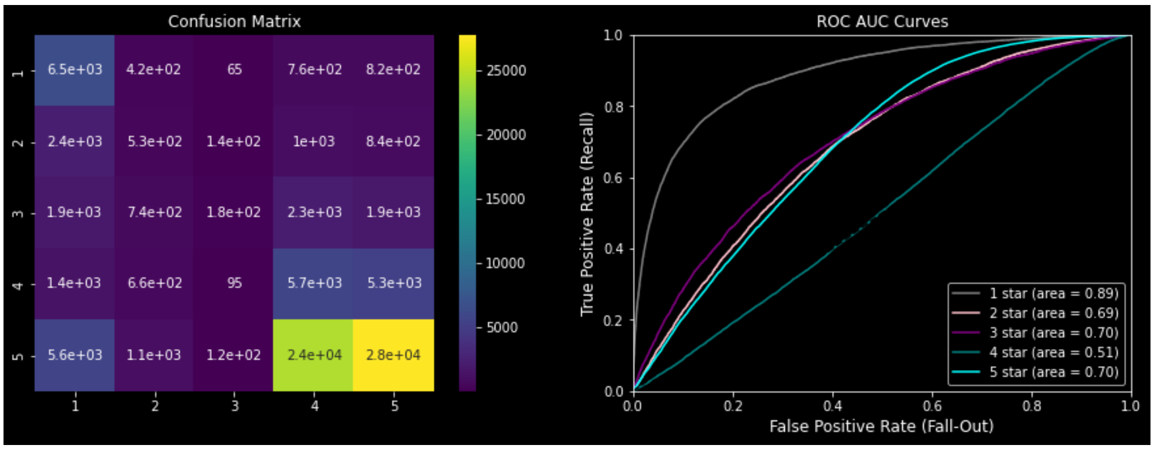
In [ ]:

```python
from sklearn.model_selection import GridSearchCV as GSCV
params = {
             'alpha': [0.0001,0.001, 0.01, 0.1, 1, 10]
             }
grid_search = GSCV(SGD(loss="hinge", penalty="l2", max_iter=10000), par
grid_search.fit(X_train_we, y_train_we)
```

```
Fitting 2 folds for each of 6 candidates, totalling 12 fits
[CV 1/2] END ......................alpha=0.0001;, score=0.408 total time=
5.7s
[CV 2/2] END ......................alpha=0.0001;, score=0.420 total time=
7.9s
[CV 1/2] END .......................alpha=0.001;, score=0.406 total time=
2.2s
[CV 2/2] END .......................alpha=0.001;, score=0.389 total time=
2.2s
[CV 1/2] END ........................alpha=0.01;, score=0.412 total time=
1.8s
[CV 2/2] END ........................alpha=0.01;, score=0.422 total time=
1.8s
[CV 1/2] END .........................alpha=0.1;, score=0.409 total time=
1.5s
[CV 2/2] END .........................alpha=0.1;, score=0.398 total time=
1.6s
[CV 1/2] END ...........................alpha=1;, score=0.405 total time=
1.6s
[CV 2/2] END ...........................alpha=1;, score=0.421 total time=
1.6s
[CV 1/2] END ..........................alpha=10;, score=0.308 total time=
5.7s
[CV 2/2] END ..........................alpha=10;, score=0.200 total time=
4.3s

GridSearchCV(cv=2, estimator=SGDClassifier(max_iter=10000),
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10]},
             verbose=3)
```

```
In [ ]: final=grid_search.best_estimator_
        final.fit(X_train_we, y_train_we)
        prediction_final = final.predict(pd.DataFrame(X_test_we))
        print(accuracy_score(prediction_final,y_test_we))
        pred_prob=final.decision_function(X_test_we)
        get_performance_measures(final, "tweets", y_test_we, prediction_final,
```

```
0.4384982817126678

{'Accuracy': 0.4384982817126678,
 'Class Specific Performance': {1: {'F': 0.49222955045106503,
   'FN': 2064,
   'FP': 11332,
   'FP Rate': 0.24952108334250797,
   'Precision': 0.7587939698492462,
   'Recall': 0.36426367461430575,
   'TN': 34083,
   'TP': 6493,
   'TP Rate': 0.7587939698492462},
  2: {'F': 0.12634344399331263,
   'FN': 4383,
   'FP': 2933,
   'FP Rate': 0.06824104234527688,
   'Precision': 0.10769543973941369,
   'Recall': 0.15280184864240323,
   'TN': 40047,
   'TP': 529,
   'TP Rate': 0.10769543973941369},
  3: {'F': 0.046499339498018495,
   'FN': 6800,
   'FP': 418,
   'FP Rate': 0.010240580136214416,
   'Precision': 0.02522935779816514,
   'Recall': 0.2962962962962963,
   'TN': 40400,
   'TP': 176,
   'TP Rate': 0.02522935779816514},
  4: {'F': 0.2403164731019018,
   'FN': 7489,
   'FP': 28422,
   'FP Rate': 0.44887709656021985,
   'Precision': 0.4313159693218923,
   'Recall': 0.16655914609113834,
   'TN': 34896,
   'TP': 5680,
   'TP Rate': 0.4313159693218923},
  5: {'F': 0.5802390254632297,
   'FN': 31222,
   'FP': 8853,
   'FP Rate': 0.40739036399613454,
   'Precision': 0.4700950441276307,
   'Recall': 0.7577904845284671,
   'TN': 12878,
   'TP': 27698,
   'TP Rate': 0.4700950441276307}},
 'Confusion Matrix': [[6493, 420, 65, 761, 818],
  [2389, 529, 136, 1020, 838],
  [1899, 740, 176, 2274, 1887],
  [1423, 661, 95, 5680, 5310],
  [5621, 1112, 122, 24367, 27698]]}
```

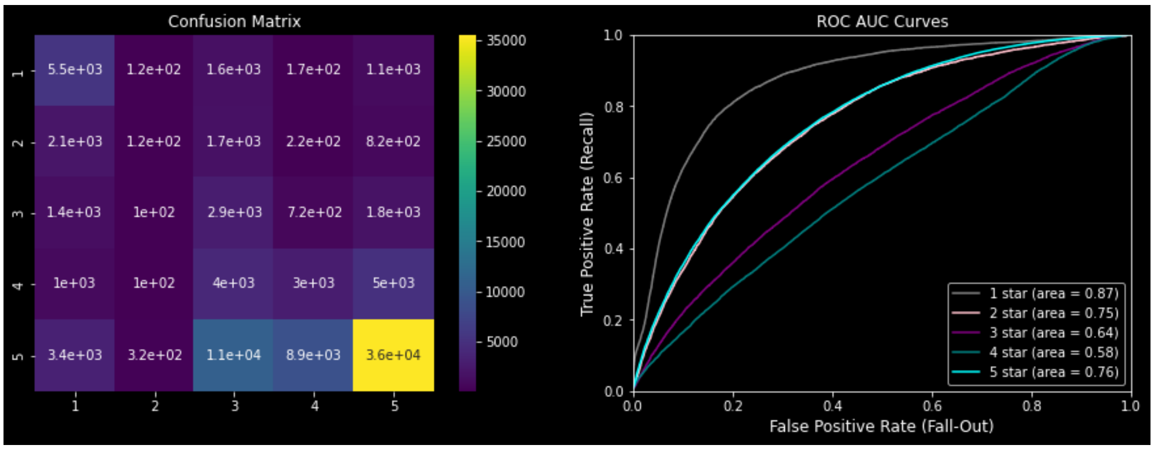


In [ ]:
```python
from sklearn.model_selection import GridSearchCV as GSCV
params = {
                'alpha': [0.0001,0.001, 0.01, 0.1, 1, 10]
                }
grid_search = GSCV(SGD(loss="hinge", penalty="l1", max_iter=10000), par
grid_search.fit(X_train_bow, y_train_bow)
```

```
Fitting 2 folds for each of 6 candidates, totalling 12 fits
[CV 1/2] END ......................alpha=0.0001;, score=0.396 total time=
1.2min
[CV 2/2] END ......................alpha=0.0001;, score=0.396 total time=
60.0s
[CV 1/2] END .......................alpha=0.001;, score=0.337 total time=
27.4s
[CV 2/2] END .......................alpha=0.001;, score=0.329 total time=
26.5s
[CV 1/2] END ........................alpha=0.01;, score=0.337 total time=
15.1s
[CV 2/2] END ........................alpha=0.01;, score=0.350 total time=
18.1s
[CV 1/2] END .........................alpha=0.1;, score=0.200 total time=
14.1s
[CV 2/2] END .........................alpha=0.1;, score=0.200 total time=
13.7s
[CV 1/2] END ...........................alpha=1;, score=0.200 total time=
12.4s
[CV 2/2] END ...........................alpha=1;, score=0.200 total time=
12.6s
[CV 1/2] END ..........................alpha=10;, score=0.200 total time=
29.7s
[CV 2/2] END ..........................alpha=10;, score=0.200 total time=
29.7s

GridSearchCV(cv=2, estimator=SGDClassifier(max_iter=10000, penalty='l1')
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10]},
             verbose=3)
```

In [ ]:
```
final=grid_search.best_estimator_
final.fit(X_train_bow, y_train_bow)
prediction_final = final.predict(pd.DataFrame(X_test_bow))
print(accuracy_score(prediction_final,y_test_bow))
pred_prob=final.decision_function(X_test_bow)
get_performance_measures(final, "tweets", y_test_bow, prediction_final,
```

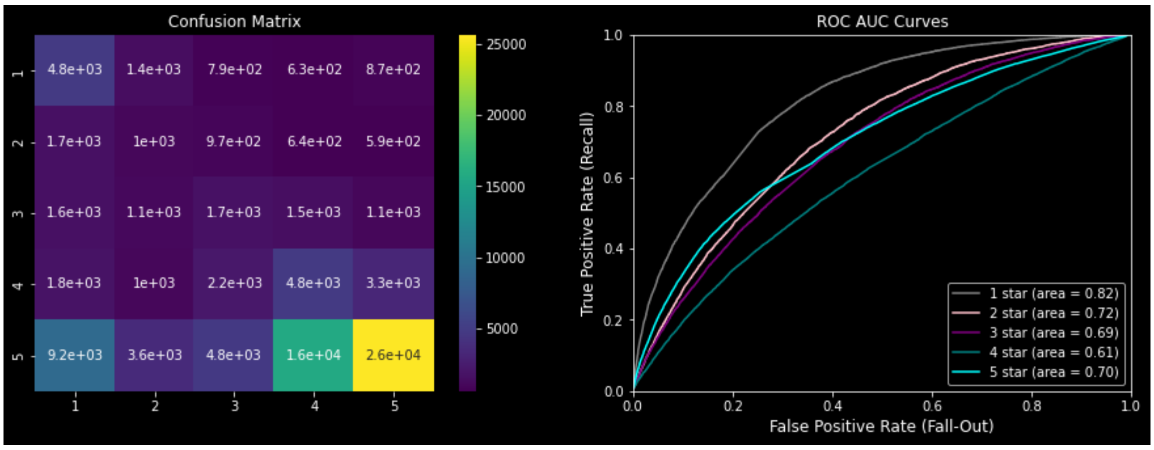
```
0.509120971750924

{'Accuracy': 0.509120971750924,
 'Class Specific Performance': {1: {'F': 0.5027755027755028,
   'FN': 3032,
   'FP': 7896,
   'FP Rate': 0.15957317812537894,
   'Precision': 0.64567021152273,
   'Recall': 0.41166828105208253,
   'TN': 41586,
   'TP': 5525,
   'TP Rate': 0.64567021152273},
  2: {'F': 0.040542922615899875,
   'FN': 4797,
   'FP': 646,
   'FP Rate': 0.013559464338188993,
   'Precision': 0.023412052117263844,
   'Recall': 0.15111695137976347,
   'TN': 46996,
   'TP': 115,
   'TP Rate': 0.023412052117263844},
  3: {'F': 0.20952516653011788,
   'FN': 4035,
   'FP': 18156,
   'FP Rate': 0.2913069986843372,
   'Precision': 0.4215883027522936,
   'Recall': 0.13940370668815472,
   'TN': 44170,
   'TP': 2941,
   'TP Rate': 0.4215883027522936},
  4: {'F': 0.22985428538646882,
   'FN': 10164,
   'FP': 9973,
   'FP Rate': 0.18441539229645518,
   'Precision': 0.22818740982610677,
   'Recall': 0.23154569271074124,
   'TN': 44106,
   'TP': 3005,
   'TP Rate': 0.22818740982610677},
  5: {'F': 0.6884890064633662,
   'FN': 23395,
   'FP': 8752,
   'FP Rate': 0.430327465827515,
   'Precision': 0.6029361846571623,
   'Recall': 0.8023352982361045,
   'TN': 11586,
   'TP': 35525,
   'TP Rate': 0.6029361846571623}},
 'Confusion Matrix': [[5525, 122, 1613, 166, 1131],
  [2103, 115, 1659, 218, 817],
  [1436, 101, 2941, 717, 1781],
  [997, 103, 4041, 3005, 5023],
  [3360, 320, 10843, 8872, 35525]]}
```



Term Frequencies

```
In [ ]: nb_model_multi = MultinomialNB()
        nb_model_multi.fit(X_train_tfbow, y_train_tfbow)
        pred_prob = nb_model_multi.predict_proba(X_test_tfbow)
        print(nb_model_multi.score(X_test_tfbow, y_test_tfbow))
        prediction_final = nb_model_multi.predict(pd.DataFrame(X_test_tfbow))
        get_performance_measures(nb_model_multi, "reviews", y_test_tfbow, predi
```

```
0.41107052542849115

{'Accuracy': 0.41107052542849115,
 'Class Specific Performance': {1: {'F': 0.3499584071756664,
   'FN': 3719,
   'FP': 14254,
   'FP Rate': 0.30037510009693597,
   'Precision': 0.5653850648591796,
   'Recall': 0.2534045673580557,
   'TN': 33200,
   'TP': 4838,
   'TP Rate': 0.5653850648591796},
  2: {'F': 0.159002812191229,
   'FN': 3866,
   'FP': 7199,
   'FP Rate': 0.16290647416894843,
   'Precision': 0.21294788273615636,
   'Recall': 0.12686476652516676,
   'TN': 36992,
   'TP': 1046,
   'TP Rate': 0.21294788273615636},
  3: {'F': 0.19486708388356203,
   'FN': 5279,
   'FP': 8744,
   'FP Rate': 0.19394477098813354,
   'Precision': 0.2432626146788991,
   'Recall': 0.1625323244899914,
   'TN': 36341,
   'TP': 1697,
   'TP Rate': 0.2432626146788991},
  4: {'F': 0.264294720684173,
   'FN': 8348,
   'FP': 18492,
   'FP Rate': 0.3576166624765515,
   'Precision': 0.3660870225529653,
   'Recall': 0.20679449234332775,
   'TN': 33217,
   'TP': 4821,
   'TP Rate': 0.3660870225529653},
  5: {'F': 0.5674003740468997,
   'FN': 33284,
   'FP': 5807,
   'FP Rate': 0.3189082321928717,
   'Precision': 0.43509843856076036,
   'Recall': 0.8153166046496836,
   'TN': 12402,
   'TP': 25636,
   'TP Rate': 0.43509843856076036}},
 'Confusion Matrix': [[4838, 1435, 788, 630, 866],
  [1655, 1046, 973, 644, 594],
  [1588, 1108, 1697, 1509, 1074],
  [1836, 1046, 2193, 4821, 3273],
  [9175, 3610, 4790, 15709, 25636]]}
```



5. Modelling Text as a Sequence

In [98]:
```python
import tensorflow as tf
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
from keras.initializers import Constant
from tensorflow.keras.optimizers import Adam
from keras.preprocessing.text import Tokenizer
import json
```

5.1. LSTM Model

All classifiers prior to this one, were trained and tested on existing BOW or GloVe representation of documents where each document had a single corresponding vector of numbers that represent it.

Since sentiment analysis and text classification (being a NLP problems) are often sequential/time-series problems, they are solved using Recurrent Neural Networks (RNNs) with much success. The LSTM variant of RNN is often preferred for text classification and sentiment because of its innate ability to better capture context of words in a document. (IBM Cloud Education, 2020)

OBSERVATION: Since here, food reviews are to be classified into classes representing ratings which are reflective of someone's sentiment towards the food, it was decided that the LSTM RNN variation would be a good choice of classifier.

OBSERVATION:

- The "many-to-one" kind of RNN shall be used here since the aim is for the model to learn to output a single rating from 1 to 5 (expressed as a vector of 5 numbers) upon receiving the multiple words in a review as input.
- As a result, feature representation of documents shall involve representing each document as a vector of vectors (sub-vectors) such that each sub-vector stands for the encoding of a corresponding word from the document.
- Thus, existing feature representations (where each document is mapped to a vector consisting of numbers representing the document as a whole and not individual words)

which was used as input with other classifiers so far, cannot be used here.

- Hence, an embedding layer shall be incorporated into the model that shall suitably embed each input document into the form required by LSTM.

The embedding layer provided by Keras is trainable and can be trained to produce better embeddings tailored to suit the needs of given classification task. But this means, it is likely more prone to overfitting. Another way to add the embedding layer is to use it with a provided embedding matrix with words in the vocabulary mapped to vectors produced by a pre-trained model similar to the GloVe model used to obtain word embeddings previously. When using a given embedding matrix, the embedding layer can be made non-trainable.

Both ways of embedding shall be explored here.

(DecisionForest, 2020)

5.1.1. Using Pre-Trained GloVe Embeddings

OBSERVATION: Generally, representing documents using word embeddings generated by a pre-trained GloVe model results in good performance of classifiers due to the superior ability of GloVe models when it comes to producing vectors that capture meaning of words better. Thus, it was decided that for this section the embedding layer of the LSTM model shall produce embeddings based on embeddings of words in the vocabulary output by a pre-trained GloVe model.

OBSERVATION: Since, LSTMs are known to work well when provided sequence of words in documents as input, additional features such as polarity, word count and longest word length shall be omitted when working with the LSTM model. Moreover, **train** and **test** labels shall be one hot encoded in order to calculate categorical cross entropy.

In [99]:
```
train_sentences = X_train_text["Review_Processed"] # processed train re
train_labels = pd.get_dummies(y_train).values      # one-hot encoded tr
test_sentences = X_test_text["Review_Processed"]   # processed test rev
test_labels = pd.get_dummies(y_test).values       # one-hot encoded tes
```

In [100]:
```
# map words in vocabulary to index of corresponding token after tokeniz
tokenizer = Tokenizer(num_words=20001, oov_token="[UNK]") # replace out
tokenizer.fit_on_texts(train_sentences)
word_index = tokenizer.word_index
```

In [101]:
```
print(len(word_index))
print(list(word_index.items())[:5])
```

```
109853
[('[UNK]', 1), ('good', 2), ('like', 3), ('taste', 4), ('great', 5)]
```

In [102]:
```
# replace words in each document with its corresponding index in word_m
train_sequences = tokenizer.texts_to_sequences(train_sentences)
test_sequences = tokenizer.texts_to_sequences(test_sentences)
```

In [103]:
```python
print(train_sequences[0], end="\n\n")
print([len(seq) for seq in train_sequences[:3]])
```

```
[476, 166, 9, 313, 241, 671, 1173, 313, 219, 10, 2, 313, 1341, 129, 52, 22
8, 12634, 1505, 19339, 39, 1977, 313, 583, 3395, 105, 1, 111, 181, 1977, 7
62, 531, 349, 88, 261, 334, 1847, 293, 1043, 2, 785, 3576, 1, 2, 39, 1977,
313, 92]

[47, 55, 13]
```

OBSERVATION: Each document has a different length. To use keras implementation of LSTM, all documents must be of same length. Thus padding shall be performed such that all documents have length 50.

In [104]:
```python
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
```

In [105]:
```python
max_len = 50 # each word shall be represented using a vector of 50 numb
```

In [106]:
```python
# pad all document sequences to have 50 words.
train_padded = pad_sequences(train_sequences, maxlen=max_len, padding="
test_padded = pad_sequences(test_sequences, maxlen=max_len, padding="po
```

In [107]:
```python
# checking to ensure uniform document length
print([len(seq) for seq in test_padded[:3]])
```

```
[50, 50, 50]
```

In [108]:
```python
# checking to ensure correct word index mapping
print("sentence =", train_sentences[0], end="\n\n")
print("sentence word index mapping =", list(train_padded[0]), end="\n\n
print('index of 4th word "cake" =', word_index["cake"])
```

```
sentence = recently receive one cake gift normally skeptical cake friend
ake good cake town however first piece wicke jack tavern chocolate rum cak
e pure bliss perfect frostingglaze right amount rum moist melt mouth happy
face smiley request next birthday good job team wjt good chocolate rum cak
e ever

sentence word index mapping = [476, 166, 9, 313, 241, 671, 1173, 313, 219,
10, 2, 313, 1341, 129, 52, 228, 12634, 1505, 19339, 39, 1977, 313, 583, 33
95, 105, 1, 111, 181, 1977, 762, 531, 349, 88, 261, 334, 1847, 293, 1043,
2, 785, 3576, 1, 2, 39, 1977, 313, 92, 0, 0, 0]

index of 4th word "cake" = 313
```

In [111]:
```
# create GloVe embedding dictionary which stores mapping of each word i
# with corresponding vector output by GloVe model
embedding_dict = {}
with open("./data/glove.6B.100d.txt", "r", encoding="utf8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], 'float32')
        embedding_dict[word] = vector
f.close()

# GloVe model is from https://nlp.stanford.edu/projects/glove/,
# Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014.
# GloVe: Global Vectors for Word Representation.
```

In [112]:
```
# create GloVe embedding matrix which is a matrix of GloVe embeddings o
# such that each embedding has index equal to the index of correspondin
num_words = len(word_index) + 1
embedding_matrix = np.zeros((num_words, 100)) # each word vector shall
for word, i in word_index.items():
    if i < num_words:
        emb_vec = embedding_dict.get(word)
        if emb_vec is not None:
            embedding_matrix[i] = emb_vec
```

In [113]:
```
# for eaxmple, embedding of the word "cake" with index 313, shall be as
print((embedding_matrix[313] == embedding_dict.get("cake")).all())
embedding_matrix[313]
```

```
True
```

Out[113]:
```
array([-0.94318002,  0.59859002, -0.29723001, -0.18531001, -0.15003   ,
        0.59964001,  0.86469001, -0.043638  ,  0.045402  ,  0.20723   ,
        0.47813001,  0.10392   ,  0.18468   ,  0.67267001, -0.10693   ,
        0.19472   ,  0.50380999, -0.20807   , -0.029778  , -0.57507002,
       -0.76516998,  0.14342   , -0.041807  , -0.36493   ,  0.12316   ,
        1.21809995, -0.065194  , -0.20683999, -0.24247   , -1.08739996,
       -0.31184   ,  0.60132003,  0.72613001, -0.50184   ,  0.1892    ,
        0.90248001,  0.048367  ,  0.14214   ,  0.23768   , -1.31840003,
       -0.082652  , -0.76125002,  0.17832001, -0.46195999,  0.13675   ,
       -0.34033   , -1.18910003,  0.021234  ,  0.42357999,  0.069305  ,
       -0.36826   , -0.39162999,  0.1022    ,  1.00320005, -1.1135    ,
       -1.29089999,  0.085113  ,  0.68630999,  0.50095999,  0.13337   ,
        0.25975999,  0.92707002, -0.3682    , -0.44510001, -0.048682  ,
       -0.81138003,  0.51595002, -0.10593   , -0.12701   , -1.49749994,
       -0.41769001,  0.41446999,  0.23630001,  0.69452   ,  0.29313999,
        0.85799998,  0.31042001, -1.0194    ,  0.90039003,  0.92777997,
       -0.18077999, -0.36829999, -0.45321   ,  0.55825001, -0.18855999,
       -0.93120998,  0.55419999, -0.31641999,  0.57753003, -0.39559001,
        0.049066  ,  0.30465999, -0.59202999, -0.34942001, -0.41821   ,
       -1.39289999,  0.74163002, -0.48108   ,  0.48195001,  0.01819   ])
```

In [114]:
```
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
from keras.initializers import Constant
from tensorflow.keras.optimizers import Adam
```

In [94]:
```
# make model
model = Sequential(name="LSTM_RNN")

model.add(Embedding(
    name="Embedding_Layer", input_dim = num_words, output_dim = 100,
    embeddings_initializer = Constant(embedding_matrix), # using predef
    input_length = max_len, # all documents shall have length 50
    trainable = False # not trainable since using pre-trained embedding
))
model.add(LSTM(name="LSTM_Layer", units=100, dropout=0.1)) # only 1 LST
model.add(Dense(name="Dense_Layer", units=5, activation='softmax'))

optimizer = Adam(learning_rate=3e-4)
model.compile(loss="categorical_crossentropy", optimizer=optimizer, met
```

In [95]:
```
model.summary()
```

```
Model: "LSTM_RNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Embedding_Layer (Embedding)  (None, 50, 100)          10985400

 LSTM_Layer (LSTM)           (None, 100)               80400

 Dense_Layer (Dense)         (None, 5)                 505

=================================================================
Total params: 11,066,305
Trainable params: 80,905
Non-trainable params: 10,985,400
_________________________________________________________________
```

OBSERVATION: Most complex problems can be solved using 1/2 LSTM layers (K. Eckhardt, 2022). Here, though the classification problem is quite complex, only 1 LSTM layer was used for faster computation that required fewer resources since the model working with a very large dataset as this one, is trained on a simple local machine.

In [96]:
```
print(train_padded.shape, train_labels.shape)
print(test_padded.shape, test_labels.shape)
```

```
(215911, 50) (215911, 5)
(92534, 50) (215911, 5)
```

OBSERVATION: Due to limitations with respect to computing power (Google Colab failed to load embeddings due to memory overflow so could'nt use it) advanced hyper parameter tuning procedures like "grid search" or evaluation procedures "k-fold cross validation" could not be performed here. Instead, **20% of the original training set is set aside for validation** and not used in training. Validation thus performed provides an opportunity to determine the model's bias variance tradeoff.

WARNING: Time consuming cell ahead! (upto 15 mins)

In [97]:

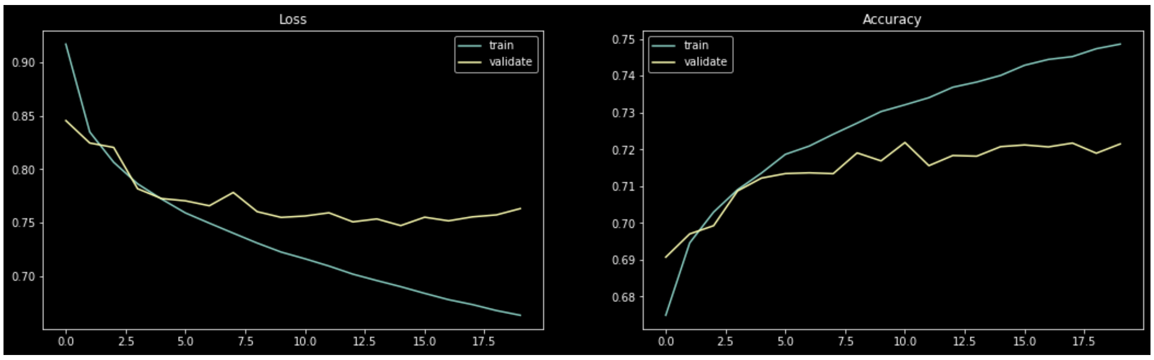
```python
# train model
clean_memory()
history = model.fit(
    x=train_padded,
    y=train_labels,
    epochs=20,
    validation_split=0.2,
    verbose=1,
    shuffle=True
)
```

```
Epoch 1/20
5398/5398 [==============================] - 39s 6ms/step - loss: 0.9169 -
accuracy: 0.6749 - val_loss: 0.8455 - val_accuracy: 0.6907
Epoch 2/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.8348 -
accuracy: 0.6946 - val_loss: 0.8245 - val_accuracy: 0.6970
Epoch 3/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.8065 -
accuracy: 0.7030 - val_loss: 0.8204 - val_accuracy: 0.6993
Epoch 4/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7864 -
accuracy: 0.7090 - val_loss: 0.7818 - val_accuracy: 0.7087
Epoch 5/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7722 -
accuracy: 0.7136 - val_loss: 0.7726 - val_accuracy: 0.7122
Epoch 6/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7592 -
accuracy: 0.7187 - val_loss: 0.7705 - val_accuracy: 0.7134
Epoch 7/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7496 -
accuracy: 0.7209 - val_loss: 0.7659 - val_accuracy: 0.7136
Epoch 8/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7403 -
accuracy: 0.7241 - val_loss: 0.7783 - val_accuracy: 0.7134
Epoch 9/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7311 -
accuracy: 0.7271 - val_loss: 0.7604 - val_accuracy: 0.7191
Epoch 10/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7227 -
accuracy: 0.7303 - val_loss: 0.7550 - val_accuracy: 0.7169
Epoch 11/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7164 -
accuracy: 0.7321 - val_loss: 0.7563 - val_accuracy: 0.7219
Epoch 12/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7096 -
accuracy: 0.7341 - val_loss: 0.7593 - val_accuracy: 0.7156
Epoch 13/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7020 -
accuracy: 0.7369 - val_loss: 0.7508 - val_accuracy: 0.7183
Epoch 14/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.6960 -
accuracy: 0.7383 - val_loss: 0.7536 - val_accuracy: 0.7182
Epoch 15/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.6903 -
accuracy: 0.7401 - val_loss: 0.7474 - val_accuracy: 0.7207
Epoch 16/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.6841 -
accuracy: 0.7429 - val_loss: 0.7552 - val_accuracy: 0.7212
```

```
Epoch 17/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.6782 -
accuracy: 0.7445 - val_loss: 0.7518 - val_accuracy: 0.7207
Epoch 18/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.6736 -
accuracy: 0.7452 - val_loss: 0.7555 - val_accuracy: 0.7217
Epoch 19/20
5398/5398 [==============================] - 31s 6ms/step - loss: 0.6680 -
accuracy: 0.7474 - val_loss: 0.7573 - val_accuracy: 0.7190
Epoch 20/20
5398/5398 [==============================] - 31s 6ms/step - loss: 0.6636 -
accuracy: 0.7486 - val_loss: 0.7633 - val_accuracy: 0.7215
```

In [98]:

```python
# plot training and validation results
plt.figure(figsize=(18,5))

plt.subplot(121)
plt.title("Loss")
plt.plot(history.history["loss"], label="train")
plt.plot(history.history["val_loss"], label="validate")
plt.legend()


plt.subplot(122)
plt.title("Accuracy")
plt.plot(history.history["accuracy"], label="train")
plt.plot(history.history["val_accuracy"], label="validate")
plt.legend()

plt.show()
```



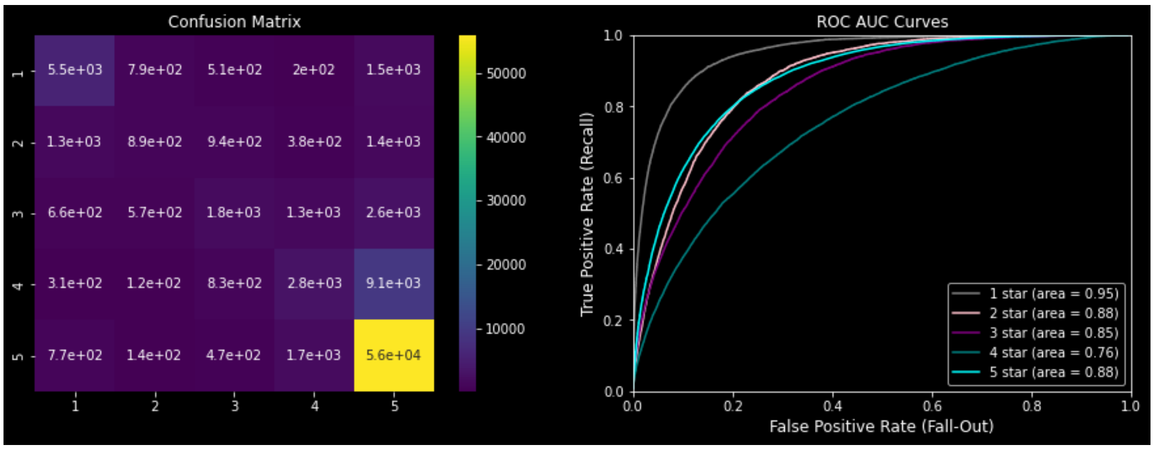
In [105]:

```python
# test model
pred_prob = model.predict(test_padded)
pred = np.array([np.argmax(p)+1 for p in pred_prob])
print("\ntrain results")
train_loss, train_acc = model.evaluate(train_padded, train_labels, verb
print("test results")
test_loss, test_acc = model.evaluate(test_padded, test_labels, verbose=
print(f"Train Accuracy = {train_acc}")
print(f"Test Accuracy = {test_acc}")
```

```
train results
6748/6748 [==============================] - 19s 3ms/step - loss: 0.6535 -
accuracy: 0.7533
test results
2892/2892 [==============================] - 8s 3ms/step - loss: 0.7611 -
accuracy: 0.7228
Train Accuracy = 0.7533242702484131
Test Accuracy = 0.7227613925933838
```

In [111]:
```python
# plot testing results
get_performance_measures(
    classifier_name = "LSTM RNN",
    df_name = "Amazon Food Review Dataset",
    truth = y_test,
    prediction = pred,
    prediction_prob = pred_prob,
    labels = [1, 2, 3, 4, 5]
)
```

Out[111]:
```
{'Accuracy': 0.7227613633907537,
 'Confusion Matrix': [[5534, 792, 511, 198, 1522],
  [1345, 886, 942, 379, 1360],
  [658, 570, 1808, 1296, 2644],
  [313, 118, 830, 2788, 9120],
  [772, 138, 472, 1674, 55864]],
 'Class Specific Performance': {1: {'TP': 5534,
   'TN': 61346,
   'FP': 3088,
   'FN': 3023,
   'Precision': 0.6467219820030384,
   'Recall': 0.6418464393412201,
   'F': 0.6442749869026136,
   'TP Rate': 0.6467219820030384,
   'FP Rate': 0.047925008535866155},
  2: {'TP': 886,
   'TN': 65994,
   'FP': 1618,
   'FN': 4026,
   'Precision': 0.18037459283387622,
   'Recall': 0.3538338658146965,
   'F': 0.2389428263214671,
   'TP Rate': 0.18037459283387622,
   'FP Rate': 0.023930663195882388},
  3: {'TP': 1808,
   'TN': 65072,
   'FP': 2755,
   'FN': 5168,
   'Precision': 0.2591743119266055,
   'Recall': 0.39623055007670394,
   'F': 0.3133720426380102,
   'TP Rate': 0.2591743119266055,
   'FP Rate': 0.040618042962242175},
  4: {'TP': 2788,
   'TN': 64092,
   'FP': 3547,
   'FN': 10381,
   'Precision': 0.21170931733616827,
   'Recall': 0.4400947119179163,
   'F': 0.28589007383100906,
   'TP Rate': 0.21170931733616827,
   'FP Rate': 0.05244016026257041},
  5: {'TP': 55864,
   'TN': 11016,
   'FP': 14646,
   'FN': 3056,
   'Precision': 0.948133061778683,
   'Recall': 0.7922847823003829,
   'F': 0.8632310901645677,
   'TP Rate': 0.948133061778683,
   'FP Rate': 0.5707271451952303}}}
```



OBSERVATION: The model has achieved a fair accuracy of ~72%. Though there is some overfitting as evident from the accuracy/loss curves as well as from the disparity between final training and testing scores. This variance is not very high likely due to 10% dropout implemented in the LSTM layer.

OBSERVATION: From ROC AUC values, it seems that w.r.t. all samples corresponding to each rating (1 to 5), the model has achieved a similar degree of correctness in prediction. Judging from ROC curves alone, the model seems to have performed best with rating 1 samples since it correctly identified most (95%) of 1 rated samples with fewer misclassifications of samples with other ratings as having rating 1. Reasons like samples with rating 1 being much fewer in number than others combined with consequently likely less variation among such reviews may have contributed to the model exhibiting the observed result w.r.t reviews with rating 1.

OBSERVATION: Considering Precision and Recall values whose tradeoff is measured using the F score though, it can be noted that the model performed best when classifying samples with rating 5 (F = 0.86) and second best when classifying samples with rating 1 (F = 0.64). With ratings in between, seemingly confusing the model a bit. This is understandable, because overwhelmingly positive and negative reviews are indeed easier to discern compared to reviews with ratings in between.

5.1.2. With Custom Training of Embedding Layer

Since same padded sequences from previous experiment is to be used here, they need not be re-computed.

In [78]:

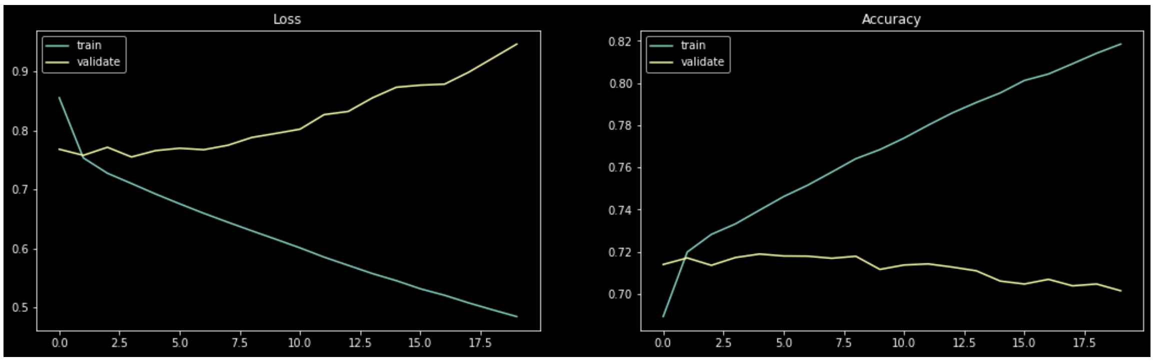
```python
# make model
model = Sequential(name="LSTM_RNN")

model.add(Embedding(
    name="Embedding_Layer", input_dim = num_words, output_dim = 100,
    input_length = max_len, # all documents shall have length 50
    trainable = True # predefined embedding matrix not set and layer ma
))
model.add(LSTM(name="LSTM_Layer", units=100, dropout=0.5)) # only 1 LST
model.add(Dense(name="Dense_Layer", units=5, activation='softmax'))

optimizer = Adam(learning_rate=3e-4)
model.compile(loss="categorical_crossentropy", optimizer=optimizer, met
```

In [79]:
```python
# train model
clean_memory()
history = model.fit(
    x=train_padded,
    y=train_labels,
    epochs=20,
    validation_split=0.2,
    verbose=1
)
```

```
Epoch 1/20
5398/5398 [==============================] - 56s 10ms/step - loss: 0.8555
- accuracy: 0.6893 - val_loss: 0.7682 - val_accuracy: 0.7140
Epoch 2/20
5398/5398 [==============================] - 53s 10ms/step - loss: 0.7535
- accuracy: 0.7199 - val_loss: 0.7578 - val_accuracy: 0.7171
Epoch 3/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.7275 -
accuracy: 0.7283 - val_loss: 0.7715 - val_accuracy: 0.7136
Epoch 4/20
5398/5398 [==============================] - 50s 9ms/step - loss: 0.7101 -
accuracy: 0.7332 - val_loss: 0.7550 - val_accuracy: 0.7173
Epoch 5/20
5398/5398 [==============================] - 50s 9ms/step - loss: 0.6923 -
accuracy: 0.7397 - val_loss: 0.7658 - val_accuracy: 0.7190
Epoch 6/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.6757 -
accuracy: 0.7462 - val_loss: 0.7699 - val_accuracy: 0.7180
Epoch 7/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.6596 -
accuracy: 0.7516 - val_loss: 0.7674 - val_accuracy: 0.7179
Epoch 8/20
5398/5398 [==============================] - 51s 10ms/step - loss: 0.6445
- accuracy: 0.7578 - val_loss: 0.7748 - val_accuracy: 0.7169
Epoch 9/20
5398/5398 [==============================] - 52s 10ms/step - loss: 0.6298
- accuracy: 0.7641 - val_loss: 0.7881 - val_accuracy: 0.7179
Epoch 10/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.6155 -
accuracy: 0.7685 - val_loss: 0.7950 - val_accuracy: 0.7117
Epoch 11/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.6009 -
accuracy: 0.7739 - val_loss: 0.8022 - val_accuracy: 0.7138
Epoch 12/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.5853 -
accuracy: 0.7800 - val_loss: 0.8268 - val_accuracy: 0.7143
Epoch 13/20
5398/5398 [==============================] - 51s 10ms/step - loss: 0.5714
- accuracy: 0.7858 - val_loss: 0.8322 - val_accuracy: 0.7128
Epoch 14/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.5575 -
accuracy: 0.7908 - val_loss: 0.8551 - val_accuracy: 0.7110
Epoch 15/20
5398/5398 [==============================] - 51s 10ms/step - loss: 0.5454
- accuracy: 0.7953 - val_loss: 0.8732 - val_accuracy: 0.7061
Epoch 16/20
5398/5398 [==============================] - 51s 9ms/step - loss: 0.5316 -
accuracy: 0.8012 - val_loss: 0.8768 - val_accuracy: 0.7047
Epoch 17/20
```

```
5398/5398 [==============================] - 51s 9ms/step - loss: 0.5206 -
accuracy: 0.8043 - val_loss: 0.8785 - val_accuracy: 0.7070
Epoch 18/20
5398/5398 [==============================] - 50s 9ms/step - loss: 0.5077 -
accuracy: 0.8092 - val_loss: 0.8987 - val_accuracy: 0.7039
Epoch 19/20
5398/5398 [==============================] - 47s 9ms/step - loss: 0.4959 -
accuracy: 0.8141 - val_loss: 0.9225 - val_accuracy: 0.7047
Epoch 20/20
5398/5398 [==============================] - 46s 9ms/step - loss: 0.4845 -
accuracy: 0.8185 - val_loss: 0.9464 - val_accuracy: 0.7015
```

In [82]:
```python
# plot training and validation results
plt.figure(figsize=(18,5))

plt.subplot(121)
plt.title("Loss")
plt.plot(history.history["loss"], label="train")
plt.plot(history.history["val_loss"], label="validate")
plt.legend()


plt.subplot(122)
plt.title("Accuracy")
plt.plot(history.history["accuracy"], label="train")
plt.plot(history.history["val_accuracy"], label="validate")
plt.legend()

plt.show()
```

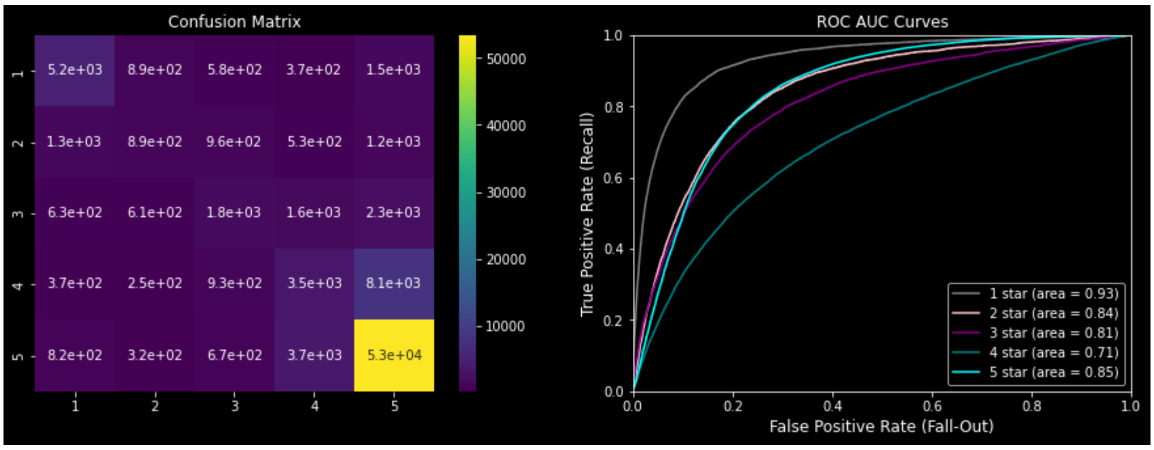


In [80]:
```python
# test model
pred_prob = model.predict(test_padded)
pred = np.array([np.argmax(p)+1 for p in pred_prob])
print("\ntrain results")
train_loss, train_acc = model.evaluate(train_padded, train_labels, verb
print("test results")
test_loss, test_acc = model.evaluate(test_padded, test_labels, verbose=
print(f"Train Accuracy = {train_acc}")
print(f"Test Accuracy = {test_acc}")
```

```
train results
6748/6748 [==============================] - 16s 2ms/step - loss: 0.5132 -
accuracy: 0.8227
test results
2892/2892 [==============================] - 7s 2ms/step - loss: 0.9466 -
accuracy: 0.7005
Train Accuracy = 0.82266765832901
Test Accuracy = 0.7005425095558167
```

In [81]:
```python
# plot testing results
get_performance_measures(
    classifier_name = "LSTM RNN",
    df_name = "Amazon Food Review Dataset",
    truth = y_test,
    prediction = pred,
    prediction_prob = pred_prob,
    labels = [1, 2, 3, 4, 5]
)
```

Out[81]:
```
{'Accuracy': 0.7005425032960858,
 'Confusion Matrix': [[5213, 893, 585, 371, 1495],
  [1297, 893, 957, 528, 1237],
  [633, 608, 1818, 1591, 2326],
  [367, 251, 933, 3507, 8111],
  [819, 320, 669, 3719, 53393]],
 'Class Specific Performance': {1: {'TP': 5213,
   'TN': 59611,
   'FP': 3116,
   'FN': 3344,
   'Precision': 0.6092088348720346,
   'Recall': 0.6258854604394285,
   'F': 0.6174345611749379,
   'TP Rate': 0.6092088348720346,
   'FP Rate': 0.049675578299615795},
  2: {'TP': 893,
   'TN': 63931,
   'FP': 2072,
   'FN': 4019,
   'Precision': 0.18179967426710097,
   'Recall': 0.3011804384485666,
   'F': 0.2267360670305954,
   'TP Rate': 0.18179967426710097,
   'FP Rate': 0.03139251246155478},
  3: {'TP': 1818,
   'TN': 63006,
   'FP': 3144,
   'FN': 5158,
   'Precision': 0.2606077981651376,
   'Recall': 0.36638452237001207,
   'F': 0.30457363042385655,
   'TP Rate': 0.2606077981651376,
   'FP Rate': 0.047528344671201816},
  4: {'TP': 3507,
   'TN': 61317,
   'FP': 6209,
   'FN': 9662,
   'Precision': 0.2663072366922318,
   'Recall': 0.36095100864553314,
   'F': 0.30648896657199043,
   'TP Rate': 0.2663072366922318,
   'FP Rate': 0.09194976749696414},
  5: {'TP': 53393,
   'TN': 11431,
   'FP': 13169,
   'FN': 5527,
   'Precision': 0.9061948404616429,
   'Recall': 0.8021543823803371,
   'F': 0.8510065188632632,
    'TP Rate': 0.9061948404616429,
```



OBSERVATION: This model does better with training data achieving accuracy of ~82% but performs poorly with testing data with only ~70% accuracy.

OBSERVATION: As expected, the model with a trainable embedding layer on its own without added dropout layers, show significant amount of over-fitting. This can however likely be reduced upon incorporating more dropout into the model.

OBSERVATION: From experiments conducted here, it can be observed that the LSTM model which worked with predefined GloVe word embeddings gave best overall results showing better bias-variance trade-off compared to the locally trained embeddings. This is because training the embedding layer makes it adapt very well to the training set but not necessarily to the testing set as it may not be capturing meaning of the words like the glove embeddings do, in the limited number of epochs the simple model was trained for.

OBSERVATION: The LSTM model, staying true to its reputation, showed best performance among all classifies so far w.r.t this NLP task.

5.2. GRU Model

Recurrent Neural Networks (RNNs) are unsuitable for carrying information from the first time-step to the subsequent ones as the RNN gradients tend to vanish in the earlier layers, leaving essential text information out. This makes RNNs ineffective for real-time text classification problems. (A. Jaiswal, 2022)

Since RNNs are unable to capture the context of a sentence and tend to leave essential text related information out, popular variants of RNNs, i.e., LSTMs and GRUs are preferred for text classification problems over regular RNNs. As an LSTM model has already been implemented above, it was decided to experiment our classification problem with another variant of RNN, i.e., the GRU model.

OBSERVATION: As seen from the above two experiments involving the LSTM classifier, the LSTM produced the best results for the predefined GloVe word embeddings as compared to the locally trained embeddings. Due to this reason, it was decided that the embedding layer of the GRU model shall produce embeddings based on the pre-trained GloVe model.

Since the same padded sequences from previous experiment is to be used here, they need not be re-computed.

In [115]:
```python
from keras.layers import GRU
```

In [116]:
```python
# make model
model_gru = Sequential(name="GRU_RNN")

model_gru.add(Embedding(
    name="Embedding_Layer", input_dim = num_words, output_dim = 100,
    embeddings_initializer = Constant(embedding_matrix), # using redefi
    input_length = max_len, # all documents shall have length 50
    trainable = False # not trainable since using pre-trained embedding
))
model_gru.add(GRU(name="GRU_Layer", units=100, dropout=0.1))
model_gru.add(Dense(name="Dense_Layer", units=5, activation='softmax'))

optimizer = Adam(learning_rate=3e-4)
model_gru.compile(loss="categorical_crossentropy", optimizer=optimizer,
```

In [117]:
```python
model_gru.summary()
```

```
Model: "GRU_RNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 Embedding_Layer (Embedding)  (None, 50, 100)          10985400  
                                                                 
 GRU_Layer (GRU)             (None, 100)               60600     
                                                                 
 Dense_Layer (Dense)         (None, 5)                 505       
                                                                 
=================================================================
Total params: 11,046,505
Trainable params: 61,105
Non-trainable params: 10,985,400
_________________________________________________________________
```

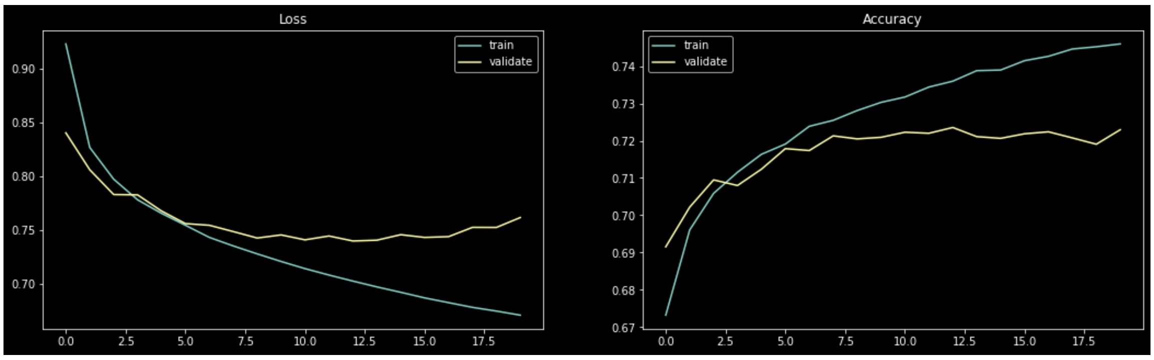
In [121]:
```python
# train model
clean_memory()
history_gru = model_gru.fit(
    x=train_padded,
    y=train_labels,
    epochs=20,
    validation_split=0.2,
    verbose=1
)
```

```
Epoch 1/20
5398/5398 [==============================] - 46s 6ms/step - loss: 0.9228 -
accuracy: 0.6732 - val_loss: 0.8403 - val_accuracy: 0.6915
Epoch 2/20
5398/5398 [==============================] - 36s 7ms/step - loss: 0.8267 -
accuracy: 0.6961 - val_loss: 0.8060 - val_accuracy: 0.7022
Epoch 3/20
5398/5398 [==============================] - 34s 6ms/step - loss: 0.7972 -
accuracy: 0.7059 - val_loss: 0.7830 - val_accuracy: 0.7095
Epoch 4/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7781 -
accuracy: 0.7116 - val_loss: 0.7827 - val_accuracy: 0.7080
Epoch 5/20
5398/5398 [==============================] - 35s 6ms/step - loss: 0.7655 -
accuracy: 0.7164 - val_loss: 0.7677 - val_accuracy: 0.7124
Epoch 6/20
5398/5398 [==============================] - 34s 6ms/step - loss: 0.7544 -
accuracy: 0.7191 - val_loss: 0.7560 - val_accuracy: 0.7179
Epoch 7/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7432 -
accuracy: 0.7239 - val_loss: 0.7544 - val_accuracy: 0.7174
Epoch 8/20
5398/5398 [==============================] - 31s 6ms/step - loss: 0.7353 -
accuracy: 0.7255 - val_loss: 0.7486 - val_accuracy: 0.7213
Epoch 9/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7279 -
accuracy: 0.7281 - val_loss: 0.7426 - val_accuracy: 0.7205
Epoch 10/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7208 -
accuracy: 0.7303 - val_loss: 0.7455 - val_accuracy: 0.7209
Epoch 11/20
5398/5398 [==============================] - 32s 6ms/step - loss: 0.7142 -
accuracy: 0.7317 - val_loss: 0.7409 - val_accuracy: 0.7223
Epoch 12/20
5398/5398 [==============================] - 34s 6ms/step - loss: 0.7083 -
accuracy: 0.7344 - val_loss: 0.7445 - val_accuracy: 0.7220
Epoch 13/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.7026 -
accuracy: 0.7360 - val_loss: 0.7399 - val_accuracy: 0.7235
Epoch 14/20
5398/5398 [==============================] - 34s 6ms/step - loss: 0.6973 -
accuracy: 0.7388 - val_loss: 0.7406 - val_accuracy: 0.7211
Epoch 15/20
5398/5398 [==============================] - 34s 6ms/step - loss: 0.6922 -
accuracy: 0.7390 - val_loss: 0.7457 - val_accuracy: 0.7206
Epoch 16/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.6870 -
accuracy: 0.7415 - val_loss: 0.7431 - val_accuracy: 0.7219
Epoch 17/20
```

```
5398/5398 [==============================] - 33s 6ms/step - loss: 0.6826 -
accuracy: 0.7426 - val_loss: 0.7439 - val_accuracy: 0.7224
Epoch 18/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.6783 -
accuracy: 0.7446 - val_loss: 0.7525 - val_accuracy: 0.7207
Epoch 19/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.6748 -
accuracy: 0.7452 - val_loss: 0.7524 - val_accuracy: 0.7191
Epoch 20/20
5398/5398 [==============================] - 33s 6ms/step - loss: 0.6710 -
```

In [122]:

```python
plt.figure(figsize=(18,5))

plt.subplot(121)
plt.title("Loss")
plt.plot(history_gru.history["loss"], label="train")
plt.plot(history_gru.history["val_loss"], label="validate")
plt.legend()


plt.subplot(122)
plt.title("Accuracy")
plt.plot(history_gru.history["accuracy"], label="train")
plt.plot(history_gru.history["val_accuracy"], label="validate")
plt.legend()

plt.show()
```



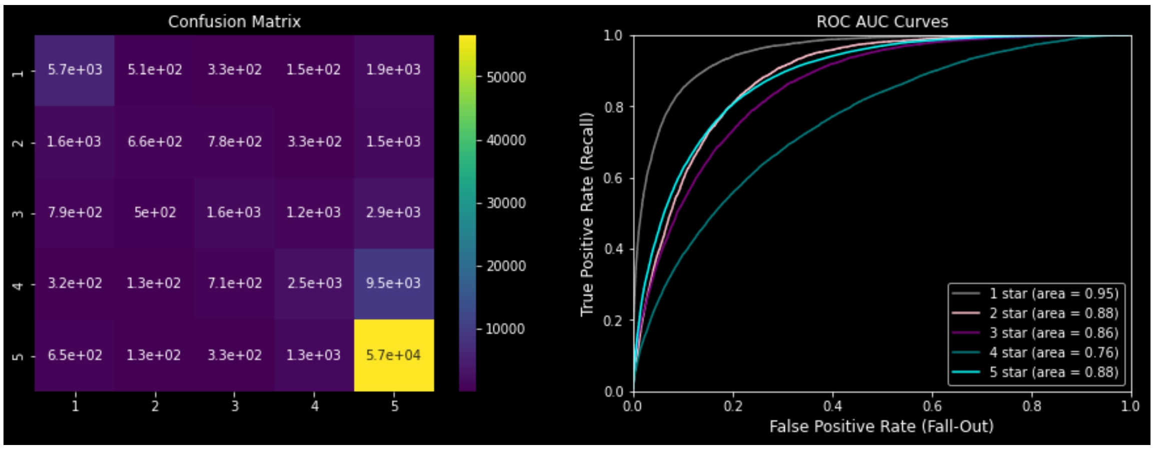
In [125]:

```python
# test model
pred_prob_gru = model_gru.predict(test_padded)
pred_gru = np.array([np.argmax(p)+1 for p in pred_prob_gru])
print("\ntrain results")
train_loss_gru, train_acc_gru = model_gru.evaluate(train_padded, train_
print("test results")
test_loss_gru, test_acc_gru = model_gru.evaluate(test_padded, test_labe
print(f"Train Accuracy = {train_acc_gru}")
print(f"Test Accuracy = {test_acc_gru}")
```

```

train results
6748/6748 [==============================] - 24s 4ms/step - loss: 0.6652 -
accuracy: 0.7494
test results
2892/2892 [==============================] - 8s 3ms/step - loss: 0.7576 -
accuracy: 0.7242
Train Accuracy = 0.7493504285812378
Test Accuracy = 0.7241878509521484
```

In [124]:
```
get_performance_measures(
    classifier_name = "GRU RNN",
    df_name = "Amazon Food Review Dataset",
    truth = y_test,
    prediction = pred_gru,
    prediction_prob = pred_prob_gru,
    labels = [1, 2, 3, 4, 5]
)
```

Out[124]:
```
{'Accuracy': 0.7241878660816564,
 'Confusion Matrix': [[5712, 506, 329, 154, 1856],
  [1624, 660, 775, 327, 1526],
  [789, 498, 1617, 1155, 2917],
  [320, 131, 707, 2502, 9509],
  [650, 126, 328, 1295, 56521]],
 'Class Specific Performance': {1: {'TP': 5712,
   'TN': 61300,
   'FP': 3383,
   'FN': 2845,
   'Precision': 0.6675236648358069,
   'Recall': 0.6280373831775701,
   'F': 0.6471787899388172,
   'TP Rate': 0.6675236648358069,
   'FP Rate': 0.052301222887002764},
  2: {'TP': 660,
   'TN': 66352,
   'FP': 1261,
   'FN': 4252,
   'Precision': 0.13436482084690554,
   'Recall': 0.3435710567412806,
   'F': 0.19318015512951853,
   'TP Rate': 0.13436482084690554,
   'FP Rate': 0.018650259565468178},
  3: {'TP': 1617,
   'TN': 65395,
   'FP': 2139,
   'FN': 5359,
   'Precision': 0.2317947247706422,
   'Recall': 0.4305111821086262,
   'F': 0.30134178158777486,
   'TP Rate': 0.2317947247706422,
   'FP Rate': 0.03167293511416472},
  4: {'TP': 2502,
   'TN': 64510,
   'FP': 2931,
   'FN': 10667,
   'Precision': 0.18999164704988988,
   'Recall': 0.4605190502484815,
   'F': 0.26900333297494894,
   'TP Rate': 0.18999164704988988,
   'FP Rate': 0.043460209664743996},
  5: {'TP': 56521,
   'TN': 10491,
   'FP': 15808,
   'FN': 2399,
   'Precision': 0.9592837746096402,
   'Recall': 0.7814431279293229,
   'F': 0.8612789430776615,
   'TP Rate': 0.9592837746096402,
'FP Rate': 0.6010874938219579}}
```



OBSERVATION: As observed from the experiment conducted above, the GRU model achieved an accuracy of ~72%. From the “Loss” graph and the difference between the train and test scores, it is quite evident that the model us still subjected to some overfitting. However, due to the utilization of the dropout strategy in the GRU model, the overfitting is limited.

OBSERVATION: From the ROC AUC Curves plot, it can be observed that the model was able to distinguish between the classes – [1, 2, 3, 4, 5]. It is clear from the ROC AUC Curves plot, that the AUC for the “1 star rated samples” ROC curve is the highest amongst the remaining four classes. This also has been observed in the experiment conducted using the LSTM RNN.

OBSERVATION: In terms of the F1 score, the model calculated the best F1 score for samples with a rating of 5 (F1 score = 0.86), whereas samples having a rating of 2, an F1 score of 0.19 was generated. The F1 score values were as such due to the imbalanced dataset used as input to the GRU model. The samples having a rating of 2 had the lowest F1 score in comparison to the other four classes, since they served as the minority class in the imbalanced dataset. Due to the reason that most of the reviews had a rating of 5, the “5 star rated” samples had the highest F1 score.

5.3. LSTM Vs GRU

OBSERVATION: From the above two experiments (GRU and LSTM) conducted, it is evident that the LSTM performed slightly better than the GRU model. This is because, as opposed to GRU, LSTMs generally deal with larger sequences due to their ability of capturing the context of the data. (V. Padia, 2020)

OBSERVATION: The LSTM took ~15 minutes to complete the execution, while the GRU model completed its execution in ~11 minutes. This is understandable, because in comparison to the LSTM model, the GRU model has fewer gates and lesser tensor’s operation (A. Jaiswal, 2022) which makes it less complex and more faster than LSTM.

LSTMs and GRUs are variants of RNNs, that use gates to solve the vanishing and exploding gradients problem faced by regular RNN models. Each of the LSTM and GRU models consist of multiple gates that are used for controlling what input data information to store and what to discard. Using this mechanism, the LSTM and GRU models solve the issue faced by regular RNN models. (A. Jaiswal, 2022)

Differences between LSTM and GRU

OBSERVATION: (A. Jaiswal, 2022)

- Unlike the LSTM network, GRUs don’t have cell states that are used for carrying input data information from the first to the subsequent time steps.
- GRUs contain two gates (Update and Reset gates), while LSTMs consists of three gates (Forget, Input, and Output gates).
- As seen from our observation, GRUs execute much faster than LSTMs and take lesser training time and memory.

While there are some distinctions between LSTMs and GRUs, the results they produce are relatively similar.

6. Topic Modelling of High and Low Ratings

Topic modelling refers to employing algorithms to **extract possibly hidden themes/topics** from a set of documents. Topics identified in this way save humans the trouble of having to go through a possibly large no of documents in order to pick out most common/important themes thereby hastening the process of understanding general content of documents. Moreover, topics and associated words thus identified may reveal patterns/trends that can otherwise go unnoticed. Since for most NLP tasks, each word is considered a feature, there is often an overwhelmingly large no. of features which make ML tasks complicated and resource consuming. Thus, feature/dimensionality reduction is necessary when modelling text for ML. Here too, topic modelling may be of aid, since it can reveal important/common themes based on which features may be filtered. (M. Manthiramoorthi, no date)

Various kinds of topic modeling algorithms exists that follow both statistical language modelling approaches as well as linear algebra based language modelling approaches. Algorithms like LDA and LSA, that follow a **probabilistic/statistical** approach, use statistical tools to perform topic modelling and may do so based on probability distribution identified over word sequences (AI - All in One, 2016). Algorithms like NMMF that follow **linear algebra based** approach, involve representing documents as matrices and factorizing them into matrices that capture relations between the document, its terms and topics. (A. Klos, 2020)

Some of the various algorithms that may be used for topic modelling are listed below.

1. Latent Dirichlet Allocation (LDA)
2. Non Negative Matrix Factorization (NMF)
3. Latent Semantic Analysis (LSA)
4. Parallel Latent Dirichlet Allocation (PLDA)
5. Pachinko Allocation Model (PAM)

(M. Manthiramoorthi, no date)

Among algorithms listed above, the most popular and frequently used one is **Latent Dirichlet Allocation (LDA)**. LDA follows a Bayesian approach and makes use of statistical probabilities to identify latent/hidden topics in a corpus (cogitotech, 2021). It fine tunes topics using a reverse engineering strategy.

The following cells explain the intuition behind LDA at the end of which, the reverse engineering aspect of this algorithm will also be more apparent. (Serrano Academy, 2020)

(the no. of topics must be provided since there is no way of knowing beforehand the ideal no. of topics, thus making it a hyperparameter)

LDA computes 2 important probabilities.

1. Probability of a document being related to each of the topics is obtained from a Document-Topic Dirichlet Distribution. (referred to here as document-topic probability)
2. Probability of a Topic containing each of the terms is obtained from a Topic-Term Dirichlet Distribution. (referred to here as topic-term probability)

Here, both Dirichlet Distributions have $\alpha < 1$. Taking the Document-Topic Dirichlet Distribution as an example, a Dirichlet Distribution having $\alpha < 1$ can be understood simply as a kind of Dirichlet Distribution where the probability of a document to be speaking of 1 topic is greater than for it to be speaking of 2 topics which is still greater than for it to be speaking of more than 2 topics and so on.

A multinomial distribution of topics is created from obtained document-topic probabilities. From this multinomial distribution, a "list of topics" is generated.

For each possible topic, topic-term probabilities are obtained from which a multinomial distribution of terms is generated. Terms from generated multinomial distribution of terms per topic is selected for each corresponding topic in generated list of topics. Terms so obtained are combined together to obtain a "generated document".

The probability of this generated document being similar to the original provided document is determined. Choices that led to probability distributions derived from the Dirichlet Distributions leading to generated documents which are more similar to original ones are favoured over others.

On a grand scale, several such generated documents are produced to emulate the corpus and its similarity to the original provided corpus is computed based on which parameters are adjusted to obtain topics that are closer to ones actually present in the provided set of documents.

The intuition behind LDA is explained very well by (Serrano Academy, 2020).

OBSERVATION: Since LDA is a most popular topic modelling algorithm that can identify topics from a set of documents using statistical methods fairly quickly, it was decided to use it to identify 20 topics among 5-star and 1-star reviews separately to investigate common themes/trends/reasons for rating.

OBSERVATION: pyLDAvis is an open source library that allows interactive visualization of topics that LDA identified (Sharma, 2021) and thus, shall be used here.

Note: Following points define what elements of the visualization generated by pyLDAvis represents.

(W. Zeng, 2017)

On the left side of the visualization is an "Intertopic Distance Map" which can be further understood as follows.

- Each circle represent a topic.
- The area of each circle represents "topic prevalence".
- The indices of the circle indicate the sorted order of the topics in terms of popularity (1 being most and 10 being least popular).
- Distance between 2 circles show approximate topic similarity.

On the right side of the visualization is a list of the "Top-30 Most Salient Terms" when no topic circle is selected and a list of the "Top-30 Most Relevant Terms" for selected topic. This list visualization can be further understood as follows.

- Red bars indicate term frequency of a term in the selected topic.
- Blue bars indicate term frequency of a term among all documents.
- As lambda value decreases, words more uniquely popular within chosen topic is selected.
- As lambda value increases, words frequent in chosen topic that's also more generally occurring in other topics is selected.
- Hovering over a term shows the "conditional topic distribution" (how other topics use this word) for the given word indicated via size of the circle of chosen topic.

6.1. Investigating 5 Star Rated Reviews

In [10]:
```
# get 5 star reviews
reviews5star = pd.read_csv("./data/train_processed.csv")
reviews5star = reviews5star[reviews5star["Score"] == 5]["Review_Process
reviews5star = reviews5star.reset_index(drop=True)
reviews5star
```

Out[10]:
```
0         receive product early seller tastey great mid ...
1           numi collection assortment melange include h...
2         careful overcook pasta make sure take bite eve...
3         buy multi pack mislead picture whole hazel nut...
4         bar good love warm definitely think great snac...
                                ...
272118    treat excellent training dog love safe peanut ...
272119    buy store hard time keep stockmy dog already h...
272120    glad company make product without gmo magnesiu...
272121    eat lot syrup house three year old favorite fo...
272122    buy give dog need lose weight get hungry cut b...
Name: Review_Processed, Length: 272123, dtype: object
```

In [11]:
```python
import pyLDAvis
import pyLDAvis.sklearn
pyLDAvis.enable_notebook()
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
```

In [12]:
```python
# convert text to document - term frequency matrix
tf_vectorizer = CountVectorizer(ngram_range=(2,2))
doc_term_freq_mat = tf_vectorizer.fit_transform(reviews5star)
```

WARNING: Time consuming cell ahead! (upto 4 mins)

In [13]:
```python
# identify 10 topics using LDA
lda = LatentDirichletAllocation(n_components=20, random_state=23, n_job
lda.fit(doc_term_freq_mat)
```

Out[13]:
```
LatentDirichletAllocation(n_components=20, n_jobs=-1, random_state=23)
```

In [14]:
```python
# visualize topics
pyLDAvis.sklearn.prepare(lda, doc_term_freq_mat, tf_vectorizer)
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\sklearn\utils
eprecation.py:87: FutureWarning:

Function get_feature_names is deprecated; get_feature_names is deprecated
in 1.0 and will be removed in 1.2. Please use get_feature_names_out instea
d.

C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\pyLDAvis\_prepa
re.py:246: FutureWarning:

In a future version of pandas all arguments of DataFrame.drop except for t
he argument 'labels' will be keyword-only.
```

Out[14]:

OBSERVATION: Topics that were identified to be distinct or fairly distinct are as below.

- `Topic 1` : This topic exclusively talks about tea!
- `Topic 2` : This topic is interesting as although it talks about baked goods in general it sheds special focus on gluten free baked/baking goods products that people really seem to appreciate.
- `Topic 3` : This topic almost exclusively concerns pet food.
- `Topic 5` : This topic has a general focus on peanut/nut butter or just butter in general along with mentions of reaction s of the body to certain foods which may stem from the fact that many people are allergic to nuts.
- `Topic 6` : This topic has to do with products that help with relaxation/better sleep of both children as well as adults.
- `Topic 7` : This topic seems to be related to vegan/organic food.
- `Topic 8` : This topic is a niche topic which focuses on female reproductive health products and related food terms.
- `Topic 10` : This topic is related to reduced price (cheap food) or reduced value in terms of nutrition or health.

- `Topic 11` : This topic seems to be related to healthy snack foods with a focus on products by the "Blue Diamond Almonds" food company as well as products that dogs love.
- `Topic 12` : This topic is related to pet health. Topic 12 is a bit close to topic 3 because they both are related to pets. Topic 12 is not very close to topic 3 because while topic three is about pet food only, topic 12 focuses mainly on pet (dog) food/pills and health.
- `Topic 13` : This topic almost exclusively talks about coconut and oils along with words that may be linked with their health benefits.
- `Topic 17` : This topic almost exclusively talks about body weight/image related terms like exercise as well as foods that are low fat or low calorie.
- `Topic 20` : This topic almost exclusively contains terms related to pregnancy.

OBSERVATION: Following are topics that did'nt seem to have easily discernable topics and seemed to consist of many overlapping themes.

- `Topic 4` : Foods that help soothe negative reactions to food like allergies or acidity in both adults and children.
- `Topic 9` : This topic is mostly a blend of many other topics and does'nt seem to have strong unique elements in it.
- `Topic 14` : This topic is not very distinct. It is simply a blend of different topics related to health, medicine and dessert.
- `Topic 15` : Like topic 14, topic 15 is also just a mix of terms related to health or products for mothers and does'nt contribute to new categories.
- `Topic 16` : Topic 16 also talks about drinks (coffee or herbal drinks) but is far away from topic 1 because it focusses more on wellbeing and relaxation making it rightly closer to topic 7.
- `Topic 18` : This topic is very similar to topic 12 and has words related to pet health.
- `Topic 19` : This topic is a blend of various terms and does not seem to have a easily discernable theme.

OBSERVATION: From the way the 20 topics are clustered, it can be observed that there are likely 3 main categories of food items that people love and thereby likely prefer to get from online stores. LDA with no. of topics 3 can be done to confirm this.

WARNING: Time consuming cell ahead! (upto 4 mins)

In [15]:
```
# identify 3 topics using LDA
lda = LatentDirichletAllocation(n_components=3, random_state=23, n_jobs
lda.fit(doc_term_freq_mat)
```

Out[15]:
```
LatentDirichletAllocation(n_components=3, n_jobs=-1, random_state=23)
```

In [16]:
```
# visualize topics
pyLDAvis.sklearn.prepare(lda, doc_term_freq_mat, tf_vectorizer)
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\sklearn\utils
eprecation.py:87: FutureWarning:

Function get_feature_names is deprecated; get_feature_names is deprecated
in 1.0 and will be removed in 1.2. Please use get_feature_names_out instea
d.

C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\pyLDAvis\_prepa
re.py:246: FutureWarning:

In a future version of pandas all arguments of DataFrame.drop except for t
he argument 'labels' will be keyword-only.
```

Out[16]:

OBSERVATION: As expected, there are indeed 3 clear topics w.r.t. online food products that people really appreciate.

- `Topic1` : Tea and coffee.
- `Topic2` : Baked/baking products, healthy snacks (like nuts), products for special dietary requirements (like gluten free, whole wheat)
- `Topic3` : Pet food.

6.2. Investigating 1 Star Rated Reviews

In [17]:
```
# get 1 star reviews
reviews1star = pd.read_csv("./data/train_processed.csv")
reviews1star = reviews1star[reviews1star["Score"] == 1]["Review_Process
reviews1star = reviews1star.reset_index(drop=True)
reviews1star
```

Out[17]:
```
0        first coffee try get keurig disappointed flavo...
1        buy large german shepherd cut piece trouble ea...
2        want sugar splurge choose shortbread mistake p...
3        chinese never bring product low quality one ev...
4        nearly twice expensive cost nespresso site eve...
                               ...
39386    might miss nutrtional information warn low car...
39387    recieve pack get pay get pack right check send...
39388    wife today buy oz pkg grocery story cost seem ...
39389    poor excuse gopher trap lot gopher use trap re...
39390    buy juice call speak john davis customer servi...
Name: Review_Processed, Length: 39391, dtype: object
```

In [18]:
```
# convert text to document - term frequency matrix
tf_vectorizer = CountVectorizer(ngram_range=(2,2))
doc_term_freq_mat = tf_vectorizer.fit_transform(reviews1star)
```

In [19]:
```
# identify 20 topics using LDA
lda = LatentDirichletAllocation(n_components=20, random_state=23, n_job
lda.fit(doc_term_freq_mat)
```

Out[19]:
```
LatentDirichletAllocation(n_components=20, n_jobs=-1, random_state=23)
```

In [20]:
```
# visualize topics
pyLDAvis.sklearn.prepare(lda, doc_term_freq_mat, tf_vectorizer)
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\sklearn\utils
eprecation.py:87: FutureWarning:

Function get_feature_names is deprecated; get_feature_names is deprecated
in 1.0 and will be removed in 1.2. Please use get_feature_names_out instea
d.

C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\pyLDAvis\_prepa
re.py:246: FutureWarning:

In a future version of pandas all arguments of DataFrame.drop except for t
he argument 'labels' will be keyword-only.
```

Out[20]:

OBSERVATION: Topics that were identified to be distinct or fairly distinct are as below.

- `Topic 1` : This topic revolves around the action of eating, hunger or diet with a strange relation to cats.
- `Topic 2` : This topic concerns meat/chicken and leftover food (veg or non-veg).
- `Topic 4` : This topic is suggestive of bad taste.
- `Topic 5` : This topic is related to digestive issues and fruit.
- `Topic 6` : This topic is related to coffee.
- `Topic 7` : This topic seems to revolve around hexane being a part of many vegan products and even a part of pet food.
- `Topic 8` : This topic seems to revolve around carcinogens.
- `Topic 9` : This topic seems to be connected to traces of drugs or other additives in food.
- `Topic 10` : This topic revolves around food being stale or having faulty packaging.
- `Topic 11` : This topic mainly concerns cost.
- `Topic 12` : This topic concerns poultry products and ailments.
- `Topic 15` : This topics refers to foul flavor.
- `Topic 18` : This topic is concerned with pet foods.
- `Topic 20` : This topic is concerned with flavor of valerian tea.

OBSERVATION: Following are topics that did'nt seem to have easily discernable topics and seemed to consist of many overlapping themes.

- `Topic 3` : This topic does not contain unique terms that follow any particular discernable theme, though more generally, it seems to be concerned with foul taste.
- `Topic 13` : This topic though contains words associated with pork or gelatin, it also contains various terms that don't seem to have much connection with each other.
- `Topic 14` : This topics is very similar to 15 but also contains other rogue terms.

- `Topic 16` : This topic also talks about just foul flavor with a slight focus on artificial sweetness.
- `Topic 17` : Ths topic also refers to bad taste with a focus on excessive spice.
- `Topic 19` : This topic like 18 is also concerned with pet foods.

OBSERVATION: From the way the 20 topics are clustered, it can be observed that there are also likely 3 main categories of food items that online shoppers had a negative experience with. LDA with no. of topics 3 can be done to confirm this.

In [21]:
```
# identify 3 topics using LDA
lda = LatentDirichletAllocation(n_components=3, random_state=23, n_jobs
lda.fit(doc_term_freq_mat)
```

Out[21]:
```
LatentDirichletAllocation(n_components=3, n_jobs=-1, random_state=23)
```

In [22]:
```
# visualize topics
pyLDAvis.sklearn.prepare(lda, doc_term_freq_mat, tf_vectorizer)
```

```
C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\sklearn\utils
eprecation.py:87: FutureWarning:

Function get_feature_names is deprecated; get_feature_names is deprecated
in 1.0 and will be removed in 1.2. Please use get_feature_names_out instea
d.

C:\Users\Gayathri Girish Nair\miniconda3\lib\site-packages\pyLDAvis\_prepa
re.py:246: FutureWarning:

In a future version of pandas all arguments of DataFrame.drop except for t
he argument 'labels' will be keyword-only.
```

Out[22]:

OBSERVATION: There are indeed 3 broad topics as expected, 2 of which are clear.

- `Topic1` : This topic is less distinct than the other 2 topics. More than indicating a particular reason for poor rating of food, this topic seems to contain words related to the action of eating/feeding, food/diet in general and water. This topic also seems to be hinting at poor customer service.
- `Topic2` : Inorganic additives and artificial flavoring.
- `Topic3` : Stale food with a focus on meat.

6.3. Topic Conclusions

There are a couple of observations we can make by looking at the 40 positive and negative topics generated. Some of them are given below:

- Food products that customers love to purchase online are tea, coffee, baked/baking goods, pet food (mostly dry food) and snacks.
- It can be observed that easily perishable foods such as meat (also wet pet food) or veggies were what was reported by customers as stale or having foul odour. This reveals that online delivery companies may want to consider improving their food

transport/storage facilities to keep perishable foods fresher longer. Adding additives to food to keep them fresher longer may not be the best strategy since customers today are very aware of food additives as evident from potentially harmful additives being the cause of a lot of foods being rated 1.

- It is to be noted that online buyers are largely health aware and highly appreciate healthy foods (organic, nutrient dense, vegan) and are fairly vocal when it comes to providing negative reviews when products lack nutrition or contain harmful ingredients even in the case of pet food. It is in best interest of online retailers to sell nutritious food items with natural quality flavors.
- Pet food seems to be amongst the most popular online food grade commodity thus, retailers may want to maintain more stock of such products. However, they must be maintained without them spoiling as pet owners are critical about feeding their fur babies quality food. This is especially critical since pets (cats, dogs) are known to have sensitive stomachs as confirmed from the many gastointestinal issues reported as a consequence of stale or contaminated pet food.
- An interesting observation was that foods/ingredients that soothe gastrointestinal ailments, lessen sleeplessness/anxiety or helps manage difficulties w.r.t. female reproductive system is highly in demand despite there being few complaints regarding the taste of such products such as “valerian tea” being foul. Online retailers may thus be interested in expanding their inventory w.r.t. such food products since it is understandable that buyers with such ailments would prefer to order online as compared to physically purchase items under limitations that ailments may impose.
- New mothers (likely due to their busy schedule) form a significantly large section of online food shoppers (food for baby/toddler) and highly appreciate good nutritious products for kids.
- An interesting observation was that “sugar rich” and “junk” food appeared amongst both 5 rated as well as 1 rated reviews. This is likely because the taste of processed foods with high sugar, fat and salt while not healthy, is still highly palatable. Thus, the task of striking a balance between “junk” and “nutritious” food surfaces as a challenge that retailers must take on. A good idea may be to make available tastier nutritious food options.
- Some of the negative reviews also seem to be about the customer service (like packaging) which could be a point of improvement that the online store could investigate.
- Finally, w.r.t. price of food, the views are conflicting. While some positive reviews hint at good value for money, it cannot be ignored that a many negative reviews point to overpriced low quality purchases as the reason for assigning a low rating. It is suggestive that retailers may be trying to provide fair prices as they should. However, they may also want to bear in mind that from analysis of positive vs negative reviews, it can be inferred that value for money is just as important as availability of cheaper products. This can be confirmed from the fact that while generally, a cheap product is highly appreciated (from positive reviews), the appreciation is received if and only if the food product meets minimum food grade quality in terms of safely, nutrition and taste as any food product (even cheap ones) that do not meet minimum standards are deemed a wasteful expenditure/mistake no matter how low its price was (from negative reviews).

References

A

- Alex Klos. (2020) *Topic modeling: LDA vs. NMF for newbies*. Link: https://alexklos.ca/blog/natural-language-processing-lda-vs-nmf-for-newbies/ (https://alexklos.ca/blog/natural-language-processing-lda-vs-nmf-for-newbies/). [Last Accessed: 24/03/2022]
- Angelica Lo Duca. (2021) How to Deal with Imbalanced Multiclass Datasets in Python, Medium. Link: https://towardsdatascience.com/how-to-deal-with-imbalanced-multiclass-datasets-in-python-fe0bb3f2b669 (https://towardsdatascience.com/how-to-deal-with-imbalanced-multiclass-datasets-in-python-fe0bb3f2b669) [Last Accessed 21 Mar 2022]
- Artificial Intelligence - All in One. (9 Apr 2016) *Lecture 17 — Probabilistic Topic Models Overview of Statistical Language Models - Part 1 | UIUC*, YouTube Video. Link: https://www.youtube.com/watch?v=znTOwpBHlbE (https://www.youtube.com/watch?v=znTOwpBHlbE). [Last Accessed: 24/03/2022]
- Abhishek Jaiswal. (2022) *Tutorial on RNN | LSTM |GRU with Implementation*, Analytics Vidhya. Link: https://www.analyticsvidhya.com/blog/2022/01/tutorial-on-rnn-lstm-gru-with-implementation/ (https://www.analyticsvidhya.com/blog/2022/01/tutorial-on-rnn-lstm-gru-with-implementation/) [Last Accessed 25 Mar 2022]

B

- Brownlee, J. (2020). One-vs-Rest and One-vs-One for Multi-Class Classification. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/ (https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/) [Accessed 26 Mar. 2022].

C

- cogitotech. (27 Sep 2021) *Topic Modeling: Algorithms, Techniques, and Application*, datasciencecentral.com. Link: https://www.datasciencecentral.com/topic-modeling-algorithms-techniques-and-application/ (https://www.datasciencecentral.com/topic-modeling-algorithms-techniques-and-application/). [Last Accessed: 24/03/2022]

D

- DecisionForest. (06 Sep 2020) *ML Classification using GloVe Vectors & Keras ❌NLP Project in Python with GloVe, TensorFlow & Keras*, YouTube Video. Link: https://www.youtube.com/watch?v=Qsmn9pL5kcU&t=1141s (https://www.youtube.com/watch?v=Qsmn9pL5kcU&t=1141s) [Last Accessed: 23/03/2022] (DecisionForest, 2020)

G

- Great Learning Team (2020). Multinomial Naive Bayes Explained. [online] GreatLearning Blog. Available at: https://www.mygreatlearning.com/blog/multinomial-naive-bayes-explained/ (https://www.mygreatlearning.com/blog/multinomial-naive-bayes-explained/) [Accessed 27 Mar. 2022].

H

- Hale, J. (2019). Don’t Sweat the Solver Stuff - Towards Data Science. [online] Medium. Available at: https://towardsdatascience.com/dont-sweat-the-solver-stuff-aea7cddc3451 (https://towardsdatascience.com/dont-sweat-the-solver-stuff-aea7cddc3451) [Accessed 28 Mar. 2022].

I

- IBM Cloud Education. (14 Sep 2020) "Recurrent Neural Networks", ibm.com. Link: https://www.ibm.com/cloud/learn/recurrent-neural-networks#toc-variant-rn-2xvhb_yi (https://www.ibm.com/cloud/learn/recurrent-neural-networks#toc-variant-rn-2xvhb_yi). [Last Accessed: 23/03/2022]

J

- Jabeen, H. (2018). Stemming and Lemmatization in Python. [online] DataCamp Community. Available at: https://www.datacamp.com/community/tutorials/stemming-lemmatization-python (https://www.datacamp.com/community/tutorials/stemming-lemmatization-python)

K

- Karsten Eckhardt. (29 Nov 2018) *Choosing the right Hyperparameters for a simple LSTM using Keras*, towardsdatascience.com. Link: https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046 (https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046). [Last Accessed: 23/03/2022]

L

- Lyashenko, V. and Jha, A. (2022). Cross-Validation in Machine Learning: How to Do It Right - neptune.ai. [online] Neptune Blog. Available at: https://neptune.ai/blog/cross-validation-in-machine-learning-how-to-do-it-right (https://neptune.ai/blog/cross-validation-in-machine-learning-how-to-do-it-right) [Accessed 28 Mar. 2022].

M

- Manikanth. (2021) *What is the use of data standardization and where do we use it in machine learning*, Analytics Vidhya, mdeium.com. Link: https://medium.com/analytics-vidhya/what-is-the-use-of-data-standardization-and-where-do-we-use-it-in-machine-learning-97b71a294e24 (https://medium.com/analytics-vidhya/what-is-the-use-of-data-standardization-and-where-do-we-use-it-in-machine-learning-97b71a294e24). [Last Accessed: 17 Mar 2022]
- Murugesh Manthiramoorthi. (no date) *Topic Modelling Techniques in NLP*, OpenGenus IQ: Computing Expertise & Legacy. Link: https://iq.opengenus.org/topic-modelling-techniques/ (https://iq.opengenus.org/topic-modelling-techniques/). [Last Accessed: 24/03/2022]

R

- Rajat Sharma. (2019) *Skewed Data: A problem to your statistical model*, towardsdatascience.com. Link: https://towardsdatascience.com/skewed-data-a-problem-to-your-statistical-model-9a6b5bb74e37 (https://towardsdatascience.com/skewed-data-a-problem-to-your-statistical-model-9a6b5bb74e37). [Last Accessed: 17 Mar 2022]
- Rakshith Vasudev. (2017) *How to deal with Skewed Dataset in Machine Learning?*, Becoming Human: Artificial Intelligence Magazine. Link: https://becominghuman.ai/how-to-deal-with-skewed-dataset-in-machine-learning-afd2928011cc (https://becominghuman.ai/how-to-deal-with-skewed-dataset-in-machine-learning-afd2928011cc). [Last Accessed: 17 Mar 2022]
- ravi (2018). Finding top features with SGD classifier & GridsearchCV. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/52644634/finding-top-features-with-sgd-classifier-gridsearchcv (https://stackoverflow.com/questions/52644634/finding-top-features-with-sgd-classifier-gridsearchcv) [Accessed 26 Mar. 2022].

S

- Serrano Academy. (10 Mar 2020) *Latent Dirichlet Allocation (Part 1 of 2)*, YouTube Video. Link: https://www.youtube.com/watch?v=T05t-SqKArY (https://www.youtube.com/watch?v=T05t-SqKArY) [Last Accessed: 24/03/2022]
- scikit-learn. (2022). 1.5. Stochastic Gradient Descent. [online] Available at: https://scikit-learn.org/stable/modules/sgd.html#classification (https://scikit-learn.org/stable/modules/sgd.html#classification) [Accessed 26 Mar. 2022].
- scikit-learn. (2022b). sklearn.naive_bayes.MultinomialNB. [online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html) [Accessed 28 Mar. 2022].
- spaCy. (2022b). spaCy 101: Everything you need to know. [online] Available at: https://spacy.io/usage/spacy-101 (https://spacy.io/usage/spacy-101) [Accessed 27 Mar. 2022].
- Srinidhi, S. (2020). Lemmatization in Natural Language Processing (NLP) and Machine Learning. [online] Medium. Available at: https://towardsdatascience.com/lemmatization-in-natural-language-processing-nlp-and-machine-learning-a4416f69a7b6 (https://towardsdatascience.com/lemmatization-in-natural-language-processing-nlp-and-machine-learning-a4416f69a7b6) [Accessed 27 Mar. 2022].

T

- TR517 (2018). How to speed up spaCy lemmatization? [online] Stack Overflow. Available at: https://stackoverflow.com/questions/51372724/how-to-speed-up-spacy-lemmatization (https://stackoverflow.com/questions/51372724/how-to-speed-up-spacy-lemmatization) [Accessed 24 Mar. 2022].

V

- Vivek Padia. (2020) *Scratching surface of RNN, GRU, and LSTM with example of sentiment analysis*, Medium. Link: https://medium.com/aubergine-solutions/scratching-surface-of-rnn-gru-and-lstm-with-example-of-sentiment-analysis-8dd4e748d426 (https://medium.com/aubergine-solutions/scratching-surface-of-rnn-gru-and-lstm-with-example-of-sentiment-analysis-8dd4e748d426) [Last Accessed 26 Mar 2022]

W

- William Zeng. (21 Nov 2017) *Topic modeling using Python and pyLDAvis: part2*, YouTube Video. Link: https://www.youtube.com/watch?v=SF50IK5XgKA (https://www.youtube.com/watch?v=SF50IK5XgKA). [Last Accessed: 23/03/2022]