University of Salford, MSc Data Science

Module: Machine Learning & Data Mining

Session: Workshop Week 10

Topic: Text Mining & Sentiment Analysis

Tools: Google Colab

Objectives:

After completing this workshop, you will be able to:

- Use NLTK to carry out tokenization, stopword removal, stemming and lemmatization
- Construct a Term Frequency Matrix from pre-processed text
- Use a Term Frequency Matrix to train a Naïve Bayes classifier
- Use the NLTK VADER SentimentIntensityAnalyzer to conduct sentiment analysis
- Use the wordcloud package to visualise word frequencies

MLDM Week 10 Workshop: Text Mining & Sentiment Analysis

University of Salford MANCHESTER

Part One: Bag-of-words Model & Text Classification

Introduction:

Text mining encompasses a vast field of theoretical approaches and methods with one thing in common: text as input information. In general, text mining is an interdisciplinary field of activity amongst data mining, linguistics, computational statistics, and computer science. Standard techniques are text classification, text clustering, ontology and taxonomy creation, document summarization and latent corpus analysis. In addition, a lot of techniques from related fields like information retrieval are commonly used.

Text mining involves a series of activities to be performed in order to efficiently mine the information. One of the simplest approaches to text mining is the bag-of-words model, which disregards grammar, sentence structure and word order and primarily considers word frequencies. Typically, the steps involved in pre-processing data for a bag-of-words model include:

- Collecting documents into a corpus (a *corpus* is a collection of documents)
- Tokenization: This involves splitting the text into separate parts, called tokens.
- Stopword, punctuation and special character removal: Stopwords are words such as articles, prepositions etc (e.g., 'at', 'in') which occur frequently but which aren't particularly useful within a Bag-of-Words model. We therefore remove them during pre-processing
- Stemming or lemmatization: Stemming involves reducing words into a root form – this is useful because it reduces the number of unique tokens while preserving semantics (e.g., *watched, watching* and *watch* would all be reduced to *watch*.) Lemmatization is an alternative which identifies the lemma of a word based on

its intended meaning. For example, *cacti* would not be altered if we used stemming, but with lemmatization it would be reduced to *cactus.* However, lemmatization is more difficult in that it requires us to correctly identify the intended part of speech. For example, *meeting* can be a verb or noun (e.g., I am meeting my colleagues at a meeting'), and it should only be replaced by *meet* if it is being used as a verb.

- Converting documents into a fixed length vector based on term frequency: This allows us to construct a Term Frequency Matrix or Term Frequency-Inverse Document Frequency (TF-IDF) Matrix for the entire corpus in question
- Once we have constructed a Term Frequency Matrix or Term Frequency-Inverse Document Frequency (TF-IDF) Matrix, this can be used for classification, clustering, topic modelling etc.

There are of course limitations to using a Bag-of-Words model, and we will come onto this when we cover sentiment analysis. However, for some of the purposes discussed above, a Bag-of-Words model can be a good starting point and achieve a reasonable performance.

The Dataset

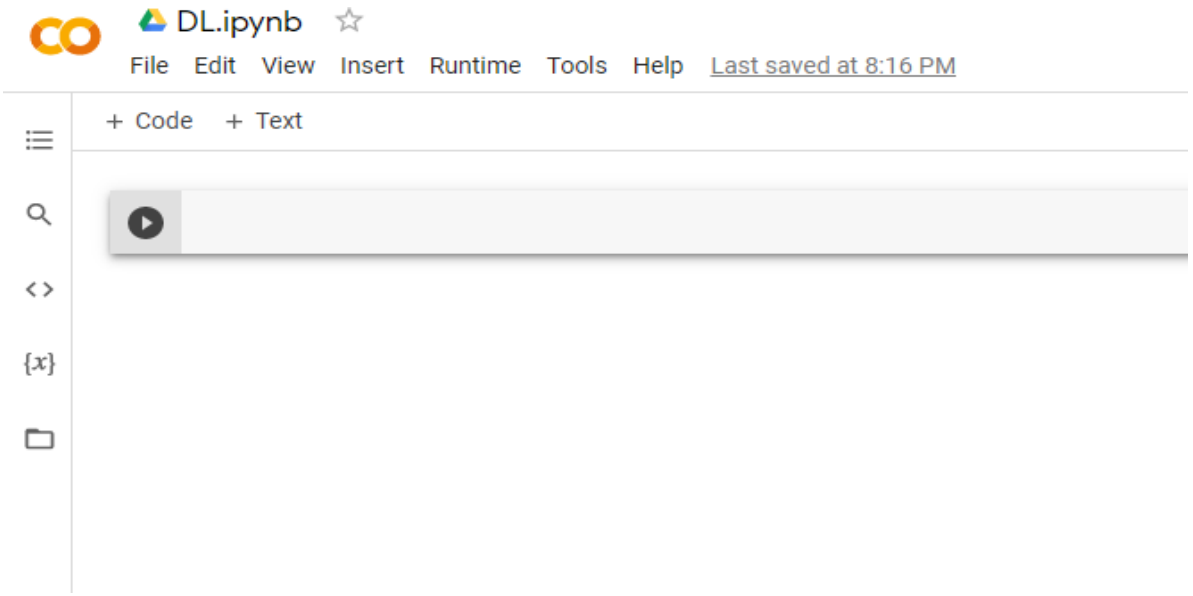
In the first part of this workshop, we are going to demonstrate these pre-processing steps on a collection of SMS messages. The SMS messages are labelled as either 'spam' or 'ham', where 'ham' refers to text messages which are not spam. Our aim is therefore to train a classifier which can identify which messages are spam – a common problem facing those developing anti-spam software.

Once we have built a Term Frequency Matrix, we will train a Naïve Bayes classifier on this dataset. We haven't covered this classification algorithm previously; however, it is commonly used for text classification using a Term Frequency Matrix. A Naïve Bayes classifier uses Bayes theorem, which we covered in the Applied Statistics & Data Visualisation module. You can read more the math behind a Naïve Bayes classifier here.

Getting Started in Google Colab:

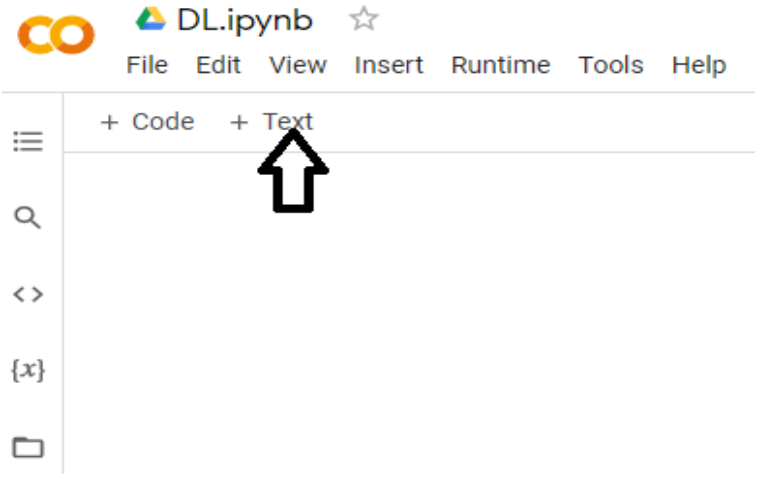
1) To use Google Colab, you will need a Google Account. If you do not have one, you will need to set one up first. Login to your Google Account, and once you are logged in, [follow this link] and click on File>New notebook

You should see the below screen:



You can rename the notebook by clicking on the file name to the top of the screen.

2) You can add two types of cells into your notebook using the +Code and +Text buttons at the top. You can use the text cells as you would in Jupyter notebooks to add comments, headings etc.



3) Add a code cell and run the imports we need. For this workshop, we will be using the NLTK (Natural Language Toolkit) library. NLTK provides a suite of text processing libraries for many text mining and natural language processing tasks, including classification, tokenization, stemming, tagging, parsing and sentiment analysis.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import re
from wordcloud import WordCloud
import nltk
nltk.download(['stopwords',
               'punkt',
               'wordnet',
               'omw-1.4',
               'vader_lexicon'
               ])
```

4) Before we start working with our text data, let us familiarise ourselves with some of the pre-processing steps discussed above, by trying them out on a simple example. Run the below code to define a variable which is a character string for us to pre-process.

```
simple_text='This isn\'t a real text, this is an example text...Notice this contains punctuation!!'
```

5) The first step is tokenization. We will use the RegexpTokenizer from NLTK to do this. This converts our character string into tokens by splitting it into words. In addition, this allows us to define a regular expression, and we can therefore define a regular expression so that we only tokenize alphanumeric characters, this removing punctuation (apart from apostrophes which we want to retain in words such as *don't*)

```
tokenizer = nltk.tokenize.RegexpTokenizer('[a-zA-Z0-9\']+')

tokenized_document = tokenizer.tokenize(simple_text)

print(tokenized_document)
```

```
['This', "isn't", 'a', 'real', 'text', 'this', 'is', 'an', 'example',
```

6) We have now tokenized our example text and inspected it. You should notice that punctuation has been removed, apart from the apostrophe in "isn't". NLTK also provides a corpus of stopwords which we can use to remove stopwords from our tokenized text.

```
stop_words = nltk.corpus.stopwords.words('english')

print(stop_words)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're
```

7) Now we can use a for loop to remove the stopwords from our tokenized text. However, there is a more compact and efficient way of doing this through *list comprehension.* You can read more on list comprehension here. We have shown you both ways below, but you only need to run one of the below code cells. Note, we will be using list comprehension from now on for today's workshop.

```
# Remove stopwords

cleaned_tokens = []

for word in tokenized_document:
    word = word.lower()
    if word not in stop_words:
        cleaned_tokens.append(word)

print(cleaned_tokens)
```

```
['real', 'text', 'example', 'text', 'notice', 'contains', 'punctuation']
```

```
# we can also remove stopwords using list comprehension

cleaned_tokens = [word.lower() for word in tokenized_document if word.lower() not in stop_words]

print(cleaned_tokens)
```

```
['real', 'text', 'example', 'text', 'notice', 'contains', 'punctuation']
```

8) Our next step is to carry out stemming or lemmatization. To demonstrate the differences between the two, run the below code. You can see that lemmatization does better at identifying the lemma of a word (i.e., the dictionary form.) For example, it identifies that *cacti* is the plural of *cactus* and *better* is the comparative form of *good.* However, lemmatization requires us to provide additional information on the *part of*

speech. In practice, therefore, stemming is often easier to implement. Stemming is the process of reducing variants of a word to a root form. Porter's Stemmer is a popular stemming algorithm which performs well compared to other stemming algorithms but has the disadvantage that it doesn't always produce real words. For example, you can see from the below that *easily* becomes *easili.* We will be using Porter's Stemmer in this workshop.

```
# Explore lemmatization vs stemming

lemmatizer = nltk.stem.WordNetLemmatizer()
stemmer = nltk.stem.PorterStemmer()

words = ['cacti','sings','hopped','rocks','better','easily']
pos = ['n','v','v','n','a','r']
lemmatized_words = [lemmatizer.lemmatize(words[i], pos=pos[i]) for i in range(6)]
stemmed_words = [stemmer.stem(word) for word in words]

print("Lemmatized words: ", lemmatized_words)
print("Stemmed words: ", stemmed_words)
```

```
Lemmatized words:  ['cactus', 'sing', 'hop', 'rock', 'good', 'easily']
Stemmed words:  ['cacti', 'sing', 'hop', 'rock', 'better', 'easili']
```

9) Now we can use the stemmer we have just instantiated to stem the example text we are working with:

```
[14] # Now carry out stemming on our example sentence

     stemmed_text = [stemmer.stem(word) for word in cleaned_tokens]

     print(stemmed_text)
```

```
['real', 'text', 'exampl', 'text', 'notic', 'contain', 'punctuat']
```

10) Now we have explored these pre-processing steps, we are going to define a function which performs each of these steps in turn. We can then use this function on the text data we're going to be using for the first part of this workshop.

```
# Lets now create a function to apply all of our data preprocessing steps which we can then use on a corpus

def preprocess_text(text):
    tokenized_document = nltk.tokenize.RegexpTokenizer('[a-zA-Z0-9\']+').tokenize(text) # Tokenize
    cleaned_tokens = [word.lower() for word in tokenized_document if word.lower() not in stop_words] #Remove
    stemmed_text = [nltk.stem.PorterStemmer().stem(word) for word in cleaned_tokens] # Stemming
    return stemmed_text
```

11) Now we can start to work with our SMS Spam Collection text. This is saved as **SMSSpamCollection.txt** on Blackboard. You will need to download this and then upload this to the session storage within Google Colab. Remember, our aim here is to build a classifier which we can use to predict whether a given SMS message is spam or not. We will start by reading this into a Pandas Data Frame and using head() to view the first few lines:

```
data = pd.read_csv("SMSSpamCollection.txt",sep="\t",header=None)
data.columns = ["Target_Label","Text"]

data.head()
```

	Target_Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

12) There are two columns, one which includes the class label and the other which includes the text itself. Since we want to train a classification model, we should check for class imbalance.

```
print("\n All Data Labels")
print(data.groupby("Target_Label").count())
```

```
 All Data Labels
              Text
Target_Label
ham           4825
spam           747
```

We can see we have imbalanced classes – we will deal with this later!

13) The next step is to use the function we defined earlier to carry out the pre-processing steps we need for a Bag-of-Words model. We can then use head() again to check this has worked:

```
data['Text'] = data['Text'].apply(preprocess_text)

data.head()
```

	Target_Label	Text
0	ham	[go, jurong, point, crazi, avail, bugi, n, gre...
1	ham	[ok, lar, joke, wif, u, oni]
2	spam	[free, entri, 2, wkli, comp, win, fa, cup, fin...
3	ham	[u, dun, say, earli, hor, u, c, alreadi, say]
4	ham	[nah, think, goe, usf, live, around, though]

14) The next step is to generate our Term Frequency Matrix. We can do this using CountVectorizer in Scikit Learn. This takes our tokenized text and vectorizes it – we can then use this to create a new Data Frame which will be our features for training. We can use head() to view the first few rows of this new Data Frame.

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
X=vectorizer.fit_transform(data['Text'].map(''.join))
X=pd.DataFrame(X.toarray())
X.head()
```

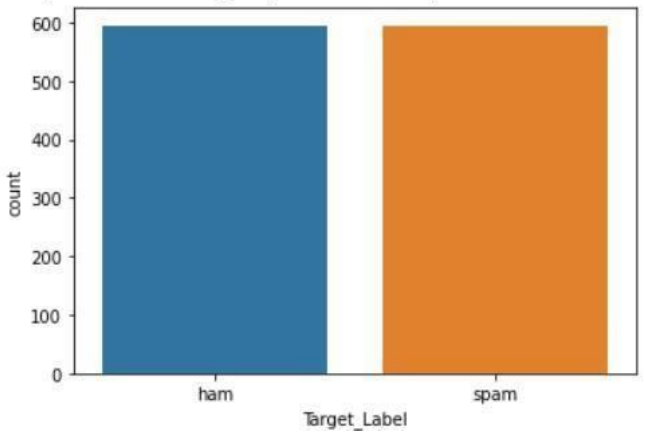
	0	1	2	3	4	5	6	7	8	9	...	5941	5942	5943	5944	5945	5946	5947	5948	5949	5950
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

15) Now we have the features we are going to use to train our classification model, we can use train_test_split to split the data into the training and test datasets, and then use the imblearn library to carry out undersampling to provide a balanced dataset.

```
from sklearn.model_selection import train_test_split

y = data['Target_Label']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=0.8,test_size=0.2,random_state=99)
```

```
[35] from imblearn.under_sampling import RandomUnderSampler

resampler = RandomUnderSampler(random_state=0)
X_train_undersampled, y_train_undersampled = resampler.fit_resample(X_train, y_train)

sns.countplot(x=y_train_undersampled)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593fd9f050>
```



16) We have our data ready for training. Next, we should instantiate the model we are using from Scikit Learn. This is the MultinomialNB (Multinomial Naïve Bayes) classifier, one of the two classic naïve Bayes variants used for text classification (see more in the Scikit Learn documentation here.) As with any other classification model, we can then use the fit method to train the model on the training dataset.

```
[36] from sklearn.naive_bayes import MultinomialNB

model = MultinomialNB()
model.fit(X_train_undersampled, y_train_undersampled)
```

```
MultinomialNB()
```

17) Now we have trained the model, we can evaluate it in the same way as we have evaluated other machine learning models.

```
[37] y_pred = model.predict(X_test)

     # Computing the accuracy and Making the Confusion Matrix
     from sklearn import metrics
     acc=metrics.accuracy_score(y_test,y_pred)
     print('accuracy:%.2f\n\n'%(acc))
     cm = metrics.confusion_matrix(y_test, y_pred)
     print("Confusion Matrix:")
     print(cm,'\n\n')
     print('-----------------------------------------------------------')
     result = metrics.classification_report(y_test, y_pred)
     print("Classification Report:\n",)
     print (result)
```

```
accuracy:0.94


Confusion Matrix:
[[898  65]
 [  5 147]]


-----------------------------------------------------------
Classification Report:

              precision    recall  f1-score   support

         ham       0.99      0.93      0.96       963
        spam       0.69      0.97      0.81       152

    accuracy                           0.94      1115
   macro avg       0.84      0.95      0.89      1115
weighted avg       0.95      0.94      0.94      1115
```

Part Two: Sentiment Analysis

For the second part of this workshop, we are going to explore sentiment analysis, and we will use it to analyse text data from Amazon reviews. This is saved as **Reviews_of_Amazon_Products.csv** on Blackboard. You will need to download this and then upload this to the session storage within Google Colab.

There are many lexicons of words known to reflect positive or negative sentiment which are publicly available, and which can be used in conjunction with a simple Bag-of-Words model. However, a Bag-of-Words model has serious limitations when trying to analyse sentiment as it does not try to account for meaning, sentence structure etc. For example, try running the below code:

```
[43] print(preprocess_text('This movie is great!'))
     print(preprocess_text('This movie is not great'))
```

```
['movi', 'great']
['movi', 'great']
```

These character strings end up as the same set of tokens once we have pre-processed them, despite the fact they represent very different sentiments! We can of course try to make our Bag-of-Words model more sophisticated (for example, we could change the stopwords which are removed), but ultimately, we really want to have a model which has at least some basic contextual understanding that "*not great*" is a negative sentiment.

One pre-trained model that we can use is implemented within NLTK already. This is VADER (Valence Aware Dictionary for Sentiment Reasoning), a model which we can use to estimate both sentiment *polarity* (whether it is positive or negative) and *intensity* (how strongly positive or negative it is).

Let's try this out on the two statements we used above. Because this model is not a simple Bag-of-Words model, we pass in raw character strings, rather than pre-processed, tokenized text.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

sentiment = SentimentIntensityAnalyzer()

print(sentiment.polarity_scores('This move is great!'))
print(sentiment.polarity_scores('This move is not great'))
```

```
{'neg': 0.0, 'neu': 0.406, 'pos': 0.594, 'compound': 0.6588}
{'neg': 0.452, 'neu': 0.548, 'pos': 0.0, 'compound': -0.5096}
```

You can see that it has correctly picked up that the first statement expresses a positive sentiment, while the second one expresses a negative sentiment.

The SentimentIntensityAnalyzer within NLTK outputs a dictionary with a negative score, a neutral score and a positive score. These sum to 1. The compound score is an overall rating between -1 and 1, with negative numbers suggesting negative sentiment and positive numbers positive sentiment. The closer to 1 or -1, the stronger this sentiment.

We will now apply this to the review data, and then analyse and visualise some of the results.

18) Read the Amazon review data into a Data Frame and use head to view the first few rows.

```
# Now lets read in our review data - we're going to use this for our analysis

reviews = pd.read_csv('Reviews_of_Amazon_Products.csv')

reviews.head()
```

	Category	name	brand	primaryCategories	
0	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics	03T
1	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics	06T
2	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics	20T
3	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics	02T
4	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics	24T

19) Generate a summary of the Data Frame using the describe method

```
reviews.describe()
```

	Category	name	brand	primaryCategories
count	1123	1123	1123	1123
unique	2	7	1	2
top	Tablet	Fire HD 8 Tablet, Wi-Fi, 16 GB- Magenta	Amazon	Electronics
freq	1016	797	1123	1112

20) We have already instantiated the SentimentIntensityAnalyzer above. We can therefore use this to generate polarity scores for the reviews in our dataset. To do this, we use list comprehension to create a new column for each from the dictionaries returned when we use the polarity_scores method.

```
# We can extract the values from the dictionary and create new columns within our dataframe

reviews['compound'] = [sentiment.polarity_scores(review)['compound'] for review in reviews['reviews.text']]
reviews['neg'] = [sentiment.polarity_scores(review)['neg'] for review in reviews['reviews.text']]
reviews['neu'] = [sentiment.polarity_scores(review)['neu'] for review in reviews['reviews.text']]
reviews['pos'] = [sentiment.polarity_scores(review)['pos'] for review in reviews['reviews.text']]
```

21) To check that this has worked as we expected, we can use head() again to inspect the first few rows of our Data Frame. If you scroll right, you can see that we have indeed added four new columns.

```
[50] reviews.head()
```

	Category	name	brand	primaryCategories
0	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics
1	Kindle	Amazon Kindle E-Reader 6" Wifi (8th Generation...	Amazon	Electronics
		Amazon Kindle E-		

22) We can also use the describe() method to get more of an insight into sentiment scores for the review data.

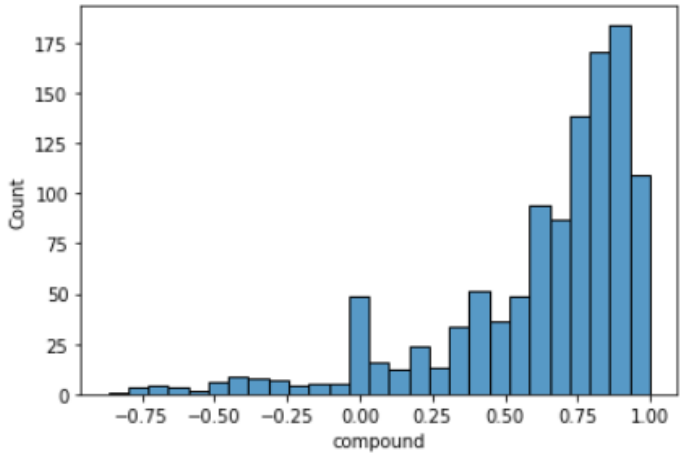
```
[51] reviews[['compound','neg','neu','pos']].describe()
```

	compound	neg	neu	pos
count	1123.000000	1123.000000	1123.000000	1123.000000
mean	0.623324	0.033545	0.694907	0.271563
std	0.347236	0.058432	0.147549	0.152466
min	-0.867400	0.000000	0.230000	0.000000
25%	0.493900	0.000000	0.592000	0.163000
50%	0.750600	0.000000	0.705000	0.259000
75%	0.865800	0.055500	0.800500	0.380000
max	0.999200	0.475000	1.000000	0.689000

It's apparent that the scores are primarily positive. In fact, we can see that the median compound score is 0.75 – which means that over 50% of the reviews have a compound score of more than 0.75, which suggests strong positive sentiment. Let's take a look at the distribution of the compound scores.

```
sns.histplot(reviews['compound'])
```

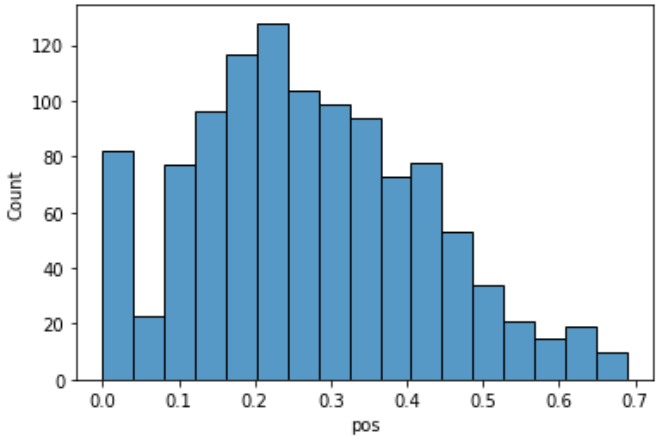
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593e0c1950>
```



23) We can also take a look at the distribution of positive and negative scores too.

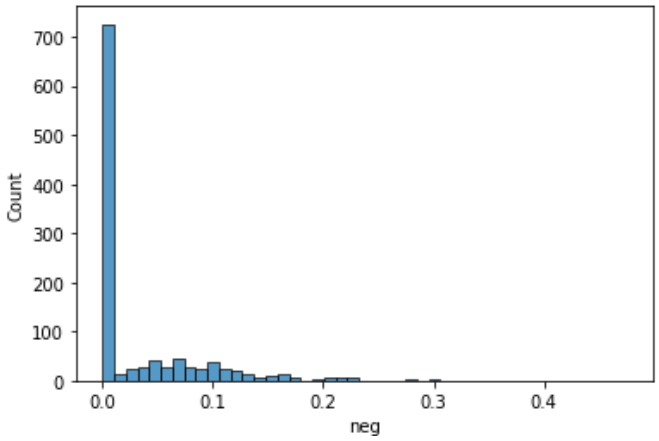
```
[54] sns.histplot(reviews['pos'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593e30e750>
```




```
[55] sns.histplot(reviews['neg'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593e1fa390>
```



23) If we're doing sentiment analysis, we're generally going to be interested in understanding how sentiment varies across different products, brands or businesses. We may also be interested in comparing the proportion of positive or negative reviews across them. In this case we are going to class anything with a compound score of zero or below as negative, although we could take this analysis further by introducing a 'neutral' classification for scores between 0.05 and -0.05. Let's have a look at how many negative reviews there are for each of the products in our dataset.

```
# Lets look at how many negative reviews we have per product

(reviews['compound']<=0).groupby(reviews['name']).sum()
```

```
name
Amazon Kindle E-Reader 6" Wifi (8th Generation, 2016)                                12
Fire HD 8 Tablet,  Wi-Fi, 16 GB-Magenta                                              67
Fire HD 8 Tablet,  Wi-Fi, 32 GB-Black                                                 5
Fire HD 8 Tablet,  Wi-Fi, 32 GB-Magenta                                               5
Fire HD 8 Tablet, Wi-Fi, 16 GB-Blue                                                   1
Fire HD 8 Tablet, Wi-Fi, 32 GB-Blue                                                  10
Kindle E-reader - White, 6 Glare-Free Touchscreen Display, Wi-Fi - Includes Special Offers     1
Name: compound, dtype: int64
```

24) Of course, the number of negative reviews for each product doesn't tell us much unless we know how many reviews there are in total for each one. So, we should look at the number of negative reviews as a proportion of the total number of review for that product.

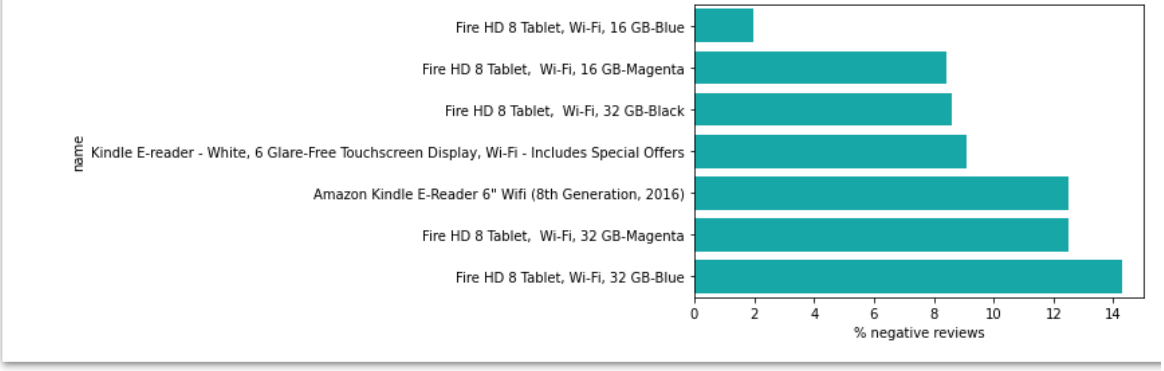
```
# Calculate as percentage of total reviews

percent_negative = pd.DataFrame((reviews['compound']<=0).groupby(reviews['name']).sum()
                                /reviews['name'].groupby(reviews['name']).count()*100,
                               columns=['% negative reviews']).sort_values(by='% negative reviews')

percent_negative
```

name	% negative reviews
Fire HD 8 Tablet, Wi-Fi, 16 GB-Blue	1.960784
Fire HD 8 Tablet, Wi-Fi, 16 GB-Magenta	8.406524
Fire HD 8 Tablet, Wi-Fi, 32 GB-Black	8.620690
Kindle E-reader - White, 6 Glare-Free Touchscreen Display, Wi-Fi - Includes Special Offers	9.090909
Amazon Kindle E-Reader 6" Wifi (8th Generation, 2016)	12.500000
Fire HD 8 Tablet, Wi-Fi, 32 GB-Magenta	12.500000
Fire HD 8 Tablet, Wi-Fi, 32 GB-Blue	14.285714

25) We can also plot this as a horizontal barplot using seaborn.

```
sns.barplot(data=percent_negative, x='% negative reviews', y=percent_negative.index, color='c')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593e26bf90>
```



26) We might also be interested in understanding what words we see more frequently in positive or negative reviews for a particular product. We can use a Wordcloud to visualise this. First, we are going to focus on one product – in this case, we have selected Fire HD 8 Tablet, Wi-Fi, 16 GB-Magenta because this has the most reviews, but you can do further investigation for the other products in your own time for additional practice.

```
[59] # Process the text data ready for wordcloud visualisation, using the function we defined earlier

     # For this part of the exercise we will focus specifically on Fire HD 8 Tablet,  Wi-Fi, 16 GB-Magenta

     reviews['processed_review'] = reviews['reviews.text'].apply(preprocess_text)

     reviews_positive_subset = reviews.loc[(reviews['name']=='Fire HD 8 Tablet,  Wi-Fi, 16 GB-Magenta')
                                     & (reviews['compound']>0),:]

     reviews_negative_subset = reviews.loc[(reviews['name']=='Fire HD 8 Tablet,  Wi-Fi, 16 GB-Magenta')
                                     & (reviews['compound']<=0),:]

     reviews_positive_subset.head()
```

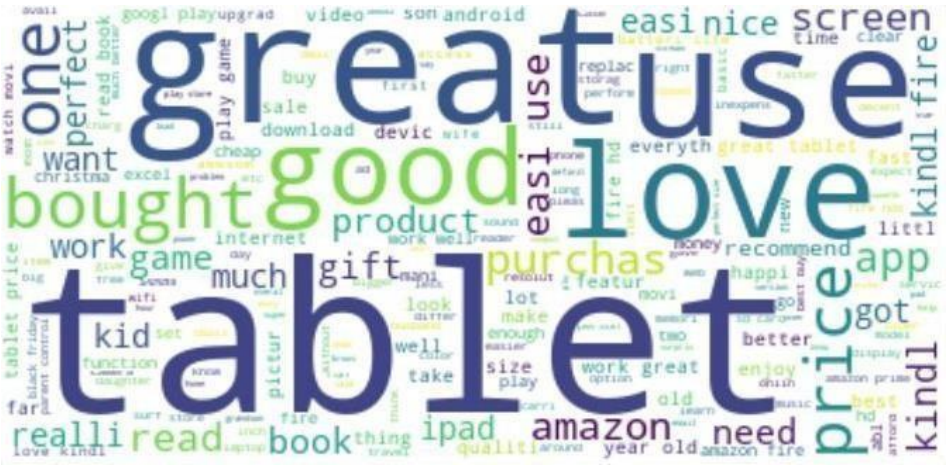
27) We can generate a wordcloud using the wordcloud library. We first use list comprehension to create one list of all the words that appear in all of the negative reviews for this product.

```
# Wordcloud of words from negative reviews by product

neg_tokens = [word for review in reviews_negative_subset['processed_review'] for word in review]

wordcloud = WordCloud(background_color='white').generate_from_text(
    ' '.join(neg_tokens))

# Display the generated image:
plt.figure(figsize=(12,12))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

Some of the words aren't particularly informative (e.g., tablet and fire are both references to the product name) but we can see, for example, 'price' and 'screen' have both been mentioned several times – perhaps these are potential concerns we should look into further?

We can also generate a wordcloud from the positive reviews.

```
# Wordcloud of words from positive reviews by product

pos_tokens = [word for review in reviews_positive_subset['processed_review'] for word in review]

wordcloud = WordCloud(background_color='white').generate_from_text(
    ' '.join(pos_tokens))

# Display the generated image:
plt.figure(figsize=(12,12))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```



28) While wordclouds provide a way to visualise word frequencies, they are sometimes difficult to interpret. Another way of understanding word frequencies is to use FreqDist from NLTK. We can use the tabulate method to understand the most frequent words, and the number of occurrences of each.

```
[62] # use the nltk FreqDist and then tabulate

     from nltk.probability import FreqDist

     pos_freqdist = FreqDist(pos_tokens)

     pos_freqdist.tabulate(10)
```

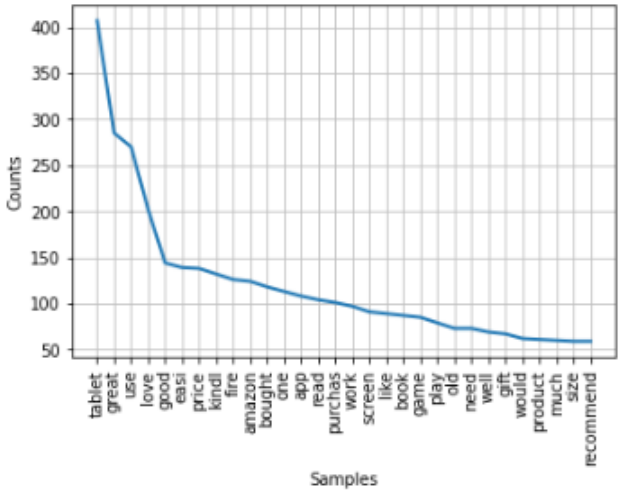
```
tablet  great    use   love   good   easi  price  kindl   fire amazon
   407    285    270    202    144    139    138    132    126    124
```

```
[63] # use the nltk FreqDist and then tabulate

     from nltk.probability import FreqDist

     neg_freqdist = FreqDist(neg_tokens)

     neg_freqdist.tabulate(10)
```

```
tablet    use   fire  kindl  price    one    old screen amazon bought
    25     19     17     12     12     10      9      9      8      7
```

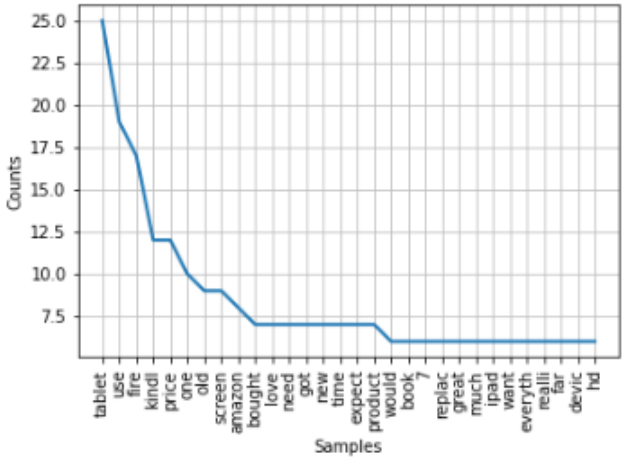
29) We can also use the plot method to create a frequency distribution plot for the most frequent words in both the positive and the negative reviews for the Fire HD 8 Tablet, Wi-Fi, 16 GB-Magenta product.

```
[64] pos_freqdist.plot(30)
```




```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593e37a950>
```

```
[65] neg_freqdist.plot(30)
```




```
<matplotlib.axes._subplots.AxesSubplot at 0x7f593e32bf50>
```