B.Tech Project

TWITTER SENTIMENT ANALYSIS USING NLP

ANURAG GUPTA
ANSHITA SRIVASTAVA
AAYSHA SHUKLA

August 2023

TWITTER SENTIMENT ANALYSIS USING NLP

A PROJECT
Submitted
In Partial Fulfilment for the Requirements
for the Degree of

BACHELOR OF TECHNOLOGY
in
INFORMATION TECHNOLOGY

by

AAYSHA SHUKLA	19301
ANSHITA SRIVASTAVA	19316
ANURAG GUPTA	19317

Under the Supervision of

Dr. VINEET KUMAR SINGH
Assistant Professor, Department of IT

to the
FACULTY OF INFORMATION TECHNOLOGY

Institute of Engineering and Technology
Dr. Rammanohar Lohia Avadh University Ayodhya, UP, INDIA

TWITTER SENTIMENT ANALYSIS USING NLP

Project report submitted
in the partial fulfilment for the award of the degree

OF

BACHELOR OF TECHNOLOGY
in
(Information Technology)

Submitted by

AAYSHA SHUKLA	19301
ANSHITA SRIVASTAVA	19316
ANURAG GUPTA	19317

August 2023

Under the Supervision of

Dr. VINEET KUMAR SINGH
Assistant Professor, Department of IT

Institute of Engineering and Technology
Dr. Rammanohar Lohia Avadh University Ayodhya, UP, INDIA

2019 - 2023

INSTITUTE OF ENGINEERING AND TECHNOLOGY, AYODHYA

DECLARATION

We hereby certify that the work which is being presented in the project report entitled "Twitter Sentiment Analysis Using NLP" by "ANURAG GUPTA, ANSHITA SRIVASTAVA and AAYSHA SHUKLA" in partial fulfilment of requirement for the award of degree of B.Tech (Information Technology) submitted in the Department of Information Technology at "INSTITUTE OF ENGINEERING & TECHNOLOGY" under Dr. Ram Manohar Lohia Avadh University, Ayodhya is an authentic record of our own work. The project was carried out during the period of March 2023 to August 2023 under the supervision of 'Dr. Vineet Kumar Singh', Assistant Professor of the Department of Information Technology.

Signature of Students :

Aaysha Shukla(19301) **Anshita Srivastava(19316)** **Anurag Gupta(19317)**

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

Signature of Supervisor :

The B. Tech Viva – Voce Examination of Anurag Gupta, Anshita Srivastava, Aaysha Shukla has been held on 12 August 2023 and accepted.

Signature of Supervisor : **Signature of External Examiner :**

Signature of HOD :

ABSTRACT

In today's world, Social Networking websites like Twitter, Facebook, Tumbler, etc. plays a very significant role. Twitter is a micro-blogging platform which provides a tremendous amount of data which can be used for various applications of Sentiment Analysis like predictions, reviews, elections, marketing, etc. Sentiment Analysis is a process of extracting information from large amounts of data, and classifies them into different classes called sentiments.

Python is simple yet powerful, high-level, interpreted and dynamic programming language, which is well known for its functionality of processing natural language data by using NLTK (Natural Language Toolkit). NLTK is a library of python, which provides a base for building programs and classification of data. NLTK also provides graphical demonstration for representing various results or trends and it also provides sample data to train and test various classifiers respectively.

The goal of this thesis is to classify twitter data into sentiments (positive or negative) by using different supervised machine learning classifiers on data collected for different Indian political parties and to show which political party is performing best for the public.

ACKNOWLEDGEMENT

We would like to place on record our deep sense of gratitude to **Dr. Vineet Kumar Singh**, Assistant Professor, Dept. of Information Technology, IET, Ayodhya, India for his generous guidance, help and useful suggestions.

We express our sincere gratitude to Mr. Rajesh Kumar Singh, Asst. Prof. in the Department of Information Technology, IET, Ayodhya, for his stimulating guidance, continuous encouragement and supervision throughout the course of present work.

We also wish to extend our thanks to Prof. Benjamin Termonia for their insightful comments and constructive suggestions to improve the quality of this research work.

We are extremely thankful to Dr. Sant Saran Mishra, Director, IET, Ayodhya, for providing us infrastructural facilities to work in, without which this work would not have been possible.

Aaysha Shukla (19301)
Anshita Srivastav (19316)
Anurag Gupta (19317)

CONTENTS

1. INTRODUCTION

There is massive information we have at our disposal today, like online news feed, webshop reviews, or social media like twitter with over 6000 tweets per second, and we can utilize it by combining text mining, natural language processing and machine learning.

It becomes possible to create an application capable of analyzing text sentiment at light speed. Capturing such a feeling is what is called sentiment analysis.

Natural Language Processing (NLP) is a hotbed of research in data science these days and one of the most common applications of NLP is sentiment analysis. From opinion polls to creating entire marketing strategies, this domain has completely reshaped the way businesses work, which is why this is an area every data scientist must be familiar with.

Thousands of text documents can be processed for sentiment (and other features including named entities, topics, themes, etc.) in seconds, compared to the hours it would take a team of people to manually complete the same task.

We will do so by following a sequence of steps needed to solve a general sentiment analysis problem. We will start with preprocessing and cleaning of the raw text of the tweets. Then we will explore the cleaned text and try to get some intuition about the context of the tweets. After that, we will extract numerical features from the data and finally use these feature sets to train models and identify the sentiments of the tweets.

Problem Statement

Twitter has become a popular platform for individuals to share their opinions and thoughts on various topics. As a result, there is an immense amount of data generated on Twitter every day, which can be valuable for businesses and organisations to understand the sentiment of their customers or the public towards their products, services, or brands. However, analysing this vast amount of data manually is time-consuming and practically impossible. Therefore, there is a growing need for automated techniques to extract valuable insights from Twitter data. The problem is that sentiment analysis on Twitter is challenging due to the short length of tweets, the use of slang, sarcasm, and context-dependent language, making it difficult to accurately determine the sentiment behind a tweet.

Objective

To accurately extract people's opinions and feelings expressed in positive or negative comments, questions and requests, by analysing a large number of unstructured texts and classifying them into sentiment classes.

1.1 TEXT MINING & NLP

Data

Data is everywhere. Text actually represents a huge part of the available data today, around 80%.

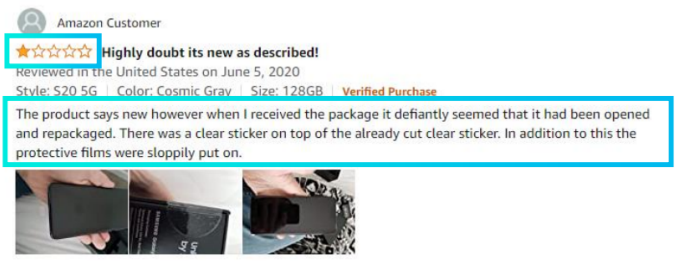
How much data has been created for the last 2 years?

- 8.5 billion … Number of Google searches per day.
- 350 million … Number of posts on Facebook per day.
- 6000 … Number of Tweets per second.
- 29 billion … Number of words in Wikipedia.
- 5.16 billion … Number of internet users worldwide.

Again, it’s you and me sharing on social media, but it’s also companies sharing their financial reports, media sharing the latest news, authors sharing books, or universities publishing their courses online. There are so many sources of written text that it would be impossible to quote them all.

The only thing to remember is that it is a wealth of information, that it is more important than ever to master. This practice is called text mining.

Text Mining



Let’s have a look at an example we should be familiar with. Say we are willing to buy a new smartphone on Amazon. We compare quite a bit the different offers available.
We compare products based on their reviews available and more specifically on two levels.
First one is simply the star rating between 1 and 5, which is very easy to understand and interpret. But there is something else which is just more valuable, the text review left with the rating, this review gives much more information about the product than the rating alone, it actually explains why such a rating was given.

There is nevertheless a difference between these two data. While the rating is easily understandable by computer because it is only a number, it is much more difficult for it to understand and interpret the text. In fact, information in this Amazon review can be classified into two categories: structured and unstructured data.

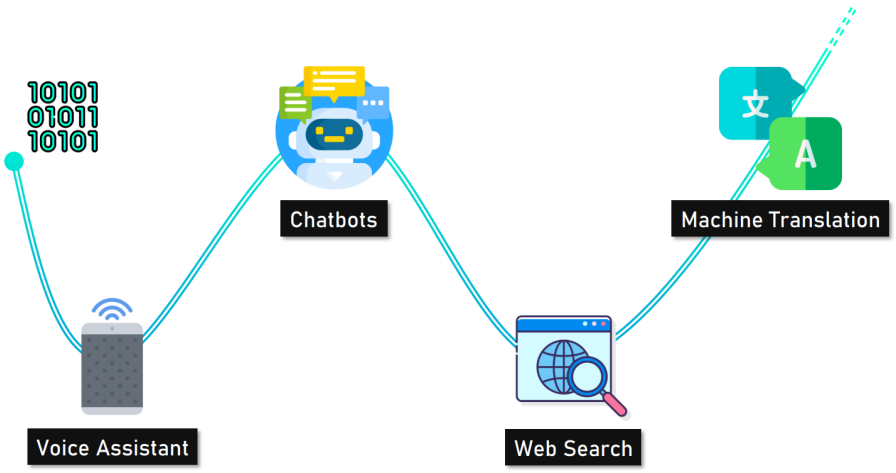
Structured data is all the data that follows a very specific format. Like in this case the rating or information such as where the review was made, the colour of the phone or its gigabyte size. All this information can be stored and searched very easily across all smartphones available on the website.

The text review falls into the second category called unstructured data. Relevant information is gathered here and there over the words which makes it much more difficult for a computer to understand and use. In this case, words like “repackaged”, or “sloppily put on” are very informative about the quality. Despite everything, such information must be detected in the text and it’s not immediately available or easily searchable.

So, text mining is exactly there to help us recover and derive meaningful information from unstructured text. Text mining is used to extract meaningful information from unstructured text data.

Text mining is a process of extracting useful information and nontrivial patterns from a large volume of text databases. It involves the use of natural language processing (NLP) techniques to extract useful information and insights from large amounts of unstructured text data. Text mining can be used as a preprocessing step for data mining or as a standalone process for specific tasks. By using text mining, the unstructured text data can be transformed into structured data that can be used for data mining tasks such as classification, clustering, and association rule mining. This allows organisations to gain insights from a wide range of data sources, such as customer feedback, social media posts, and news articles. Text mining is widely used in various fields, such as natural language processing, information retrieval, and social media analysis. It has become an essential tool for organisations to extract insights from unstructured text data and make data-driven decisions.

Natural Language Processing



NLP or Natural Language Processing. It aims at studying how machines understand human language. There are a bunch of NLP applications we know about, like Virtual Assistants such as Siri, Alexa or Google Assistant, Chatbots we can find on pretty any websites, the simple Google Search or Google Translate functionality we use daily. All these examples are pure NLP applications as they strive to understand all languages.

Why do we need Text Mining?

There are actually a number of reasons why we might want to do this. Machines simply cannot understand the text as we humans do. They need an entire field of research. We know that a computer is binary and works in succession of zeros and ones. That's the actual reason. Even though the language we speak is obvious to us, it contains many complexities, ambiguities, and it's pretty diverse. This is something that needs to be conveyed to the computer one way or another.

Text Mining and NLP are used together so a succession of letters in a document can actually make sense to a computer and that it becomes possible to leverage its power to carry out language related tasks.

If you're not convinced, just think about why we are happy to use Google Translate rather than learning French, Spanish or Chinese from scratch.

1.2 SENTIMENT ANALYSIS

Another very specific application of Text Mining and NLP: Sentiment Analysis.
Sentiment analysis is used to derive sentiment from text data by using Text Mining and NLP techniques.

Let's use a very specific example. Say we are traders monitoring the oil market. We basically have two kinds of data to make our decisions. We have structured data like oil inventories, oil economic growth indicators that are definitely driving oil prices up and down. But we also have a lot of unstructured data available. We have a news feed that is updating all day along, these news feed is a goldmine of information, provided we can use it in an efficient way. We could read every news one by one, but we would probably spend the entire day doing it as it's constantly updating. So we need another solution as a trader, one of our objective is to grab the sentiment of news, positive or negative, so we can make better informed decision as quickly as possible and make money. That's exactly something Text mining and NLP can help us with.

So we need a model to feed it with some preprocessed text and return an accurate sentiment in a matter of milliseconds. Sentiment analysis aims at detecting sentiment in text and is applicable to areas such as customer services, brand recognition or even trading.

Sentiment analysis (also known as opinion mining or emotion AI) is the use of natural language processing, text analysis, computational lingustics, and biometrics to systematically identify, extract, quantify and study effective states and subjective information. Sentiment analysis is widely applied to the voice of the customer materials such as reviews and surveys responses, online and social media and healthcare materials for applications that range from marketing to customer service to clinical medicine.

With the rise of deep language models, such as RoBERTa, also more difficult data domains can be analysed, e.g., news texts where authors typically express their opinion/sentiment less explicitly.

Roadmap Model

Let's take a look at the different stages we need to go through before we can actually build our own sentiment model.

We choose to combine Twitter data set on one side and a sentiment model on the other. Twitter data, tweets actually offers a very diverse and complex range of text data.

Users write what they think at the moment of writing, there isn't really any spelling rule or grammar taken into account, just words, URLs, emojis and something to share. It is precisely the fact that the text is not particularly well-written. That makes it interesting to use.

Tweets can be very messy, that's why we will clean up and normalize all the tweets in our data set by using text mining and NLP tools. We will try to understand which information is useful and which is not. So we only keep useful elements for later.

Here, we will use machine learning to see whether the emotion of a tweet is positive, or negative.
And it is quite a challenging task as we can see tweets are different than other type of text data that we have because:
So, we will do sentiment analysis on some tweets data

Now when we are talking about the text data of a tweet,
We will consider it in into four categories:
i) one there is text regarding the tweet itself,
ii) then there is emoji,
iii) then there is normally mentions of a user, and
iv) there could also be some link to some website inside a tweet.

In order to do the analysis we will do preprocessing on the text to make it appropriate for the way the model is trained.

We will represent remaining text data, it has to respect a specific structure so it can be used by our machine learning model. And then we will finally build our sentiment analysis application that will be able to classify new and unknown tweets.

1.3 GOOGLE COLAB

Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.

Colab is a free Jupyter notebook environment that runs entirely in the cloud. Most importantly, it does not require a setup and the notebooks that we create can be simultaneously edited by our team members - just the way we edit documents in Google Docs. Colab supports many popular machine learning libraries which can be easily loaded in our notebook.

Google Colab Features

Google Colab offers following features:

- Write and execute code in Python.
- Document our code that supports mathematical equations.
- Create/Upload/Share Notebooks.
- Import/Save Notebooks from/to Google Drive.
- Import/Publish notebooks from Github.
- Import external datasets.
- Integrate Pytorch, Tensorflow, Keras, OpenCV.
- Free Cloud service with free GPUs.

1.4 DATASET OVERVIEW

We will now authenticate and connect Google Drive with our Google Colab to retrieve our stored dataset from Drive. Then we should specify the file we want to load by using its ID.

The data we will use is stored under the very common CSV format. We can easily manipulate such files by using the Pandas library. So we can load this package together within the numpy package. We created the new data frame by using read_csv function and specifying the name

```
[ ] import pandas as pd
    import numpy as np
```

- Read Dataframe stored in Google Drive under `.csv` format

```
[ ] df = pd.read_csv("tweet_data.csv")
```

- Used the `sample` method to look at some random tweets present in our dataset

```
df.sample(10)
```

	textID	tweet_text	sentiment
7903	1965424562	@fionaaa_ YEEEAH :'D lmao, dentists aren't nic...	positive
4755	1962781914	@grrachel that made me sad..	negative
574	1957237598	Change my hairstyle,but it isn't good as it su...	negative
231	1957083786	is really, really bored... I guess I will go t...	negative
630	1957269543	@iamivanxxx That sucks to hear. I hate days li...	negative
5457	1963271626	I can't breathe well but turns out I'm not cra...	negative
4215	1962320333	ah remember the days when you'd sleep in until...	negative
5682	1963421477	@amandalaur i know right, that's so weak but ...	negative
18634	1753885116	i wish the birthday massacre would come to aus...	positive
17196	1753381126	OowweeeIII China club wuz poppinIII Lipstic n ...	positive

First, we used a sample function to get a number of random rows among the dataframe. In this case, we are showing 10 different and totally random rows. First column only contains a text ID. The two other columns, however, contain very important information, a tweet and a sentiment. These are all the data we are going to use to create a pretty effective sentiment model.

- Checked how many tweets there are in total

```
[ ] print("Number of tweets: {}".format(len(df)))

    Number of tweets: 18727
```

- To Print a tweet and its sentiment based on a tweet ID

```
[ ] tweet_id = 4879
    tweet = df.iloc[tweet_id]
```

```
[ ] print("Tweet: {}".format(tweet["tweet_text"]))
    print("Tweet sentiment: {}".format(tweet["sentiment"]))

    Tweet: Bad Day. History Test Tommorrow. And I want to go out in the sun and play..
    Tweet sentiment: negative
```

To check the data set in a bit more detail, we printed the number of tweets by using the len function with the data frame. Number equals 18727 tweets, which is reasonable to build a model.

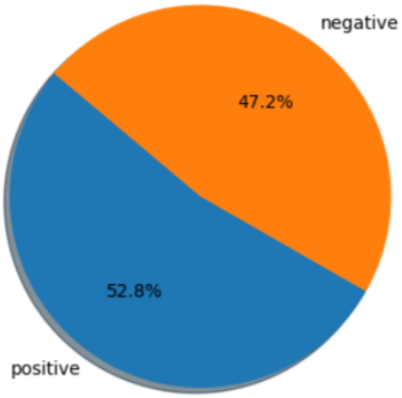
To access a specific row of data, for instance, we accessed the tweet at a specific row by using the iloc function. We then printed the tweet by using the column name where it is stored. We did the same for the sentiment. As we can see, someone seems to have a pretty bad day, which is why the sentiment is shown as negative.

1.5 DATASET VISUALISATION

```
[ ]  import matplotlib.pyplot as plt
```

- `pyplot` helps understanding and representing how tweets are distributed over the dataset

```
sentiment_count = df["sentiment"].value_counts()
plt.pie(sentiment_count, labels=sentiment_count.index,
        autopct='%1.1f%%', shadow=True, startangle=140)
plt.show()
```



- Print the count of positive and negative tweets

```
[ ]  print("Number of + tweets: {}".format(df[df["sentiment"]=="positive"].count()[0]))
     print("Number of - tweets: {}".format(df[df["sentiment"]=="negative"].count()[0]))

     Number of + tweets: 9897
     Number of - tweets: 8830
```

We have a small idea of what is contained in our data set, so graphical representation can be quite informative. For that we need the Matplotlib package, which we definitely need when it comes to making graphics.

We have first built the pie chart representing the sentiment split between positive and negative tweets. This can be done in two ways. First way is by accessing the data frame and filtering positive or negative tweets in the corresponding column and adding a count. The second way is by using the value_counts function of pandas, which is way quicker. We have our data so we can create the pie chart. Here 47.2% of negative tweets against 52.8% of positive tweets. It shows that the data is pretty well balanced between positive and negative.

We checked the exact number of positive or negative tweets by filtering the sentiment column and adding a count function. 9897 positive tweets for 8830 negative tweets.

1.3.2 Wordclouds

- The `Wordclouds` package is very useful to get a quick overview of most recurrent words in the text corpus

```
[ ] from wordcloud import WordCloud
```

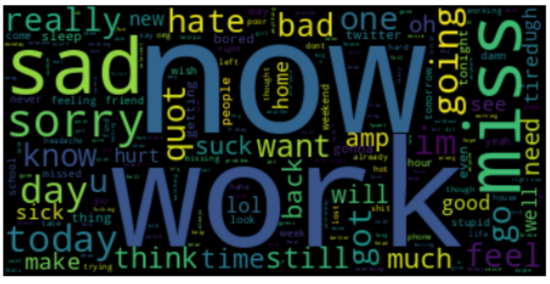
- What are the words most often present in positive tweets?

```
pos_tweets = df[df["sentiment"]=="positive"]
txt = " ".join(tweet.lower() for tweet in pos_tweets["tweet_text"])
wordcloud = WordCloud().generate(txt)
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

We used a very useful application of the wordcloud package. This lets us represent the words that appear most often in the positive or negative tweets. First we filtered the data frame on positive tweets. We then joined all the tweets together, so it only forms one long sentence. The sentence is then analysed by the WordCloud module, which can be plotted. This creates a very visual representation of the words that appear most often in positive tweets like thank love, good, nice, happy or hope.

- ... and in negative tweets?

```
neg_tweets = df[df["sentiment"]=="negative"]
txt = " ".join(tweet.lower() for tweet in neg_tweets["tweet_text"])
wordcloud = WordCloud().generate(txt)
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```



The same representation can be created for negative tweets. This time, we have more negative words such as miss, sad, hate, sorry or even work.

2. NORMALISATION

In this section, we used different text mining techniques to perform Text Normalisation. We cleaned up the messy tweets we have in our dataset as best as possible. Which means going through the cleaning of certain precise characteristics, but also using powerful tools such as Tokenization, Stemming or Lemmatization.

2.1 TEXT NORMALISATION

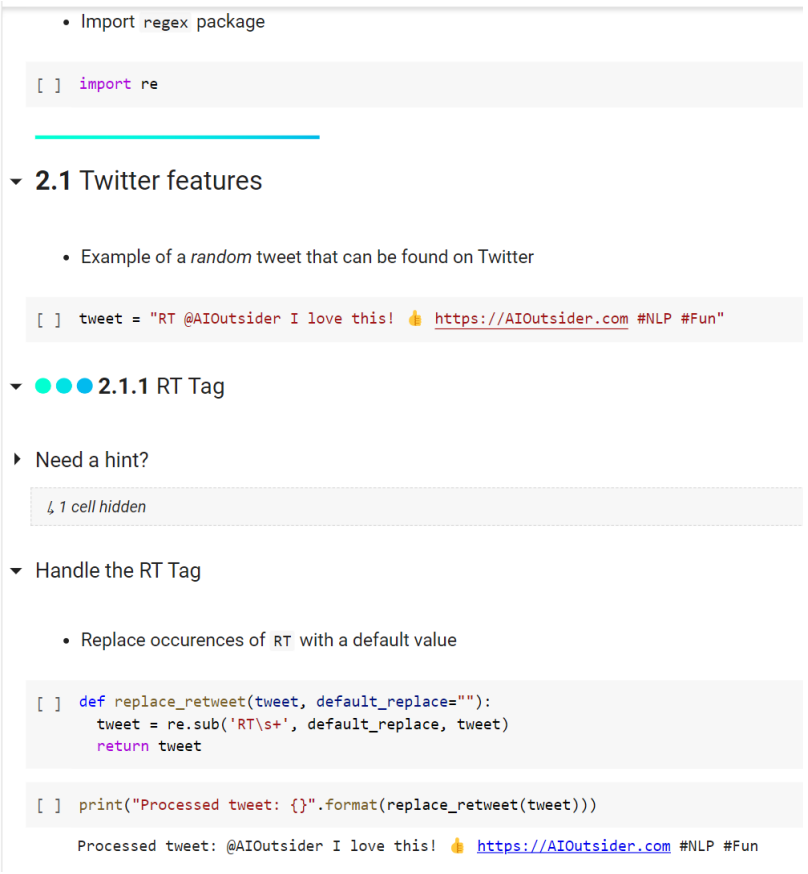
Text normalisation actually aimed at reducing randomness in a particular piece of text.

It is the process of transforming text into a single canonical form that it might not have had before. Normalising text before storing or processing it allows for separation of concerns, since input is guaranteed to be consistent before operations are performed on it. Text normalisation requires being aware of what type of text is to be normalised and how it is to be processed afterwards.

Text normalisation is a very important and integral step in Natural Language Processing. It is used in speech recognition, text to speech, spam email identification and many other applications of NLP. Normalisation is helpful in reducing the number of unique tokens present in the text and removing variations in text, also cleaning the text by removing redundant information. Two popular methods used for normalisation are stemming and lemmatization. When we normalise text, we attempt to reduce its randomness bringing it closer to a predefined “standard”. This helps us to reduce the amount of different information that the computer has to deal with, and therefore improves efficiency.

In text cleaning, we remove or replace all items that do not provide additional information.

When handling tweets these elements can easily be managed using something called regex for regular expressions. It can be used to capture and replace any specific expression, pattern, or a string of characters like URLs, user tags and hashtags.



RT Tag

Here is the tweet we are going to clean, "*RT @AIOutsider I love this! 👍 https://AIOutsider.com #NLP #Fun*". The first thing to replace is the retweet, as it does not bring much value, as it can be replaced by some default value or even simply deleted. For that we created a new function that takes the tweet as input and returns a modified version of the tweet without the retweet part.

We then created a new version of the tweet by using this sub function, this function is simply used to replace occurrences of a possible substring with another substring. This function allows us to use the regex. We need to match the RT and space pattern in this case. That's where the regex is particularly useful as it lets us match a specific combination of characters.
After running the function the retweet has magically disappeared.

@User Tag

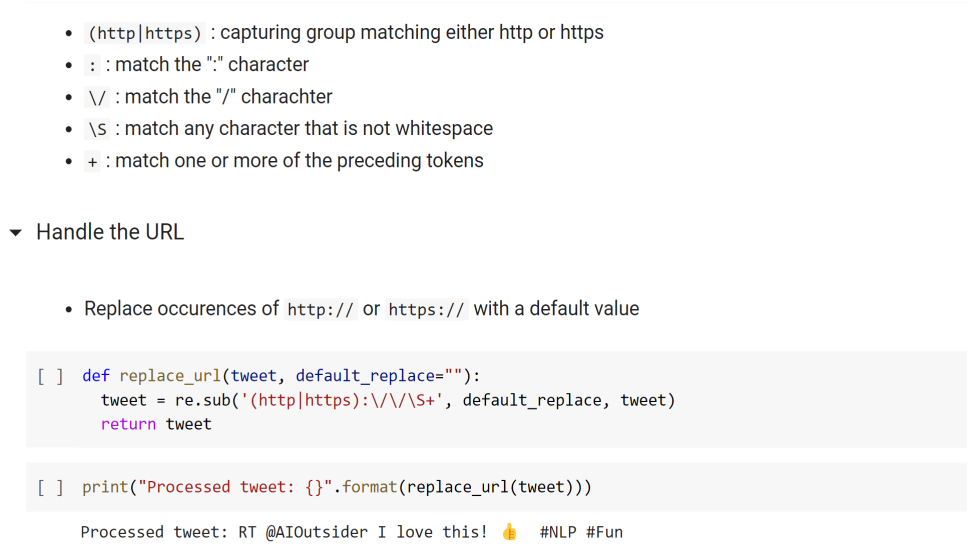
Second element is the user tag. This time we should match any string that begins with a character followed by any user tag. Same as before, we created a new function that takes the tweet as input. Replaced the user tagged by a default tag. And returned the modified version of the tweet.

Emojis

Now let's move on to emojis, emojis often carry a lot of meaning but are unusable in this pictorial form. Fortunately, there's a very nice package called emoji, which can be used to do just what we want. Our objective is now to use the package and convert all the emojis to a string.

Only thing to do is to demojise the tweet by using the emoji package. We created a new function which takes it as input and returned a modified tweet. No emoji anymore, but the clean string with thumbs_up.

URL



- `(http|https)` : capturing group matching either http or https
- `:` : match the ":" character
- `\/` : match the "/" charachter
- `\S` : match any character that is not whitespace
- `+` : match one or more of the preceding tokens

▾ Handle the URL

- Replace occurences of `http://` or `https://` with a default value

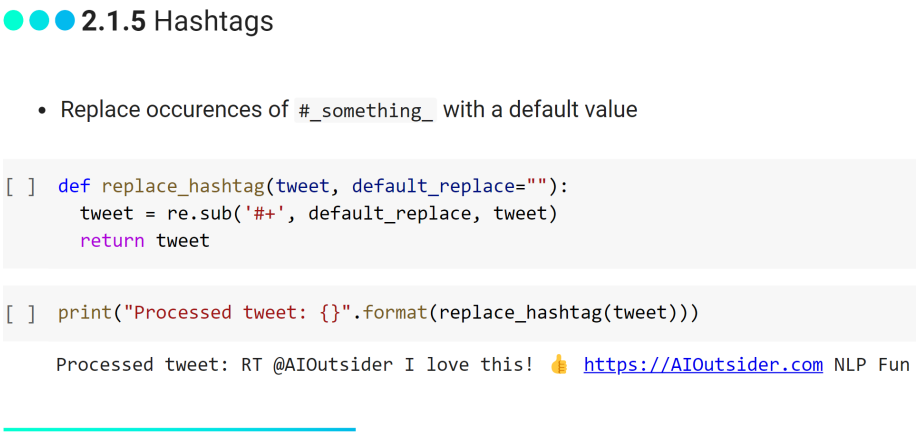
```
[ ]  def replace_url(tweet, default_replace=""):
       tweet = re.sub('(http|https):\/\/\S+', default_replace, tweet)
       return tweet
```

```
[ ]  print("Processed tweet: {}".format(replace_url(tweet)))
```

```
Processed tweet: RT @AIOutsider I love this! 👍  #NLP #Fun
```

We created a new function and off we go. Just like user text URLs do not bring very meaningful information to a tweet at least again from a sentiment perspective.

Hashtags



2.1.5 Hashtags

- Replace occurences of `#_something_` with a default value

```
[ ]  def replace_hashtag(tweet, default_replace=""):
       tweet = re.sub('#+', default_replace, tweet)
       return tweet
```

```
[ ]  print("Processed tweet: {}".format(replace_hashtag(tweet)))
```

```
Processed tweet: RT @AIOutsider I love this! 👍 https://AIOutsider.com NLP Fun
```

Hashtag which often provide precise information on the content and even avoid the sentiment of the tweet, it must therefore be kept without the assigned.

2.2 TEXT CLEANING

Data cleaning is the process of fixing or removing incorrect, corrupted, incorrectly formatted, duplicate or incomplete data within a data set. When combining multiple data sources, there are many opportunities for data to be duplicated or mislabeled. If data is incorrect, outcomes and algorithms are unreliable, even though they may look correct. There is no one absolute way to prescribe the exact steps in the data cleaning process because it will vary from dataset to dataset.

2.2 Word Features

Let's now have a look at some other features that are not really Twitter-dependant

```
[ ]  tweet = "LOOOOOOOK at this ... I'd like it so much!"
```

2.2.1 Remove upper capitalization

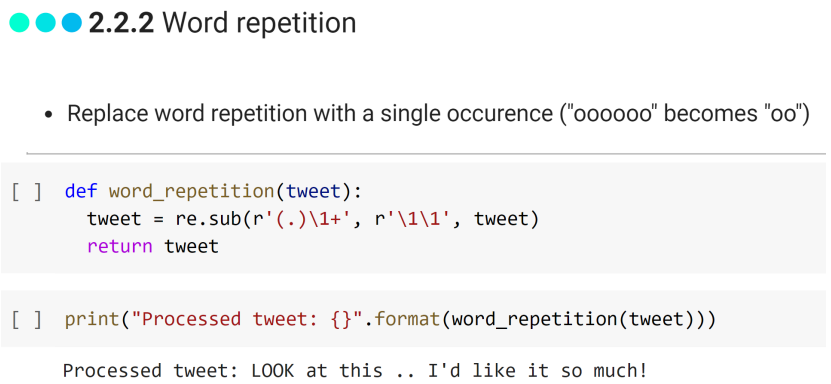
- Lower case each letter in a specific tweet

```
[ ]  def to_lowercase(tweet):
       tweet = tweet.lower()
       return tweet
```

```
[ ]  print("Processed tweet: {}".format(to_lowercase(tweet)))
```

```
Processed tweet: looooooook at this ... i'd like it so much!
```

Let's lowercase all the words in a tweet, and create a new function that takes the tweet as an input. We can simply lower the tweet by using the default lower method. We can then test and bring the result. As expected the tweet is completely lower case.



2.2.2 Word repetition

- Replace word repetition with a single occurence ("oooooo" becomes "oo")

```
[ ]  def word_repetition(tweet):
       tweet = re.sub(r'(.)\1+', r'\1\1', tweet)
       return tweet
```

```
[ ]  print("Processed tweet: {}".format(word_repetition(tweet)))
```

```
Processed tweet: LOOK at this .. I'd like it so much!
```

LOOOOOOK is an extended version of word look, such a letter repetition is often encountered when someone wants to emphasise a word. There is an easy way to match characters which repeat like "O" in this case. This can be done by combining a capturing group with the token which is used to match any character except line breaks.

2.2.3 Punctuation repetition

- Replace punctuation repetition with a single occurence ("!!!!!" becomes "!")

```
def punct_repetition(tweet, default_replace=""):
  tweet = re.sub(r'[\?\.\!]+(?=[\?\.\!])', default_replace, tweet)
  return tweet
```

```
[ ] print("Processed tweet: {}".format(punct_repetition(tweet)))
```

```
Processed tweet: LOOOOOOOOK at this . I'd like it so much!
```

Punctuation repetition can be handled by a new function. Here we are going to use the character's set which is used to match any character in the set. Inside this set we can specify all the punctuation that we want to match, all preceded by a backslash.

2.2.4 Word contraction

- Install the `contractions` package

```
pip install contractions
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting contractions
  Downloading contractions-0.1.73-py2.py3-none-any.whl (8.7 kB)
Collecting textsearch>=0.0.21
  Downloading textsearch-0.0.24-py2.py3-none-any.whl (7.6 kB)
Collecting anyascii
  Downloading anyascii-0.3.2-py3-none-any.whl (289 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 289.9/289.9 kB 8.8 MB/s eta 0:00:00
Collecting pyahocorasick
  Downloading pyahocorasick-2.0.0-cp39-cp39-manylinux_2_5_x86_64.manylinux1_x86_64.whl (103 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 103.2/103.2 kB 13.5 MB/s eta 0:00:00
Installing collected packages: pyahocorasick, anyascii, textsearch, contractions
Successfully installed anyascii-0.3.2 contractions-0.1.73 pyahocorasick-2.0.0 textsearch-0.0.24
```

- Import the installed package

```
[ ] import contractions
```

- Use `contractions_dict` to list most common contractions

```
[ ] print(contractions.contractions_dict)
```

```
{"I'm": 'I am', "I'm'a": 'I am about to', "I'm'o": 'I am going to', "I've": 'I have', "I'll": 'I will',
```

- Create a `_fix_contractions` function used to replace contractions with their extended forms by using the contractions dictionnary

```
def _fix_contractions(tweet):
  for k, v in contractions.contractions_dict.items():
    tweet = tweet.replace(k,v)
    return tweet
```

```
print("Processed tweet: {}".format(_fix_contractions(tweet)))
```

```
Processed tweet: LOOOOOOOOK at this ... I'd like it so much!
```

- Create a `_fix_contractions` function used to replace contractions with their extended forms by using the contractions package

```
def fix_contractions(tweet):
  tweet = contractions.fix(tweet)
  return tweet
```

```
print("Processed tweet: {}".format(fix_contractions(tweet)))
```

```
Processed tweet: LOOOOOOOOK at this ... I would like it so much!
```

Contractions are words like I'm, don't, won't or shouldn't. There is a package that is used to transform these contractions. It is simply called a contractions package. Just like stopwords the contractions package provides a list of contractions and more especially a dictionary that maps the contractions to its expanded form.

We created a function that fixes these contractions, we will create two different functions. The first one by using the dictionary and the second one by using the package directly. As we can see, the construction has been replaced.

2.3 TOKENIZATION

Tokenization is a way to separate text into smaller chunks that can be used by computers to learn. Computers and machine learning models don't need white spaces, but they need tokenized sentences to learn. More than just indicating what words are, tokenization also represents a crucial step towards word representation via vectorization. We humans construct sentences by combining words and so goes for computers. This is why tokenization is used.

Tokenization is more than just splitting sentences into words, it is an additional opportunity to choose words or items to keep and choose not to.

Three concrete examples are:
First one is punctuation.
Often and especially in tweets punctuation is added chaotically for no specific reason other than attracting attention. In some cases, it may well be useful, though.

Second example is stopwords.
Stopwords are simply a set of words commonly used in a particular language like “I”, “am”, “do”, “or”, “and”, “for”. We want to remove these words because they often do not carry much of the sentence's meaning. Removing them sometimes allows the machine to focus on other words that are really important.

Finally, we have numbers.
While they most of the time carry variable quantitative information, it does not mean numbers are always up. Let’s take the example of tweets about the lottery. Someone might tweet something like, “Whoa! I won 50 bucks today, so nice.” and another one might tweet at the same time “Almost won a million dollars today, so angry.”, as we can see, the number has nothing to do with the sentiment expressed in the sentence.

Well, these three examples are basically things we control when tokenizing a sentence, whether we want to keep them or not is totally up to us. It doesn’t mean that we should always keep or get rid of everything. It's certainly something we should consider based on what we think is important and relevant to our application.

- Import `NLTK`
- Import the `word_tokenize` module from NLTK
- Download the `Punkt` tokenizer model from NLTK

```
[ ] import nltk
    from nltk.tokenize import word_tokenize
    nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
True
```

- Simple tweet to be tokenized

```
[ ] tweet = "These are 5 different words!"
```

- Create a `tokenize()` function that takes a tweet as input and returns a list of tokens

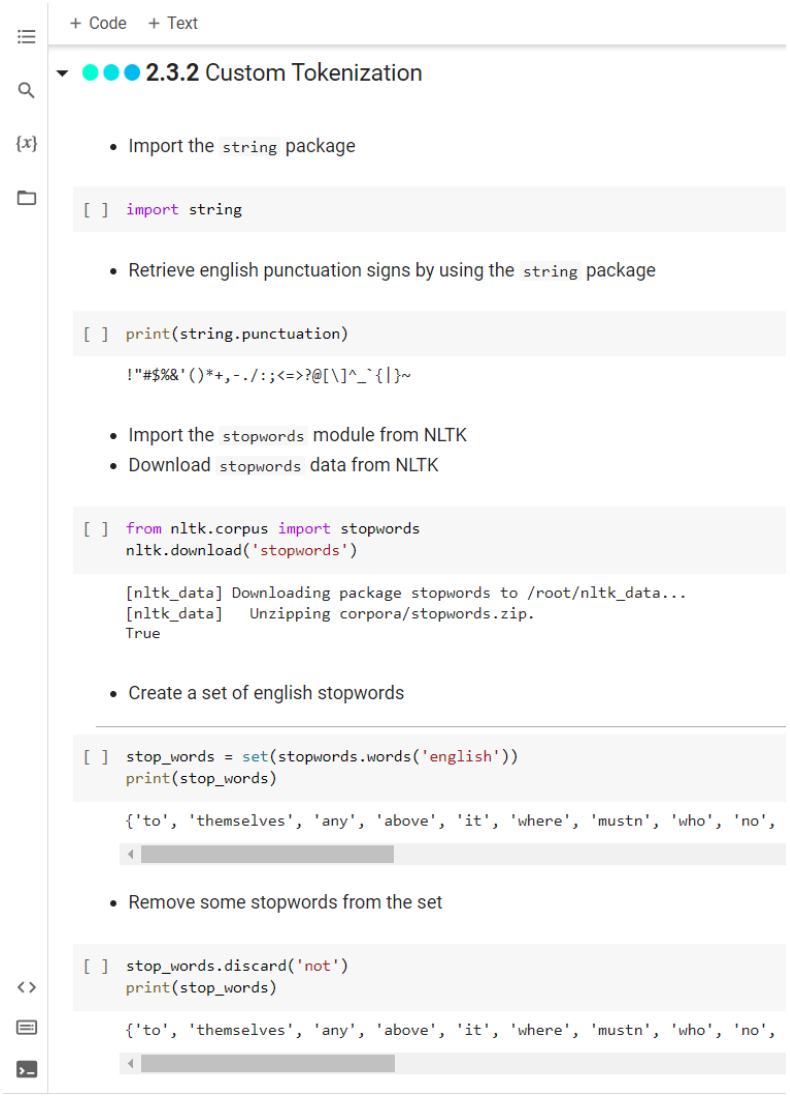
```
[ ] def tokenize(tweet):
      tokens = word_tokenize(tweet)
      return tokens
```

- Use the `tokenize()` function to print the tokenized version of a tweet

```
[ ] print(type(tokenize(tweet)))
    print("Tweet tokens: {}".format(tokenize(tweet)))
```

```
<class 'list'>
Tweet tokens: ['These', 'are', '5', 'different', 'words', '!']
```

Tokenization can be done by using NLTK for this method, make sure the package is installed in Collab. Once done we can import the installed package as well as the tokenization module and download the Punkt model data.



Sometimes you might want to keep some specific items in a sentence or conversely, get rid of them. This is the case, for instance, for punctuation, stop words, and even in some cases numbers.

Let's have a look at Punctuations first. There is a specific package that can help us handle these signs, the string package. It can be used to print most of the existing punctuation signs, which is perfect in case we decide we don't want to keep them. In the same logic, it would be great to have a list of all the stop words in a specific language. The NLTK package can help. All these words can be downloaded out of the stopwork module, and stored in a set.

Let’s have a look at the list of items we have. “not” be a stopword while it brings so much meaning to a sentence, why would we want to get rid of it? Once again, it all depends on what we are trying to achieve and especially what kind of model we use. We can just remove it from the set with the discard function. As we can see, it is not in the list anymore.

2.4 STEMMING

Stemming is the process of reducing words to their root form. Stemming follows a Rule-based Approach. This means that it is an algorithm that decides which parts of a word are cut off. It can be very fast, but it also means we might end up with non dictionary words. In practice, this is not always an issue. For instance, if we are building a search engine, we might not need dictionary words as we try to match similar or query and document words as fast as possible. If we want to build a more complex language system like live translation, stemming might not be the best solution we can find.

There are two types of stemming: Under-stemming and Over-stemming. Over-stemming happens when too much of the word is chopped-off. Under-stemming happens when a word is not chopped-off enough.

```
[ ] from nltk.stem import PorterStemmer
    from nltk.stem import LancasterStemmer
    from nltk.stem.snowball import SnowballStemmer
```

- List of tokens to stem (remember that we stem tokens and not entire sentences)

```
[ ] tokens = ["manager", "management", "managing"]
```

- Stemmers can be defined by directly using NLTK

```
[ ] porter_stemmer = PorterStemmer()
    lancaster_stemmer = LancasterStemmer()
    snoball_stemmer = SnowballStemmer('english')
```

- Create a `stem_tokens` function that takes the list of tokens as input and returns a list of stemmed tokens

```
def stem_tokens(tokens, stemmer):
  token_list = []
  for token in tokens:
    token_list.append(stemmer.stem(token))
  return token_list
```

- Print the different results and compare the stemmed tokens

```
[ ] print("Porter stems: {}".format(stem_tokens(tokens, porter_stemmer)))
    print("Lancaster stems: {}".format(stem_tokens(tokens, lancaster_stemmer)))
    print("Snowball stems: {}".format(stem_tokens(tokens, snoball_stemmer)))

    Porter stems: ['manag', 'manag', 'manag']
    Lancaster stems: ['man', 'man', 'man']
    Snowball stems: ['manag', 'manag', 'manag']
```

- Check over-stemming and under-stemming

```
[ ] tokens = ["international", "companies", "had", "interns"]

[ ] print("Porter stems: {}".format(stem_tokens(tokens, porter_stemmer)))
    print("Lancaster stems: {}".format(stem_tokens(tokens, lancaster_stemmer)))
    print("Snowball stems: {}".format(stem_tokens(tokens, snoball_stemmer)))

    Porter stems: ['intern', 'compani', 'had', 'intern']
    Lancaster stems: ['intern', 'company', 'had', 'intern']
    Snowball stems: ['intern', 'compani', 'had', 'intern']
```

There are many different ways to do so, and that's why we only focused on three well-known stemming algorithms here. These are the PorterStemmer, LancastersStemmer and SnowballStemmer.

The list of tokens we will try to tokenize are manager, management, and managing. We then created three variables, one for each stemmer and assigned them with the different modules we just loaded and specified the language for the SnowballStemmer.
We then created a function that takes the list of tokens, and the Stemmer as input. The objective is to use the Stemmer to stem each token and return a list of stem tokens. We then printed the results below for the different stemmers. As expected, the PorterStemmer and SnowballStemmer are very similar while the LancastersStemmer has cut the much bigger part of words.
Let's look at another example, international companies, had, interns. The results are of over-stemming here, international and interns both have the same stem. This is quite unexpected as the original words carry a completely different meaning. So we now have to look at a second NLP tool called Lemmatization, it is better at handling these kinds of words.

2.5 LEMMATIZATION

Lemmatization is the process of grouping together different inflected forms of a word so they can be analysed as a single item. It serves the same purpose as Stemming but makes use of word context.

Stemming and Lemmatization vary on different points. First is that Lemmatization follows a dictionary-based approach. This means in particular, that it will always return the words which exist in the dictionary, whereas Stemming could and often do produce unknown words. Secondly, the lemmatization makes use of context to shorten a word.
Let's take an example, say we have three times the same word, ‘running’ once as a noun, once as an adjective, and once as a verb. While stemming will shorten these three words the exact same way, Lemmatization will use their context to determine their shortest with existing dictionary form. So, only the extended form of the verb run is shortened to its infinity form.

The other words take their shortest form while respecting their type. Obviously taking context into account, make this method slower than stemming, which is only a rule-based algorithm. In conclusion, we can see that the use of stemming or the lemmatization will mostly depend on the importance given to the output, especially meaning-wise.

```
[ ]  from nltk.stem import WordNetLemmatizer
     from nltk.corpus import wordnet
     nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
True
```

- List of tokens to lemmatize (remember that we lemmatize tokens and not entire sentences)

```
[ ]  tokens = ["international", "companies", "had", "interns"]
```

- Part of Speech (POS) tagging

```
[ ]  word_type = {"international": wordnet.ADJ,
                  "companies": wordnet.NOUN,
                  "had": wordnet.VERB,
                  "interns": wordnet.NOUN
                  }
```

- Create the lemmatizer by using the `WordNet` module

```
[ ]  lemmatizer = WordNetLemmatizer()
```

- Create a `lemmatize_tokens` function that takes the list of tokens as input and returns a list of lemmatized tokens

```
[ ]  def lemmatize_tokens(tokens, word_type, lemmatizer):
       token_list = []
       for token in tokens:
         token_list.append(lemmatizer.lemmatize(token, word_type[token]))
       return token_list
```

```
[ ]  print("Tweet lemma: {}".format(
         lemmatize_tokens(tokens, word_type, lemmatizer)))
```

```
Tweet lemma: ['international', 'company', 'have', 'intern']
```

WordNet is a very wide lexical database developed by Princeton University. It encapsulates 1000s of words in more than 200 languages and offers Lemmatization capabilities. So let's have a look at the same tokenized sentence we had before this time, we have to add the step code, Part of Speech (POS) tagging. It allows us to associate a word with its grammatical form.

We have ‘international’ as an adjective so we can associate it with the wordnet.ADJ part of speech, ‘companies’ is a noun, ‘had’ is a verb and ‘interns’ is also a noun. We now have a list of tokens and a part of the speech dictionary for these words. We can just initialise them at lemmatizer, which will be the WordNetLemmatizer from the NLTK package. We then created a function that takes a list of tokens and returns a list of lemmatized tokens. We then got the result, ‘international’, ‘company’, ‘have’, ‘intern’. This time they have been lemmatized to correct root form and they’re no longer reduced to the same words. Words are transformed to their singular form and verbs to their infinity form.

2.6 PUTTING IT ALL TOGETHER

2.6 Putting it all together

- Long and complex tweet to be processed

```
[ ] complex_tweet = r"""RT @AIOutsider : he looooook,
    THis is a big and complex TWeet!!! 👍 ...
    We'd be glad if you couldn't normalize it!
    Check https://t.co/7777 and LET ME KNOW!!! #NLP #Fun"""
```

- Create a custom `process_tweet` function that can be used to process tweets end-to-end
- **Note**: this function will be used as a base for the following sections, so be careful!

```
def process_tweet(tweet, verbose=False):
  if verbose: print("Initial tweet: {}".format(tweet))

  ## Twitter Features
  tweet = replace_retweet(tweet) # replace retweet
  tweet = replace_user(tweet, "") # replace user tag
  tweet = replace_url(tweet) # replace url
  tweet = replace_hashtag(tweet) # replace hashtag
  if verbose: print("Post Twitter processing tweet: {}".format(tweet))

  ## Word Features
  tweet = to_lowercase(tweet) # lower case
  tweet = fix_contractions(tweet) # replace contractions
  tweet = punct_repetition(tweet) # replace punctuation repetition
  tweet = word_repetition(tweet) # replace word repetition
  tweet = demojize(tweet) # replace emojis
  if verbose: print("Post Word processing tweet: {}".format(tweet))

  ## Tokenization & Stemming
  tokens = custom_tokenize(tweet, keep_alnum=False, keep_stop=False) # tokenize
  stemmer = SnowballStemmer("english") # define stemmer
  stem = stem_tokens(tokens, stemmer) # stem tokens

  return stem
```

- Test your `process_tweet` function!

```
[ ] print(process_tweet(complex_tweet, verbose=False))
```

```
['look', 'big', 'complex', 'tweet', 'thumbs_up', 'would', 'glad', 'could', 'not', 'normal', 'check', 'let', 'know', 'nlp', 'fun']
```

- Look at some more examples!
- **Note:** it's totally possible you encounter some strange tweet processing (happens if the original tweet is initially strangely written)

```
[ ] import random
```

```
for i in range(5):
  tweet_id = random.randint(0,len(df))
  tweet = df.iloc[tweet_id]["tweet_text"]
  print(process_tweet(tweet, verbose=True))
  print("\n")
```

```
Initial tweet: @amygrant Chocolate with peanut butter. One of my favorite combonations
Post Twitter processing tweet:  Chocolate with peanut butter. One of my favorite combonations
Post Word processing tweet:  chocolate with peanut butter. one of my favorite combonations
['chocol', 'peanut', 'butter', 'one', 'favorit', 'combon']


Initial tweet: worried about Mr. Socks tonight.
Post Twitter processing tweet: worried about Mr. Socks tonight.
Post Word processing tweet: worried about mr. socks tonight.
['worri', 'mr.', 'sock', 'tonight']


Initial tweet: @SherriEShepherd hey Sherri -- don't give up b/c they're married; they may have a brother or a friend!
Post Twitter processing tweet:  hey Sherri -- don't give up b/c they're married; they may have a brother or a friend!
Post Word processing tweet:  hey sherri -- do not give up b/c they are married; they may have a brother or a friend!
['hey', 'sherri', '--', 'not', 'give', 'b/c', 'marri', 'may', 'brother', 'friend']


Initial tweet: This is not fair... bath again....
Post Twitter processing tweet: This is not fair... bath again....
Post Word processing tweet: this is not fair. bath again.
['not', 'fair', 'bath']


Initial tweet: @reemerband wooooo the tour has started yay, 13 days  have an awesome time! loved the video xD
Post Twitter processing tweet:  wooooo the tour has started yay, 13 days  have an awesome time! loved the video xD
Post Word processing tweet:  woo the tour has started yay, 13 days  have an awesome time! loved the video xd
['woo', 'tour', 'start', 'yay', '13', 'day', 'awesom', 'time', 'love', 'video', 'xd']
```

3. TEXT REPRESENTATION

Feature Extraction is a general term that is also known as a text representation of text vectorization which is a process of converting text into numbers. We call vectorization because when text is converted in numbers it is in vector form.
Computers work with numbers and not letters. Knowing how to transform text into a vectorized number, therefore appears to be an essential step in the creation of our application. There are many more, we will see three vectorization methods in this section, Positive/Negative frequency, Bag-of-Words and TF-IDF.

3.1 PROCESSING TWEETS

This step is used to represent tax in a way that is understandable and computable by computers and machine learning models. Computers only use and understand numbers.
Here we will use only three of many methods to represent text as numbers: positive/negative frequencies, bag of words, and TF-IDF. The objective is simply to extract features from text and represent these features as numbers in a vector. These numbers represent the goal to be achieved. In each case, the goal is completely different. So are the features that best represent the text. Text representation aims at representing text as a vector so this vector can be used by computers and machine learning models.

```
[ ] pip install -U scikit-learn
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.3.0)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.22.4)
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.10.1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.3.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.2.0)
```

- Apply `process_tweet` function created in section 2 to the entire DataFrame
- Convert sentiment to 1 for "positive" and 0 for "negative" sentiment

```
df["tokens"] = df["tweet_text"].apply(process_tweet)
df["tweet_sentiment"] = df["sentiment"].apply(lambda i: 1
                                              if i == "positive" else 0)
df.head(10)
```

	textID	tweet_text	sentiment	tokens	tweet_sentiment
0	1956967666	Layin n bed with a headache ughhhh...waitin o...	negative	[layin, n, bed, headach, ughh.waitin, call]	0
1	1956967696	Funeral ceremony...gloomy friday...	negative	[funer, ceremony.gloomi, friday]	0
2	1956967789	wants to hang out with friends SOON!	positive	[want, hang, friend, soon]	1
3	1956968477	Re-pinging @ghostridah14: why didn't you go to...	negative	[re-ping, not, go, prom, bf, not, like, friend]	0
4	1956968636	Hmmm. http://www.djhero.com/ is down	negative	[hmm]	0
5	1956969035	@charviray Charlene my love. I miss you	negative	[charlen, love, miss]	0
6	1956969172	@kelcouch I'm sorry at least it's Friday?	negative	[sorri, least, friday]	0
7	1956969531	Choked on her retainers	negative	[choke, retain]	0
8	1956970047	Ugh! I have to beat this stupid song to get to...	negative	[ugh, beat, stupid, song, get, next, rude]	0
9	1956970424	@BrodyJenner if u watch the hills in london u ...	negative	[watch, hill, london, realls, tourtur, week, w...	0

- Convert DataFrame to two lists: one for the tweet tokens (X) and one for the tweet sentiment (y)

```
[ ] X = df["tokens"].tolist()
    y = df["tweet_sentiment"].tolist()
```

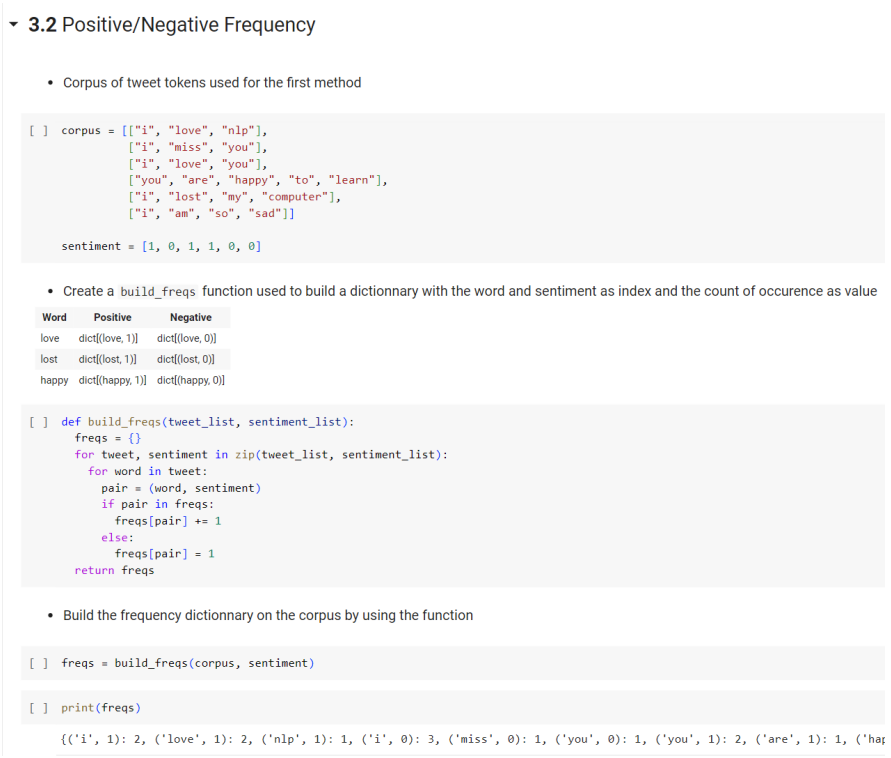
```
[ ] print(X)
    print(y)
```

```
[['layin', 'n', 'bed', 'headach', 'ughh.waitin', 'call'], ['funer', 'ceremony.gloomi', 'friday'], ['want', 'hang', 'friend', '
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
```

We used a specific library quite many times, which is called Scikit-Learn. Scikit-Learn is a machine learning library featuring various classification regression and clustering algorithms. It also provides a lot of useful preprocessing tools.
We created a new column in our data frame, which contains a tokenized version of the tweet.
We applied the *process_tweet* function to the *tweet_text* column of our data frame. We then created a new *tweet_sentiment* column containing a binary sentiment instead of text sentiment. This column is 1 in case a sentiment is positive and 0 in the other case.
We then converted this data frame to a list, created two lists, one for the tokens that we would call X, and one for the sentiment that we call Y.

3.2 POSITIVE/NEGATIVE FREQUENCY

Let's have a look at the following tweets: ‘I am glad I got hired.’, ‘This is great.’, ‘I am sad I got fired.’, ‘This is bad.’.
As we can see, they contain more or less the same words, but they do not convey the same sentiment. The positive/negative frequencies method actually tries to give a feeling of how much the word is used to express a particular positive or negative sentiment.
In many representation methods, the first step is to list the different words present in the tweet corpus. We splitted the view between positive and negative sentiment. We counted the number of times the word appears in positive tweets and for negative tweets. Which gave us all positive and negative frequencies tables.



3.2 Positive/Negative Frequency

- Corpus of tweet tokens used for the first method

```
[ ]  corpus = [["i", "love", "nlp"],
               ["i", "miss", "you"],
               ["i", "love", "you"],
               ["you", "are", "happy", "to", "learn"],
               ["i", "lost", "my", "computer"],
               ["i", "am", "so", "sad"]]

     sentiment = [1, 0, 1, 1, 0, 0]
```

- Create a `build_freqs` function used to build a dictionnary with the word and sentiment as index and the count of occurence as value

Word	Positive	Negative
love	dict[(love, 1)]	dict[(love, 0)]
lost	dict[(lost, 1)]	dict[(lost, 0)]
happy	dict[(happy, 1)]	dict[(happy, 0)]

```
[ ]  def build_freqs(tweet_list, sentiment_list):
       freqs = {}
       for tweet, sentiment in zip(tweet_list, sentiment_list):
         for word in tweet:
           pair = (word, sentiment)
           if pair in freqs:
             freqs[pair] += 1
           else:
             freqs[pair] = 1
       return freqs
```

- Build the frequency dictionnary on the corpus by using the function

```
[ ]  freqs = build_freqs(corpus, sentiment)
```

```
[ ]  print(freqs)

     {('i', 1): 2, ('love', 1): 2, ('nlp', 1): 1, ('i', 0): 3, ('miss', 0): 1, ('you', 0): 1, ('you', 1): 2, ('are', 1): 1, ('hap
```

- Build the frequency dictionnary on the entire dataset by using the function

```
[ ] freqs_all = build_freqs(X, y)
```

```
[ ] print("Frequency of word 'love' in + tweets: {}".format(freqs_all[("love", 1)]))
    print("Frequency of word 'love' in - tweets: {}".format(freqs_all[("love", 0)]))
```

```
Frequency of word 'love' in + tweets: 1359
Frequency of word 'love' in - tweets: 67
```

- Create a `tweet_to_freqs` function used to convert tweets to a 2-d array by using the frequency dictionnary

```
[ ] def tweet_to_freq(tweet, freqs):
      x = np.zeros((2,))
      for word in tweet:
        if (word, 1) in freqs:
          x[0] += freqs[(word, 1)]
        if (word, 0) in freqs:
          x[1] += freqs[(word, 0)]
      return x
```

- Print the 2-d vector by using the `tweet_to_freqs` function and the *corpus* dictionnary

```
[ ] print(tweet_to_freq(["i", "love", "nlp"], freqs))
```

```
[5. 3.]
```

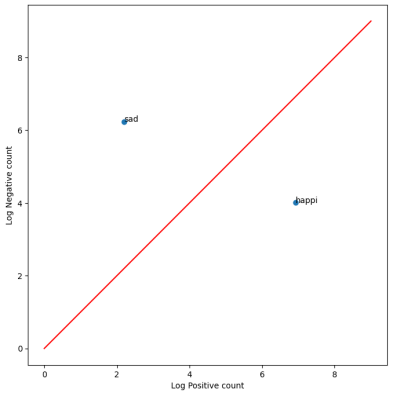
- Print the 2-d vector by using the `tweet_to_freqs` function and the *dataset* dictionnary

```
[ ] print(tweet_to_freq(["i", "love", "nlp"], freqs_all))
```

```
[1359.   67.]
```

- Plot word vectors in a chart and see where they locate

```
fig, ax = plt.subplots(figsize = (8, 8))

word1 = "happi"
word2 = "sad"

def word_features(word, freqs):
  x = np.zeros((2,))
  if (word, 1) in freqs:
    x[0] = np.log(freqs[(word, 1)] + 1)
  if (word, 0) in freqs:
    x[1] = np.log(freqs[(word, 0)] + 1)
  return x

x_axis = [word_features(word, freqs_all)[0] for word in [word1, word2]]
y_axis = [word_features(word, freqs_all)[1] for word in [word1, word2]]

ax.scatter(x_axis, y_axis)

plt.xlabel("Log Positive count")
plt.ylabel("Log Negative count")

ax.plot([0, 9], [0, 9], color = 'red')
plt.text(x_axis[0], y_axis[0], word1)
plt.text(x_axis[1], y_axis[1], word2)
plt.show()
```



3.3 BAG OF WORDS

Another method to vectorize the corpus of tweets is, bag of words. It can be used to extract features from text documents. Let's say we have the following tweet corpus: 'I like to learn.' and 'We all like and all want to learn.'. The idea of bag of words is to bring together all the words within a single entity, the bag, without any grammatical or order considerations between the words, words are just words. Once in the bag, words can be transferred to a matrix, forming the different features of the vector. Note that one word in the corpus creates only one feature, even if it appears multiple times, features are based on unique words. Tweets can then be transferred to the Matrix as well. The idea is simply to look at the first word in the Matrix and check how many times it appears in the tweet.
We convert the two tweets in two different arrays. Arrays, which only keeps information of the word present in the tweet, not the other, or any grammatical considerations.
MATRIX DIMENSION = (# of tweets, # unique words in corpus)
So it means the more unique words in the corpus, the bigger the matrix.

3.3 Bag of Word

- Corpus of tweet tokens used for the second method

```
[ ]  corpus = [["love", "nlp"],
               ["miss", "you"],
               ["hate", "hate", "hate", "love"],
               ["happy", "love", "hate"],
               ["i", "lost", "my", "computer"],
               ["i", "am", "so", "sad"]]
```

- Import `CountVectorizer` from the Scikit-learn Library

```
[ ]  from sklearn.feature_extraction.text import CountVectorizer
```

- Create a `fit_cv` function used to build the Bag-of-Words vectorizer with the corpus

```
[ ]  def fit_cv(tweet_corpus):
       cv_vect = CountVectorizer(tokenizer=lambda x: x,
                                 preprocessor=lambda x: x)
       cv_vect.fit(tweet_corpus)
       return cv_vect
```

- Use the `fit_cv` function to fit the vectorizer on the corpus

```
[ ]  cv_vect = fit_cv(corpus)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:525:
  warnings.warn(
```

- Get the vectorizer features (matrix columns)

```
[ ]  ft = cv_vect.get_feature_names_out()
```

```
[ ]  print("There are {} features in this corpus".format(len(ft)))
     print(ft)
```

```
There are 13 features in this corpus
['am' 'computer' 'happy' 'hate' 'i' 'lost' 'love' 'miss' 'my' 'nlp' 'sad'
 'so' 'you']
```

- Convert the corpus to a matrix by using the vectorize

```
[ ]  cv_mtx = cv_vect.transform(corpus)
```

- Print the matrix shape

```
[ ]  print("Matrix shape is: {}".format(cv_mtx.shape))
```

```
Matrix shape is: (6, 13)
```

- Convert the matrix to an array

```
     cv_mtx.toarray()
```

```
array([[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
       [0, 0, 0, 3, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0],
       [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]])
```

- Transform a new tweet by using the vectorizer

```
[ ]  new_tweet = [["lost", "lost", "miss", "miss"]]
     cv_vect.transform(new_tweet).toarray()
```

```
array([[0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0]])
```

```
[ ]  unknown_tweet = [["John", "drives", "cars"]]
     cv_vect.transform(unknown_tweet).toarray()
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

We imported a module which is actually the same as Bag of Words. This vectorizer will be used to vectorize the corpus of a tweet, so let's create a new function that only takes a tweet corpus as input. First, we defined our vectorizer of variables as a CountVectorizer from Scikit-Learn. We then used a custom function to create the vectorizer based on the original corpus, and we simply used the get_feature_names function to get the different features contained in this vectorizer. As we can see, there are 13 features representing the different unique words present in the corpus. Now, we created a full matrix by using the vectorizer, this can be done easily by using the transform method of Scikit-Learn. The Matrix is defined by the number of tweet times the number of unique words in the corpus, in this case, it would be 6 X 13. We checked if it's correct by using the shape property. We then saw what it looks like by using the *toarray()* method.

3.4 TERM FREQUENCY - INVERSE DOCUMENT FREQUENCY (TF-IDF)

TF-IDF stands for Term Frequency Inverse Document Frequency of records. It is actually one of the most widely used techniques to represent text. The Term frequency, this element represents the frequency of each word in each document of the corpus. It is just computed by counting the number of times a specific word appears in a document and dividing this number by the total count of words in this document. The second term Inverse Document Frequency, this element is used to compute the weight of words across all documents. A bit like stopwords, this element aimed at limiting the impact of words appearing often in a set of documents. This in order to focus on words appearing less often and which may carry more information on the sentiment. It is simply computed by taking the log of the ratio between the total number of documents and the number of documents where a specific word appears. The more the words appear in the document, the lower the IDF will be. Unlike TF, IDF is only calculated at the word level, not at the document level.

For each corpus word in each document. We can simply multiply the TF together with the IDF and we get that words appearing in the two documents have zero weights, while this is not the case for words which are not commonly found in these documents. This method is slightly different from the bag of words in the sense it gives more value to some words while a bag of words simply vectorizes all the words the exact same way.

```
[ ]  corpus = [["love", "nlp"],
               ["miss", "you"],
               ["hate", "hate", "hate", "love"],
               ["happy", "love", "hate"],
               ["i", "lost", "my", "computer"],
               ["i", "am", "so", "sad"]]
```

- Import `TfidfVectorizer` from the Scikit-learn Library

```
[ ]  from sklearn.feature_extraction.text import TfidfVectorizer
```

- Create a `fit_tfidf` function used to build the TF-IDF vectorizer with the corpus

```
[ ]  def fit_tfidf(tweet_corpus):
       tf_vect = TfidfVectorizer(preprocessor=lambda x: x,
                                 tokenizer=lambda x: x)
       tf_vect.fit(tweet_corpus)
       return tf_vect
```

- Use the `fit_cv` function to fit the vectorizer on the corpus, and transform the corpus

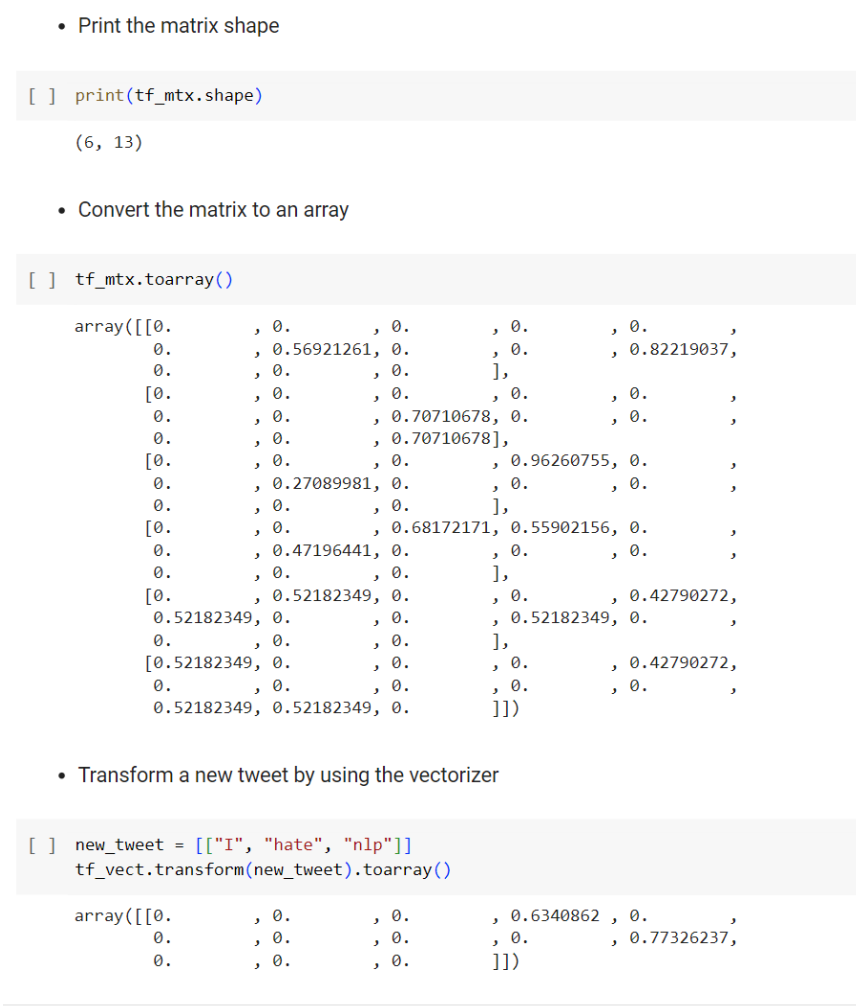
```
[ ]  tf_vect = fit_tfidf(corpus)
     tf_mtx = tf_vect.transform(corpus)
```

- Get the vectorizer features (matrix columns)

```
[ ]  ft = tf_vect.get_feature_names_out()
```

```
[ ]  print("There are {} features in this corpus".format(len(ft)))
     print(ft)

     There are 13 features in this corpus
     ['am' 'computer' 'happy' 'hate' 'i' 'lost' 'love' 'miss' 'my' 'nlp' 'sad'
      'so' 'you']
```

First step is to import the Vectorizer from the package. We then created a new TF-IDF function and fed the tweet_corpus. We then created a vectorizer by using the TF-IDFf Vectorizer. We used two lambdas, one for the preprocessor and one for the tokenizer. We then fit the vectorizer with the corpus that was fed to the function. Then we used the function to create the vectorizer based on our corpus and used it to create the transfer matrix for a corpus. We again checked the features by using the get_feature_names method. There are 13 features representing the different unique words present in the corpus.



- Print the matrix shape

```
[ ] print(tf_mtx.shape)
```

```
(6, 13)
```

- Convert the matrix to an array

```
[ ] tf_mtx.toarray()
```

```
array([[0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.56921261, 0.        , 0.        , 0.82219037,
        0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.70710678, 0.        , 0.        ,
        0.        , 0.        , 0.70710678],
       [0.        , 0.        , 0.        , 0.96260755, 0.        ,
        0.        , 0.27089981, 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.68172171, 0.55902156, 0.        ,
        0.        , 0.47196441, 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        ],
       [0.        , 0.52182349, 0.        , 0.        , 0.42790272,
        0.52182349, 0.        , 0.        , 0.52182349, 0.        ,
        0.        , 0.        , 0.        ],
       [0.52182349, 0.        , 0.        , 0.        , 0.42790272,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.52182349, 0.52182349, 0.        ]])
```

- Transform a new tweet by using the vectorizer

```
[ ] new_tweet = [["I", "hate", "nlp"]]
    tf_vect.transform(new_tweet).toarray()
```

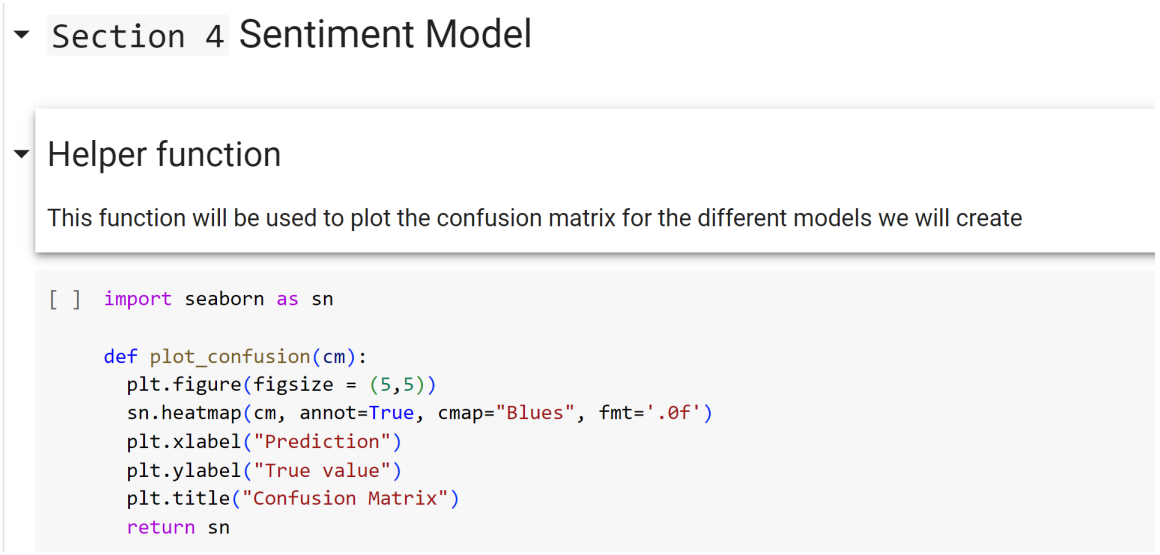
```
array([[0.        , 0.        , 0.        , 0.6340862 , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.77326237,
        0.        , 0.        , 0.        ]])
```

We checked the size and it is 6 by 13, which represents the number of tweets in the corpus times the features. We also checked the size of our transfer matrix, and visualized the matrix by using the toarray() method. We have decimal numbers and not integers like the bag of words. The more rare a word is in the corpus the higher the number present in the matrix would be. Finally, we transformed our new tweet not present in the corpus by using the transform function.

4. SENTIMENT MODEL

A sentiment model, also known as a sentiment analysis or opinion mining model, is a machine learning or natural language processing (NLP) algorithm designed to analyse and determine the sentiment expressed in a piece of text. The model's primary objective is to classify the text as positive, negative, or neutral based on the emotions or opinions conveyed. To build a sentiment model, the algorithm is typically trained on a large dataset of labelled text examples. The labelled data helps the model learn patterns and features that indicate positive or negative sentiment. Once trained, the sentiment model can be applied to new, unseen text to predict its sentiment.

Helper function



4.1 TRAIN/TEST SPLIT

The model is fitted on training data only, it is never trained on test data. For the simple reason that test data is only used during the testing phase to measure how well a trained model performs. The goal of a model is to be used on new data that has never been encountered. It is to check and evaluate the predictions made by the model on the test set.

4.1 Train/Test Split

- Check what X and y looks like

```
[ ]  print(X)
     print(y)
```

```
[['layin', 'n', 'bed', 'headach', 'ughh.waitin', 'call'], ['funer', 'ceremony.gloomi', 'f
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0
```

- Import the `train_test_split` function from the Scikit-Learn package

```
[ ]  from sklearn.model_selection import train_test_split
```

- Use the `train_test_split` function to split arrays of X and y into training and testing variables

```
[ ]  X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                         random_state=0,
                                                         train_size=0.80)
```

- Print the size of these news variables

```
[ ]  print("Size of X_train: {}".format(len(X_train)))
     print("Size of y_train: {}".format(len(y_train)))
     print("\n")
     print("Size of X_test: {}".format(len(X_test)))
     print("Size of y_test: {}".format(len(y_test)))
     print("\n")
     print("Train proportion: {:.0%}".format(len(X_train)/
                                             (len(X_train)+len(X_test))))
```

```
Size of X_train: 14981
Size of y_train: 14981


Size of X_test: 3746
Size of y_test: 3746


Train proportion: 80%
```

- Print random tweets, just to verify everything goes as expected

```
[ ]  id = random.randint(0,len(X_train))
     print("Train tweet: {}".format(X_train[id]))
     print("Sentiment: {}".format(y_train[id]))
```

```
Train tweet: ['watch', 'ellen', 'love', 'dish', 'tan', '.it', 'gorgeous']
Sentiment: 1
```

Our dataset is splitted between train and test data. As we can see, we are shown with the entire dataset, first the different tokenized tweets which are stored in variable X. And second, the different sentiments which are stored in variable Y.

Our objective is to split this entire dataset, both X and Y in two. The first part will be used to train the model and the second one will be used to test and evaluate how the trained model performs. We've done it using the *train_test_split* module. What this function does is simply splitting X and Y into four different variables x_train, x_test, y_train and y_test.

4.2 LOGISTIC REGRESSION

The statistical model we used to predict tweets sentiment is called logistic regression. Regression is a way to estimate the relationship between a dependent variable and one or several explanatory variables.

Model

4.2.1 Model

- Import the `LogisticRegression` model from Scikit-Learn

```
[ ]  from sklearn.linear_model import LogisticRegression
```

- Create a `fit_lr` function used to fit a Logistic Regression model on X and y *training* data

```
[ ]  def fit_lr(X_train, y_train):
         model = LogisticRegression()
         model.fit(X_train, y_train)
         return model
```

The first thing is to import the LogisticRegression classifier from the Scikit-Learn package. We then created a new function that will be used to train the logistic regression model. This function takes two variables as input, a train set x_train, which is the set of explanatory variables and a train set y_train, which is a set of dependent variables or simply the set of sentiments. We define the model, which is the LogisticRegression classifier. This model can simply be fitted by using x_train and y_train together with the fit function. Remember, fitting the model means finding the optimal set of beta parameters, so the relationship between x_train and y_train is as accurately described as possible.
Let's now try to use this function to train three different models.

Positive/Negative Frequency

4.2.2 Pos/Neg Frequency

- Use the `build_freqs` function on training data to create a frequency dictionnary
- Use the frequency dictionnary together with the `tweet_to_freq` function to convert X_train and X_test data to 2-d vectors

```
[ ]  freqs = build_freqs(X_train, y_train)
     X_train_pn = [tweet_to_freq(tweet, freqs) for tweet in X_train]
     X_test_pn = [tweet_to_freq(tweet, freqs) for tweet in X_test]
```

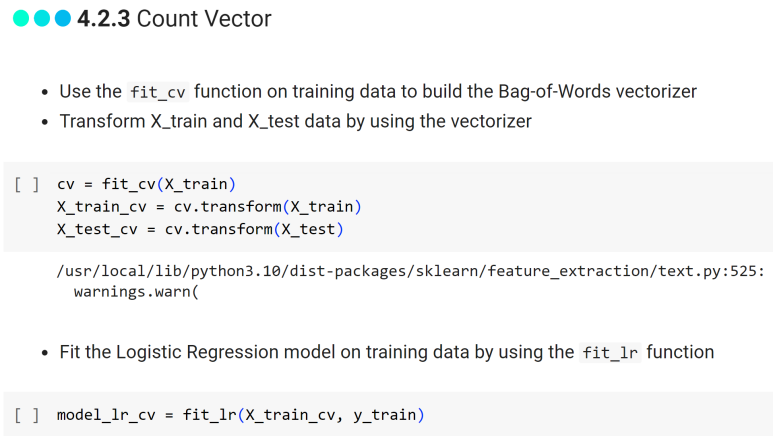
- Fit the Logistic Regression model on training data by using the `fit_lr` function
- Print the model coefficients (betas and intercept)

```
[ ]  model_lr_pn = fit_lr(X_train_pn, y_train)
     print(model_lr_pn.coef_, model_lr_pn.intercept_)

     [[ 0.00252129 -0.00189421]] [-0.54728242]
```

First method is positive/negative frequencies. This method requires a frequency dictionary to look up words based on sentiment. We can build such a dictionary by using the *build_freqs*

function. We built such a dictionary based on the training set and not the entire set. This is to prevent data leakage as the model needs to be trained on training data only. And it's not supposed to access any of the test data. We then created a new *x_train_pn* variable by converting each tweet in the train set to a frequency vector. We did the same for the test set. Data is now ready to be used. We created a specific model variable with our x_train set and with our y_train set. And printed model coefficient. Beta's coefficients are opposite but of similar magnitude, which is great. This means the prediction will mostly depend on the actual positive or negative frequencies found in the vector.

Count Vector



4.2.3 Count Vector

- Use the `fit_cv` function on training data to build the Bag-of-Words vectorizer
- Transform X_train and X_test data by using the vectorizer

```
[ ]  cv = fit_cv(X_train)
     X_train_cv = cv.transform(X_train)
     X_test_cv = cv.transform(X_test)
```

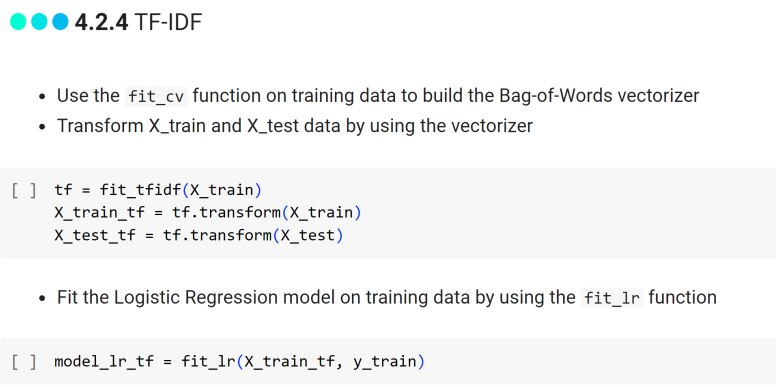
```
/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:525:
  warnings.warn(
```

- Fit the Logistic Regression model on training data by using the `fit_lr` function

```
[ ]  model_lr_cv = fit_lr(X_train_cv, y_train)
```

Let's have a look at the second method, bag of words. First step is to fit the count vectorizer on the training set. We fit the vectorizer on x_train only, so there is no data leakage, we can then use the fitted vector to transform our x_train and x_test variables. Both x_train and x_test are now represented as a matrix of numbers. Again, we can simply create a new model by fitting x_train count vectorizer with y_train. We want to bring the coefficients in this case as they will be as many Betas as there are columns in the Matrix.

TF-IDF



4.2.4 TF-IDF

- Use the `fit_cv` function on training data to build the Bag-of-Words vectorizer
- Transform X_train and X_test data by using the vectorizer

```
[ ]  tf = fit_tfidf(X_train)
     X_train_tf = tf.transform(X_train)
     X_test_tf = tf.transform(X_test)
```

- Fit the Logistic Regression model on training data by using the `fit_lr` function

```
[ ]  model_lr_tf = fit_lr(X_train_tf, y_train)
```

Finally, we used the third method TF-IDF, very similar to bag of words, the TF-IDF vectorizer can be fitted by using x-train. *x_train_tf* and *x_test_tf* can then be created by transforming

x_train and x_test. We finally created our third model by fitting *x_train_tf* with y_train. And there we go, we now have three different models with different beta coefficients.

4.3 PERFORMANCE METRICS

4.3 Performance Metrics

- Import the `accuracy score` and `confusion matrix` from Scikit-Learn

```
[ ]  from sklearn.metrics import accuracy_score
     from sklearn.metrics import confusion_matrix
```

We evaluated our models by using the accuracy score and confusion matrix. For that, we imported the accuracy score and confusion matrix from Scikit-Learn.

Positive/Negative Frequencies

4.3.1 Positive/Negative Frequencies

- Use the fitted `model_lr_pn` (positive/negative frequencies) to predict X_test

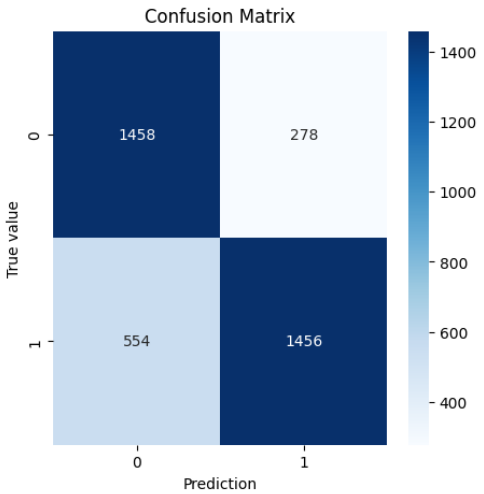
```
[ ]  y_pred_lr_pn = model_lr_pn.predict(X_test_pn)
```

- Print the model accuracy by comparing predictions and real sentiments

```
[ ]  print("LR Model Accuracy: {:.2%}".format(accuracy_score(y_test, y_pred_lr_pn)))

     LR Model Accuracy: 77.79%
```

- Plot the confusion matrix by using the `plot_confusion` helper function

```
[ ]  plot_confusion(confusion_matrix(y_test, y_pred_lr_pn))

     <module 'seaborn' from '/usr/local/lib/python3.10/dist-packages/seaborn/__init__.py'>
```



Now, it is the test set that we are going to use to evaluate our model. It wouldn't really make sense to evaluate the model on data from which it was created anyway. So, we created a new variable y prediction and predicted our test data by using our model. This can be done very easily by using the predict function. We need to compare predictions with actual values. And

that's the reason why we splitted our dataset into a training and testing set in the first place. We then used the accuracy score metric and checked how accurately our predictions match the actual sentiment of the test set. We then plotted the confusion matrix by using the plot_confusion helper function. All right, so we have a model with around 70-80 percent accuracy. But let's see if we do better with the other two methods.

Count Vector

4.3.2 Count Vector

- Use the fitted `model_lr_cv` (Bag-of-words) to predict X_test

```
[ ] y_pred_lr_cv = model_lr_cv.predict(X_test_cv)
```

- Print the model accuracy by comparing predictions and real sentiments

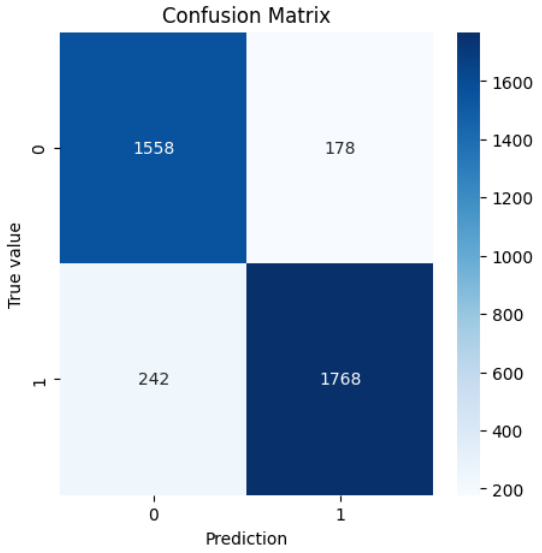
```
[ ] print("LR Model Accuracy: {:.2%}".format(accuracy_score(y_test, y_pred_lr_cv)))
```

```
LR Model Accuracy: 88.79%
```

- Plot the confusion matrix by using the `plot_confusion` helper function

```
[ ] plot_confusion(confusion_matrix(y_test, y_pred_lr_cv))
```

```
<module 'seaborn' from '/usr/local/lib/python3.10/dist-packages/seaborn/__init__.py'>
```



Second method is a bag of words, we predicted our sentiments by using the count vectorizer model and the correct test set. We then printed the accuracy and the confusion matrix. This time we reached an accuracy of almost 88.79%.

TF-IDF

4.3.3 TF-IDF

- Use the fitted `model_lr_tf` (TF-IDF) to predict X_test

```
[ ] y_pred_lr_tf = model_lr_tf.predict(X_test_tf)
```

- Print the model accuracy by comparing predictions and real sentiments

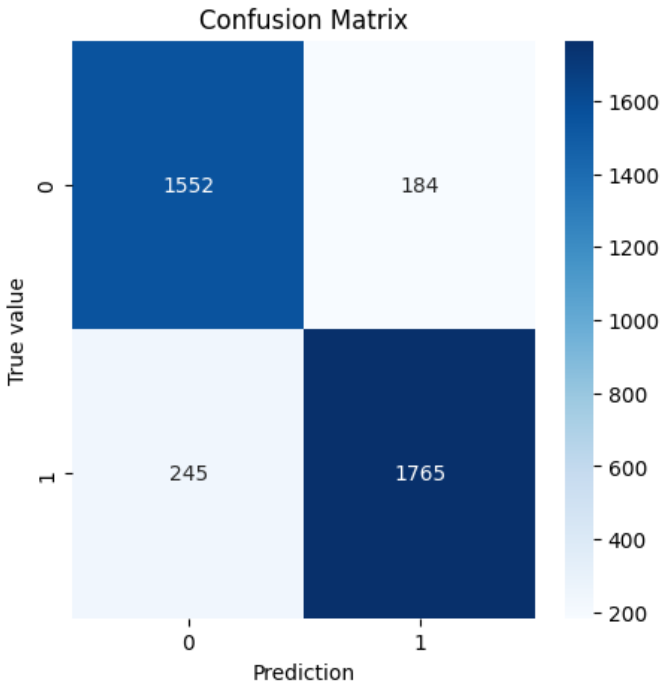
```
[ ] print("LR Model Accuracy: {:.2%}".format(accuracy_score(y_test, y_pred_lr_tf)))
```

```
LR Model Accuracy: 88.55%
```

- Plot the confusion matrix by using the `plot_confusion` helper function

```
[ ] plot_confusion(confusion_matrix(y_test, y_pred_lr_tf))
```

```
<module 'seaborn' from '/usr/local/lib/python3.10/dist-packages/seaborn/__init__.py'>
```



Let's see if we can improve it even more. The results are still better, but still quite close. We only saw a bit more of false positive and a bit less of false negative. The real conclusion is that bag of words and TF-IDF seems to be more accurate than positive and negative frequencies. Again, this does not mean that these are the only or the best methods, but it seems sufficiently accurate for the type of application we want to develop.

4.4 MINI PIPELINE

4.4 Mini-Pipeline

- Final tweet used to check if the model works as well as expected
- **Note:** don't hesitate to input your own tweet!

```
[1]  your_tweet = """RT @AIOutsider: tune in for more amazing NLP content!
     And don't forget to visit https://AIOutsider.com ..."""
```

- Create a `predict_tweet` function used to pre-process, transform and predict tweet sentiment

```
[ ]  def predict_tweet(tweet):
       processed_tweet = process_tweet(tweet)
       transformed_tweet = tf.transform([processed_tweet])
       prediction = model_lr_tf.predict(transformed_tweet)

       if prediction == 1:
         return "Prediction is positive sentiment"
       else:
         return "Prediction is negative sentiment"
```

- ... Predict your tweet sentiment by using the `predict_tweet` function!

```
[ ]  predict_tweet(your_tweet)

     'Prediction is positive sentiment'
```

MiniPipeline is flexible, allowing for generic graph topologies to fit any workflow. A data pipeline is a set of tools that are used to extract, transform, and load data from one or more sources into a target system. We worked through the different steps to predict the sentiment of a piece of text, we cleaned an entire dataset and we learned how to represent it so it could be used to train the machine learning model. Now is the time we tested it with real, custom and unknown tweets.

First, we need to pre-process our tweets by using our *process_tweet* function. We can then transform this preprocess tweet to a vector of numbers by using our TF-IDF vectorizer, which appears to be the most accurate method. We can then simply use our trained logistic regression model to make a prediction about the transform tweet. The return number will be 1 for positive and 0 for negative, which allows us to return the specific message based on the result. Our pipeline is ready to predict any text string.

CONCLUSIONS AND FUTURE SCOPE

Conclusion

Sentiment analysis is used to identify people's opinion, attitude and emotional states. The views of the people can be positive or negative. Commonly, parts of speech are used as feature to extract the sentiment of the text. An adjective plays a crucial role in identifying sentiment from parts of speech. Sometimes words having adjective and adverb are used together, so it is difficult to identify sentiment and opinion.

To do the sentiment analysis of tweets, the proposed system first extracts the twitter posts from twitter by user. The system can also compute the frequency of each term in a tweet. Using a machine learning supervised approach helps to obtain the results. Twitter is a large source of data, which makes it more attractive for performing sentiment analysis. We perform analysis on around 18,727 tweets total for each party, so that we analyse the results, understand the patterns and give a review on people's opinion. We saw different parties have different sentiment results according to their progress and working procedure. We also saw how any social event, speech or rally causes a fluctuation in sentiment of people. We also get to know which policies are getting more support from people which are started by any of these parties. It is not necessary that our model can only be used for political parties. It can be used for any purpose based on tweets we collect with the help of keywords. It can be used for finance, marketing, reviewing and many more.

Future Scope

Some of future scopes that can be included in our research work are:

- **Domain-specific Sentiment Analysis:** Tailoring sentiment analysis models to specific domains like healthcare, finance, politics, and more can provide more accurate and specialised sentiment insights.
- **Social Media and Brand Management:** Businesses are keen to monitor their brand sentiment on social media platforms. Developing tools that provide real-time sentiment analysis for brand management and customer engagement is a promising application.
- **Market Research and Predictive Analytics:** Sentiment analysis can be used to gauge public opinion and predict market trends. It can also help businesses understand consumer preferences and improve products or services.
- **Healthcare Applications:** Sentiment analysis can be used to monitor patients' emotional states through their communication, aiding mental health professionals and healthcare providers.

REFERENCES

1. Go, A., Bhayani, R., & Huang, L. (2009). Twitter sentiment classification using distant supervision. CS224N Project Report, Stanford University.
   Link: https://cs224d.stanford.edu/reports/ArunBhayani.pdf

2. Pak, A., & Paroubek, P. (2010). Twitter as a corpus for sentiment analysis and opinion mining. Proceedings of LREC, 10, 1320-1326.
   Link: https://www.lrec-conf.org/proceedings/lrec2010/pdf/866_Paper.pdf

3. Tala, F. Z., & De Pauw, G. (2019). Twitter sentiment analysis using a simple logistic regression model. International Journal of Computer Applications, 186(7), 11-16.
   Link: https://www.ijcaonline.org/archives/volume186/number7/31040-2019915512

4. "Sentiment Analysis with Logistic Regression." Medium.com
   Link:
   https://towardsdatascience.com/sentiment-analysis-with-logistic-regression-7784e18c8d49

5. Sharma, S., Jha, M. K., & Rani, M. (2018). Sentiment analysis of twitter data using logistic regression model. International Journal of Computer Applications, 181(40), 1-5.
   Link: https://www.ijcaonline.org/archives/volume181/number40/29792-2018911594

6. Saif, H., Fernandez, M., He, Y., & Alani, H. (2013). Evaluation datasets for Twitter sentiment analysis: A survey and a new dataset, the STS-Gold. Proceedings of the 1st International Workshop on Emotion and Sentiment in Social and Expressive Media (ESSEM), 81-91.
   Link:
   https://www.researchgate.net/publication/266712153_Evaluation_Datasets_for_Twitter_Sentiment_Analysis_A_Survey_and_a_New_Dataset_the_STS-Gold

7. "Twitter Sentiment Analysis with Logistic Regression and XGBoost" by Kevin Morgado. This blog post from Kaggle describes how to build a Twitter sentiment analysis model using logistic regression and XGBoost. https://www.kaggle.com/code/kevinmorgado/twitter-sentiment-analysis-logistic-regression

8. "A Logistic Regression Approach to Sentiment Analysis on Twitter" by Haotian Zhang, Wenshuo Wang, and Xueqi Cheng. This paper from the University of Pennsylvania describes the use of logistic regression for sentiment analysis of Twitter data. https://arxiv.org/abs/1608.04862