Sentiment Analysis on Twitter

Robin KHATIB

Data Mining

University of Rijeka

2024-2025

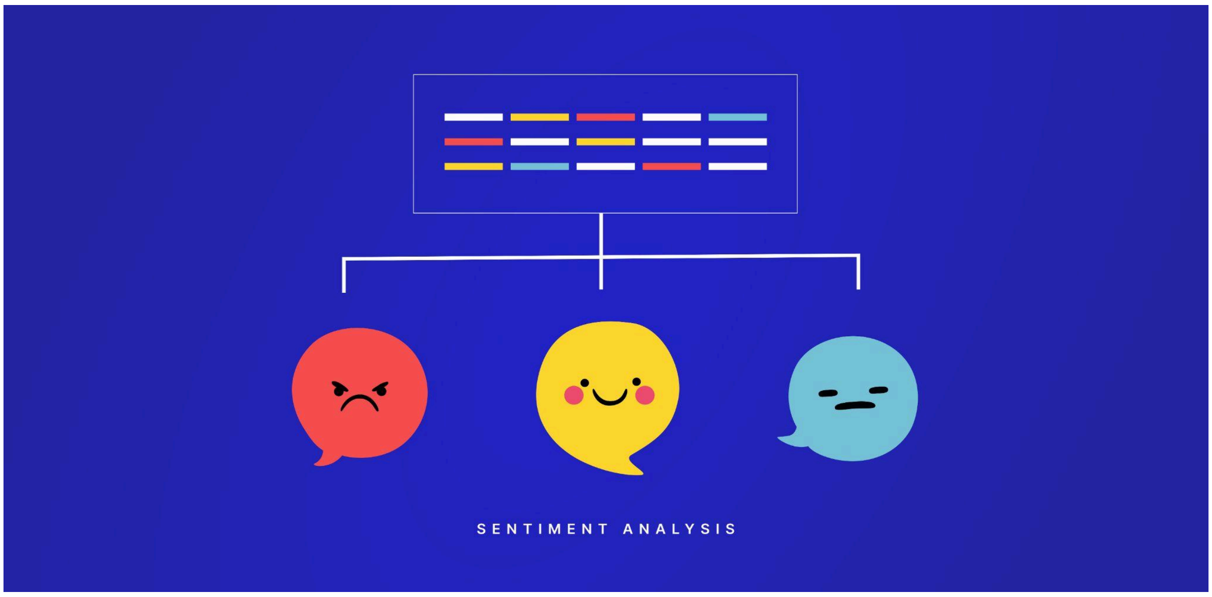


Table of Contents

Introduction

Sentiment analysis is a kind of opinion detection that tries to find and sort the emotion in a text. Studying sentiment on Twitter helps us understand what people think about different topics, see how opinions change with time, and maybe even guess future social or economic trends.

Because Twitter data comes very fast and in big amounts, normal tools don't work well enough. For this reason, **Apache Spark** is a good tool to use. It is a strong system for big data that works in parallel on many computers. Spark can do fast in-memory work and also supports machine learning tools, which makes it good for doing real-time sentiment tasks.

In this paper, I will use Python code as a common thread to guide the analysis and development of the project. At each step, I will explain the concepts that are used such as the algorithms or the data mining mechanism behind it. Sometimes the code parts are too big to be in this paper so I put the link to the code just below this, you also have a final presentation video of the code where you can find all the parts of the code.

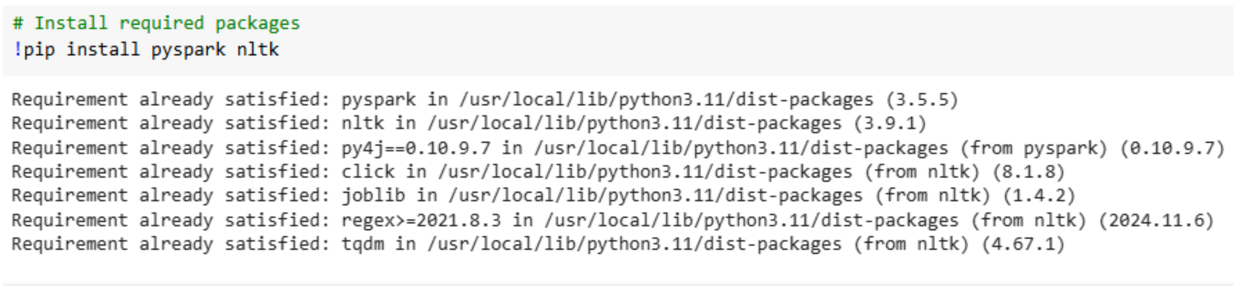
The goal is not only to implement a working solution, but also to deeply understand how and why it works. The report will conclude with a practical demonstration of the final code to showcase the results in action.

The code :
https://colab.research.google.com/drive/1csB3BvnNxlPK8c1c3SEgGipG3ZHlvNFk

Preparing the environment for the Analysis

To begin the sentiment analysis process, we first need to set up the technical environment. This includes installing the necessary tools and preparing the data so that it can be processed and analyzed correctly.

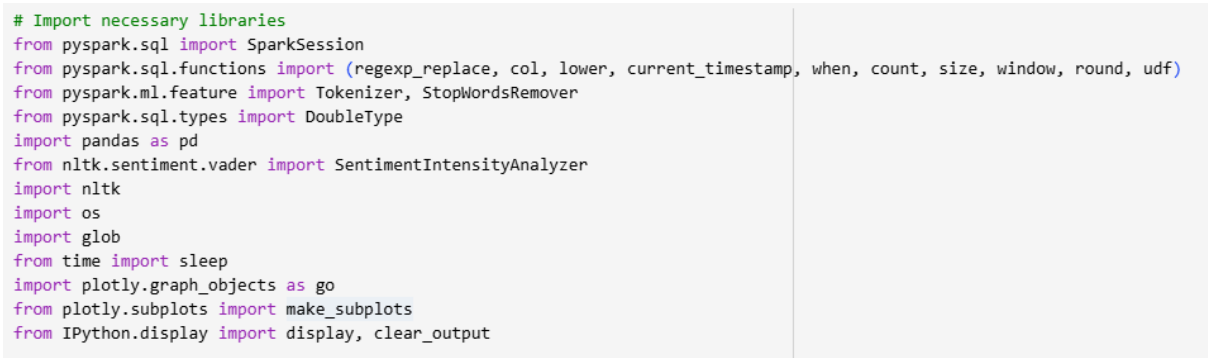
Firstly we will work on Google Colab because of its practicality. We will install both package pyspark and nltk. Pyspark is essential because it is the python version of Apache Spark which is a framework to manage a massive amount of data in an efficient way. Here we will use it to read, transform and clean the tweets at big scale, tokenization and execution of the analysis. Spark is a key thing here because of the amount of data that we manipulate.
We will need nltk (Natural language Toolkit) because we will use VADER, a tool that will help us for the analysis of sentiment.



```
# Install required packages
!pip install pyspark nltk
```

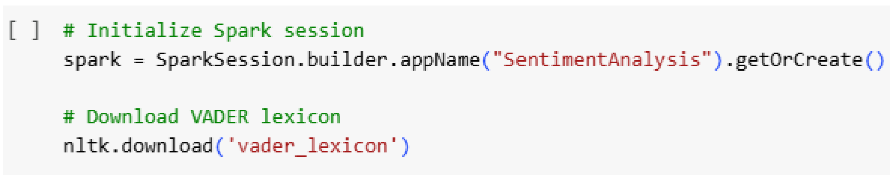
```
Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.5)
Requirement already satisfied: nltk in /usr/local/lib/python3.11/dist-packages (3.9.1)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.11/dist-packages (from pyspark) (0.10.9.7)
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-packages (from nltk) (8.1.8)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (from nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.11/dist-packages (from nltk) (2024.11.6)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from nltk) (4.67.1)
```

Then we will install the library of these packages and more libraries like pandas to read CSV files, plotly to visualise in real time the evolution of sentiments and os/glob/time/ sleep to manage the file and simulate the real time analysis. IPython.display is there to dynamically update the graphic in a notebook.



```python
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import (regexp_replace, col, lower, current_timestamp, when, count, size, window, round, udf)
from pyspark.ml.feature import Tokenizer, StopWordsRemover
from pyspark.sql.types import DoubleType
import pandas as pd
from nltk.sentiment.vader import SentimentIntensityAnalyzer
import nltk
import os
import glob
from time import sleep
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from IPython.display import display, clear_output
```

We start the Spark session and download VADER to classify the tweets later :



```
[ ]  # Initialize Spark session
     spark = SparkSession.builder.appName("SentimentAnalysis").getOrCreate()

     # Download VADER lexicon
     nltk.download('vader_lexicon')
```

Once the environment is ready, the next step is to collect and load the data that will be used for the sentiment analysis. Let's see how we will do that, because here the data must be in real time and on X (twitter).

Get the data

The main question here is how can we get the data, the tweets and furthermore in real time. Let's go step by step, first we need to get tweets, to do that we will use the API of X (Twitter).

Twitter API v2 is a tool made by Twitter that lets developers get access to its data (tweets, users, trends, etc.) by sending requests. It is made to help collect, check, and use Twitter data in a flexible way. Theses are the features/advantages of the X APi that will be useful for us :

- **Advanced search and filters:** Get only the tweets that match specific rules.
- **Real-time tweet analysis:** Catch tweets as they are posted using the *Filtered Stream*.
- **Enriched data:** Extra info added to tweets like number of likes, retweets, mentions, etc.
- **Good for modern projects:** Useful for big analysis or real-time needs.

How is it working ? Well there is both key that allows us to use the API : API Key & API Key Secret
 They are unique codes used to identify an app when it talks to the Twitter API. It's like a "username" and "password" that let your app connect and work with Twitter's servers.

To resume :

- **API Key:** This is the main thing to start a connection with the API. It tells Twitter which app is trying to connect.
- **API Key Secret:** It works like a password for the API Key. It helps make sure the app is allowed by the developer.

Then we have the bearer token which is a key access generated automatically from the API key and de Api key secret. Is it used to simplify the authentication of the app when we just use the read only of the API.

To do that we need to create a twitter developer account and ask for approval, and then create an app on the developer dashboard.

Then we can generate the keys and the bearer token (which is the most important token for requests), we save the key in a safe file to avoid losing them.

The first big issue occurred now, indeed the access to Twitter/X API has become more limited. Now, many advanced features like the real-time *Filtered Stream* need a paid subscription. Free access often gives errors like 401 or 403 when permissions are missing. The free plan also has strict limits, like only 500 requests or 100 tweets per month, which is too low for serious analysis. Some important tools and data, like full metadata or live tweets, are no longer free and are only available with paid plans like the Basic Plan, which costs $100 per month.

Now we have another option, we can use a dataset of tweets that we find on Kaggle and simulate it like it's the tweets that are coming. So the first part of our pipeline will not be exactly in "real time", but we will simulate the analysis and the visualization in real time. If we didn't have any restrictions we could get the tweets in real time with the API of X, put them in a real time Database such as Cassandra and after it's the same process that we will use (instead of coming from Cassandra it will come from a csv file loaded on Google Colab.

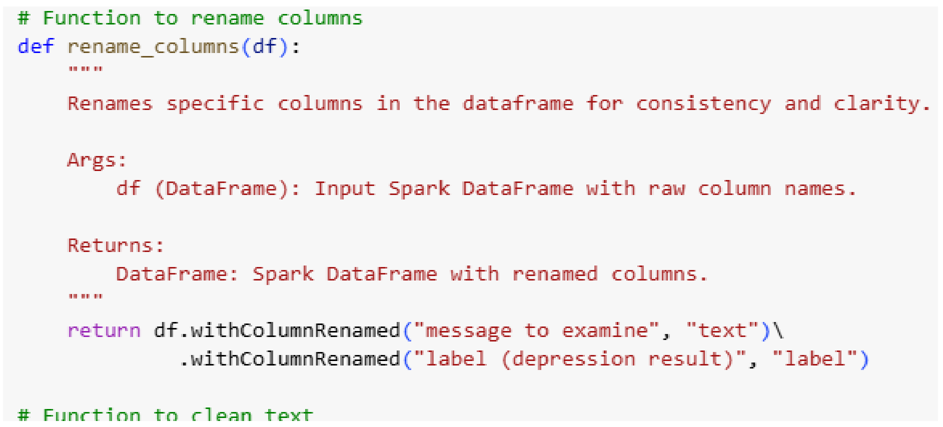
After collecting the data, it is necessary to clean and prepare the text so that it can be analyzed properly. This step is essential to improve the quality and accuracy of the sentiment analysis results. Let's see how it works.

The dataset that i will use :
https://drive.google.com/file/d/1cr9GjOUK-j5cV-s6mz2i1aYjJbtFz16B/view?usp=drive_link

Preprocessing of textual data

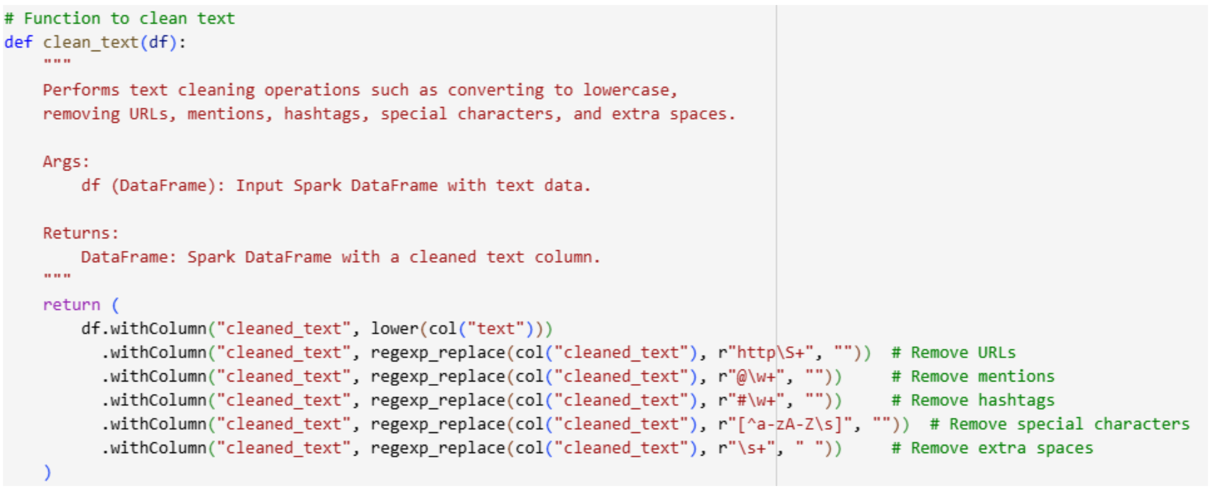
So the goal here is to make tweets analysable by a machine learning algorithm.

Firstly we rename columns :



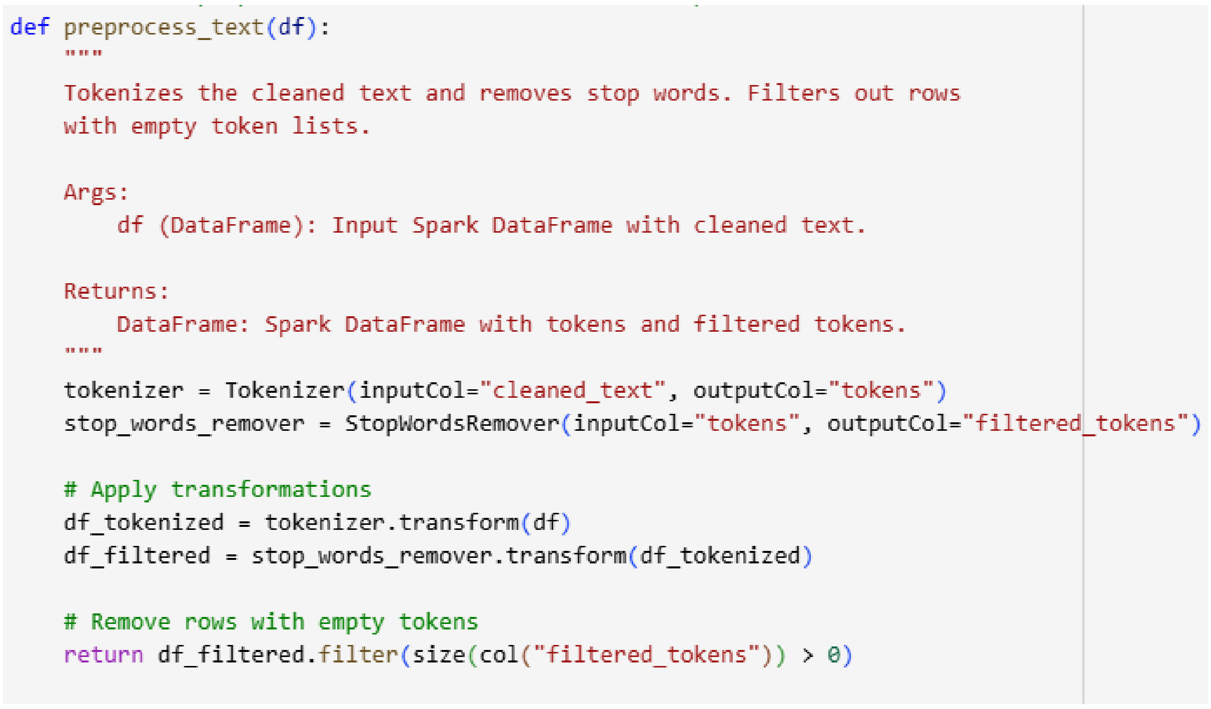
```python
# Function to rename columns
def rename_columns(df):
    """
    Renames specific columns in the dataframe for consistency and clarity.

    Args:
        df (DataFrame): Input Spark DataFrame with raw column names.

    Returns:
        DataFrame: Spark DataFrame with renamed columns.
    """
    return df.withColumnRenamed("message to examine", "text")\
             .withColumnRenamed("label (depression result)", "label")

# Function to clean text
```

Then we code a function to clean the text, meaning that we convert to lowercase symbols, removings urls, mentions…



```python
# Function to clean text
def clean_text(df):
    """
    Performs text cleaning operations such as converting to lowercase,
    removing URLs, mentions, hashtags, special characters, and extra spaces.

    Args:
        df (DataFrame): Input Spark DataFrame with text data.

    Returns:
        DataFrame: Spark DataFrame with a cleaned text column.
    """
    return (
        df.withColumn("cleaned_text", lower(col("text")))
          .withColumn("cleaned_text", regexp_replace(col("cleaned_text"), r"http\S+", ""))  # Remove URLs
          .withColumn("cleaned_text", regexp_replace(col("cleaned_text"), r"@\w+", ""))     # Remove mentions
          .withColumn("cleaned_text", regexp_replace(col("cleaned_text"), r"#\w+", ""))     # Remove hashtags
          .withColumn("cleaned_text", regexp_replace(col("cleaned_text"), r"[^a-zA-Z\s]", ""))  # Remove special characters
          .withColumn("cleaned_text", regexp_replace(col("cleaned_text"), r"\s+", " "))     # Remove extra spaces
    )
```

We need to tokenize the text and remove stop words. Tokenize is the process that cuts the text into small units called “tokens”. It’s necessary here because VADER will analyse each word and put a weight on it, so every word should be separated. The removal of stop words is the removal of all frequent words that are not bringing some information like “the, it, on, of…”. With these two operations we will have text appropriate to be analysis.



```python
def preprocess_text(df):
    """
    Tokenizes the cleaned text and removes stop words. Filters out rows
    with empty token lists.

    Args:
        df (DataFrame): Input Spark DataFrame with cleaned text.

    Returns:
        DataFrame: Spark DataFrame with tokens and filtered tokens.
    """
    tokenizer = Tokenizer(inputCol="cleaned_text", outputCol="tokens")
    stop_words_remover = StopWordsRemover(inputCol="tokens", outputCol="filtered_tokens")

    # Apply transformations
    df_tokenized = tokenizer.transform(df)
    df_filtered = stop_words_remover.transform(df_tokenized)

    # Remove rows with empty tokens
    return df_filtered.filter(size(col("filtered_tokens")) > 0)
```

Once the text data is preprocessed and cleaned, we can move on to the core part of the project: performing sentiment analysis using different models and evaluating their performance.

Sentiment Analysis

Now we will start the analysis of the dataset, we will present and compare two different approaches to sentiment classification:

- **VADER**, a rule-based model that uses a predefined sentiment lexicon, and
- **Logistic Regression**, a supervised machine learning model trained on labeled data.

In sentiment analysis, choosing the right model is essential, especially when dealing with sensitive topics. To make the best decision, we will compare these different models using evaluation metrics such as **accuracy**, **recall**, and **F1-score**. These metrics help us measure how well each model can detect emotional expressions in the tweets.

By evaluating both **VADER** (a rule-based sentiment model) and **Logistic Regression** (a supervised machine learning model) with the same metrics we will compare them.. This comparison will guide us in selecting the most suitable model for the final sentiment classification task.

Before comparing the models, it is important to first understand the evaluation metrics that will be used. In the next section, we explain how **accuracy**, **recall**, and **F1-score** are calculated, and why they are important for assessing the performance of sentiment analysis models.

Metrics on models

So first let's talk about accuracy. It's the main metric of a model that measures the percentage of good predictions out of all predictions. The formula is the following :

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{TP + TN}{TP + TN + FP + FN}$$

With True Positive (TP) / Negative (TN) and False Positive (FP) / Negative (FN)

Nevertheless we should be careful because accuracy alone is not enough to detect if the model is efficient or not.
Let's give an example to show the limitations of it. If we have a model of 100 predictions, 90 of them are correct and 10 not. The accuracy would be 0.90 (90%), but we can have this accuracy with 90% of the data were on class A and 10 on class B and the model predicts

always A. It’s a big issue because we think our model is accurate but in reality it just misses all B so it’s a really bad prediction.

To get a better understanding of how good the model is and to avoid this issue we can consider the Recall metric.Here it’s important because missing a positive case is costly (undetected depression on a tweet), we want to minimize false negatives.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

So it calculates out of all the actual positive cases, how many did the model correctly identify. It tells you how good your model is at finding the positives.

The problem now is that these two metrics can conflict, High recall but low precision = you find most positives, but with many false alarms. High precision but low recall = you're very sure when you say "positive", but miss a lot.

That’s why another metric is here, the F1-Score, it’s the harmonic mean of precision and recall, the formula is given by :

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

It gives you one number that Is low if either precision or recall is low, and only high if both are good.
We keep that in mind to compare our VADER model and the logistic regression one.

Sentiment Analysis with VADER

So we start to do the analysis with VADER, but before that let’s see how VADER works theoretically.

The mechanism behind VADER

Here we use VADER which is a natural language processing tool. It's a field of artificial intelligence which aims to understand, generate and translate the human language. It’s often used to analyze sentiment and it’s the best thing here. Nevertheless VADER is not a machine learning algorithm, it’s using a lexic and some heuristic rules to put some weight on words that he goes through. We can use a logistic regression too to see the difference in the efficiency of the analysis.

Let’s see how VADER is working in detail. Firstly VADER has a dictionary of more than 7500 words and idioms where each word has a sentiment score between -4(very negative) and 4 (very positive).

Then VADER also considers the adverbs before the words such as “very good” or “barely good”, VADER has a list of adverbs with a multiplicator coefficient.

The punctuation such as “!” or “?” is taken into consideration. For example if a word is followed by “!” it will amplify its score. The number of punctuation is also taken into account up to 4.

Then a word write in upper case is consider stronger than the other”s (only if it is not surrounded by upper case words too)

VADER takes the negation into account, if it sees “not”, “never”, “isn’t”... It will invert or reduce the score of the following word. Conjunctions are also considered, words before “but” are reduced and after they are amplified.

Considering all these rules let’s make an example with a sentence to see how it works.

This sentence will be our example :

"I absolutely loved the new phone, but the battery life is not good and the camera is terrible."

Firstly with the removal of stop words we don't take into consideration “I”, “the”, “is” .

This become “ absolutely loved new phone, but battery life not good and camera terrible”

Then VADER calculates a score for each of these words.

Word	Lexicon base score	Rule applied	Final score
absolutely	intensifier	Boosts "loved" by ×1.293	—
loved	+3.2	×1.293 due to "absolutely"	+4.14
new	+1.2	No rule	+1.2
phone	0	Neutral	—
but	—	Weakens prior clause (emphasis	—

		shift)	
battery	0	Neutral	—
life	0	Neutral	—
not	negation	Inverts the polarity of next word	—
good	+1.9	Negated → becomes –1.9	–1.9
and	—	Neutral conjunction	—
camera	0	Neutral	—
terrible	–2.1	No rule	–2.1

Then we sum all of the final scores :

x= 4.14+1.2-1.9-2.1 = 1.34

Then we calculate the compound of x to get a value between -1 and 1 that we can interpret easily.

The formula for the compound is :

$$\text{compound} = \frac{x}{\sqrt{x^2 + \alpha}}, \quad \alpha = 15$$

Alpha is a smoothing constant, it prevents the raw score x from taking extreme value by making the function more stable.

Then we calculate, compound(x) = 0.327. So we can say the sentiment of this sentence is slightly positive (0 would be neutral).

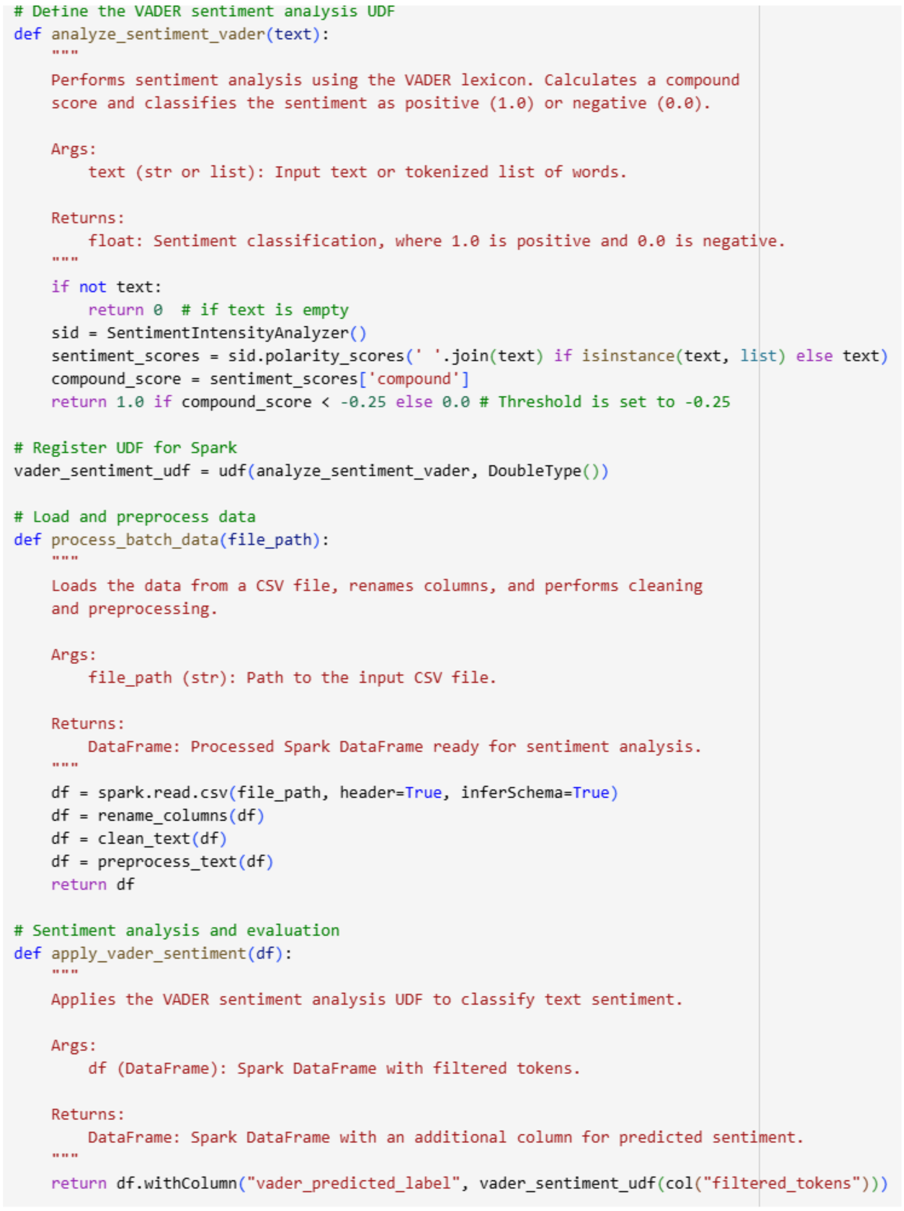
VADER does that for all the sentences in the text (the tweet here) and gets a score of the tweet to deduce the emotion.

Now let’s see how we implement that in python.

The implementation of VADER in python

To initialize VADER we implement an UDF Spark, it's a function that we create and then we transform it into a UDF to use it in the framework SPark. It's better to take the function that is already created, but if you really need a specific one you can do that.

The code of the implementation of Vader look like this :



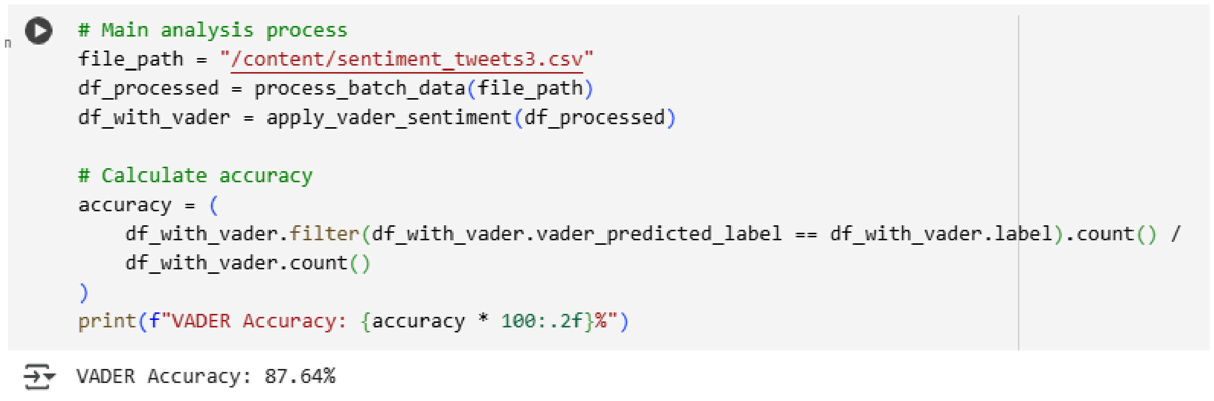
```python
# Define the VADER sentiment analysis UDF
def analyze_sentiment_vader(text):
    """
    Performs sentiment analysis using the VADER lexicon. Calculates a compound
    score and classifies the sentiment as positive (1.0) or negative (0.0).

    Args:
        text (str or list): Input text or tokenized list of words.

    Returns:
        float: Sentiment classification, where 1.0 is positive and 0.0 is negative.
    """
    if not text:
        return 0  # if text is empty
    sid = SentimentIntensityAnalyzer()
    sentiment_scores = sid.polarity_scores(' '.join(text) if isinstance(text, list) else text)
    compound_score = sentiment_scores['compound']
    return 1.0 if compound_score < -0.25 else 0.0 # Threshold is set to -0.25

# Register UDF for Spark
vader_sentiment_udf = udf(analyze_sentiment_vader, DoubleType())

# Load and preprocess data
def process_batch_data(file_path):
    """
    Loads the data from a CSV file, renames columns, and performs cleaning
    and preprocessing.

    Args:
        file_path (str): Path to the input CSV file.

    Returns:
        DataFrame: Processed Spark DataFrame ready for sentiment analysis.
    """
    df = spark.read.csv(file_path, header=True, inferSchema=True)
    df = rename_columns(df)
    df = clean_text(df)
    df = preprocess_text(df)
    return df

# Sentiment analysis and evaluation
def apply_vader_sentiment(df):
    """
    Applies the VADER sentiment analysis UDF to classify text sentiment.

    Args:
        df (DataFrame): Spark DataFrame with filtered tokens.

    Returns:
        DataFrame: Spark DataFrame with an additional column for predicted sentiment.
    """
    return df.withColumn("vader_predicted_label", vader_sentiment_udf(col("filtered_tokens")))
```

Then we apply the model on our dataset of tweets.

Now we want to see how good our model is, so we need to implement the metrics that we saw earlier, let's see how we do that in python.

Implementation of major metrics in python

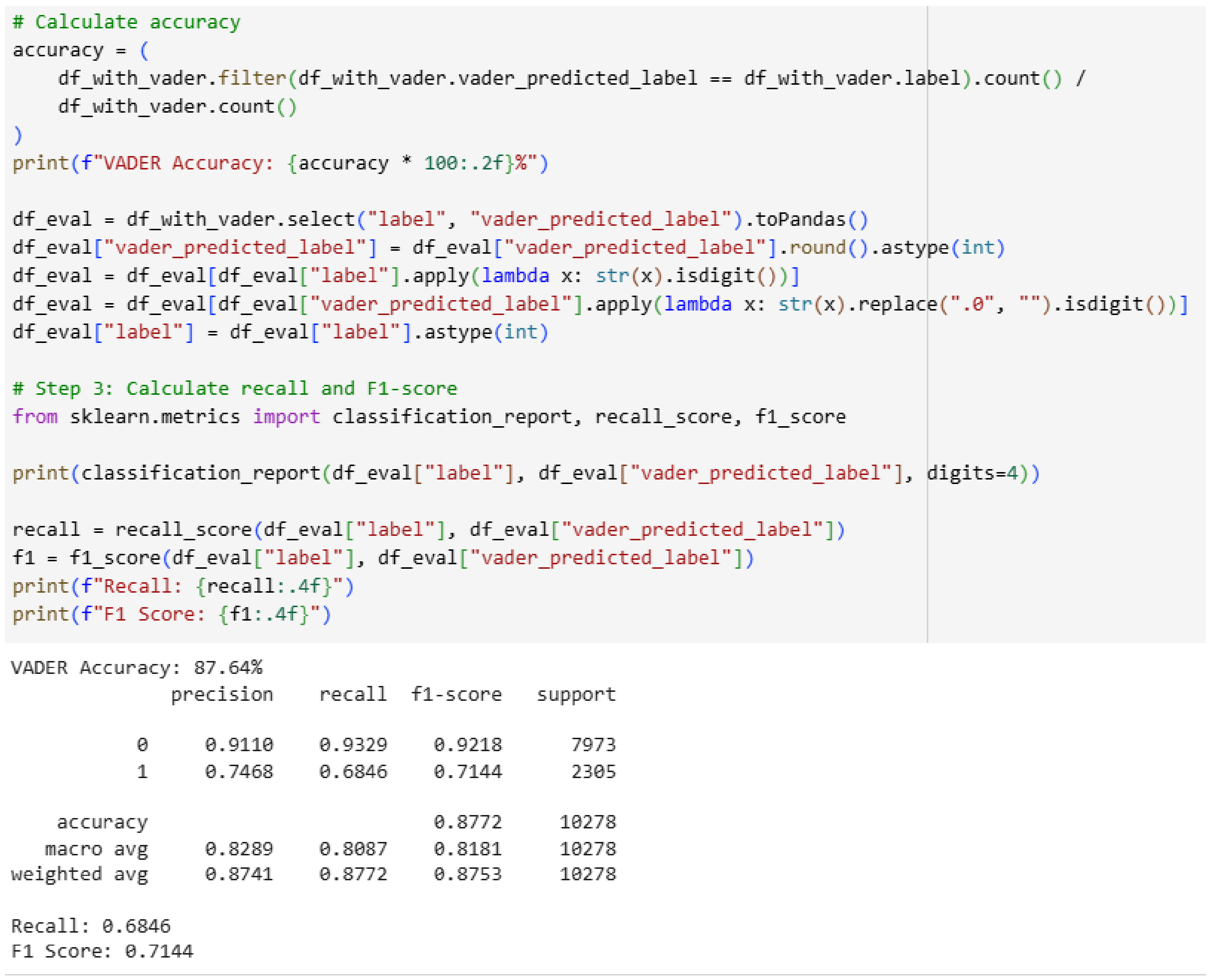
Therefore we calculate the accuracy of the model which gives us the percentage of good prediction out of all the predictions.



```python
# Main analysis process
file_path = "/content/sentiment_tweets3.csv"
df_processed = process_batch_data(file_path)
df_with_vader = apply_vader_sentiment(df_processed)

# Calculate accuracy
accuracy = (
    df_with_vader.filter(df_with_vader.vader_predicted_label == df_with_vader.label).count() /
    df_with_vader.count()
)
print(f"VADER Accuracy: {accuracy * 100:.2f}%")
```

```
VADER Accuracy: 87.64%
```

Here we saw that the accuracy obtained is 87.64% which is pretty good.
To calculate the Recall and the F1 score we need to get the confusion matrix to get each value of True Positive, False Positive, False negative and False Positive.



```python
# Calculate accuracy
accuracy = (
    df_with_vader.filter(df_with_vader.vader_predicted_label == df_with_vader.label).count() /
    df_with_vader.count()
)
print(f"VADER Accuracy: {accuracy * 100:.2f}%")

df_eval = df_with_vader.select("label", "vader_predicted_label").toPandas()
df_eval["vader_predicted_label"] = df_eval["vader_predicted_label"].round().astype(int)
df_eval = df_eval[df_eval["label"].apply(lambda x: str(x).isdigit())]
df_eval = df_eval[df_eval["vader_predicted_label"].apply(lambda x: str(x).replace(".0", "").isdigit())]
df_eval["label"] = df_eval["label"].astype(int)

# Step 3: Calculate recall and F1-score
from sklearn.metrics import classification_report, recall_score, f1_score

print(classification_report(df_eval["label"], df_eval["vader_predicted_label"], digits=4))

recall = recall_score(df_eval["label"], df_eval["vader_predicted_label"])
f1 = f1_score(df_eval["label"], df_eval["vader_predicted_label"])
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
```

```
VADER Accuracy: 87.64%
              precision    recall  f1-score   support

           0     0.9110    0.9329    0.9218      7973
           1     0.7468    0.6846    0.7144      2305

    accuracy                         0.8772     10278
   macro avg     0.8289    0.8087    0.8181     10278
weighted avg     0.8741    0.8772    0.8753     10278

Recall: 0.6846
F1 Score: 0.7144
```

We got for the 0 class an accuracy of 0.91, a recall of 0.93 and a f1 score of 0.92. These are excellent metrics, the model identifies with success the 0 class.
For the 1 class the model got an accuracy of 0.75, a recall of 0.68 and a f1 score of 0.71. The model miss a lot of True Positives and do a lot of False positives (accuracy<75%)

Macro avg is the mean of each metrics considering all class, weighted avg is the same thing but it take into account the size of each class. It gaves us a see of how good is the model in general considering all the metrics.

These scores are typical for a model with a majority class bias, it is good with the dominant class but not with the minority class.

Sentiment Analysis with a logistic Regression

How a logistic regression works

Logistic regression is one of the most known models in machine learning for binary classification. The goal of this model is to predict a probability that an observation is in a class.

Here we will have our two classes : positive(1) and negative(0). We need a dataset of tweets that are already classified to train our model.

Let x=(x1,x2,....,xn) be the characteristic vector of a tweet. The logistic regression calculates the probability that the tweet is either in class 1 (positive) or class 2 (negative). It calculates that with the sigmoid function define as following :

$$P(y=1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}+b) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$$

w= weight vector
b = bias
σ = sigmoid function that bring every value in [0,1]

If P(y=1|x) > 0.5 then the model predict positive
If P(y=1|x) < 0.5 then the model predict negative

The model learns and change parameters w and b considering the minimization of the prediction error via the log loss function :

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

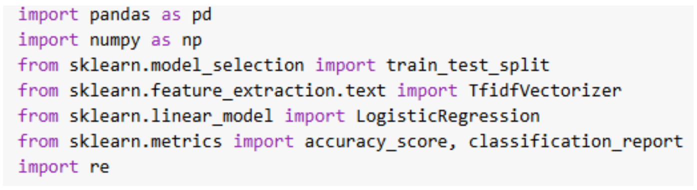
By changing w and b the model will be more and more optimal.

Now that we see how logistic regression works in theory, let's see how we implement it in python

Implementation of the logistic regression in python

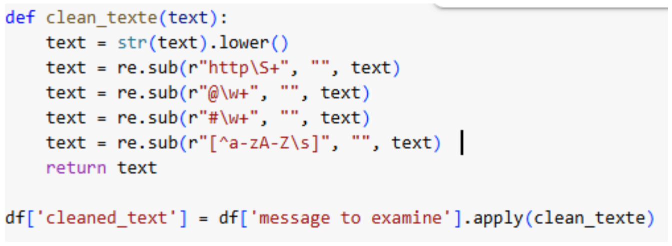
The code :
https://colab.research.google.com/drive/1sNrv9k84t4CaQP6GgYwYmHQbkbWnOVf9#scrollTo=mBubSPPjIkg9

The implementation is based on the same idea as VADER, we import all the libraries that we need.



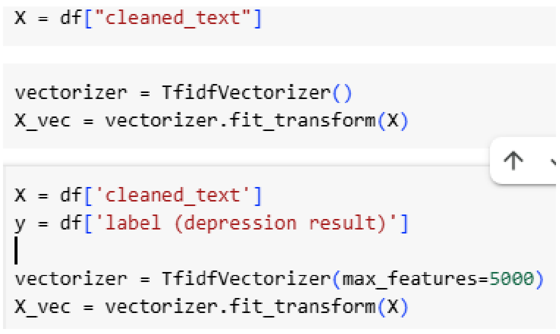
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
import re
```

Then we import our dataset (the same to make sure the comparison between both models is fair). We continue by clean the data (in an easier way here but it will be enough for us).



```
def clean_texte(text):
    text = str(text).lower()
    text = re.sub(r"http\S+", "", text)
    text = re.sub(r"@\w+", "", text)
    text = re.sub(r"#\w+", "", text)
    text = re.sub(r"[^a-zA-Z\s]", "", text)
    return text

df['cleaned_text'] = df['message to examine'].apply(clean_texte)
```

We separate both variables X (cleaned texts) and Y (sentiments) and we transform the text into numeric vectors with the TF-IDF method which allows to the model to understand the text.

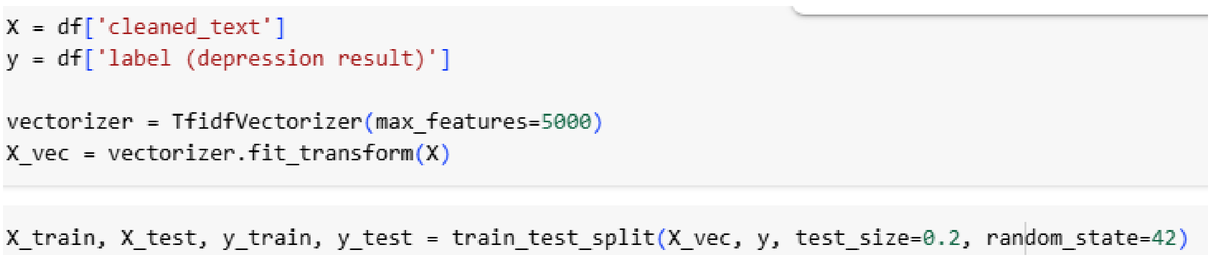


```
X = df["cleaned_text"]
```

```
vectorizer = TfidfVectorizer()
X_vec = vectorizer.fit_transform(X)
```

```
X = df['cleaned_text']
y = df['label (depression result)']

vectorizer = TfidfVectorizer(max_features=5000)
X_vec = vectorizer.fit_transform(X)
```

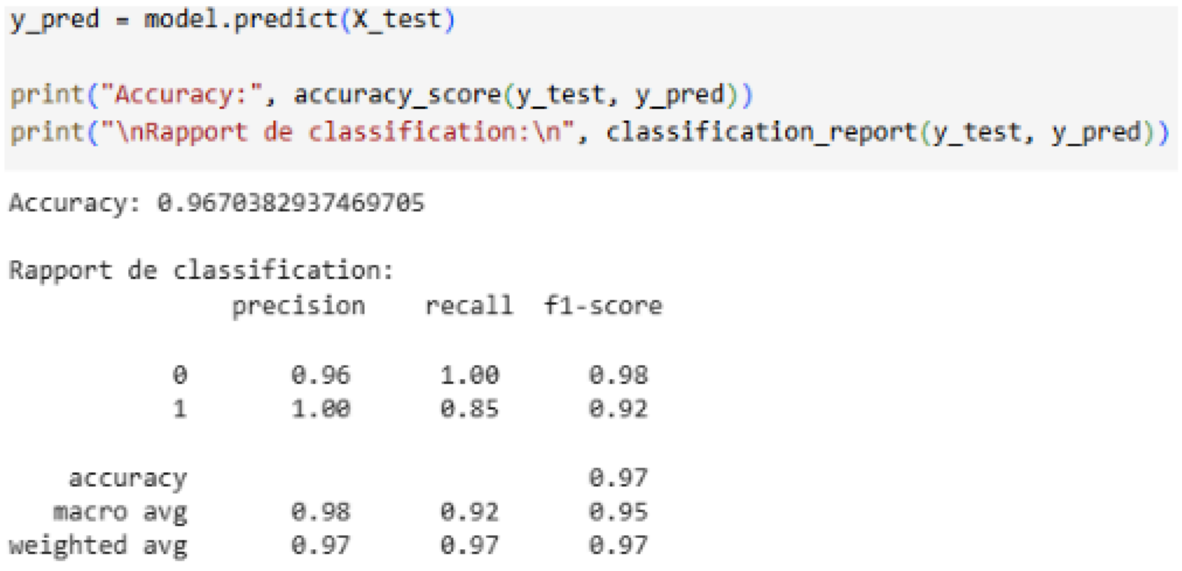
Then we divide the dataset into trainings sets and we start the logistic regression on it.



```
X = df['cleaned_text']
y = df['label (depression result)']

vectorizer = TfidfVectorizer(max_features=5000)
X_vec = vectorizer.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_vec, y, test_size=0.2, random_state=42)
```

Finally we evaluate the performance of the model by measuring accuracy, recall and f1 score.



```
y_pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nRapport de classification:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.9670382937469705

Rapport de classification:
              precision    recall  f1-score

           0       0.96      1.00      0.98
           1       1.00      0.85      0.92

    accuracy                           0.97
   macro avg       0.98      0.92      0.95
weighted avg       0.97      0.97      0.97
```

So we see that the performance is really great with top scores in all metrics for both classes.

Comparison of both models

Well when we compare the VADER and the logistic regression we see that without any doubt the logistic regression is far better. In all of the metrics it outperformed VADER and this for both classes. We can just look at the weighted average which is far better than on VADER (0.97 in all 3 metrics against 0.87 in all 3 metrics for VADER).

We can ask ourselves if logistic regression is always better than a NLP tool and why is that. Well firstly logistic regression is a machine learning model, it learns from labeled examples, it can adapt to the specific dataset and it finds patterns that are not in any predefined list. For example “Thank you so much for this day, it couldn't be better” can be sarcastic but VADER will not catch that, but logistic regression if trained can catch that.

Furthermore logistic regression is trainable on any domain (tweets, reviews,....), VADER is not customizable. Nevertheless we need to give some good points to VADER, it's much easier, it’s fast and light and it’s ready for real time use.

Nevertheless, for the real-time part, I used VADER. The reason is that VADER was much easier to set up. It doesn’t need any training, while for logistic regression I had to train the model myself, which takes more time and needs more resources. Since I work on a regular laptop without a lot of power, it was hard to run a real-time simulation and train the model at the same time, especially with graphics and Spark running too. So I chose the simpler option for the real-time part.However, based on the results, logistic regression would likely have performed better if I had more time to fully develop it.

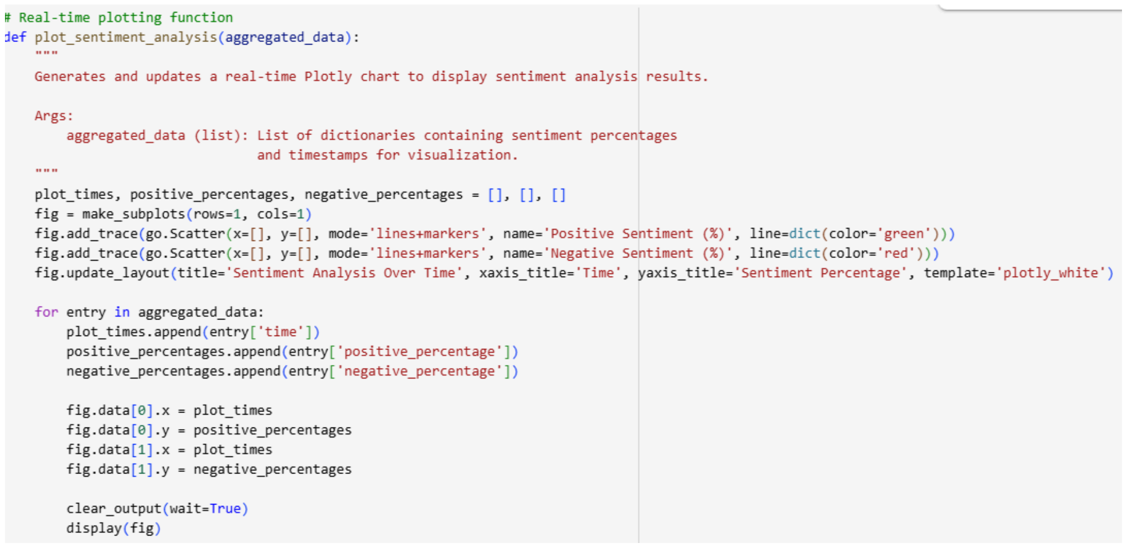
So now let’s continue with VADER, let’s see how to simulate the real time treatment of the tweets.

Real time treatment

Now we enter into the big part of the analysis, we must simulate the treatment of the tweets and their analysis by the model and then the visualisation on a graphic in real time. Let’s see through the code how we can do that.

Real Time graphic

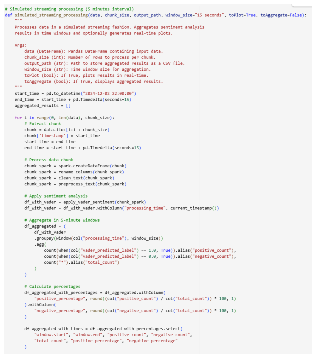
We want to create a real time graphic to visualize in real time the evolution of the sentiments (positive and negative) on the analyzed tweets. We will use Plotly to draw and update the graphic in real time when the data will come.



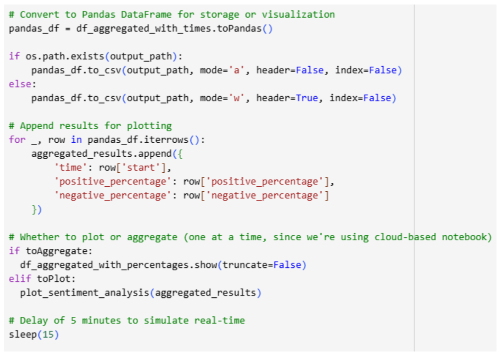
```python
# Real-time plotting function
def plot_sentiment_analysis(aggregated_data):
    """
    Generates and updates a real-time Plotly chart to display sentiment analysis results.

    Args:
        aggregated_data (list): List of dictionaries containing sentiment percentages
                                and timestamps for visualization.
    """
    plot_times, positive_percentages, negative_percentages = [], [], []
    fig = make_subplots(rows=1, cols=1)
    fig.add_trace(go.Scatter(x=[], y=[], mode='lines+markers', name='Positive Sentiment (%)', line=dict(color='green')))
    fig.add_trace(go.Scatter(x=[], y=[], mode='lines+markers', name='Negative Sentiment (%)', line=dict(color='red')))
    fig.update_layout(title='Sentiment Analysis Over Time', xaxis_title='Time', yaxis_title='Sentiment Percentage', template='plotly_white')

    for entry in aggregated_data:
        plot_times.append(entry['time'])
        positive_percentages.append(entry['positive_percentage'])
        negative_percentages.append(entry['negative_percentage'])

        fig.data[0].x = plot_times
        fig.data[0].y = positive_percentages
        fig.data[1].x = plot_times
        fig.data[1].y = negative_percentages

        clear_output(wait=True)
        display(fig)
```

Real Time Analysis

The main goal now is the heart of the work, the simulation of the real time treatment of the tweets. To do that we will divide the data into batches (chunks) with a determined number of tweets at each iteration. We will attribute to each chunk a simulate timestamp to create an artificial time effect. Then we apply the pipeline of cleaning, tokenization and analysis. We will group the results into a time window (15 seconds for example). Then calculate the percentage of positive and negative sentiments on each window. Then show the results on a graphic or store them in a csv.
There are the screenshots of the code but it's too big to be convenient to put in real size, so the code is available in the link in the introduction or in the presentation video at the end.



```python
# Simulated streaming processing (5 minutes interval)
def simulated_streaming_processing(data, chunk_size, output_path, window_size="15 seconds", toPlot=True, toAggregate=False):
    """
    Processes data in a simulated streaming fashion. Aggregates sentiment analysis
    results in time windows and optionally generates real-time plots.

    Args:
        data (DataFrame): Pandas DataFrame containing input data.
        chunk_size (int): Number of rows to process per chunk.
        output_path (str): Path to store aggregated results as a CSV file.
        window_size (str): Time window size for aggregation.
        toPlot (bool): If True, plots results in real-time.
        toAggregate (bool): If True, displays aggregated results.
    """
    start_time = pd.to_datetime("2024-12-02 22:00:00")
    end_time = start_time + pd.Timedelta(seconds=15)
    aggregated_results = []

    for i in range(0, len(data), chunk_size):
        # Extract chunk
        chunk = data.iloc[i:i + chunk_size]
        chunk['timestamp'] = start_time
        start_time = end_time
        end_time = start_time + pd.Timedelta(seconds=15)

        # Process data chunk
        chunk_spark = spark.createDataFrame(chunk)
        chunk_spark = rename_columns(chunk_spark)
        chunk_spark = clean_text(chunk_spark)
        chunk_spark = preprocess_text(chunk_spark)

        # Apply sentiment analysis
        df_with_vader = apply_vader_sentiment(chunk_spark)
        df_with_vader = df_with_vader.withColumn("processing_time", current_timestamp())

        # Aggregate in 5-minute windows
        df_aggregated = (
            df_with_vader
            .groupBy(window(col("processing_time"), window_size))
            .agg(
                count(when(col("vader_predicted_label") == 1.0, True)).alias("positive_count"),
                count(when(col("vader_predicted_label") == 0.0, True)).alias("negative_count"),
                count("*").alias("total_count")
            )
        )

        # Calculate percentages
        df_aggregated_with_percentages = df_aggregated.withColumn(
            "positive_percentage", round((col("positive_count") / col("total_count")) * 100, 1)
        ).withColumn(
            "negative_percentage", round((col("negative_count") / col("total_count")) * 100, 1)
        )

        df_aggregated_with_times = df_aggregated_with_percentages.select(
            "window.start", "window.end", "positive_count", "negative_count",
            "total_count", "positive_percentage", "negative_percentage"
        )
```



```python
# Convert to Pandas DataFrame for storage or visualization
pandas_df = df_aggregated_with_times.toPandas()

if os.path.exists(output_path):
    pandas_df.to_csv(output_path, mode='a', header=False, index=False)
else:
    pandas_df.to_csv(output_path, mode='w', header=True, index=False)

# Append results for plotting
for _, row in pandas_df.iterrows():
    aggregated_results.append({
        'time': row['start'],
        'positive_percentage': row['positive_percentage'],
        'negative_percentage': row['negative_percentage']
    })

# Whether to plot or aggregate (one at a time, since we're using cloud-based notebook)
if toAggregate:
  df_aggregated_with_percentages.show(truncate=False)
elif toPlot:
  plot_sentiment_analysis(aggregated_results)

# Delay of 5 minutes to simulate real-time
sleep(15)
```

Then we use this function to clean the former results store in the output file. It's allows us to avoid to conflict between the .csv file create by the simulation.

```python
def clean_output_folder(output_folder):
    """
    Cleans the output folder by removing all files.

    Args:
        output_folder (str): Path to the folder to be cleaned.
    """
    files = glob.glob(f"{output_folder}/*")
    for file in files:
        os.remove(file)
        print(f"Deleted {file}")
```

The code is almost finished, now we will just launch the analysis and print the evolution of it.

Firstly we launch the simulation and aggregate the results.

```python
clean_output_folder("output")
data = pd.read_csv(file_path)
output_path = "output/sentiment_results.csv"
# Aggregate the streaming data results
simulated_streaming_processing(data, chunk_size=100, output_path=output_path, toPlot=False, toAggregate=True)
```

Now we can launch the analysis with the graphic simulation too :

```python
clean_output_folder("output")
data = pd.read_csv(file_path)
output_path = "output/sentiment_results.csv"
# Plot the streaming data results
simulated_streaming_processing(data, chunk_size=100, output_path=output_path, toPlot=True, toAggregate=False)
```

Then we can stop the Spark session.

Here is the full video of the simulation : https://youtu.be/kKxzQV-b1ik

The video is accelerated because otherwise it will be too long.

Conclusion

In this project, I worked on sentiment analysis using Twitter data. The goal was to understand how different models can detect emotions in tweets, and to build a working system that can analyse them, even in real time. To do this, I used two different models: VADER, which is based on rules and a sentiment dictionary, and logistic regression, which is a machine learning model.

After testing both, it was clear that logistic regression gave better results, especially when looking at the metrics like accuracy, recall and F1-score. It was better at finding the right sentiment in tweets, even for the less common cases. However, I used VADER for the real-time part because it is faster and easier to set up. Logistic regression takes more time and is more complex to run in a real-time situation.

To process the data, I used **Apache Spark**, which is very useful when working with big amounts of data. Spark helped me clean, transform and analyse tweets in a fast and efficient way. Even though I couldn't access Twitter's real-time API because of new restrictions, I built a **simulated real-time pipeline**. This means I used a dataset from Kaggle and processed it in small parts, like tweets were coming in live. I cleaned the text, analysed it, and then showed the results in real time on a graph using Plotly.

In the end, the project showed how we can combine theory and practice: from data mining, to text preprocessing, to model comparison and finally to a live demonstration. There is still more that could be done, like improving the real-time part with a better model or even studying how people are grouped on Twitter depending on their mood. But for now, this gives a good base to understand how sentiment analysis works and how it can be used in real situations.

We can extend the question by asking ourselves if people that are negative and people that are positive are grouped together (after a football match for example). Then we can extend the question of cluster in Twitter and see how it works

References

- Hutto, C. J., & Gilbert, E. (2014). *VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text*. In Proceedings of the International AAAI Conference on Web and Social Media (ICWSM), 8(1), 216-225.
  https://ojs.aaai.org/index.php/ICWSM/article/view/14550

- Hosmer, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). *Applied Logistic Regression* (3rd ed.). Wiley.
  ISBN: 978-0470582473

- Powers, D. M. W. (2011). *Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation*. Journal of Machine Learning Technologies, 2(1), 37–63.
  http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research, 12, 2825–2830.
  https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

- rock3125 (2022) https://github.com/rock3125/vaderSentiment

- sharmaroshan (2019) https://github.com/sharmaroshan/Twitter-Sentiment-Analysis
- Hackers realm(2022) https://www.youtube.com/watch?v=26ZSHmUoBeM
- Apache Spark documentation https://spark.apache.org/docs/latest/quick-start.html

In this project, I also utilized various online resources, including ChatGPT, GitHub repositories for practical implementations, and discussion forums to troubleshoot technical issues.