Sentiment Analysis Pipeline – Documented Code

Step 1: Import Libraries

These libraries are essential for data loading, preprocessing, and visualization.

```
import pandas as pd
 import numpy as np
 import matplotlib.pyplot as plt
 import seaborn as sns
 import re
 import string
```

Step 2: Load Data

The training and test datasets are loaded from CSV files.

```
train = pd.read_csv("train.csv")
 test = pd.read_csv("test.csv")
```

Step 3: Combine and Label

We add a 'type' column to distinguish train and test data, then combine them.

```
train["type"] = "train"
 test["type"] = "test"
 test["label"] = np.nan
 combined_data = pd.concat([train, test], axis=0).reset_index(drop=True)
```

Step 4: Clean Tweets

Removes mentions and special characters, keeping only alphabets and hashtags.

```
def remove_pattern(input_txt, pattern):
    matches = re.findall(pattern, input_txt)
    for match in matches:
        input_txt = re.sub(match, "", input_txt)
    return input_txt

combined_data["clean_tweet"] = combined_data["tweet"].apply(lambda x:
remove_pattern(x, "@[\w]*"))
 combined_data["clean_tweet"] =
combined_data["clean_tweet"].apply(lambda x: re.sub("[^a-zA-Z#]", " ", x))
```

Step 5: Tokenization
Splits each cleaned tweet into a list of words.

```
tokenized_tweet = combined_data["clean_tweet"].apply(lambda x: x.split())
```

Step 6: Stemming
Reduces words to their root form using Porter Stemmer.

```
from nltk.stem.porter import PorterStemmer
 stemmer = PorterStemmer()
 tokenized_tweet = tokenized_tweet.apply(lambda x: [stemmer.stem(i) for i in
x])
```

Step 7: Join Tokens
Rejoins the stemmed tokens into a string to finalize cleaned tweets.

```
for i in range(len(tokenized_tweet)):
    tokenized_tweet[i] = " ".join(tokenized_tweet[i])
 combined_data["tidy_tweet"] = tokenized_tweet
```

Tweet Sentiment Analysis – Visualization & Hashtag Insights

A. WordCloud: Most Common Words in Entire Dataset
This helps us identify the most frequent words across all tweets regardless of their sentiment.

```
from wordcloud import WordCloud
 import matplotlib.pyplot as plt

all_words = ' '.join([text for text in combi['tidy_tweet']])
 wordcloud = WordCloud(width=800, height=500, random_state=21,
max_font_size=110).generate(all_words)

plt.figure(figsize=(10, 7))
 plt.imshow(wordcloud, interpolation="bilinear")
 plt.axis('off')
 plt.title("Most Common Words in All Tweets")
 plt.show()
```

B. WordCloud: Words in Non-Racist/Sexist Tweets
This wordcloud shows commonly used terms in tweets labeled as non-offensive (label=0). These are often positive or neutral.

```
normal_words = ' '.join([text for text in combi['tidy_tweet'][combi['label'] ==
0]])
 wordcloud = WordCloud(width=800, height=500, random_state=21,
max_font_size=110).generate(normal_words)

plt.figure(figsize=(10, 7))
 plt.imshow(wordcloud, interpolation="bilinear")
 plt.axis('off')
 plt.title("Words in Non-Racist/Sexist Tweets")
 plt.show()
```

C. WordCloud: Words in Racist/Sexist Tweets
This plot reveals the frequently used offensive or sensitive words in tweets labeled as hate speech (label=1).

```
negative_words = ' '.join([text for text in combi['tidy_tweet'][combi['label'] ==
1]])
 wordcloud = WordCloud(width=800, height=500, random_state=21,
max_font_size=110).generate(negative_words)

plt.figure(figsize=(10, 7))
 plt.imshow(wordcloud, interpolation="bilinear")
 plt.axis('off')
 plt.title("Words in Racist/Sexist Tweets")
 plt.show()
```

D. Hashtag Extraction Function
We extract hashtags from each tweet using regex. These help analyze trending tags used in each sentiment class.

```
import re

def hashtag_extract(x):
    hashtags = []
    for i in x:
        ht = re.findall(r"#(\w+)", i)
        hashtags.append(ht)
    return hashtags

HT_regular = hashtag_extract(combi['tidy_tweet'][combi['label'] == 0])
 HT_negative = hashtag_extract(combi['tidy_tweet'][combi['label'] == 1])

HT_regular = sum(HT_regular, [])
 HT_negative = sum(HT_negative, [])
```

E. Hashtag Frequency Plot (Top 20)
We count the top 20 most frequent hashtags in both non-hateful and hateful tweets and visualize them using seaborn barplots.

```
import pandas as pd
 import seaborn as sns
 import nltk

a = nltk.FreqDist(HT_regular)
 d = pd.DataFrame({'Hashtag': list(a.keys()), 'Count': list(a.values())})
 d = d.nlargest(columns="Count", n=20)
 plt.figure(figsize=(16,5))
 sns.barplot(data=d, x="Hashtag", y="Count").set(ylabel='Count')
 plt.title("Top 20 Hashtags in Non-Racist/Sexist Tweets")
 plt.show()

b = nltk.FreqDist(HT_negative)
 e = pd.DataFrame({'Hashtag': list(b.keys()), 'Count': list(b.values())})
 e = e.nlargest(columns="Count", n=20)
 plt.figure(figsize=(16,5))
 sns.barplot(data=e, x="Hashtag", y="Count").set(ylabel='Count')
 plt.title("Top 20 Hashtags in Racist/Sexist Tweets")
 plt.show()
```

Feature Engineering in Sentiment Analysis

To convert raw tweets (text) into numerical features so machine learning models can understand and learn from them.
Feature engineering helps models:

- Understand word importance
- Capture context and meaning
- Improve sentiment prediction accuracy

1. Bag-of-Words (BoW)
The Bag-of-Words model creates a vocabulary from all the unique words in the corpus and uses their frequency of occurrence as features. It ignores grammar and word order.

Example:
D1: He is a lazy boy. She is also lazy.
D2: Smith is a lazy person.
Vocabulary: ['He', 'She', 'lazy', 'boy', 'Smith', 'person']
Resulting matrix:
D1 -> [1, 1, 2, 1, 0, 0]
D2 -> [0, 0, 1, 0, 1, 1]

Code Snippet:

```
from sklearn.feature_extraction.text import CountVectorizer

bow_vectorizer = CountVectorizer(max_df=0.90, min_df=2,
max_features=1000, stop_words='english')
 bow = bow_vectorizer.fit_transform(combil['tidy_tweet'])
 bow.shape  # (49159, 1000)
```

2. TF-IDF (Term Frequency-Inverse Document Frequency)
TF-IDF assigns weights to words based on their frequency in a document and rarity across all documents.

TF = (Number of times term t appears in a document)/(Total terms in the document)
IDF = log(N/n), where N = total documents, n = documents containing term t
TF-IDF = TF * IDF

Code Snippet:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(max_df=0.90, min_df=2,
max_features=1000, stop_words='english')
 tfidf = tfidf_vectorizer.fit_transform(combil['tidy_tweet'])
 tfidf.shape  # (49159, 1000)
```

3. Word2Vec Embedding
Word2Vec converts words into dense vectors that represent their meanings.
Two models: CBOW (predicts word from context) and Skip-gram (predicts context from word).

Advantages:
- Captures semantic relationships
- Reduces dimensionality
Example: vec('King') - vec('Man') + vec('Woman') ≈ vec('Queen')

Code Snippet:

```
from gensim.models import Word2Vec

tokenized_tweet = combil['tidy_tweet'].apply(lambda x: x.split())
 model_w2v = Word2Vec(
     tokenized_tweet,
     size=200,
     window=5,
     min_count=2,
     sg=1,
     negative=10,
     workers=2,
     seed=34
 )
```

4. Doc2Vec Embedding
Doc2Vec is an extension of Word2Vec that provides embeddings for entire documents/tweets.
It learns an additional vector for each document.

Advantages:
- Better at capturing the overall context of a tweet or paragraph.

Code Snippet:

```
from gensim.models import Doc2Vec
 from gensim.models.doc2vec import TaggedDocument

labeled_tweets = [TaggedDocument(words=tweet.split(),
tags=[f'tweet_{i}']) for i, tweet in enumerate(combil['tidy_tweet'])]
 model_d2v = Doc2Vec(
    dm=1,
    vector_size=200,
    window=5,
    min_count=5,
    workers=3,
    epochs=15,
    seed=23
 )
 model_d2v.build_vocab(labeled_tweets)
 model_d2v.train(labeled_tweets, total_examples=len(labeled_tweets),
epochs=model_d2v.epochs)
 docvec_arrays =
np.array([model_d2v.infer_vector(tweet.words).reshape(1, 200) for
tweet in labeled_tweets])
```

Modelling:

Objective:

To build a robust sentiment analysis model using text data from tweets. We convert textual data into numerical representations (features), train various machine learning models, evaluate their performance using the F1-score, and fine-tune the best model (XGBoost).

Models Used & Why

1. Logistic Regression
    - What it is: A simple linear classifier that models the probability of a binary outcome.
    - Why used: It's fast, interpretable, and serves as a strong baseline for classification tasks.
2. Support Vector Machine (SVM)
    - What it is: A powerful classifier that tries to find the optimal boundary (hyperplane) between classes.
    - Why used: Works well with high-dimensional data (like text features from BoW/TF-IDF).
3. Random Forest
    - What it is: An ensemble of decision trees trained on different data subsets with random features.
    - Why used: Handles overfitting better than individual trees, and works well with noisy datasets.
4. XGBoost
    - What it is: Gradient Boosting algorithm that builds trees sequentially, focusing on correcting errors from previous trees.
    - Why used: Excellent performance in Kaggle/NLP competitions. Very powerful for imbalanced datasets and capable of learning complex patterns.

Feature Representations

For each model, tested these feature types:

- Bag-of-Words (BoW): Basic frequency of words
- TF-IDF: Adjusts word frequency by how common a word is across documents
- Word2Vec: Vector embeddings capturing semantic meaning
- Doc2Vec: Embeds entire documents instead of words

Findings:

📌 Word2Vec + XGBoost consistently gave the best F1-score.

XGBoost Hyperparameter Tuning

Why tune parameters?

XGBoost has many hyperparameters. Tuning helps find the best combo that gives the highest F1 score. We did this step-by-step:

🔧 Step-by-Step Tuning:

1. max_depth and min_child_weight:
    a. Control model complexity
    b. Higher max_depth = more splits
    c. Higher min_child_weight = more conservative splits
1. subsample and colsample_bytree:
    - Introduce randomness to prevent overfitting
    - subsample: % of rows sampled
    - colsample_bytree: % of columns sampled
1. eta (learning rate):
    - Smaller eta = slower learning but better convergence
    - Tuned to balance performance vs. training time

Each time, we used 5-fold cross-validation with early stopping to:

- Prevent overfitting
- Find the best number of rounds (trees)
- Maximize validation F1-score

Final Model: XGBoost with Word2Vec

- Params used:

```
{
 'objective': 'binary:logistic',
 'max_depth': 8,
 'min_child_weight': 6,
 'subsample': 0.9,
 'colsample_bytree': 0.5,
 'eta': 0.1
}
```

- Evaluation Metric: F1-Score
- Best Public Score: 0.703

What Is Fine-Tuning in XGBoost?

Fine-tuning is the process of optimizing hyperparameters to improve model performance—specifically, F1 score in your sentiment analysis task.

Instead of using default settings, we iteratively try different values for important parameters and use cross-validation to select the best combination.

Step 1: Tune max_depth & min_child_weight
Method: Use xgb.cv() to check F1 score for each combo and choose the best.

Step 2: Tune subsample & colsample_bytree
Result: Prevents trees from memorizing data and reduces variance.

Step 3: Tune eta (learning rate)
Final selection: Value that gives highest cross-validated F1 score with early stopping

Evaluation Technique: xgb.cv

Each tuning stage used:

- 5-fold cross-validation
- Early stopping to save time
- Custom F1-score evaluation function

This helps you:

- Choose best params
- Avoid overfitting
- Save compute resources

Fine-tuned XGBoost by systematically adjusting max_depth, min_child_weight, subsample, colsample_bytree, and eta using 5-fold cross-validation with early stopping to maximize F1-score—ultimately achieving our best model with max_depth=8, min_child_weight=6, subsample=0.9, colsample_bytree=0.5, and eta=0.1, boosting performance on Word2Vec features and delivering highest leaderboard score.

Summary:

- Tested 4 classifiers across 4 feature sets.
- Word2Vec features + XGBoost yielded the best performance.
- Hyperparameter tuning using cross-validation improved the score significantly.
- Early stopping helped optimize training duration and prevent overfitting.