IBM Naan Mudhalvan

Project Report

Sentiment Analysis for Marketing

Team Leader:Devnath R
Team Member 01: Arun N
Team Member 02: Dharshan S
Team Member 03: Ahamed Alufar B A

Problem Definition:

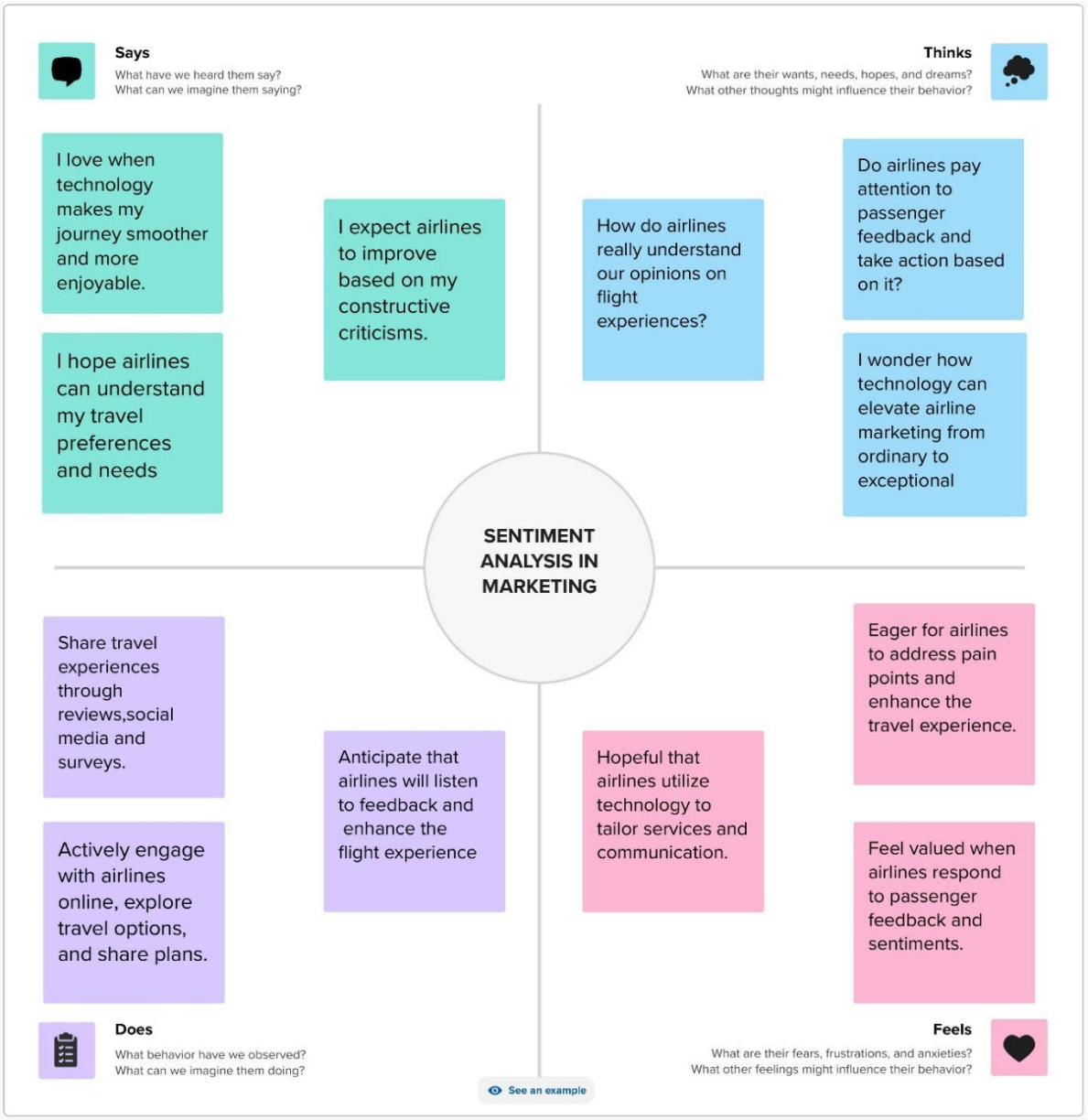
The central objective of this project is to conduct sentiment analysis on customer feedback to gain valuable insights into competitor products. Understanding customer sentiments allows companies to identify strengths and weaknesses in rival offerings, thus enhancing their own products. This project necessitates the utilization of various Natural Language Processing (NLP) methods for extracting insights from customer feedback.

Primary Goals:

1. **Understanding Customer Sentiment:**
    - **Objective**: Gain a comprehensive understanding of customer sentiment toward a product, service, or brand.
    - **Significance**: Sentiment analysis helps identify whether customers express positive, negative, or neutral feelings and opinions. This understanding is vital for evaluating customer satisfaction, loyalty, and areas in need of improvement.
2. **Informing Data-Driven Marketing Strategies:**
    - **Objective**: Utilize sentiment analysis insights to inform data-driven marketing strategies.
    - **Significance**: Marketers can tailor messaging, content, and advertising campaigns based on sentiment analysis. This approach ensures better resonance with the target audience. Additionally, sentiment analysis aids in identifying trends, market shifts, and emerging issues that influence marketing decisions and product development.

Ideation:

Empathy Map:



Brainstorming Ideas:

Team Leader	Team Member 01	Team Member 02	Team Member 03
Create a basic webpage with a user feedback form where visitors can submit comments about their airline experiences.	Store the user feedback and sentiment analysis results in a database for future reference.	Based on the sentiment, display a personalized response message on the webpage to acknowledge the user's feedback.	Include a basic trend graph that shows the historical sentiment trends over time,
After a user submits feedback, display the sentiment score and result on the webpage, such as "Positive," "Negative," or "Neutral."	Provide a brief explanation of how sentiment analysis works and how the sentiment result is determined (e.g., based on keywords).	Create a basic visualization, like a pie chart, that shows the distribution of feedback sentiments (e.g., the percentage of positive, negative, and neutral comments).	Add a section that shows basic statistics about user submissions, such as the total number of feedback entries and the distribution of sentiment (e.g., 60% positive, 30% neutral, 10% negative).

Prioritization of Ideas:

Design Thinking:

To ensure the effective realization of these objectives, we have devised a structured approach:

Step 1: Data Collection

- The main objective is to gather customer reviews and sentiments about the airline's services.
- The initial step involves collecting a comprehensive dataset comprising customer feedback. This dataset will serve as the foundational source of information for our sentiment analysis.
- Data sources include online customer reviews, social media platforms, the airline's website, and pre-existing datasets available from various repositories.

Tech Stack:

- *Data Source*: Online customer reviews, social media, airline's website etc.
- *Datasets*: Datasets from sources like Kaggle or academic repositories.

Step 2: Data Preprocessing

Before performing sentiment analysis, we need to clean and preprocess the textual data to ensure its quality and uniformity. This involves:

- Removing special characters, HTML tags, and irrelevant information.
- Tokenization: Splitting text into words or subword units.
- Lowercasing all text to maintain consistency.
- Removing stop words (common words like "the," "and," "is") that do not carry significant meaning.
- Handling misspellings and abbreviations.
- Lemmatization or stemming to reduce words to their root form.

Tech Stack:

- *Python Libraries*: Utilize NLTK (Natural Language Toolkit), spaCy, or similar libraries for efficient preprocessing.

Step 3: Sentiment Analysis Techniques

- This step focuses on the application of sentiment analysis techniques such as NLP to understand the emotional tone of the customer feedback.
- Techniques may include Bag of Words (BoW), Word Embeddings (Word2Vec, GloVe), or advanced deep learning models like Transformer models (e.g., BERT, GPT-3).
- The choice of technique will depend on the complexity of the problem and the availability of computational resources.

Tech Stack:

- *Bag of Words (BoW)*: Implement a count-based technique to convert text into numerical data.
- *Word Embeddings*: Transform words into vector representations for richer context.
- *Transformer Models*: Utilize advanced deep learning models for sophisticated sentiment analysis.
- *Python Libraries*: Rely on libraries such as scikit-learn, TensorFlow, PyTorch, and Hugging Face Transformers for pre-trained models.

Step 4: Feature Extraction

Once we have applied sentiment analysis techniques, we will extract features from the text data. These features may include:

- Sentiment scores (positive, negative, neutral) for each review.
- Keywords or phrases associated with positive and negative sentiments.
- Summary statistics of sentiment distributions.

Tech Stack:

- *Feature Engineering*: Create sentiment-related features to capture insights.
- *Python Libraries*: Leverage scikit-learn, pandas, and numpy for efficient feature extraction.

Step 5: Visualization

- Visualizations play a pivotal role in understanding sentiment patterns within the dataset.
- Creation of visualizations helps to depict the sentiment distribution and analyze trends.
- Common visualizations include sentiment distribution plots, word clouds, and trend analysis to visualize how sentiments change over time.

Tech Stack:

- *Visualization Tools*: Utilize libraries such as Matplotlib, Seaborn, Plotly, or similar tools for creating informative visualizations.
- *Types of Visualizations*: Craft sentiment distribution plots, word clouds, and trend analysis graphs to facilitate data interpretation

Step 6: Insights Generation

- In this step, we delve into the sentiment analysis results to derive valuable insights that can inform the airline's strategic decisions.
- Insights may encompass identifying customer preferences, pinpointing areas of strength and improvement, and identifying emerging trends.

Tech Stack:

- *Statistical Analysis*: Conduct statistical analyses to identify significant patterns and trends in sentiment data.
- *Textual Analysis*: Apply text analysis techniques to extract valuable insights from customer feedback.
- *Python Libraries*: Leverage pandas, numpy, and Natural Language Toolkit (NLTK) for comprehensive insights generation.

Fine-tuning Pre-trained Sentiment Analysis Models:

1. **Select a Pre-trained Model:**

2. Data Preparation:

Data Collection: Gather a substantial amount of labeled sentiment data for finetuning. The more diverse and representative the data, the better the model's performance.

Data Preprocessing: Clean and preprocess the data, including text normalization, removing special characters, and handling imbalanced class distributions if applicable.

3. Tokenization and Input Formatting:

4. Architecture Modification:

5. Hyperparameter Tuning:

6. Training and Validation:

7. Regularization Techniques:

8. Evaluate and Fine-tune:

Fine-tuning is a process in machine learning where a pre-trained model is further trained on a specific task to adapt it to perform that task more effectively.

In the context of sentiment analysis using pre-trained models like BERT, RoBERTa, or other transformer-based architectures, fine-tuning involves taking a pre-trained language model and training it on a dataset specifically designed for sentiment analysis. Here's a detailed explanation of the fine-tuning process:

Fine-Tuning Process for Sentiment Analysis:

1. Selecting a Pre-trained Model:

Choose a pre-trained model suitable for your sentiment analysis task, considering factors like architecture, size, and language support.

2. Data Preparation:

Prepare your sentiment analysis dataset as explained in the previous response. Clean, tokenize, and split your data into training, validation, and test sets.

3. Model Architecture Modification:

Modify the pre-trained model by adding a classification head on top of it. The existing pre-trained layers act as feature extractors, and you add a new set of layers (usually a dense layer) for sentiment classification.

For binary sentiment analysis, use a single output neuron with a sigmoid activation function. For multi-class sentiment analysis, use softmax activation with multiple output neurons corresponding to different classes.

4. ***Loss Function:***

Choose an appropriate loss function based on your task. For binary classification, use binary cross-entropy loss; for multi-class classification, use categorical cross- entropy loss. This function quantifies the difference between predicted and actual sentiment labels.

5. ***Optimizer and Learning Rate:***

Select an optimizer like Adam or SGD (Stochastic Gradient Descent) and set an initial learning rate. The optimizer adjusts the model's weights during training to minimize the chosen loss function.

6. Training:

Train the modified model on your sentiment analysis dataset. Use the training data to update the model's weights iteratively. Monitor the validation performance to prevent overfitting. Implement techniques like early stopping, where training is halted if the validation performance stops improving, to avoid overfitting.

7. Hyperparameter Tuning:

Experiment with different hyperparameters such as learning rates, batch sizes, and the number of epochs. You can perform grid search or random search to find the optimal combination of hyperparameters.

8. Regularization Techniques:

Use regularization techniques like dropout within the classification head to prevent overfitting. Dropout randomly deactivates a fraction of neurons during training, reducing the risk of the model relying too heavily on specific features.

9. ***Evaluation:***

After training, evaluate the fine-tuned model on the test dataset using appropriate evaluation metrics (accuracy, precision, recall, F1-score, etc.). This step helps assess how well your model generalizes to new, unseen data.

10. ***Iterative Optimization:***

Based on the evaluation results, iterate on the model, fine-tuning process, and hyperparameters if necessary. Fine-tuning can be an iterative process where you experiment with different architectures and hyperparameters to achieve the best results.

9. Inference and Deployment:

After achieving a desirable performance, the model is ready for inference. Integrate the model into your application, ensuring it can handle real-time predictions efficiently. Monitor the model's performance in the production environment and re-train periodically with fresh data if necessary.

10. Ensemble Methods (Optional):

Ensemble methods are machine learning techniques that combine the predictions of multiple models to improve overall predictive performance. The idea behind ensembles is to harness the collective intelligence of diverse models, each with its strengths and weaknesses, to make more accurate and robust predictions.

Deep Learning Architectures:

Deep learning is a subset of machine learning that focuses on neural networks with multiple layers, enabling the model to automatically learn hierarchical features from the data. Here are some key deep learning architectures:

Feedforward Neural Networks (FNNs):

FNNs, or multilayer perceptrons, consist of an input layer, one or more hidden layers, and an output layer. Neurons are connected in a feedforward manner. These networks are versatile and can be used for various tasks, including classification and regression.

Convolutional Neural Networks (CNNs):

CNNs are designed for processing grid-like data, such as images and video. They use convolutional layers to automatically learn features from local patches of data.

CNNs have achieved remarkable success in image classification, object detection, and other computer vision tasks.

Recurrent Neural Networks (RNNs):

RNNs are suited for sequential data, such as time series and natural language. They have loops that allow information to persist and be passed from one step to the next. RNNs are used in tasks like speech recognition, language modeling, and machine translation.

Long Short-Term Memory (LSTM) Networks:

LSTMs are a type of RNN designed to address the vanishing gradient problem. They can capture long-range dependencies in data. LSTMs are commonly used in tasks involving sequential data, such as text generation and sentiment analysis.

Autoencoders:

Autoencoders are neural networks used for unsupervised learning and feature extraction. They consist of an encoder that compresses data and a decoder that reconstructs it. Autoencoders are used in dimensionality reduction and feature learning.

Reinforcement Learning Networks:

These networks are used in reinforcement learning tasks where agents interact with an environment to maximize a reward. Deep Q-Networks (DQNs) and policy gradients are examples of deep reinforcement learning models.

Transformers:

Transformers are a type of deep learning architecture that has revolutionized natural language processing. They are highly effective in tasks such as machine translation, text summarization, and sentiment analysis. Transformers employ self-attention mechanisms to capture contextual information. Each of these architectures has its unique characteristics and is suitable for specific types of data and tasks. Deep learning architectures have been pivotal in achieving state-of-the-art results in a wide range of applications, from computer vision to natural language understanding..

BERT(Bidirectional Encoder Representations from Transformers)

In the realm of Natural Language Processing (NLP), BERT (Bidirectional Encoder Representations from Transformers) has redefined the landscape with its bidirectional training and versatile applications. This in-depth report delves into BERT's core concepts, its groundbreaking training strategies, and provides an extensive examination of the fine-tuning process that allows BERT to adapt to a myriad of NLP tasks. BERT's introduction represents a pivotal moment in NLP, enabling transfer learning and fine-tuning for diverse language tasks.

Core Concepts

1. Bidirectional Training Mastery

BERT's core innovation lies in its bidirectional training. Unlike earlier models that processed text unidirectionally, BERT comprehends language by scanning the entire sequence at once. This enables a comprehensive understanding of context and nuances.

2. Leveraging the Transformer Architecture

BERT is an offspring of the Transformer architecture, known for its powerful attention mechanisms. Notably, BERT uses only the encoder mechanism, which is vital in its non-directional approach to language understanding.

3. The Masked Language Model (MLM)

To facilitate bidirectional training, BERT introduces the Masked Language Model (MLM). It involves replacing 15% of words with a "[MASK]" token. BERT's objective is to predict the original values of these masked words,

utilizing context for enhanced understanding.

4. Next Sentence Prediction (NSP)

Complementing MLM, BERT incorporates Next Sentence Prediction (NSP). It trains the model on pairs of sentences, predicting whether the second sentence follows the first. NSP enriches the model's grasp of sentence-level context and relationships.

5. Tokenization Strategies

BERT relies on WordPiece tokenization, slicing words into subword units. This flexibility enables BERT to handle out-of-vocabulary words and capture intricate linguistic subtleties. Tokenization remains consistent during pretraining and fine-tuning phases.

6. Layers and Embeddings

BERT's architecture includes a stack of transformer layers, typically 12 or 24 layers. Each layer incorporates multi-head self-attention mechanisms and feedforward neural networks. BERT harnesses word embeddings and position embeddings, considering word context and position within a sequence.

The Fine-Tuning Process

The fine-tuning process is where BERT's adaptability shines. It allows BERT to be tailored for specific NLP tasks while retaining the knowledge gained during pretraining. The following steps illustrate how the fine-tuning process unfolds:

1. Task-Specific Layers

For each specific task, task-specific layers are added to the core BERT model. These layers are typically small compared to the pretraining model, as BERT's general-purpose understanding is leveraged.

2. Data Preparation

Training data for the task is prepared, including labeled examples, such as sentences with sentiment labels, question-and-answer pairs, or entity recognition annotations. This data serves as the foundation for fine-tuning.

3. Loss Function

A task-specific loss function is defined based on the nature of the task. For example, in sentiment analysis, a binary cross-entropy loss might be used, while question answering tasks would employ losses suited for answer span prediction.

4. Backpropagation

The fine-tuning process includes backpropagation, where gradients from the loss function are calculated and used to update the model's weights. This process is performed on the task-specific layers while preserving the pretraining knowledge in the core BERT model.

5. Hyperparameter Tuning

Fine-tuning may involve hyperparameter tuning, such as adjusting learning rates, batch sizes, and the number of training epochs to optimize task performance.

6. Evaluation

The model is evaluated on a separate validation dataset to monitor its performance. Fine-tuning continues iteratively until the model achieves satisfactory results.

Key Insights

- The fine-tuning process empowers BERT to adapt to a wide array of NLP tasks while retaining the benefits of pretraining.

• The choice of task-specific layers, data preparation, loss functions, and hyperparameters plays a critical role in the success of fine-tuning.

RoBERTa (Robustly Optimized BERT Approach)

RoBERTa is a testament to the ever-evolving landscape of NLP models. Building upon the foundations laid by BERT, RoBERTa introduces a range of novel elements and strategies. This report delves into RoBERTa's technical underpinnings, emphasizing its unique attributes and how its fine-tuning process leverages these distinctions for diverse NLP tasks.

Modifications to BERT:

RoBERTa has almost similar architecture as compare to BERT, but in order to improve the results on BERT architecture, the authors made some simple design changes in its architecture and training procedure. These changes are:

1. Elimination of the NSP Objective

RoBERTa diverges from BERT by discarding the Next Sentence Prediction (NSP) objective. In BERT, NSP trains the model to predict whether two document segments are from the same or distinct documents. RoBERTa's extensive experiments revealed that removing the NSP objective either maintains or slightly enhances performance on real-world language tasks. This change underscores the importance of the primary Masked Language Modeling (MLM) objective in language understanding

.**2. Training with Larger Batch Sizes & Longer Sequences**

RoBERTa rethinks its training strategy by embracing larger batch sizes and extended text sequences. While BERT uses smaller batches and shorter text segments, RoBERTa takes a more ambitious approach. It undergoes training with 125,000 steps using batch sizes of 2,000 sequences and further trains for 31,000 steps with batch sizes of 8,000 sequences. This shift offers several advantages, including improved language nuance comprehension and more efficient parallelization in distributed training.

3. Introduction of Dynamic Masking

In contrast to BERT's static masking approach, RoBERTa adopts dynamic masking. Instead of using a single static mask throughout training, RoBERTa introduces variability by duplicating the training data and applying different mask strategies during each pass. This dynamic masking strategy aims to prevent overfitting to specific masking patterns and fosters a deeper understanding of language context.

Core Concepts

1. Advancing Bidirectional Training

RoBERTa builds upon BERT's bidirectional training, taking it to new heights. The uniqueness lies in RoBERTa's extensive bidirectional training, which encompasses more data and training steps. This results in a deeper understanding of language context and nuance.

2. RoBERTa's Tokenization Expertise

In contrast to BERT's use of WordPiece tokenization, RoBERTa employs the Byte-Pair Encoding (BPE) algorithm for tokenization. This tokenization method excels in representing both words and subwords, allowing RoBERTa to capture linguistic intricacies with precision.

3. Fine-Tuning with No NSP

RoBERTa stands out by excluding the Next Sentence Prediction (NSP) task during pretraining. This approach streamlines the pretraining process, allowing more focus on the masked language modeling (MLM) objective. The omission of NSP emphasizes the importance of MLM and context modeling

.**4. Large-Scale Pretraining**

RoBERTa undergoes extensive pretraining, using more data and training steps compared to BERT. This unique approach results in a model with an even more profound understanding of language, positioning it as a versatile solution for diverse downstream tasks.

The Fine-Tuning Process in RoBERTa

RoBERTa's fine-tuning process builds upon BERT's approach with several unique aspects:

1. Task-Specific Adaptations

To adapt RoBERTa for a specific NLP task, task-specific adaptations are implemented. These modifications encompass the addition of task-specific layers and other adjustments that cater to the specific requirements of the task.

2. Data Preparation Expertise

RoBERTa's fine-tuning process involves meticulous data preparation. The comprehensive nature of pretraining allows RoBERTa to perform exceptionally well with less task-specific data, reducing the need for extensive task-specific datasets.

3. Customized Loss Functions

RoBERTa utilizes task-specific loss functions, uniquely designed for each task objective. This approach ensures that the model's fine-tuning aligns precisely with the requirements of the task, whether it involves classification, question answering, or entity recognition.

4. Training without NSP

RoBERTa's fine-tuning is unique in that it excludes the NSP task, focusing solely on the masked language modeling (MLM) objective. This streamlined approach simplifies fine-tuning while enhancing the model's adaptability to a wide range of tasks.

5. Hyperparameter Optimization

RoBERTa's fine-tuning process often includes specialized hyperparameter tuning. Parameters such as learning rates, batch sizes, and training epochs are meticulously optimized to ensure the model's optimal task-specific performance.

6. Extensive Evaluation

The fine-tuned model is subjected to a rigorous evaluation on a dedicated validation dataset. RoBERTa's extensive pretraining and unique fine-tuning approach enable it to achieve a high level of proficiency with fewer fine-tuning iterations.

DEVELOPMENT OF SENTIMENT ANALYSIS MODEL

Loading and preprocessing the dataset

For loading and preprocessing the dataset the necessary libraries for data preprocessing , model training and evaluation of text classification tasks.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import pickle
import warnings
warnings.filterwarnings(action='ignore')

# nltk
import nltk
nltk.download('stopwords')

## Preprocessing libraries
import re
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer

# For Model training
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import BernoulliNB
from sklearn.svm import LinearSVC            # a variant of SVC optimized for large datasets

# Metrics for accuracy
from sklearn.metrics import accuracy_score,confusion_matrix, classification_report
```

Reading of dataset:

The datas in the csv file can be read by

```
df = pd.read_csv('Tweets.csv')
df.head()
```

The code reads a CSV (Comma-Separated Values) file named 'Tweets.csv' into a Pandas DataFrame and then displays the first few rows of the DataFrame using the head() function.

pd.read_csv('Tweets.csv'): This line of code uses the read_csv() function from the Pandas library (pd is an alias commonly used for Pandas) to read data from a CSV file named 'Tweets.csv'. The CSV file is assumed to be in the same directory as the Python script or Jupyter Notebook where this code is executed. The data from the CSV file is loaded into a Pandas DataFrame called df.

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativereason_confidence	airline	airline_se
0	570306133677760513	neutral	1.0000	NaN	NaN	Virgin America	
1	570301130888122368	positive	0.3486	NaN	0.0000	Virgin America	
2	570301083672813571	neutral	0.6837	NaN	NaN	Virgin America	
3	570301031407624196	negative	1.0000	Bad Flight	0.7033	Virgin America	
4	570300817074462722	negative	1.0000	Can't Tell	1.0000	Virgin America	

Checking of null values:

The df.isnull().sum() expression is used to identify and count the missing (null or NaN) values in each column of the DataFrame df

```
df.isnull().sum()
```

```
tweet_id                      0
airline_sentiment             0
airline_sentiment_confidence  0
negativereason             5462
negativereason_confidence  4118
airline                       0
airline_sentiment_gold    14600
name                          0
negativereason_gold       14608
retweet_count                 0
text                          0
tweet_coord               13621
tweet_created                 0
tweet_location             4733
user_timezone              4820
dtype: int64
```

Checking the distirbution of airlines:

```
plt.figure(figsize=(7, 3))
sns.countplot(data=df, x='airline', hue='airline', palette=['#1f78b4', '#33a02c', '#e31a1c', '#ff7f00', '#6a3d9a', '#a6cee3'], legend=False)
plt.show()
```

plt.figure(figsize=(7, 3)): This line of code creates a new figure for the plot with a specific size. The figsize parameter specifies the width and height of the figure in inches. In this case, the width is 7 inches, and the height is 3 inches.

sns.countplot(): This function from the Seaborn library is used to create a count plot. It takes several parameters:

data=df: Specifies the DataFrame df from which the data for the plot will be taken.

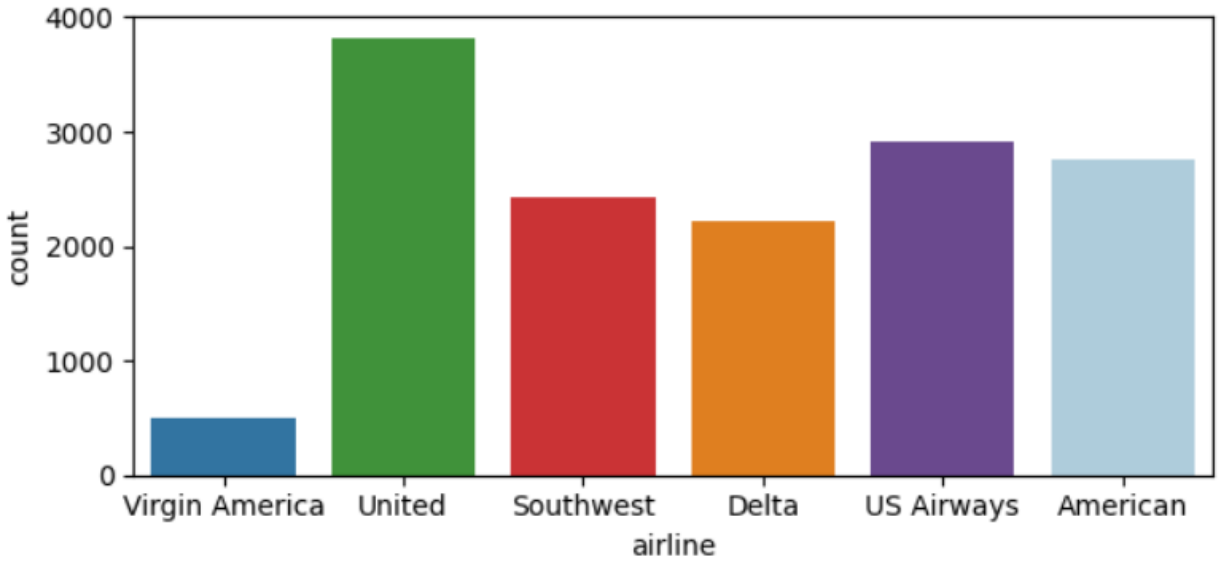
x='airline': Indicates the column in the DataFrame that will be plotted on the x-axis. In this case, it's the 'airline' column, representing different airlines.

hue='airline': This parameter is used to color the bars based on the 'airline' column, creating a separate bar for each unique value in the 'airline' column.

palette=['#1f78b4', '#33a02c', '#e31a1c', '#ff7f00', '#6a3d9a', '#a6cee3']: Specifies the colors to be used for each unique value in the 'airline' column. Each color in the list corresponds to a unique airline.

legend=False: This parameter is set to False to hide the legend in the plot. The legend is not needed in this case because the colors are already defined based on the 'airline' column.

plt.show(): This line of code displays the created plot. Without this line, the plot would not be visible.



Seeing the distribution of positive and negative tweet reviews in target column:

```
plt.figure(figsize=(7, 3))
sns.countplot(data=df, x='airline_sentiment', hue='airline_sentiment', palette=['yellow', 'green', 'red'], legend=False)
plt.show()
```

sns.countplot(): This function from the Seaborn library is used to create a count plot. It takes several parameters:

data=df: Specifies the DataFrame df from which the data for the plot will be taken.

x='airline_sentiment': Indicates the column in the DataFrame that will be plotted on the x-axis. In this case, it's the 'airline_sentiment' column, representing different sentiment labels (positive, negative, or neutral).

hue='airline_sentiment': This parameter is used to color the bars based on the 'airline_sentiment' column, creating a separate bar for each unique sentiment label.

palette=['yellow', 'green', 'red']: Specifies the colors to be used for each sentiment label. Yellow represents neutral sentiment, green represents positive sentiment, and red represents negative sentiment.

legend=False: This parameter is set to False to hide the legend in the plot. The legend is not needed in this case because the colors are already defined based on the 'airline_sentiment' column.

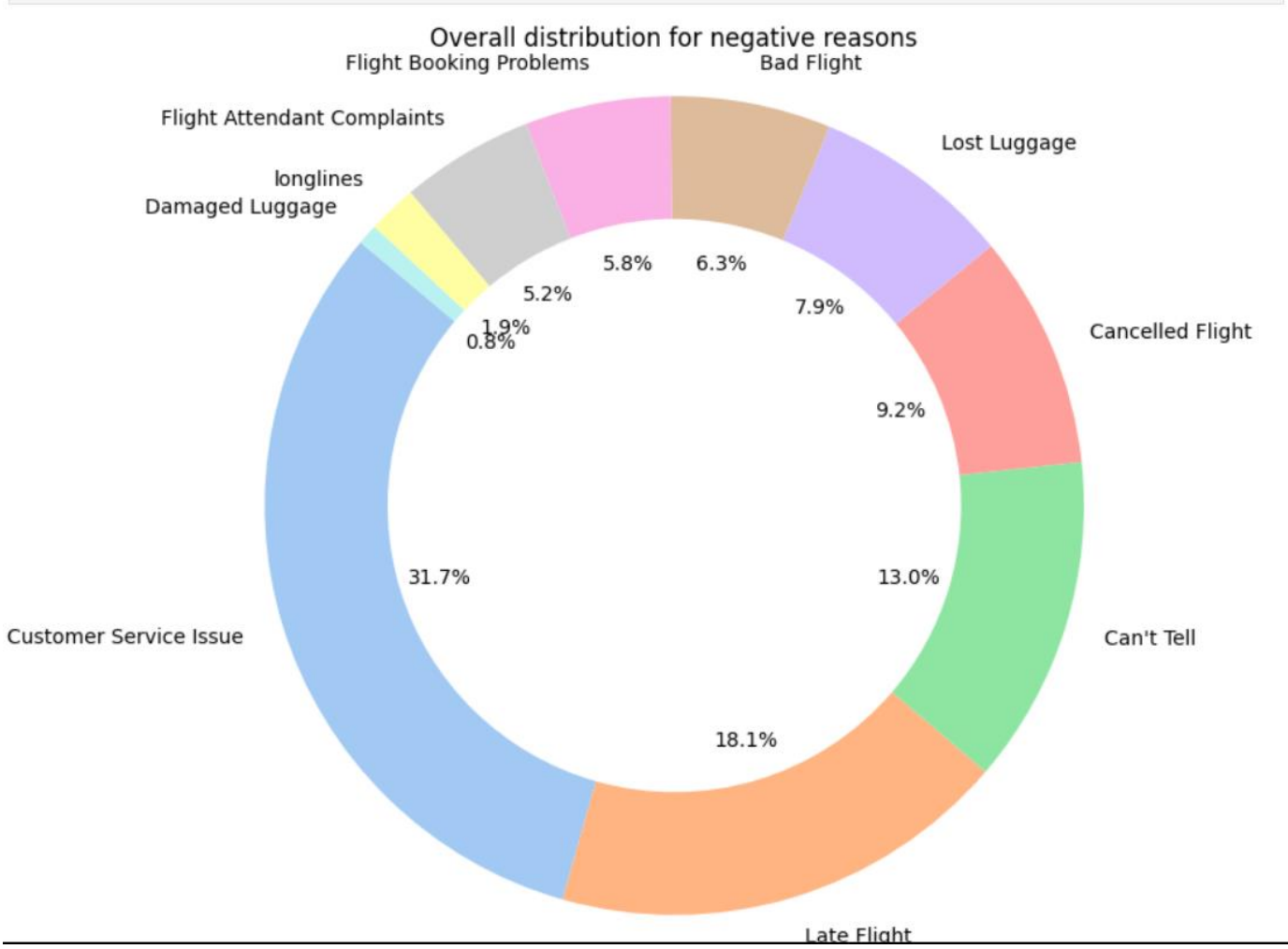
CALCULATION

The code is calculating the count of occurrences for each unique value in the 'negativereason' column of the DataFrame df. The result is stored in the value_counts variable.

```
# Calculate the value counts for each negative reason
value_counts = df['negativereason'].value_counts()

# Create a donut-like pie chart using matplotlib and seaborn
plt.figure(figsize=(8, 8))
labels = value_counts.index
values = value_counts.values
colors = sns.color_palette('pastel')[0:len(labels)]  # Use pastel colors for the chart
plt.pie(values, labels=labels, colors=colors, autopct='%1.1f%%', startangle=140, wedgeprops=dict(width=0.3))
plt.title('Overall distribution for negative reasons')
plt.axis('equal')  # Equal aspect ratio ensures the pie chart is drawn as a circle.
plt.show()
```

labels = value_counts.index: This line extracts the unique negative reasons (index of value_counts), which will be used as labels in the pie chart.
values = value_counts.values: This line extracts the corresponding counts for each negative reason, which will be represented as the sizes of the sectors in the pie chart.
plt.pie(): This function creates the pie chart. It takes several parameters:
values: The sizes of the sectors in the pie chart (counts of negative reasons).
labels: The labels for each sector (unique negative reasons).
colors: The colors for each sector.
autopct='%1.1f%%': A string formatting specification for the percentages displayed on the pie chart.
startangle=140: The angle at which the first sector starts. In this case, it starts at an angle of 140 degrees.
wedgeprops=dict(width=0.3): Specifies the width of the wedges in the pie chart, creating a donut-like appearance.

plt.title('Overall distribution for negative reasons'): This line sets the title of the pie chart to 'Overall distribution for negative reasons'.
plt.axis('equal'): This line ensures that the pie chart is drawn as a circle by setting the aspect ratio to be equal.



Data clearing and preprocessing:

```
corpus = []
ps = PorterStemmer()
for i in range(len(df)):
    # Removing special characters from text (message)
    review = re.sub('[^a-zA-Z]', ' ', df['text'][i])

    # Converting entire text into lowercase
    review = review.lower()

    # Splitting our text into words
    review = review.split()

    # Stemming and removing stopwords
    review = [ps.stem(word) for word in review if not word in set(stopwords.words('english'))]

    # Joining all the words into a complete text
    review = ' '.join(review)

    # Appending each text into the list corpus
    corpus.append(review)
```

Explanation

The code uses a for loop to iterate through each row in the 'text' column of the DataFrame df.

re.sub('[^a-zA-Z]', ' ', df['text'][i]): This line removes special characters and keeps only alphabetical characters (letters) in the text. It substitutes any non-alphabetic character with a space.

review = review.lower(): Converts the entire text to lowercase. This step is done to ensure uniformity, as 'Word' and 'word' should be treated the same way in text analysis.

review = review.split(): Splits the processed text into individual words. This step converts the text into a list of words.

[ps.stem(word) for word in review if not word in set(stopwords.words('english'))]: For each word in the list, stemming is performed using the Porter Stemmer (ps.stem(word)). Stemming reduces words to their root forms. Additionally, it checks if the word is not in the set of English stopwords (common words like 'and', 'the', 'is' that are usually removed as they do not carry significant meaning in text analysis). If the word is not a stopword, it is stemmed and added to the list.

```
cv = TfidfVectorizer(ngram_range=(1, 2), max_features=500000)
```

```
# We will use X as independent feature section
X = cv.fit_transform(corpus)
# We will use y as dependent feature section
y=df['airline_sentiment']
```

```
print('No. of feature_words: ', len(cv.get_feature_names_out()))
```

```
No. of feature_words:  91436
```

```
import pickle

# Creating a pickle file for the TfidfVectorizer
with open('cv-transform.pkl', 'wb') as f:
    pickle.dump(cv, f)
```

```
import pickle

# Load the contents of the pickle file
with open('cv-transform.pkl', 'rb') as f:
    loaded_cv = pickle.load(f)

# Now you can inspect the loaded object
print(loaded_cv)

# You can also access the attributes and methods of the loaded object, depending on what it is
print(loaded_cv.get_feature_names_out())
```

cv = TfidfVectorizer(ngram_range=(1, 2), max_features=500000)

This line initializes a TfidfVectorizer object with the specified parameters: ngram_range=(1, 2) indicates that both unigrams (single words) and bigrams (two adjacent words) will be considered. max_features=500000 sets the maximum number of features (words or word combinations) to be considered, limiting the vocabulary size to 500,000.

cv.fit_transform(corpus): This line uses the TfidfVectorizer to transform the corpus data (preprocessed text) into numerical vectors. It creates a sparse matrix X where each row represents a document (text) and each column represents a unique feature (word or word combination) based on the Tfidf values.

y = df['airline_sentiment']: This line assigns the 'airline_sentiment' column from the DataFrame df to the variable y. y represents the target variable or dependent feature for the classification task.

This code saves the TfidfVectorizer object cv into a binary pickle file named 'cv-transform.pkl'. The 'wb' mode is used to write the file in binary mode.
The section of the code loads the contents of the pickle file 'cv-transform.pkl' back into a new TfidfVectorizer object named loaded_cv. The 'rb' mode is used to read the file in binary mode.

Model Training

```
model2=BernoulliNB()
model3=LinearSVC()
model=[model1, model2, model3]
i = 0
for algo in model:
  i += 1
  print("M-O-D-E-L :",i)
  algo.fit(X_train, y_train)
  y_pred=algo.predict(X_test)
  # Checking the accuracy
  print("Confusion matrix : \n",confusion_matrix(y_pred,y_test))
  print("Accuracy score : ",accuracy_score(y_pred,y_test))
  print("Classification Report : \n",classification_report(y_pred,y_test))
  print("------------------------------------------------------------\n")

from sklearn.ensemble import RandomForestClassifier

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

# Building a sentiment classification model (Random Forest Classifier in this example)
model4 = RandomForestClassifier(n_estimators=100, random_state=0)  # Define Model 4 as Random Forest
model4.fit(X_train, y_train)

# Making predictions
y_pred = model4.predict(X_test)

# Model Evaluation
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

print(f'Model 4 (Random Forest) - Accuracy: {accuracy}')
print(f'Confusion Matrix:\n{conf_matrix}')
print(f'Classification Report:\n{classification_rep}')
```

model2 is initialized as a Bernoulli Naive Bayes classifier (BernoulliNB).
model3 is initialized as a Linear Support Vector Classification (LinearSVC).
Both model2 and model3 are added to a list called model.

The code iterates through each model in the model list.
For each model, it trains the model using the training data (X_train, y_train) and makes predictions on the test data (X_test).
It then prints the confusion matrix, accuracy score, and classification report for the model's predictions.

A Random Forest Classifier (RandomForestClassifier) is initialized with 100 decision trees (n_estimators=100) and a fixed random state for reproducibility (random_state=0).
The data is split into training and test sets using train_test_split.
The Random Forest model is trained using the training data and evaluated on the test data.
The accuracy score, confusion matrix, and classification report are printed for the Random Forest model's predictions.
In summary, this code trains and evaluates multiple classification models (Naive Bayes, Linear SVM, and Random Forest) for sentiment analysis using the provided features (X) and target labels (y). The evaluation metrics are printed for each model, allowing for a comparison of their performance.

```
M-O-D-E-L : 1
Confusion matrix :
 [[2694  532  285]
 [  77  351   81]
 [  17   36  319]]
Accuracy score :  0.7659380692167578
Classification Report :
              precision    recall  f1-score   support

    negative       0.97      0.77      0.86      3511
     neutral       0.38      0.69      0.49       509
    positive       0.47      0.86      0.60       372

    accuracy                           0.77      4392
   macro avg       0.60      0.77      0.65      4392
weighted avg       0.86      0.77      0.79      4392

------------------------------------------------------------

M-O-D-E-L : 2
Confusion matrix :
 [[2780  850  670]
 [   8   69   13]
 [   0    0    2]]
Accuracy score :  0.6491347905282332
Classification Report :
              precision    recall  f1-score   support

    negative       1.00      0.65      0.78      4300
     neutral       0.08      0.77      0.14        90
    positive       0.00      1.00      0.01         2
```

```
weighted avg       0.98      0.65      0.77      4392

------------------------------------------------------------

M-O-D-E-L : 3
Confusion matrix :
 [[2620  428  197]
 [ 135  426  100]
 [  33   65  388]]
Accuracy score :  0.7818761384335154
Classification Report :
               precision    recall  f1-score   support

    negative       0.94      0.81      0.87      3245
     neutral       0.46      0.64      0.54       661
    positive       0.57      0.80      0.66       486

    accuracy                           0.78      4392
   macro avg       0.66      0.75      0.69      4392
weighted avg       0.83      0.78      0.80      4392

------------------------------------------------------------

Model 4 (Random Forest) - Accuracy: 0.7619535519125683
Confusion Matrix:
[[1804   63   22]
 [ 356  199   25]
 [ 189   42  228]]
Classification Report:
              precision    recall  f1-score   support

    negative       0.77      0.96      0.85      1889
     neutral       0.65      0.34      0.45       580
    positive       0.83      0.50      0.62       459

    accuracy                           0.76      2928
   macro avg       0.75      0.60      0.64      2928
weighted avg       0.76      0.76      0.74      2928
```

Count of tweets:

```
print("Total number of tweets for each airline \n
",df.groupby('airline')['airline_sentiment'].count().sort_values(ascending=False))
airlines= ['US Airways','United','American','Southwest','Delta','Virgin America']
plt.figure(1,figsize=(12, 12))
for i in airlines:
    indices= airlines.index(i)
    plt.subplot(2,3,indices+1)
    new_df=df[df['airline']==i]
    count=new_df['airline_sentiment'].value_counts()
    Index = [1,2,3]
    plt.bar(Index,count, color=['red', 'green', 'blue'])
    plt.xticks(Index,['negative','neutral','positive'])
    plt.ylabel('Mood Count')
    plt.xlabel('Mood')
    plt.title('Count of Moods of '+i)
```

airlines: A list containing the names of specific airlines.

plt.figure(1, figsize=(12, 12)): This line initializes a new figure for the subplot with a size of 12x12 inches.

The code then iterates through each airline in the airlines list and creates a subplot for each airline.

For each subplot, it filters the DataFrame df to include only tweets related to the specific airline.

It counts the distribution of sentiment labels ('negative', 'neutral', 'positive') in the filtered data.

A bar chart is created for each airline, showing the count of each sentiment label.

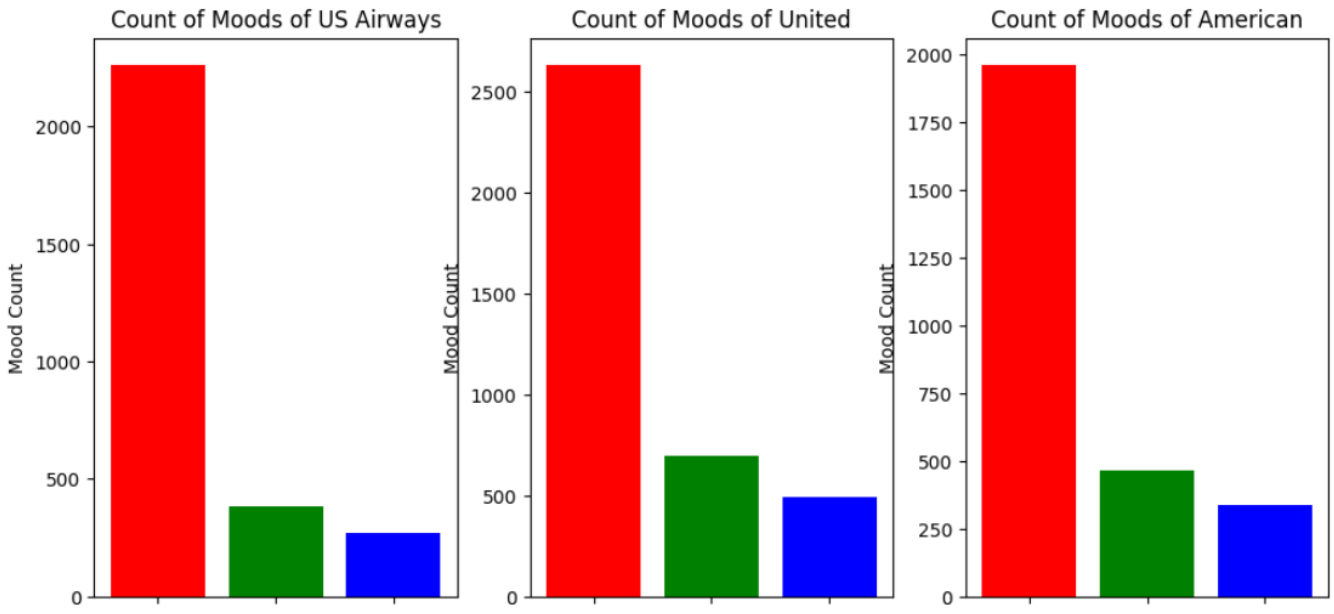
X-axis labels are set as ['negative', 'neutral', 'positive'].

Y-axis represents the count of tweets for each sentiment label.

The subplot title indicates the airline for which the mood distribution is being displayed.

In summary, this code provides a visual representation of the sentiment distribution (negative, neutral, and positive) for tweets related to specific airlines, allowing for a quick comparison of the sentiment distribution across different airlines. The code achieves this by creating individual bar charts for each airline's sentiment distribution within a larger subplot.

```
Total number of tweets for each airline
  airline
United            3822
US Airways        2913
American          2759
Southwest         2420
Delta             2222
Virgin America     504
Name: airline_sentiment, dtype: int64
```



Count of positive tweets:

```
# Percentage of Positive Tweets in Each Airlines
pos_tweets = df.groupby(['airline', 'airline_sentiment']).count().iloc[:, 0]
total_tweets = df.groupby(['airline'])['airline_sentiment'].count()

my_dict = {'American': pos_tweets[2] / total_tweets[0], 'Delta': pos_tweets[5] / total_tweets[1], 'Southwest': pos_tweets[8] / total_tweets[2],
           'US Airways': pos_tweets[11] / total_tweets[3], 'United': pos_tweets[14] / total_tweets[4], 'Virgin': pos_tweets[17] / total_tweets[5]}
perc = pd.DataFrame.from_dict(my_dict, orient='index')
perc.columns = ['Percent Positive']
print(perc)
ax = perc.plot(kind='bar', rot=0, colormap='Blues_r', figsize=(15, 6))
ax.set_xlabel('Airlines')
ax.set_ylabel('Percentage of positive tweets')
plt.show()
```

pos_tweets: Counts the number of positive tweets for each airline.

total_tweets: Counts the total number of tweets for each airline.

The code calculates the percentage of positive tweets for each airline using the formula (number of positive tweets / total number of tweets) * 100.

The results are stored in a dictionary called my_dict.

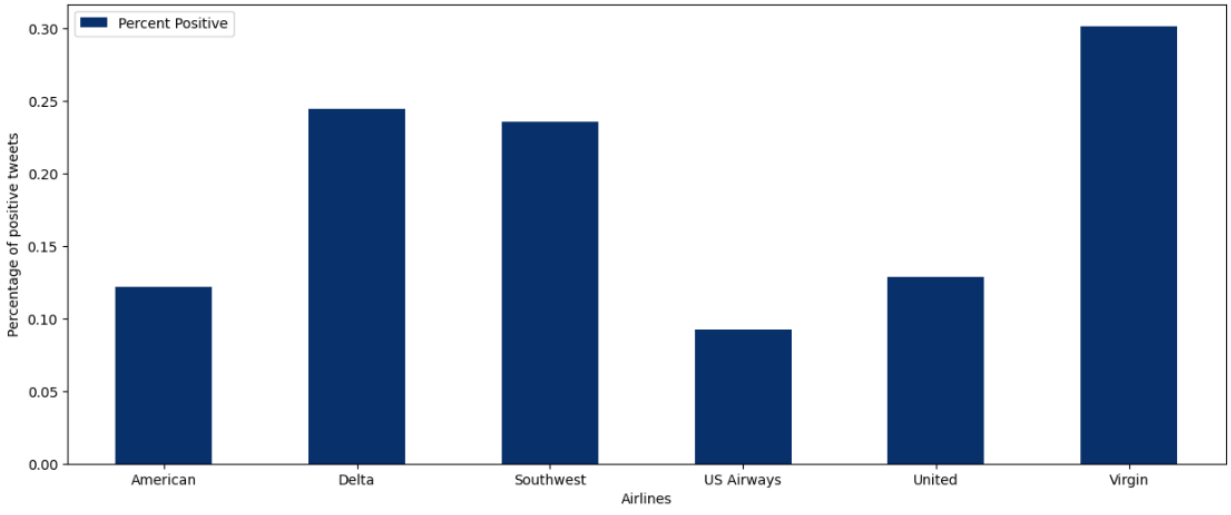
The dictionary is then converted into a Pandas DataFrame called perc with the airlines as index and 'Percent Positive' as the column name.

perc.plot(kind='bar', rot=0, colormap='Blues_r', figsize=(15, 6)): Creates a bar chart using the Pandas DataFrame perc. The kind='bar' parameter specifies the type of plot. rot=0 sets the x-axis labels to be horizontal, and colormap='Blues_r' defines the color scheme for the bars. figsize=(15, 6) sets the size of the plot.

ax.set_xlabel('Airlines'): Sets the label for the x-axis.

ax.set_ylabel('Percentage of positive tweets'): Sets the label for the y-axis.

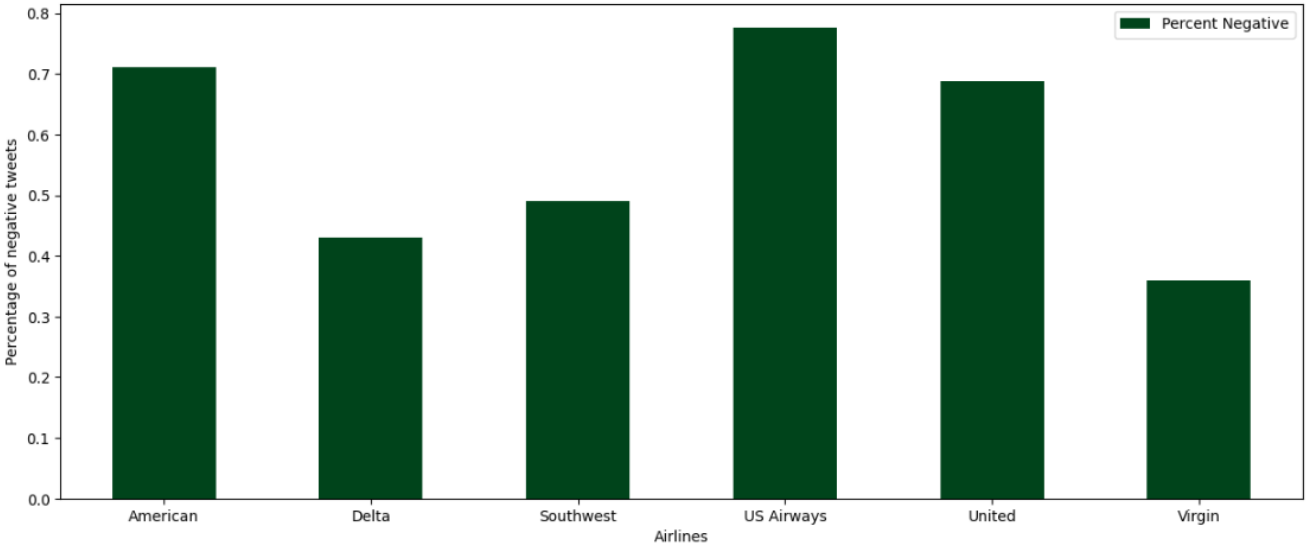
```
            Percent Positive
American            0.121783
Delta               0.244824
Southwest           0.235537
US Airways          0.092345
United              0.128728
Virgin              0.301587
```



Count of negative tweets:

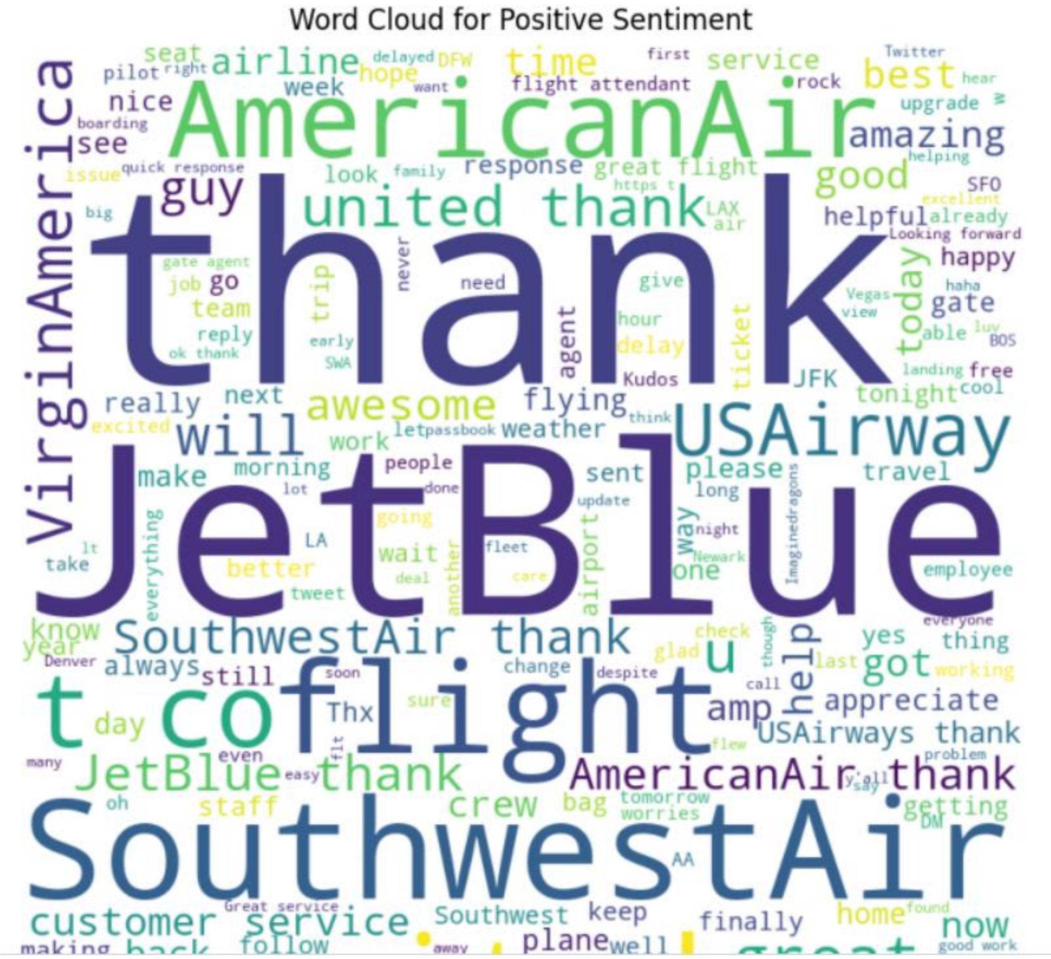
```
#Percentage of Negative Tweets in Each Airlines
neg_tweets = df.groupby(['airline','airline_sentiment']).count().iloc[:,0]
total_tweets = df.groupby(['airline'])['airline_sentiment'].count()

my_dict = {'American':neg_tweets[0] / total_tweets[0],'Delta':neg_tweets[3] / total_tweets[1],'Southwest':
neg_tweets[6] / total_tweets[2],
'US Airways': neg_tweets[9] / total_tweets[3],'United': neg_tweets[12] / total_tweets[4],'Virgin': neg_tweets[15] /
total_tweets[5]}
perc = pd.DataFrame.from_dict(my_dict, orient = 'index')
perc.columns = ['Percent Negative']
print(perc)
ax = perc.plot(kind = 'bar', rot=0, colormap = 'Greens_r', figsize = (15,6))
ax.set_xlabel('Airlines')
ax.set_ylabel('Percentage of negative tweets')
plt.show()
```

```
            Percent Negative
American            0.710402
Delta               0.429793
Southwest           0.490083
US Airways          0.776862
United              0.688906
Virgin              0.359127
```

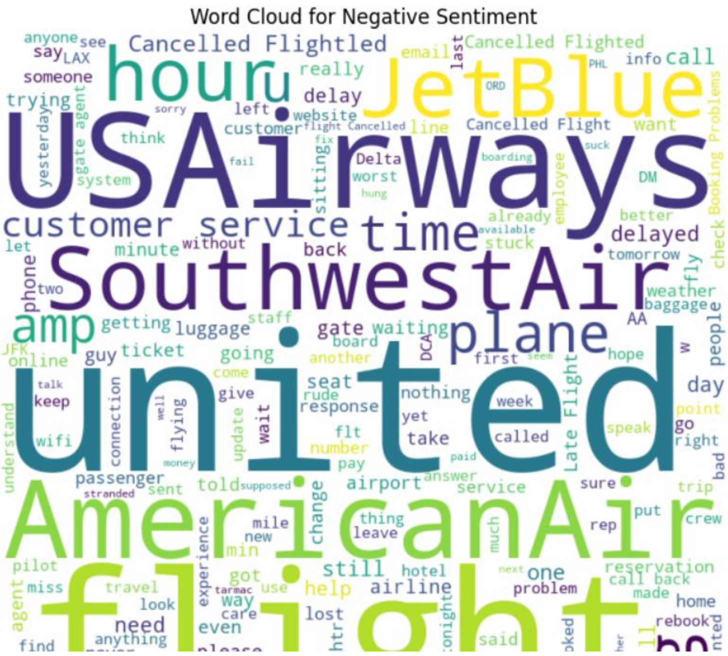


Word cloud for positive and negative sentiments:

```
from wordcloud import WordCloud

positive_reviews = ' '.join(df[df['airline_sentiment'] == 'positive']['text'])
wordcloud = WordCloud(width=800, height=800, background_color='white').generate(positive_reviews)
plt.figure(figsize=(8, 8), facecolor=None)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word Cloud for Positive Sentiment")
plt.show()
```

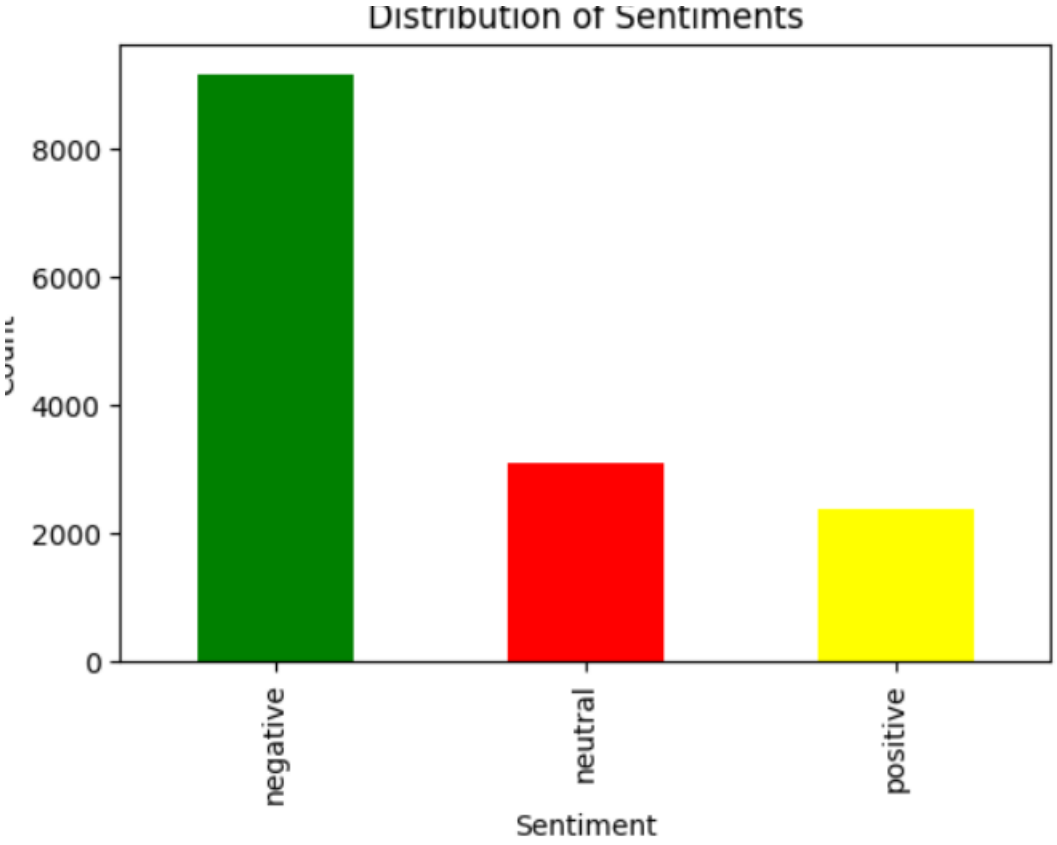


# Word cloud for negative sentiment
negative_reviews = ' '.join(df[df['airline_sentiment'] == 'negative']['text'])
wordcloud = WordCloud(width=800, height=800, background_color='white').generate(negative_reviews)
plt.figure(figsize=(8, 8), facecolor=None)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word Cloud for Negative Sentiment")
plt.show()



Distribution of sentiments:

```
# Example of a bar chart to show the distribution of sentiments
sentiment_counts = df['airline_sentiment'].value_counts()
plt.figure(figsize=(6, 4))
sentiment_counts.plot(kind='bar', color=['green', 'red', 'yellow'])
plt.title('Distribution of Sentiments')
plt.xlabel('Sentiment')
plt.ylabel('Count')
plt.show()
```



<u>Count reasons for all</u>

```
#Count of Negative Reasons for all Airlines
df['negativereason'].nunique()

NR_Count=dict(df['negativereason'].value_counts(sort=False))
def NR_Count(Airline):
    if Airline=='All':
        a=df
    else:
        a=df[df['airline']==Airline]
    count=dict(a['negativereason'].value_counts())
    Unique_reason=list(df['negativereason'].unique())
    Unique_reason=[x for x in Unique_reason if str(x) != 'nan']
    Reason_frame=pd.DataFrame({'Reasons':Unique_reason})
    Reason_frame['count']=Reason_frame['Reasons'].apply(lambda x: count[x])
    return Reason_frame
def plot_reason(Airline):

    a=NR_Count(Airline)
    count=a['count']
    Index = range(1,(len(a)+1))
    plt.bar(Index,count, color=['red','yellow','blue','green','black','brown','gray','cyan','purple','orange'])
    plt.xticks(Index,a['Reasons'],rotation=90)
```

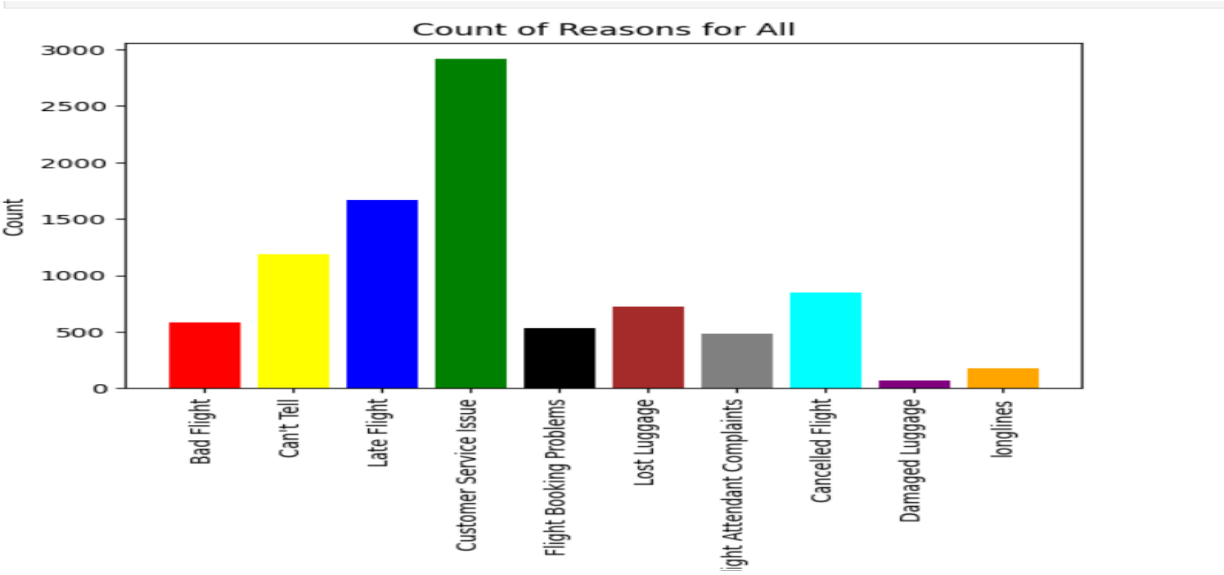
```
    plt.ylabel('Count')
    plt.xlabel('Reason')
    plt.title('Count of Reasons for '+Airline)

plot_reason('All')
plt.figure(2,figsize=(13, 13))
for i in airlines:
    indices= airlines.index(i)
    plt.subplot(2,3,indices+1)
    plt.subplots_adjust(hspace=0.9)
    plot_reason(i)
```

NR_Count(Airline): This function takes an airline name as input ('All' or a specific airline) and returns a DataFrame containing unique negative reasons and their respective counts for the specified airline.

plot_reason(Airline): This function takes an airline name as input and plots a bar chart showing the count of negative reasons for the specified airline.

This part of the code creates subplots for individual airlines. For each airline in the airlines list, it calls the plot_reason function to generate a separate bar chart showing the count of negative reasons for that specific airline.

In summary, this code calculates and visualizes the count of negative reasons for both individual airlines and all airlines combined. The results are displayed using multiple subplots, allowing for a detailed comparison of negative reasons across different airlines. The y-axis represents the count of negative reasons, and the x-axis represents the specific negative reasons for each airline.



Distribution of positive and negative reviews in the world

```
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt

# Load your "tweets.csv" dataset using pandas
df = pd.read_csv('tweets.csv')

# Group data by "tweet location" and "sentiment" and calculate counts
sentiment_counts = df.groupby(['tweet_location', 'airline_sentiment']).size().unstack().fillna(0)

# Load a shapefile of world countries or regions
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))

# Merge sentiment data with geographical data based on "tweet location"
world = world.merge(sentiment_counts, left_on='name', right_on='tweet_location', how='left')
world.fillna(0, inplace=True)

# Create a color-coded map with two colors (green for positive and red for negative, white for no data)
fig, ax = plt.subplots(1, 1, figsize=(15, 10))
world['sentiment_color'] = 'grey'  # Initialize the sentiment color as white
world.loc[world['positive'] > 0, 'sentiment_color'] = 'green'  # Set positive regions to green
world.loc[world['negative'] > 0, 'sentiment_color'] = 'red'  # Set negative regions to red

world.plot(facecolor=world['sentiment_color'], ax=ax, legend=False)

plt.title("Distribution of Positive (Green) and Negative (Red) Reviews in the World")
plt.show()
```

The data is grouped by 'tweet_location' and 'airline_sentiment'.

The size() function calculates the count of tweets for each combination of location and sentiment.

unstack() transforms the grouped data into a table format.

fillna(0) fills NaN values with 0.

Geographical data of world countries or regions is loaded using Geopandas.

The sentiment data (counts of positive and negative reviews by location) is merged with the geographical data based on the 'name' column.

left_on='name' and right_on='tweet_location' specify the columns used for merging.

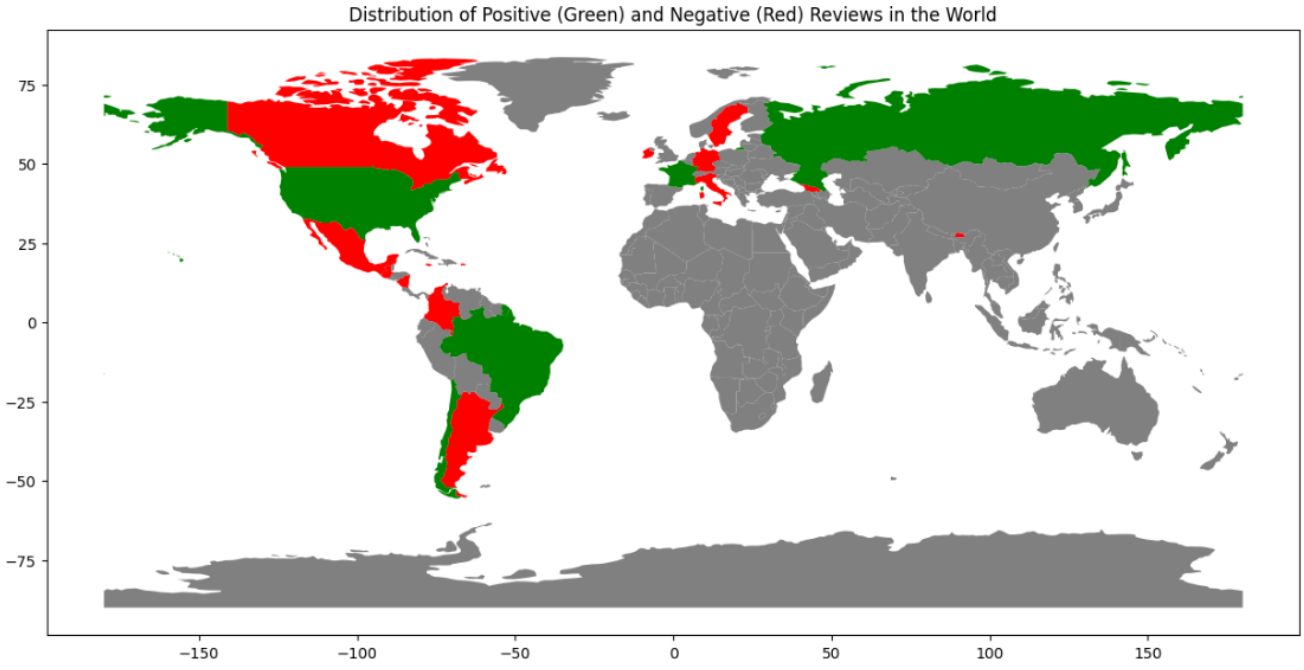
Regions with no sentiment data are filled with 0.

A figure and an axis are created for the plot.

The world map is plotted with face colors determined by the 'sentiment_color' column.

The legend is turned off (legend=False).

In summary, this code creates a world map where countries or regions are color-coded based on the sentiment of tweets (positive, negative, or no data) from the provided CSV file. Positive regions are shown in green, negative regions in red, and regions with no data in grey. This visualization provides an overview of the global sentiment distribution of the tweets.


Distribution of Positive (Green) and Negative (Red) Reviews in the World
75
50
25
0
−25
−50
−75
−150
−100
−50
0
50
100
150