SENTIMENT ANALYSIS SOLUTION FOR KENYA AIRWAYS

SYDNEY NZUNGULI

2024

1 OVERVIEW OF THE SYSTEM

The sentiment analysis solution is designed to gather, analyze, and present customer feedback from various online platforms. The primary goal is to extract insights from social media and travel review sites to understand customer sentiments toward the airline's services. The system design will cover the architecture, components, and key considerations.

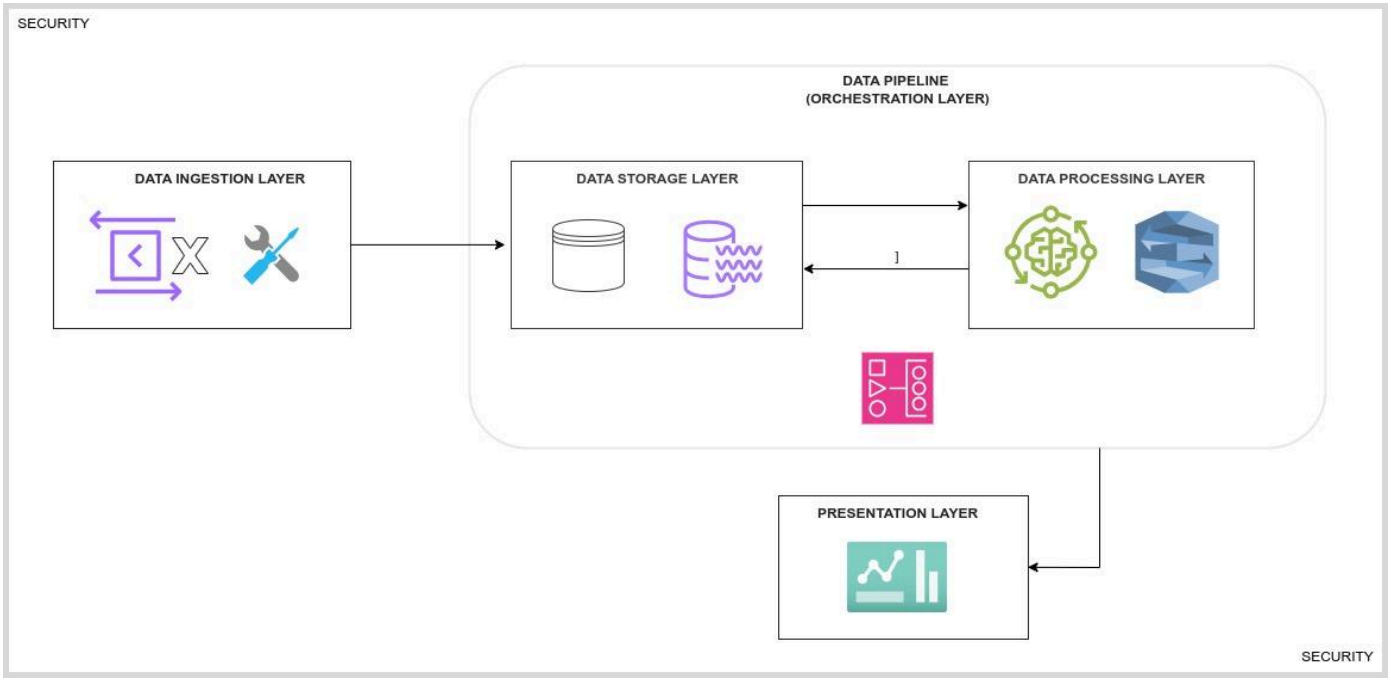
2 SYSTEM DESIGN

2.1 Data Flow

1. *Data Collection*: Data is ingested from social media platforms and travel review sites via APIs or web scraping. Selected platform is X (X), whereas the travel site is TripAdvisor (TripAdvisor).
2. *Data Storage*: The raw data is stored in the database or data lake.
3. *Data Processing*: Sentiment analysis is performed on the data, and the results are stored in the database.
4. *Data Presentation*: The processed data is visualized and presented to users via a dashboard.

2.2 Architectural Design and Discussion

Below is a diagram representing the system architectural design. The diagram includes several components; Data Ingestion (Scraping) Layer, Data Storage Layer, Data Processing Layer, Data Pipeline Layer and the Presentation Layer. The diagram also shows the data flow from one component to the next with each component having its own distinctive features and technologies which are discussed below.



The system is divided into several key components:

Data Ingestion Layer (Scraping Module):

This layer is responsible for gathering data from various sources, such as social media platforms (e.g., X) and travel review sites (e.g., TripAdvisor). It utilizes API integrations like the X API, and for platforms without APIs, it employs web scraping tools like BeautifulSoup and Scrapy. Key processes in this layer include:

1. Scheduling regular data scraping intervals
2. Managing API rate limits to comply with service terms
3. Implementing error handling mechanisms to address failed requests or changes in website structure.

Data Storage Layer:

The primary function of this layer is to store both the raw data from the ingestion layer and the processed data, such as analyzed sentiment results. This layer can utilize relational databases like PostgreSQL for structured data and NoSQL databases like MongoDB for unstructured data. A data lake is also included to handle large volumes of raw, unstructured data, ensuring that historical data is retained. Core processes involve:

1. Storing raw data in its original format for potential future reprocessing
2. Indexing processed data to enable efficient retrieval.

Data Processing Layer (Sentiment Analysis Module):

This layer focuses on analyzing the collected data to assess the sentiment of each mention, categorizing it as positive, negative, or neutral. It leverages Natural Language Processing (NLP) libraries such as NLTK, TextBlob, or spaCy, alongside machine learning models like BERT or RoBERTa, either pre-trained or custom-trained using frameworks like TensorFlow or PyTorch, which are also responsible for handling Optical Character Recognition (OCR) in multimedia processing. This ensures that text extracted from images, video frames, or transcribed audio is included in the sentiment analysis pipeline. Key processes include:

1. Text preprocessing: Tokenization, stop word removal, and stemming/lemmatization to prepare the text for analysis.
2. Sentiment classification: Using NLP techniques to categorize sentiment as positive, negative, or neutral.
3. Multimedia text extraction: Applying OCR to extract text from images, videos, or other multimedia formats, integrating this data into the sentiment analysis workflow.
4. Aggregation of sentiment scores: Compiling sentiment scores over time or by specific keywords to identify trends or patterns.

Data Pipeline (Orchestration Layer):

This layer automates the entire ETL workflow, ensuring smooth data flow from ingestion through to analysis and storage. It utilizes workflow orchestration tools like Apache Airflow, and for real-time data processing, data streaming platforms like Kafka or Apache Pulsar. The main processes here involve:

1. Scheduling scraping jobs
2. Triggering sentiment analysis upon data ingestion
3. Managing dependencies between various processing tasks to ensure seamless execution.

Presentation Layer (User Interface/Dashboard):

The presentation layer is designed to display the results of sentiment analysis in a clear and user-friendly interface. It utilizes frontend frameworks like React.js, Angular, or Vue.js, with visualization libraries like D3.js or Chart.js, or even business intelligence tools like Tableau or PowerBI. This layer focuses on:

1. Creating visual representations of sentiment trends
2. Offering data filtering options by date, sentiment, or keywords

3. Providing summary statistics, alerts, and actionable insights derived from the sentiment analysis.

Security Layer:

A protective boundary surrounding the entire system, ensuring encryption, secure API key management, and compliance with data regulations like GDPR across all layers.

2.3 System Implementation (Algorithms and Pseudocodes)

Data Ingestion Layer (Scraping)

1. *API data collection*

   This script authenticates with X API v2 using the `tweepy` library to search for recent tweets mentioning "Kenya Airways." The fetched tweets are extracted and saved to a JSON file named `kenya_airways_tweets_v2.json` for later analysis. Error handling is implemented to manage potential issues during the API request.

```
import tweepy
import json

# set up API credentials
bearer_token = ''

# authenticate with X API v2
client = tweepy.Client(bearer_token=bearer_token)

# query for recent tweets mentioning 'Kenya Airways'
query = 'Kenya Airways'
max_results = 100

try:
    # fetch tweets
            response    =    client.search_recent_tweets(query=query,
max_results=max_results)

    # extract tweet data
    tweets = response.data

    if tweets:
        # save tweets to a JSON file
        with open('kenya_airways_tweets_v2.json', 'w') as f:
```

```
                json.dump([tweet.text for tweet in tweets], f,
indent=4)
            print(f"Successfully fetched and saved {len(tweets)}
tweets to 'kenya_airways_tweets_v2.json'.")
    else:
        print("No tweets found for the given query.")

except tweepy.TweepyException as e:
    print(f"An error occurred: {e}")
```

2. *Web Scraping*

This script uses the `requests` library to fetch the HTML content of a TripAdvisor review page and `BeautifulSoup` to parse and extract reviews. The reviews are then written to a CSV file named `kenya_airways_tripadvisor_reviews.csv` for storage and further analysis.

```
import requests
from bs4 import BeautifulSoup
import csv

url                                                              =
'https://www.tripadvisor.com/Hotel_Review-g294207-d302925-Reviews
-Kenya_Airways-Nairobi.html'
headers = {'User-Agent': 'Mozilla/5.0'}

response = requests.get(url, headers=headers)
soup = BeautifulSoup(response.content, 'html.parser')

reviews = soup.find_all('q', {'class': 'IRsGHoPm'})

# Open a CSV file to write the reviews
with open('kenya_airways_tripadvisor_reviews.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerow(['Review'])

    for review in reviews:
        writer.writerow([review.get_text()])

print("Reviews scraped and saved successfully.")
```

Data Storage Layer

The Data Storage Layer is responsible for saving both raw and processed data collected from the Data Ingestion Layer. The goal is to efficiently store structured data in a relational database (PostgreSQL) and unstructured data in a NoSQL database (MongoDB).

1. *Store raw data*

```
# configure database connections
# Connect to PostgreSQL for structured data
connect_to_postgresql()
# Connect to MongoDB for unstructured data
connect_to_mongodb()

# store raw data in MongoDB
def store_raw_data_in_mongodb(raw_data):
   insert_raw_data_into_mongodb(raw_data)

# process data
def process_data(raw_data)...

# store processed data in PostgreSQL
def store_processed_data_in_postgresql(processed_data):
   for data in processed_data:
                        insert_data_into_postgresql(data['text'],
data['sentiment'])

# commit and close conn
commit_transactions()
close_database_connections()
```

Despite PostgreSQL capabilities, MongoDB is chosen for handling large-scale unstructured data due to its optimized performance for such use cases. Thus, both databases are retained in the design to handle different data types efficiently.

2. *Data Lake*

A data lake is a centralized repository that allows one to store vast amounts of raw, unstructured, or semi-structured data at any scale. Unlike traditional databases that require data to be organized and structured, a data lake can handle data in its native format, making it ideal for storing raw data that might need to be processed or analyzed later.

The script demonstrates how to store raw data in a data lake using ‘Amazon S3’, a widely used storage service that acts as a data lake.

```
import boto3

s3 = boto3.client('s3')

# Function to store raw data in the data lake
def store_in_data_lake(bucket_name, file_name, data):
   s3.put_object(Bucket=bucket_name, Key=file_name, Body=data)

# convert data to a json string then uploaded to the s3 bucket
store_in_data_lake('my-data-lake',                'raw_data.json',
json.dumps(data))
```

Data Processing Layer

1. *Text Processing*

   The script cleans and prepares the text data for sentiment analysis by removing irrelevant words and standardizing the text.

```
# nltk is used as a powerful library used for text processing in
python
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

#download relevant tools, a tokenizer for splitting sentences
into words and a list of common stop words in english
nltk.download('punkt')
nltk.download('stopwords')

def preprocess_text(text):
   tokens = word_tokenize(text)
   tokens = [word.lower() for word in tokens if word.isalnum()]
      tokens = [word for word in tokens if word not in
stopwords.words('english')]
   return tokens
```

2. *Sentiment Classification*

   This script evaluates the sentiment of the input text, categorizing it as positive, negative, or neutral based on the polarity score provided by TextBlob.

```
#textblob is a simple library for processing textual data and
performing nlp  tasks like sentiment analysis
from textblob import TextBlob

#automatically process input text, calculates and classifies
sentiment scores
def analyze_sentiment(text):
    analysis = TextBlob(text)
    if analysis.sentiment.polarity > 0:
        return 'Positive'
    elif analysis.sentiment.polarity < 0:
        return 'Negative'
    else:
        return 'Neutral'
```

3. *Aggregation of Sentiment Scores*

   This script aggregates sentiment results, summarizing how many times each sentiment (positive, negative, neutral) appears in the dataset, providing a clear picture of overall sentiment distribution.

```
from collections import Counter

#count how many times each sentiment appears in the list
def aggregate_sentiments(sentiments):
    counter = Counter(sentiments)
    return counter
```

Data Pipeline Layer

The Data Pipeline Layer, also known as the Orchestration Layer, automates and manages the entire data workflow from ingestion through processing to storage. This layer ensures that each step in the data pipeline is executed in the correct sequence, dependencies between tasks are handled, and any issues are managed efficiently.

```
# task Scheduling for regular data scraping jobs and schedule other
tasks like sentiment analysis and data storage once ingestions tasks
are completed
def schedule_tasks():
    # schedule data scraping tasks
    schedule_scraping_task(interval='daily', task=scrape_twitter)
    schedule_scraping_task(interval='daily', task=scrape_tripadvisor)
```

```
    # schedule sentiment analysis to run after scraping
                        schedule_analysis_task(dependency=scrape_twitter,
task=analyze_twitter_data)
                    schedule_analysis_task(dependency=scrape_tripadvisor,
task=analyze_tripadvisor_data)
    # schedule data storage tasks to run after analysis
                  schedule_storage_task(dependency=analyze_twitter_data,
task=store_twitter_data)
              schedule_storage_task(dependency=analyze_tripadvisor_data,
task=store_tripadvisor_data)

# handle task dependencies to ensure sentiment analysis only runs after
data has been ingested and storage tasks only run after sentiment
analysis is completed
# Scraping X data
def scrape_tweets()...
# Scraping TripAdvisor data
def scrape_tripadvisor()...
# analyze sentiment for X data
def analyze_tweets_data(x_data)...
# Analyze sentiment for TripAdvisor data
def analyze_tripadvisor_data(tripadvisor_data)...
# Store processed X data
def store_tweets_data(analyzed_x_data)...
# Store processed TripAdvisor data
def store_tripadvisor_data(analyzed_tripadvisor_data)...

# monitor and manage Task Execution
def monitor_tasks():
    # Implement error handling and retries to manage failed tasks
    try:
        execute_pipeline()
    except Exception as e:
        log_error(e)
        retry_task()

# orchestrate ETL data flow
def execute_pipeline():
     # orchestrating the entire data pipeline to automate the transition
from one task to another
    x_data = scrape_tweets()
    analyzed_x_data = analyze_tweets_data(x_data)
    store_tweets_data(analyzed_x_data)
```

```
    tripadvisor_data = scrape_tripadvisor()
                                    analyzed_tripadvisor_data            =
analyze_tripadvisor_data(tripadvisor_data)
    store_tripadvisor_data(analyzed_tripadvisor_data)

# close and clean up resources
def close_resources():
    # close db connections, API sessions etc
    close_database_connections()
    log_pipeline_completion()
```

Security Layer

This layer is crucial for safeguarding data, managing access, and ensuring compliance with regulations throughout all stages of the data lifecycle and is integrated across all layers of the system to provide comprehensive protection. Here's how the security aspects can be implemented:

1. *Data Encryption*
    I. **At Rest:** Ensure that all data stored in the database (PostgreSQL, MongoDB) and data lake (Amazon S3) is encrypted. In Amazon S3, you can enable server-side encryption (SSE) to automatically encrypt the data.

```
# use SSL for PostgreSQL connection
conn = psycopg2.connect(
    database="sentiment_db",
    user="user",
    password="password",
    host="localhost",
    port="5432",
    sslmode='require'  # SSL encryption
)

# enable SSL/TLS for S3 communication
s3 = boto3.client('s3', use_ssl=True)
```

    II. **In Transit:** Use HTTPS/TLS for secure communication between services and when accessing external APIs (e.g., Twitter API, TripAdvisor scraping). This prevents man-in-the-middle attacks and eavesdropping.

2. *API Key Management*

I. **Secure Storage:** Store API keys and secrets (e.g., Twitter API, MongoDB credentials) in a secure vault or environment variables, rather than hardcoding them in the script.

```
import os

# Retrieve API keys securely
bearer_token = os.getenv('TWITTER_BEARER_TOKEN')
db_password = os.getenv('DB_PASSWORD')
```

II. **Access Control:** Limit API key permissions to only what is necessary, following the principle of least privilege. Rotate API keys regularly and monitor their usage for any suspicious activity.

3. *Rate Limiting Gateway*

I. **Prevent DoS Attacks:** By limiting the number of requests from a single source within a specific time frame, the gateway helps to prevent malicious actors from overwhelming the system. One can use middleware or external services like NGINX.

II. **API Protection:** Protects APIs from abuse by enforcing limits on how frequently they can be called, which is especially important for public-facing endpoints.

AWS API gateway: creating a usage plan with specific rate limits (e.g., 1000 requests per minute) in the API Gateway console

4. *Access Control*

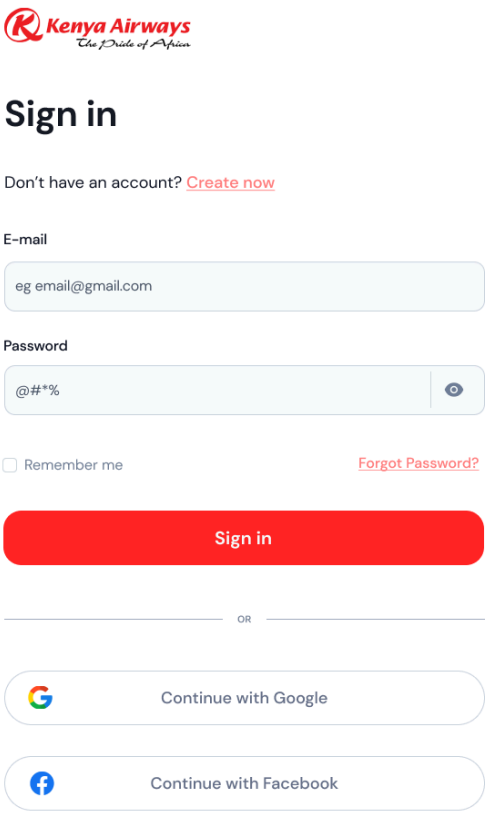
I. **User Authentication:** Implement authentication mechanisms for accessing the database, dashboards, and other critical resources. This can involve username/password, OAuth, or multi-factor authentication (MFA).

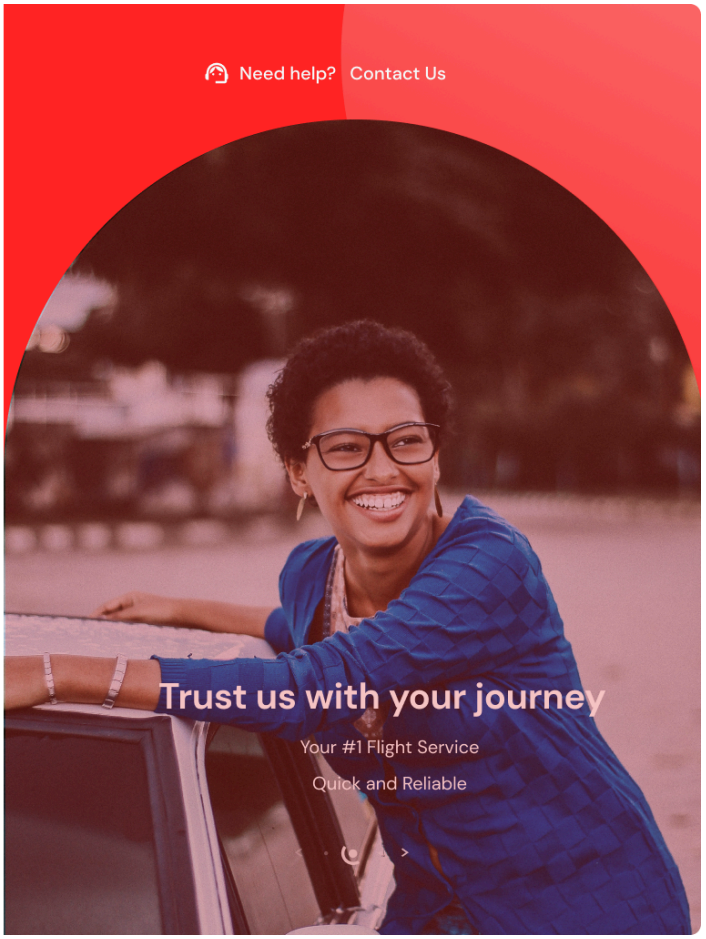
II. **Role-Based Access Control (RBAC):** Define roles and permissions to control which users or services can access or modify data. For instance, restrict write access to only specific services or users.

3 UI/UX DESIGN

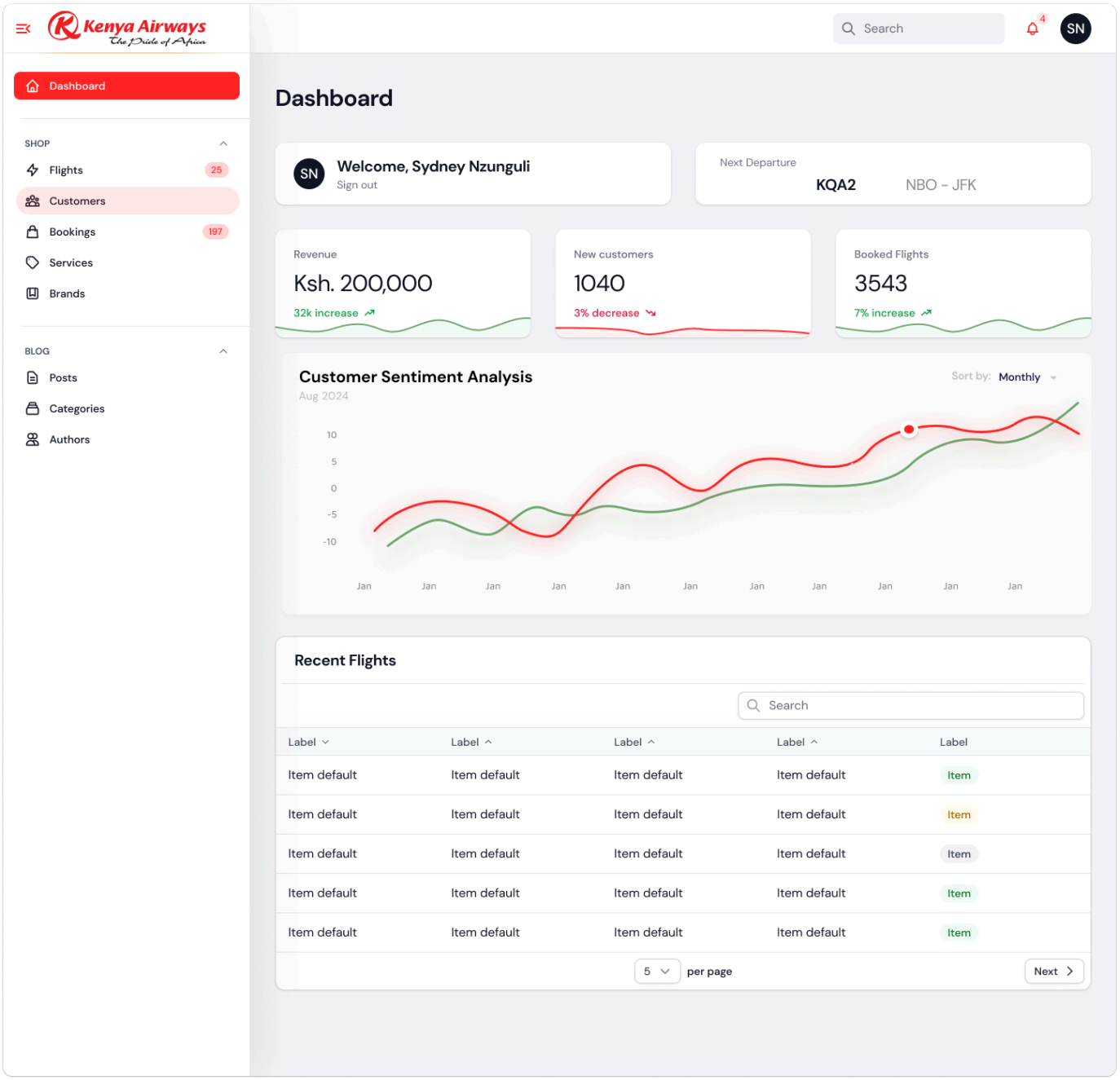
The UI/UX designs were finally created to give a seamless user experience. The designs include a simple sign in/sign up page to access the system dashboard, and once successful the user is directed to the main dashboard where they're presented with the necessary information on the sentimental graph. The theme chosen for the designs was a minimalistic professional style, giving the users a comfortable yet easy feel to the system. Below are the pages that were designed:

1. Login/ Sign up Page

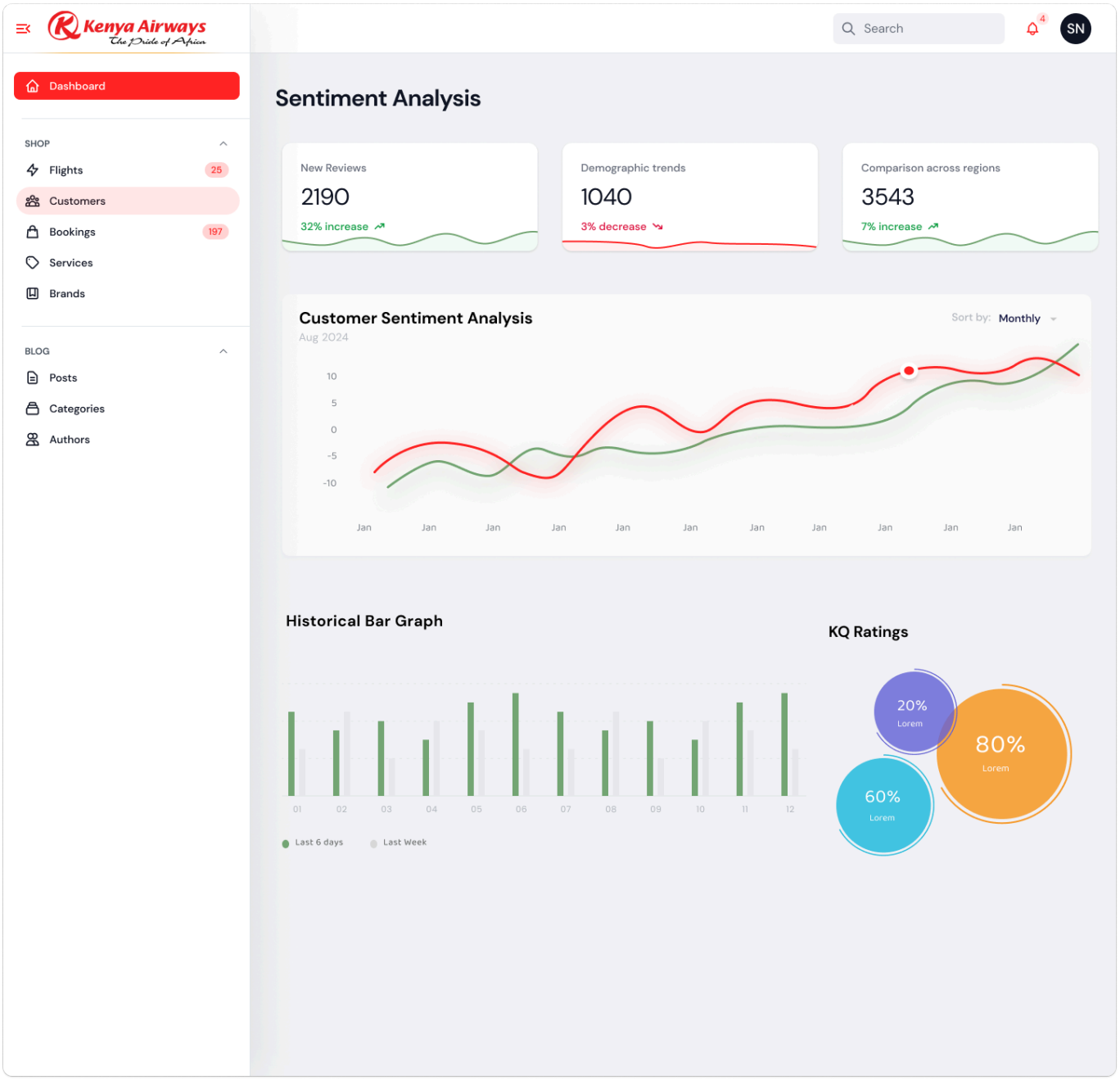




2. Main Dashboard

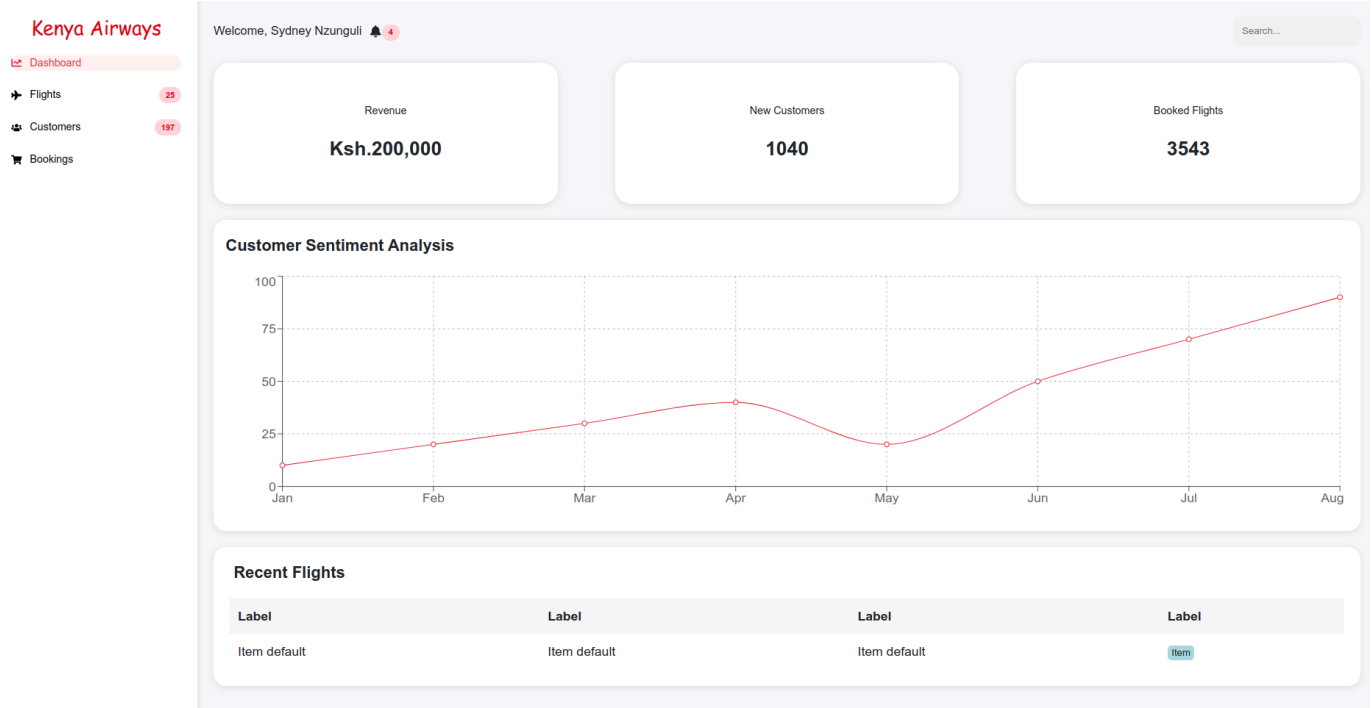


3. Sentiment Analysis



4 UI IMPLEMENTATION RESULTS

The User Interface implementation included using Reactjs as the frontend framework. Rechart is a library in Reactjs that was chosen to create a visual of the sentiment graph. The styling of the components was done via bootstrap css. The backend and frontend connections were also established. However the overall functionality of the system still needs adjustments and improvements.



5 CONCLUSION

In conclusion, the design and implementation of the system were accomplished within a minimum timeframe of 48 hours. Several challenges arose during this period, particularly in setting up developer accounts for social media API access, which significantly complicated the ETL process. Despite these challenges, the experience was both demanding and rewarding, as it required the development of a solution under tight deadlines. This process also sparked my curiosity and interest in exploring the various technologies and trends encountered during the research and development stages.