Designing a MailChimp Marketing Trends Engine

Section 1 – Background & Methodology

Background

In the age of digital information, social media platforms like Facebook, Twitter have become invaluable sources of real-time insights and trends. Organizations increasingly turn to social media data for applications such as market analysis, customer sentiment tracking, and trend forecasting. However, the dynamic nature and sheer volume of Social Media Platforms require a robust infrastructure to automate data collection, enrichment, and storage processes.

To handle these needs, ETL (Extract, Transform, Load) pipelines are commonly employed. The **Extract** phase retrieves data from a source (e.g, Twitter). The **Transform** phase processes and enriches this data by adding metadata like sentiment and engagement scores, making it more insightful. Finally, the **Load** phase stores the transformed data in databases (MongoDB) and data lakes(s3), making it available for future analysis.

A combination of **Apache Spark** and **Apache Airflow** provides a solution that can efficiently do the job. Apache Spark, enables distributed processing & Apache Airflow, serves as a workflow orchestrator, automating and scheduling the execution of ETL tasks.

Once the processed Social Media data is available for analysis in storage (e.g., Amazon S3 or MongoDB), the pipeline extends to further enrich this data by converting it into **vector embeddings using Hugging Face Transformer like BERT**. These embeddings capture semantic information from the text data, csv's, etc., enabling advanced analysis techniques like **clustering** for grouping similar tweets and trends. The embeddings are then stored in **Pinecone**, a vector database optimized for similarity search. With this setup, users can access a front-end application connected to Pinecone to retrieve outcomes based on semantic queries. This allows for highly relevant, context-aware responses, enhancing the utility of the data for decision-making.

Methodology

This methodology outlines the complete end-to-end pipeline for automating social media (I am using Twitter data as an example) processing, enrichment, and semantic retrieval. The pipeline is composed of TWO stages: **Data Processing & Workflow Orchestration with Apache Spark** & **Airflow**, and **Semantic Storage and Querying with BERT Transformer & Pinecone**.

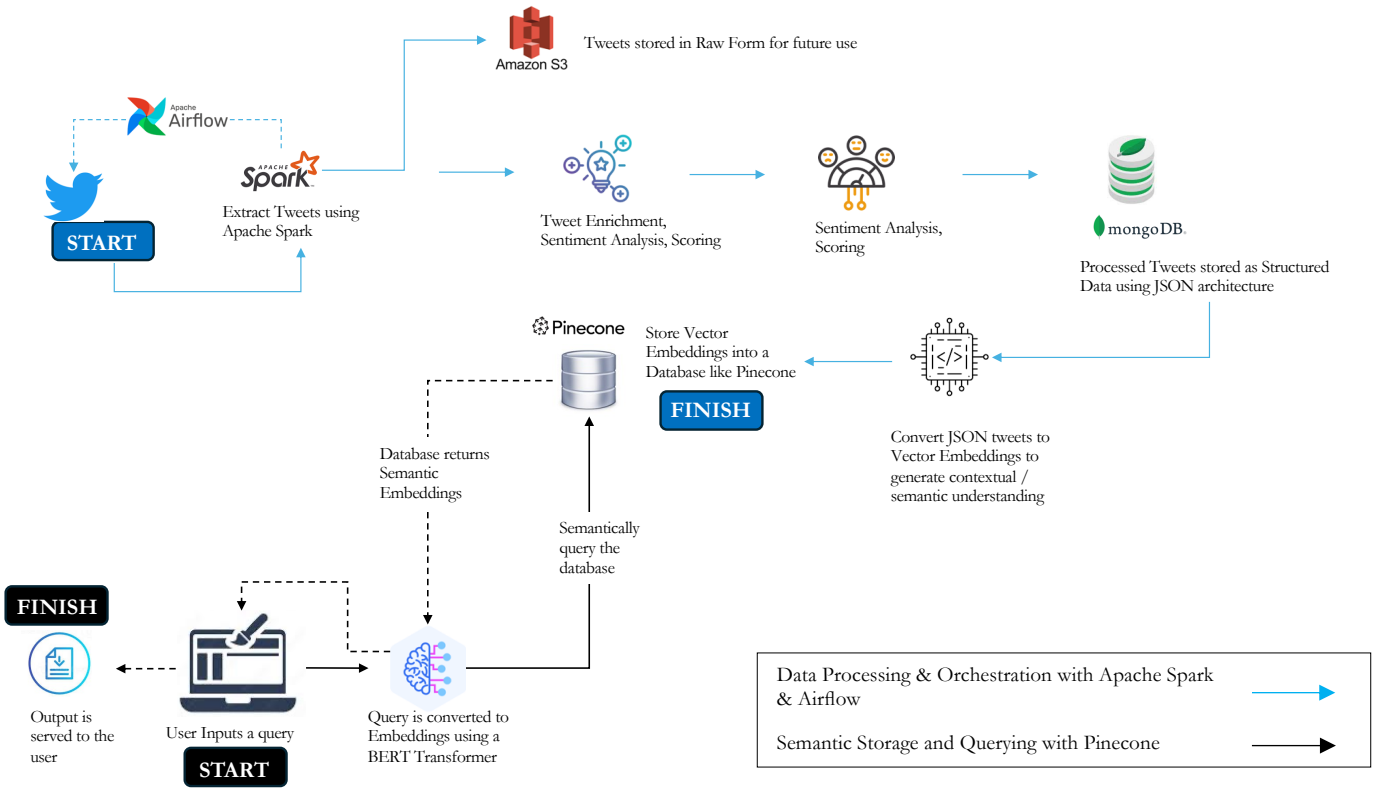
1. **Data Processing & Orchestration with Apache Spark & Airflow**

    - **Extraction**: The Spark script initiates the ETL pipeline by fetching tweets based on specified keywords or topics.
    - **Transformation**: The extracted tweets are then filtered and enriched. This process includes analyzing the sentiment of each tweet using NLP techniques and calculating engagement scores based on metrics like retweets and likes.
    - **Loading**: The enriched data is saved to two storage solutions. MongoDB serves as a flexible database for real-time querying, while S3 acts as a scalable data lake, enabling long-term storage and easy access.
    - **Scheduling and Execution**: Apache Airflow orchestrates and automates the entire ETL pipeline. A Directed Acyclic Graph (DAG) defines the workflow and schedules it to run at specified intervals (e.g., weekly / Monthly).

2. **Semantic Storage and Querying with Pinecone**

    - **Vector Embedding Generation**: Once the data is processed and stored, it is further enriched by converting it into vector embeddings, which capture the semantic information from each tweet. This conversion allows for deeper analysis, such as identifying similar topics or trends within the data.
    - **Storage in Pinecone**: The embeddings are then stored in Pinecone, a vector database optimized for similarity search. Pinecone enables the organization and retrieval of embeddings based on semantic similarity, making it possible to cluster related tweets or topics.
    - **Extract intents & subjects**: LLM would extract intents & meanings using embeddings and group similar topic embeddings across different documents together. This will help in identifying themes & trends.
    - **Consumer Interest Mapping (Tweets)**: Analyze tweets to identify key themes and trends in consumer interest.
    - **Blog and Article Analysis for Topic Heatmap**. By applying named entity recognition (NER) and frequency analysis we create a heatmap of discussion topics. to visualize which themes are most prominent and allows for topic grouping.
    - **Opportunity Sizing Based on Product Sales Data**: Using historical sales data, we forecast potential growth and sales, allowing for opportunity sizing.
    - **Query Interface**: A front-end application connected to Pinecone allows users to query the database and retrieve contextually relevant insights. Through semantic search, users can submit natural language queries and receive results that match the underlying meaning, providing a powerful tool for trend analysis and decision-making.

Process Flow Chart



Section 2 – Data Processing & Orchestration

1. **Setting up Data Crawlers across social media platforms** (Twitter used as an example below to demonstrate concept)

   a. Set up a developer account for social media platforms which would provide user engagement information.

```
# Authenticate with Twitter
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)

# Basic Twitter crawler
def fetch_tweets(query, count=10):
            tweets = api.search_tweets(q=query, count=count, lang='en')
            for tweet in tweets:
                print(f"Tweet by {tweet.user.name}: {tweet.text}\n")
```

   b. Filter tweets based on content

```
def filter_contents(tweets, keywords):
    filtered_tweets = [tweet for tweet in tweets if any (keyword in tweet.text.lower() for keyword in
  keywords)]
      tweets = api.search_tweets(q=query, count=count, lang='en')
      for tweet in tweets:
          print(f"Tweet by {tweet.user.name}: {tweet.text}\n")
```

2. **Data Storage and ETL Pipeline**
Build an Apache Spart session to continuously extract code from the social media engine (e.g., Twitter). Also set up an Apache Airflow pipeline which would initiate the Spark session on a periodic basis.

```
# Initialize Spark session
spark = SparkSession.builder \
    .appName("TwitterETL") \
    .config("spark.mongodb.output.uri", "mongodb://localhost:27017/twitter_data.tweets") \
    .getOrCreate()


def etl_pipeline():
  tweets = fetch_tweets("marketing trends", count = 10)
  enriched_tweets = [enrich_tweet(tweet) for tweet in tweets]

  # Convert to Spark DataFrame
  tweets_df = spark.createDataFrame(tweets)
  enriched_tweets_df = spark.createDataFrame(enriched_tweets)

  # Save enriched tweets to MongoDB
  load_to_mongo(enriched_tweets_df)

  upload_to_s3(enriched_tweets_df, s3_bucket, s3_key)


etl_pipeline()
```

Store the tweets in raw form using Data Lake like Amazon s3. Also enrich the tweet and convert to a JSON format to be stored in a NoSQL Database like MongoDB. This will be key to preparing ETL pipelines and making the data available for analysis.

```
! pip install boto3 pymongo
import boto3 # library to connect to s3
from pymongo import MongoClient

s3 = boto3.client('s3')
bucket_name = 'twitter-data-bucket'
file_name = 'tweets.json'


# upload JSON to s3 (Upload data in Raw Form)
def upload_to_s3(df, s3_bucket, s3_key):
  s3 = boto3.client("s3")
  tweets_JSON = df.toJSON().collect()
  tweets_data = "\n".join(tweets_json)  # Join JSON strings for S3 upload
  s3.put_object(Bucket=s3_bucket, Key=s3_key, Body=tweets_data)
  print(f"Uploaded to S3 bucket: {s3_bucket}, key: {s3_key}")
```

3. **Perform sentiment analysis & scoring to enrich tweets**
This should be done to improve the data quality and drive transformations to store data in defined schema for further analysis and feeding the same to an NLP Transformer (BERT in this case)

```
from transformers import pipeline

# Sentiment Analysis Pipeline
sentiment_analyzer = pipeline("sentiment-analysis")

def analyze_sentiment(text):
  result = sentiment_analyzer(text)
  return result[0]["label"]

# Example Usage
for tweet in tweets:
  sentiment = analyze_sentiment(tweet["text"])
  print(f"Sentiment for {tweet['text']} is {sentiment}")

#  Scoring System Code with Example
def score_tweet(tweet):
  engagement_score = tweet.retweet_count + tweet.favorite_count
  return engagement_score

# Example Usage
for tweet in tweets:
  score = score_tweet(tweet)
  print(f"Engagement Score for {tweet['text']} is {score}")
```

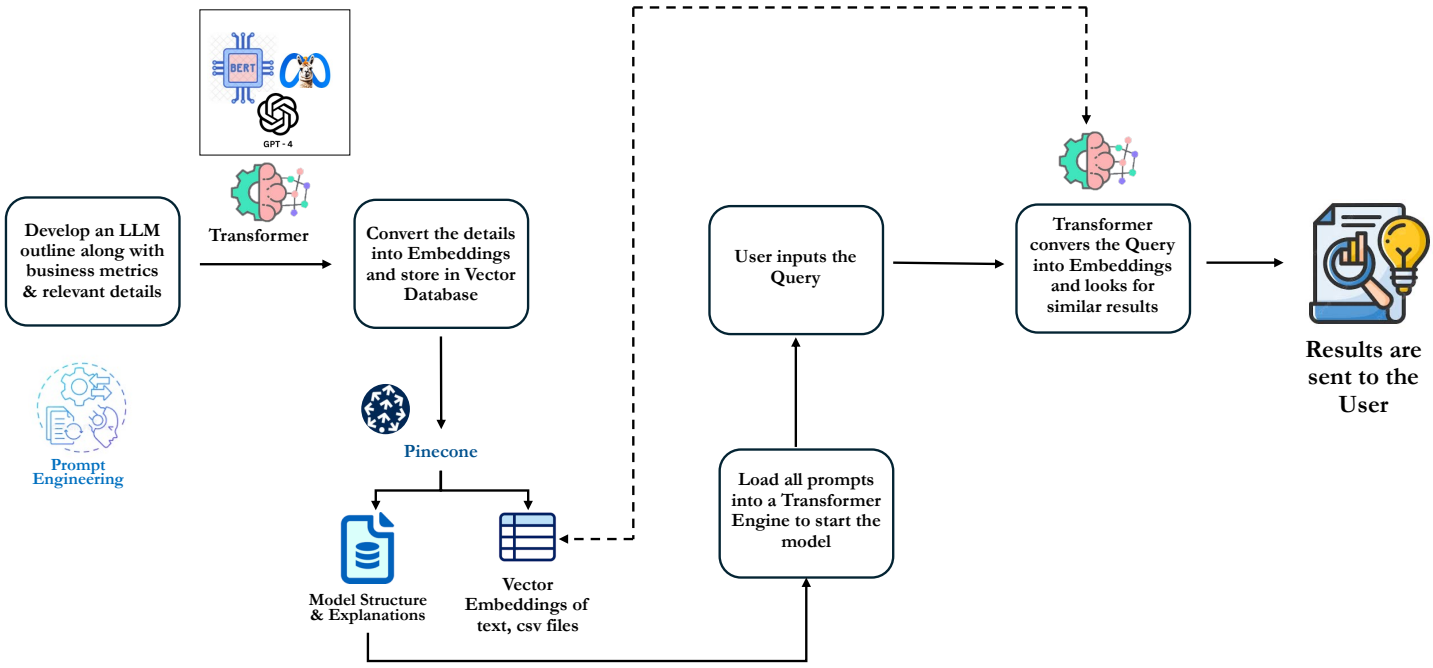
4. **Store enriched tweets in MongoDB which will be used for further analysis**

```
def enrich_tweet(tweet):
  # Add metadata fields
  enriched_tweeet = {
      "text" : tweet.text,
      "user" : tweet.user.name,
      "sentiment" : analyse_sentiment(tweet.text),
      "engagement_score" : score_tweet(tweet)
  }
  return enriched_tweet

def load_to_mongo(df):
  df.write.format("com.mongodb.spark.sql.DefaultSource").mode("append").save()
  print("Data loaded to MongoDB")
```

Section 3 – Semantic Storage & Querying (Building & Deploying the LLM)

1. **LLM Model Development Setup**

We would start by building a Python application locally, where development and testing will be done. Once the app is complete, we will containerize it using Docker, creating a portable image that includes all necessary dependencies. This Docker image will then be pushed to a virtual environment running an app service like Heroku, which simplifies deployment and scaling. Finally, we'll configure environment variables and connection settings in Heroku, allowing the app to connect securely to Snowflake or other external data sources. This approach ensures a seamless and secure deployment pipeline, with the flexibility to scale and update the app as needed.



a. **Connect MongoDB to Snowflake & migrate data into Snowflake**
Snowflake is optimized for data analytics, making it more efficient for running complex SQL queries, aggregations, and transformations.

```
# Connect to MongoDB and retrieve data

client = MongoClient(mongo_uri)
db = client[mongo_db_name]
collection = db[mongo_collection_name]

# Fetch data and convert to dataframe
data = list(collection.find())
df = pd.DataFrame(data)
```

b. **Migrate Data from MongoDB into Snowflake**
Use the Snowflake Connector for Python to connect to Snowflake and load data.

```
# Connect to Snowflake

conn = snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
)


# Write data to Snowflake

def load_data_to_snowflake(df, table_name = "my_table"):

        # Convert pandas DF to Snowflake DF
        snowflake_df = session.write_pandas(df, table_name)

load_data_to_snowflake(df)
```

2. **Develop an LLM by following the below.** This is just one way for creating an LLM and other alternatives can be considered.

    a. **Undertake prompt engineering to supply basic guidelines** to the selected Transformer engine like BERT, GPT-4, Llama (I have used GPT-4 as an example below)

```
def explain_variables_to_gpt():

    """
    Retrieves the stored data from Snowflake, sends it to GPT-4, and gets an explanation back.
    """


    # Fetch variables_explanation, formula_functions, and available_formulas from Snowflake

    variables_explanation, formula_functions, available_formulas = fetch_stored_data()

    # Send the explanation to GPT-4
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are an assistant that understands business terms."},
            {"role": "user", "content": f"Variables Explanation: {variables_explanation}\nFormula
Functions: {formula_functions}\nAvailable Formulas: {available_formulas}"}
        ],
        max_tokens=500
    )



    # Return the response from GPT-4
        return variables_explanation, formula_functions, available_formulas,
        response['choices'][0]['message']['content']



# Example usage

variables_explanation, formula_functions, available_formulas, gpt_response = explain_variables_to_gpt()

print("GPT-4 Response:\n", gpt_response)
```

b. Convert the tweets dataset to embeddings and store as vectors in Pinecone
This should be processed data with all metadata associated with user tweets. This is instrumental for carrying out semantic searches later when user queries need to be answered.

```
# Function to store dataset in Pinecone
def store_dataset_in_pinecone(file):
    """
    Loads dataset from Snowflake, creates embeddings for each row based on relevant columns,
    and stores them in Pinecone.
    """

    # Load the dataset
    user_tweets = pd.read_csv(file)

    # Define relevant columns
    relevant_columns = [    ]

    # Upsert each row to Pinecone
    for idx, row in user_tweets.iterrows():
        # Create a descriptive text for each row based on relevant columns
        row_text = ' '.join([f"{col}: {row[col]}" for col in relevant_columns if pd.notnull(row[col])])

        # Create embedding for the row text
        embedding = openai.Embedding.create(input=row_text, model="text-embedding-ada-
xxxxxxx002")['data'][0]['embedding']

        # Upsert the row to Pinecone with a unique ID (use idx as the ID)
        index_1.upsert([(str(idx), embedding, {'text': row_text})])

    print("Dataset has been stored in Pinecone.")



# Call the function to store the dataset in Pinecone
store_dataset_in_pinecone(tweets_dataset.csv')
```

c. Convert market research documents, sales data to embeddings and store as a separate database in Pinecone
This would need the Transformer to encode PDF's, Text files, CSV's to embeddings.

```
Convert CSV's, EXCEL, DataFrames to Embeddings
from sentence_transformers import SentenceTransformer

# Create or connect to a Pinecone index (We would need separate indexes for each document set)
index_name = "market_research"
if index_name not in pinecone.list_indexes():
    pinecone.create_index(index_name, dimension=768)
index = pinecone.Index(index_name)

# Load data (example with CSV file)
data = pd.read_csv('market_research_data.csv')
documents = data['content'].tolist()  # Extract text content

# Load the BERT model
model = SentenceTransformer('bert-base-nli-mean-tokens')
# Generate embeddings for each document
embeddings = model.encode(documents, convert_to_tensor=False)
```

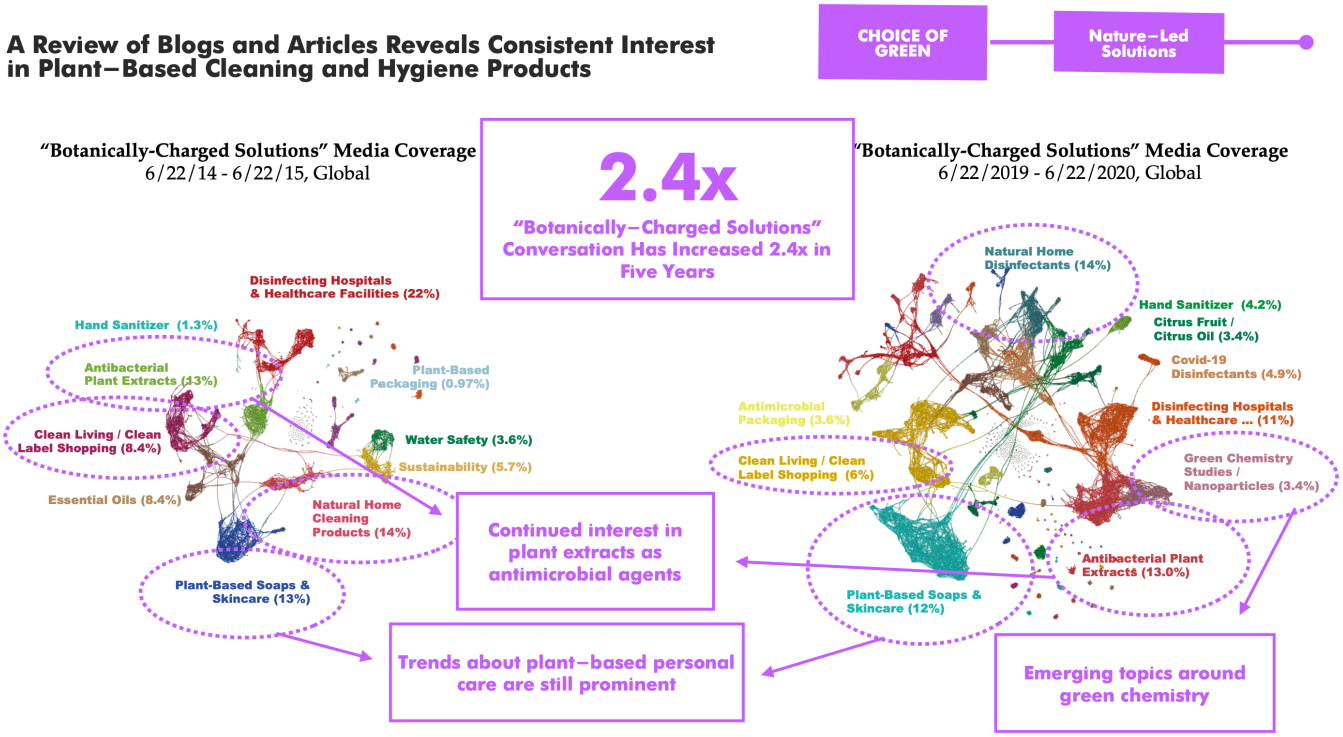
```
Extract Texts from PDF's & Convert to Embeddings
# Encode sentences in chunks (assuming 512 tokens per chunk)
chunk_size = 512
embeddings = []
chunk = []

for sentence in sentences:
    chunk.append(sentence)
    if len(model.tokenizer.tokenize(" ".join(chunk))) > chunk_size:
        # Encode the current chunk and reset
        embeddings.append(model.encode(" ".join(chunk[:-1])))
        chunk = [sentence]  # Start new chunk with the overflow sentence

# Encode any remaining sentences in the last chunk
if chunk:
    embeddings.append(model.encode(" ".join(chunk)))

# Aggregate embeddings (e.g., by averaging)
document_embedding = np.mean(embeddings, axis=0)
```

d. **Consumer Interest Mapping (Tweets)**
We can use **Latent Dirichlet Allocation (LDA)** to identify common themes in tweets related to a brand, product, or market segment

```
from sklearn.decomposition import LatentDirichletAllocation

# LDA for topic modeling
def extract_themes(tweets):

        vectorizer = CountVectorizer(max_df = 0.95, min_df = 2, stop_words = 'english')
        dtm = vectorizer.fit_transform(tweets)

        lda =  LatentDirichletAllocation(n_components = 2, random_state = 0)

        lda.fit(dtm) # Fits the LDA model to the tweets data finding 3 main themes

extract_themes(tweets)


df = tweets[1,2,3,…n]

vectors []

for tweet in tweets:



import spacy

# Load spaCy's pretrained model with NER capabilities

nlp = spacy.load("en_core_web_sm") # spaCy's NER model stored in nlp. This returns a processed doc object
where all identified entities are accessible as doc.ents

# Example tweets
tweets = [
"Just bought the new iPhone! Apple really nailed it with the camera quality.",
"Had an amazing experience at Starbucks! Their new holiday drinks are fantastic.",
"Google's AI advancements are changing the game in the tech industry.",
"Tesla's new update makes driving even more fun. Autopilot is incredible!", "Amazon's delivery speed
keeps getting better every year."
]

# Functions to extract entities & themes
def extract_entities(tweets):
        entity_data = []
        for tweet in tweets:
                doc = nlp(tweet)

                # for each ent it retrieves ent.txt e.g., "Apple" and ent.label_ e.g., ORG for
                Organization
                entities = [(ent.text, ent.label_) for ent in doc.ents]

                entity_data.append("tweet" : tweet, "entities" : entities})

                return entity_data



extract_entities(tweets)


Outputs

[
        {
                "tweet" : "Apple just released a new iPhone in the US and it is awesome and it works very
                fast"
                "entities" : [("Apple", "ORG"), ("iPhone", "PRODUCT"), ("US", "GEOGRAPHY")]
        },

        {
                "tweet": "Tesla's new Autopilot update is amazing!",
                "entities": [("Tesla", "ORG"), ("Autopilot", "PRODUCT")]
        }
]
```

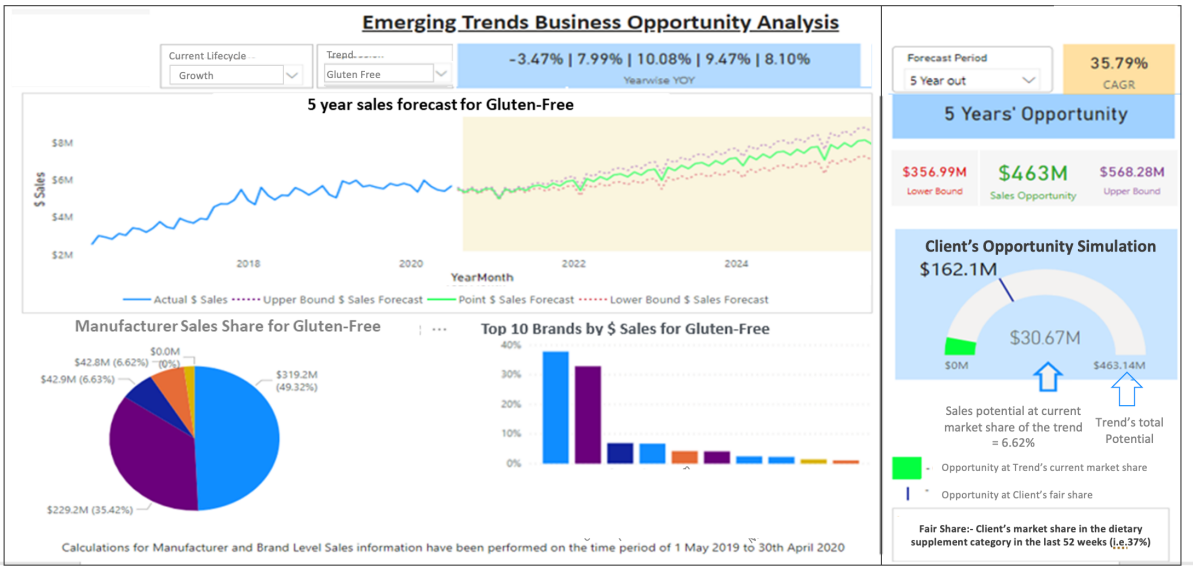
f. Blog and Article Analysis for Topic Heatmaps

Create a heatmap by involving visualizing the intensity of various topics or clusters over time or by category. Please see an example below which could be created by using Python libraries, which will focus on generating a network map with color-coded clusters and labeled regions to represent different topics. We can use libraries like NetworkX for network clustering and Matplotlib for heatmap visualization.



g. Opportunity Sizing Based on Product Sales Data

Using historical sales and growth rates, predict future sales potential and market share. Machine learning models like regression or forecasting algorithms (ARIMA, Prophet)



Section 4 – **Bringing it all together** (Compiling the application & developing a front end)

Create a requirements.txt file

```
flask
pandas
numpy
# Add other dependencies here
```

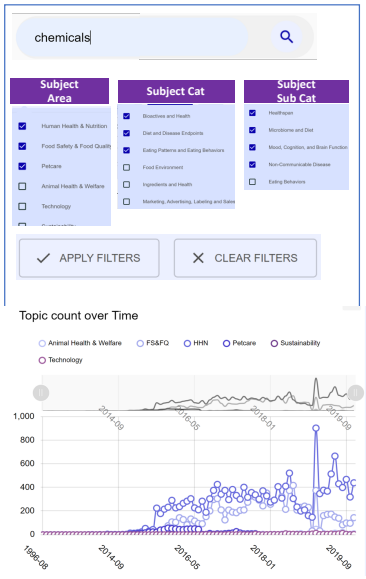
Create a Dockerfile for Containerization

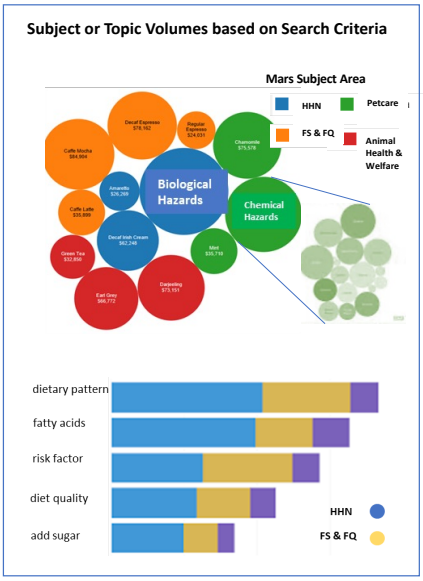
```
# Use a base Python image
FROM python:3.9
# Set the working directory
WORKDIR /app
# Copy the requirements file and install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt
# Copy the rest of the application code
COPY . .
# Expose port 5000 for Flask
EXPOSE 5000
# Define the command to run your application
CMD ["python", "app.py"]
```

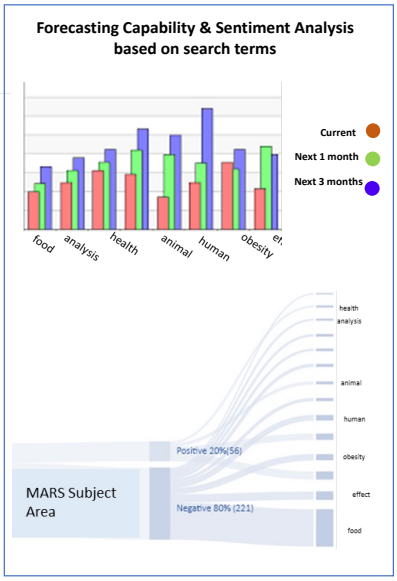
Push the container to Heroku

```
bash

heroku container:push web -a myapp
heroku container:release web -a myapp
```

Create an HTML Frontend







Update Flask Backend to handle requests from Frontend

```
from flask import Flask, request, jsonify, render_template
import json

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json.get("data")
    prediction = "Sample Prediction Output"  # Placeholder
    return jsonify({"prediction": prediction})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Commit & push changes to Heroku

```
bash

git add .
git commit -m "Add HTML front end and
prediction endpoint"
git push heroku master  # or git push
heroku main if using main branch
```