**Malvin Maposa**

**202003513**

**Special Topics – Project**

**Documentation**

**Data Preprocessing**

The Sneaker Review Dataset consists of a collection of 100,000 instances, each containing information on various aspects of sneaker reviews. The dataset is structured with six columns, each serving a specific purpose:

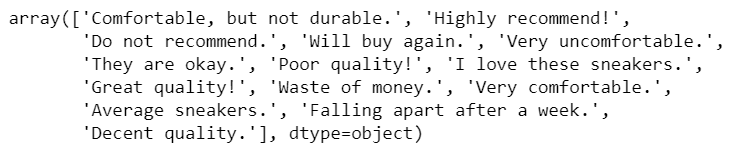
* “review\_id”: A unique identifier for each review
* “product\_id”: The identifier of the sneaker product being reviewed
* “user\_id”: The identifier of the user who provided the review.
* “rating”: The rating assigned to the sneaker by the user, indicating their satisfaction level.
* “review\_text”: The rating assigned to the sneaker by the user, indicating their satisfaction level.
* “timestamp”: The timestamp indicating when the review was submitted.

This dataset is a valuable resource for various data analysis and machine learning tasks, including sentiment analysis, recommendation systems, and understanding customer feedback on sneaker products.

In addition to the dataset description, several tasks have been performed on this dataset, including data cleaning, feature engineering, and model training for sentiment analysis.

Upon a detailed examination of the Sneaker Review Dataset, it has been observed that there are no missing values within the dataset, indicating completeness in data collection.

However, a notable observation pertains to the content of the review\_text column. In this regard, it is important to emphasize that there are only 14 unique words found within this column. This limited vocabulary size could potentially pose challenges when performing sentiment analysis on this dataset. When a model encounters words that are not part of these 14 unique words, it may struggle to accurately predict sentiment, potentially leading to inaccuracies in sentiment analysis.



To enhance the quality of text data in the Sneaker Review Dataset and prepare it for sentiment analysis, several text preprocessing techniques from the Natural Language Toolkit (NLTK) library were applied. These techniques included:

1. Word Tokenization: The text data was tokenized, which involves splitting it into individual words or tokens. This step helps in analyzing text at a more granular level.
2. Stop Words Removal: Common stop words, such as "and," "the," and "in," were removed from the text. Stop words are typically not informative for sentiment analysis and are excluded to focus on more meaningful words.
3. WordNet Lemmatization: Lemmatization was employed to reduce words to their base or root form. This ensures that variations of words are treated as a single entity, reducing the complexity of the vocabulary.

Notably, the word "not" was retained during stop words removal to preserve its impact on sentiment. For example, phrases like "not bad" should be considered neutral sentiments, as the negation "not" reverses the sentiment of "bad."

Subsequently, sentiments were mapped based on the values in the rating column. The mapping was as follows:

* Rating 1 and 2: Mapped to "Negative" sentiment.
* Rating 3: Mapped to "Neutral" sentiment.
* Rating 4 and 5: Mapped to "Positive" sentiment.

This sentiment mapping strategy allows for a straightforward classification of reviews into negative, neutral, or positive sentiments based on the user-provided ratings. The preprocessed and sentiment-mapped dataset is then suitable for training sentiment analysis models to predict sentiment labels for new reviews effectively.

After mapping the statements, we move on to trail our dataset with ‘rating’ and ‘review\_text’ being the x-axis and ‘sentiment’ being the y-axis, we split the dataset into train and test at a 80 to 20 split and vectorized it

Key reasons for choosing logistic regression for sentiment analysis in this project include:

1. Binary Classification: Logistic regression can be adapted for binary classification tasks, such as distinguishing between positive and negative sentiments. In this case, the sentiment labels were mapped to three categories (negative, neutral, and positive), which are compatible with logistic regression.
2. Interpretability: Logistic regression provides interpretable results, making it easier to understand the relationship between input features (text data in this case) and the predicted sentiment labels. This interpretability is valuable for analyzing model predictions and identifying influential words or phrases.
3. Efficiency: Logistic regression models are relatively simple and computationally efficient compared to more complex models like neural networks. They can be trained quickly, which is advantageous when working with large datasets.
4. Effectiveness with Text Data: Logistic regression has been proven effective in handling text data for sentiment analysis tasks. By encoding text features appropriately (e.g., using techniques like TF-IDF or word embeddings), logistic regression can capture the underlying patterns in textual data.

Overall, the choice of logistic regression as the sentiment analysis model is well-founded, considering its suitability for text classification tasks and its balance between model complexity and effectiveness. This choice aligns with the project's goal of predicting sentiment labels for sneaker reviews based on their text content and ratings.

In this project, we developed a sentiment analysis API using the Django web framework. The API serves as an interface for users to input product ratings and reviews, and it provides predictions of sentiment labels (negative, neutral, or positive) based on the provided data. Here is an overview of the steps involved:

1. Project Structure: The project is organized into a Django project named "mlapi" and an app named "api." This separation helps maintain a clean and structured codebase.
2. Frontend Interface: To interact with the API, users access an HTML-based frontend. This frontend allows users to input the product's rating and their review comments.
3. Model Loading: In the backend, we load a pre-trained sentiment analysis model. This model has been trained on a labeled dataset and is capable of predicting sentiment labels based on text data.
4. Text Cleaning: Before making predictions, the review text entered by the user undergoes the same preprocessing steps as the training data. This includes removing unnecessary punctuation and special characters, tokenizing the text, removing stopwords (except for the word "not"), and lemmatizing the words.
5. Vectorization: The cleaned text is then passed through the same vectorizer used during model training. This step converts the text into a numerical format suitable for input into the model.
6. Sentiment Prediction: After vectorization, the API applies the loaded sentiment analysis model to predict the sentiment label. The model assigns one of the following labels: negative, neutral, or positive, based on the input data.
7. API Endpoint: The API exposes an endpoint that accepts POST requests containing the user's product rating and review text. It processes the input, predicts the sentiment, and returns the result to the user.

This sentiment analysis API provides a user-friendly way to assess product sentiment based on customer reviews and ratings. It leverages a pre-trained machine learning model to make accurate predictions and is integrated with Django to offer a robust and scalable solution for sentiment analysis.

**Docker**

The next step in the project was to containerize the Django application using Docker. Dockerization allows for easy packaging of the application and its dependencies into a container, ensuring consistency and portability across different environments. Here are the key steps involved in Dockerizing the Django project:

1. Dockerfile: A Dockerfile was created to define the Docker image for the Django application. This file specifies the base image (Python 3.10), sets environment variables, installs necessary system dependencies (e.g., libpq-dev), and copies project files into the container.
2. Image Building: Docker was used to build an image based on the Dockerfile. This image includes all the dependencies required to run the Django application.
3. docker-compose.yaml: A docker-compose.yaml file was created to define the services and containers needed for the application. In this file, two services were defined: the Django app (app) and a PostgreSQL database (db). It also specified the volumes, ports, and environment variables required for the services.
4. Image Creation: The docker-compose up command was used to create and start the containers based on the services defined in the docker-compose.yaml file.
5. Database Setup: The PostgreSQL database container (db) was used as the database for the Django application. Environment variables for the database name, user, and password were configured in the docker-compose.yaml file.
6. Running the Application: With the containers up and running, the Django application was accessible at localhost:8000 in the local development environment.
7. Data Migration: Django's manage.py commands were used to migrate the database schema and perform other necessary operations.
8. Testing: The Dockerized application was tested locally to ensure that it functions correctly within the containerized environment.

**AWS**

The deployment of the Django API was carried out using Amazon Elastic Container Service (ECS) with Fargate as the compute engine. This approach allows for scalable and efficient deployment of containerized applications. Here are the key steps involved in this deployment:

1. Amazon ECS Cluster: An ECS cluster was created on AWS to manage the deployment of containerized applications. The cluster serves as the resource pool for running containers.
2. Task Definition: A task definition was created in ECS, specifying details about the Docker image to be deployed, container settings, resource requirements, and network configurations. The task definition serves as a blueprint for running containers.
3. Elastic Container Registry (ECR): The Docker image of the Django application, which was previously built and tagged, was pushed to Amazon Elastic Container Registry (ECR). ECR is a managed Docker container registry that allows storing and managing Docker images.
4. Security Groups and IAM Roles: Security groups and IAM roles were configured to define the network and access permissions for the containers. These settings help control inbound and outbound traffic and allow ECS tasks to access other AWS services securely.
5. ECS Service: An ECS service was created to manage the deployment of the Django application. The service uses the task definition and specifies the number of desired tasks (containers) to run. Fargate was chosen as the launch type to abstract the underlying infrastructure.
6. Load Balancer: An Application Load Balancer (ALB) was set up to distribute incoming traffic to the deployed containers. The ALB provides high availability and load balancing for the application.
7. Subnets Configuration: Public and private subnets were configured in the Amazon Virtual Private Cloud (VPC) to ensure network isolation and security. The ALB was placed in the public subnet to receive external traffic, while the ECS tasks were deployed in private subnets.
8. Target Group: A target group was created to route traffic from the ALB to the containers running in the ECS service. It defines the health check settings and port mappings.
9. Listener Rules: Rules were configured in the ALB listener to direct incoming requests to the target group based on specific conditions, such as path patterns.
10. Security Group Rules: Security group rules were defined to control traffic flow to and from the ECS tasks. Inbound rules allowed traffic from the ALB, while outbound rules were restricted to necessary destinations.
11. Continuous Integration and Continuous Deployment (CI/CD): A CI/CD pipeline was set up using GitHub Actions to automate the testing, building, and deployment of the API. This pipeline ensures that code changes are automatically deployed to ECS when pushed to the main branch.