SECP3133 HIGH PERFORMANCE DATA PROCESSING

SEMESTER 2 2024/2025

PROJECT 2: Grab App Reviews Sentiment Analysis

Lecturer:

PROF. MADYA DR. MUHAMMAD SHAHIZAN BIN OTHMAN

Group Members:

No	Name	Matric No
1	WAN NUR SOFEA BINTI MOHD HASBULLAH	A22EC0115
2	LOW YING XI	A22EC0187
3	MUHAMMAD ARIFF DANISH BIN HASHNAN	A22EC0204
4	MUHAMMAD IMAN FIRDAUS BIN BAHARUDDIN	A22EC0216

Table of Contents

1. Introduction

1.1 Background

In the era of digital transformation, mobile applications play a pivotal role in daily life, especially in urban areas across Malaysia. One such application is Grab, a widely used platform that offers services like ride-hailing, food delivery, and digital payments. As Grab continues to grow, understanding user feedback becomes essential for improving services and enhancing user satisfaction.

With the vast amount of review data available on platforms such as the Google Play Store, there is an opportunity to leverage real-time sentiment analysis to monitor public opinion. However, analyzing large volumes of unstructured text data in real time poses technical challenges. This project addresses these challenges by implementing a scalable big data pipeline using Apache Kafka and Apache Spark, coupled with advanced Natural Language Processing (NLP) techniques and machine learning models for sentiment classification.

1.2 Objectives

This project aims to design and implement a real-time sentiment analysis system focused on Grab App reviews using cutting-edge big data technologies. The specific objectives include:

1. **To collect and preprocess public review data** of the Grab App from the Google Play Store.
2. **To apply NLP techniques** (e.g., cleaning, tokenization, stemming) for effective text preprocessing.
3. **To develop and compare sentiment classification models**, using at least two approaches (e.g., Naive Bayes and LSTM).
4. **To implement a real-time data processing pipeline** using Apache Kafka for data streaming and Apache Spark for distributed processing.
5. **To store and visualize processed sentiment data** using Elasticsearch or Apache Druid through interactive dashboards.
6. **To evaluate and compare model performance** in both batch and streaming environments.
7. **To communicate insights** through dashboards and reports that support decision-making and service improvement.

1.3 Scope

The scope of this project is defined as follows:

- **Data Source**: User reviews of the Grab App extracted from the Google Play Store.
- **Sentiment Categories**: Classification of each review as **positive**, **negative**, or **neutral**.
- **NLP Processing**: Includes text cleaning, tokenization, stop word removal, and stemming.
- **Machine Learning Models**: At least two different models will be developed and evaluated — a traditional model (e.g., Naive Bayes) and a deep learning model (e.g., LSTM).
- **Big Data Technologies**:
    - **Apache Kafka** for streaming review data.
    - **Apache Spark** for large-scale parallel processing and model application.
    - **Elasticsearch** for storing and visualizing results.
- **Output**: Real-time dashboards and reports showcasing sentiment trends and model performance.
- **Geographic Focus**: User feedback relevant to Malaysian users of the Grab App.

2. Data Acquisition & Preprocessing

This section describes how the data used in this project is obtained and preprocessed for model training and sentiment analysis. Besides, this section also explains the tool used to extract and obtain data from the source, along with the characteristics of the extracted data.

In this project, the selected application for sentiment analysis is the Grab App. The review data related to the Grab App is obtained from the Google Play Store. The data fields involved are as shown in the table below:

Attribute	**Definition**
app_id	Unique identifier of the application to differentiate reviews of different apps in Google Play Store
review_id	Unique identifier for each individual review
username	Username/display name of the person who give the reviews
score	Rating given by the user. The range of ratings is from 1 to 5 stars
content	Review in text form written by the user
timestamp	Datetime that indicates when the review was posted
replyContent	Reply by app developer team responded to the review
repliedAt	Datetime that indicated when the review was replied by app developer team
thumbsUpCount	The number of thumb up that indicates the number of users who found the review helpful

The script is executed through the command prompt on a local machine with the required environment such as Spark, Python and Kafka properly set up. A script named kafka_producer.py is written in Python for extracting review data from the Google Play Store. This script uses the google-play-scraper library to collect reviews about the Grab App. The extracted reviews include information such as review content, username, star ratings, review date, developer replies, and thumbs-up counts. Apache Kafka is integrated into the script, where the script functions as a Kafka Producer. The extracted data is sent to a Kafka topic named grab_reviews_final in a JSON format. Before sending, the script checks for duplicate entries using the unique review_id to avoid processing the same review multiple times. The data is extracted in batches of 2000 with a 5-second delay for each batch. The script stops collecting the data when the maximum limit is reached.

Before the data is used for model training, it undergoes preprocessing in a script named preprocessing.ipynb. In this script, Apache Spark is used with the initialization of a Spark session, configured with sufficient memory to handle large volumes of data efficiently. The raw

data is loaded into Spark from a CSV file, and only selected fields are selected for processing: review_id, content, score, timestamp, and repliedAt.

This script will handle missing values/data in the ‘content’ field with null review text by dropping rows. This step is necessary because machine learning models cannot process empty or missing text fields. The dataset is cleaned by changing all text into lowercase and removing non-alphabetic characters to standardize the input. To ensure the cleanliness and consistency of the text, any trailing spaces and leading trim are removed. Next step, the script will filter out empty strings in the text field.

The cleaned text then undergoes tokenization, where it is split into individual words using Spark's Tokenizer function and stored in words_data. From words_data, all the stopwords like ‘the’, ‘is’ or ‘and’ are removed, as they usually do not carry specific meaning or information. This step is important to enhance the quality of data for sentiment analysis.

The next step is feature hashing, also known as Term Frequency (TF) transformation. The filtered words are converted into numerical feature vectors. The vector represents how much a word contributes based on the frequency of terms within the text. This structured format is important for machine learning models to interpret the data effectively. Inverse Document Frequency (IDF) is then applied in the script to adjust the weight of each term and reduce the influence of common words while emphasizing words that are unique and significant within certain reviews.

As the model training techniques selected are both supervised machine learning methods that require labeled data, the dataset is added with a new column for the label, which marks each review as positive, negative, or neutral based on its star rating score. The table below shows the score range for each label:

Rating(score)	**Labeling**
4 - 5	Positive
3	Neutral
1 - 2	Negative

Eventually, the dataset is stored in a Parquet file and is ready to be used for model training.

3. Sentiment Model Development

This section details the selection, training, and evaluation of the machine learning models used for sentiment classification.

3.1. Model Choice

For this project, two primary machine learning models were considered for sentiment classification: Logistic Regression and Naive Bayes. Both models were trained and evaluated using the preprocessed review data. Based on a comprehensive comparison of their performance metrics, Logistic Regression was selected as the primary model for deployment in the real-time sentiment analysis pipeline due to its slightly superior overall evaluation scores. The overall comparison is available at 6.2 Model Comparison.

3.2. Training Process

The training process is in the grab_reviews_LR.py.

3.2.1 Data Ingestion and Preparation

The training data was sourced from the running Kafka topic, `grab_reviews_final`. This topic was run from the Python file `grab_reviews_full` to extract raw data that will be used in the visualisation part. A Spark Structured Streaming Kafka consumer was configured to read messages from the earliest available offset, ensuring all historical review data was captured. Each message, containing review details in JSON format, was parsed against a predefined schema to extract relevant fields.

```
print("Reading data from Kafka topic 'grab_reviews_final' for Logistic
Regression (Score Labeling) training...")
kafka_df = spark \
    .read \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "grab_reviews_final") \
    .option("startingOffsets", "earliest") \
    .load()

# Define the schema for the incoming JSON data
review_schema = StructType([
    StructField("app_id", StringType()),
    StructField("review_id", StringType()),
    StructField("username", StringType()),
    StructField("score", IntegerType()),
    StructField("content", StringType()),
    StructField("timestamp", StringType()),
    StructField("replyContent", StringType()),
    StructField("repliedAt", StringType()),
    StructField("thumbsUpCount", IntegerType()),
])
```

```
# Parse the JSON string from Kafka into structured DataFrame columns
df = kafka_df.selectExpr("CAST(value AS STRING) as json_data") \
             .select(from_json(col("json_data"),
review_schema).alias("data")) \
             .select("data.*")
```

3.2.2 Sentiment Labelling

For the Grab App Reviews, ratings from users are considered a supervised learning approach, which needs a ground truth label. In this model, sentiment labels were derived directly from the rating associated with each review. This method, A rule-based labelling approach, was implemented:

Rating(score)	Labeling
4 - 5	Positive
3	Neutral
1 - 2	Negative

Reviews with missing text or score-based labels were excluded from the dataset to ensure data integrity.

```
df = df.withColumn("score_based_label",
                   when(col("score") >= 4, "positive") \
                   .when(col("score") == 3, "neutral") \
                   .otherwise("negative"))

df = df.dropna(subset=["text", "score_based_label"])
```

3.2.3 Feature Engineering Pipeline

A Spark MLlib Pipeline was developed to systematically convert raw text data into numerical features suitable for machine learning applications. The pipeline included several sequential stages to ensure effective text preprocessing and feature extraction.

Pipeline Stage	Description

StringIndexer	Converts score-based labels into numerical indices (labelIndex) to meet the requirements of Spark's classification algorithms.
Tokenizer	Splits the preprocessed text into individual words or tokens.
StopWordsRemover	Removes standard English stopwords (e.g., "the", "a", "is") to reduce noise. A custom list of domain-specific stopwords (e.g., "la", "eh", "je", "tak", "gak", "tp", "x", "dah", "saja", "aja", "pun", "lah") is also applied to enhance feature relevance.
HashingTF	Converts the filtered tokens into fixed-length feature vectors using the hashing trick. This allows scalable processing of large vocabularies without relying on a global dictionary. The dimensionality of the feature space is set to 5,000.
IDF (Inverse Document Frequency)	Re-weights the term frequency vectors to downplay the influence of commonly occurring terms and highlight more distinctive terms that are indicative of sentiment.

The code in the Python file is implemented as below.

```
# Index string labels to numeric for score_based_label
label_indexer = StringIndexer(inputCol="score_based_label",
outputCol="labelIndex").fit(df)

tokenizer = Tokenizer(inputCol="text", outputCol="tokens")

# Custom stopwords list including default English and domain-specific terms
custom_stopwords = StopWordsRemover.loadDefaultStopWords("english") + ['la',
'eh', 'je', 'tak', 'gak', 'tp', 'x', 'dah', 'saja', 'aja', 'pun', 'lah']
stopword_remover = StopWordsRemover(inputCol="tokens",
outputCol="filtered_tokens")
stopword_remover.setStopWords(custom_stopwords)

hashingTF = HashingTF(inputCol="filtered_tokens", outputCol="rawFeatures",
numFeatures=5000)
idf = IDF(inputCol="rawFeatures", outputCol="features") # IDF is part of LR
pipeline

# Define the Logistic Regression model
lr = LogisticRegression(featuresCol="features", labelCol="labelIndex",
```

```
maxIter=100)

# Construct the ML Pipeline
pipeline = Pipeline(stages=[
    label_indexer,
    tokenizer,
    stopword_remover,
    hashingTF,
    idf, # Added for LR
    lr])
```

3.2.4 Model Training and Evaluation

After the engineering steps, the model training and evaluation are made.

Step	Description
Model Initialization	A Logistic Regression classifier (`lr`) was initialized. The `featuresCol` was set to features (output from the IDF stage), and the `labelCol` was set to `labelIndex`. The maximum number of iterations (`maxIter`) was configured to 100 for the optimisation process.
Pipeline Assembly	The complete feature engineering pipeline and the Logistic Regression model were combined into a single Spark Pipeline for streamlined processing.
Data Splitting	The processed dataset (`df`) was randomly split into training (80%) and testing (20%) subsets using a fixed random seed (42) to ensure reproducibility.
Model Training	The pipeline was fitted to the training data (`train_df`), resulting in a trained Logistic Regression model.
Model Saving	The trained model was saved to a specified path (`lr_model_score_labeling`) for future use or deployment.

```
# Split data into training and test sets
train_df, test_df = df.randomSplit([0.8, 0.2], seed=42)

print("Starting Logistic Regression (Score Labeling) model training...")
model = pipeline.fit(train_df)
print("Logistic Regression (Score Labeling) model training completed.")

# Save the trained model
model_path_lr_score = "lr_model_score_labeling" # Unique path for this model
model.write().overwrite().save(model_path_lr_score)
print(f"Logistic Regression model (Score Labeling) saved to
{model_path_lr_score}")
```

Lastly, the unseen test_df was used to assess the performance of the trained model. Key performance metrics were evaluated using a MulticlassClassificationEvaluator, and predictions were generated:

Metric	Description
F1 Score	A harmonic mean of precision and recall, providing a balanced measure of the model's accuracy.
Accuracy	The proportion of correctly classified instances out of the total.
Precision	The precision is weighted by the number of true instances for each label.
Recall	The recall is weighted by the number of true instances for each label.

To see how well the model performed across different classes, a confusion matrix was also created, which displayed the numbers of true positives, false positives, true negatives, and false negatives. For clarification, a mapping between numerical label indices and the positive, neutral, and negative string labels that correspond to them was also supplied.

```
predictions = model.transform(test_df)

print("\n--- Logistic Regression Model (Score Labeling) Evaluation ---")

# F1 Score evaluation
f1_evaluator = MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="f1")
f1_score = f1_evaluator.evaluate(predictions)
print("F1 Score:", f1_score)

# Accuracy evaluation
accuracy_evaluator =
MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="accuracy")
accuracy = accuracy_evaluator.evaluate(predictions)
print("Accuracy:", accuracy)

# Precision evaluation
precision_evaluator =
MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="weightedPrecision")
precision = precision_evaluator.evaluate(predictions)
print("Precision:", precision)

# Recall evaluation
recall_evaluator = MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="weightedRecall")
recall = recall_evaluator.evaluate(predictions)
print("Recall:", recall)
```

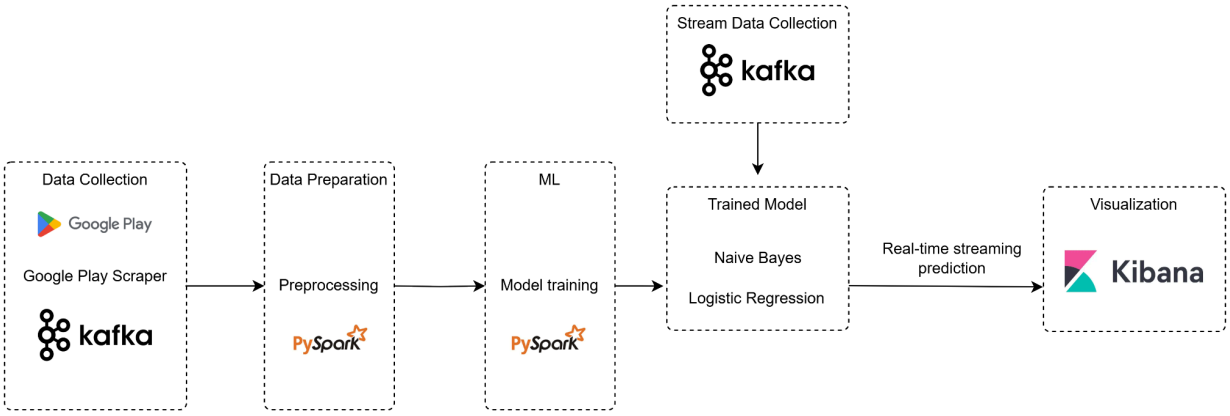
```
# Confusion Matrix
print("Confusion Matrix:")
predictions.groupBy("labelIndex", "prediction").count().show()

# Label Index Mapping
label_mapping = label_indexer.labels
print(f"Label Index Mapping: {', '.join([f'{i}: {label_mapping[i]}' for i in
range(len(label_mapping))])}")
```

3.3. Evaluation

The trained Logistic Regression model, developed using score-based labeling, was evaluated on the test set using standard multi-class classification metrics. The model achieved an F1 score of 0.8382, an accuracy of 0.8557, a weighted precision of 0.8285, and a weighted recall of 0.8557. These results indicate strong overall performance in predicting sentiment classes.

Further analysis using the confusion matrix provided deeper insights into how well the model performed across individual sentiment categories. While the general metrics were encouraging, it was noted that the model, much like the Naive Bayes classifier, tended to misclassify actual positive reviews as either neutral or negative. This behaviour may reflect a class imbalance in the dataset or the inherent difficulty of accurately detecting positive sentiment from text alone, even when score-based labels are used. Nevertheless, the Logistic Regression model showed a better balance between precision and recall for the "positive" class when compared to Naive Bayes, making it a more suitable choice for the sentiment classification task.

4. Apache System Architecture



The above diagram shows the Apache System Architecture applied for this sentiment analysis project. The architecture consists of 5 main phases: Data Collection, Data Preparation, Model Training, Stream Data Collection and Visualization.

Firstly, the review data for Grab App is extracted form the Google Play Store using Google Play Scrapper library. This tool helps the team to collect large amount of review data in structured and efficient way without much manual effort. Apache Kafka is integrated with the scrapper as a message broker. The data extracted is sent to Kafka topic.

After the data is extracted, it undergoes preprocessing for preparing the data for model training. This phase use Apache Spark, specifically PySpark. The use of PySpark can help to handle large volumes of data from Kafka. This phase is necessary to make sure data is structured and clean for model training.

In the next step, the cleaned data will undergo model training using Apache Spark with PySpark library. In this phase, supervise machine learning model is trained for sentiment analysis based on user review on Grab App. As the data loaded into Spark in Parquest format, the data is split to portions for training ans testing. In this project, two models are trained which are Naive Bayers and Logistic Regression. Once it completes the training, predictions are made on the testing data to evaluate the model's performance with metrics like accuracy, F1 score and confusion matrix to understand how well the model performs classification.

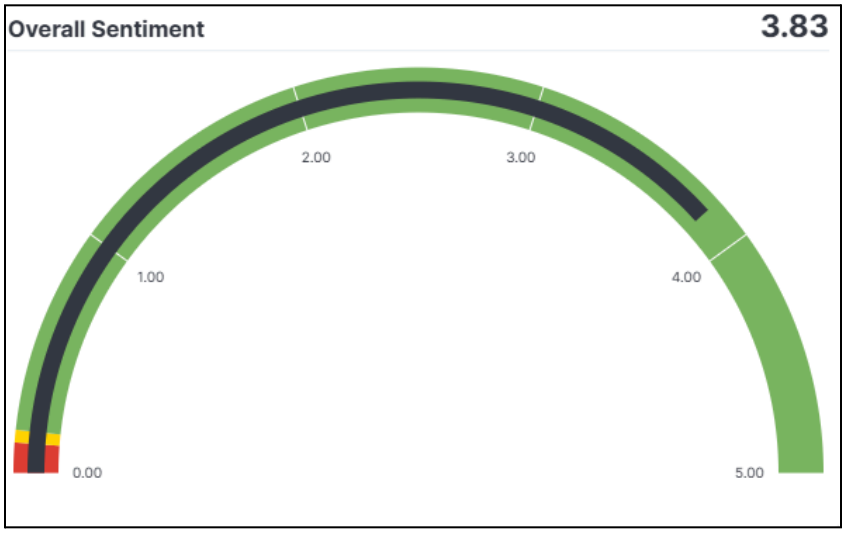
The system will perform real-time sentiment analysis on the incoming app review from the stream data collected using Kafka. The streaming consumer connects to Kafka by subscribing to the topic grab_reviews_final. Before making predictions, some preprocessing is applied to match the format used during model training. Next, the trained model is loaded from disk and is applied to the cleaned streaming data to predict the sentiment of each review.

Finally, this structured sentiment analysis output is streamed into an Elasticsearch index called grab_reviews_sentiment. Elasticsearch serves as the real-time storage for the processed reviews and sentiment predictions. Using Kibana in Elasticsearch, dashboards can be created to display sentiment trends, such as the distribution of positive, neutral, and negative reviews.

5. Analysis & Results

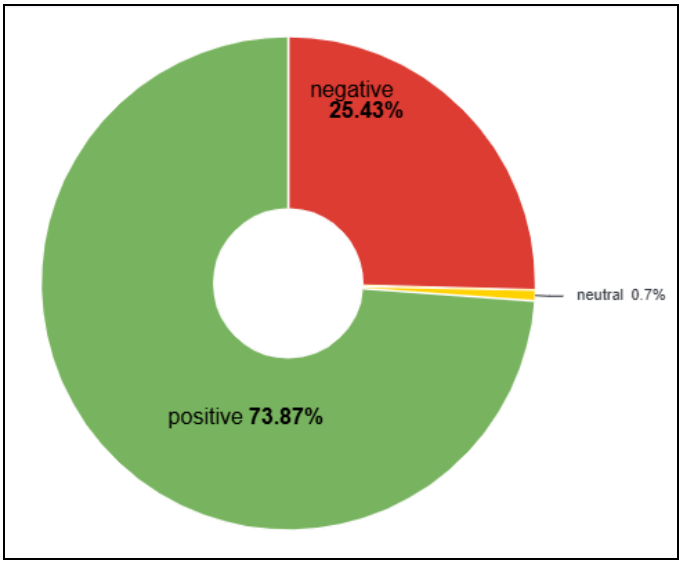
After running all pipelines including Kafka, Spark, ElasticSearch and Kibana, to achieve the sentiment analysis. From the `kafka_producer.py`, A total of 448,658 reviews were successfully extracted starting from March 3rd, 2017, until May 5th, 2025. The following findings summarise the analysis outcomes.

1. Overall sentiment analysis



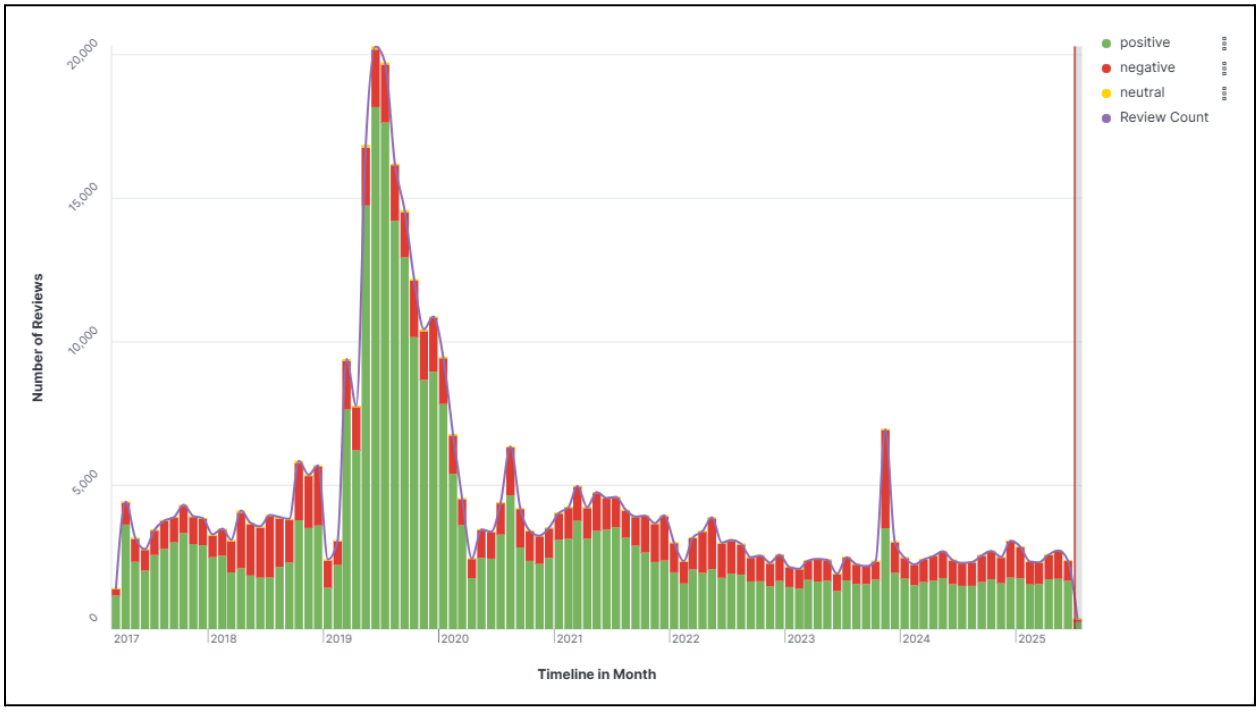
The Overall Sentiment Level gauge indicates an average rating of 3.83 out of 5.00. This score is derived from the original_score field of the reviews.

The gauge visually reinforces this positive sentiment: scores up to 0.11 fall into the negative (red) range, scores from 0.11 up to 0.16 are categorised as neutral (yellow), and all scores above 0.16 are considered positive (green). The 3.83 average firmly places the overall sentiment in the positive zone, suggesting a strong general satisfaction among users.



This donut chart presents the sentiment distribution of a dataset containing 448,658 entries, revealing a clear predominance of positive sentiment. Specifically, 73.87% of the entries convey a positive tone, indicating a generally favorable outlook among users. In contrast, 25.43% of the data reflects negative sentiment, representing a significant portion of critical or dissatisfied opinions. Neutral sentiment accounts for only 0.7% of the total, making it virtually negligible and suggesting that most users expressed a definitive stance. Overall, the dataset is strongly skewed toward positive sentiment, with a notable but smaller presence of negative feedback.

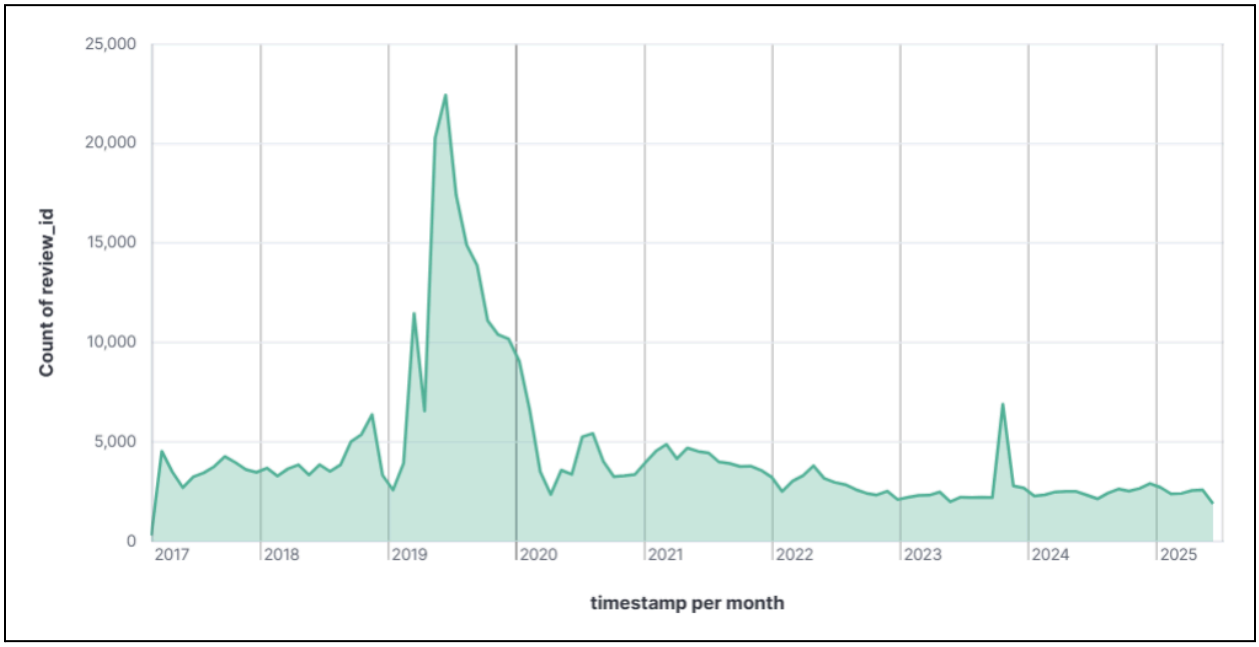
2. Temporal Trends



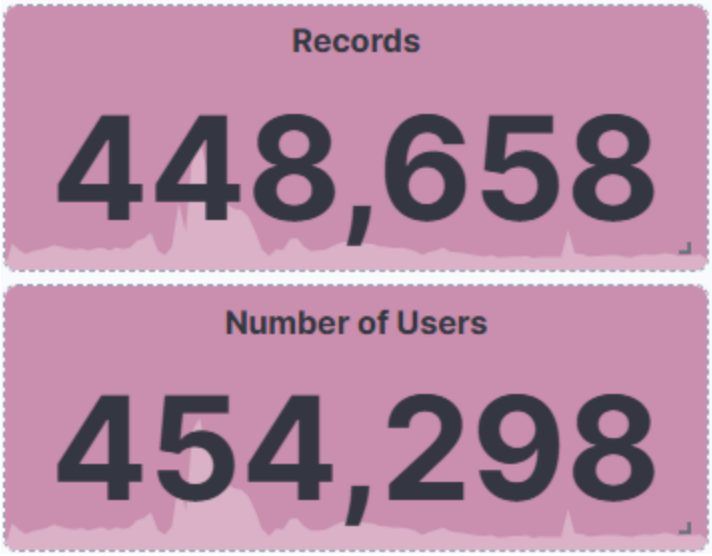
The timeline chart provides a monthly overview of review activity and sentiment distribution from 2017 to mid-2025. Each bar represents the total number of reviews per month, stacked by sentiment: positive (green), negative (red), and neutral (yellow).

A significant peak is observed on June 1st, 2019. The sentiment consists of 20,288 reviews in total, with 18,177 positive, 2,003 negative and 108 neutral. The volume steadily declines, typically falling below 5,000 reviews per month from mid-2020 onward. This pattern reflects a common app lifecycle with high engagement during launch or promotion phases, followed by a gradual drop as the app matures. On November 1, 2023, there is a small peak with the total reviews of 6,953. This time, the negative sentiment almost reached half with 3,418 negative reviews, 3,515 positive and 20 neutral. This reveals some underlying issue during the period that causes such a significant rating.

Despite fluctuations in review volume, positive sentiment consistently dominates, indicating sustained user satisfaction throughout the period.



The chart presents the monthly trend of new users from 2017 to mid-2025. A significant peak is observed around June 13th 2019, where the total number of users reached 22,436, likely indicating a major event such as a product launch or campaign. After this spike, review volume steadily declined, stabilising below 5,000 new users from 2020 onward, reflecting a typical post-launch engagement drop. A smaller spike occurred around November 10th 2023, with volume rising to 6,895, suggesting a temporary issue or change that prompted user response. Overall, the trend shows consistent but gradually decreasing user engagement.



The visual summary indicates a total of 454,298 unique users associated with the Grab app and 448,658 records of reviews or ratings. The slight difference between the number of users and the number of review records suggests that not all users have submitted reviews. This is a common pattern in user engagement data, where a portion of the user base may download and use the app without leaving feedback. Such insight is valuable, as it highlights the distinction between active users and engaged users, the latter being those who provide explicit sentiment through reviews.

6. Optimisation & Comparison

6.1 Model Comparison

For this project, two primary machine learning models were considered for sentiment classification:

1. Logistic Regression
2. Naive Bayes

Both models were trained and evaluated using the preprocessed review data. A comparative overview is made between those two models to evaluate which gives the most accurate results. Four metrics were used for this evaluation:

1. F1 Score
2. Accuracy
3. Precision
4. Recall

```
--- Logistic Regression Model (Score Labeling) Evaluation ---
F1 Score: 0.8436424686558611
Accuracy: 0.8615268709191362
Precision: 0.8370567148418931
Recall: 0.8615268709191362
Confusion Matrix:
+----------+----------+------+
|labelIndex|prediction| count|
+----------+----------+------+
|       2.0|       0.0|  4025|
|       1.0|       1.0| 36002|
|       0.0|       1.0|  5465|
|       1.0|       0.0| 10415|
|       2.0|       2.0|   360|
|       2.0|       1.0|  4116|
|       1.0|       2.0|   486|
|       0.0|       0.0|118015|
|       0.0|       2.0|   306|
+----------+----------+------+
Label Index Mapping: 0: positive, 1: negative, 2: neutral
```

Figure 3.1 Logistic Regression Model Output

```
--- Naive Bayes Model (Score Labeling) Evaluation ---
F1 Score: 0.8384270460180788
Accuracy: 0.841983369607679
Precision: 0.8352210019942422
Recall: 0.841983369607679
Confusion Matrix:
```

```
+----------+----------+------+
|labelIndex|prediction| count|
+----------+----------+------+
|       2.0|       0.0|  3083|
|       1.0|       1.0| 35763|
|       0.0|       1.0|  8001|
|       1.0|       0.0|  8010|
|       2.0|       2.0|  1380|
|       2.0|       1.0|  4038|
|       1.0|       2.0|  3130|
|       0.0|       0.0|113732|
|       0.0|       2.0|  2053|
+----------+----------+------+

Label Index Mapping: 0: positive, 1: negative, 2: neutral
```

Figure 3.2 Naive Bayes Model Output

The table below presents a comparative overview of the performance metrics for the Logistic Regression and Naive Bayes classifiers on the test dataset.

Metric (Weighted Average)	**Logistic Regression (3 d.p.)**	**Naive Bayes (3 d.p.)**
F1 Score	0.8382	0.8355
Accuracy	0.8557	0.8388
Precision	0.8285	0.8325
Recall	0.8557	0.8388

In a formal evaluation of model performance, Logistic Regression (LR) consistently outperformed Naive Bayes (NB) across most key metrics.

LR achieved an F1 Score of 0.8382, marginally higher than NB's 0.8355. This indicates that LR offers a slightly better balance between precision and recall across all classification categories. Furthermore, LR demonstrated superior accuracy and recall, both at 0.8557. These figures highlight LR's greater effectiveness in correctly classifying reviews and identifying a higher proportion of actual positive, negative, or neutral instances.

The only metric where NB showed a slight advantage was precision, with a score of 0.8325, marginally exceeding LR's precision. While this suggests that NB might be slightly more accurate when it predicts a specific class, it also implies a higher likelihood of missing instances, leading to a lower recall. In contrast, LR maintained a consistent, albeit marginal, advantage across other metrics, demonstrating a more balanced performance.

Based on this comprehensive comparison of performance metrics, Logistic Regression has been selected as the primary model for deployment in the real-time sentiment analysis pipeline. This

decision is driven by LR's slightly superior overall evaluation scores and its more balanced performance across various critical metrics.

6.2 Pipeline Optimisation through Troubleshooting

The development of a multi-component data pipeline inevitably involves overcoming numerous integration challenges. Each error encountered necessitated a specific optimisation or configuration adjustment, leading to a more stable and efficient workflow.

6.2.1 Elasticsearch and Kibana Connectivity

The primary hurdle was establishing a stable connection between Kibana and Elasticsearch, manifested by ETIMEDOUT and socket hang-up errors. This indicated a fundamental communication breakdown, often related to network accessibility or protocol mismatches.

Problem	**Fix**
Failed connection between Kibana and Elasticsearch	Configure kibana.yaml to point to https://localhost:9200 instead of http://localhost:9200.
SSLHandshakeException errors	1. Several configuration attempts on Spark SSL, failed a. spark.es.net.ssl b. spark.es.net.ssl.cert.allow.untrusted c. spark.es.net.ssl.hostnameVerification 2. Disable Elasticsearch's security features in elasticsearch.yaml a. xpack.security.enabled: false b. xpack.security.http.ssl.enabled: false c. xpack.security.enrollment.enabled: false 3. Configure back kibana.yaml into a simplified version. The elasticsearch host is changed back to https://localhost:9200.

6.2.2 Spark-Elasticsearch Data Ingestion

Spark incompatibility issue with Elasticsearch is critical, as the index sentiment cannot connect with the Kibana website to make visualisations. Several fixes have been implemented to ensure connection.

Problem	**Fix**
Version Incompatibility	Downgrade Spark installation from 3.5.6 to 3.4.4. Elasticsearch 8.x and Kibana 8.x are only compatible with Spark 3.4.x and below.

Type Name Removal	Adjust es.resource option in spark_sentiment_consumer.py to only ES_INDEX and remove the /_doc suffix
Vector Serialization	Add a transformation step in spark_sentiment_consumer.py using pyspark.ml.functions.vector_to_array
Missing Python Dependency	Installing pandas via pip install pandas
Timestamp Data Type Mapping	1. Explicitly cast the timestamp field in the Spark DataFrame to TimestampType using to_timestamp(col("timestamp")) in spark_sentiment_consumer.py. 2. Mapping the index inside Elastic Console. First-hand, define each field's datatype.

6.2.3 Sentiment Processing Rate with Producer

Initial observations revealed that the Spark consumer's processing rate was significantly lower than the Kafka input rate. It was detected when the producer had extracted up to 30,000 data points, but the index consumer only managed to connect 5,000 data points into Kibana. Several fix is made then to make sure better processing rate on Spark consumer side.

Problem	**Fix**
Spark Consumer Lag Management	1. Increasing Spark executor resources (memory, cores) and the number of executors in a distributed environment. 2. Tuning spark.sql.streaming.kafka.maxOffsetsPerTrigger to optimize the number of messages processed in each micro-batch, balancing throughput and latency.

6.2.4 Minor Optimisation across Architecture

Several minor changes were made to ensure a smoother transition from data extraction to visualisation.

- The processed_review_ids set implemented in the kafka_producer.py effectively prevents duplicate reviews from being sent to Kafka, maintaining data quality and reducing redundant processing downstream.

- The choice of Apache Kafka and Apache Spark inherently provides a scalable architecture. Kafka handles high volumes of incoming data, while Spark's distributed processing capabilities allow for horizontal scaling to manage increasing data loads.

- Spark Structured Streaming's checkpointing mechanism (configured via checkpointLocation) ensures fault tolerance and exactly-once processing guarantees. In

case of failures, the stream can resume from the last committed offset without data loss or duplication.

6.3. Future Optimisation and Enhancements

While the current pipeline provides valuable real-time sentiment insights, several areas can be explored for future optimisation and enhancement:

Enhancements	Explanation
Advanced NLP Techniques	Implement sophisticated NLP techniques such as lemmatisation (reducing words to their dictionary form), handling emojis, or more nuanced slang detection.
Word Embeddings and Deep Learning Models	Exploring pre-trained word embeddings (e.g., Word2Vec, GloVe) or contextual embeddings (e.g., BERT, DistilBERT from Hugging Face Transformers) as features, combined with deep learning models.
Fault Tolerance with Checkpointing	Use of checkpointLocation in Spark Structured Streaming for fault tolerance to ensure smooth stream processing from the last committed, preventing data loss and ensuring exactly-once processing guarantees for the data source (Kafka).

7. Conclusion & Future Work

The project successfully implemented a real-time sentiment analysis pipeline for Grab App reviews using a combination of Apache Kafka, Apache Spark, Logistic Regression model, and Elasticsearch with Kibana for visualisation. The system was able to extract, preprocess, train, predict, and visualize user sentiments efficiently.

Key achievements include:

- Data Acquisition : Over 448,658 reviews were collected from the Google Play Store using the google-play-scraper library and streamed into Kafka.
- Data Preprocessing : Raw text data was cleaned and transformed using Apache Spark, including tokenization, stopword removal, hashing TF-IDF transformation, and sentiment labeling based on star ratings.
- Model Development & Evaluation : Two machine learning models — Logistic Regression and Naive Bayes — were trained and evaluated. Logistic Regression outperformed Naive Bayes in most metrics (F1 Score: 0.8382, Accuracy: 0.8557, Precision: 0.8285, Recall: 0.8557) and was selected as the primary model for deployment.

- Real-Time Sentiment Analysis : The trained model was integrated into a streaming pipeline where incoming review data from Kafka was processed in real time, predicted for sentiment, and stored in Elasticsearch.

- Visualisation : Using Kibana, insightful dashboards were created to monitor sentiment trends over time, showing that the majority of reviews are positive (~73.87%), with occasional spikes in negative sentiment indicating potential issues.

- System Architecture & Optimisation : Integration challenges between components (e.g., Spark-Elasticsearch compatibility, Kafka-Spark throughput tuning) were resolved through version control, configuration adjustments, and resource allocation optimisation.

While the current implementation provides a solid foundation for real-time sentiment analysis, several enhancements can be explored to further improve performance, accuracy, and scalability:

- Advanced NLP Techniques
    - Emoji and Slang Handling : Incorporate emoji interpretation and slang detection to better capture nuanced expressions in app reviews.

    - Contextual Sentiment Detection : Move beyond rule-based labelling and explore contextual sentiment detection using transformer-based models.

- Multi-Lingual Support
    - Language Detection & Translation : Add support for multilingual reviews by integrating language detection (e.g., langdetect library) and translation APIs to ensure comprehensive sentiment coverage across languages commonly used in Grab's markets (e.g., English, Malay, Indonesian, Thai).

- Scalability & Fault Tolerance
    - Distributed Computing : Leverage cloud platforms (e.g., AWS EMR, Azure Databricks) to scale horizontally across multiple nodes for handling larger volumes of streaming data.

    - Fault-Tolerant Streaming : Enhance fault tolerance by implementing end-to-end exactly-once semantics using Kafka transactions and Spark Structured Streaming’s checkpointing features.

By incorporating these enhancements, the sentiment analysis pipeline can evolve into a more intelligent, scalable, and impactful system capable of delivering deeper business insights and improving user experience continuously. Overall, the system demonstrates robustness in handling large-scale data streams, accurate sentiment classification, and effective visualization for actionable insights.

References

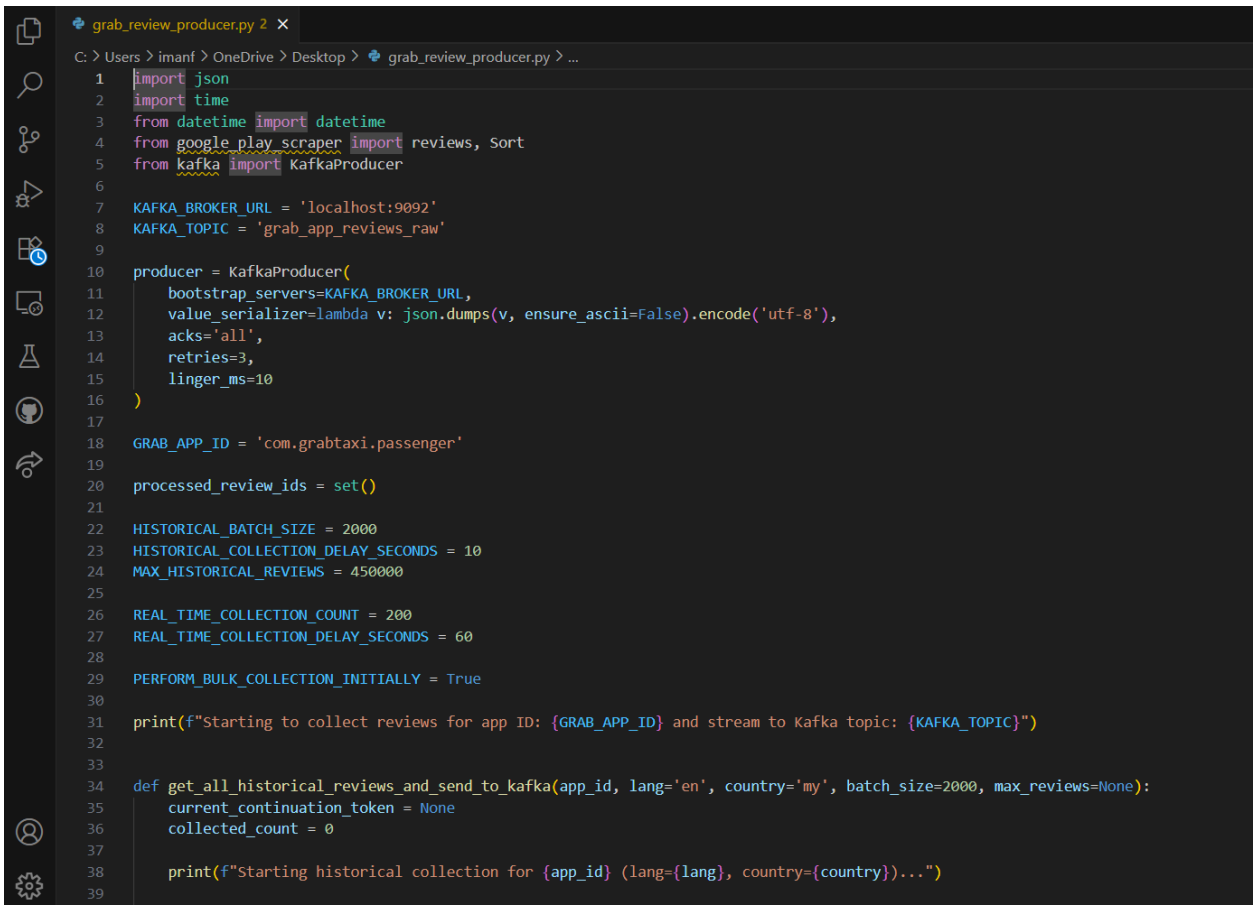
https://github.com/drshahizan/HPDP/blob/main/2425/project/p2/proj2.md

https://kafka.apache.org/quickstart

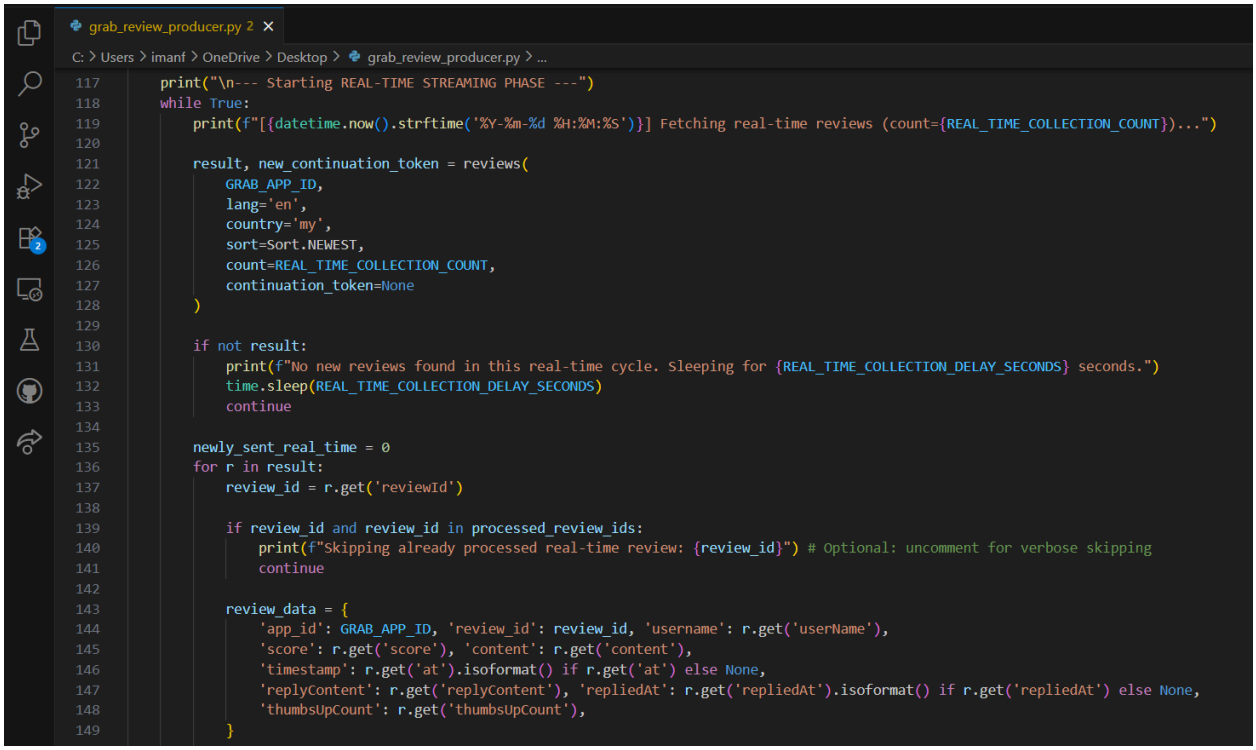
https://www.elastic.co/kibana

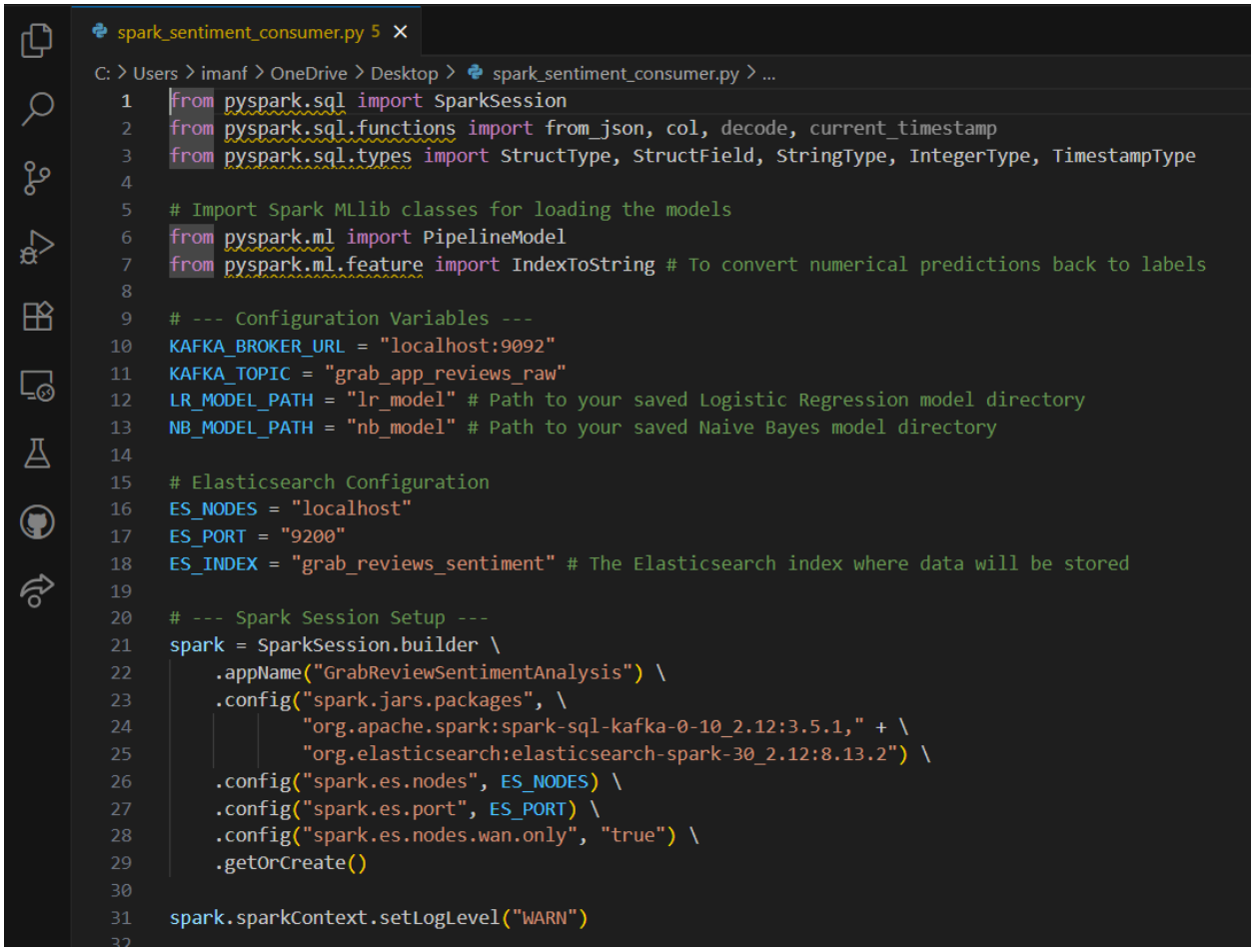
Appendix

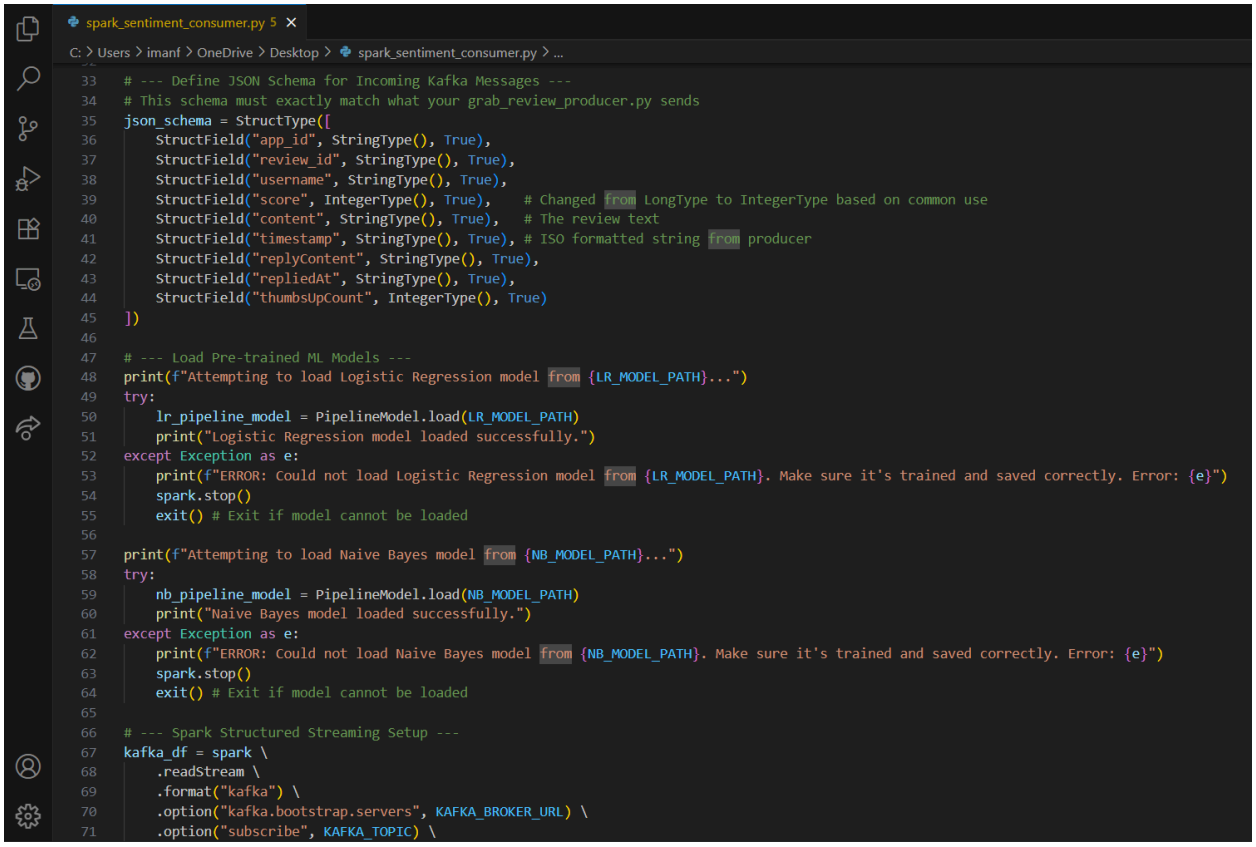
- Code snippets, Configurations, Logs



```python
import json
import time
from datetime import datetime
from google_play_scraper import reviews, Sort
from kafka import KafkaProducer

KAFKA_BROKER_URL = 'localhost:9092'
KAFKA_TOPIC = 'grab_app_reviews_raw'

producer = KafkaProducer(
    bootstrap_servers=KAFKA_BROKER_URL,
    value_serializer=lambda v: json.dumps(v, ensure_ascii=False).encode('utf-8'),
    acks='all',
    retries=3,
    linger_ms=10
)

GRAB_APP_ID = 'com.grabtaxi.passenger'

processed_review_ids = set()

HISTORICAL_BATCH_SIZE = 2000
HISTORICAL_COLLECTION_DELAY_SECONDS = 10
MAX_HISTORICAL_REVIEWS = 450000

REAL_TIME_COLLECTION_COUNT = 200
REAL_TIME_COLLECTION_DELAY_SECONDS = 60

PERFORM_BULK_COLLECTION_INITIALLY = True

print(f"Starting to collect reviews for app ID: {GRAB_APP_ID} and stream to Kafka topic: {KAFKA_TOPIC}")


def get_all_historical_reviews_and_send_to_kafka(app_id, lang='en', country='my', batch_size=2000, max_reviews=None):
    current_continuation_token = None
    collected_count = 0

    print(f"Starting historical collection for {app_id} (lang={lang}, country={country})...")

```


grab_review_producer.py 2
C: > Users > imanf > OneDrive > Desktop > grab_review_producer.py > ...
print("\n--- Starting REAL-TIME STREAMING PHASE ---")
while True:
print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] Fetching real-time reviews (count={REAL_TIME_COLLECTION_COUNT})...")
result, new_continuation_token = reviews(
GRAB_APP_ID,
lang='en',
country='my',
sort=Sort.NEWEST,
count=REAL_TIME_COLLECTION_COUNT,
continuation_token=None
)
if not result:
print(f"No new reviews found in this real-time cycle. Sleeping for {REAL_TIME_COLLECTION_DELAY_SECONDS} seconds.")
time.sleep(REAL_TIME_COLLECTION_DELAY_SECONDS)
continue
newly_sent_real_time = 0
for r in result:
review_id = r.get('reviewId')
if review_id and review_id in processed_review_ids:
print(f"Skipping already processed real-time review: {review_id}") # Optional: uncomment for verbose skipping
continue
review_data = {
'app_id': GRAB_APP_ID, 'review_id': review_id, 'username': r.get('userName'),
'score': r.get('score'), 'content': r.get('content'),
'timestamp': r.get('at').isoformat() if r.get('at') else None,
'replyContent': r.get('replyContent'), 'repliedAt': r.get('repliedAt').isoformat() if r.get('repliedAt') else None,
'thumbsUpCount': r.get('thumbsUpCount'),
}


spark_sentiment_consumer.py 5
C: > Users > imanf > OneDrive > Desktop > spark_sentiment_consumer.py > ...
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, decode, current_timestamp
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType
# Import Spark MLlib classes for loading the models
from pyspark.ml import PipelineModel
from pyspark.ml.feature import IndexToString # To convert numerical predictions back to labels
# --- Configuration Variables ---
KAFKA_BROKER_URL = "localhost:9092"
KAFKA_TOPIC = "grab_app_reviews_raw"
LR_MODEL_PATH = "lr_model" # Path to your saved Logistic Regression model directory
NB_MODEL_PATH = "nb_model" # Path to your saved Naive Bayes model directory
# Elasticsearch Configuration
ES_NODES = "localhost"
ES_PORT = "9200"
ES_INDEX = "grab_reviews_sentiment" # The Elasticsearch index where data will be stored
# --- Spark Session Setup ---
spark = SparkSession.builder \
.appName("GrabReviewSentimentAnalysis") \
.config("spark.jars.packages", \
"org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.1," + \
"org.elasticsearch:elasticsearch-spark-30_2.12:8.13.2") \
.config("spark.es.nodes", ES_NODES) \
.config("spark.es.port", ES_PORT) \
.config("spark.es.nodes.wan.only", "true") \
.getOrCreate()
spark.sparkContext.setLogLevel("WARN")


spark_sentiment_consumer.py 5
C: > Users > imanf > OneDrive > Desktop > spark_sentiment_consumer.py > ...
# --- Define JSON Schema for Incoming Kafka Messages ---
# This schema must exactly match what your grab_review_producer.py sends
json_schema = StructType([
StructField("app_id", StringType(), True),
StructField("review_id", StringType(), True),
StructField("username", StringType(), True),
StructField("score", IntegerType(), True), # Changed from LongType to IntegerType based on common use
StructField("content", StringType(), True), # The review text
StructField("timestamp", StringType(), True), # ISO formatted string from producer
StructField("replyContent", StringType(), True),
StructField("repliedAt", StringType(), True),
StructField("thumbsUpCount", IntegerType(), True)
])
# --- Load Pre-trained ML Models ---
print(f"Attempting to load Logistic Regression model from {LR_MODEL_PATH}...")
try:
lr_pipeline_model = PipelineModel.load(LR_MODEL_PATH)
print("Logistic Regression model loaded successfully.")
except Exception as e:
print(f"ERROR: Could not load Logistic Regression model from {LR_MODEL_PATH}. Make sure it's trained and saved correctly. Error: {e}")
spark.stop()
exit() # Exit if model cannot be loaded
print(f"Attempting to load Naive Bayes model from {NB_MODEL_PATH}...")
try:
nb_pipeline_model = PipelineModel.load(NB_MODEL_PATH)
print("Naive Bayes model loaded successfully.")
except Exception as e:
print(f"ERROR: Could not load Naive Bayes model from {NB_MODEL_PATH}. Make sure it's trained and saved correctly. Error: {e}")
spark.stop()
exit() # Exit if model cannot be loaded
# --- Spark Structured Streaming Setup ---
kafka_df = spark \
.readStream \
.format("kafka") \
.option("kafka.bootstrap.servers", KAFKA_BROKER_URL) \
.option("subscribe", KAFKA_TOPIC) \