National University of Computer and Emerging Sciences

Fundamentals of Big Data Analytics Phase

Pre-Processing and Analysis Report

Group Members:

Name	Roll Number	Section
M. Hassan Mustansar	22L-7521	BDS-4A
Huzaifa Amir	22L-7518	BDS-4A
Mannan Ul Haq	22L-7556	BDS-4A
M. Abdullah Waseem	22L-7522	BDS-4A

Fundamentals of Big Data Analytics
Department of Data Science
FAST-NU, Lahore, Pakistan

Phase 1

1. Introduction: In this phase a detailed analysis of the preprocessing steps and analysis conducted on the sentiment dataset using PySpark.

2. Data Collection: The dataset "Sentiment.csv" was loaded using PySpark. It contains Tweet's text data and corresponding sentiment labels.

```
Data = spark.read.csv("Sentiment.csv", header = True, inferSchema = True)
Data = Data.select(Data.text, Data.sentiment)
Data.show()
```

3. Data Cleaning and Preprocessing:
    a. Retweets, which are often redundant and don't contribute to sentiment analysis, were removed from the dataset.
    b. Duplicate entries, if any, were eliminated to ensure that each piece of text is unique and contributes independently to the analysis.
    c. Null values, if present, were dropped from the dataset to prevent any inconsistencies in subsequent analysis steps.

```
# Removing Retweets
Data = Data.filter(~Data.text.startswith("RT"))
# Removing Duplicates
Data = Data.dropDuplicates(["text"])
# To Drop the Null Values ##
Data = Data.na.drop(how = "all")
```

    d. Special characters, emojis, URLs, mentions, and hashtags were removed.
    e. Text was converted to lowercase to ensure uniformity in text representation. This step helps in treating words with different cases (e.g., "Happy" and "happy") as the same, thus avoiding redundancy in the analysis.

```
def Tweet_Preprocessing(text):
    # Remove special characters
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Remove emojis
    text = text.encode('ascii', 'ignore').decode('ascii')
    # Remove URLs
    text = re.sub(r'http\S+', '', text)
    # Remove mentions
    text = re.sub(r'@\w+', '', text)
    # Remove hashtags
    text = re.sub(r'#\w+', '', text)
    # Convert to lowercase
    text = text.lower()

    return text
```

f. The text was tokenized into individual words or tokens using NLTK's word_tokenize function. Tokenization breaks down the text into smaller units, which facilitates further analysis.

g. Stop words, which are commonly occurring words that do not carry significant meaning for sentiment analysis (e.g., "the", "is", "are"), were removed from the tokenized text. This helps in reducing the dimensionality of the data and focusing on more meaningful words for sentiment classification.

```
# Tokenization
def Tokenization(text):
  Tokens = word_tokenize(text)

  return Tokens

tokenize_udf = udf(Tokenization, ArrayType(StringType()))
Data = Data.select(tokenize_udf(Data.text).alias("text"), Data.sentiment)

# Removing StopWords
remover = StopWordsRemover(inputCol = "text", outputCol = "filtered_text")
Data = remover.transform(Data)

Data = Data.select(Data.filtered_text.alias("Tweets"), Data.sentiment)
Data.show(truncate = False)
```

4. Feature Extraction:
    a. HashingTF was applied to convert tokenized text into numerical features. This technique converts a collection of text documents into numerical feature vectors. Each feature represents the frequency of a token (word) in the document.

b. IDF was applied to weigh the importance of features. IDF measures the importance of a term in a document relative to the entire corpus. It downweights terms that appear frequently across documents and upweights terms that are rare.

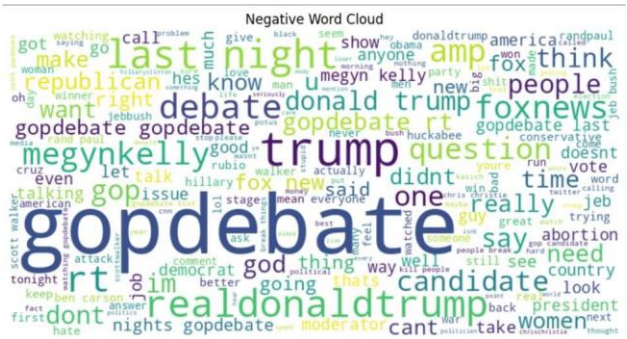
```
# Step 3: Feature Extraction
from pyspark.ml.feature import HashingTF, IDF

hashingTF = HashingTF(inputCol = "Tweets", outputCol = "rawFeatures", numFeatures
Data = hashingTF.transform(Data)

idf = IDF(inputCol = "rawFeatures", outputCol = "Features")
idf_model = idf.fit(Data)
Data = idf_model.transform(Data)

Data = Data.select(Data.Tweets, Data.Features, Data.sentiment)
Data.show(truncate = False)
```

5. Sentiment Analysis and Visualization:
   Word clouds were generated for each sentiment category (positive, negative, neutral) to visualize the most frequent words.



6. Train-Test Split:
   The dataset was split into training and testing sets with an 80-20 ratio.

```
# Step 5: Train-Test Split

train_data, test_data = Data.randomSplit([0.8, 0.2], seed = 42)
```

7. String Indexer:

String Indexer is used to convert sentiment labels (e.g., positive, neutral, negative) into numerical values. To further apply the regression models on the targeted column.

```
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol = "sentiment", outputCol = "label")
train_data = indexer.fit(train_data).transform(train_data)
test_data = indexer.fit(test_data).transform(test_data)
```

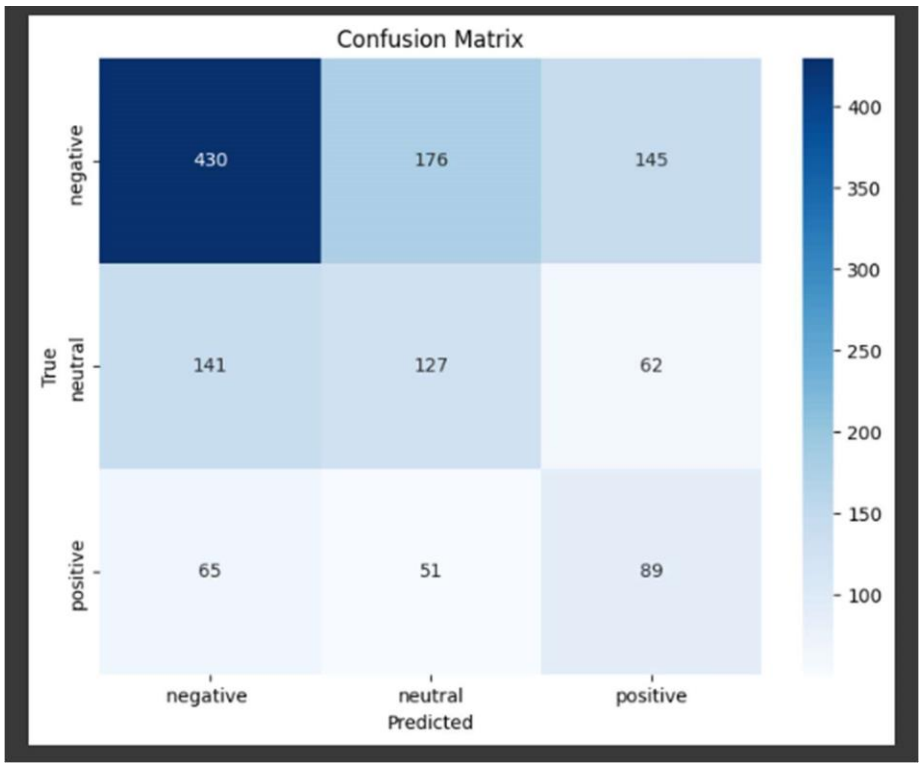
8. Model Training and Evaluation:
   a. Logistic Regression:

Logistic Regression is a supervised learning algorithm used for binary classification tasks. It models the probability that a given input belongs to a particular class. The algorithm fits a sigmoid function to the input features, which maps the input to a probability between 0 and 1. This probability is then thresholded to make binary predictions. In Spark ML, Logistic Regression is implemented as a classification algorithm.

- Achieved an accuracy of approximately 50%.
- Precision, recall, and F1-score were computed.
- A confusion matrix was plotted to visualize the performance.

```
Logistic Regression:

Accuracy: 0.5023328149300156
Precision: 0.5348210858079565
Recall: 0.5023328149300156
F1 Score: 0.5140207452327284
```



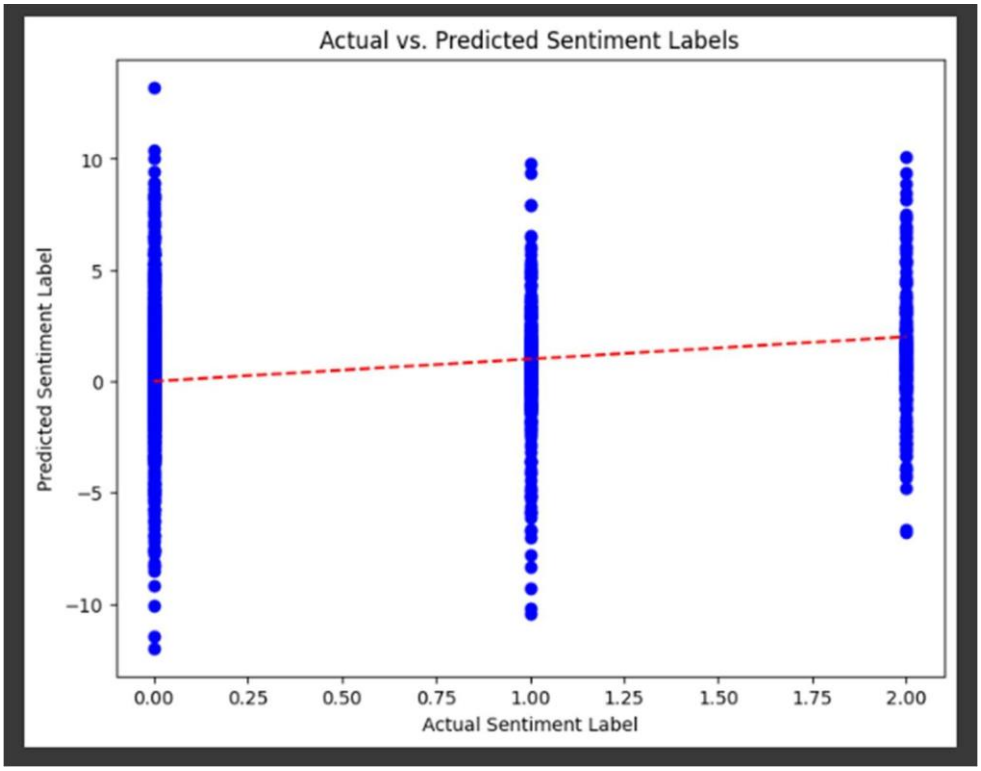
b. Linear Regression:

Linear Regression is a supervised learning algorithm used for regression tasks. It models the relationship between independent variables (features) and a dependent variable (target) by fitting a linear equation to the observed data. In its simplest form, linear regression assumes a linear relationship between the input features and the target variable.

- RMSE was used as the evaluation metric.
- Achieved an accuracy of around 44%.
- Precision, recall, and F1-score were calculated.
- A scatter plot was generated to compare actual and predicted sentiment labels.

```
Linear Regression:

Root Mean Squared Error (RMSE): 3.0330106356423268
Accuracy: 0.44245723172628304
Precision: 0.3314814814814815
Recall: 0.5424242424242425
F1-Score: 0.4114942528735632
```



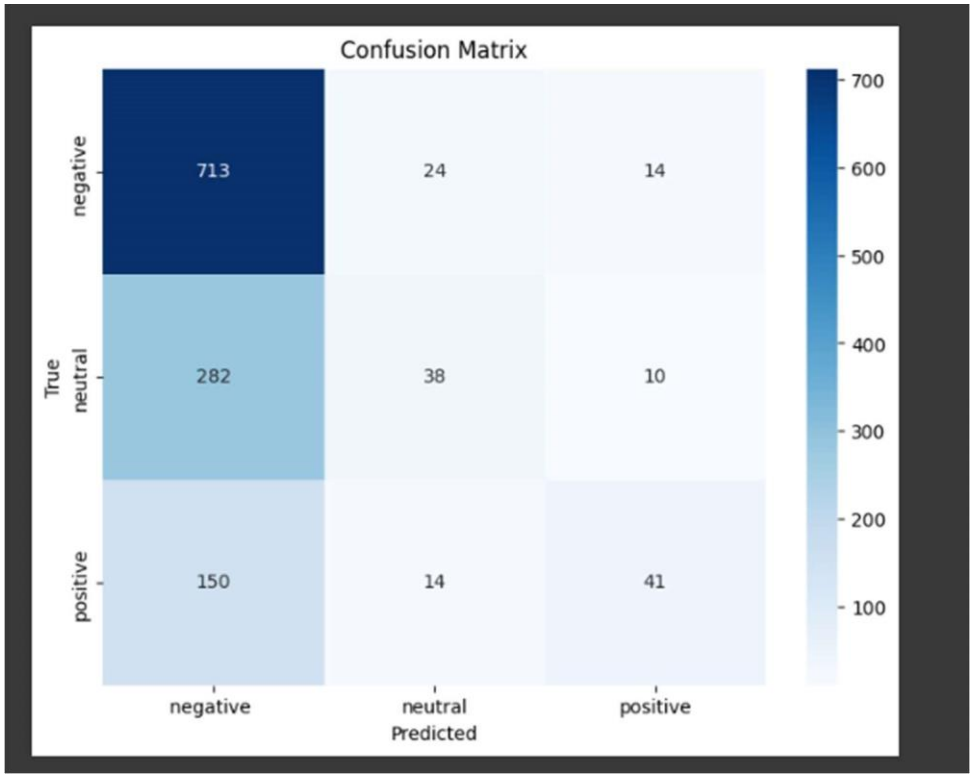
c. Stochastic Gradient Descent (SGD):

Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used for training machine learning models, particularly for large-scale datasets. It is a variant of gradient descent optimization, where the model parameters are updated iteratively based on the gradient of the loss function with respect to the parameters.

- Accuracy, F1-score, precision, and recall were computed.
- A confusion matrix was plotted.

```
Stochastic Gradient Descent:

Accuracy: 0.6158631415241057
F1-score: 0.5356664435922952
Precision: 0.5925046402930503
Recall: 0.6158631415241058
```



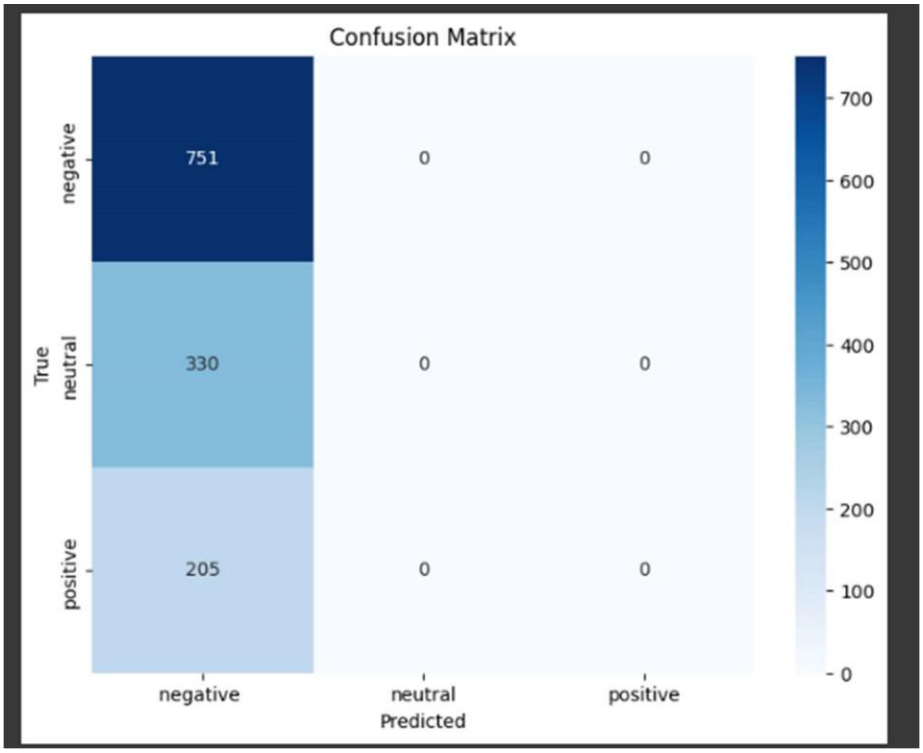
d. Batch Gradient Descent:

Batch Gradient Descent is a classical optimization algorithm used for training machine learning models, particularly in the context of gradientbased optimization. It aims to minimize a cost function by iteratively updating the model parameters in the direction of the steepest descent of the cost function.

- Accuracy, F1-score, precision, and recall were calculated.
- A confusion matrix was visualized.

```
Batch Gradient:

Accuracy: 0.5839813374805599
F1-score: 0.43060381389091845
Precision: 0.3410342025255836
Recall: 0.5839813374805599
```

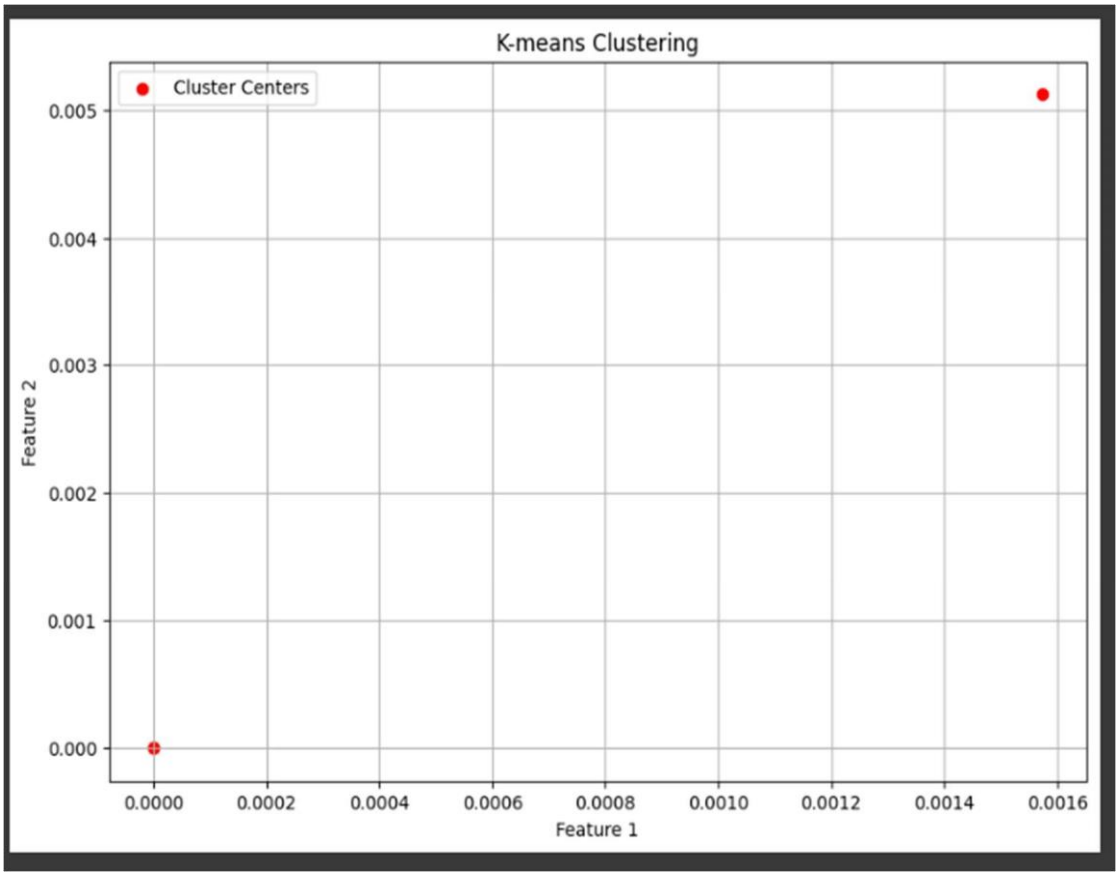


e. K-Means Clustering:

K-Means Clustering is an unsupervised learning algorithm used for clustering similar data points into groups or clusters. It aims to partition the data into K clusters, where each cluster represents a group of data points with similar characteristics.

- Feature vectors were clustered using K-means with 3 clusters.
- Cluster centers were identified and visualized on a scatter plot.

```
Cluster Centers:
Cluster 1: [0.00157245 0.00512476 0.00679854 ... 0.00157245 0.         0.00157245]
Cluster 2: [0. 0. 0. ... 0. 0. 0.]
Cluster 3: [0. 0. 0. ... 0. 0. 0.]
```



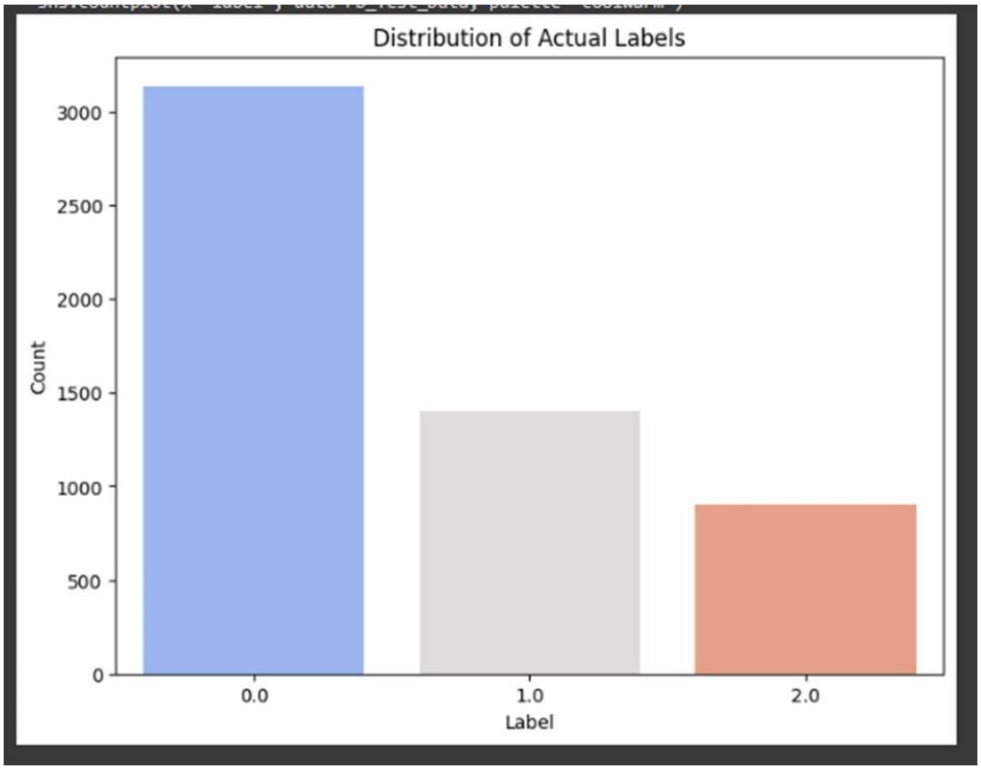
f. K-Nearest Neighbors (KNN):

K-Nearest Neighbors (KNN) is a non-parametric and lazy learning algorithm used for classification and regression tasks. It is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance function) to the k nearest neighbors.

- KNN model was trained and evaluated.
- Accuracy, precision, recall, and F1-score were computed.
- A count plot was generated to show the distribution of actual labels.

```
K-Neighbors:

Accuracy: 0.583701250919794
Precision: 0.6140706542099357
Recall: 0.583701250919794
F1 Score: 0.5612391348526783
```



Phase 2

1. Introduction: In this phase, a detailed analysis of the preprocessing steps and similarity between ingredients and recipes is conducted on the recipes dataset using PySpark.
2. Data Collection: The dataset " recipes.csv" was loaded using PySpark. It contains recipes and corresponding ingredients labels.

```
data = spark.read.csv("recipes_combined.csv", header=True,inferSchema=True)
data.show()
Col=data.columns
print(Col)
```

3. Data Cleaning and Preprocessing:

   a. All unnecessary columns have been dropped, and all special symbols and brackets have been replaced with spaces.

```
columns_to_drop = ['_c2', '_c3', '_c4', '_c5', '_c6', '_c7', '_c8', '_c9', '_c10', '_c11', '_c12', '_c13', '_c14', '_c15','all_ingredients']
Filter_data = data.drop(*columns_to_drop)
Filter_data.show(truncate=False)

from pyspark.sql.functions import regexp_replace

filtered_data = Filter_data.withColumn('ingredients', regexp_replace('ingredients', '//', ' ')) \
    .withColumn('ingredients', regexp_replace('ingredients', '[(|)]', ''))
```

b. In a similar manner to the previous phase, a tokenizer is employed to segment text into smaller units, thereby enhancing its utility and facilitating further analysis.

```
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
tokenizer = Tokenizer(inputCol="ingredients", outputCol="words")
```

4. Feature Extraction:
   a. HashingTF was applied to convert tokenized text into numerical features. This technique converts a collection of text documents into numerical feature vectors. Each feature represents the frequency of a token (word) in the document.
   b. IDF was applied to weigh the importance of features. IDF measures the importance of a term in a document relative to the entire corpus. It downweights terms that appear frequently across documents and upweights terms that are rare.
   c. Data is pipelined to prevent any loss and to automate and streamline the process of applying transformations sequentially.

```
from pyspark.ml.feature import  HashingTF, IDF
from pyspark.ml import Pipeline
from pyspark.sql.functions import col

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures")

idf = IDF(inputCol="rawFeatures", outputCol="features")


pipeline = Pipeline(stages=[tokenizer,hashingTF, idf])
pipeline_model = pipeline.fit(filtered_data)
transformed_data = pipeline_model.transform(filtered_data)
```

5. Similarity Algorithms:

a. A user-defined function is employed to design the cosine similarity function, which computes the cosine of angle between two vectors.

```
from pyspark.ml.linalg import Vectors
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType

def cosine_similarity(vec1, vec2):
    dot_product = float(vec1.dot(vec2))
    norm_vec1 = float(vec1.norm(2))
    norm_vec2 = float(vec2.norm(2))
    return dot_product / (norm_vec1 * norm_vec2)

cosine_similarity_udf = udf(cosine_similarity, DoubleType())
joined_data = Filter_data.alias("df1").crossJoin(Filter_data.alias("df2"))
cosine_similarity_result = joined_data.select(
    "df1.recipeNames",
    "df2.recipeNames",
    cosine_similarity_udf("df1.features", "df2.features").alias("cosine_similarity")
)
cosine_similarity_result = cosine_similarity_result.filter("df1.recipeNames != df2.recipeNames")
```

b. Centered cosine similarity function, which calculates the Pearson correlation coefficient between two vectors after adjusting them by subtracting their means.

```
def centered_cosine_similarity(vec1, vec2):
    # Compute means of each vector's components
    mean_vec1 = sum(vec1) / len(vec1)
    mean_vec2 = sum(vec2) / len(vec2)

    # Center the vectors by subtracting the means
    centered_vec1 = DenseVector(vec1 - mean_vec1)
    centered_vec2 = DenseVector(vec2 - mean_vec2)

    # Compute cosine similarity between centered vectors
    dot_product = float(centered_vec1.dot(centered_vec2))
    norm_centered_vec1 = float(centered_vec1.norm(2))
    norm_centered_vec2 = float(centered_vec2.norm(2))

    return dot_product / (norm_centered_vec1 * norm_centered_vec2)

centered_cosine_similarity_udf = udf(centered_cosine_similarity, DoubleType())
centered_cosine_similarity_result = joined_data.select(
    "df1.recipeNames",    "df2.recipeNames",
    centered_cosine_similarity_udf("df1.features", "df2.features").alias("centered_cosine_similarity")
)
```