```
# Databricks notebook source
# MAGIC %md
# MAGIC
# MAGIC This notebook will show the Multiclass Sentiment Analysis in Online Food
Reviews: A PySpark and Spark ML Approach

# COMMAND ----------

# File location and type
file_location = "/FileStore/tables/Reviews.csv"
file_type = "csv"

# CSV options
infer_schema = "false"
first_row_is_header = "false"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be
ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df)

# COMMAND ----------

# Create a view or table

temp_table_name = "Reviews_csv"

df.createOrReplaceTempView(temp_table_name)

# COMMAND ----------

# MAGIC %sql
# MAGIC
# MAGIC /* Query the created temp table in a SQL cell */
# MAGIC
# MAGIC select * from `Reviews_csv`

# COMMAND ----------

.

permanent_table_name = "Reviews_csv"

# df.write.format("csv").saveAsTable(permanent_table_name)

# COMMAND ----------

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf, lower, regexp_replace, when
from pyspark.sql.types import StringType, IntegerType
from pyspark.ml import Pipeline
from pyspark.ml.feature import RegexTokenizer, StopWordsRemover, HashingTF, IDF,
StringIndexer, CountVectorizer
```

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Initialize Spark Session
spark = SparkSession.builder.appName("SentimentAnalysis").getOrCreate()

# File location and type
file_location = "/FileStore/tables/Reviews.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# Read the CSV file to DataFrame
df = spark.read.csv(file_location, inferSchema=True, header=True, sep=delimiter)

# Drop duplicates and rows with missing values
df = df.dropDuplicates(['Text']).na.drop(subset=['Text', 'Score'])

# Convert the score to a sentiment label using withColumn and when
df = df.withColumn("Sentiment", when(col("Score") < 3, 'Negative')
                                  .when(col("Score") == 3, 'Neutral')
                                  .otherwise('Positive'))

# Remove punctuation, hashtags, and links
df = df.withColumn("text", lower(col("Text")))
df = df.withColumn("text", regexp_replace(col("text"), r"http\S+", ""))
df = df.withColumn("text", regexp_replace(col("text"), "[^a-zA-Z\s]", ""))




# COMMAND ----------

from pyspark.sql import functions as F

from pyspark.sql.functions import col

# Calculate the counts of each class
class_counts = df.groupBy("Sentiment").count()
class_counts.show()

# Extract the counts for each sentiment
positive_count = class_counts.filter(col("Sentiment") == "Positive").collect()[0]
["count"]
neutral_count = class_counts.filter(col("Sentiment") == "Neutral").collect()[0]
["count"]
negative_count = class_counts.filter(col("Sentiment") == "Negative").collect()[0]
["count"]


# Target count for balancing (using the smallest class count)
target_count = min(positive_count, neutral_count, negative_count)

# Calculate sampling ratios
positive_sample_ratio = target_count / positive_count
negative_sample_ratio = target_count / negative_count
```

```
# Undersampling the 'Positive' class and oversampling 'Neutral' class
positive_df = df.filter(col("Sentiment") == "Positive").sample(False,
positive_sample_ratio)
negative_df = df.filter(col("Sentiment") == "Negative").sample(True,
negative_sample_ratio)

# Keeping the 'Negative' class as is
neutral_df = df.filter(col("Sentiment") == "Neutral")

# Combine the datasets
balanced_df = positive_df.unionAll(negative_df).unionAll(neutral_df)

# COMMAND ----------

import matplotlib.pyplot as plt

sentiment_counts = balanced_df.groupBy('Sentiment').count().toPandas()
sentiment_counts.plot(kind='bar', x='Sentiment', y='count')
plt.title('Distribution of Sentiments of Balanced Set')
plt.xlabel('Sentiment')
plt.ylabel('Count')
plt.show()

# COMMAND ----------

# MAGIC %pip install wordcloud

# COMMAND ----------


import pandas as pd
from wordcloud import WordCloud
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from matplotlib import style
# Assuming you have a SparkSession named "spark" created
# and 'balanced_df' is your PySpark DataFrame
positive_dfnp = positive_df.toPandas()

textcol = ' '.join([word for word in positive_dfnp['Summary']])
plt.figure(figsize=(20,15), facecolor='None')
wordcloud = WordCloud(max_words=500, width=1600, height=800).generate(textcol)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Most frequent words in positive reviews', fontsize = 19)
plt.show()

# COMMAND ----------


# Assuming you have a SparkSession named "spark" created
# and 'balanced_df' is your PySpark DataFrame
negative_dfnp = negative_df.toPandas()

textcol1 = ' '.join([word for word in negative_dfnp['Summary']])
plt.figure(figsize=(20,15), facecolor='None')
```

```
wordcloud = WordCloud(max_words=500, width=1600, height=800).generate(textcol1)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Most frequent words in Negative reviews', fontsize = 19)
plt.show()

# COMMAND ----------

neutral_dfnp = neutral_df.toPandas()

textcol1 = ' '.join([word for word in neutral_dfnp['Summary']])
plt.figure(figsize=(20,15), facecolor='None')
wordcloud = WordCloud(max_words=500, width=1600, height=800).generate(textcol1)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Most frequent words in Neutral Reviews', fontsize = 19)
plt.show()

# COMMAND ----------

class_counts = balanced_df.groupBy("Sentiment").count()
display(class_counts)
class_counts.show()

# COMMAND ----------

# Preprocessing Pipeline
from pyspark.ml.feature import RegexTokenizer, StopWordsRemover, HashingTF, IDF,
StringIndexer, CountVectorizer
from pyspark.sql.types import IntegerType
from pyspark.ml.feature import StringIndexerModel
from pyspark.sql.functions import col, udf, lower, regexp_replace
from pyspark.ml.feature import NGram

tokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="\\W")
stopWordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered")
ngram = NGram(n=2, inputCol="filtered", outputCol="ngrams")  # Set n=2 for bi-
grams, n=3 for tri-grams

cv = CountVectorizer(inputCol="ngrams", outputCol="features")
label_stringIdx = StringIndexer(inputCol="Sentiment", outputCol="labelIndex")

# COMMAND ----------

from pyspark.ml.feature import RegexTokenizer
from collections import Counter

tokenizerS = RegexTokenizer(inputCol="Summary", outputCol="words", pattern="\\W")


df_tokenized = tokenizerS.transform(balanced_df)


from pyspark.ml.feature import StopWordsRemover


Remover = StopWordsRemover(inputCol="words", outputCol="filtered")
```

```
df_filtered = Remover.transform(df_tokenized)


# Collected the filtered words into a Pandas DataFrame for processing
filtered_words = df_filtered.select('filtered').rdd.flatMap(lambda x: x).collect()
flat_list = [word for sublist in filtered_words for word in sublist]

# Count the words
count = Counter(flat_list)
count.most_common(15)


# COMMAND ----------

import pandas as pd
import plotly.express as px

# Converting the Counter object into a Pandas DataFrame
word_counts = pd.DataFrame(count.most_common(15), columns=['word', 'count'])

#  bar chart
fig = px.bar(word_counts, x='count', y='word', title='Common Words Across Reviews',
color='word', orientation='h')

#  plot
fig.show()


# COMMAND ----------


(trainingData, testData) = balanced_df.randomSplit([0.7, 0.3], seed=100)


# COMMAND ----------

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf, lower, regexp_replace
from pyspark.ml import Pipeline
from pyspark.ml.feature import RegexTokenizer, StopWordsRemover, HashingTF, IDF,
StringIndexer, CountVectorizer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql.types import IntegerType
from pyspark.ml.feature import StringIndexerModel


# Classifie
lr = LogisticRegression(featuresCol="features", labelCol="labelIndex")

# Pipeline
pipeline = Pipeline(stages=[tokenizer, stopWordsRemover,ngram, cv, label_stringIdx,
lr])


# Training
model1 = pipeline.fit(trainingData)

# Predicting on the test data
```

```
predictions = model1.transform(testData)

# Evaluating the model with all metrics
evaluator = MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="labelIndex", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Accuracy: {:.2f}".format(accuracy))

# Evaluate and print precision
precision = evaluator.evaluate(predictions, {evaluator.metricName:
"weightedPrecision"})
print(f"Precision: {precision:.2f}")

# Evaluate and print recall
recall = evaluator.evaluate(predictions, {evaluator.metricName: "weightedRecall"})
print(f"Recall: {recall:.2f}")

# Evaluate and print F1 Score
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
print(f"F1 Score: {f1:.2f}")



# COMMAND ----------

from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml import Pipeline

# Logistic Regression
lr = LogisticRegression(featuresCol="features", labelCol="labelIndex")

# Pipeline
pipeline_lr = Pipeline(stages=[tokenizer, stopWordsRemover,ngram, cv,
label_stringIdx, lr])

# Hyperparameter Tuning
paramGrid_lr = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.maxIter, [10, 20]) \
    .build()

crossval_lr = CrossValidator(estimator=pipeline_lr,
                             estimatorParamMaps=paramGrid_lr,

evaluator=MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="accuracy"),
                             numFolds=5)

# Training and Evaluation
cvModel_lr = crossval_lr.fit(trainingData)
prediction_lr = cvModel_lr.transform(testData)
accuracy_lr = evaluator.evaluate(prediction_lr)
print("Logistic Regression (with CV) Accuracy: {:.2f}".format(accuracy_lr))


# COMMAND ----------
```

```
from pyspark.ml.feature import Word2Vec
from pyspark.ml.classification import RandomForestClassifier

# Random Forest Classifier
rf = RandomForestClassifier(featuresCol="features", labelCol="labelIndex",
numTrees=100)


pipeline_rf = Pipeline(stages=[tokenizer, stopWordsRemover,ngram,  cv,
label_stringIdx, rf])

# Training the models

model_rf = pipeline_rf.fit(trainingData)

predictions_rf = model_rf.transform(testData)

# Evaluate the models
evaluator2 = MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="labelIndex", metricName="accuracy")

accuracy_rf = evaluator2.evaluate(predictions_rf)

print("Random Forest Accuracy: {:.2f}".format(accuracy_rf))
# Evaluate and print precision
precision_rf = evaluator2.evaluate(predictions_rf, {evaluator2.metricName:
"weightedPrecision"})
print(f"Precision: {precision:.2f}")

# Evaluate and print recall
recall_rf = evaluator2.evaluate(predictions_rf, {evaluator2.metricName:
"weightedRecall"})
print(f"Recall: {recall:.2f}")

# Evaluate and print F1 Score
f1_rf = evaluator2.evaluate(predictions_rf, {evaluator2.metricName: "f1"})
print(f"F1 Score: {f1:.2f}")



# COMMAND ----------

from pyspark.sql import SparkSession
from pyspark.ml.classification import LinearSVC, OneVsRest
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Initializing the base SVM classifier
svm = LinearSVC(maxIter=10, regParam=0.1)

# Instantiate OneVsRest classifier
ovr = OneVsRest(classifier=svm,featuresCol="features", labelCol="labelIndex")

pipeline_svm = Pipeline(stages=[tokenizer, stopWordsRemover, ngram, cv,
label_stringIdx, ovr])

model_svm = pipeline_svm.fit(trainingData)

predictions_svm = model_svm.transform(testData)
```

```
# Evaluate the model
evaluator3 = MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="accuracy")
accuracy_svm = evaluator3.evaluate(predictions_svm)
print(f"Test Accuracy: {accuracy_svm:.2f}")

precision_svm = evaluator3.evaluate(predictions_svm, {evaluator3.metricName:
"weightedPrecision"})
print(f"SVM Precision: {precision_svm:.2f}")

recall_svm = evaluator3.evaluate(predictions_svm, {evaluator3.metricName:
"weightedRecall"})
print(f"SVM Recall: {recall_svm:.2f}")

f1_svm = evaluator3.evaluate(predictions_svm, {evaluator3.metricName: "f1"})
print(f"SVM F1 Score: {f1_svm:.2f}")


# COMMAND ----------

from pyspark.ml.classification import NaiveBayes
from pyspark.ml.feature import IDF
from pyspark.ml.feature import HashingTF

# Hashing term frequency
#hashingTF = HashingTF(inputCol="filtered", outputCol="rawFeatures")


# Inverse document frequency
##idf = IDF(inputCol="rawFeatures", outputCol="features")

# Naive Bayes Classifier
nb = NaiveBayes(featuresCol="features", labelCol="labelIndex")

pipeline_nb = Pipeline(stages=[tokenizer, stopWordsRemover,ngram, cv,
label_stringIdx, nb])


# Train the model
model_nb = pipeline_nb.fit(trainingData)

predictions_nb = model_nb.transform(testData)
evaluator1 = MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="accuracy")

# Evaluate the model
accuracy_nb = evaluator1.evaluate(predictions_nb)
print("Naive Bayes Accuracy: {:.2f}".format(accuracy_nb))


# COMMAND ----------


# Evaluate and print precision
precision_nb = evaluator1.evaluate(predictions_nb, {evaluator1.metricName:
"weightedPrecision"})
print(f"Naive Bayes Precision: {precision_nb:.2f}")
```

```
# Evaluate and print recall
recall_nb = evaluator1.evaluate(predictions_nb, {evaluator1.metricName:
"weightedRecall"})
print(f"Naive Bayes Recall: {recall_nb:.2f}")

# Evaluate and print F1 Score
f1_nb = evaluator1.evaluate(predictions_nb, {evaluator1.metricName: "f1"})
print(f"Naive Bayes F1 Score: {f1_nb:.2f}")



# COMMAND ----------

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

paramGrid = ParamGridBuilder() \
    .addGrid(nb.smoothing, [0.5, 1.0, 2.0]) \
    .build()

evaluator2 = MulticlassClassificationEvaluator(labelCol="labelIndex",
predictionCol="prediction", metricName="accuracy")

# Set up cross-validation
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator2,
                          numFolds=5)  # Number of folds can be adjusted

# Run cross-validation
cvModel = crossval.fit(trainingData)

# Using the best model
bestModel = cvModel.bestModel


predictionscv = bestModel.transform(testData)

# Calculating and printing accuracy
accuracy = evaluator2.evaluate(predictionscv, {evaluator2.metricName: "accuracy"})
print(f"Accuracy: {accuracy:.2f}")

# Calculate and print precision
precision = evaluator2.evaluate(predictionscv, {evaluator2.metricName:
"weightedPrecision"})
print(f"Precision: {precision:.2f}")

# Calculate and print recall
recall = evaluator2.evaluate(predictionscv, {evaluator2.metricName:
"weightedRecall"})  # Note: weightedRecall is the same as Recall in multiclass
classification
print(f"Recall: {recall:.2f}")

# Calculate and print F1 score
f1 = evaluator2.evaluate(predictionscv, {evaluator2.metricName: "f1"})
print(f"F1 Score: {f1:.2f}")



# COMMAND ----------
```

```
from pyspark.sql import Row

# Checking on SVM model
sentence = "The food taste is awful"

# Create a DataFrame
df_test = spark.createDataFrame([Row(text=sentence)])


df_transformed = model_svm.transform(df_test)

predicted_index = df_transformed.select("prediction").collect()[0]["prediction"]

label_model = model_svm.stages[-2]  # Adjust index as per your pipeline

if isinstance(label_model, StringIndexerModel):
    # Access the labels attribute
    labels = label_model.labels
    # Map the numerical prediction to the original label
    predicted_label = labels[int(predicted_index)]
    print(f"Predicted sentiment: {predicted_label}")
else:
    print("Error: The retrieved stage is not a StringIndexerModel.")


# COMMAND ----------

# MAGIC
# MAGIC %pip install pandas textblob spacy
# MAGIC python -m spacy download en_core_web_sm
# MAGIC
# MAGIC import pandas as pd
# MAGIC from textblob import TextBlob
# MAGIC import spacy
# MAGIC
# MAGIC # Load your dataset
# MAGIC df = pd.read_csv('/Workspace/Users/nsggh@mail.umkc.edu/export.csv')
# MAGIC
# MAGIC df['text'] = df['text'].astype(str)
# MAGIC
# MAGIC
# MAGIC nlp = spacy.load('en_core_web_sm')
# MAGIC
# MAGIC
# MAGIC def calculate_sentiment(text):
# MAGIC     blob = TextBlob(text)
# MAGIC     return blob.sentiment.polarity, blob.sentiment.subjectivity
# MAGIC
# MAGIC def label_sentiment(polarity):
# MAGIC     if polarity > 0:
# MAGIC         return 'positive'
# MAGIC     elif polarity < 0:
# MAGIC         return 'negative'
# MAGIC     else:
# MAGIC         return 'neutral'
# MAGIC
# MAGIC # Function to perform NER using spaCy
# MAGIC def extract_entities(text):
```

```
# MAGIC     doc = nlp(text)
# MAGIC     return [(ent.text, ent.label_) for ent in doc.ents]
# MAGIC
# MAGIC df['polarity'], df['subjectivity'] =
zip(*df['text'].apply(calculate_sentiment))
# MAGIC
# MAGIC df['sentiment_label'] = df['polarity'].apply(label_sentiment)
# MAGIC
# MAGIC df['entities'] = df['text'].apply(extract_entities)
# MAGIC
# MAGIC # Write the enriched dataset to a CSV file
# MAGIC output_csv_path = 'enhanced_dataset.csv'
# MAGIC df.to_csv(output_csv_path, index=False)
# MAGIC
# MAGIC print(f"Dataset with sentiment and NER saved to {output_csv_path}")
# MAGIC display(df)
```