Assessed Coursework 3

CID 01252821

```python
# Needed imports for the whole notebook
import math
from os.path import join
from typing import List

from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.ml import Pipeline
from pyspark.sql import SparkSession
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.feature import CountVectorizer, VectorAssembler
from pyspark.ml.functions import array_to_vector, vector_to_array
from pyspark.sql.functions import (
  col, collect_set, exp, explode, expr, log, size, split, sum, udf
)
from pyspark.sql.types import ArrayType, DoubleType, IntegerType
```

Question 1

```python
# Creating a SparkSession
spark = SparkSession.builder.appName("q1-jtl22").getOrCreate()
```

In order to read the file `mushrooms.csv` we first need to know the `hdfs_path`. We can get it via executing the following command from Athena: `hdfs getconf -confKey fs.defaultFS` returning `hdfs://howitzer:9000/`. Now, we can read the file directly from `HDFS` and load the data in PySpark as a `DataFrame`:

```python
HDFS_PATH = "hdfs://howitzer:9000/shared_data"
```

```python
df = (
    spark.read.option("header", True)
    .option("inferSchema", True)
    .option("delimiter", ";")
    .csv(join(HDFS_PATH, "mushrooms.csv"))
)
```

We can look at the schema

```python
df.printSchema()
```

```
root
 |-- class: string (nullable = true)
 |-- cap-diameter: double (nullable = true)
 |-- cap-color: string (nullable = true)
 |-- stem-height: double (nullable = true)
 |-- stem-width: double (nullable = true)
```

Part a

Number of mushrooms having `'stem-height'` equal to zero. From the schema we can see that `'stem-height'` is of type `double`, therefore it is ok to directly compare with the number `0`.

```python
zero_stem_height = df.filter(col("stem-height") == 0).count()
zero_stem_height
```

```
1059
```

Part b

In order to calculate the first (0.25) and third (0.75) exact quartiles of `'stem-width'` for mushrooms such that `'class'` is `'e'` and `'cap-color'` is `'b'`, we are first filtering the DataFrame by the specified `class` and `cap-color`. After that, `approxQuantile` is used to calculate the quartiles. Setting `relativeError` to `0` gives us the exact quartiles, and setting `probabilities` to `[0.25, 0.75]` specifies the first and third quartile, allowing for calculation in one function call.

```python
df_stem_width = (
    df.filter(col("class") == "e").filter(col("cap-color") == "b").select("stem-width")
)
first_quartile, third_quartile = df_stem_width.approxQuantile(
    col="stem-width", probabilities=[0.25, 0.75], relativeError=0
)
```

```python
print(f"The first quartile is {first_quartile}.")
print(f"The third quartile is {third_quartile}.")
```

```
The first quartile is 9.45.
The third quartile is 19.38.
```

Part c

```python
# checking there are no NA values
df.count() == df.na.drop().count()
```

```
True
```

```python
# checking there are no negative values in the columns stem-width and stem-height
df.filter((col("stem-width") < 0) | (col("stem-height") < 0)).count()
```

```
0
```

The formula for calculating a geometric mean is $\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 x_2 ... x_n}$ or in logscale $\exp\left(\frac{1}{n}\sum_{i=1}^{n}\log x_i\right)$. We can use the logscale formula because we checked that all of our values are > 0.

```
# creating a df excluding 0 values for stem-width and stem-height
df_non_zero_height_width = df.filter(
    (col("stem-width") != 0) | (col("stem-height") != 0)
)

# calculating 'class'-specific geometric means of 'cap-diameter'
geometric_means = (
    df_non_zero_height_width.withColumn("log_cap_diameter", log(col("cap-diameter")))
    .groupBy("class")
    .agg(exp(expr("avg(log_cap_diameter)")).alias("cap-diameter"))
)
geometric_means.show()
```

```
+-----+-----------------+
|class|     cap-diameter|
+-----+-----------------+
|    e|6.234850623894219|
|    p|4.657240107404865|
+-----+-----------------+
```

```
# Closing the SparkSession
spark.stop()
```

Question 2

```
# Creating a Sparksession
spark = SparkSession.builder.appName("q2-jtl22").getOrCreate()
```

```
# paths to train and test files
train_path = join(HDFS_PATH, "clickbait_train.csv")
test_path = join(HDFS_PATH, "clickbait_test.csv")
```

Part a

```
# loading train and test data as DataFrames
df_train = (
    spark.read.option("header", False)
    .option("inferSchema", True)
    .option("delimiter", ";")
    .csv(train_path)
)
df_test = (
    spark.read.option("header", False)
    .option("inferSchema", True)
    .option("delimiter", ";")
```

```
    .csv(test_path)
)
```

```
# creating a column words containing arrays of string type elements
# because this format is needed for the CountVectorizer input
df_train = df_train.withColumn("words", split(df_train["_c1"], ","))
df_test = df_test.withColumn("words", split(df_test["_c1"], ","))

# renaming default name _c0 to label
df_test = df_test.withColumnRenamed("_c0", "label")
df_train = df_train.withColumnRenamed("_c0", "label")
```

```
# words to feature vectors
vect = CountVectorizer(inputCol="words", outputCol="words_features")
# combine words_features into a single vector
assembler = VectorAssembler(inputCols=["words_features"], outputCol="features")
# build NaiveBayes estimator object
nb = NaiveBayes(smoothing=1, modelType="multinomial")
# pipeline with transformers and estimator
pipeline = Pipeline(stages=[vect, assembler, nb])
# fit to training data
model = pipeline.fit(df_train)
```

```
# get predictions
preds = model.transform(df_test)
```

```
# evaluator, by default returns the area under reciever
evaluator = BinaryClassificationEvaluator()
auc = evaluator.evaluate(preds)
print(f"AUC is {auc}")
```

```
AUC is 0.790596210787321
```

Part b

```
# create df containing only the true values and predictions
predictions = preds.select("label", "prediction")

# define true/false positives/negatives
true_positives = predictions.filter(
    (predictions.label == 1) & (predictions.prediction == 1)
).count()
true_negatives = predictions.filter(
    (predictions.label == 0) & (predictions.prediction == 0)
).count()
false_positives = predictions.filter(
    (predictions.label == 0) & (predictions.prediction == 1)
).count()
false_negatives = predictions.filter(
    (predictions.label == 1) & (predictions.prediction == 0)
```

```
).count()

# calculate sensitivity, specificity, precision and accuracy
sensitivity = true_positives / (true_positives + false_negatives)
specifcity = true_negatives / (true_negatives + false_positives)
precision = true_positives / (true_positives + false_positives)
accuracy = (true_positives + true_negatives) / (
    true_positives + true_negatives + false_positives + false_negatives
)

print(f"Sensitivity is {sensitivity}")
print(f"Specifcity is {specifcity}")
print(f"Precision is {precision}")
print(f"Accuracy is {accuracy}")
```

```
Sensitivity is 0.4568273092369478
Specifcity is 0.8916256157635468
Precision is 0.5796178343949044
Accuracy is 0.7844592922543925
```

Part c

```
# get list of unique words from the train set
unique_train_words = (
    df_train.select(explode("words").alias("exploded")).groupby("exploded").count()
)
list_unique_train_words = unique_train_words.select(collect_set("exploded")).first()[0]
unique_train_words_set = set(list_unique_train_words)


# count of words in the train set for each label
nk = df_train.groupBy("label").agg(sum(size("words")).alias("nk"))
nk_0 = nk.filter(nk.label == 0).select("nk").first()[0]
nk_1 = nk.filter(nk.label == 1).select("nk").first()[0]
```

```
# defining UDFs

def count_words(
    row_words: List[str], unique_train_words=unique_train_words_set
) -> int:
    """Counts the new and old words (not unique) for each tweet.
    """
    new_words_counter = 0
    for word in row_words:
        if word not in unique_train_words:
            new_words_counter += 1

    old_words_counter = len(row_words) - new_words_counter
    return new_words_counter, old_words_counter

def old_words_correction_constant(old_words_count, alpha=1, v_new=27_017, v_old=25_823):
```

```python
    """Calcluates a correction constant for each tweet related to
    old words being calculated with V not including new test words,
    which later should be added to the rawPrediction columns
    in order to generate the correctedRawPrediction column.
    Note: this doesn't correct the tweets with no new words.
    """
    l_alpha = math.log(alpha)

    alpha_v_n0_old = alpha * v_old + nk_0
    alpha_v_n1_old = alpha * v_old + nk_1

    alpha_v_n0_new = alpha * v_new + nk_0
    alpha_v_n1_new = alpha * v_new + nk_1

    correction_0 = old_words_count * (
        math.log(alpha_v_n0_old) - math.log(alpha_v_n0_new)
    )
    correction_1 = old_words_count * (
        math.log(alpha_v_n1_old) - math.log(alpha_v_n1_new)
    )

    return [correction_0, correction_1]


def new_words_correction_constant(new_words_count, alpha=1, v=27017):
    """Calcluates a correction constant for each tweet related to
    having new words in the test set not available in the train set,
    which later should be added to the rawPrediction columns
    in order to generate the correctedRawPrediction column.
    """
    l_alpha = math.log(alpha)

    alpha_v_n0 = alpha * v + nk_0
    alpha_v_n1 = alpha * v + nk_1

    correction_0 = new_words_count * (l_alpha - math.log(alpha_v_n0))
    correction_1 = new_words_count * (l_alpha - math.log(alpha_v_n1))

    return [correction_0, correction_1]


def corrected_prediction(corrected_raw):
    """Compares the two values of correctedRawPrediction,
    and returns a binary value, which is equivalent to
    deriving the final corrected prediction.
    """
    return int(corrected_raw[0] < corrected_raw[1])

def vector_sum(arr):
    """Sum of vectors, accepts arrays.
    """
    return Vectors.dense(np.sum(arr))
```

```
# register the UDFs with the required return types
count_words_udf = udf(count_words, ArrayType(IntegerType()))
new_words_correction_constant_udf = udf(
  new_words_correction_constant, ArrayType(DoubleType())
)
old_words_correction_constant_udf = udf(
  old_words_correction_constant, ArrayType(DoubleType())
)
corrected_prediction_udf = udf(corrected_prediction, IntegerType())
vector_sum_udf = udf(vector_sum, VectorUDT())
```

```
# creating the column words_count
preds = preds.withColumn("words_count", count_words_udf(col("words")))
# creating the column new_words_count
preds = preds.withColumn("new_words_count", preds.words_count[0])
# creating the column old_words_count
preds = preds.withColumn("old_words_count", preds.words_count[1])
```

```
# creating the column new_words_correction_constant later to be added to rawPrediction
preds = preds.withColumn(
    "new_words_correction_constant",
    new_words_correction_constant_udf(col("new_words_count")),
)
# creating the column old_words_correction_constant later to be added to rawPrediction
preds = preds.withColumn(
    "old_words_correction_constant",
    old_words_correction_constant_udf(col("old_words_count")),
)
```

```
# convert rawPrediction from vector to array so a sum operation can be defined
preds = preds.withColumn(
    "rawPrediction", vector_to_array("rawPrediction").alias("rawPrediction")
)
# now we can create the column correctedRawPrediction
preds = preds.withColumn(
    "correctedRawPrediction",
    expr(
        ("transform(rawPrediction, (x, i) -> x + new_words_correction_constant[i]"
        "+ old_words_correction_constant[i])")
    ),
)
# create correctedPrediction column
preds = preds.withColumn(
    "correctedPrediction", corrected_prediction_udf(col("correctedRawPrediction"))
)
```

Evaluation:

```
# selecting only the true labels and corrected
predictions = preds.select("label", "correctedPrediction")

# define true/false positives/negatives
true_positives = predictions.filter(
```

```
    (predictions.label == 1) & (predictions.correctedPrediction == 1)
).count()
true_negatives = predictions.filter(
    (predictions.label == 0) & (predictions.correctedPrediction == 0)
).count()
false_positives = predictions.filter(
    (predictions.label == 0) & (predictions.correctedPrediction == 1)
).count()
false_negatives = predictions.filter(
    (predictions.label == 1) & (predictions.correctedPrediction == 0)
).count()

# calculate sensitivity, specificity, precision and accuracy
sensitivity = true_positives / (true_positives + false_negatives)
specfcity = true_negatives / (true_negatives + false_positives)
precision = true_positives / (true_positives + false_positives)
accuracy = (true_positives + true_negatives) / (
    true_positives + true_negatives + false_positives + false_negatives
)

print(f"Sensitivity is {sensitivity}")
print(f"Specfcity is {specfcity}")
print(f"Precision is {precision}")
print(f"Accuracy is {accuracy}")
```

```
Sensitivity is 0.5040160642570282
Specfcity is 0.8545155993431856
Precision is 0.5312169312169313
Accuracy is 0.7681267013115566
```

```
predictions = preds.select("label","correctedRawPrediction")
# converting correctedRawPrediction to vector so it can be passed to the evaluator
predictions = predictions.withColumn(
  "correctedRawPrediction", array_to_vector("correctedRawPrediction")
)
# evaluator, by default returns the area under reciever
evaluator = BinaryClassificationEvaluator(rawPredictionCol="correctedRawPrediction")
auc = evaluator.evaluate(predictions)
print(f"The AUC is {auc}")
```

```
The AUC is 0.7982081033493593
```

```
# Closing the Sparksession
spark.stop()
```