School of Business
University of Dundee

Module: AC51047

Advanced Big Data Analysis
Assignment 02

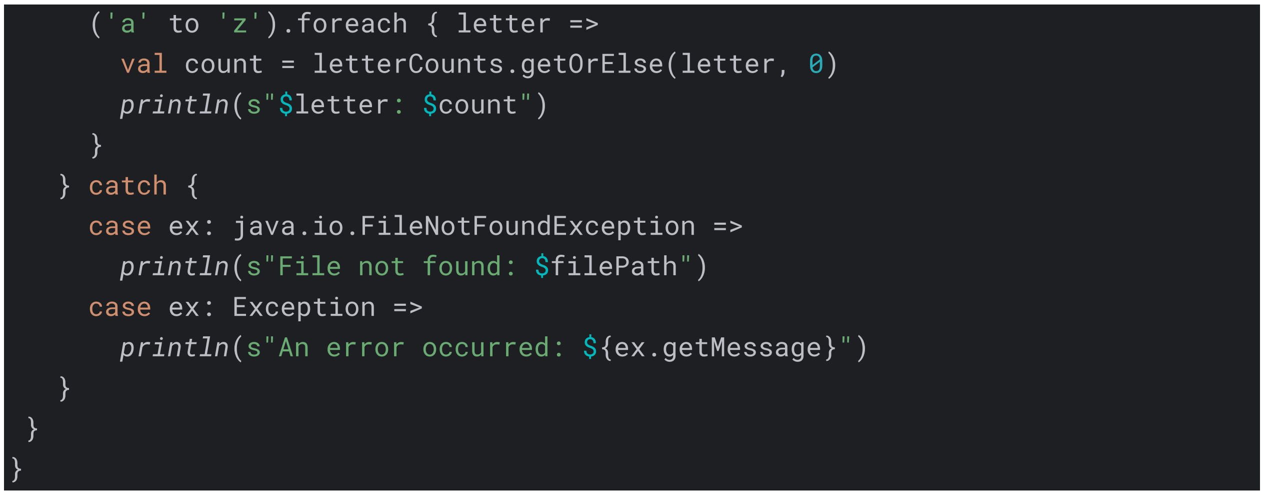
Submitted by
Ana Das
2622436

Dated: 10/12/2024

Question 1:

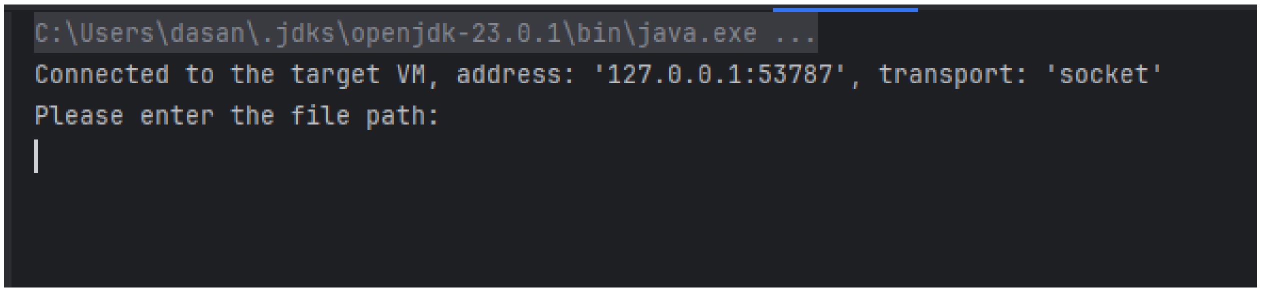
Write a Scala program that reads in a file (it can be either given as an argument to the main method, or hardcoded into it) that counts the number of occurrences of each letter a-z in the file and prints these counts on the screen.

```scala
// import scala.io.BufferedSource
import java.io.FileInputStream
import java.io.File
import scala.io
import scala.io.Source

object LetterFrequencyCounter {
 def main(args: Array[String]): Unit = {
   val filePath = if (args.isEmpty || args(0).isEmpty) {
     println("Please enter the file path:")
     val userInput = scala.io.StdIn.readLine().trim
     if (userInput.isEmpty) {
       println("No input provided. Using a hardcoded file path.")
       "\\Users\\dasan\\OneDrive\\Desktop\\Data Analysis\\Potato.txt"
     } else {
       userInput
     }
   } else {
     args(0)
   }
   println(s"Using file path: $filePath")

   try {
     val fileContent =
scala.io.Source.fromFile(filePath).getLines.mkString.toLowerCase
     val letterCounts = fileContent
       .filter(_.isLetter)
       .groupBy(identity)
       .view
       .mapValues(_.length)
       .toMap


     println("Letter frequencies:")
```



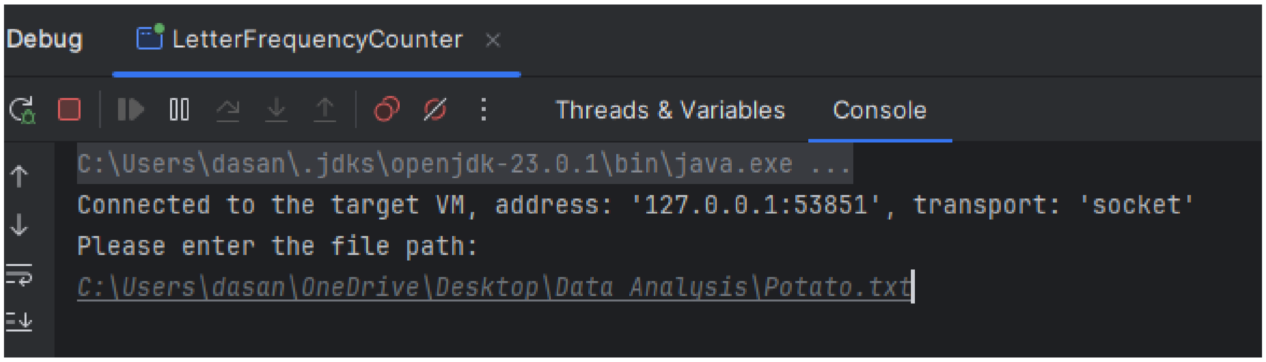
```
      ('a' to 'z').foreach { letter =>
        val count = letterCounts.getOrElse(letter, 0)
        println(s"$letter: $count")
      }
    } catch {
      case ex: java.io.FileNotFoundException =>
        println(s"File not found: $filePath")
      case ex: Exception =>
        println(s"An error occurred: ${ex.getMessage}")
    }
  }
}
```

On debug:



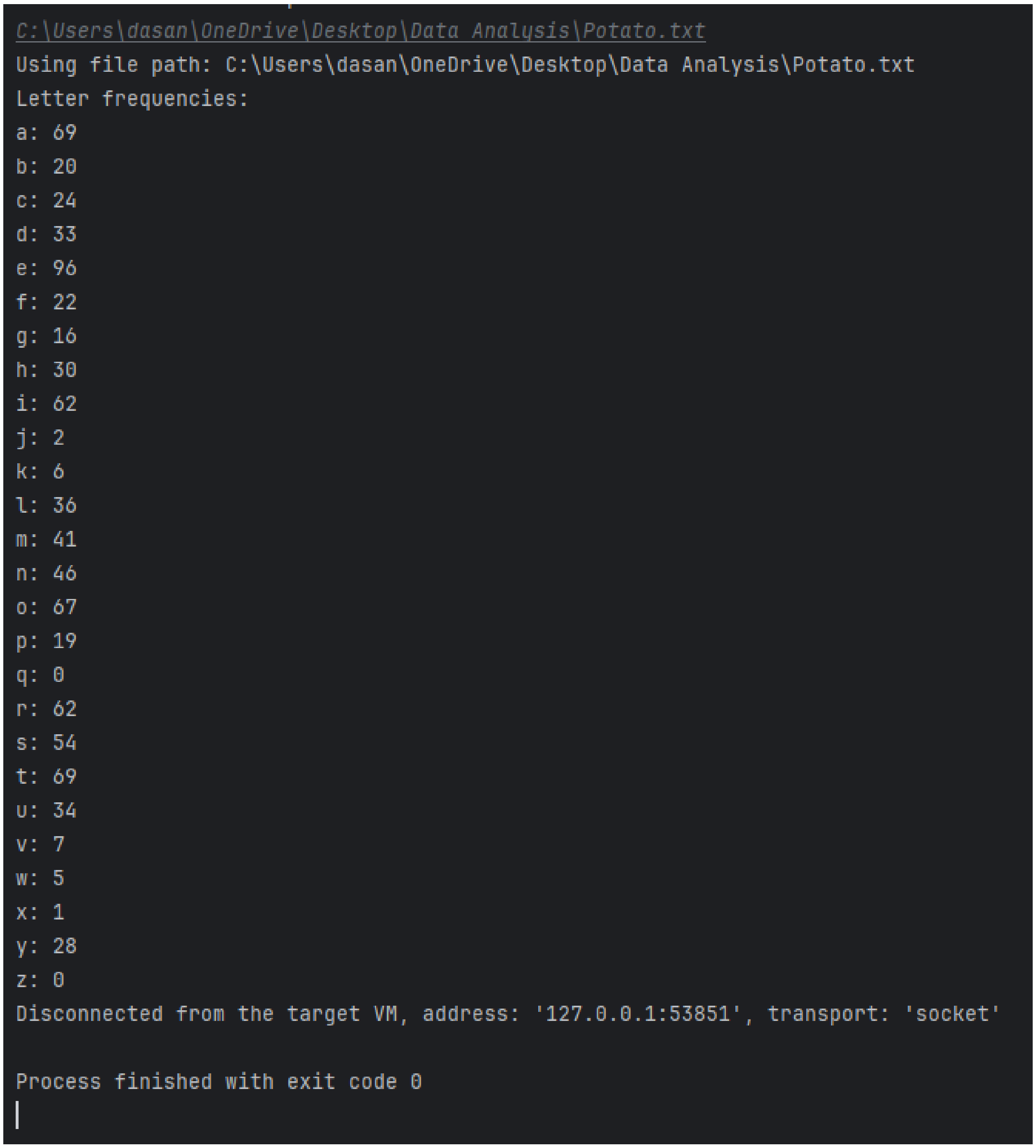
```
C:\Users\dasan\.jdks\openjdk-23.0.1\bin\java.exe ...
Connected to the target VM, address: '127.0.0.1:53787', transport: 'socket'
Please enter the file path:
```

Paste the Path and remove “”:



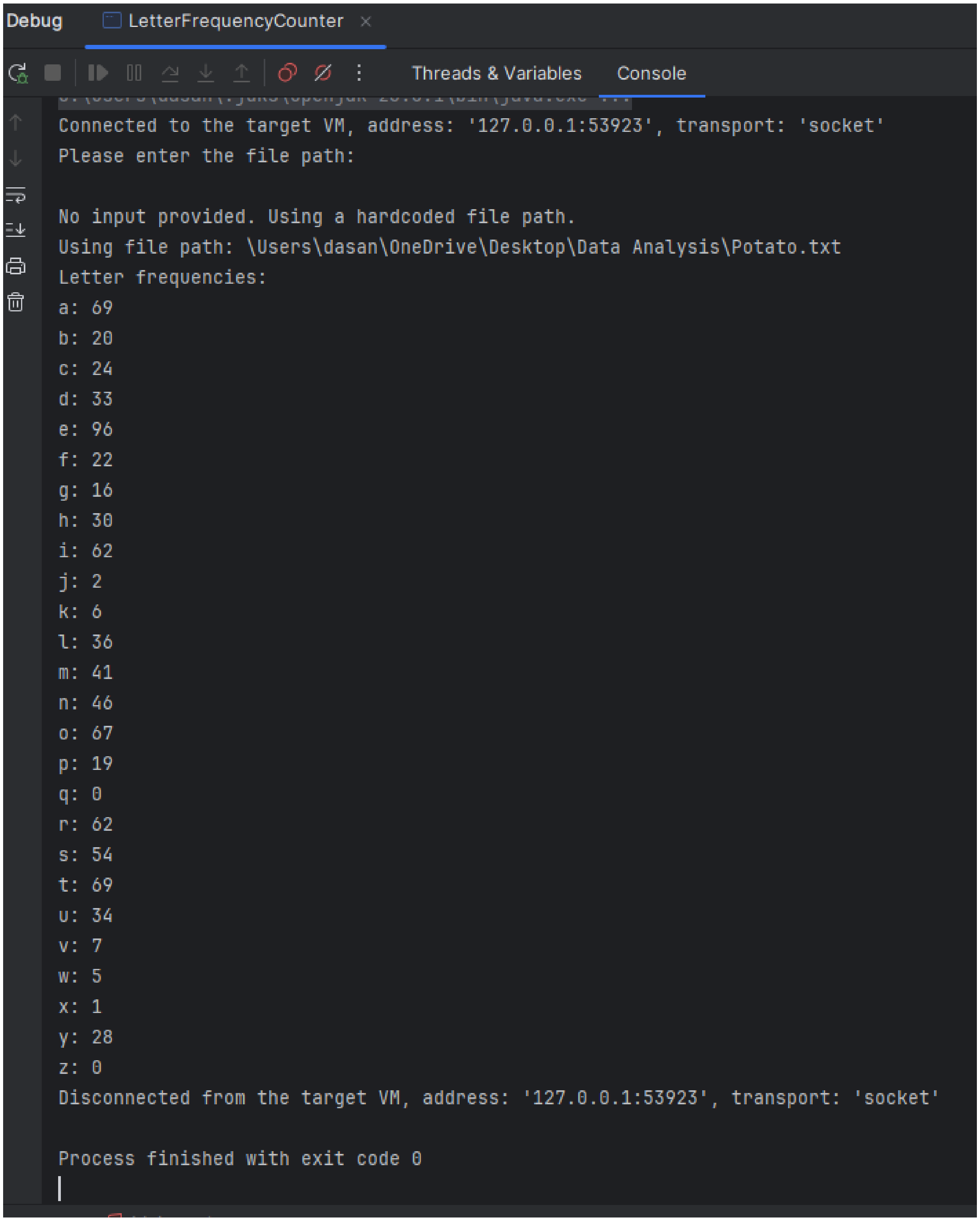
```
C:\Users\dasan\.jdks\openjdk-23.0.1\bin\java.exe ...
Connected to the target VM, address: '127.0.0.1:53851', transport: 'socket'
Please enter the file path:
C:\Users\dasan\OneDrive\Desktop\Data Analysis\Potato.txt
```

Output:



```
C:\Users\dasan\OneDrive\Desktop\Data Analysis\Potato.txt
Using file path: C:\Users\dasan\OneDrive\Desktop\Data Analysis\Potato.txt
Letter frequencies:
a: 69
b: 20
c: 24
d: 33
e: 96
f: 22
g: 16
h: 30
i: 62
j: 2
k: 6
l: 36
m: 41
n: 46
o: 67
p: 19
q: 0
r: 62
s: 54
t: 69
u: 34
v: 7
w: 5
x: 1
y: 28
z: 0
Disconnected from the target VM, address: '127.0.0.1:53851', transport: 'socket'

Process finished with exit code 0
```

Incase, the user do not enter the path:



Debug | LetterFrequencyCounter | Threads & Variables | Console

```
Connected to the target VM, address: '127.0.0.1:53923', transport: 'socket'
Please enter the file path:

No input provided. Using a hardcoded file path.
Using file path: \Users\dasan\OneDrive\Desktop\Data Analysis\Potato.txt
Letter frequencies:
a: 69
b: 20
c: 24
d: 33
e: 96
f: 22
g: 16
h: 30
i: 62
j: 2
k: 6
l: 36
m: 41
n: 46
o: 67
p: 19
q: 0
r: 62
s: 54
t: 69
u: 34
v: 7
w: 5
x: 1
y: 28
z: 0
Disconnected from the target VM, address: '127.0.0.1:53923', transport: 'socket'

Process finished with exit code 0
```

Explanation:

`scala.io.Source`: Used to read files.

Main Method:

```
object LetterFrequencyCounter {
  def main(args: Array[String]): Unit = {
```

- `object LetterFrequencyCounter`: Defines a singleton object (a static class) for the program.
- `main(args: Array[String])`: The entry point of the Scala program. Accepts command-line arguments as an array of strings.

File Path Handling:

```
val filePath = if (args.isEmpty || args(0).isEmpty) {
```

- Checks if a file path is provided via command-line arguments:
  - If No Argument:
    - Prompts the user to input a file path.
    - If the user provides nothing, a hardcoded default path is used (`"\\Users\\dasan\\OneDrive\\Desktop\\Data Analysis\\Potato.txt"`).
  - If Argument Exists, uses the provided argument as the file path.

Note: The backslashes (`\\`) are escape sequences required for Windows

Debugging Information:

`println(s"Using file path: $filePath"):`

Prints the determined file path for debugging purposes, helping the user verify the file location being accessed.

Reading File Content:

```
val fileContent =
scala.io.Source.fromFile(filePath).getLines.mkString.toLowerCase
```

- `scala.io.Source.fromFile(filePath)`: Opens the file at the specified path.
- `getLines.mkString`: Reads all lines in the file and combines them into a single string.

- toLowerCase: Converts the content to lowercase to ensure case-insensitivity when counting letters.

Filtering and Counting Letters:

- filter(_.isLetter): Keeps only alphabetic characters, discarding digits, punctuation, and whitespace.
- groupBy(identity): Groups the remaining characters into categories (each unique letter becomes a group).
- view: Optimizes performance by creating a lazy map.
- mapValues(_.length): Counts the number of occurrences in each group.
- toMap: Converts the view back to a normal map for easier manipulation.

Printing Letter Frequencies:
```
println("Letter frequencies:")
('a' to 'z').foreach { letter =>
  val count = letterCounts.getOrElse(letter, 0)
println(s"$letter: $count")
}
```

- 'a' to 'z': Iterates over all lowercase letters from a to z.
- getOrElse(letter, 0): Retrieves the frequency of the current letter from the map; returns 0 if the letter is not found.
- println(s"$letter: $count"): Prints the letter and its frequency

Error Handling:
```
catch {
  case ex: java.io.FileNotFoundException =>
    println(s"File not found: $filePath")
  case ex: Exception =>
    println(s"An error occurred: ${ex.getMessage}")
}
```

try: Encapsulates the file-reading and processing logic to handle errors. FileNotFoundException: Catches cases where the specified file does not exist. Generic Exception: Catches all other errors and prints the error message.

Question 2:

Given is a dataset in the table Bakery.csv that contains transactions in the bakery. Each row in the transaction contains transaction identifier, item bought, date and time, daytime (morning/afternoon) and daytype (weekend, weekday). Write Scala or Python code for Spark that will derive frequent itemsets of bakery items that are frequently bought together and association rules from this data set.

```
from pyspark.ml.fpm import FPGrowth

dataFrame = spark.read.csv("/home/dasan/IdeaProjects/Bakery.csv", header=True).rdd

keyValuePairs = dataFrame.map (lambda row : (row['TransactionNo'], {row['Items']}))

collectedKeyValuePairs = keyValuePairs.reduceByKey( lambda a,b : a.union(b) )

itemsToList=collectedKeyValuePairs.map(lambda x : (x[0], list(x[1])))

transactionsFrame=spark.createDataFrame(itemsToList, ["id","items"])

model = FPGrowth(itemsCol="items", minSupport=0.03, minConfidence=0.3)

modelResults = model.fit(transactionsFrame)

modelResults.associationRules.show(20, False)

modelResults.freqItemsets.show(20, False)
```

Explanation:

`from pyspark.ml.fpm import FPGrowth` This imports the implementation of the `FPGrowth` model from the PySpark MLlib library, allowing you to use it in the code.

Here, we have fetched the file from the local drive instead of S3 bucket.

Reading File:
`dataFrame = spark.read.csv("/home/dasan/IdeaProjects/Bakery.csv", header=True).rdd`

This reads the CSV file *“Bakery.csv”* into a Spark RDD named `dataFrame`. The `header=True` argument of read.csv specifies that the first row in the CSV file contains column names. The .rdd converts the resulting DataFrame into an RDD for further processing.
Now we have an RDD dataFrame with named columns. We are interested in columns *'TransactionNo'*, which is an identifier of a transaction, and *‘Items’*, which is the description of the item bought in that transaction. We want to group all the items corresponding to a single transaction. These will be the transaction items that we will use as an input to our FPGrowth model.

Mapping to Key-Value Pairs:

```
keyValuePairs = dataFrame.map(lambda row: (row['TransactionNo'],
{row['Items']}))
```

This maps each row of `dataFrame` into a key-value pair. The key is the `TransactionNo` column (transaction identifier). The value is a set containing the `Items` column (item(s) bought in the transaction). The use of a set ensures there are no duplicate items within a transaction, which aligns with the FPGrowth algorithm's requirement.

Reducing by Key:

```
collectedKeyValuePairs = keyValuePairs.reduceByKey(lambda a, b:
a.union(b))
```

Groups all key-value pairs with the same transaction identifier (`TransactionNo`). Uses the `reduceByKey` operation to merge sets of items for each transaction into a single set using the `union` operation. This step ensures all items in a transaction are grouped into one set. We do not want duplicate items in our set of items bought in each transaction, as this breaks the assumptions of the FPGrowth model, that each item in a transaction appears only once. Our new data frame that contains key-value pairs, where key is a transaction identifier and value is the set of all items bought in that transactions is now `collectedKeyValuePairs`.

Converting Sets to Lists:

```
itemsToList = collectedKeyValuePairs.map(lambda x: (x[0], list(x[1])))
```

Creates yet another data frame, with key-value pairs, which will be identical to our previous data frame (`collectedKeyValuePairs`) and converts the values (sets of items) into lists. This is necessary because the FPGrowth model expects the items in each transaction to be in list format rather than a set.

Creating a DataFrame:

```
transactionsFrame = spark.createDataFrame(itemsToList, ["id",
"items"])
```

Creates a new Spark DataFrame transactionsFrame with two columns:

id: Transaction identifier.

items: List of items bought in that transaction.

This format matches the input requirements for the FPGrowth model. So we simply create a new Spark RDD based on the `itemsToList` data frame, that will have exactly the same data, but split into named columns.

Creating and Fitting the FPGrowth Model:

```
model = FPGrowth(itemsCol="items", minSupport=0.03, minConfidence=0.3)
modelResults = model.fit(transactionsFrame)
```

Creating the Model:

- `itemsCol="items"`: Specifies the column containing the items.
- `minSupport=0.03`: Minimum support value (an itemset must appear in at least 3% of all transactions to be considered frequent).
- `minConfidence=0.3`: Minimum confidence value (for association rules to be considered valid).

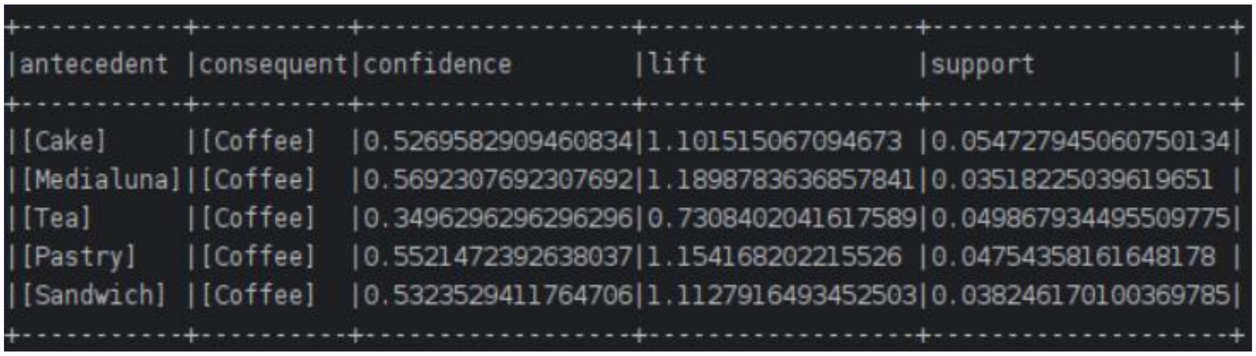
Fitting the Model: Applies the FPGrowth model to the `transactionsFrame` DataFrame, generating frequent itemsets and association rules.

Result:

```
modelResults.associationRules.show(20, False)
modelResults.freqItemsets.show(20, False)
```

associationRules.show: Displays the discovered association rules, showing how frequently items are bought together. The parameter `20` specifies that up to 20 rules will be shown. The `False` flag disables truncation of the results.

freqItemsets.show: Displays the frequent itemsets, i.e., groups of items that frequently appear together in transactions.



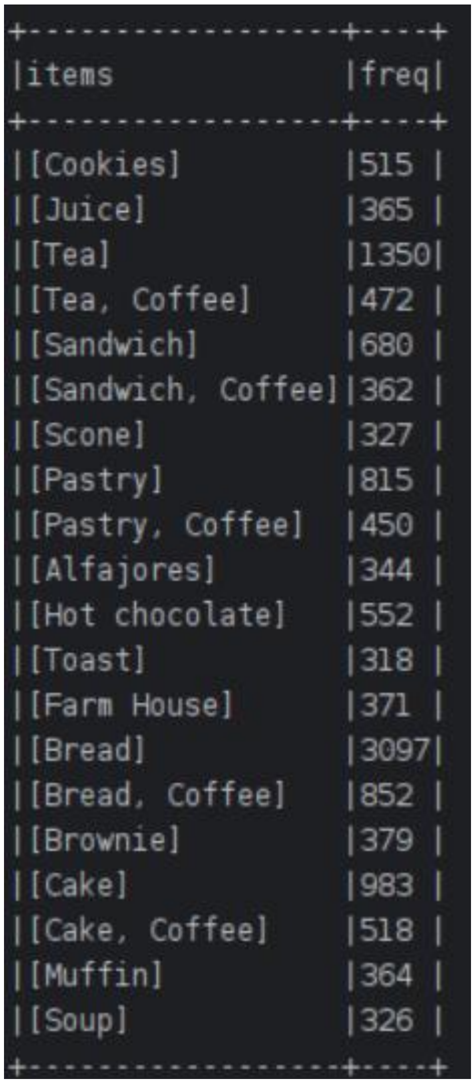
```
+-----------+----------+------------------+------------------+--------------------+
|antecedent |consequent|confidence        |lift              |support             |
+-----------+----------+------------------+------------------+--------------------+
|[Cake]     |[Coffee]  |0.5269582909460834|1.101515067094673 |0.05472794506075013 4|
|[Medialuna]|[Coffee]  |0.5692307692307692|1.1898783636857841|0.0351822503961965 1 |
|[Tea]      |[Coffee]  |0.3496296296296296|0.7308402041617589|0.04986793449550977 5|
|[Pastry]   |[Coffee]  |0.5521472392638037|1.154168202215526 |0.0475435816164817 8 |
|[Sandwich] |[Coffee]  |0.5323529411764706|1.1127916493452503|0.03824617010036978 5|
+-----------+----------+------------------+------------------+--------------------+
```

Key Metrics:
Antecedent: The item(s) whose purchase predicts the consequent.

1. Consequent: The item(s) predicted to be purchased together with the antecedent.
2. Confidence: The likelihood that the consequent is purchased when the antecedent is purchased.
    - Example: For [Cake] -> [Coffee], confidence = 0.526 (about 52.7%), meaning 52.7% of transactions containing Cake also contain Coffee.
3. Lift: Measures how much more often the antecedent and consequent appear together compared to if they were independent.
    - Lift > 1 indicates a strong positive association.
    - Lift ≈ 1 suggests no association.
    - Example: [Pastry] -> [Coffee] has a lift of 1.15, meaning they are 15% more likely to be bought together than by chance.
4. Support: Fraction of transactions where both antecedent and consequent appear together.

Interpretation:

- The strongest association is [Medialuna] -> [Coffee], with a confidence of 56.9% and a lift of 1.18, indicating customers who buy Medialuna are highly likely to buy Coffee.
- [Tea] -> [Coffee] has a low confidence of 34.9% and lift < 1, suggesting this pair is less correlated.



```
+------------------+----+
|items             |freq|
+------------------+----+
|[Cookies]         |515 |
|[Juice]           |365 |
|[Tea]             |1350|
|[Tea, Coffee]     |472 |
|[Sandwich]        |680 |
|[Sandwich, Coffee]|362 |
|[Scone]           |327 |
|[Pastry]          |815 |
|[Pastry, Coffee]  |450 |
|[Alfajores]       |344 |
|[Hot chocolate]   |552 |
|[Toast]           |318 |
|[Farm House]      |371 |
|[Bread]           |3097|
|[Bread, Coffee]   |852 |
|[Brownie]         |379 |
|[Cake]            |983 |
|[Cake, Coffee]    |518 |
|[Muffin]          |364 |
|[Soup]            |326 |
+------------------+----+
```

Key Metrics:

1. Items: The combination of items in a transaction.
2. Frequency: The number of transactions containing the itemset.

Interpretation:

[Bread] is the most frequent item, appearing in 3,097 transactions. [Bread, Coffee] appears together in 852 transactions, making it one of the most frequent pairs. [Tea, Coffee] and [Pastry, Coffee] also appear frequently together, with 472 and 450 occurrences, respectively.

Insights and Recommendations:

1. **Strong Recommendations**: If a customer buys [Cake], suggest [Coffee], as they are frequently bought together (confidence = 52.6%). [Pastry] buyers are likely to add [Coffee] as well (confidence = 55.2%). [Medialuna] buyers are the most likely to also buy [Coffee] (confidence = 56.9%).
2. **Promotions or Bundles**: Consider bundling [Bread] and [Coffee] due to their high frequency together (852 transactions). [Cake] and [Coffee] also make a great promotional pair.
3. **Inventory Planning**: Ensure [Bread], [Coffee], [Tea], and [Cake] are always well-stocked since they dominate transaction volumes.

Question 3:

Iris dataset (Iris.csv) is a dataset that contains measurement of iris flowers and their species classification. Write code in Scala or Python for Spark that will use Decision Tree Classification for predicting the flower species based on the four measurement of iris flowers.

```python
from pyspark.sql import SparkSession
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType

spark = SparkSession.builder.appName("IrisClassification").getOrCreate()


df = spark.read.option("inferSchema", "true").csv("/home/dasan/IdeaProjects/iris.csv",
header=True)

assembler = VectorAssembler(
   inputCols=["sepal_length", "sepal_width", "petal_length", "petal_width"],
   outputCol="features"
)
df = assembler.transform(df)


species_indexer = StringIndexer(inputCol="variety", outputCol="label")
species_model = species_indexer.fit(df)
df = species_model.transform(df)


(trainingData, testData) = df.randomSplit([0.7, 0.3])
```

```
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features", maxDepth=5)
model = dt.fit(trainingData)


predictions = model.transform(testData)


index_to_species = {i: label for i, label in enumerate(species_model.labels)}
species_udf = udf(lambda x: index_to_species[int(x)], StringType())

predictions = predictions.withColumn("predicted_species",
species_udf(predictions["prediction"]))


mismatches = predictions.filter(col("variety") != col("predicted_species"))
print("Rows where actual variety differs from predicted species:")
mismatches.select("features", "variety", "predicted_species").show()


testErr = mismatches.count() / float(testData.count())
print(f"Test Error: {testErr}")


print("Learned classification tree model:")
print(model.toDebugString)
```

Explanation:

Setup Spark:

`spark = SparkSession.builder.appName("IrisClassification").getOrCreate()`

This initializes a Spark session with the application name `"IrisClassification"`. A `SparkSession` is the entry point for using Spark SQL and DataFrame APIs.

`df = spark.read.option("inferSchema", "true").csv("/home/dasan/IdeaProjects/iris.csv", header=True)`

This reads the CSV file containing the Iris dataset into a Spark DataFrame. `inferSchema="true"` automatically infers the data types of columns. `header=True` treats the first row of the file as column headers.

Assemble Features:

```
assembler = VectorAssembler(
     inputCols=["sepal_length", "sepal_width", "petal_length",
"petal_width"],
     outputCol="features"
)
df = assembler.transform(df)
```

This creates a `VectorAssembler` to combine the feature columns (`sepal_length`, `sepal_width`, `petal_length`, `petal_width`) into a single vector column named `"features"`. The `transform` method applies the assembler, adding the `"features"` column to the DataFrame.

Index Species Column:

```
species_indexer = StringIndexer(inputCol="variety", outputCol="label")
species_model = species_indexer.fit(df)
df = species_model.transform(df)
```

`StringIndexer` encodes the `variety` (flower species) column into numeric labels stored in a new column called "label". `fit` creates the indexer model using the input DataFrame, mapping each species to a unique number. `transform` adds the "label" column to the DataFrame.

```
(trainingData, testData) = df.randomSplit([0.7, 0.3])
```

This splits the dataset into two parts.

trainingData: 70% of the data, used to train the model.

testData: 30% of the data, used to evaluate the model.

Train the Decision Tree Classifier:

```
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features",
maxDepth=5)
model = dt.fit(trainingData)
```

`labelCol="label"` specifies the target column. `featuresCol="features"` specifies the input features and `maxDepth=5` limits the depth of the tree to avoid overfitting. `fit` trains the model using the training data.

Prediction Generation:

```
predictions = model.transform(testData)
```

This applies the trained model to the testData, generating predictions. Adds a new column, "prediction", containing the predicted numeric label.

Mapping:

```
index_to_species = {i: label for i, label in
enumerate(species_model.labels)}
species_udf = udf(lambda x: index_to_species[int(x)], StringType())
predictions = predictions.withColumn("predicted_species",
species_udf(predictions["prediction"]))
```

Maps numeric predictions (`prediction` column) back to species names using the `StringIndexer` labels. A user-defined function (`species_udf`) is created to perform the mapping. `withColumn` adds a new column `"predicted_species"` containing the species name corresponding to the prediction.

Filter and Display Mismatched Predictions:

```
mismatches = predictions.filter(col("variety") !=
col("predicted_species"))
print("Rows where actual variety differs from predicted species:")
mismatches.select("features", "variety", "predicted_species").show()
```

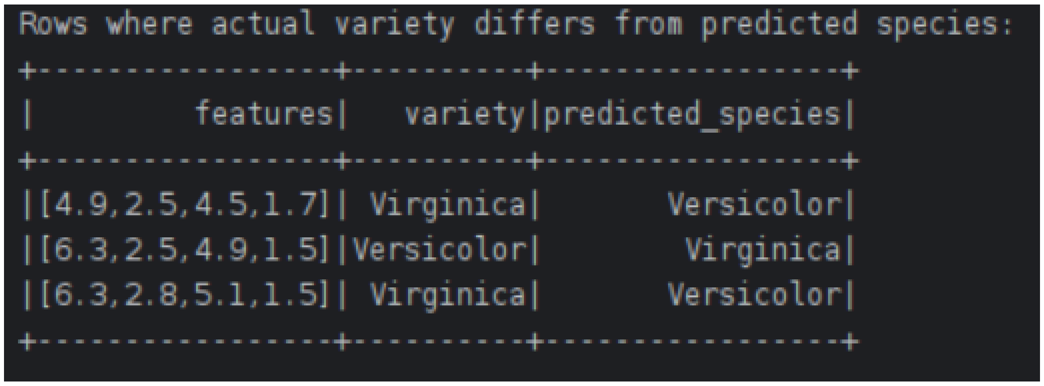
Filters the DataFrame to retain rows where the actual species (`variety`) differs from the predicted species (`predicted_species`). Displays these rows, including the feature vector, actual species, and predicted species.

Test Error calculation:

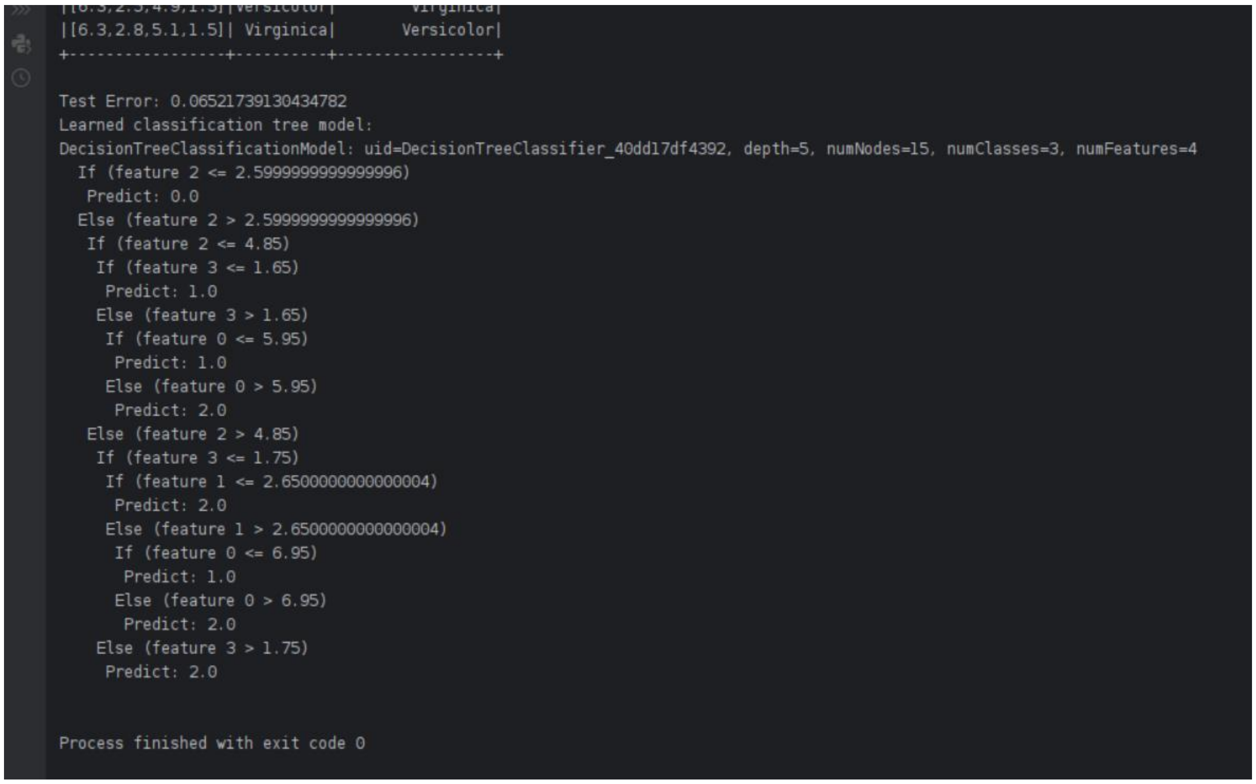
```
testErr = mismatches.count() / float(testData.count())
print(f"Test Error: {testErr}")
```

The test error is calculated as the ratio of mismatched rows to the total number of rows in the test set.

Output:



```
Rows where actual variety differs from predicted species:
+-----------------+----------+----------------+
|         features|   variety|predicted_species|
+-----------------+----------+----------------+
|[4.9,2.5,4.5,1.7]| Virginica|      Versicolor|
|[6.3,2.5,4.9,1.5]|Versicolor|       Virginica|
|[6.3,2.8,5.1,1.5]| Virginica|      Versicolor|
+-----------------+----------+----------------+
```



```
|[6.3,2.5,4.9,1.5]|Versicolor|       Virginica|
|[6.3,2.8,5.1,1.5]| Virginica|      Versicolor|
+-----------------+----------+----------------+

Test Error: 0.06521739130434782
Learned classification tree model:
DecisionTreeClassificationModel: uid=DecisionTreeClassifier_40dd17df4392, depth=5, numNodes=15, numClasses=3, numFeatures=4
  If (feature 2 <= 2.5999999999999996)
   Predict: 0.0
  Else (feature 2 > 2.5999999999999996)
   If (feature 2 <= 4.85)
    If (feature 3 <= 1.65)
     Predict: 1.0
    Else (feature 3 > 1.65)
     If (feature 0 <= 5.95)
      Predict: 1.0
     Else (feature 0 > 5.95)
      Predict: 2.0
   Else (feature 2 > 4.85)
    If (feature 3 <= 1.75)
     If (feature 1 <= 2.6500000000000004)
      Predict: 2.0
     Else (feature 1 > 2.6500000000000004)
      If (feature 0 <= 6.95)
       Predict: 1.0
      Else (feature 0 > 6.95)
       Predict: 2.0
    Else (feature 3 > 1.75)
     Predict: 2.0


Process finished with exit code 0
```