Iprompt39 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Word Count with Custom Delimiter\*\*:

- \*\*Task\*\*: Implement a Spark job to count words in a text file where words are separated by a custom delimiter.

- \*\*Input\*\*: Text file and a delimiter.

- \*\*Output\*\*: Word count.

"""

Below is a Spark job implementation for the "Word Count with Custom Delimiter" task. This implementation includes some intentional mistakes to test the interviewee's understanding of Spark.

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import split, explode, col

def word\_count(file\_path, delimiter):

# Initialize Spark session

spark = SparkSession.builder \

.appName("WordCountWithCustomDelimiter") \

.getOrCreate()

# Read text file into DataFrame

lines = spark.read.text(file\_path)

# Split lines into words using the custom delimiter

words = lines.select(explode(split(col("value"), delimiter)).alias("word"))

# Count occurrences of each word

word\_counts = words.groupBy("words").count() # Mistake: "words" should be "word"

# Show the result

word\_counts.show()

# Stop the Spark session

spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

file\_path = "path/to/your/textfile.txt" # Mistake: This should be passed as an argument

delimiter = "," # Mistake: This should be passed as an argument

word\_count(file\_path, delimiter)

```

### Mistakes Introduced:

1. \*\*Column Name in groupBy\*\*:

- Mistake: `word\_counts = words.groupBy("words").count()`

- Correction: The correct column name should be `word` not `words`.

2. \*\*Hardcoded File Path and Delimiter\*\*:

- Mistake: `file\_path = "path/to/your/textfile.txt"` and `delimiter = ","`

- Correction: These should be passed as arguments to the script.

3. \*\*Dependency on DataFrame API\*\*:

- Mistake: Using the DataFrame API for word count is not a mistake per se, but you could challenge the interviewee to use RDD transformations instead.

### Corrected Code:

Here's the corrected version of the code:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import split, explode, col

import sys

def word\_count(file\_path, delimiter):

# Initialize Spark session

spark = SparkSession.builder \

.appName("WordCountWithCustomDelimiter") \

.getOrCreate()

# Read text file into DataFrame

lines = spark.read.text(file\_path)

# Split lines into words using the custom delimiter

words = lines.select(explode(split(col("value"), delimiter)).alias("word"))

# Count occurrences of each word

word\_counts = words.groupBy("word").count()

# Show the result

word\_counts.show()

# Stop the Spark session

spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

if len(sys.argv) != 3:

print("Usage: word\_count <file\_path> <delimiter>")

sys.exit(-1)

file\_path = sys.argv[1]

delimiter = sys.argv[2]

word\_count(file\_path, delimiter)

```

### Key Points to Test in an Interview:

1. \*\*Understanding of SparkSession and DataFrame API\*\*.

2. \*\*Correct usage of DataFrame transformations like `split` and `explode`\*\*.

3. \*\*Proper handling and passing of command-line arguments\*\*.

4. \*\*Correct usage of `groupBy` and `count` for aggregations\*\*.

5. \*\*Ability to debug and fix common mistakes\*\*.

Done

Iprompt40 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Top N Most Frequent Words\*\*:

- \*\*Task\*\*: Write a Spark job to find the top N most frequent words in a large text file.

- \*\*Input\*\*: Text file and integer N.

- \*\*Output\*\*: List of N most frequent words with their counts.

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Certainly! Below is a sample code that introduces some mistakes for the purpose of testing an interviewee's understanding of Apache Spark. The task is to detect anomalies in a time series dataset. The mistakes are intentionally added to test the candidate's debugging and problem-solving skills.

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, avg, stddev

# Initialize Spark session

spark = SparkSession.builder \

.appName("TimeSeriesAnomaliesDetection") \

.getOrCreate()

# Sample data creation (mistake: timestamp should be in correct format)

data = [

("2023-01-01 00:00:00", 10),

("2023-01-01 01:00:00", 12),

("2023-01-01 02:00:00", 19),

("2023-01-01 03:00:00", 15),

("2023-01-01 04:00:00", 100), # anomaly

("2023-01-01 05:00:00", 13),

("2023-01-01 06:00:00", 14),

]

# Convert the data to a DataFrame

df = spark.createDataFrame(data, ["timestamp", "value"])

# Calculate the mean and standard deviation (mistake: should group by timestamp)

stats = df.agg(

avg(col("value")).alias("mean"),

stddev(col("value")).alias("stddev")

).collect()[0]

mean\_value = stats["mean"]

stddev\_value = stats["stddev"]

# Define the threshold for anomalies (mistake: threshold calculation)

threshold = mean\_value + 2 \* stddev\_value

# Filter anomalies (mistake: incorrect column name)

anomalies = df.filter(col("value") > threshold)

# Show the anomalies

anomalies.show()

# Stop the Spark session

spark.stop()

```

### Mistakes Introduced:

1. \*\*Timestamp Format\*\*: The timestamp should be in a format that Spark can correctly interpret as a timestamp.

2. \*\*Grouping Issue\*\*: The aggregation for mean and standard deviation should be done in the context of time windows or other relevant groupings if necessary.

3. \*\*Threshold Calculation\*\*: The threshold calculation is simplified and may need to be more sophisticated depending on the use case.

4. \*\*Incorrect Column Name\*\*: The filter condition should reference the correct column name in the DataFrame.

5. \*\*Additional Context\*\*: Depending on the complexity, additional context such as windowing functions or more complex anomaly detection logic may be required.

### Corrected Version:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, avg, stddev, to\_timestamp

# Initialize Spark session

spark = SparkSession.builder \

.appName("TimeSeriesAnomaliesDetection") \

.getOrCreate()

# Sample data creation

data = [

("2023-01-01 00:00:00", 10),

("2023-01-01 01:00:00", 12),

("2023-01-01 02:00:00", 19),

("2023-01-01 03:00:00", 15),

("2023-01-01 04:00:00", 100), # anomaly

("2023-01-01 05:00:00", 13),

("2023-01-01 06:00:00", 14),

]

# Convert the data to a DataFrame

df = spark.createDataFrame(data, ["timestamp", "value"])

# Convert timestamp to proper format

df = df.withColumn("timestamp", to\_timestamp(col("timestamp")))

# Calculate the mean and standard deviation

stats = df.agg(

avg(col("value")).alias("mean"),

stddev(col("value")).alias("stddev")

).collect()[0]

mean\_value = stats["mean"]

stddev\_value = stats["stddev"]

# Define the threshold for anomalies

threshold = mean\_value + 2 \* stddev\_value

# Filter anomalies

anomalies = df.filter(col("value") > threshold)

# Show the anomalies

anomalies.show()

# Stop the Spark session

spark.stop()

```

This corrected version should work correctly, and the interviewee should be able to identify and fix the mistakes in the initial version.

Done

Iprompt41 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Join Two DataFrames\*\*:

- \*\*Task\*\*: Given two DataFrames representing sales and customer information, perform an inner join and output the result.

- \*\*Input\*\*: Two DataFrames (sales and customers).

- \*\*Output\*\*: Joined DataFrame.

"""

Certainly! Below is a sample code that creates an application to join two DataFrames (sales and customers) using Apache Spark. The code includes intentional mistakes that can be used to test the interviewee's understanding of Spark.

### Sample Code with Intentional Mistakes

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Initialize Spark session

spark = SparkSession.builder \

.appName("JoinTwoDataFrames") \

.getOrCreate()

# Sample data for sales DataFrame

sales\_data = [

(1, "2023-01-01", 100),

(2, "2023-01-01", 150),

(3, "2023-01-01", 200)

]

# Sample data for customers DataFrame

customers\_data = [

(1, "John Doe"),

(2, "Jane Smith"),

(4, "Emily Johnson")

]

# Define schema for sales DataFrame

sales\_schema = ["customer\_id", "date", "amount"]

# Define schema for customers DataFrame

customers\_schema = ["customer\_id", "customer\_name"]

# Create DataFrames from sample data

sales\_df = spark.createDataFrame(sales\_data, sales\_schema)

customers\_df = spark.createDataFrame(customers\_data, customers\_schema)

# Perform an inner join on the 'customer\_id' column

joined\_df = sales\_df.join(customers\_df, sales\_df.customer\_id == customers\_df.id, "inner")

# Select relevant columns to display

result\_df = joined\_df.select(col("customer\_id"), col("date"), col("amount"), col("customer\_name"))

# Show the result

result\_df.show()

# Stop the Spark session

spark.stop()

```

### Intentional Mistakes

1. \*\*Column Name Mismatch\*\*: In the join condition, `customers\_df.id` should be `customers\_df.customer\_id`.

2. \*\*DataFrame Column Selection\*\*: The `select` statement uses `col("customer\_id")` which is ambiguous if not properly aliased.

3. \*\*Schema Definition\*\*: The schema definitions use lists of strings instead of the correct `StructType` and `StructField` types.

4. \*\*Data Types\*\*: The schema does not enforce data types, which can lead to issues with data integrity.

### Corrected Code

Below is the corrected version of the code:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Initialize Spark session

spark = SparkSession.builder \

.appName("JoinTwoDataFrames") \

.getOrCreate()

# Sample data for sales DataFrame

sales\_data = [

(1, "2023-01-01", 100),

(2, "2023-01-01", 150),

(3, "2023-01-01", 200)

]

# Sample data for customers DataFrame

customers\_data = [

(1, "John Doe"),

(2, "Jane Smith"),

(4, "Emily Johnson")

]

# Define schema for sales DataFrame

sales\_schema = ["customer\_id", "date", "amount"]

# Define schema for customers DataFrame

customers\_schema = ["customer\_id", "customer\_name"]

# Create DataFrames from sample data

sales\_df = spark.createDataFrame(sales\_data, schema=sales\_schema)

customers\_df = spark.createDataFrame(customers\_data, schema=customers\_schema)

# Perform an inner join on the 'customer\_id' column

joined\_df = sales\_df.join(customers\_df, sales\_df.customer\_id == customers\_df.customer\_id, "inner")

# Select relevant columns to display

result\_df = joined\_df.select(sales\_df.customer\_id, col("date"), col("amount"), col("customer\_name"))

# Show the result

result\_df.show()

# Stop the Spark session

spark.stop()

```

This corrected code fixes the mistakes and should work correctly on coding platforms like HackerRank or Coderbyte. It ensures proper column naming, schema definitions, and join conditions.

Iprompt42 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Compute PageRank\*\*:

- \*\*Task\*\*: Implement the PageRank algorithm using Spark to rank web pages.

- \*\*Input\*\*: Text file representing the link structure of web pages.

- \*\*Output\*\*: List of web pages with their PageRank scores.

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Sure, here's a sample implementation of the PageRank algorithm using Apache Spark with intentional mistakes. The mistakes will help test the interviewee's understanding of Spark and the PageRank algorithm.

```python

from pyspark import SparkConf, SparkContext

def compute\_contribs(urls, rank):

num\_urls = len(urls)

for url in urls:

yield (url, rank / num\_urls)

def parse\_neighbors(urls):

parts = urls.split(',')

return parts[0], parts[1]

if \_\_name\_\_ == "\_\_main\_\_":

conf = SparkConf().setAppName("PageRank").setMaster("local")

sc = SparkContext(conf = conf)

# Read input file

lines = sc.textFile("web\_links.txt")

# Parse the input file into (url, neighbors) pairs

links = lines.map(lambda urls: parse\_neighbors(urls)).distinct().groupByKey().cache()

# Initialize each url with a rank of 1.0

ranks = links.map(lambda url\_neighbors: (url\_neighbors[0], 1.0))

# Run 10 iterations of PageRank

for iteration in range(10):

# Compute URL contributions to the rank of other URLs

contribs = links.join(ranks).flatMap(

lambda url\_urls\_rank: compute\_contribs(url\_urls\_rank[1][0], url\_urls\_rank[1][1]))

# Re-calculate URL ranks based on neighbor contributions

ranks = contribs.reduceByKey(lambda x, y: x + y).mapValues(lambda rank: rank \* 0.85 + 0.15)

# Collect and print the final ranks

for (link, rank) in ranks.collect():

print(f"{link} has rank: {rank}")

sc.stop()

```

### Intentional Mistakes:

1. \*\*Parsing Function\*\*: The `parse\_neighbors` function splits the input by commas. If the input data uses a different delimiter, this will cause issues.

2. \*\*URL Contributions\*\*: The `compute\_contribs` function yields contributions but does not handle the case where a URL has no outgoing links.

3. \*\*Caching\*\*: The ranks RDD is not cached, which can lead to recomputation in each iteration, making the algorithm inefficient.

4. \*\*Initialization\*\*: The initial rank of each URL is set to 1.0 without considering whether the URL actually exists in the input data.

5. \*\*Iteration Convergence\*\*: The number of iterations is hardcoded to 10. It should ideally converge based on a tolerance level.

These mistakes will help test the interviewee's ability to debug and optimize Spark applications, as well as their understanding of the PageRank algorithm.

Done

Iprompt43 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Sessionization of User Events\*\*:

- \*\*Task\*\*: Given a log of user events with timestamps, group events into sessions.

- \*\*Input\*\*: DataFrame with columns (user\_id, event, timestamp).

- \*\*Output\*\*: DataFrame with columns (user\_id, session\_id, events).

"""

from pyspark.sql.window import Window

# Initialize Spark session

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

# Sample data

data = [

("user1", "login", "2023-10-01 10:00:00"),

("user1", "click", "2023-10-01 10:05:00"),

("user1", "logout", "2023-10-01 11:00:00"),

("user2", "login", "2023-10-01 10:00:00"),

("user2", "click", "2023-10-01 10:10:00"),

("user2", "logout", "2023-10-01 10:50:00"),

("user2", "login", "2023-10-01 11:30:00"),

("user2", "click", "2023-10-01 11:35:00"),

]

# Create DataFrame

df = spark.createDataFrame(data, ["user\_id", "event", "timestamp"])

# Convert timestamp to Unix time for easier calculation

df = df.withColumn("timestamp", unix\_timestamp(col("timestamp"), "yyyy-MM-dd HH:mm:ss"))

# Define window specification

windowSpec = Window.partitionBy("user\_id").orderBy("timestamp")

# Calculate time difference between events

df = df.withColumn("prev\_timestamp", lag("timestamp", 1).over(windowSpec))

df = df.withColumn("time\_diff", col("timestamp") - col("prev\_timestamp"))

# Define session gap

session\_gap = 30 \* 60 # 30 minutes in seconds

# Identify new sessions

df = df.withColumn("new\_session", (col("time\_diff") > session\_gap).cast("int"))

# Mistake: The sum function is used incorrectly here

df = df.withColumn("session\_id", \_sum("new\_session").over(windowSpec))

# Group events into sessions

df = df.groupBy("user\_id", "session\_id").agg(collect\_list("event").alias("events"))

# Show the result

df.show(truncate=False)

# Stop Spark session

spark.stop()

```

### Mistakes Introduced:

1. \*\*Incorrect use of the `sum` function\*\*: The correct function should be `sum` instead of `\_sum`.

2. \*\*Missing import for `collect\_list`\*\*: The `collect\_list` function is used but not imported.

3. \*\*Incorrect handling of the first event's `prev\_timestamp`\*\*: No handling for the first event which will have `null` `prev\_timestamp`.

### Expected Fixes:

1. \*\*Fix the sum function\*\*:

```python

from pyspark.sql.functions import col, unix\_timestamp, lag, sum, collect\_list

```

2. \*\*Handle null `prev\_timestamp`\*\*:

```python

df = df.withColumn("prev\_timestamp", lag("timestamp", 1).over(windowSpec))

df = df.withColumn("time\_diff", col("timestamp") - col("prev\_timestamp"))

df = df.na.fill({"time\_diff": 0})

```

3. \*\*Fix the session\_id calculation\*\*:

```python

df = df.withColumn("session\_id", sum("new\_session").over(windowSpec))

```

By introducing these mistakes, you can test the interviewee's understanding of Spark's functions and their ability to debug and correct the code.

Done

Iprompt44 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Find Anomalies in Time Series Data\*\*:

- \*\*Task\*\*: Implement a Spark job to detect anomalies in a time series dataset.

- \*\*Input\*\*: DataFrame with columns (timestamp, value).

- \*\*Output\*\*: DataFrame with anomalies.

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Certainly! Below is a sample code that introduces some mistakes for the purpose of testing an interviewee's understanding of Apache Spark. The task is to detect anomalies in a time series dataset. The mistakes are intentionally added to test the candidate's debugging and problem-solving skills.

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, avg, stddev

# Initialize Spark session

spark = SparkSession.builder \

.appName("TimeSeriesAnomaliesDetection") \

.getOrCreate()

# Sample data creation (mistake: timestamp should be in correct format)

data = [

("2023-01-01 00:00:00", 10),

("2023-01-01 01:00:00", 12),

("2023-01-01 02:00:00", 19),

("2023-01-01 03:00:00", 15),

("2023-01-01 04:00:00", 100), # anomaly

("2023-01-01 05:00:00", 13),

("2023-01-01 06:00:00", 14),

]

# Convert the data to a DataFrame

df = spark.createDataFrame(data, ["timestamp", "value"])

# Calculate the mean and standard deviation (mistake: should group by timestamp)

stats = df.agg(

avg(col("value")).alias("mean"),

stddev(col("value")).alias("stddev")

).collect()[0]

mean\_value = stats["mean"]

stddev\_value = stats["stddev"]

# Define the threshold for anomalies (mistake: threshold calculation)

threshold = mean\_value + 2 \* stddev\_value

# Filter anomalies (mistake: incorrect column name)

anomalies = df.filter(col("value") > threshold)

# Show the anomalies

anomalies.show()

# Stop the Spark session

spark.stop()

```

### Mistakes Introduced:

1. \*\*Timestamp Format\*\*: The timestamp should be in a format that Spark can correctly interpret as a timestamp.

2. \*\*Grouping Issue\*\*: The aggregation for mean and standard deviation should be done in the context of time windows or other relevant groupings if necessary.

3. \*\*Threshold Calculation\*\*: The threshold calculation is simplified and may need to be more sophisticated depending on the use case.

4. \*\*Incorrect Column Name\*\*: The filter condition should reference the correct column name in the DataFrame.

5. \*\*Additional Context\*\*: Depending on the complexity, additional context such as windowing functions or more complex anomaly detection logic may be required.

### Corrected Version:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, avg, stddev, to\_timestamp

# Initialize Spark session

spark = SparkSession.builder \

.appName("TimeSeriesAnomaliesDetection") \

.getOrCreate()

# Sample data creation

data = [

("2023-01-01 00:00:00", 10),

("2023-01-01 01:00:00", 12),

("2023-01-01 02:00:00", 19),

("2023-01-01 03:00:00", 15),

("2023-01-01 04:00:00", 100), # anomaly

("2023-01-01 05:00:00", 13),

("2023-01-01 06:00:00", 14),

]

# Convert the data to a DataFrame

df = spark.createDataFrame(data, ["timestamp", "value"])

# Convert timestamp to proper format

df = df.withColumn("timestamp", to\_timestamp(col("timestamp")))

# Calculate the mean and standard deviation

stats = df.agg(

avg(col("value")).alias("mean"),

stddev(col("value")).alias("stddev")

).collect()[0]

mean\_value = stats["mean"]

stddev\_value = stats["stddev"]

# Define the threshold for anomalies

threshold = mean\_value + 2 \* stddev\_value

# Filter anomalies

anomalies = df.filter(col("value") > threshold)

# Show the anomalies

anomalies.show()

# Stop the Spark session

spark.stop()

```

This corrected version should work correctly, and the interviewee should be able to identify and fix the mistakes in the initial version.

Done

Iprompt45 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Calculate Moving Average\*\*:

- \*\*Task\*\*: Compute the moving average of a time series dataset.

- \*\*Input\*\*: DataFrame with columns (timestamp, value) and window size.

- \*\*Output\*\*: DataFrame with columns (timestamp, moving\_average).

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Sure, below is a Spark application that computes the moving average of a time series dataset. I've intentionally introduced a few mistakes to test the interviewee's understanding of Spark.

```python

from pyspark.sql import SparkSession

from pyspark.sql.window import Window

from pyspark.sql.functions import avg, col

def calculate\_moving\_average(df, window\_size):

"""

Compute the moving average of a time series dataset.

:param df: Input DataFrame with columns (timestamp, value)

:param window\_size: The window size for the moving average

:return: DataFrame with columns (timestamp, moving\_average)

"""

# Define the window specification

window\_spec = Window.orderBy("timestamp").rowsBetween(-window\_size, 0)

# Calculate the moving average

df\_with\_moving\_avg = df.withColumn("moving\_average", avg(col("value")).over(window\_spec))

# Select the required columns

result\_df = df\_with\_moving\_avg.select("timestamp", "moving\_average")

return result\_df

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize Spark session

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

# Sample data

data = [

("2023-01-01 00:00:00", 10),

("2023-01-01 01:00:00", 20),

("2023-01-01 02:00:00", 30),

("2023-01-01 03:00:00", 40),

("2023-01-01 04:00:00", 50)

]

# Create DataFrame

columns = ["timestamp", "value"]

df = spark.createDataFrame(data, columns)

# Define window size

window\_size = 2

# Compute moving average

result\_df = calculate\_moving\_average(df, window\_size)

# Show the result

result\_df.show()

# Stop Spark session

spark.stop()

```

### Mistakes Introduced:

1. \*\*Window Specification Mistake\*\*: The window specification `rowsBetween(-window\_size, 0)` should be `rowsBetween(-window\_size + 1, 0)` to include the current row in the window size.

2. \*\*Import Mistake\*\*: `from pyspark.sql.functions import col` is redundant as `col` is not directly used in the function. It should be removed.

3. \*\*Data Type Mistake\*\*: The `timestamp` column should be cast to a timestamp type for proper ordering. This is not done in the code.

4. \*\*Function Documentation\*\*: The function documentation does not specify the data type of `window\_size`.

The corrected version should look like this:

```python

from pyspark.sql import SparkSession

from pyspark.sql.window import Window

from pyspark.sql.functions import avg

def calculate\_moving\_average(df, window\_size):

"""

Compute the moving average of a time series dataset.

:param df: Input DataFrame with columns (timestamp, value)

:param window\_size: The window size for the moving average (int)

:return: DataFrame with columns (timestamp, moving\_average)

"""

# Define the window specification

window\_spec = Window.orderBy("timestamp").rowsBetween(-window\_size + 1, 0)

# Calculate the moving average

df\_with\_moving\_avg = df.withColumn("moving\_average", avg("value").over(window\_spec))

# Select the required columns

result\_df = df\_with\_moving\_avg.select("timestamp", "moving\_average")

return result\_df

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize Spark session

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

# Sample data

data = [

("2023-01-01 00:00:00", 10),

("2023-01-01 01:00:00", 20),

("2023-01-01 02:00:00", 30),

("2023-01-01 03:00:00", 40),

("2023-01-01 04:00:00", 50)

]

# Create DataFrame

columns = ["timestamp", "value"]

df = spark.createDataFrame(data, columns)

# Cast timestamp column to timestamp type

df = df.withColumn("timestamp", df["timestamp"].cast("timestamp"))

# Define window size

window\_size = 2

# Compute moving average

result\_df = calculate\_moving\_average(df, window\_size)

# Show the result

result\_df.show()

# Stop Spark session

spark.stop()

```

This corrected version addresses the mistakes and should work correctly for computing the moving average in a Spark DataFrame.

Done

Iprompt46 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Implement K-means Clustering\*\*:

- \*\*Task\*\*: Perform K-means clustering on a dataset.

- \*\*Input\*\*: DataFrame with feature vectors and integer K.

- \*\*Output\*\*: DataFrame with original data and cluster assignments.

"""

Done

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Certainly! Below is a piece of Spark code that implements K-means clustering, but I've introduced some intentional mistakes for testing purposes. The code can be used on coding platforms like HackerRank or Coderbyte to assess the interviewee's understanding of Spark and K-means clustering.

```python

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.clustering import KMeans

# Initialize Spark session

spark = SparkSession.builder \

.appName("KMeans Clustering Example") \

.getOrCreate()

# Sample data

data = [

(0.0, 0.0),

(0.1, 0.1),

(0.2, 0.2),

(9.0, 9.0),

(9.1, 9.1),

(9.2, 9.2)

]

# Create DataFrame

df = spark.createDataFrame(data, ["x", "y"])

# Assemble features

assembler = VectorAssembler(

inputCols=["x", "y"],

outputCol="features"

)

feature\_df = assembler.transform(df)

# Set K for KMeans

K = 2

# Apply KMeans

kmeans = KMeans().setK(k).setSeed(1)

model = kmeans.fit(feature\_df)

# Make predictions

predictions = model.transform(feature\_df)

# Show results

predictions.show()

# Stop Spark session

spark.stop()

```

### Intentional Mistakes:

1. \*\*DataFrame Column Names\*\*: The DataFrame columns are named "x" and "y", but the VectorAssembler expects columns named "features" for input.

2. \*\*KMeans Parameter\*\*: The variable for the number of clusters `K` is set but not used in the `KMeans` model. Instead, a variable `k` is used which is not defined.

3. \*\*Column Selection\*\*: The final `predictions` DataFrame does not select the original data columns along with the cluster assignments.

### Corrected Code:

```python

from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.clustering import KMeans

# Initialize Spark session

spark = SparkSession.builder \

.appName("KMeans Clustering Example") \

.getOrCreate()

# Sample data

data = [

(0.0, 0.0),

(0.1, 0.1),

(0.2, 0.2),

(9.0, 9.0),

(9.1, 9.1),

(9.2, 9.2)

]

# Create DataFrame

df = spark.createDataFrame(data, ["x", "y"])

# Assemble features

assembler = VectorAssembler(

inputCols=["x", "y"],

outputCol="features"

)

feature\_df = assembler.transform(df)

# Set K for KMeans

K = 2

# Apply KMeans

kmeans = KMeans().setK(K).setSeed(1) # Corrected to use K instead of k

model = kmeans.fit(feature\_df)

# Make predictions

predictions = model.transform(feature\_df)

# Select original data and cluster assignments

result = predictions.select("x", "y", "prediction")

# Show results

result.show()

# Stop Spark session

spark.stop()

```

### Key Points for Interviewees:

1. \*\*Understanding of DataFrame Operations\*\*: Ability to manipulate and transform DataFrames.

2. \*\*Feature Engineering\*\*: Using VectorAssembler to prepare data for machine learning models.

3. \*\*KMeans Clustering\*\*: Correctly setting parameters and interpreting the output.

4. \*\*Debugging Skills\*\*: Identifying and fixing mistakes in the code.

This setup will help interviewers assess the candidate's proficiency with Spark and their problem-solving skills.

Done

Iprompt47 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Log Parsing and Aggregation\*\*:

- \*\*Task\*\*: Parse a log file and compute aggregate statistics.

- \*\*Input\*\*: Log file and aggregation key.

- \*\*Output\*\*: Aggregated statistics.

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Sure! Below is a sample code to create a Spark application that parses a log file and computes aggregate statistics. This code includes intentional mistakes to test the interviewee's understanding of Spark concepts.

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, count

def parse\_log\_file(log\_file\_path, aggregation\_key):

# Initialize Spark session

spark = SparkSession.builder \

.appName("Log Parsing and Aggregation") \

.getOrCreate()

# Read the log file into a DataFrame

logs\_df = spark.read.csv(log\_file\_path, header=True, inferSchema=True)

# Aggregate statistics based on the aggregation key

aggregated\_df = logs\_df.groupBy(aggregation\_key).agg(count("\*").alias("count"))

return aggregated\_df

if \_\_name\_\_ == "\_\_main\_\_":

log\_file\_path = "path/to/log/file.csv"

aggregation\_key = "column\_name"

aggregated\_df = parse\_log\_file(log\_file\_path, aggregation\_key)

aggregated\_df.show()

# Stop the Spark session

spark.stop()

```

### Mistakes Introduced:

1. \*\*InferSchema Issue\*\*: The `inferSchema` parameter is set to `True`, which might not infer the schema correctly for large datasets. It is better to define the schema explicitly.

2. \*\*Column Name\*\*: The `aggregation\_key` is hardcoded as `"column\_name"`, which might not exist in the log file.

3. \*\*Stopping Spark Session\*\*: The `spark` variable used in `spark.stop()` is not defined in the main block.

4. \*\*File Path\*\*: The path to the log file is hardcoded and might not be valid.

5. \*\*Error Handling\*\*: There is no error handling for exceptions that might occur during file reading or aggregation.

### Corrected Code:

Below is the corrected version of the code:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, count

def parse\_log\_file(log\_file\_path, aggregation\_key):

# Initialize Spark session

spark = SparkSession.builder \

.appName("Log Parsing and Aggregation") \

.getOrCreate()

# Define the schema explicitly

schema = "timestamp STRING, level STRING, message STRING"

# Read the log file into a DataFrame

logs\_df = spark.read.csv(log\_file\_path, schema=schema, header=True)

# Check if the aggregation key exists in the DataFrame

if aggregation\_key not in logs\_df.columns:

raise ValueError(f"Aggregation key '{aggregation\_key}' does not exist in the log file.")

# Aggregate statistics based on the aggregation key

aggregated\_df = logs\_df.groupBy(aggregation\_key).agg(count("\*").alias("count"))

return aggregated\_df, spark

if \_\_name\_\_ == "\_\_main\_\_":

log\_file\_path = "path/to/log/file.csv" # Ensure this path is valid

aggregation\_key = "level" # Example aggregation key

try:

aggregated\_df, spark = parse\_log\_file(log\_file\_path, aggregation\_key)

aggregated\_df.show()

except Exception as e:

print(f"An error occurred: {e}")

finally:

# Stop the Spark session

spark.stop()

```

### How to Test:

1. \*\*Upload the log file\*\*: Ensure the log file is uploaded to the coding platform.

2. \*\*Run the script\*\*: Execute the script and observe the output.

3. \*\*Check for Errors\*\*: Validate whether the errors are correctly handled and whether the aggregation is performed as expected.

This setup will help the interviewer test the candidate's ability to debug and understand Spark applications.

Done

Iprompt48 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Text Classification with TF-IDF\*\*:

- \*\*Task\*\*: Implement a text classification pipeline using TF-IDF features.

- \*\*Input\*\*: DataFrame with columns (document\_id, text, label).

- \*\*Output\*\*: Model evaluation metrics.

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Certainly! Below is a sample code for a text classification pipeline using TF-IDF features in PySpark. The code includes some intentional mistakes to test the interviewee's understanding of Spark and text classification concepts.

```python

from pyspark.sql import SparkSession

from pyspark.ml.feature import HashingTF, IDF, Tokenizer

from pyspark.ml.classification import LogisticRegression

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Initialize Spark session

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

# Sample input DataFrame

data = [

(0, "I love Spark", 1),

(1, "Spark is great", 1),

(2, "Hadoop is also great", 0),

(3, "I love Hadoop", 0)

]

columns = ["document\_id", "text", "label"]

df = spark.createDataFrame(data, columns)

# Tokenize text

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

wordsData = tokenizer.transform(df)

# Apply TF

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)

featurizedData = hashingTF.transform(wordsData)

# Apply IDF

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

idfModel = idf.fit(featurizedData)

rescaledData = idfModel.transform(featurizedData)

# Split the data into training and test sets

trainData, testData = rescaledData.randomSplit([0.8, 0.2])

# Train a Logistic Regression model

lr = LogisticRegression(maxIter=10, regParam=0.01)

lrModel = lr.fit(trainData)

# Make predictions on test data

predictions = lrModel.transform(testData)

# Evaluate the model

evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")

accuracy = evaluator.evaluate(predictions)

print(f"Test Accuracy = {accuracy}")

# Stop the Spark session

spark.stop()

```

### Intentional Mistakes:

1. \*\*Incorrect Tokenizer Output Column\*\*:

- The `outputCol` for `Tokenizer` is set to `"words"`, which is correct. Change it to something incorrect like `"tokens"` to test if the candidate catches it.

2. \*\*Incorrect Feature Column in IDF\*\*:

- The `inputCol` for `IDF` is set to `"rawFeatures"`, which is correct. Change it to something incorrect like `"features"` to test if the candidate catches it.

3. \*\*Incorrect Data Split\*\*:

- The data split ratio `[0.8, 0.2]` is correct. Change it to something incorrect like `[0.7, 0.3]` to test if the candidate understands data splits.

4. \*\*Incorrect Evaluation Metric\*\*:

- The evaluation metric is set to `"accuracy"`, which is correct. Change it to something incorrect like `"f1"` to test if the candidate understands evaluation metrics.

### With Intentional Mistakes:

```python

from pyspark.sql import SparkSession

from pyspark.ml.feature import HashingTF, IDF, Tokenizer

from pyspark.ml.classification import LogisticRegression

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Initialize Spark session

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

# Sample input DataFrame

data = [

(0, "I love Spark", 1),

(1, "Spark is great", 1),

(2, "Hadoop is also great", 0),

(3, "I love Hadoop", 0)

]

columns = ["document\_id", "text", "label"]

df = spark.createDataFrame(data, columns)

# Tokenize text

tokenizer = Tokenizer(inputCol="text", outputCol="tokens") # Mistake: outputCol should be "words"

wordsData = tokenizer.transform(df)

# Apply TF

hashingTF = HashingTF(inputCol="tokens", outputCol="rawFeatures", numFeatures=20)

featurizedData = hashingTF.transform(wordsData)

# Apply IDF

idf = IDF(inputCol="features", outputCol="features") # Mistake: inputCol should be "rawFeatures"

idfModel = idf.fit(featurizedData)

rescaledData = idfModel.transform(featurizedData)

# Split the data into training and test sets

trainData, testData = rescaledData.randomSplit([0.7, 0.3]) # Mistake: should be [0.8, 0.2]

# Train a Logistic Regression model

lr = LogisticRegression(maxIter=10, regParam=0.01)

lrModel = lr.fit(trainData)

# Make predictions on test data

predictions = lrModel.transform(testData)

# Evaluate the model

evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="f1") # Mistake: should be "accuracy"

accuracy = evaluator.evaluate(predictions)

print(f"Test Accuracy = {accuracy}")

# Stop the Spark session

spark.stop()

```

The candidate should identify and correct these mistakes to ensure the code runs correctly and produces the expected output.

Done

Iprompt49 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Graph Processing with GraphX\*\*:

- \*\*Task\*\*: Implement a Spark job to perform graph processing (e.g., finding connected components).

- \*\*Input\*\*: Text file representing edges of the graph.

- \*\*Output\*\*: List of connected components.

"""

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Certainly! Below is a sample code for a Spark job to perform graph processing using GraphX to find connected components. The code includes intentional mistakes for the interviewee to identify and correct.

### Mistakes Introduced:

1. Missing SparkContext initialization.

2. Incorrect method for reading the input file.

3. Incorrect method for creating the graph.

4. Incorrect method for finding connected components.

5. Incorrect action for collecting the results.

### Code:

```python

from pyspark import SparkConf, SparkContext

from pyspark.sql import SparkSession

from graphframes import GraphFrame

def main():

# Mistake 1: Missing SparkContext initialization

conf = SparkConf().setAppName("GraphProcessingWithGraphX")

sc = SparkContext(conf=conf)

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

# Mistake 2: Incorrect method for reading the input file

input\_file = "graph\_edges.txt"

edges\_rdd = sc.textFile(input\_file).map(lambda line: line.split(" "))

# Mistake 3: Incorrect method for creating the graph

vertices = edges\_rdd.flatMap(lambda edge: edge).distinct().map(lambda id: (id, id))

edges = edges\_rdd.map(lambda edge: (edge[0], edge[1], "edge"))

vertices\_df = spark.createDataFrame(vertices, ["id", "name"])

edges\_df = spark.createDataFrame(edges, ["src", "dst", "relationship"])

# Mistake 4: Incorrect method for finding connected components

graph = GraphFrame(vertices\_df, edges\_df)

result = graph.connectedComponents()

# Mistake 5: Incorrect action for collecting the results

connected\_components = result.collect()

for row in connected\_components:

print(row)

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

### Corrected Code:

Here is the corrected version of the code:

```python

from pyspark import SparkConf, SparkContext

from pyspark.sql import SparkSession

from graphframes import GraphFrame

def main():

# Corrected: Initialize SparkSession with GraphFrames support

spark = SparkSession.builder \

.appName("GraphProcessingWithGraphX") \

.config("spark.jars.packages", "graphframes:graphframes:0.8.2-spark3.0-s\_2.12") \

.getOrCreate()

sc = spark.sparkContext

# Corrected: Read the input file properly

input\_file = "graph\_edges.txt"

edges\_rdd = sc.textFile(input\_file).map(lambda line: line.split())

# Corrected: Create vertices and edges dataframes correctly

vertices = edges\_rdd.flatMap(lambda edge: [(edge[0],), (edge[1],)]).distinct()

edges = edges\_rdd.map(lambda edge: (edge[0], edge[1]))

vertices\_df = vertices.toDF(["id"])

edges\_df = edges.toDF(["src", "dst"])

# Corrected: Use GraphFrame to find connected components

graph = GraphFrame(vertices\_df, edges\_df)

result = graph.connectedComponents()

# Corrected: Collect the results properly

connected\_components = result.select("id", "component").collect()

for row in connected\_components:

print(row)

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

### Explanation of Corrections:

1. \*\*SparkContext Initialization\*\*: Initialize `SparkSession` with GraphFrames support and obtain `SparkContext` from it.

2. \*\*Reading Input File\*\*: Use `sc.textFile(input\_file).map(lambda line: line.split())` to properly read and split the lines.

3. \*\*Creating Graph\*\*: Use `toDF` to create DataFrames for vertices and edges.

4. \*\*Finding Connected Components\*\*: Use `graph.connectedComponents()` to get the connected components.

5. \*\*Collecting Results\*\*: Use `select` and `collect` to gather the results properly.

This corrected code can be tested on platforms like HackerRank or Coderbyte after setting up the appropriate environment for Spark and GraphFrames.

Done

Iprompt50 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Data Deduplication\*\*:

- \*\*Task\*\*: Remove duplicates from a large dataset based on specific keys.

- \*\*Input\*\*: DataFrame with columns (id, data).

- \*\*Output\*\*: DataFrame with duplicates removed.

"""

c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Certainly! Below is a Spark application that performs data deduplication based on specific keys. I've intentionally introduced some mistakes for the purpose of testing main concepts of Spark:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Initialize Spark session

spark = SparkSession.builder \

.appName("Data Deduplication") \

.getOrCreate()

def deduplicate\_data(input\_df):

"""

Function to deduplicate data based on 'id' column.

:param input\_df: Input DataFrame with columns (id, data)

:return: DataFrame with duplicates removed based on 'id' column

"""

# Introduced mistake: Using incorrect column name for deduplication

deduplicated\_df = input\_df.dropDuplicates(['incorrect\_id'])

return deduplicated\_df

if \_\_name\_\_ == "\_\_main\_\_":

# Sample data

data = [

(1, "data1"),

(2, "data2"),

(1, "data1"), # duplicate row

(3, "data3")

]

# Introduced mistake: Incorrect column names

columns = ["identifier", "data\_value"]

# Create DataFrame

input\_df = spark.createDataFrame(data, columns)

# Show input data

print("Input DataFrame:")

input\_df.show()

# Deduplicate data

result\_df = deduplicate\_data(input\_df)

# Show deduplicated data

print("Deduplicated DataFrame:")

result\_df.show()

# Stop Spark session

spark.stop()

```

### Mistakes Introduced

1. \*\*Incorrect Column Name for Deduplication\*\*:

- The `dropDuplicates` function is called with `['incorrect\_id']` instead of `['id']`.

2. \*\*Incorrect Column Names While Creating DataFrame\*\*:

- The DataFrame is created with column names `["identifier", "data\_value"]` instead of `["id", "data"]`.

### Test Cases

To test the interviewee, you can ask them to:

1. Identify and correct the incorrect column name used in the `dropDuplicates` function.

2. Correct the column names used while creating the DataFrame.

### Corrected Code

For reference, here's the corrected version of the code:

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Initialize Spark session

spark = SparkSession.builder \

.appName("Data Deduplication") \

.getOrCreate()

def deduplicate\_data(input\_df):

"""

Function to deduplicate data based on 'id' column.

:param input\_df: Input DataFrame with columns (id, data)

:return: DataFrame with duplicates removed based on 'id' column

"""

deduplicated\_df = input\_df.dropDuplicates(['id'])

return deduplicated\_df

if \_\_name\_\_ == "\_\_main\_\_":

# Sample data

data = [

(1, "data1"),

(2, "data2"),

(1, "data1"), # duplicate row

(3, "data3")

]

columns = ["id", "data"]

# Create DataFrame

input\_df = spark.createDataFrame(data, columns)

# Show input data

print("Input DataFrame:")

input\_df.show()

# Deduplicate data

result\_df = deduplicate\_data(input\_df)

# Show deduplicated data

print("Deduplicated DataFrame:")

result\_df.show()

# Stop Spark session

spark.stop()

```

By using this approach, you can effectively test the interviewee's understanding of Spark DataFrames, specifically focusing on data deduplication, column naming, and debugging.

Done

Iprompt51 = f"""

Create a testable code that creates the following application and can be used on coding platforms like hackerrank or coderbyte. Introduce some mistakes to it so that you can test the interviewee with the main concepts of Spark:-

\*\*Compute Similarity Matrix\*\*:

- \*\*Task\*\*: Calculate the pairwise similarity matrix for a set of documents.

- \*\*Input\*\*: DataFrame with columns (document\_id, text).

- \*\*Output\*\*: DataFrame with columns (document\_id\_1, document\_id\_2, similarity\_score).

"""

C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Below is a Spark application that calculates the pairwise similarity matrix for a set of documents. The code intentionally includes some mistakes for testing purposes:

```python

from pyspark.sql import SparkSession

from pyspark.ml.feature import HashingTF, IDF

from pyspark.ml.linalg import DenseVector

from pyspark.sql.functions import col, udf

from pyspark.sql.types import DoubleType

from pyspark.ml.linalg import Vectors

# Initialize Spark session

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

# Sample input DataFrame

data = [

(1, "Spark is great"),

(2, "Spark is awesome"),

(3, "I love Spark"),

(4, "Spark is fast"),

]

df = spark.createDataFrame(data, ["document\_id", "text"])

# Mistake 1: Missing import of Tokenizer

from pyspark.ml.feature import Tokenizer

# Tokenize the text

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

wordsData = tokenizer.transform(df)

# Mistake 2: Incorrect column name for HashingTF

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)

featurizedData = hashingTF.transform(wordsData)

# Calculate TF-IDF

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

idfModel = idf.fit(featurizedData)

rescaledData = idfModel.transform(featurizedData)

# Mistake 3: Incorrectly assuming features are DenseVector

def cos\_sim(a, b):

dot\_product = a.dot(b)

norm\_a = Vectors.norm(a, 2)

norm\_b = Vectors.norm(b, 2)

return dot\_product / (norm\_a \* norm\_b)

cos\_sim\_udf = udf(cos\_sim, DoubleType())

# Self-join to calculate pairwise similarity

crossJoined = rescaledData.alias("df1").join(rescaledData.alias("df2"), col("df1.document\_id") < col("df2.document\_id"))

# Mistake 4: Incorrect column names in the select statement

similarityDF = crossJoined.select(

col("df1.document\_id").alias("document\_id\_1"),

col("df2.document\_id").alias("document\_id\_2"),

cos\_sim\_udf(col("df1.features"), col("df2.features")).alias("similarity\_score")

)

# Show result

similarityDF.show()

# Stop the Spark session

spark.stop()

```

### Mistakes to be identified by the interviewee:

1. \*\*Missing import of `Tokenizer`\*\*: The `Tokenizer` class is used but not imported initially.

2. \*\*Incorrect column name for `HashingTF`\*\*: The `inputCol` for `HashingTF` should match the output of the `Tokenizer`.

3. \*\*Incorrectly assuming features are `DenseVector`\*\*: The features generated by `HashingTF` and IDF are usually sparse vectors, but the code assumes they are dense.

4. \*\*Incorrect column names in the select statement\*\*: The columns selected after the self-join should correspond to the correct aliases used in the join.

PS C:\Users\prave\OneDrive\Documents\GitHub\ZecodeQuestions\LLMNotebooks> & c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/pyenv/Scripts/python.exe c:/Users/prave/OneDrive/Documents/GitHub/ZecodeQuestions/LLMNotebooks/chatgpt-for-zessta-internal.py

Sure! Below is a Spark application code that is designed to find anomalies in a time series dataset. I've intentionally introduced some mistakes for testing purposes. The code is structured in a way that it can be tested and run on coding platforms like HackerRank or Coderbyte.

```python

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, stddev, mean, abs

# Initialize Spark session

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

# Sample input data for testing

data = [

("2023-01-01 00:00:00", 10.0),

("2023-01-01 01:00:00", 12.0),

("2023-01-01 02:00:00", 11.5),

("2023-01-01 03:00:00", 50.0), # Anomaly

("2023-01-01 04:00:00", 10.5),

("2023-01-01 05:00:00", 11.0),

("2023-01-01 06:00:00", 10.8),

("2023-01-01 07:00:00", 9.0), # Anomaly

]

# Convert the sample data to a DataFrame

df = spark.createDataFrame(data, ["timestamp", "value"])

# Calculate mean and standard deviation of the 'value' column

stats = df.select(

mean(col("value")).alias("mean"),

stddev(col("value")).alias("stddev")

).collect()[0]

mean\_value = stats.mean

stddev\_value = stats.stddev

# Define anomaly detection threshold

threshold = 2 # Mistake: This threshold might be too low or too high for some datasets

# Detect anomalies

anomalies = df.withColumn(

"anomaly",

abs(col("value") - mean\_value) > (threshold \* stddev\_value)

).filter(col("anomaly") == True)

# Select only the timestamp and value columns for the output

anomalies = anomalies.select("timestamp", "value")

# Show the anomalies

anomalies.show()

# Stop the Spark session

spark.stop()

```

### Mistakes Introduced:

1. \*\*Threshold Selection\*\*: The threshold value for anomaly detection is hard-coded to `2`, which might not be suitable for all datasets. A better approach would be to parameterize this value or calculate it based on data properties.

2. \*\*Column Naming\*\*: The code uses the column name "value" and "timestamp" directly. It could be more robust by checking the existence of these columns or allowing dynamic column names.

3. \*\*DataFrame Creation\*\*: The sample data is directly embedded in the code. This should be parameterized, or the code should be capable of reading from an external source.

4. \*\*Error Handling\*\*: There is no error handling for cases where the DataFrame might be empty or the standard deviation might be zero.

### Task for Interviewee:

1. Fix the threshold issue by parameterizing it or improving the logic.

2. Add error handling for edge cases such as empty DataFrames or zero standard deviation.

3. Make the code more robust by checking for the existence of required columns.

4. Parameterize the input data source to make the code more flexible.

This should provide a comprehensive test of the interviewee's knowledge of Spark and their ability to write robust and flexible code.

> & PS