Cleaning Data with PySpark (Complex processing and data pipelines)

Quick pipeline

Before you parse some more complex data, your manager would like to see a simple pipeline example including the basic steps. For this example, you'll want to ingest a data file, filter a few rows, add an ID column to it, then write it out as JSON data.

The `spark` context is defined, along with the `pyspark.sql.functions` library being aliased as `F` as is customary.

```
# Import the data to a DataFrame
departures_df = spark.read.csv('2015-departures.csv.gz', header=True)

# Remove any duration of 0
departures_df = departures_df.filter(departures_df[3]> 0)

# Add an ID column
departures_df = departures_df.withColumn('id', F.monotonically_increasing_id())

# Write the file out to JSON format
departures_df.write.json('output.json')
```

Removing commented lines

Your boss would like you to perform some complex parsing on a new dataset. The data represents annotation data for the ImageNet dataset, but focusing specifically on dog breeds and identifying them in images. Before any actual analysis can occur, you'll need to clear out several components of invalid / incorrect data. The general schema of the document is unknown so you'd like to import the rows into a single column, allowing for quick analysis.

To start, you need to remove all commented rows in the dataset.

The `spark` context, and the base CSV file (`annotations.csv.gz`) are available for you to work with. The `col` function is also available for use.

```
# Import the file to a DataFrame and perform a row count
annotations_df = spark.read.csv('annotations.csv.gz', sep='|')
full_count = annotations_df.count()

```

```
# Count the number of rows beginning with '#'
comment_count = annotations_df.where(col('_c0').startswith('#')).count()

# Import the file to a new DataFrame, without commented rows
no_comments_df = spark.read.csv('annotations.csv.gz', sep='|', comment='#')

# Count the new DataFrame and verify the difference is as expected
no_comments_count = no_comments_df.count()
print("Full count: %d\nComment count: %d\nRemaining count: %d" % (full_count, comment_count,
no_comments_count))
```

Removing invalid rows

Now that you've successfully removed the commented rows, you have received some information about the general format of the data. There should be at minimum 5 tab separated columns in the DataFrame. Remember that your original DataFrame only has a single column, so you'll need to split the data on the tab (`\t`) characters.

The DataFrame `annotations_df` is already available, with the commented rows removed. The `spark.sql.functions` library is available under the alias `F` . The initial number of rows available in the DataFrame is stored in the variable `initial_count` .

```
# Split _c0 on the tab character and store the list in a variable
tmp_fields = F.split(annotations_df['_c0'], '\t')

# Create the colcount column on the DataFrame
annotations_df = annotations_df.withColumn('colcount', F.size(tmp_fields))

# Remove any rows containing fewer than 5 fields
annotations_df_filtered = annotations_df.filter(~ (annotations_df.colcount >= 5))

# Count the number of rows
final_count = annotations_df_filtered.count()
print("Initial count: %d\nFinal count: %d" % (initial_count, final_count))
```

Splitting into columns

You've cleaned up your data considerably by removing the invalid rows from the DataFrame. Now you want to perform some further transformations by generating specific meaningful columns based on the DataFrame content.

You have the `spark` context and the latest version of the `annotations_df` DataFrame. `pyspark.sql.functions` is available under the alias `F` .

```
# Split the content of _c0 on the tab character (aka, '\t')
split_cols = F.split(annotations_df['_c0'], '\t')

# Add the columns folder, filename, width, and height
split_df = annotations_df.withColumn('folder', split_cols.getItem(0))
```

```
split_df = split_df.withColumn('filename', split_cols.getItem(1))
split_df = split_df.withColumn('width', split_cols.getItem(2))
split_df = split_df.withColumn('height', split_cols.getItem(3))

# Add split_cols as a column
split_df = split_df.withColumn('split_cols', split_cols)
```

Further parsing

You've molded this dataset into a significantly different format than it was before, but there are still a few things left to do. You need to prep the column data for use in later analysis and remove a few intermediary columns.

The `spark` context is available and `pyspark.sql.functions` is aliased as `F`. The types from `pyspark.sql.types` are already imported. The `split_df` DataFrame is as you last left it. Remember, you can use `.printSchema()` on a DataFrame in the console area to view the column names and types.

```
def retriever(cols, colcount):
  # Return a list of dog data
  return cols[4:colcount]

# Define the method as a UDF
udfRetriever = F.udf(retriever, ArrayType(StringType()))

# Create a new column using your UDF
split_df = split_df.withColumn('dog_list', udfRetriever(split_df.split_cols,
split_df.colcount))

# Remove the original column, split_cols, and the colcount
split_df = split_df.drop('_c0').drop('colcount').drop('split_cols')
```

Validate rows via join

Another example of filtering data is using joins to remove invalid entries. You'll need to verify the folder names are as expected based on a given DataFrame named `valid_folders_df`. The DataFrame `split_df` is as you last left it with a group of split columns.

The `spark` object is available, and `pyspark.sql.functions` is imported as `F`.

```
# Rename the column in valid_folders_df
valid_folders_df = valid_folders_df.withColumnRenamed('_c0', 'folder')

# Count the number of rows in split_df
split_count = split_df.count()

# Join the DataFrames
joined_df = split_df.join(F.broadcast(valid_folders_df), 'folder')
```

```
9
10 # Compare the number of rows remaining
11 joined_count = joined_df.count()
12 print("Before: %d\nAfter: %d" % (split_count, joined_count))
```

Examining invalid rows

You've successfully filtered out the rows using a join, but sometimes you'd like to examine the data that is invalid. This data can be stored for later processing or for troubleshooting your data sources.

You want to find the difference between two DataFrames and store the invalid rows.

The `spark` object is defined and `pyspark.sql.functions` are imported as `F`. The original DataFrame `split_df` and the joined DataFrame `joined_df` are available as they were in their previous states.

```
1  # Determine the row counts for each DataFrame
2  split_count = split_df.count()
3  joined_count = joined_df.count()
4
5  # Create a DataFrame containing the invalid rows
6  invalid_df = split_df.join(F.broadcast(joined_df), 'folder', 'LeftAnti')
7
8  # Validate the count of the new DataFrame is as expected
9  invalid_count = invalid_df.count()
10 print(" split_df:\t%d\n joined_df:\t%d\n invalid_df: \t%d" % (split_count, joined_count,
   invalid_count))
11
12 # Determine the number of distinct folder rows removed
13 invalid_folder_count = invalid_df.select('folder').distinct().count()
14 print("%d distinct invalid folders found" % invalid_folder_count)
```