entity-resolution

April 7, 2024

Importing necessary libraries and modules
- `pyspark`: the main PySpark library
- `os` and `sys`: used for setting environment variables

Setting environment variables
- `PYSPARK_PYTHON` and `PYSPARK_DRIVER_PYTHON` are set to the current Python executable path

Importing `SparkSession` from `pyspark.sql`

[ ]:
```
import pyspark
import os
import sys
from pyspark import SparkContext

os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable

from pyspark.sql import SparkSession
```

Creating a `SparkSession` instance
- `SparkSession.builder`: used to create a new `SparkSession`
- `.config("spark.driver.memory", "16g")`: sets the driver memory to 16GB
- `.appName('chapter_2')`: sets the application name to "chapter_2"

[ ]:
```
spark = SparkSession.builder.config("spark.driver.memory", "16g").
↪appName('chapter_2').getOrCreate()
```

Reading a CSV file without specifying options
- `spark.read.csv("linkage/block_1.csv")`: reads the CSV file "block_1.csv" from the "linkage" directory
- `.show(2)`: displays the first 2 rows of the DataFrame

Reading a CSV file with specified options
- `.option("header", "true")`: treats the first row as the header
- `.option("nullValue", "?")`: sets the null value placeholder to "?"
- `.option("inferSchema", "true")`: infers the data types of columns automatically
- `.printSchema()`: prints the schema of the DataFrame
- `.show(5)`: displays the first 5 rows of the DataFrame
- `.count()`: returns the number of rows in the DataFrame

Caching the DataFrame in memory

[ ]:
```
prev = spark.read.csv("linkage/block_1.csv")
prev.show(2)

parsed = spark.read.option("header", "true").option("nullValue", "?").
  ↪option("inferSchema", "true").csv("linkage/block_1.csv")
parsed.printSchema()
parsed.show(5)
print(parsed.count())

parsed.cache()
```

Importing the `col` function from `pyspark.sql.functions`
- `col`: used to refer to a column in a DataFrame

Grouping the DataFrame by a column and counting the occurrences
- `.groupBy("is_match")`: groups the DataFrame by the "is_match" column
- `.count()`: counts the number of rows in each group
- `.orderBy(col("count").desc())`: sorts the resulting DataFrame by the "count" column in descending order

[ ]:
```
from pyspark.sql.functions import col

parsed.groupBy("is_match").count().orderBy(col("count").desc()).show()
```

Creating a temporary view of the DataFrame
- `.createOrReplaceTempView("linkage")`: creates a temporary view named "linkage" from the `parsed` DataFrame

[ ]:
```
parsed.createOrReplaceTempView("linkage")
```

Running a SQL query on the temporary view
- `spark.sql()`: executes a SQL query on the temporary view "linkage"
- The SQL query:
- Selects the "is_match" column and counts the number of rows for each distinct value using `COUNT(*)`
- Groups the result by the "is_match" column using `GROUP BY`
- Orders the result by the count column alias "cnt" in descending order using `ORDER BY cnt DESC`

[ ]:
```
spark.sql("""
SELECT is_match, COUNT(*) cnt
FROM linkage
GROUP BY is_match
ORDER BY cnt DESC
""").show()
```

Generating summary statistics for the DataFrame
- `.describe()`: computes summary statistics for numeric and string columns in the DataFrame
- The resulting DataFrame contains statistics like count, mean, standard deviation, min, and max for each column

- `.select("summary", "cmp_fname_c1", "cmp_fname_c2")`: selects specific columns from the summary DataFrame

[ ]:
```
summary = parsed.describe()
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()
```

Filtering the DataFrame to get only the matched records
- `.where("is_match = true")`: filters the DataFrame to include only rows where the "is_match" column is true
- `.describe()`: generates summary statistics for the filtered DataFrame (`matches`)

Filtering the DataFrame to get only the non-matched records
- `.filter(col("is_match") == False)`: filters the DataFrame to include only rows where the "is_match" column is false

[ ]:
```
matches = parsed.where("is_match = true")
match_summary = matches.describe()
misses = parsed.filter(col("is_match") == False)
miss_summary = misses.describe()
```

Converting the summary DataFrame to a pandas DataFrame
- `.toPandas()`: converts the PySpark DataFrame (`summary`) to a pandas DataFrame
- The resulting pandas DataFrame is assigned to the variable `summary_p`

[ ]:
```
summary_p = summary.toPandas()
```

Displaying the first few rows of the pandas DataFrame
- `.head()`: shows the first 5 rows of the pandas DataFrame (`summary_p`) by default

Checking the dimensions of the pandas DataFrame
- `.shape`: returns a tuple representing the dimensions of the DataFrame (`summary_p`)

[ ]:
```
summary_p.head()
summary_p.shape
```

summary_p = summary_p.set_index('summary').transpose().reset_index()
Reshaping the pandas DataFrame
- `.set_index('summary')`: sets the 'summary' column as the index of the DataFrame
- `.transpose()`: transposes the DataFrame, swapping rows and columns
- `.reset_index()`: resets the index, moving the index to a regular column

Renaming a column in the pandas DataFrame
- `.rename(columns={'index':'field'})`: renames the 'index' column to 'field'

Removing the column index name
- `.rename_axis(None, axis=1)`: removes the name of the column index (axis=1) by setting it to `None`

Checking the dimensions of the modified pandas DataFrame

```
[ ]: summary_p = summary_p.set_index('summary').transpose().reset_index()
summary_p = summary_p.rename(columns={'index':'field'})
summary_p = summary_p.rename_axis(None, axis=1)
summary_p.shape
```

Creating a PySpark DataFrame from a pandas DataFrame
- `spark.createDataFrame(summary_p)`: creates a new PySpark DataFrame (`summaryT`) from the pandas DataFrame (`summary_p`)
- This allows for using PySpark operations and functions on the data

```
[ ]: summaryT = spark.createDataFrame(summary_p)
summaryT.printSchema()
```

Importing the `DoubleType` from `pyspark.sql.types`
- `DoubleType`: represents a double precision floating point number data type in PySpark

Casting columns to `DoubleType`
- Iterates over each column in the DataFrame (`summaryT`)
- Skips the 'field' column using `if c == 'field': continue`
- For each remaining column:
- `.withColumn(c, summaryT[c].cast(DoubleType()))`: casts the column to `DoubleType` using `.cast()`
- Overwrites the DataFrame (`summaryT`) with the updated column data type

```
[ ]: from pyspark.sql.types import DoubleType
for c in summaryT.columns:
    if c == 'field':
        continue
    summaryT = summaryT.withColumn(c, summaryT[c].cast(DoubleType()))
summaryT.printSchema()
```

Defining a function to pivot and transform a summary DataFrame
- `pivot_summary(desc)`: takes a PySpark DataFrame (`desc`) as input and returns a transformed DataFrame
- The function performs the following steps:
1. Converts the PySpark DataFrame to a pandas DataFrame using `.toPandas()`
2. Transposes the pandas DataFrame:
- Sets the 'summary' column as the index using `.set_index('summary')`
- Transposes the DataFrame using `.transpose()`
- Resets the index using `.reset_index()`
3. Renames the 'index' column to 'field' using `.rename(columns={'index':'field'})`
4. Removes the column index name using `.rename_axis(None, axis=1)`
5. Converts the transformed pandas DataFrame back to a PySpark DataFrame using `spark.createDataFrame(desc_p)`
6. Casts the metric columns to `DoubleType`:
- Iterates over each column in the DataFrame
- Skips the 'field' column
- Casts the remaining columns to `DoubleType` using `.withColumn(c,`

descT[c].cast(DoubleType()))
7. Returns the transformed PySpark DataFrame (descT)

```
[ ]: from pyspark.sql import DataFrame
from pyspark.sql.types import DoubleType
def pivot_summary(desc):
    # convert to pandas dataframe
    desc_p = desc.toPandas()
    # transpose
    desc_p = desc_p.set_index('summary').transpose().reset_index()
    desc_p = desc_p.rename(columns={'index':'field'})
    desc_p = desc_p.rename_axis(None, axis=1)
    # convert to Spark dataframe
    descT = spark.createDataFrame(desc_p)
    # convert metric columns to double from string
    for c in descT.columns:
        if c == 'field':
            continue
        else:
            descT = descT.withColumn(c, descT[c].cast(DoubleType()))
    return descT
```

Applying the pivot_summary function to the match_summary and miss_summary DataFrames

- match_summaryT = pivot_summary(match_summary): calls the pivot_summary function with the match_summary DataFrame as input
- The function pivots and transforms the match_summary DataFrame
- The resulting transformed DataFrame is assigned to match_summaryT
- miss_summaryT = pivot_summary(miss_summary): calls the pivot_summary function with the miss_summary DataFrame as input
- The function pivots and transforms the miss_summary DataFrame
- The resulting transformed DataFrame is assigned to miss_summaryT

```
[ ]: match_summaryT = pivot_summary(match_summary)
miss_summaryT = pivot_summary(miss_summary)
```

Creating temporary views for the transformed DataFrames

- .createOrReplaceTempView("match_desc"): creates a temporary view named "match_desc" from the match_summaryT DataFrame
- .createOrReplaceTempView("miss_desc"): creates a temporary view named "miss_desc" from the miss_summaryT DataFrame
- The temporary views allow running SQL queries on the DataFrames using spark.sql()

Running a SQL query on the temporary views

- spark.sql(): executes a SQL query on the temporary views "match_desc" and "miss_desc"
- The SQL query:
- Selects the "field" column from the "match_desc" view aliased as "a"
- Calculates the total count by adding the "count" columns from both views using a.count + b.count

- Calculates the difference in means between the views using `a.mean - b.mean`
- Joins the "match_desc" and "miss_desc" views based on the "field" column using `INNER JOIN`
- Filters out the "id_1" and "id_2" fields using `WHERE a.field NOT IN ("id_1", "id_2")`
- Orders the result by the calculated delta in descending order and then by the total count in descending order using `ORDER BY delta DESC, total DESC`

[ ]:
```
match_summaryT.createOrReplaceTempView("match_desc")
miss_summaryT.createOrReplaceTempView("miss_desc")
spark.sql("""
SELECT a.field, a.count + b.count total, a.mean - b.mean delta
FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
WHERE a.field NOT IN ("id_1", "id_2")
ORDER BY delta DESC, total DESC
""")
```

Creating a sum expression for selected features
- `good_features`: a list of selected feature column names
- `" + ".join(good_features)`: concatenates the feature column names with " + " as the separator
- The resulting `sum_expression` is a string that represents the sum of the selected features

Displaying the sum expression
- The `sum_expression` variable contains the string representation of the sum of the selected features
- The expression will be in the format: "cmp_lname_c1 + cmp_plz + cmp_by + cmp_bd + cmp_bm"

[ ]:
```
good_features = ["cmp_lname_c1", "cmp_plz", "cmp_by", "cmp_bd", "cmp_bm"]
sum_expression = " + ".join(good_features)
sum_expression
```

Importing the `expr` function from `pyspark.sql.functions`
- `expr`: used to create a column expression from a string

Creating a new DataFrame with a score column
- `.fillna(0, subset=good_features)`: fills null values with 0 for the selected feature columns
- `.withColumn('score', expr(sum_expression))`: adds a new column named 'score' calculated using the `sum_expression`
- `expr(sum_expression)`: evaluates the `sum_expression` string as a column expression
- `.select('score', 'is_match')`: selects only the 'score' and 'is_match' columns from the resulting DataFrame
- The resulting DataFrame is assigned to the variable `scored`

Displaying the `scored` DataFrame
- `.show()`: displays the first 20 rows of the `scored` DataFrame
- The output will show the 'score' and 'is_match' columns for each row

[ ]:
```
from pyspark.sql.functions import expr
scored = parsed.fillna(0, subset=good_features).withColumn('score',␣
 ↪expr(sum_expression)).select('score', 'is_match')
```

```
scored.show()
```

Defining a function to create a cross-tabulation DataFrame
- `crossTabs(scored: DataFrame, t: DoubleType) -> DataFrame`: takes a DataFrame (`scored`) and a threshold value (`t`) as input and returns a cross-tabulation DataFrame
- The function performs the following steps:
1. `.selectExpr(f"score >= {t} as above", "is_match")`: selects the 'is_match' column and creates a new boolean column named 'above' indicating whether the 'score' is greater than or equal to the threshold `t`
2. `.groupBy("above")`: groups the DataFrame by the 'above' column
3. `.pivot("is_match", ("true", "false"))`: pivots the DataFrame based on the 'is_match' column, creating columns for 'true' and 'false' values
4. `.count()`: counts the number of occurrences for each combination of 'above' and 'is_match' values
- The resulting cross-tabulation DataFrame is returned

[ ]:
```
def crossTabs(scored: DataFrame, t: DoubleType) -> DataFrame:
    return scored.selectExpr(f"score >= {t} as above", "is_match").
 ↪groupBy("above").pivot("is_match", ("true", "false")).count()
```

Calling the `crossTabs` function with a threshold of 4.0 and displaying the result
- `crossTabs(scored, 4.0)`: calls the `crossTabs` function with the `scored` DataFrame and a threshold value of 4.0
- The function creates a cross-tabulation DataFrame based on the 'score' and 'is_match' columns, using a threshold of 4.0 for the 'above' column
- `.show()`: displays the resulting cross-tabulation DataFrame

The output will be a cross-tabulation DataFrame with the following columns:
- 'above': indicates whether the 'score' is greater than or equal to 4.0 (true or false)
- 'true': count of occurrences where 'is_match' is true
- 'false': count of occurrences where 'is_match' is false

The DataFrame will have two rows:
- Row 1: counts for 'score' values less than 4.0
- Row 2: counts for 'score' values greater than or equal to 4.0

[ ]:
```
crossTabs(scored, 4.0).show()
```

Calling the `crossTabs` function with a threshold of 2.0 and displaying the result
- `crossTabs(scored, 2.0)`: calls the `crossTabs` function with the `scored` DataFrame and a threshold value of 2.0
- The function creates a cross-tabulation DataFrame based on the 'score' and 'is_match' columns, using a threshold of 2.0 for the 'above' column
- `.show()`: displays the resulting cross-tabulation DataFrame

The output will be a cross-tabulation DataFrame with the following columns:
- 'above': indicates whether the 'score' is greater than or equal to 2.0 (true or false)
- 'true': count of occurrences where 'is_match' is true
- 'false': count of occurrences where 'is_match' is false

The DataFrame will have two rows:
- Row 1: counts for ‘score’ values less than 2.0
- Row 2: counts for ‘score’ values greater than or equal to 2.0

This cross-tabulation provides a summary of the counts for different combinations of ‘above’ and ‘is_match’ values, allowing for analysis of the relationship between the ‘score’ and ‘is_match’ columns based on the specified threshold of 2.0.

[ ]:
```
crossTabs(scored, 2.0).show()
```