DE Task

Ricardo Holthausen

1. Preprocessing

- Elements from the supplier data are *grouped by* their ID field, and then aggregated using *first*.
- When the data is in the “Attribute Names” / ”Attribute Values” fields, they are extracted in this way:

```
first(
    when(
        col("Attribute Names").contains("BodyTypeText"),
        col("Attribute Values"),
    ),
    ignorenulls=True,
).alias("BodyTypeText"),
```

2. Normalization (I)

- Two UDF were implemented for the normalization of required fields.
- Two auxiliary python dictionaries were created (based on info obtained through Exploratory Data Analysis) to keep the logic simple.

```python
@udf(returnType=StringType())
def normalize_make(text: str) -> str:
    if not text:
        return text
    if len(text) < 4:
        return text.upper()
    if text.lower() in SPECIAL_CASES:
        return SPECIAL_CASES[text.lower()]
    words = re.split(r"(\W)", text)
    return "".join([item.capitalize() for item in words])
```

2. Normalization (II)

- Normalization step is performed by a function that uses *withColumn* function

```
def normalize_df(df: DataFrame, attributes: List[str]) -> DataFrame:
    return df.withColumn(attributes[0], normalize_make(df[attributes[0]])).withColumn(
        attributes[1], normalize_color(df[attributes[1]])
    )
```

3. Extraction

- Similarly to Normalization step, two UDF were defined, and applied by means of *withColumn*

```
def extract_value_and_unit(df: DataFrame) -> DataFrame:
    return df.withColumn(
        "extracted-value-ConsumptionTotalText",
        extract_value_from_consumptiontotaltext(df["ConsumptionTotalText"]),
    ).withColumn(
        "extracted-unit-ConsumptionTotalText",
        extract_unit_from_consumptiontotaltext(df["ConsumptionTotalText"]),
    )
```

4. Integration

- To integrate the extracted data to the target data format, *select* was used:

```
def integrate(input_df: DataFrame) -> DataFrame:
    if len(input_df.head(1)) == 0:
        return input_df

    return input_df.select(
        col("BodyColorText").alias("color"),
        col("MakeText").alias("make"),
        col("ModelText").alias("model"),
        col("ModelTypeText").alias("model_variant"),
        col("City").alias("city"),
    )
```

5. Product matching (I)

- TF-IDF (Term Frequency - Inverse Document Frequency) and cosine similarity can be used to perform the matching.
- TF-IDF is normally used for studying word relevance in texts. We can assume each field in the target data and the integrated supplier data is a word (or generate n-grams for each record, assuming each record is a sentence), then obtain the TF-IDF values vectors for each record and compare them using cosine similarity.

5. Product matching (II)

- In the end we would have for each pair of values (this approach's complexity is worse than quadratic, as it implies multiplying two matrices) a cosine similarity value is obtained. We can then keep the pairs whose cosine similarity is above a given threshold (v.g.: 0.8).
- Besides, improvements on the performance can be obtained by using sparse-matrix implementations.

5. Product Matching (III)

- Another issue is matching records for which we have additional fields (for instance, supplier data has information about the interior color, or number of seats). To avoid this we can (1) use the intersection of data available or (2) use a lower threshold (increasing the probability of having false positive matchings).