Entity Resolution

- Import the necessary libraries.
- Create a Spark session, which is the entry point for all Spark applications. This allows users to perform operations over distributed datasets

In [ ]:
```
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

spark = SparkSession.builder.config("spark.driver.memory", "16g").appName('chapter_
```

Read the data from the csv file specified by the path and convert it into a DataFrame

In [ ]:
```
prev = spark.read.csv("data/linkage/donation/block_1/block_1.csv")
prev
```

Display the first two rows of the DataFrame created

In [ ]:
```
 prev.show(2)
```

- spark.read function allows reading data from several sources
- the option function allows the read options to be set by the user.
  - the given data contains a header
  - null values in the data are replaced with '?'
  - the schema is automatically inferred from the input data
- the .csv function specifies that the data is of csv format being read from the given path

In [ ]:
```
 parsed = spark.read.option("header", "true").option("nullValue", "?").\
option("inferSchema", "true").csv("data/linkage/donation/block_1/block_1.csv")
```

Display the schema of the DataFrame and print the first 5 rows

In [ ]:
```
parsed.printSchema()
parsed.show(5)
```

Display the number of rows in the DataFrame

In [ ]:
```
parsed.count()
```

Cache the DataFrame to memory. This stores the intermediate computations/state of the dataframe in the cluster node's memory so the can be reused for subsequent actions. This prevents repeated computations. The .cache() method on a DataFrame saves it to the default storage level 'MEMORY_AND_DISK'

In [ ]:
```
parsed.cache()
```

Perform an aggregate operation and display:

- Group the rows of the dataframe by the distinct values of the 'is_match' column
- count the number of rows for each group
- order the groups by descending order of the count

This displays the number of entities in the DataFrame that are matched correctly and incorrectly

In [ ]:
```
from pyspark.sql.functions import col
parsed.groupBy("is_match").count().orderBy(col("count").desc()).show()
```

The createOrReplaceTempView() function creates a temporary view of the DataFrame if one does not exist, otherwise replaces any existing one. The view exists only for the life of the spark session. SQL queries can be run on the views to select or manipulate data, without the results being materialized as a permanent table.
The sql function allows the SQL code to be specified. This is run on the views and tables specified in the FROM cluase of the query and the results are displayed. In this case, the SQL query performs the same function as the previous cell, grouping the data by distinct values of the 'is_match' column and displaying the count of each value in descending order.

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

The describe() function provides summary statistics for all numeric columns of the dataframe, such as minimum and maximum values, mean, standard deviation. These are stored in a DataFrame named summary. This creates a column for each variable along with a 'summary' column which describes the metric that is present.

In [ ]:
```
summary = parsed.describe()
```

The summary statistics for the columns 'cmp_fname_c1' and 'cmp_fnamec2' are displayed by selecting the necessary columns from the summary table

In [ ]:
```
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()
```

The parsed dataframe is filtered to store only the rows with the value True in the 'is_match' column in the matches table and rows with value False in the 'is_match' column in the misses table. For each table, the summary statistics are stored.

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

The summary table is converted to a Pandas dataframe using the toPandas() function. On the pandas dataframe, the table can be manipulated

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

Display the first 5 rows of the dataframe and its shape (no. of rows, no. of columns).

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

The dataframe is manipulated to:

- set the 'summary' column of the dataframe as the index
- transpose the dataframe, that is, replace the rows and the columns
- reset the indices of the transposed dataframe back to the default values of 0, 1, 2...
- rename the 'index' column of the transposed dataframe to 'field'
- rename axis 1, that is, the column axis to None (removes the previous name of the column axis)

The shape of this new dataframe is then displayed

This set of functions converts the summary table, so that each row displays one variable with each column representing the calculated metric for that variable.

In [ ]:
```
 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
```

The manipulated pandas dataframe is converted back to a Spark DataFrame and the schema is displayed

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

For each numeric column in the dataframe, as in the .columns attribute of the dataframe, the column is casted to be of double type, and replace the original column, which was of type string. The withColumn function adds a new column to the dataframe as a result of some operation, in this case .cast() on an existing column. Here, since both the existing and newly computed columns have the same name, the existing column is replaced with the newer one.
This is done for all columns except 'field' which contains inherently data of string type. The results of this will be displayed in the new schema of the dataframe, with all the numeric columns converted to type double

In [ ]:
```
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()
```

The proceesses carried in the above few cells are consolidated into a single function, which

- takes as input a summary dataframe
- converts it to pandas
- transposes it, resets the indices and renames the columns
- converts it back to a spark dataframe
- converts all numeric fields to double type

It then returns the new dataframe

In [ ]:
```
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 desc
```

Both the miss_summary and match_summary dataframes are pivoted and reshaped using the pivot_summary function defined above

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

Temporary views are created for both the match and miss summary pivoted tables.

- An SQL query is then executed on the join of the two tabales, joined so that the 'field' column of both tables match. Inner join only takes the rows of both tables where there is a match
- It then displays the sums of the count values for matched rows of the tables and the difference of the means, for all rows excluding 'id1' and 'id2', ordering it in descending order of the difference field

This is used to select good features that determine whether two entities are the same, as the good features usually have significantly different values for matches and nonmatches (large difference in means) and they occur often enough in the data (large value of count)

In [ ]:
```
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
""")
```

Using the results of the previous query, good features are selected and an string expression is created to describe the sum of all the good features

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

A new table, scored is created from the parsed table, using only a subset of the columns corresponding to the good features. A new column, 'score' is added, computed by summing the value for each of these columns in the row. Finally, only the 'score' and 'is_match' column are included in the new table.

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

Finally, a suitable threshold needs to be selected so that scores above this threshold can be classified as matches and scores below this are classified as nonmatches. The threshold must be such that when the rows are classified using it, there should be maximum accuracy. For this a function is defined which creates a Cross Tabulation (essentially, a confusion matrix) which counts the number of rows whose rows fall above or below a certain threshold along with the number of rows in each of those categories (above/below) that were or were not a match respectively. This can allow comparing and choosing a good value of the threshold

In [ ]:
```
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()
```

In [ ]:
```
crossTabs(scored, 4.0).show()
crossTabs(scored, 2.0).show()
```

Cross tabulations are calculated for thresholds of 4 and 2. A high threshold filters out almost all nonmatches and most of the matches. A lower threshold captures all matches, but has a much higher count of false positives.

In [ ]:
```
```