DATA CLEANING TECHNIQUES - Big Data

Have you ever heard sentences like “Hey ! This data has a lot of null values”, “The schema of the data is not matching with the source!!”, “Oh no!! This data is not cleaned and unfit for processing”, working as a data engineer might familiarise you with these nuances. Does this make you wonder, whether data cleaning is actually important in real time? If you have thought for a second then let me make you understand its importance along with a few live examples.

Data cleaning is one of the primary responsibilities of a data engineer as they are responsible to hand over clean and processed data to their downstream like analytics and data science teams. Data cleaning is one of the very initial processes to handle bad, unstructured and unorganized data at the very starting stages of building your data pipeline. If the raw data ingested is not cleaned thoroughly and got a sign-off to be used by the downstream, it has the power to create chaos by rendering wrong reports and disrupting your training models by rendering wrong results.

Data cleaning helps to identify and correct errors, inconsistencies, and inaccuracies in the data. These errors can arise due to missing data, data ingestion issues and data entry mistakes. By cleaning the data, data engineers can ensure that the data is accurate and consistent across all sources, making it easier to analyze and draw insights from. Moreover, data cleaning can also help to standardize the data format, such as converting dates to a consistent format or removing special characters and whitespace. This ensures that the data is easier to read, manipulate, and analyze.

In this blog post, we will make use of the spark framework which provides a powerful toolset for data cleaning, with built-in functions and libraries that make the process fast, efficient and effortless using pyspark queries.

Below is the sample dataset on which we perform the data cleaning to make it usable and consumable in data pipelines. The dataset is referenced from the recent payment transactions of customers.

Table +

	loan_id	member_id	total_payment_recorded	installment	last_payment_amount	last_payment_date	next_payment_date
1	LN1	MEM1	741.6 EUR	373.63 USD	373.63	9/18/2022	10/18/2022
2	LN2	MEM2	882.34 EUR	455.68 USD	455.68	9/18/2022	10/18/2022
3	LN3	MEM3	586.25 EUR	301.34 USD	301.34	9/18/2022	10/18/2022
4	LN4	MEM4	486.73 EUR	246.4 USD	null	9/18/2022	10/18/2022
5	LN5	MEM5	1061.98 EUR	552.34 USD	552.34	9/18/2022	10/18/2022
6	LN6	MEM6	413.65 EUR	212.31 USD	212.31	9/18/2022	10/18/2022
7	null	MEM7	666.99 EUR	314.07 USD	314.07	null	10/18/2022

Initial Steps:

Read the CSV data file into a data frame to make it ready for the data cleaning process.

```
import requests
import json
from pyspark.sql.functions import udf
from pyspark.sql.functions import regexp_replace,coalesce,lit,to_date
from pyspark.sql.types import StructType,StructField,FloatType,DateType,StringType

```

```
payment_schema =StructType(fields=[StructField("loan_id", StringType(), False),
                                   StructField("member_id", StringType(), False),
                                   StructField("total_payment_recorded", StringType(), True),
                                   StructField("installment", StringType(), True),
                                   StructField("last_payment_amount", FloatType(), True),
                                   StructField("last_payment_date", StringType(), True),
                                   StructField("next_payment_date", StringType(), True),

                                  ])
```

```
df_payment = spark.read \
.format("csv") \
.option("header", "true") \
.schema(payment_schema) \
.load("dbfs:/FileStore/shared_uploads/dataCleaning_blog_.csv")
df_payment.display()
```

▸ (1) Spark Jobs

▸ df_payment: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

Step 01: Take charge of NULL values

Ah! What is the better way to start our data cleaning by taking charge of the NULL values? when the data is queried we got to know the data is filled with many “null” typed strings which can be converted to pure null values (None datatype) or replaced by empty strings “ “. Here we convert them to null values which can be easily queryable and help in the analysis to track down the records.

1. Replace "null" strings to null values

Cmd 6

```
df_null=df_payment.replace("null",None)
df_null.createOrReplaceTempView("df_null_values")
spark.sql("select * from df_null_values").display()
```

▸ (1) Spark Jobs

▸ df_null: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

Table +

	loan_id	member_id	total_payment_recorded	installment	last_payment_amount	last_payment_date	next_payment_date
1	LN1	MEM1	741.6 EUR	373.63 USD	373.63	9/18/2022	10/18/2022
2	LN2	MEM2	882.34 EUR	455.68 USD	455.68	9/18/2022	10/18/2022
3	LN3	MEM3	586.25 EUR	301.34 USD	301.34	9/18/2022	10/18/2022
4	LN4	MEM4	486.73 EUR	246.4 USD	null	9/18/2022	10/18/2022
5	LN5	MEM5	1061.98 EUR	552.34 USD	552.34	9/18/2022	10/18/2022
6	LN6	MEM6	413.65 EUR	212.31 USD	212.31	9/18/2022	10/18/2022
7	null	MEM7	666.99 EUR	314.07 USD	314.07	null	10/18/2022

10 rows | 0.19 seconds runtime

Step 02: Remove the non-numerics from the data frame

If you have noticed it keenly your eyes would have caught attention to the columns which have the currency type appended to them like instalment, total_payment_recorded. This can hamper the processing and querying as they are of string type which needs to be corrected.

Don't forget to cast the schema of these columns to FLOAT values as they are stored as STRINGS before. This can again avoid errors popping up when trying to query these columns. Can you optimize this code in a better way? Do post in the comments and let us know.

2. Remove the non-numeric values from the dataframe

Cmd 8

```
# remove non-numeric characters from 'total_payment_recorded' column
df_installment = df_null.withColumn('installment', regexp_replace('installment', '[^0-9\.]', '').cast(FloatType()))
df_eur.show()
```

(1) Spark Jobs

df_installment: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

```
# remove non-numeric characters from 'total_payment_recorded' column
df_eur = df_installment.withColumn('total_payment_recorded', regexp_replace('total_payment_recorded', '[^0-9\.]', '').cast(FloatType()))
df_eur.show()
```

(1) Spark Jobs

df_eur: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

```
+-------+---------+---------------------+-----------+------------------+----------------+----------------+
|loan_id|member_id|total_payment_recorded|installment|last_payment_amount|last_payment_date|next_payment_date|
+-------+---------+---------------------+-----------+------------------+----------------+----------------+
|    LN1|     MEM1|                741.6|     373.63|            373.63|       9/18/2022|       10/18/2022|
|    LN2|     MEM2|               882.34|     455.68|            455.68|       9/18/2022|       10/18/2022|
|    LN3|     MEM3|               586.25|     301.34|            301.34|       9/18/2022|       10/18/2022|
|    LN4|     MEM4|               486.73|      246.4|              null|       9/18/2022|       10/18/2022|
|    LN5|     MEM5|              1061.98|     552.34|            552.34|       9/18/2022|       10/18/2022|
|    LN6|     MEM6|               413.65|     212.31|            212.31|       9/18/2022|       10/18/2022|
|   null|     MEM7|               666.99|     314.07|            314.07|            null|       10/18/2022|
|    LN8|     MEM8|              1880.26|     984.66|            984.66|            null|       10/18/2022|
|    LN9|     MEM9|                null|      216.8|             216.8|            null|       10/18/2022|
|   LN10|    MEM10|               304.21|      154.0|              154.0|            null|       10/18/2022|
+-------+---------+---------------------+-----------+------------------+----------------+----------------+
```

Step 03: Standardize the currency of the payments

We are considering the base currency to be USD in our example due to which the payment data we received in other currencies need to be converted into base currency to maintain consistency in the data throughout. In our example, we are

getting the exchange rate of EUR to USD dynamically on a daily basis from a free exchange rate API service and converting our existing data frame column value into USD value. There are other ways in which you can implement this function only for the incremental load of data based on the ingest date. For now, we will apply this function on the entire data frame column of 'total_payment_recorded'

3. Convert the EUR to USD using UDF for column 'total_payment_recorded'

Cmd 12

```
#API call to fetch exchange rate
def get_exchange_rate():
    url = "https://api.apilayer.com/currency_data/live?source=eur&currencies=usd"
    payload = {}
    headers= {
    "apikey": "your_api_key"
    }

    response = requests.request("GET", url, headers=headers, data = payload)
    status_code = response.status_code
    result = response.text
    data = json.loads(result)
    if 'quotes' in data and 'EURUSD' in data['quotes']:
        exchange_rate=data['quotes']['EURUSD']
        return exchange_rate
    else:
        print('Failed to extract exchange rate from JSON data')

def eur_to_usd(eur_exchange):
    if eur_exchange is None:
        return 0
    else:
        exchange_rate = get_exchange_rate()
        return (eur_exchange) * exchange_rate
```

```
def eur_to_usd(eur_exchange):
    if eur_exchange is None:
        return 0
    else:
        exchange_rate = get_exchange_rate()
        return (eur_exchange) * exchange_rate

# define the UDF
eur_to_usd_udf = udf(eur_to_usd, FloatType())

# apply the UDF to the 'total_payment_recorded' column
df_usd_conversion = df_eur.withColumn('total_payment_recorded', eur_to_usd_udf(df_eur['total_payment_recorded']))

df_usd_conversion.show()
```

▸ (1) Spark Jobs

▸ df_usd_conversion: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

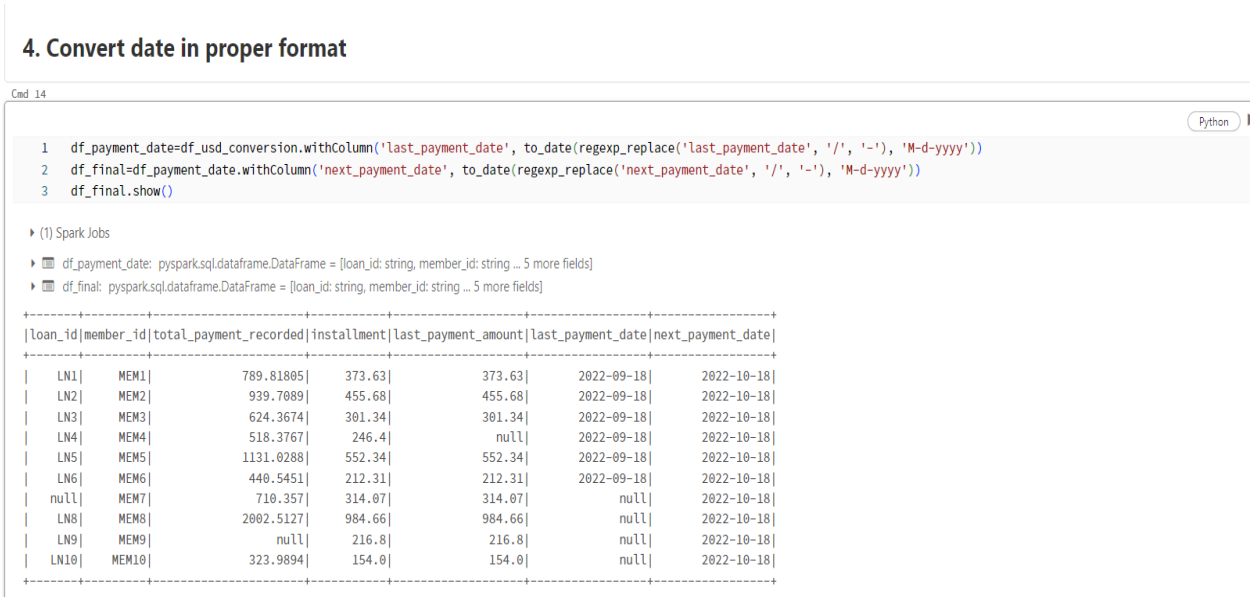
▸ (1) Spark Jobs

▸ df_usd_conversion: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

loan_id	member_id	total_payment_recorded	installment	last_payment_amount	last_payment_date	next_payment_date
LN1	MEM1	789.81805	373.63	373.63	9/18/2022	10/18/2022
LN2	MEM2	939.7089	455.68	455.68	9/18/2022	10/18/2022
LN3	MEM3	624.3674	301.34	301.34	9/18/2022	10/18/2022
LN4	MEM4	518.3767	246.4	null	9/18/2022	10/18/2022
LN5	MEM5	1131.0288	552.34	552.34	9/18/2022	10/18/2022
LN6	MEM6	440.5451	212.31	212.31	9/18/2022	10/18/2022
null	MEM7	710.357	314.07	314.07	null	10/18/2022
LN8	MEM8	2002.5127	984.66	984.66	null	10/18/2022
LN9	MEM9	null	216.8	216.8	null	10/18/2022
LN10	MEM10	323.9894	154.0	154.0	null	10/18/2022

Step 04: Standardize the date format

We can follow a standard date format across our data pipelines and data mart to ensure everyone is aligned with the date format being followed and can reframe their queries accordingly to avoid failures.



4. Convert date in proper format

Cmd 14

Python

```
1  df_payment_date=df_usd_conversion.withColumn('last_payment_date', to_date(regexp_replace('last_payment_date', '/', '-'), 'M-d-yyyy'))
2  df_final=df_payment_date.withColumn('next_payment_date', to_date(regexp_replace('next_payment_date', '/', '-'), 'M-d-yyyy'))
3  df_final.show()
```

▸ (1) Spark Jobs

▸ df_payment_date: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

▸ df_final: pyspark.sql.dataframe.DataFrame = [loan_id: string, member_id: string ... 5 more fields]

loan_id	member_id	total_payment_recorded	installment	last_payment_amount	last_payment_date	next_payment_date
LN1	MEM1	789.81805	373.63	373.63	2022-09-18	2022-10-18
LN2	MEM2	939.7089	455.68	455.68	2022-09-18	2022-10-18
LN3	MEM3	624.3674	301.34	301.34	2022-09-18	2022-10-18
LN4	MEM4	518.3767	246.4	null	2022-09-18	2022-10-18
LN5	MEM5	1131.0288	552.34	552.34	2022-09-18	2022-10-18
LN6	MEM6	440.5451	212.31	212.31	2022-09-18	2022-10-18
null	MEM7	710.357	314.07	314.07	null	2022-10-18
LN8	MEM8	2002.5127	984.66	984.66	null	2022-10-18
LN9	MEM9	null	216.8	216.8	null	2022-10-18
LN10	MEM10	323.9894	154.0	154.0	null	2022-10-18

Now our data frame is cleaned and ready to be stored as a hive table to be made accessible across teams to work on it and process it further for their needs. This is just an example taken up to help you to understand some of the cleaning techniques adopted by data engineers to clean up their data before it can be staged into the phase of data processing/ data transformation. Can you name the cleaning techniques you are aware of in the comments below?