Cowlar Recruitment

Data Engineering - Technical Screening Task – SAMA Team

Saif Ullah
Senior SWE Student, NUST
Senior Vice President, ACM NUST
Brand Ambassador, Tkxel & Vyro
LinkedIn | GitHub

Table of Contents

TASK 1: SQL

Scenario:

You have been provided with a dataset named `customer orders.` This dataset contains information about customer orders in an e-commerce platform.

Dataset Columns:

Feature Name	Description
order_id	Unique ID for the order
customer_id	Unique ID for the customer
customer_name	Customer Name
shipment_date	Product shipment date
product_id	Unique product ID
order_date	Product Order date
product_description	Description for Product
quantity	Quantity of product ordered
unit_price	Unit price for a single quantity of product

Preliminary Script

This script performs all the necessary tasks before we can start playing with the given data in SQL. Following steps are taken to successfully execute the tasks' answers:

1. It will create the database "`cowlar`" only if it does not exist.
2. Then creates the table with required column names, data types, and storage size.
3. Then the data is imported from the given CSV file.
4. Some wrangling operations are performed in this script i.e., changing date columns from string to date data type & removing formatting commas from `unit_price` column.

scripts/create db sql.sql:

```sql
CREATE DATABASE IF NOT EXISTS cowlar;
USE cowlar;

CREATE TABLE customer_dataset(
    customer_id VARCHAR(10),
    customer_name VARCHAR(50),
    order_id INT,
    order_date DATE,
    shipment_date DATE,
    product_id VARCHAR(15),
    product_description VARCHAR(100),
    quantity INT,
    unit_price FLOAT
);

-- place your csv file on the given filepath to make sure everything runs
smoothly
LOAD DATA INFILE "C:\\ProgramData\\MySQL\\MySQL Server
8.0\\Uploads\\customer_dataset.csv"
INTO TABLE customer_dataset
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 LINES
(customer_id, customer_name, order_id, @order_date_csv, @shipment_date_csv,
product_id, product_description, quantity, @unit_price_csv)
SET order_date = STR_TO_DATE(@order_date_csv, '%d/%m/%Y'),
shipment_date = STR_TO_DATE(@shipment_date_csv, '%d/%m/%Y'),
unit_price = REPLACE(@unit_price_csv, ',', '');
```

Query Optimization

This query provides the top 5 customers who made the highest total purchases.

Task 1/T1 (i).sql:

```sql
SELECT customer_id,
       SUM(quantity * unit_price) AS total_purchases  --purchases per product
FROM customer_dataset
GROUP BY customer_id
ORDER BY total_purchases DESC
LIMIT 5;
```

Output:

customer_id	total_purchases
2SUR03	4856782.902893066
2SER01	2001300.9700775146
4DAI02	203695.39999389648
2RFD01	98230
6PRI02	97753.28024291992

Data Transformation

This query calculates the total revenue for each product and displays them in descending order.

Task 1/T1 (ii).sql:

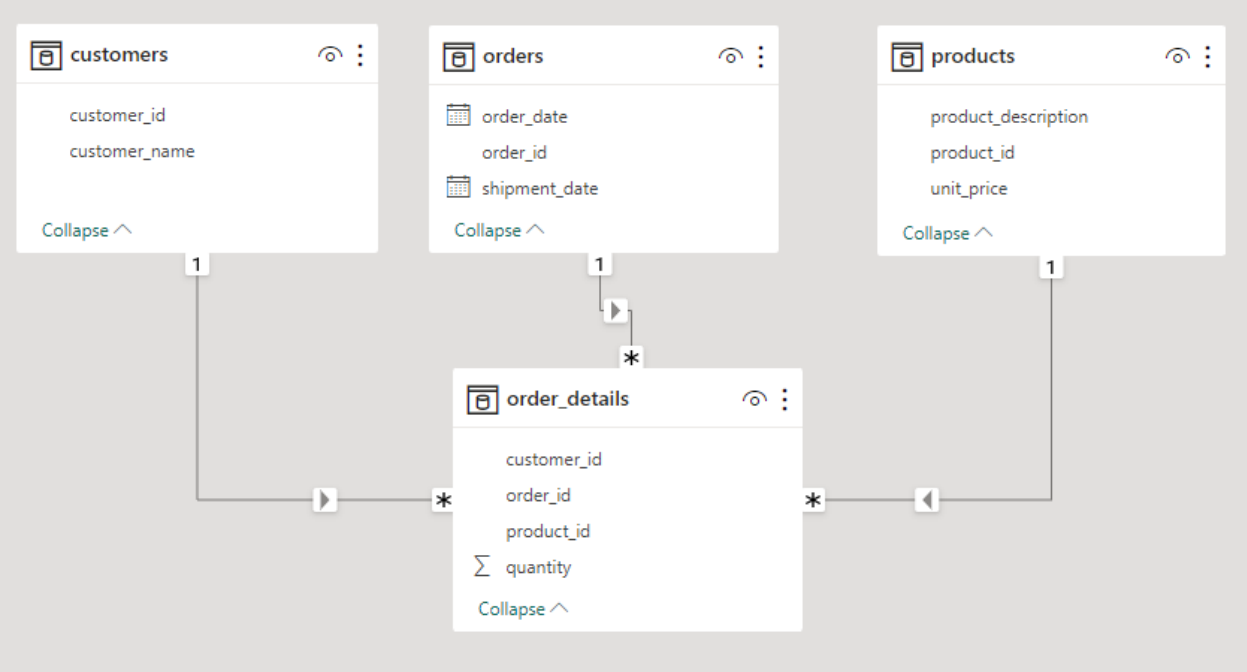
```
SELECT product_id,
       SUM(unit_price * quantity) AS total_revenue  --revenue per product
FROM customer_dataset
GROUP BY product_id
ORDER BY total_revenue DESC;
```

Output:

product_id	total_revenue
06959009	3000000
41423001	885158.3999633789
06857009	796400
05163009	736800
08220009	240000
05606009	200161.50000095367
00002123	194147.5
20883001	145301.99999809265
02136011	104000
21360011	99550
00001011	93060
08413009	78587.00170516968
00013628	66900
06231001	62400

Database Design

In this scenario, the Star Schema is used to store information. The resultant normalized database design about customers is given below. The entities, attributes, and the type of relationships can be observed here:



Indexing

Indexing is used to optimize database design and performance. This is achieved by indexing necessary columns only i.e., columns frequently used in grouping, sorting, conditional retrieval or joining two tables, to optimize space requirements of the system.

Significance of Indexing can be observed here:

1. Accelerated Data Retrieval using WHERE Clause
2. Increased Performance for ORDER BY Clause
3. Improved Aggregations Performance
4. Support for GROUP BY Clause
5. Enhanced Join Performance

Columns in customer_orders table to be indexed for efficient querying:

1. `order_id`
2. `customer_id`
3. `product_id`
4. date columns i.e., `order_date & shipment_date`

The primary keys for every table in the normalized database design can be indexed for efficient querying. The dates columns are one of the many columns that hold a higher potential of being used more frequently for querying & reporting purposes. Hence date columns can be indexed too.

TASK 2: MongoDB

Scenario:

You are working with a MongoDB database that stores information about orders and products in an e-commerce platform. For this use the same dataset provided for Task 1. (customer_orders.)

Preliminary Script

This script performs all the necessary tasks to be done before executing tasks' answers. The following steps are taken in this script to make sure the data is imported properly.

1. Imports the CSV file into a temporary collection in a database on the local connection string.
2. NOTE: The database & the collection do not need to be created beforehand.
3. The next part creates another collection to store the clean data i.e., the output of the aggregate query on this temporary collection.
4. Similar wrangling operations are applied in this script as were applied in the preliminary SQL script.

scripts/create mongo db.sh:

```
# NOTE:
# Replace "cowlar", "temp_collection", "data/customer_dataset.csv" with your db-name, temporary-collection-name, csv-filepath respectively

# creating the db, and a temporary collection to import csv
mongoimport mongodb://localhost:27017/ --db cowlar --collection temp_collection --type csv --file data/customer_dataset.csv --headerline;

# This script runs the mongo shell commands to clean the imported data and semi-normalize it
# & store in the customers_dataset collection
mongosh mongodb://localhost:27017/cowlar --eval '
    // creating collection to store clean data
    db.createCollection("customers_dataset");

    // cleaning and semi-normalizing data
    db.temp_collection.aggregate(
        [
            {
                $group: {
                    _id: {
                        order_id: "$order_id",
```

```
                customer_id: "$customer_id",
                customer_name: "$customer_name",
            },
            products: {
                $push: {
                    product_id: {
                        $toString: "$product_id"
                    },
                    product_description:
                        "$product_description",
                    quantity: "$quantity",
                    unit_price: {
                        $toDouble: {
                            $replaceAll: {
                                input: {
                                    $toString: "$unit_price",
                                },
                                find: ",",
                                replacement: "",
                            },
                        },
                    },
                },
            },
            order_date: {
                $first: {
                    $dateFromString: {
                        dateString: "$order_date",
                        format: "%d/%m/%Y",
                    },
                },
            },
            shipment_date: {
                $first: {
                    $dateFromString: {
                        dateString: "$shipment_date",
                        format: "%d/%m/%Y",
                    },
                },
            },
        },
    },
    {
        $group: {
            _id: {
```

```
                        customer_id: "$_id.customer_id",
                        customer_name: "$_id.customer_name",
                    },
                    orders: {
                        $push: {
                            order_id: "$_id.order_id",
                            order_date: "$order_date",
                            shipment_date: "$shipment_date",
                            products: "$products",
                        },
                    },
                },
            },
            {
                $project: {
                    _id: 0,
                    customer_id: "$_id.customer_id",
                    customer_name: "$_id.customer_name",
                    orders: 1,
                },
            },
            {
                $out: "customers_dataset",
            },
        ]
    );

    // cleaning up
    db.temp_collection.drop();
';
```

Schema Design

There can be multiple levels of normalization that can be achieved here in MongoDB. But there are two that we can discuss here:

1. Normalized Schema:

   This schema will contain the same 4 tables as mentioned in the normalized schema proposed in SQL above, which includes 3 dimension/lookup tables (`orders, customers, products`) and 1 fact table (`order_details`).

2. Semi-Normalized Schema:

   This schema will not divide the dataset into 4 separate tables. Instead, every document will contain all the information related to that document with minimizing redundancy of data. The structure of every document looks like follows:


```
{
    "_id": {
        "$oid": "65b104284ad4570e6953fd7e"
    },
    "orders": [
        {
            "order_id": 1051,
            "order_date": {
                "$date": "2014-01-20T00:00:00.000Z"
            },
            "shipment_date": {
                "$date": "2014-01-24T00:00:00.000Z"
            },
            "products": [
                {
                    "product_id": "6096009",
                    "product_description": "BATTERY SEACELL L8S",
                    "quantity": 50,
                    "unit_price": 55.45
                }
            ]
        }
    ],
        "customer_id": "3DAE01",
        "customer_name": "DAECO LTD"
}
```


Why Semi-Normalized Schema:

In this semi-normalized schema, data redundancy is reduced in a way that if a customer has multiple orders, the customer data will not be repeated, and if an order has multiple products, the corresponding customer data and the order data will not be repeated.

However, if two different customers have ordered same products, products data will be repeated in their documents.

Why can't we simply normalize the data the way we proposed in SQL?

That will require a lot of lookup queries (like JOIN in SQL) in MongoDB. That has a few cons mentioned below:

1. Lookup queries in MongoDB make your aggregate queries extremely complex, decreasing readability and understanding of the code over long periods of time.
2. Looking up data that is divided into such distinct structures is an expensive task in terms of resources required. Hence, this decision to choose one of the two proposed schemas depends on the reading vs. writing rate in our database.
3. Eventually, in queries involving multiple joins, the performance may be negatively impacted, especially for read-heavy workloads.

Hence, Semi-Normalized Schema is proposed at this stage for the given data.

The data types for the columns are:

Columns	Data Types
`_id`	ObjectID
`customer_id, customer_name, product_id, product_description`	String
`orders, products`	Array
`order_id, quantity`	Int32
`order_date, shipment_date`	Date
`unit_price`	Double

Model Creation

Model creation and predicting some target value using the data collected over the years is clearly a read-heavy task. Hence, to improve performance, we must ensure that the schema used in this database is semi-normalized design using nested array for orders and products for each customer.

Also, the columns used as features and target value are as follows:

Features	Target
order_date, quantity	Future order quantities

Knowing the attributes we have; we cannot predict any of these columns using other columns. We can predict the quantity ordered but for that, we need financial details of the customers or something similar.

We can still predict the quantity ordered but the only option we have, is to conduct a Time Series Analysis, using `order_date` column.

We can explore if the column quantity exhibits stationarity over the `order_date` column & then decide which one of the ARMA or ARIMA models can be used to predict the target here.

Query to update existing data

This MongoDB shell command updates the `shipment_date` for the `order_id` 1239.

Task 2/T2 (iii).sh:

```
# NOTE:
# Replace "cowlar", "customers_dataset" with your db-name & collection-name
respectively

mongosh mongodb://localhost:27017/cowlar --eval '
    db.customers_dataset.updateOne(
        {
            "orders.order_id": 1239
        },
        {
            $set: {
                "orders.$[elem].shipment_date": new Date("2014-08-31")
            }
        },
        {
            arrayFilters: [
                {
                    "elem.order_id": 1239
                }
            ]
        }
    )
';
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Query to Add New Data

This MongoDB shell command adds new data in the collection of the given data.

Task 2/T2 (iv).sh:

```
# NOTE:
# Replace "cowlar", "customers_dataset" with your db-name & collection-name
respectively

mongosh mongodb://localhost:27017/cowlar --eval '
    db.customers_dataset.insertOne(
        {
            customer_id: "1108SU",
            customer_name: "SAIF ULLAH",
            orders: [
                {
                    order_id: 9999,
                    order_date: new Date("2002-08-11"),
                    shipment_date: new Date("2002-08-11"),
                    products: [
                        {
                            product_id: "1108SAIF",
                            product_description: "PRODUCT BY SAIF ULLAH",
                            quantity: 11,
                            unit_price: 1108
                        }
                    ]
                }
            ]
        }
    )
';
```

Output:

```
{
  acknowledged: true,
  insertedId: ObjectId('65b243bba2743596b277e714')
}
```

TASK 3: Feature Engineering

Scenario:

You are given the "Titanic" dataset, which includes the following columns:

Feature Name	Description
`survived`	Binary (0 or 1), indicating whether the passenger survived (1) or not (0)
`pclass`	Ticket class (1st, 2nd or 3rd)
`name`	Passengers name
`sex`	Gender of the passenger
`age`	Age of the passenger
`sibsp`	Number of the siblings or spouses aboard
`parch`	Number of parent or children aboard
`ticket`	Ticker number
`fare`	Passenger fare
`cabin`	Cabin number
`embarked`	Post from which embarked (C, Q or S)

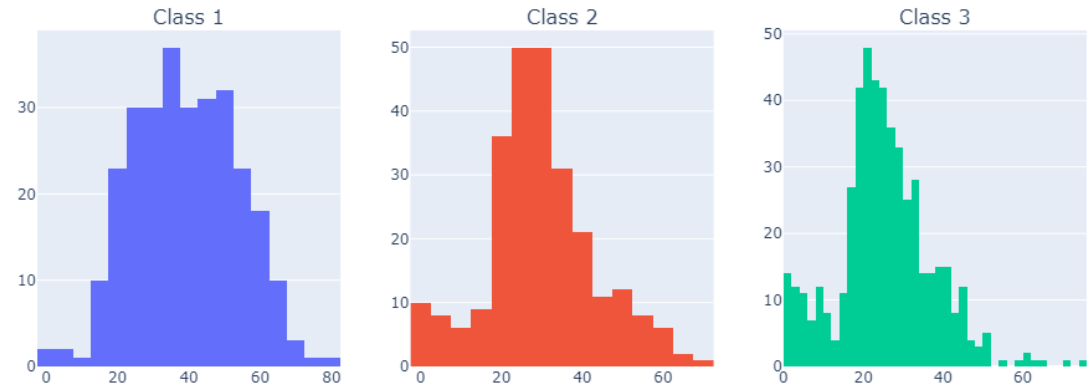
Data Cleaning

The whole data cleaning process is mentioned below with references to the notebook "`Task 3/T3 (i).ipynb`":

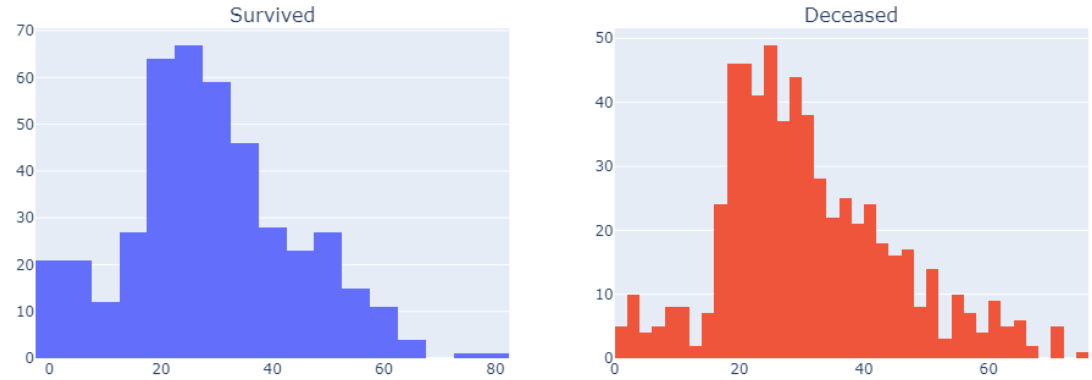
1. The `age` column has 264 null values. Age can be imputed by taking mean of the given column. But a more intelligent strategy can be to divide the age column into parts, based on other potential columns that age might be affected by. Then impute the subset of the age column by the mean of that certain subset.

Here we can see, columns like `pclass`, `survived`, `sex`, `embarked` show different age distributions for every different category in these columns:

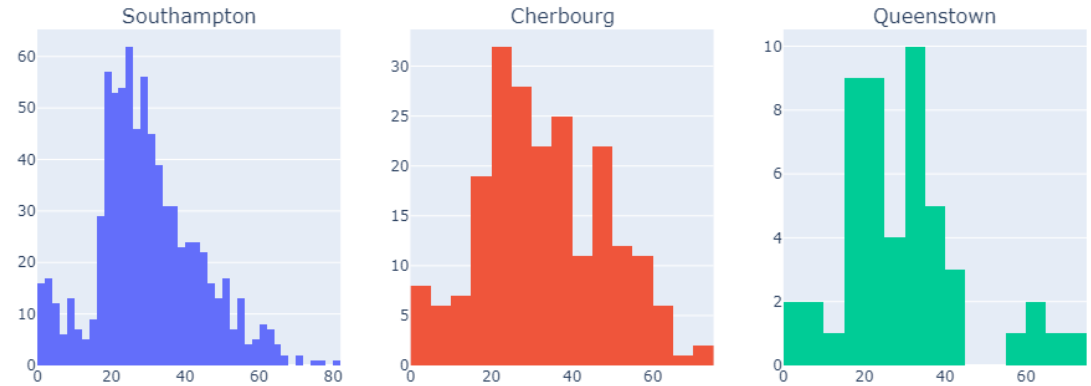
Age Distribution



Age Distribution



Age Distribution



Hence, age is imputed by the mean of the age column after grouping by these 4 columns as shown below:

```
age_mean = titanic_df.groupby(['pclass', 'survived', 'sex',
'embarked'])['age'].transform('mean')
titanic_df['age'].fillna(age_mean, inplace=True)
```

2. The `fare` column only has 2 null value. In such cases, we can just conduct some research to find the missing value. Here we have their ticket number & name, so after some research, I found the fare paid by the passenger and imputed the null value. [1]

```
# Few null values can be imputed after some research, if applicable
titanic_df['fare'].fillna(7.0, inplace=True)
```

3. The `embarked` column only has 3 null values. Like `fare` column, we can find the location these passengers embarked from & found that all 3 passengers embarked the journey from Southampton. [1]

```
# Similar strategy applied, found the location after some research
titanic_df['embarked'].fillna('S', inplace=True)
```

4. The `boat` column has 824 null values, out of which, only 23 null values lie in the survived subset of data. The remaining 800 null values represent the dead passengers. Since the `boat` column represents the number of life boat the passenger was found on, we realize that the `boat` column cannot have a value for the deceased passengers. Hence these null values are imputed as follows:

```
survived_mask = titanic_df['survived'] == 1.0
dead_mask = titanic_df['survived'] == 0.0

titanic_df.loc[survived_mask, 'boat'] =
titanic_df[survived_mask]['boat'].fillna('boat_unknown')
titanic_df.loc[dead_mask, 'boat'] =
titanic_df[dead_mask]['boat'].fillna('not_found')
```

5. The `body` column has 1188 null values. The remaining 121 values only represent the dead passengers. Since the `body` column represents the number of dead body found, we can impute these null values as follows:

```
titanic_df.loc[survived_mask, 'body'] =
titanic_df[survived_mask]['body'].fillna('survived')
titanic_df.loc[dead_mask, 'body'] =
titanic_df[dead_mask]['body'].fillna('not_found')
```

6. The columns like `cabin` and `home.dest` have 500+ null values each and hence cannot be imputed or dropped either. So, a new category for the null values can be formed and imputed here:

```
titanic_df['cabin'].fillna('unknown', inplace=True)
titanic_df['home.dest'].fillna('unknown', inplace=True)
```

7. Finally, the last row is completely empty so we can drop this example.

```
titanic_df = titanic_df.iloc[:-1]
```

Feature Extraction

Some new features were extracted from the existing features, addressed below:

1. The `name` column has names in the following format: "<family-name>, <title>. <name>". So, we can extract these three attributes from the `name` column.

```
# Splitting the name column into name & family-name.
split_name = titanic_df['name'].str.split(', ')
titanic_df['family_name'] = split_name.str[0]

# extracting the title from name column
split_title = split_name.str[1].str.split('. ', regex=False)
titanic_df['title'] = split_title.str[0] + '.'
titanic_df['name'] = split_title.str[1]
```

title	name	family_name
Miss.	Elisabeth Walton	Allen
Master.	Hudson Trevor	Allison
Miss.	Helen Loraine	Allison
Mr.	Hudson Joshua Creighton	Allison
Mrs.	Hudson J C (Bessie Waldo Daniels)	Allison

2. The column `alone` represents if a person was travelling alone or with a family member. Note: This does not include the maids or friends travelling on the same ticket.

```
titanic_df['alone'] = ((titanic_df['sibsp'] == 0) & (titanic_df['parch'] ==
0)).astype(int)
```

3. New column named `family_size` can be represented by adding the `sibsp` and `parch` column. We must also add the person whose family members we are calculating in this row.

```
titanic_df['family_size'] = titanic_df['sibsp'] + titanic_df['parch'] + 1
```

4. We can extract deck information from the `ticket` column.

```
titanic_df['deck'] = titanic_df['cabin'].str[0]
```

5. As discussed earlier, the number of persons travelling on a ticket is not equal to the number of family members of that person, because of the friends and maids people had with them. Hence, this can be a new column.

```
titanic_df['persons_on_ticket'] =
titanic_df.groupby('ticket').transform('count').iloc[:, 0]
```

6. The column `fare` gives us the fare paid for each ticket. But when more than one person was able to travel on a single ticket, we can calculate fare per head on every ticket.

```
titanic_df['fare_per_head'] = titanic_df['fare'] /
titanic_df['persons_on_ticket']
```

Age Grouping

`Age` is a continuous numeric column and can be transformed into a categorical column by using bins and labels. Here, `age` column is converted into 3 categories namely, `child, adult, senior`. These categories have age boundaries as [0 – 21), [21 – 50), [50 – 100).

```
bins = [0, 21, 50, 100]
labels = ['child', 'adult', 'senior']

titanic_df['age_category'] = pd.cut(titanic_df['age'], bins=bins, labels=labels,
right=False)
```

Fare Binning

The `fare` column can also be converted into a categorical column using the same concept of binning. Here, the `fare` column is transformed into 3 categories called `low, medium, high`. These categories have boundaries as [0 – 100), [100 – 250), [250 – 550) since the max ticket cost 518 Euros in the given data.

```
bins = [0, 100, 250, 550]
labels = ['low', 'medium', 'high']
```

```
titanic_df['fare_category'] = pd.cut(titanic_df['fare'], bins=bins,
labels=labels, right=False)
```

Encoding Categorical Variables

Encoding categorical variables can be useful when using the data to predict a target value in a Machine Learning project. Hence, it's always best to encode our categorical variables if applicable.

1. Encoding variables like `sex` & `embarked` with manually writing a dictionary since the values are very few in numbers.

```
sex_dict = {
    'female': 0,
    'male': 1
}

titanic_df['sex'].replace(sex_dict, inplace=True)

embarked_dict = {
    'S': 0,
    'C': 1,
    'Q': 2
}

titanic_df['embarked'].replace(embarked_dict, inplace=True)
```

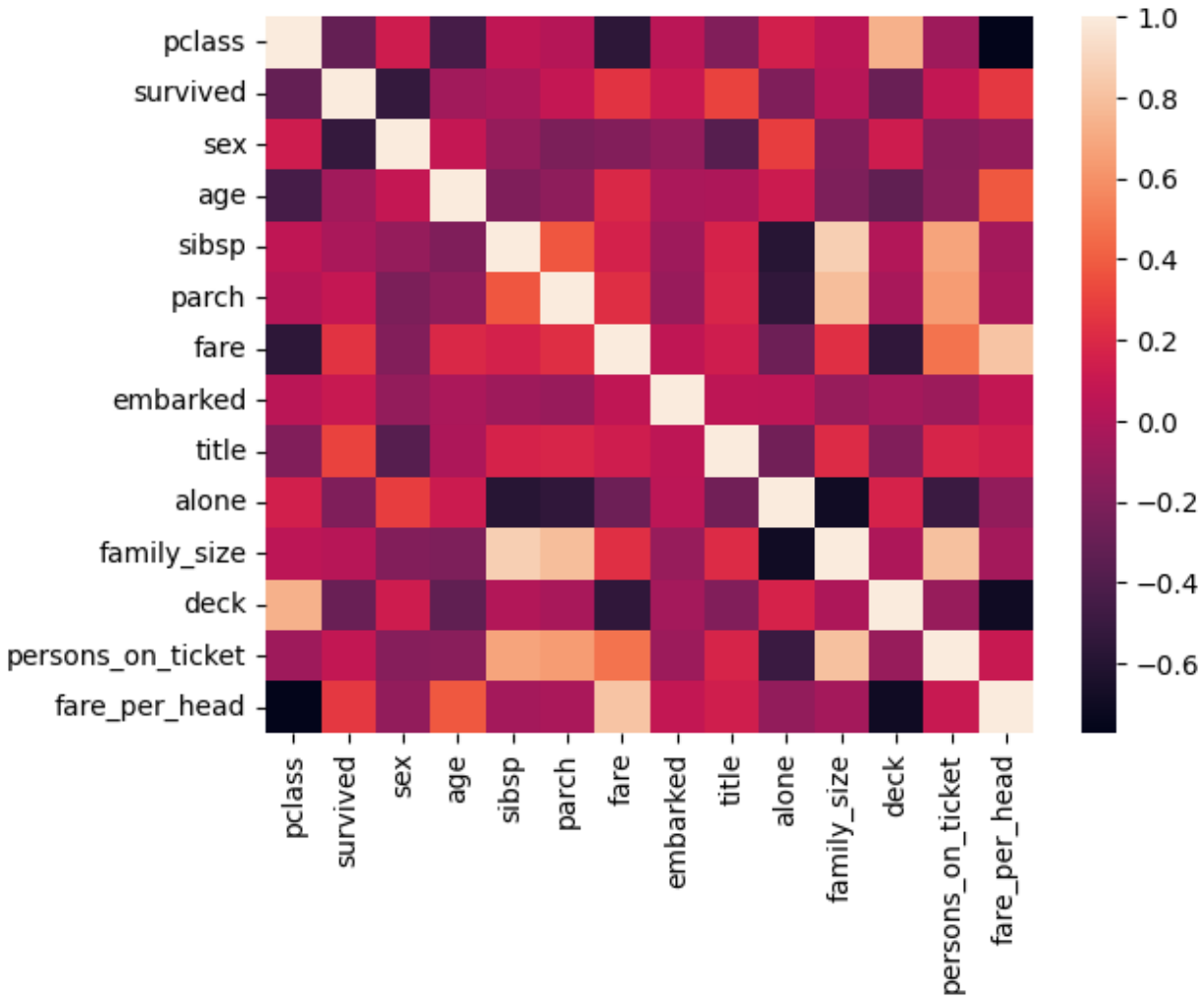
2. Encoding variables like `title` & `deck` using dictionary comprehension because of multiple possible categories in these variables.

```
titles = titanic_df['title'].value_counts().index
title_dict = {title:index for index, title in enumerate(titles)}

titanic_df['title'].replace(title_dict, inplace=True)

titles = titanic_df['title'].value_counts().index
title_dict = {title:index for index, title in enumerate(titles)}

titanic_df['title'].replace(title_dict, inplace=True)
```

Correlation Analysis

Correlation Heatmap can be made using the following snippet:

```
sns.heatmap(titanic_df.corr(numeric_only=True))
```



Following conclusions can be made from the given correlation heatmap:

1. `pclass` shows extremely negative correlation with `fare` & `fare_per_head` columns. That tells us that as we move up in the passenger classes, the fare reduces and Class 1 is the elitest class of passengers in the Titanic.
2. `pclass` also shows moderately negative correlation with the `age` column. This means that older people tended to buy inexpensive tickets for Titanic.
3. `alone` column shows negative correlation with `sibsp`, `parch`, & `family_size` columns, while `family_size` shows positive correlation with `sibsp` & `parch`.
4. As we move higher in the decks, the fare tends to go down. This means that deck A was the most inexpensive one and G was the most expensive deck.
5. There are some other correlations which display some obvious trends like positive correlation between `family_size` & `persons_on_ticket`, `fare` & `fare_per_head` etc.

References

[1] https://www.encyclopedia-titanica.org/titanic/

GitHub Repository

https://github.com/saifsafsf/Cowlar-Recruitment-Task