Group6 - IB9HP0 - Technical Report

Table of contents

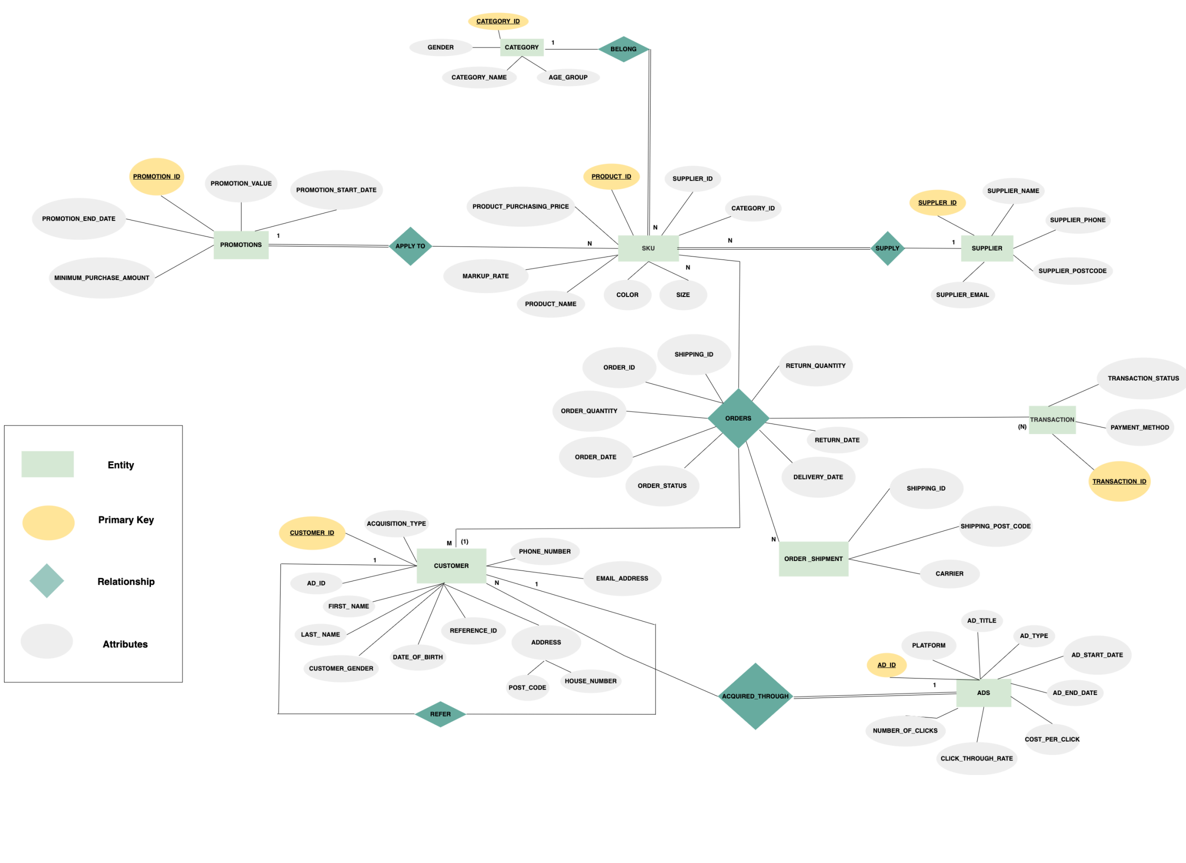
Github Link: https://github.com/pvk99/DM_Group_6

Introduction

This project undertook a comprehensive approach to data management for an apparel e-commerce retailer based in the UK, encompassing database design, data analysis, and reporting. The database architecture was structured around eight entities including products, customers, shipments, promotions, advertisements, suppliers, categories, and transactions, depicted through an Entity-Relationship (ER) diagram. An SQL schema was then implemented to substantiate this design. R was employed to generate synthetic data aligned with our schema to simulate real-world retail transactions. Rigorous data quality assurance was conducted before writing the data into the database. Subsequently, Quarto with R was utilised for data analysis to offer practical insights for strategic decision-making. Automation of data validation, loading, and analysis processes was achieved through a GitHub workflow, ensuring collaborative oversight and accountability throughout the project lifecycle.

Part 1: Database Design and Implementation

1.1: E-R Diagram Design



The E-R diagram contained eight entities and relationships, depicting the operational structure of an apparel e-commerce retailer based in the UK. It illustrated the journey of an order starting from a registered customer selecting a product (SKU), through to the order's delivery, and handling of any subsequent returns. It also covered the process of customer acquisition through three main channels, organic, advertisement and referrals. Please refer to Appendix 1 for the definition of each attribute of each entity.

We made the following assumptions:

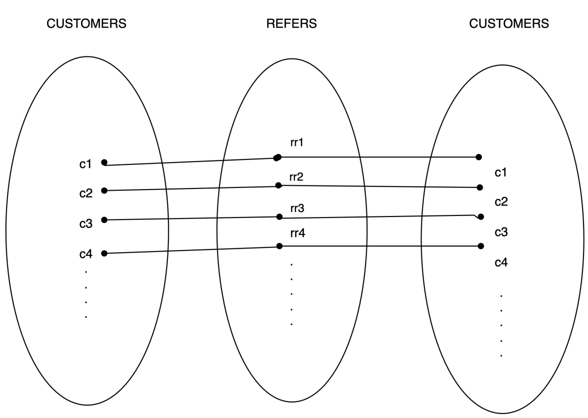
- Product is tracked at the SKU level.
- Promotion codes are applied at SKU level.
- Customers must register before placing orders.
- Third-party analytics (e.g., Apps flyer) is used for marketing and customer attribution, thus we can track paid customers are acquired from which ads.
- One order can be paid by only one transaction.
- One new customer can be referred by only one existing customer.
- Customers can only return the order within 30 days from order placement date.
- Every order will be delivered in one shipment.

Relationships Between Entities:

Our E-R diagram had three types of relationships between different entities, one-to-one, many-to-one, and many-to-many.

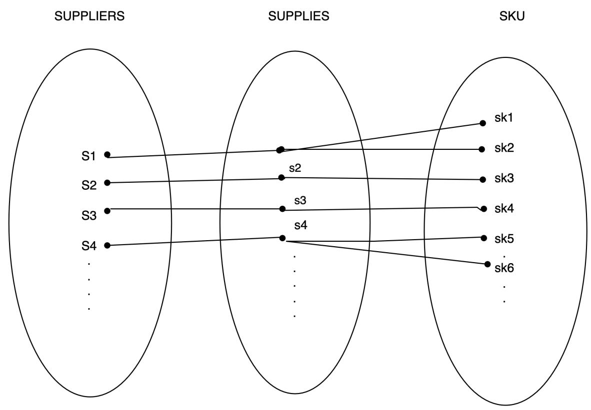
One-to-one:

- CUSTOMERS refers CUSTOMERS: A self-recursive relationship, in which one customer can refer another customer.

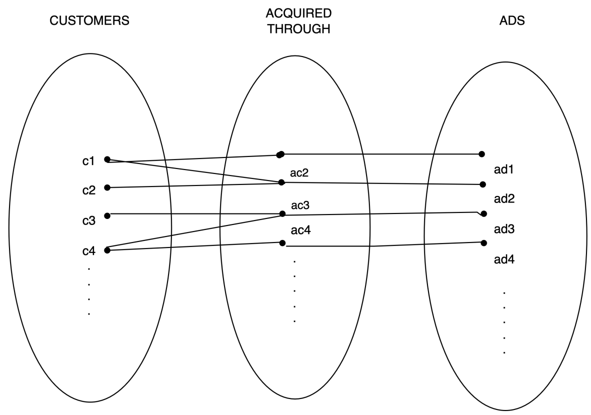


One-to-many:

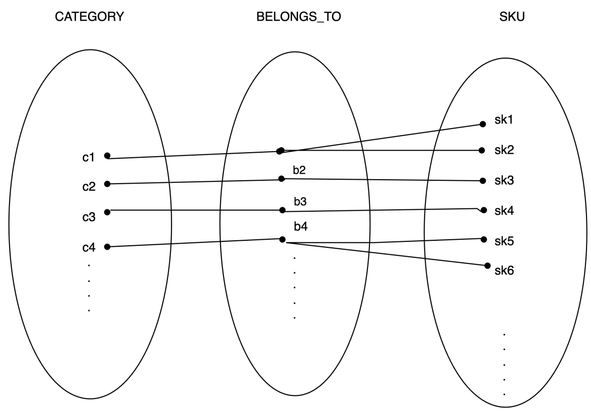
- SUPPLIERS supply SKU: One supplier can supply many products while one SKU is only provided by only one supplier.



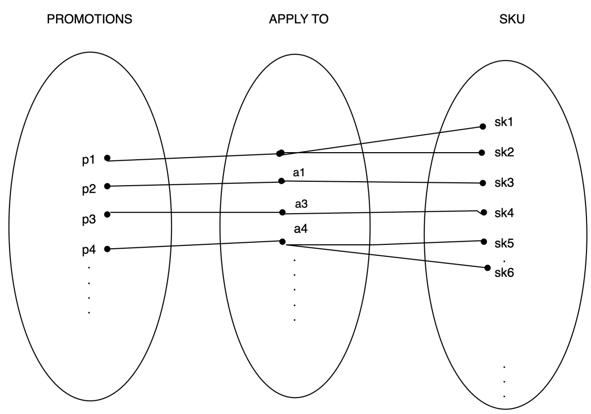
- CUSTOMERS acquired through ADS: One advertisement can be used to acquire many customers whilst one customer can only be acquired through one advertisement.



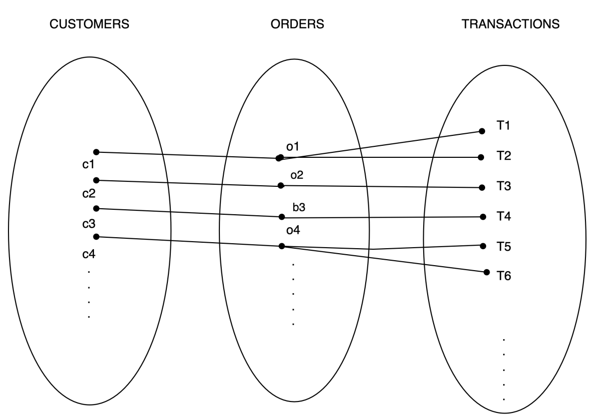
- SKU belongs to CATEGORY: Many products can belong to one category.



- PROMOTIONS apply to SKU: One promotion can be applied to many products.

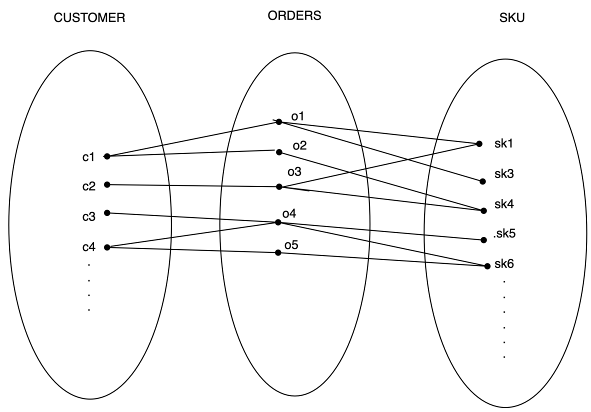


- CUSTOMERS pay TRANSACTIONS: one transaction can only be paid by one customer while one customer can pay multiple transactions.

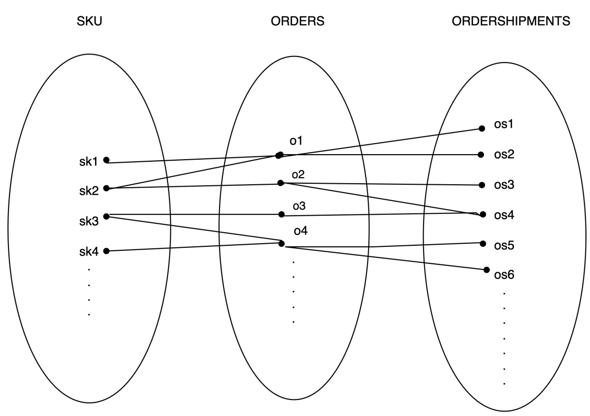


Many-to-many:

- CUSTOMERS order SKU: One customer can order many products and one product can be ordered multiple times.



- SKU delivered through ORDER_SHIPMENT: Many products can have multiple order shipments.



1.2: SQL Database Schema Creation

In creating logical and physical schema, to satisfy 3NF normalisation, we ensured that (1) each attribute was single-value, (2) each table had its unique primary key, (3) all attributes in all tables represented distinctive information, (4) no inferred data was stored (e.g. calculated field), (5) in a table, except for the primary key, other attributes were not dependent on each other. Subsequently, considerations were made for data volume and performance requirements, selecting appropriate data types and indexes to optimise query performance. Based on these designs, the physical schema was created using the SQL, including table structures, indexes, and constraints. Notably, as ORDER is a many-to-many relationship, an ORDERS table was created in the logical and physical schema.

To derive the physical schema of the database, entity-relationship modeling and normalization were conducted initially. This involved identifying entities, attributes, and relationships, ensuring that the data model adhered to third normal form (3NF). Subsequently, considerations were made for data volume and performance requirements, selecting appropriate data types and indexes to optimize qu ery performance. Finally, based on these designs, the physical schema was created using the SQL language, including table structures, indexes, and constraints.

Logical Schema

ORDERS (ORDER_ID, CUSTOMER_ID, TRANSACTION_ID, PRODUCT_ID, SHIPPING_ID, ORDER_DATE, ORDER_STATUS, DELIVERED_DATE, ORDER_QUANTITY, RETURN_QUANTITY, RETURN_DATE)

ORDER_SHIPMENT (SHIPPING_ID, POST_CODE, CARRIER)

TRANSACTION (TRANSACTION_ID, PAYMENT_METHOD, TRANSACTION_STATUS)

PROMOTION (PROMOTION_ID, PRODUCT_ID, PROMOTION_VALUE, PROMOTION_START_DATE, PROMOTION_END_DATE, MINIMUM_PURCHASE_AMOUNT)

ADS (AD_ID, PLATFORM, AD_TITLE, AD_TYPE, AD_START_DATE, AD_END_DATE, COST_PER_CLICK, CLICK_THROUGH_RATE, NUMBER_OF_CLICKS)

CUSTOMERS (CUSTOMER_ID, AD_ID, REFERENCE_ID, CUSTOMER_EMAIL, CUSTOMER_GENDER, PHONE_NUMBER, LAST_NAME, FIRST_NAME, DATA_OF_BIRTH, POST_CODE, HOUSE NUMBER, ACQUISITION_TYPE)

CATEGORY (CATEGORY_ID, CATEGORY_NAME, GENDER, AGE_GROUP)

SUPPLIER (SUPPLER_ID, SUPPLIER_NAME, SUPPLIER_PHONE, POST_CODE, SUPPLIER_EMAIL)

SKU (PRODUCT_ID, SUPPLER_ID, CATEGORY_ID, PRODUCT_NAME, PRODUCT_PURCHASING_PRICE, COLOR, SIZE, MARKUP)

Physical Schema

```
rm(list=ls())
library(readr)
library(RSQLite)
library(dplyr)

#Creating a connection to a database
my_db <- RSQLite::dbConnect(RSQLite::SQLite(),"/cloud/project/ecommerce.db")
```

- ADS

```
CREATE TABLE ADS(
  AD_ID VARCHAR(30) PRIMARY KEY,
  PLATFORM VARCHAR(255),
  AD_TITLE VARCHAR(255),
  AD_TYPE VARCHAR(70),
  AD_START_DATE VARCHAR(15),
  AD_END_DATE VARCHAR(15),
  COST_PER_CLICK FLOAT,
  CLICK_THROUGH_RATE FLOAT,
  NUMBER_OF_CLICK FLOAT
);
```

- CATEGORY

```
CREATE TABLE CATEGORY(
  CATEGORY_NAME TEXT NOT NULL,
  GENDER TEXT,
  AGE_GROUP VARCHAR(30),
  CATEGORY_ID VARCHAR(30) PRIMARY KEY
);
```

- SUPPLIER

```
CREATE TABLE SUPPLIER (
    SUPPLIER_NAME VARCHAR(50),
    SUPPLIER_EMAIL VARCHAR(30),
    SUPPLIER_PHONE NUMERIC(10),
    POST_CODE VARCHAR(30),
    SUPPLIER_ID VARCHAR(30) PRIMARY KEY
);
```

- CUSTOMERS

```
CREATE TABLE IF NOT EXISTS CUSTOMERS(
  CUSTOMER_ID VARCHAR(30) PRIMARY KEY,
  ACQUISITION_TYPE TEXT,
  REFERENCE_ID VARCHAR(30),
  PHONE_NUMBER NUMERIC(10),
  CUSTOMER_GENDER TEXT,
  DATE_OF_BIRTH VARCHAR(15),
  FIRST_NAME TEXT,
  LAST_NAME TEXT,
  CUSTOMER_EMAIL VARCHAR(30),
  POST_CODE VARCHAR(30),
  HOUSE_NUMBER INT,
  AD_ID VARCHAR(30),
  FOREIGN KEY (AD_ID) REFERENCES ADS (AD_ID)
  FOREIGN KEY (REFERENCE_ID) REFERENCES CUSTOMER(CUSTOMER_ID)
);
```

- PRODUCT

```
CREATE TABLE IF NOT EXISTS SKU(
    COLOR TEXT,
    SIZE TEXT,
    PRODUCT_NAME TEXT,
    PRODUCT_ID VARCHAR(30) PRIMARY KEY,
    PRODUCT_PURCHASING_PRICE FLOAT,
    MARKUP FLOAT,
    SUPPLIER_ID VARCHAR(30),
    CATEGORY_ID VARCHAR(30),
    FOREIGN KEY (SUPPLIER_ID) REFERENCES SUPPLIER(SUPPLIER_ID),
    FOREIGN KEY (CATEGORY_ID) REFERENCES CATEGORY(CATEGORY_ID)
);
```

- PROMOTION

```
CREATE TABLE PROMOTION (
  PROMOTION_ID VARCHAR(30) PRIMARY KEY,
  PROMOTION_VALUE FLOAT,
  PROMOTION_START_DATE VARCHAR(15),
  PROMOTION_END_DATE VARCHAR(15),
  MINIMUM_PURCHASE_AMOUNT FLOAT,
```

```
  PRODUCT_ID VARCHAR(30),
  FOREIGN KEY (PRODUCT_ID) REFERENCES SKU(PRODUCT_ID)
);
```

- TRANSACTION

```
CREATE TABLE TRANSACTIONS (
  TRANSACTION_ID VARCHAR(30) PRIMARY KEY,
  PAYMENT_METHOD VARCHAR(50),
  TRANSACTION_STATUS VARCHAR(50)
);
```

- ORDER_SHIPMENT

```
CREATE TABLE ORDER_SHIPMENT(
  SHIPPING_ID VARCHAR(30) PRIMARY KEY,
  POST_CODE VARCHAR(30),
  CARRIER TEXT
);
```

- ORDERS

```
CREATE TABLE ORDERS (
  ORDER_ID VARCHAR(30),
  CUSTOMER_ID VARCHAR(30),
  ORDER_DATE VARCHAR(15),
  ORDER_STATUS TEXT,
  SHIPPING_ID VARCHAR(30),
  DELIVERY_DATE VARCHAR(15),
  TRANSACTION_ID VARCHAR(30),
  PRODUCT_ID VARCHAR(30),
  ORDER_QUANTITY INTEGER,
  RETURN_QUANTITY INTEGER,
  RETURN_DATE VARCHAR(15),
  PRIMARY KEY (ORDER_ID, PRODUCT_ID, CUSTOMER_ID),
  FOREIGN KEY (PRODUCT_ID) REFERENCES SKU(PRODUCT_ID),
  FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMERS(CUSTOMER_ID),
  FOREIGN KEY (TRANSACTION_ID) REFERENCES TRANSACTIONS(TRANSACTION_ID)
  FOREIGN KEY (SHIPPING_ID) REFERENCES SHIPMENT(SHIPPING_ID)
);
```

Part 2: Data Generation and Management

2.1: Synthetic Data Generation

All synthetic data generation was conducted solely in R and saved into csv files before pushed to GitHub, ensuring adherence to attribute conditionalities and inter-entity connections with the support of LLM. Initially, Mockaroo was explored for data generation, but its high level of randomisation proved more complex compared to R for setting precise rules to control data values (see Figure 1).

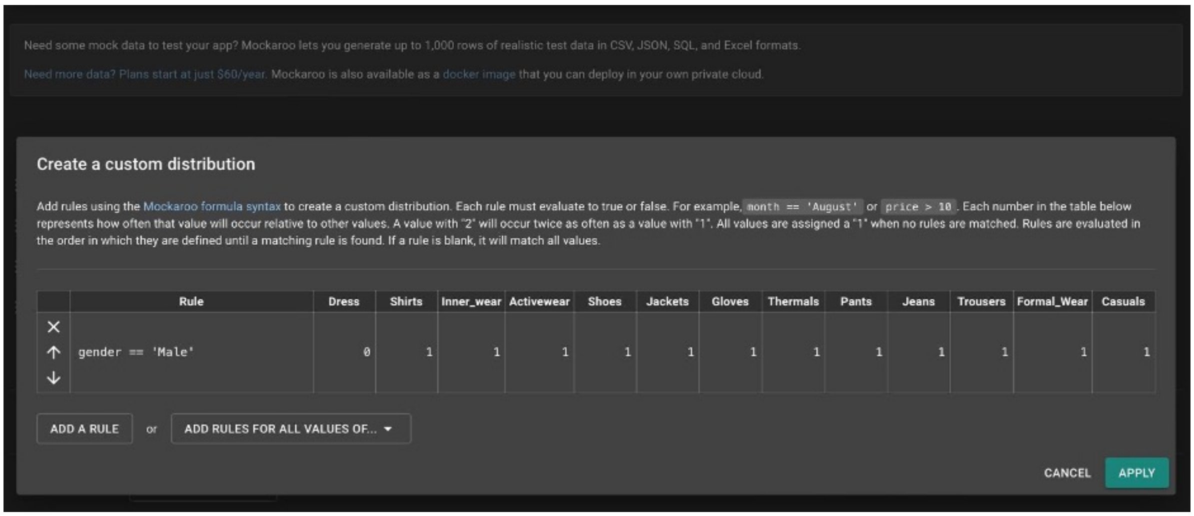


Figure 1 - Customising data values in Mockaroo

In R, independent entities such as CATEGORY, SUPPLIER, PROMOTION, ADS, SKU and CUSTOMERS were created first. We asked LLM to generate values for CATEGORY_NAME, PRODUCT_NAME, SUPPLIER_NAME, AD_TITLE, tailored to the fashion retail industry (Figure 2 and 3). The generation of SUPPLIER_EMAIL and CUSTOMER_EMAIL utlised a function suggested by LLM (Figure 4). Postal codes were randomly generated from UK postcodes using "PostcodesioR" package drawing on data from Office for National Statistic (Walczak, E., 2021). Customer names were randomly created employing "randomNames" package (Betebenner, D.W., 2021). All numerical fields such PRODUCT_PURCHASING_PRICE, MARKUP, etc were randomised using either "runif" or "sample" functions to ensure realistic value distributions. Furthermore, for CUSTOMERS, REFERENCE_ID, denoting CUSTOMER_ID of the referees, was conditioned such that the referred customers were only included in our database after their referees. REFERENCE_ID was exclusively limited to customers with an ACQUISITION_TYPE of "Referral" while AD_ID was randomly assigned from ADS table for customers with an ACQUISITION_TYPE of "Paid".

Please refer to Appendix A2 for the full prompt sequences in LLM.

```
# Create name
## Filter out names with apostrophes
filtered_names <- randomNames(nrow(CUSTOMER))
filtered_names <- filtered_names[!grepl("'", filtered_names)]

## If the number of filtered names is less than
## the number of rows in CUSTOMER, generate additional names
if (length(filtered_names) < nrow(CUSTOMER)) {
  additional_names <- randomNames(nrow(CUSTOMER) - length(filtered_names))
  additional_names <- additional_names[!grepl("'", additional_names)]
  filtered_names <- c(filtered_names, additional_names)
}

## Generate random full names
random_full_names <- sample(filtered_names, nrow(CUSTOMER), replace = TRUE)

#W Split full names into first and last names
split_names <- strsplit(random_full_names, ",")

## Extract first names
CUSTOMER$FIRST_NAME <- sapply(split_names, "[", 2)

## Extract last names
CUSTOMER$LAST_NAME <- sapply(split_names, "[", 1)

# CREATE CUSTOMER_POST_CODE
CUSTOMER$POST_CODE <-
sapply(1:nrow(CUSTOMER), function(x) random_postcode()$postcode)
```

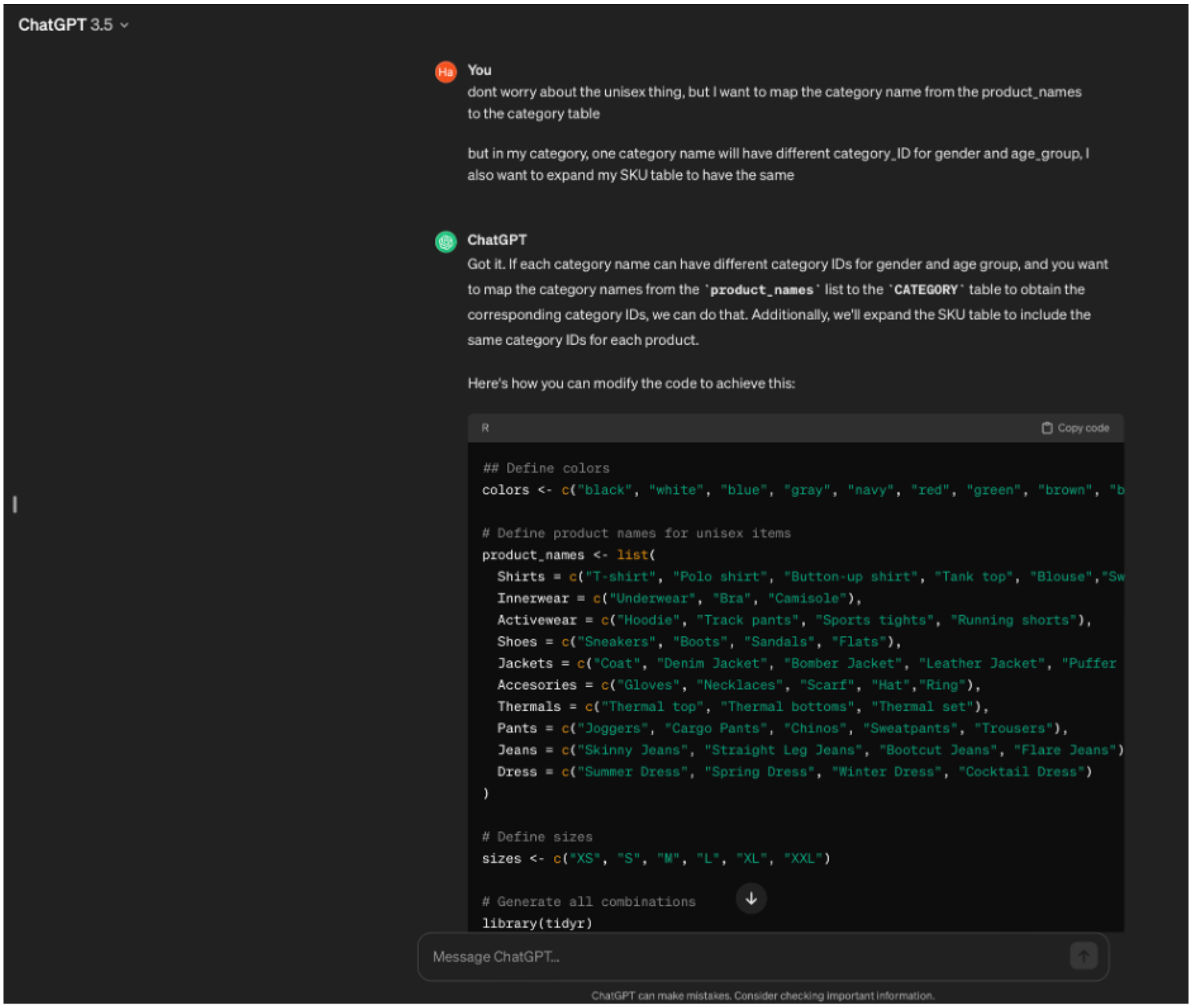


Figure 2 - LLM prompts for product name by category

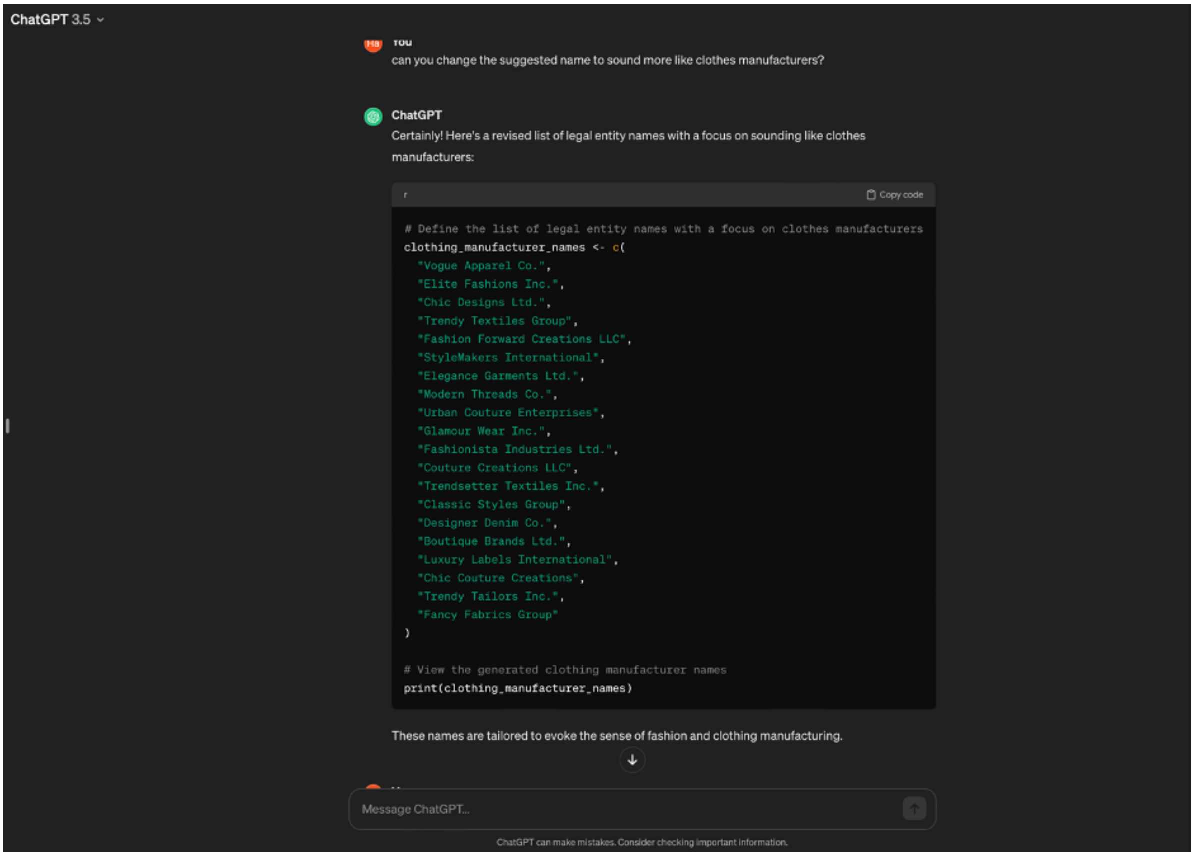


Figure 3 - LLM prompts for supplier names.

Figure 4 - LLM prompt for email generation

Subsequently, dependent tables including ORDERS, ORDER_SHIPMENT and TRANSACTIONS were created based on the data from the previous tables. Specifically, in ORDERS, combinations of CUSTOMER_ID and PRODUCT_ID were randomly selected from SKU and CUSTOMERS tables so that a customer can purchase multiple products in an order. The generation of DELIVERY_DATE and RETURN_DATE adhered to constraints derived from ORDER_DATE, ensuring that DELIVERY_DATE fell within a maximum of 7 days after order placement, and RETURN_DATE occurred after DELIVERY_DATE while remaining within 30 days from ORDER_DATE. ORDER_STATUS was randomised to reflect an authentic distribution encompassing “Returned”, “Delivered” and “In transit”. RETURN_QUANTITY and RETURN_DATE was exclusively assigned to orders marked as “Returned”, with RETURN_QUANTITY restricted not to exceed ORDER_QUANTITY. Moreover, for TRANSACTIONS table, we also ensured that all orders have matching successful transactions, with the proportion of failed transaction set to 6% of total transaction numbers.

```
# Create ORDER dataframe
ORDER <- data.frame(ORDER_ID = paste0("OD", seq_len(1000) + 10000),
  stringsAsFactors = TRUE
)

# Add CUSTOMER_ID
ORDER$CUSTOMER_ID <- sample(CUSTOMER$CUSTOMER_ID,nrow(ORDER),replace=TRUE)


# Add ORDER_DATE
ORDER$ORDER_DATE <-
sample(seq(start_date, end_date, by = "day"), nrow(ORDER), replace = TRUE)

# Add ORDER_STATUS:
order_status <- c("Delivered", "Returned")
proportions <- c("Delivered" = 0.8, "Returned" = 0.2)

# Sample proportionally with replacement and assign to ORDER$ORDER_STATUS
ORDER$ORDER_STATUS <-
sample(order_status, size = nrow(ORDER), replace = TRUE, prob = proportions)
ORDER$ORDER_STATUS[ORDER$ORDER_DATE>'2024-03-05'] = "In transit"

# Add SHIPPING_ID
ORDER$SHIPPING_ID <- ORDER_SHIPMENT$SHIPPING_ID

# Add DELIVERY_DATE

ORDER$DELIVERY_DATE <- ORDER$ORDER_DATE + sample(7,nrow(ORDER),replace=TRUE)
ORDER$DELIVERY_DATE[ORDER$ORDER_STATUS %in% c("Order placed", "In transit")] <- NULL

# Add TRANSACTION_ID
ORDER$TRANSACTION_ID <-
TRANSACTION$TRANSACTION_ID[TRANSACTION$TRANSACTION_STATUS == "Successful"]

# Create an empty dataframe to store order-product mappings
order_product_mapping <-
data.frame(ORDER_ID = character(), PRODUCT_ID = character(), stringsAsFactors = FALSE)
```

```
# Define the number of products per order
products_per_order <- round(runif(nrow(ORDER), min = 1, max = 10))

# Loop through each order ID and sample products
for (i in 1:nrow(ORDER)) {
  order_id <- ORDER$ORDER_ID[i]
  product_ids <-
  sample(SKU$PRODUCT_ID, size = products_per_order[i], replace = FALSE)
  order_product_mapping <- rbind(order_product_mapping,
  data.frame(ORDER_ID = rep(order_id, length(product_ids)), PRODUCT_ID = product_ids))
}

# Merge order_product_mapping with ORDER dataframe to retain other order details
ORDER <- merge(ORDER, order_product_mapping, by = "ORDER_ID", all.x = TRUE)

# Add ORDER_QUANTITY
ORDER$ORDER_QUANTITY <-round(runif(nrow(ORDER),min=1,max=5),0)

# Add RETURN_QUANTITY
ORDER$RETURN_QUANTITY <-
ifelse(ORDER$ORDER_STATUS=="Returned",
mapply(function(x) sample(1:x, 1), ORDER$ORDER_QUANTITY),"")

# Add RETURN_DATE
ORDER$RETURN_DATE <- ORDER$ORDER_DATE + sample(9:30,nrow(ORDER),replace=TRUE)
ORDER$RETURN_DATE[ORDER$ORDER_STATUS != "Returned"] <- ""

# Change DATE to CHARACTER
ORDER$ORDER_DATE <- as.character(ORDER$ORDER_DATE)
ORDER$RETURN_DATE <- as.character(ORDER$RETURN_DATE)
ORDER$DELIVERY_DATE <- as.character(ORDER$DELIVERY_DATE)

#Save to csv
write.csv(ORDER[1:15,],
file = file.path("/cloud/project/Data","ORDERS.1.csv"), row.names = FALSE)
write.csv(ORDER[16:nrow(ORDER),],
file = file.path("/cloud/project/Data","ORDERS.2.csv"), row.names = FALSE)
```

2.2: Data Import and Quality Assurance

An R script was developed to load all data files in their respective corresponding tables within our database, ensuring robust data quality and integrity. This process involved connecting to the database and iteratively validating data entries against predefined criteria.

The script employed a loop cross-checking primary key values to prevent duplications and enforcing strict validation rules on a row by row basis. Entries were scrutinised for null values, adherence to formatting standards for phone numbers and emails, and non-negativity in numeric fields. For formatting check, we defined functions to automatically validate. Data entries satisfying these validation criteria were then appended to the database whilst problematic entries were recorded in "error_log.txt" file for further reviews.

```
library(readr)
library(RSQLite)
library(dplyr)
library(DBI)

# Define a function to check if phone numbers are of length 10
# and contain only numeric characters
validate_phone_numbers <- function(phone) {
  phone <- as.character(phone)  # Convert integer to character
  if (!is.na(phone) && nchar(phone) == 10 && !any(is.na(as.numeric(phone)))) {
    return(TRUE)  # Phone number is valid
  } else {
    return(FALSE) # Phone number is not valid
  }
}



# Function to validate email addresses
email_pattern <- "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"
validate_emails <- function(email) {
  grepl(email_pattern, email)
}

# Function to validate that numeric attributes are non-negative
check_non_negative_numeric <- function(data) {
  numeric_cols <- sapply(data, is.numeric)
  negative_values <- sapply(data[numeric_cols], function(col) any(col < 0))
  return(!any(negative_values))
}
```

```
# Establishing the connection to db
my_db <- RSQLite::dbConnect(RSQLite::SQLite(), "ecommerce.db")

file_paths <- list(
  "ADS" = list.files(path = "Data", pattern = "ADS.*\\.csv$", full.names = TRUE),
  "CATEGORY" = list.files(path = "Data",
  pattern = "CATEGORY.*\\.csv$", full.names = TRUE),
  "SUPPLIER" = list.files(path = "Data",
  pattern = "SUPPLIER.*\\.csv$", full.names = TRUE),
  "CUSTOMERS" = list.files(path = "Data",
  pattern = "CUSTOMERS.*\\.csv$", full.names = TRUE),
  "SKU" = list.files(path = "Data",
  pattern = "SKU.*\\.csv$", full.names = TRUE),
  "PROMOTION" = list.files(path = "Data",
  pattern = "PROMOTION.*\\.csv$", full.names = TRUE),
  "TRANSACTIONS" = list.files(path = "Data",
  pattern = "TRANSACTIONS.*\\.csv$", full.names = TRUE),
  "ORDER_SHIPMENT" = list.files(path = "Data",
  pattern = "ORDER_SHIPMENT.*\\.csv$", full.names = TRUE),
  "ORDERS" = list.files(path = "Data",
  pattern = "ORDERS.*\\.csv$", full.names = TRUE)
)

tables <- list(
  "ADS" = "AD_ID",
  "CATEGORY" = "CATEGORY_ID",
  "SUPPLIER" = "SUPPLIER_ID",
  "CUSTOMERS" = "CUSTOMER_ID",
  "SKU" = "PRODUCT_ID",
  "PROMOTION" = "PROMOTION_ID",
  "TRANSACTIONS" = "TRANSACTION_ID",
  "ORDER_SHIPMENT" = "SHIPPING_ID",
  "ORDERS" = c("ORDER_ID", "PRODUCT_ID", "CUSTOMER_ID")
)

# Define write_errors function with folder path argument
write_errors <- function(errors, folder_path, file_name) {
  # Ensure the folder exists, if not, create it
  if (!dir.exists(folder_path)) {
    dir.create(folder_path, recursive = TRUE)
  }
```

```
  file_path <- file.path(folder_path, file_name)

  if (length(errors) > 0) {
    cat("Errors:\n", file = file_path)
    for (error in errors) {
      cat(error, "\n", file = file_path, append = TRUE)
    }
    cat("\n", file = file_path, append = TRUE)
  }
}

# List to store errors
error_list <- c()

# Function to check if data entries exist and load new entries
for (table_name in names(tables)) {
  for (file_path in file_paths[[table_name]]) {
    table_data <- read_csv(file_path,n_max = Inf)

    ## Apply specific rules for attributes based on the table
    if (table_name == "ADS") {
      # Initialize error list
      error_list <- vector("list")

      # Convert AD_START_DATE and AD_END_DATE to character
      table_data$AD_START_DATE <- as.character(table_data$AD_START_DATE)
      table_data$AD_END_DATE <- as.character(table_data$AD_END_DATE)

      # Initialize vector to store indices of invalid rows
      invalid_rows <- vector("numeric")

      # Ensure numeric attributes are non-negative
      numeric_attrs <- c("COST_PER_CLICK",
      "CLICK_THROUGH_RATE", "NUMBER_OF_CLICK")
      for (attr in numeric_attrs) {
        if (any(table_data[[attr]] < 0)) {
          error_list <- c(error_list,
          paste("Negative values found in", attr,
          "column of ADS table."))
          invalid_rows <- c(invalid_rows, which(table_data[[attr]] < 0))
        }
      }
```

```
    # Remove invalid rows from table_data
    if (length(invalid_rows) > 0) {
      table_data <- table_data[-invalid_rows, ]
    }
  }
  if (table_name == "CUSTOMERS") {
    # Initialize error list
    error_list <- vector("list")

    # Convert DATE_OF_BIRTH to character
    table_data$DATE_OF_BIRTH <- as.character(table_data$DATE_OF_BIRTH)

    # Initialize vector to store indices of invalid rows
    invalid_rows <- vector("numeric")

    # Check phone numbers and emails
    for (i in 1:nrow(table_data)) {
      if (!validate_phone_numbers(table_data$PHONE_NUMBER[i])) {
        error_list <-
        c(error_list,
        paste("Invalid phone number in CUSTOMERS table:", table_data$PHONE_NUMBER[i]))
        invalid_rows <- c(invalid_rows, i)
      }

      if (!validate_emails(table_data$CUSTOMER_EMAIL[i])) {
        error_list <- c(error_list,
        paste("Invalid email in CUSTOMERS table:", table_data$CUSTOMER_EMAIL[i]))
        invalid_rows <- c(invalid_rows, i)
      }
    }

    # Remove invalid rows from table_data
    if (length(invalid_rows) > 0) {
      table_data <- table_data[-invalid_rows, ]
    }
  }
  if (table_name == "SKU") {
    # Initialize error list
    error_list <- vector("list")

    # Ensure numeric attributes are non-negative
    numeric_attrs <- c("MARKUP", "PRODUCT_PURCHASING_PRICE")
```

```
    for (attr in numeric_attrs) {
      if (any(table_data[[attr]] < 0)) {
        error_list <- c(error_list,
        paste("Negative values found in", attr, "column of SKU table."))
        invalid_rows <- c(invalid_rows, which(table_data[[attr]] < 0))
      }
    }

    # Remove invalid rows from table_data
    if (length(invalid_rows) > 0) {
      table_data <- table_data[-invalid_rows, ]
    }
  }
  if (table_name == "PROMOTION") {
    # Initialize error list
    error_list <- vector("list")

    # Convert PROMOTION_START_DATE and PROMOTION_END_DATE to character
    table_data$PROMOTION_START_DATE <- as.character(table_data$PROMOTION_START_DATE)
    table_data$PROMOTION_END_DATE <- as.character(table_data$PROMOTION_END_DATE)

    # Initialize vector to store indices of invalid rows
    invalid_rows <- vector("numeric")

    # Ensure numeric attributes are non-negative
    numeric_attrs <- c("MINIMUM_PURCHASE_AMOUNT")
    for (attr in numeric_attrs) {
      if (any(table_data[[attr]] < 0)) {
        error_list <- c(error_list,
        paste("Negative values found in", attr, "column of PROMOTION table."))
        invalid_rows <- c(invalid_rows, which(table_data[[attr]] < 0))
      }
    }

    # Remove invalid rows from table_data
    if (length(invalid_rows) > 0) {
      table_data <- table_data[-invalid_rows, ]
    }
  }
  if (table_name == "SUPPLIER") {
    # Initialize error list
    error_list <- vector("list")
```

```
    # Initialize vector to store indices of invalid rows
    invalid_rows <- vector("numeric")

    # Check phone numbers and emails
    for (i in 1:nrow(table_data)) {
      if (!validate_phone_numbers(table_data$SUPPLIER_PHONE[i])) {
        error_list <- c(error_list,
        paste("Invalid phone number in SUPPLIER table:", table_data$SUPPLIER_PHONE[i]))
        invalid_rows <- c(invalid_rows, i)
      }

      if (!validate_emails(table_data$SUPPLIER_EMAIL[i])) {
        error_list <- c(error_list,
        paste("Invalid email in SUPPLIER table:",
        table_data$SUPPLIER_EMAIL[i]))
        invalid_rows <- c(invalid_rows, i)
      }
    }

    # Remove invalid rows from table_data
    if (length(invalid_rows) > 0) {
      table_data <- table_data[-invalid_rows, ]
    }
  }
  if (table_name == "ORDERS") {
    # Initialize error list
    error_list <- vector("list")

    # Convert ORDER_DATE, DELIVERY_DATE, RETURN_DATE to character
    table_data$ORDER_DATE <- as.character(table_data$ORDER_DATE)
    table_data$DELIVERY_DATE <- as.character(table_data$DELIVERY_DATE)
    table_data$RETURN_DATE <- as.character(table_data$RETURN_DATE)

    # Initialize vector to store indices of invalid rows
    invalid_rows <- vector("numeric")

    # Check numeric attributes for non-negativity
    numeric_attrs <- c("ORDER_QUANTITY", "RETURN_QUANTITY")
    for (attr in numeric_attrs) {
      if (any(!is.na(table_data[[attr]]) & table_data[[attr]] < 0)) {
        error_list <- c(error_list,
        paste("Invalid or negative values found in", attr, "column of ORDERS table."))
```

```
      invalid_rows <- c(invalid_rows,
      which(!is.na(table_data[[attr]]) & table_data[[attr]] < 0))
    }
  }

  # Remove invalid rows from table_data
  if (length(invalid_rows) > 0) {
    table_data <- table_data[-invalid_rows, ]
  }
}

## Check for primary key duplication
for (i in seq(nrow(table_data))) {
  new_record <- table_data[i, ]
  pk_columns <- tables[[table_name]]
  pk_values <- new_record[pk_columns]
  # Check if primary key values are non-null
  if (any(is.na(pk_values))) {
    error_list <- c(error_list,
    paste("Null primary key value found in", table_name, "table."))
    next  # Skip to the next record if primary key is null
  }

  conditions <- paste(pk_columns, "=", paste0("'", pk_values, "'"),
  collapse = " AND ")

  key_exists <- dbGetQuery(my_db,
  paste("SELECT COUNT(*) FROM", table_name, "WHERE", conditions))

  if (key_exists == 0) {
    tryCatch({
      RSQLite::dbAppendTable(my_db, table_name, new_record)
    }, error = function(e) {
      error_list <- c(error_list,
      paste("Error inserting record with primary key",
      paste(pk_values, collapse = ", "), "into table", table_name))
      print(paste("Error inserting record with primary key",
      paste(pk_values, collapse = ", "), "into table", table_name))
      print(e)
    })
  } else {
    print(paste("Record with primary key",
```

```
        paste(pk_values, collapse = ", "),
        "already exists in table", table_name))
      }
    }
  }
}

# Save errors to a folder named "Error logs" within the current directory
write_errors(error_list, "Error logs", "error_log.txt")
```

Part 3: Data Pipeline Generation

3.1: GitHub Repository and Workflow Setup

We initiated our project by creating a new Git repository, connecting Posit Cloud/ RStudio to the repository and uploading essential files, including (1) database, (2) data schema, (3) synthetic data generation, (4) data validation and database writing, (5) data query and analysis scripts. This setup allows us to efficiently track changes and revert to previous versions as needed.

3.2: GitHub Actions for Continuous Integration

To automate our project's operations, we implemented a GitHub Actions workflow, detailed through the following key components:

- Trigger: Activated by either a push event to the main branch or a scheduled run every 24 hours, ensuring real-time integration of contributions.
- Runner: Utilise the most recent Ubuntu environment to execute the job.
- Steps: Comprising eight sequential tasks, each step executes a specific operation within the job:
- Repository Checkout: Clone the project's code into the runner, providing a foundation for subsequent tasks.
- R Environment Setup: Prepare the R environment, ensuring all R-based operations can be performed without hitches.
- R Package Caching: Preserve installed R packages between runs, significantly reducing setup time by bypassing redundant installations.
- Package Installation: Engage only if the cache does not contain the necessary packages, ensuring all dependencies are available for the script execution.
- Script Execution: Run our R scripts from the repository to validate and load satisfying data entries to the database, subsequently creating analyses.
- Add Changes: Scans the project's database for any changes following the script's execution and notifies with a "Changes found" message if updates are identified. Meanwhile, new analyses are automatically generated and saved to folder "figures".
- Commit Changes: If changes are detected, this step prepares and commits the updated files, maintaining a current state within the repository.
- Push Updates: Conclude the workflow by uploading the latest commit to the repository, ensuring all changes are synchronised and stored.

```
name: ETL workflow

on:
  push:
    branches: [ main ]
  schedule:
    - cron: '0 */24 * * *' # Runs every 24 hours

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Setup R environment
        uses: r-lib/actions/setup-r@v2
        with:
          r-version: '4.2.0'

      - name: Cache R packages
        uses: actions/cache@v2
        with:
          path: ${{ env.R_LIBS_USER }}
          key: ${{ runner.os }}-r-${{ hashFiles('**/lockfile') }}
          restore-keys: |
            ${{ runner.os }}-r-

      - name: Create figures directory
        run: mkdir -p figures

      - name: Install packages
        if: steps.cache.outputs.cache-hit != 'true'
        run: |
          Rscript -e 'install.packages(c("ggplot2", "dplyr", "readr", "RSQLite", "DBI"))'

      - name: Execute R script
        run: |
          Rscript R_codes/Workflow.R

      - name: Execute Data Analysis
        run: |
```

```
        Rscript R_codes/Query_workflow.R
    - name: Add database changes and commit
      run: |
        git config --global user.email "Teng-Yi.Chen@warwick.ac.uk"
        git config --global user.name "DylanCTY"
        git add ecommerce.db
        git add --all figures/
        git commit -m "Update database" || echo "No changes to commit"
        # Check if error logs exist
        if [ -d "Error logs" ] && [ "$(ls -A Error\ logs)" ]; then
          echo "Error logs exist"
          # Create error logs folder
          mkdir -p "Error logs"
          # Write error_log.txt
          echo "Errors:" > "Error logs/error_log.txt"
          # Append each error to the error_log.txt file
          for error in Error\ logs/*; do
            cat "$error" >> "Error logs/error_log.txt"
            echo "" >> "Error logs/error_log.txt"  # Add a newline after each error
          done
          # Commit error logs
          git add "Error logs/error_log.txt"
          git commit -m "Add error log" || echo "No error log changes to commit"
        else
          echo "No error logs found"
        fi

    - name: Push changes
      uses: ad-m/github-push-action@v0.6.0
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        branch: main
```

Part 4: Advanced Data Analysis

4.1: Advanced Data Analysis in R

Once the database was updated with the newly generated data via GitHub automation, the data was analysed utilising the R package packages dplyr, GGplot2, and tidyr in conjunction with the SQL DQL command. The procedure entailed retrieving the data and converting it into a format suitable for subsequent analysis. The complete procedure is outlined below.

Data Query

The data from the database was obtained using a SQL Data Query Language (DQL) statement. By utilising the calculated function and aggregate command, the data were converted into a format and value suitable for analysis. The following example is a query that displays the top revenue earned by product SKUs in the last 1 year, calculated by multiplying unit sales with the selling price.

```
Revenue_analysis_df <- RSQLite::dbGetQuery(my_db,
"SELECT T1.PRODUCT_ID AS PRODUCT_ID,
       T2.PRODUCT_NAME AS PRODUCT_NAME,
       T2.SIZE AS SIZE,
       T2.COLOR AS COLOR,
       T1.UNIT_SOLD AS UNIT_SOLD,
       T2.SELLING_PRICE_PER_UNIT AS SELLING_PRICE,
       T1.UNIT_SOLD * T2.SELLING_PRICE_PER_UNIT AS TOTAL_REVENUE
FROM
( SELECT ORDERS.PRODUCT_ID AS PRODUCT_ID,
         ORDERS.ORDER_DATE AS ORDER_DATE,
       SUM(ORDERS.ORDER_QUANTITY) AS UNIT_SOLD
FROM ORDERS
WHERE cast(julianday('now') - julianday(ORDERS.ORDER_DATE) AS INTEGER ) <= 365
GROUP BY ORDERS.PRODUCT_ID

) AS T1

LEFT JOIN

( SELECT SKU.PRODUCT_ID AS PRODUCT_ID,
         SKU.PRODUCT_NAME AS PRODUCT_NAME,
         SKU.SIZE AS SIZE,
         SKU.COLOR AS COLOR,
      SKU.PRODUCT_PURCHASING_PRICE * (1 + SKU.MARKUP) AS
      SELLING_PRICE_PER_UNIT
```

```
FROM SKU
) AS T2

ON T1.PRODUCT_ID = T2.PRODUCT_ID"
)
```

Data Manipulation

Once the data was imported into a R data frame, data manipulation was carried out using R to prepare the data for visualisation. The following code example generates a new column containing the Product SKU description.

```
# Create product description name of each product_ID for using in analysis
Revenue_analysis_df$PRODUCT_DESCRIPTION <- paste(Revenue_analysis_df$PRODUCT_ID,
Revenue_analysis_df$PRODUCT_NAME, Revenue_analysis_df$SIZE,
Revenue_analysis_df$COLOR, sep = " ")
```

Data Visualisation

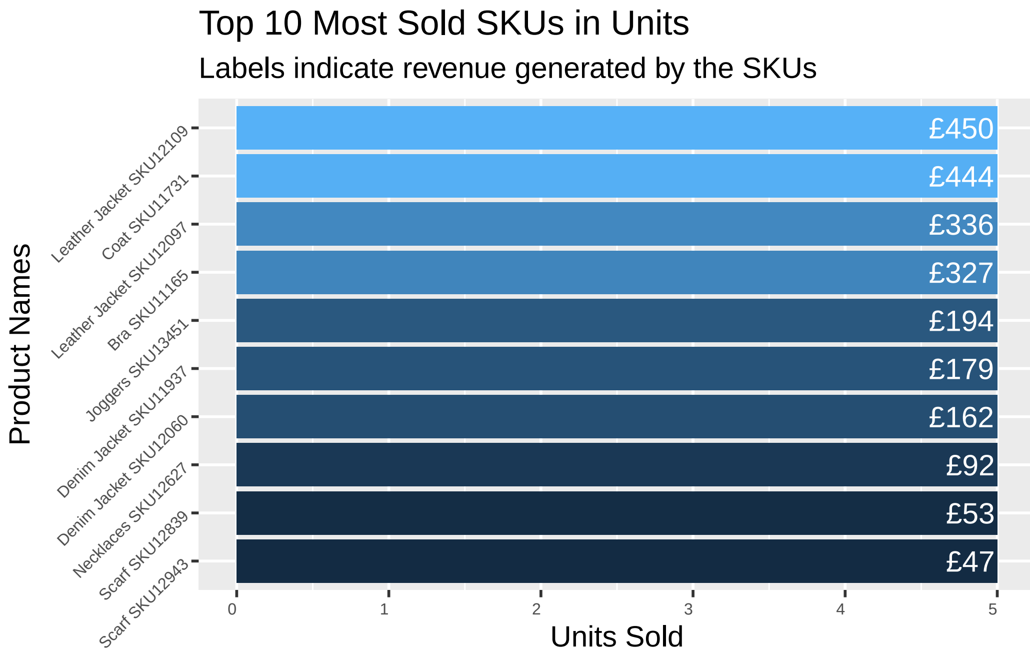
Finally, once we retrieved and formatted the data for analysis, we used the ggplot2 package to create visual representations of the data. The code snippet shown below demonstrates a bar chart that visually represents the top SKUs with the highest sales in the past year, as obtained from the preceding phases of our query.

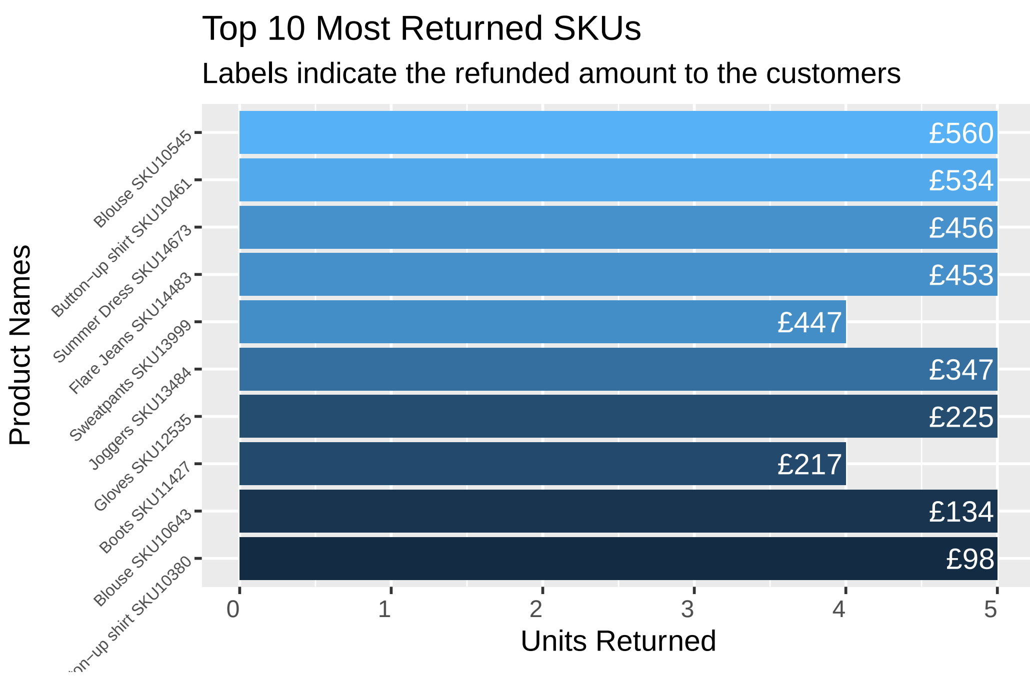
```
ggplot(data= Revenue_analysis_df %>% slice_max(TOTAL_REVENUE, n = 10)  ,
       aes(x = TOTAL_REVENUE,
       y = reorder( PRODUCT_DESCRIPTION, TOTAL_REVENUE ))) +
  geom_bar(stat = "identity", position = position_dodge(width = 0.75),
  aes(fill=TOTAL_REVENUE), show.legend = FALSE) +
  labs(x = "Revenue", y = "Product ID and Description",
  title = "10 Best Selling Products in Last 1 Year ",
  subtitle = "Labels indicate the revenue in GBP
  generating from each product") +
  geom_text(aes(label = paste("£", round(TOTAL_REVENUE),
  sep="")), color='white', hjust = 1) +
  theme(axis.text.x = element_text(hjust = 1))
```

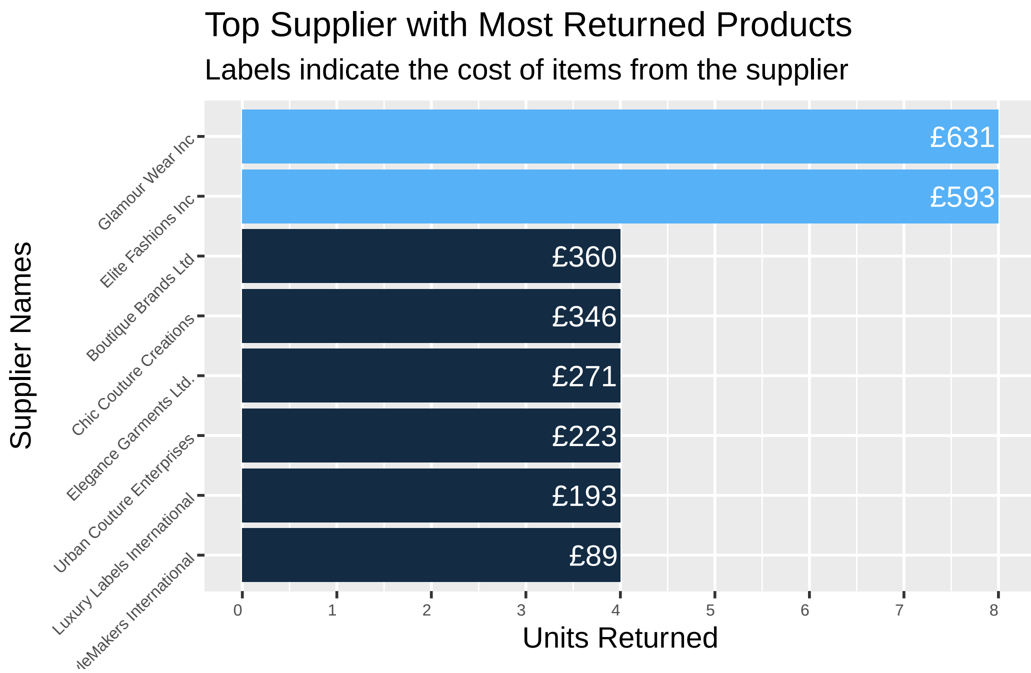
4.2: Comprehensive Reporting with Quarto

To present the data analysis, we use the R Quarto report, which can be easily changed to reflect the latest dataset. The analysis was divided into two parts. The first is a short-term analysis for monitoring current performance in the last thirty days. The second is performing long-term analysis, where we analyse changes on a yearly basis to demonstrate long-term progress.

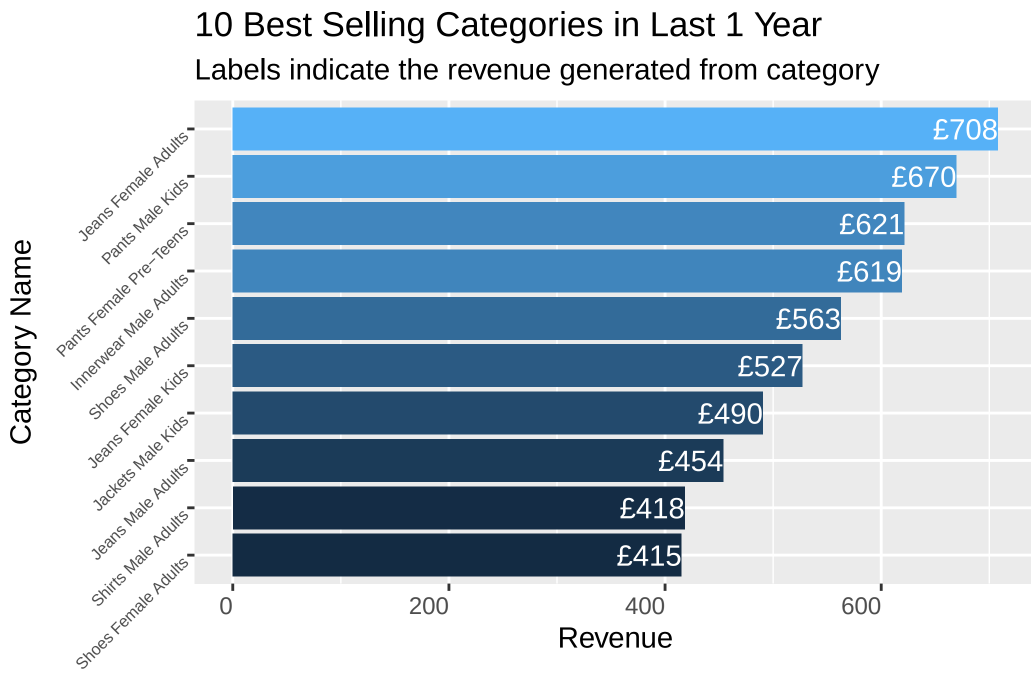
Short-Term Analysis

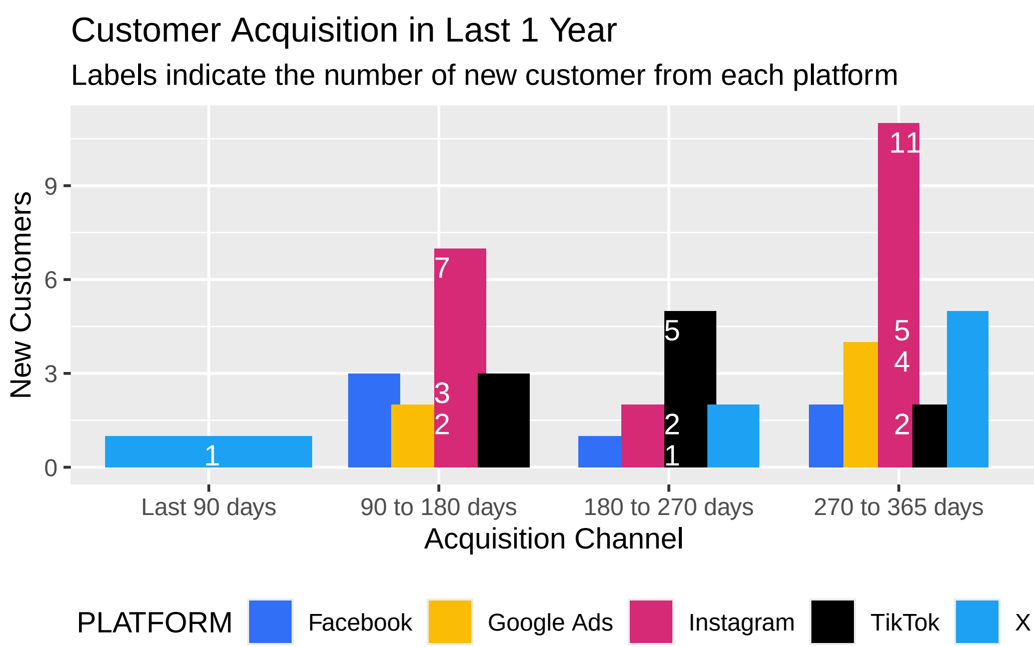


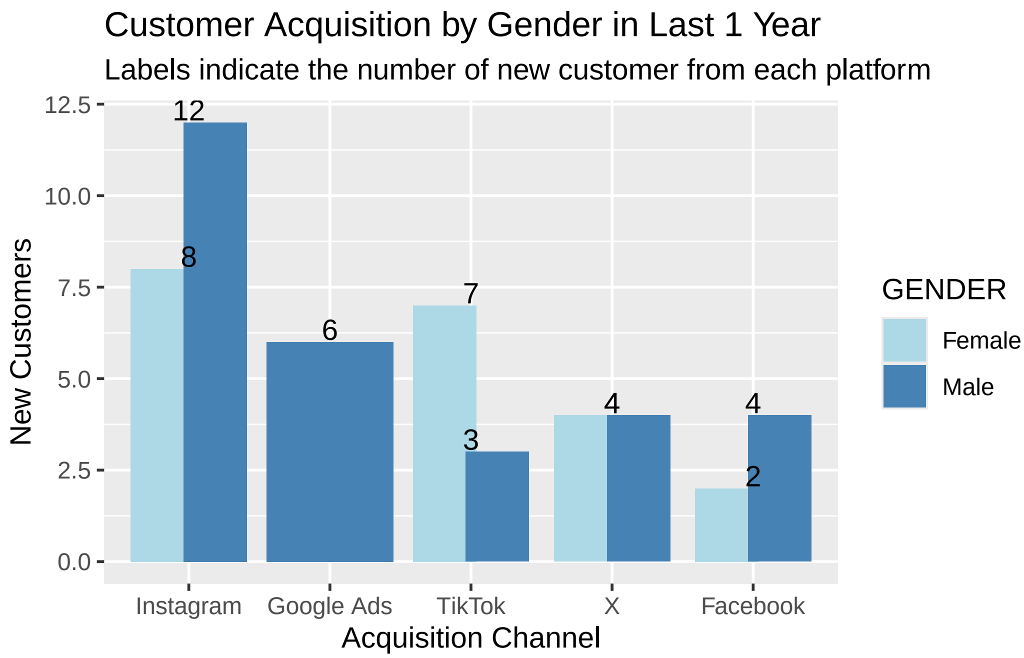




Long-Term Analysis


10 Best Selling Categories in Last 1 Year
Labels indicate the revenue generated from category
Category Name
Jeans Female Adults
Pants Male Kids
Pants Female Pre−Teens
Innerwear Male Adults
Shoes Male Adults
Jeans Female Kids
Jackets Male Kids
Jeans Male Adults
Shirts Male Adults
Shoes Female Adults
£708
£670
£621
£619
£563
£527
£490
£454
£418
£415
0
200
400
600
Revenue


Customer Acquisition in Last 1 Year
Labels indicate the number of new customer from each platform
New Customers
0
3
6
9
1
3
7
3
2
2
1
5
11
5
4
2
Last 90 days
90 to 180 days
180 to 270 days
270 to 365 days
Acquisition Channel
PLATFORM
Facebook
Google Ads
Instagram
TikTok
X



Conclusion

This project implemented a thorough approach for data management for a UK-based apparel e-commerce store. The database structure, depicted conceptually through an ER diagram, featured eight principal entities: products, customers, shipments, promotions, advertisements, suppliers, categories, and transactions, implemented via a SQL schema. Synthetic data, simulating genuine retail transactions, was generated in R. Rigorous quality assurance procedures preceded data insertion into the database. Subsequent data analysis was conducted using Quarto with R to provide actionable insights for management. Automation of data validation, loading, and analysis processes via a GitHub workflow enabled multi-stakeholder oversight and accountability across all project phases.

References

- Betebenner, D.W. (2021) *Generate Random Given and Surnames [R package randomNames version 1.5-0.0]*. https://cran.r-project.org/web/packages/randomNames/index.html. [date accessed: 1 March 2024

- Walczak, E. (2021) *API Wrapper Around 'Postcodes.io'* [R package PostcodesioR version 0.3.1]. https://cran.r-project.org/web/packages/PostcodesioR/index.html.

Appendix:

A1. Data Dictionary

Relationship: ORDERS

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
ORDER_ID	VARCHAR(30)	ODXXXX	PK		Uniquely identifies the different orders. Cannot be null
CUSTOMER_ID	VARCHAR(30)	CXXXX	FK	CUSTOMER	ID of the customer placing the order
ORDER_DATE	DATE	YYYY-MM-DD			Date when the order was placed
ORDER_STATUS	TEXT				Current status of the order
SHIPPING_ID	TEXT	SHXXXXX	FK	SHIPMENT	Uniquely identifies the different shipment. Cannot be null
DELIVERED_DATE	VARCHAR(15)	YYYY-MM-DD			The delivery date of the latest product in the order. Will be "NULL" if the order was cancelled
TRANSACTION_ID	VARCHAR(30)	TRXXXXX	FK	TRANSACTION	Unique transaction identifier
PRODUCT_ID	VARCHAR(30)	SKUXXXXX	FK	SKU	Unique identifier of Product
ORDER_QUANTITY	INTEGER				The quantity of products ordered by customer
RETURN_QUANTITY	INTEGER				The quantity to be refunded/ refunded to the customer
RETURN_DATE	DATE	YYYY-MM-DD			The date when the return transaction was initiated.

Entity: ORDER_SHIPMENT

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
SHIPPING_ID	TEXT	SHXXXXX	PK		Uniquely identifies the different shipment. Cannot be null
POST_CODE	VARCHAR(30)	XXX XXX			The address of the customer order
CARRIER	TEXT				The company name of carrier

Entity: TRANSACTION

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
TRANSACTION_ID	VARCHAR(30)	TRXXXXX	PK		Unique transaction identifier
PAYMENT_METHOD	VARCHAR(50)				The payment platforme that customer use
TRANSACTION_STATUS	VARCHAR(50)				Current status of the transaction - PROCESSING/DECLINED/COMPLETED

Entity: PROMOTION

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
PROMOTION_ID	VARCHAR(30)	PRXXX	PK		Unique identifer of promotion applied on each product
PROMOTION_VALUE	PERCENTAGE				The value of the promotion applied - as %
PROMOTION_START_DATE	VARCHAR(15)	YYYY-MM-DD			The time when the promotion starts to be valid
PROMOTION_END_DATE	VARCHAR(15)	YYYY-MM-DD			The time when the promotion code expires
MINIMUM_PURCHASE_AMOUNT	FLOAT				The minimum order value to apply the promotion/discount code
PRODUCT_ID	VARCHAR(30)	SKUXXXXX	FK	SKU	Unique identifier of Product

Entity: ADS

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
AD_ID	VARCHAR(30)	ADXXXX	PK		Uniquer identifer of advertisement
PLATFORM	VARCHAR(255)				The platform on which the advertisement is posted (e.g. Facebook, Instagram, GoogleAds, Tiktok, etc)
AD_TITLE	VARCHAR(255)				The title of the advertisement
AD_TYPE	VARCHAR(70)				Types of advertisement - DISPLAY ADS/ SOCIAL MEDIA ADS/ SEARCH ADS/ VIDEO ADS
AD_START_DATE	DATE	YYYY-MM-DD			The date when the advertisement starts running
AD_END_DATE	DATE	YYYY-MM-DD			The date when the advertisement ends
COST_PER_CLICK	FLOAT				The cost our company has to pay per each customer's click on the advertisement
CLICK_THROUGH_RATE	FLOAT				The percentage of users who clicked on the advertisement after seeing it
NUMBER_OF_CLICKS	FLOAT				The total number of times the advertisement has been clicked

Entity: CUSTOMERS

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
CUSTOMER_ID	VARCHAR(30)	CXXXX	PK		Uniquer identifer of customers
AD_ID	VARCHAR(30)	ADXXXX	FK	ADS	Uniquer identifer of advertisement
CUSTOMER_EMAIL	VARCHAR(30)	XXXXXX@XXX.COM			The registred email address of the customer
CUSTOMER_GENDER	TEXT				Gender of the customer representing male(M), female (F) and others (O)
PHONE_NUMBER	NUMERIC(10)	XXXXXXXXXXX			The registered phone number of the customer for contact purposes
LAST_NAME	TEXT				The last name of customer
FIRST_NAME	TEXT				The first name of customer
DATA_OF_BIRTH	VARCHAR(15)	YYYY-MM-DD			Represents date of birth of the customer
POST_CODE	VARCHAR(30)	XXX XXX			The postcode of billing adress of customer
HOUSE NUMBER	INT	XXX			The house number of billing adress of customer
ACQUISITION_TYPE	TEXT				Whether customers are acquired through ads (Organic/Paid)
REFERENCE_ID	VARCHAR(30)	CXXXX	FK	CUSTOMERS	Old customers who recommend products to new customers

Entity: CATEGORY

Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
CATEGORY_ID	VARCHAR(30)	CATXXXXX	PK		Unique identifier of CATEGORY which will tell class, department and division the product can belong.
CATEGORY_NAME	TEXT				Description of Category_ID at product type level (Jeans, Tops, sweatshirts, T-shirts, etc.)
GENDER	TEXT				Gender for cloths (male, female)
AGE_GROUP	VARCHAR(30)				Age range of target customer (Kids, Teenage, Young Adult, Adults, Eldery)

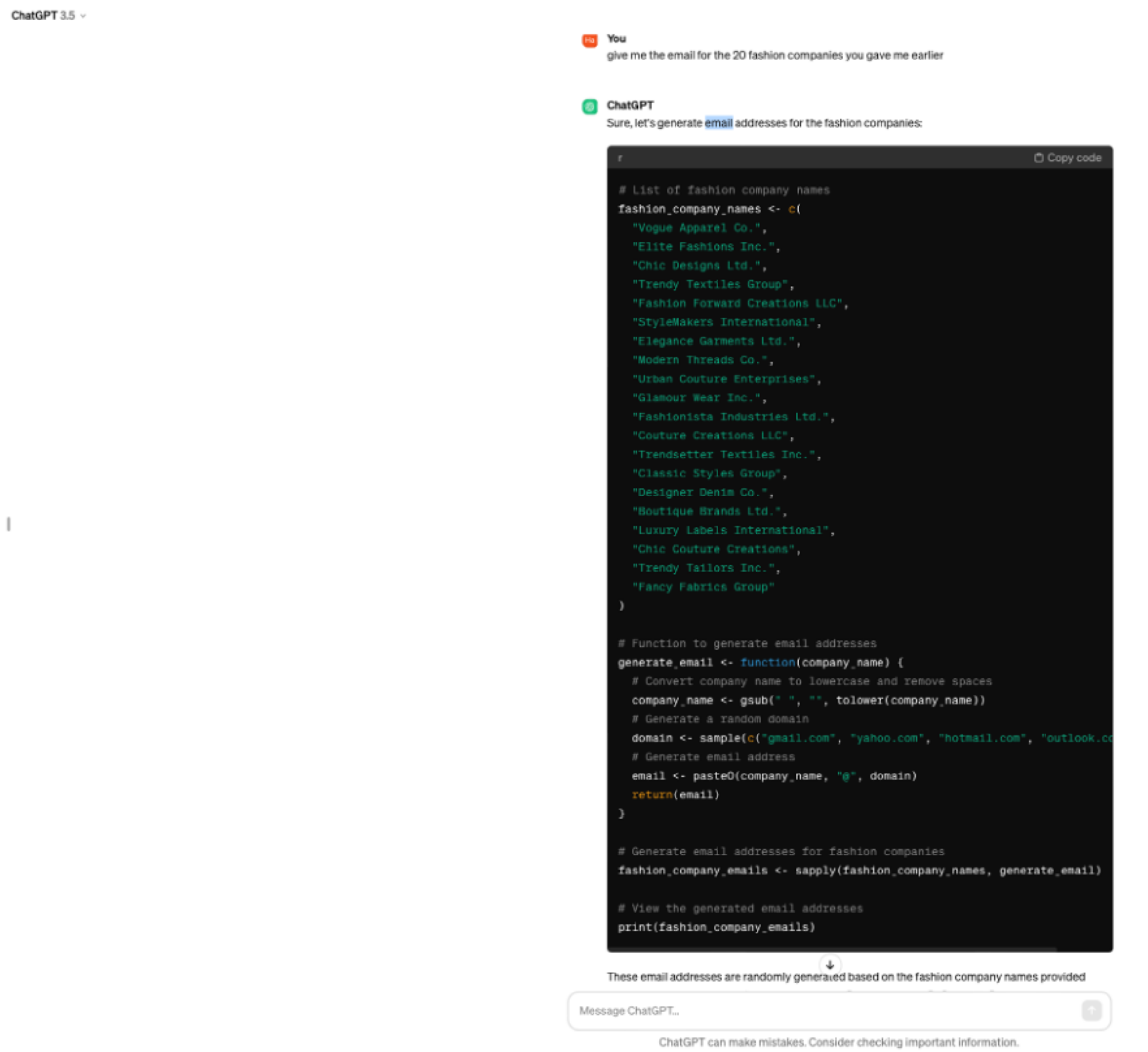
Entity: SUPPLIER

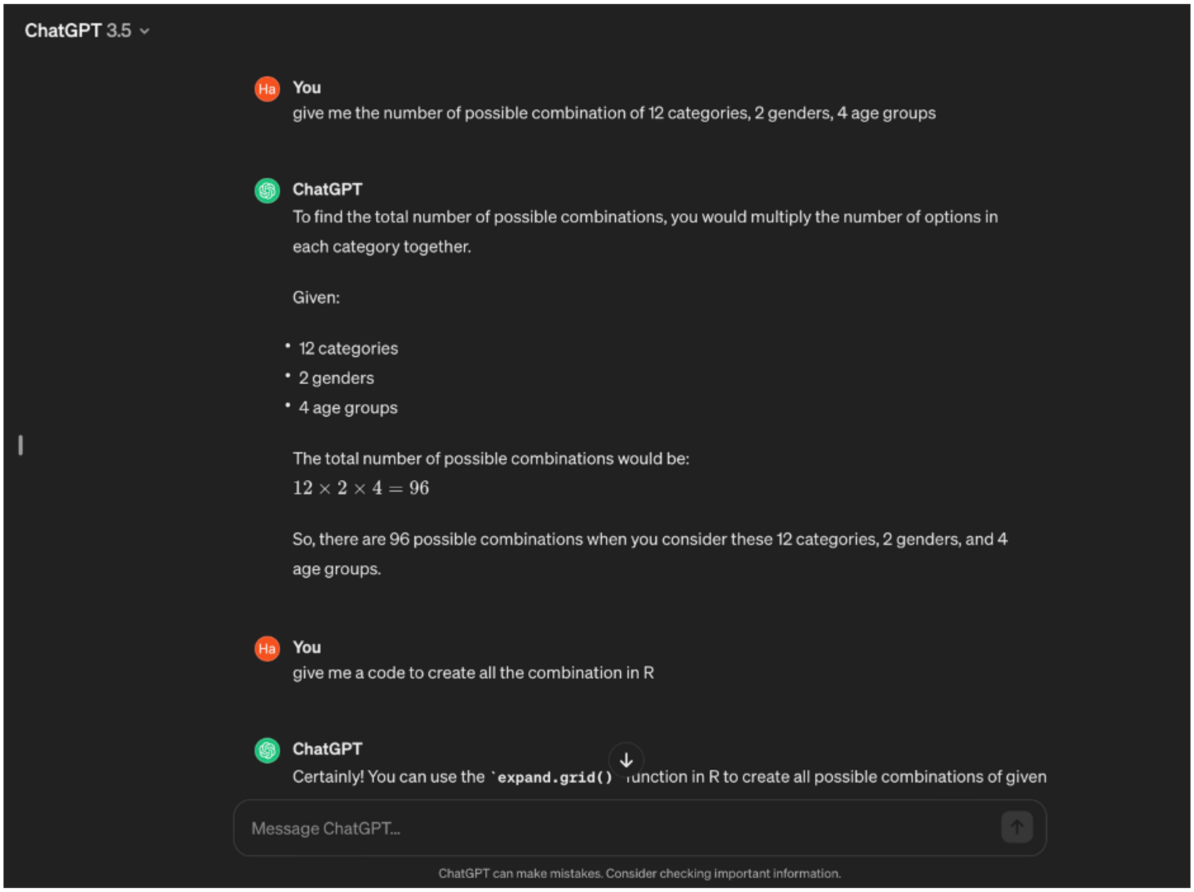
Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
SUPPLER_ID	VARCHAR(30)	SPXXXXX	PK		Unique identifier of Supplier company
SUPPLIER_NAME	VARCHAR(50)				Description of Supplier (example; Evergreen Trading Co., etc.)
SUPPLIER_PHONE	VARCHAR(10)	XXXXXXXXXX			registered phone number of the supplier
POST_CODE	VARCHAR(30)	XXX XXX			The location of the supplier
SUPPLIER_EMAIL	VARCHAR(30)	XXXXXX@XXX.COM			registered email address of the supplier

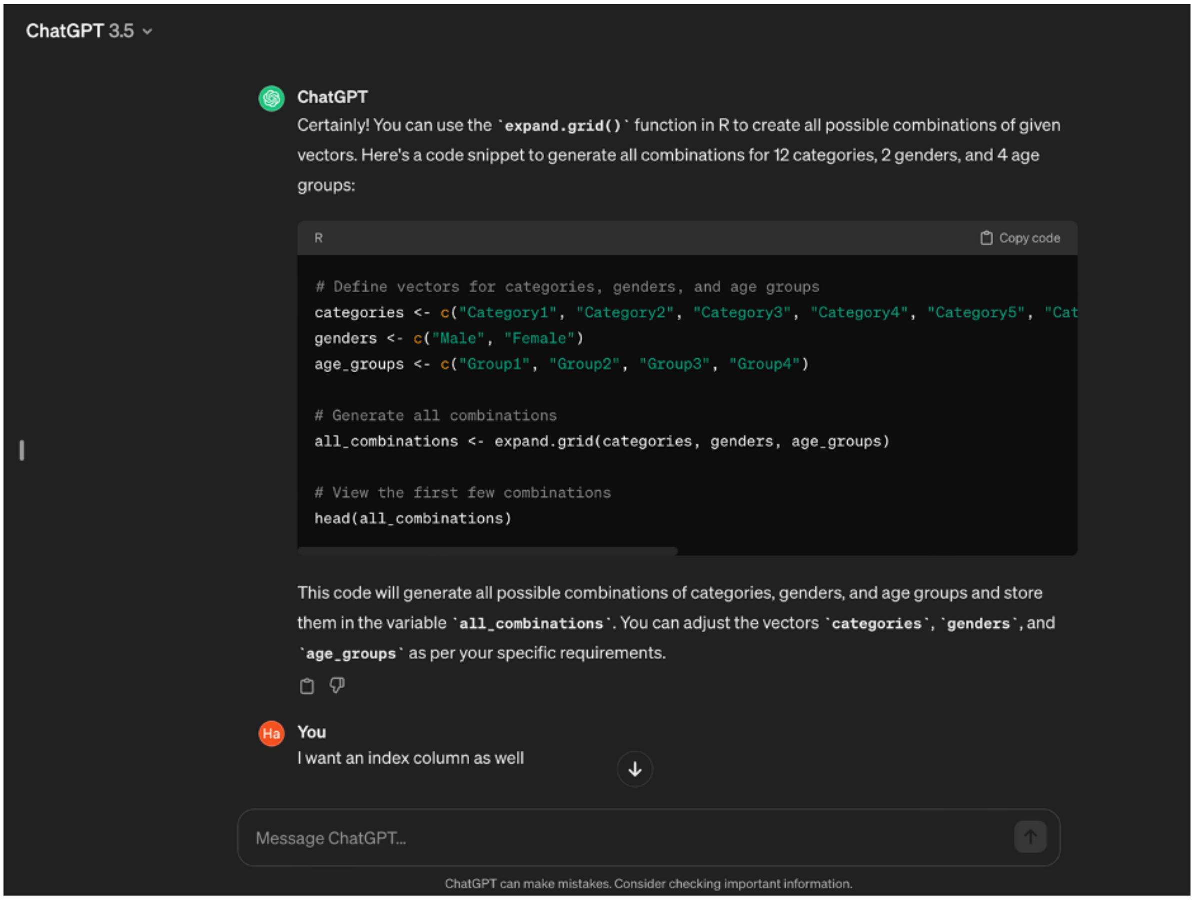
Entity: SKU

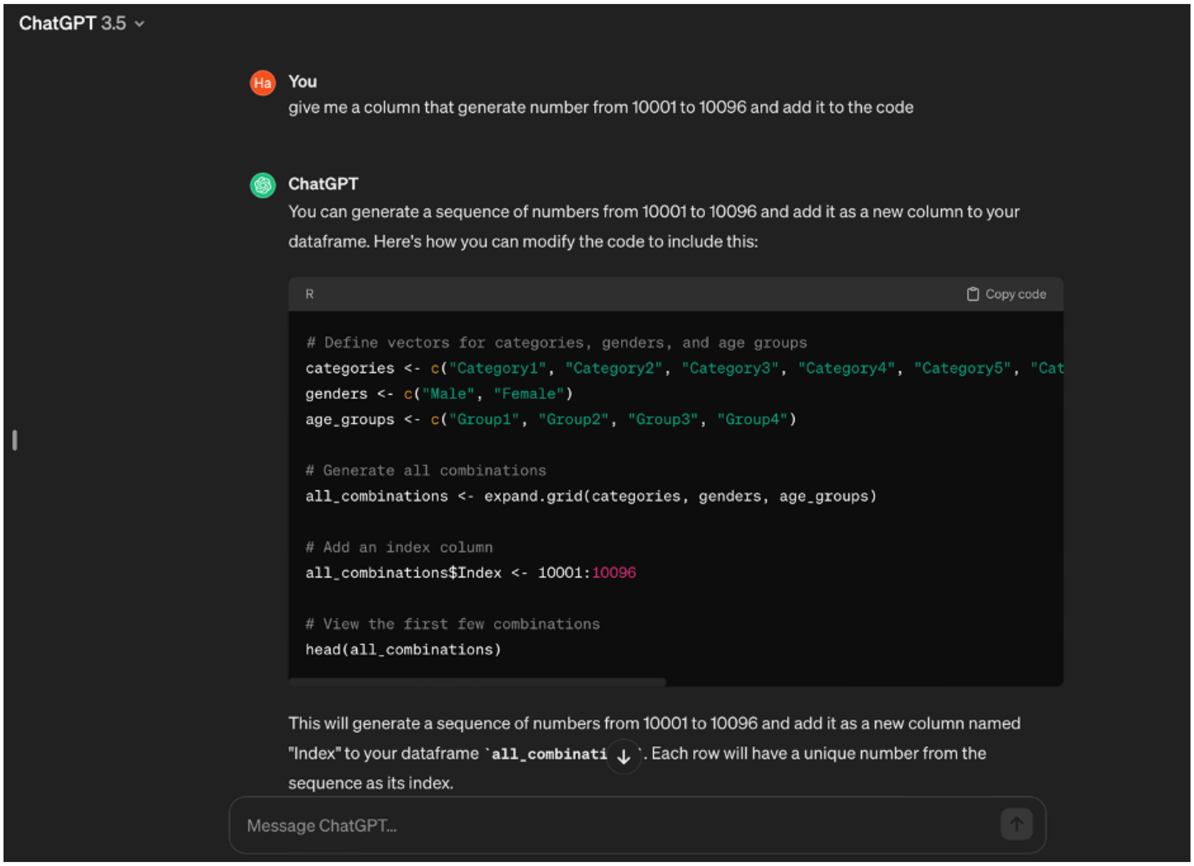
Attribute Name	Data Type	Format	PK (or) FK	FK Reference Entity	Description
PRODUCT_ID	VARCHAR(30)	SKUXXXXX	PK		Unique identifier of Product
SUPPLER_ID	VARCHAR(30)	SPXXXXX	FK	SUPPLIER	Unique identifier of Supplier company
CATEGORY_ID	VARCHAR(30)	CATXXXXX	FK	CATEGORY	Unique identifier of CATEGORY which will tell class, department and division the product can belong.
PRODUCT_NAME	TEXT				The name of the product
PRODUCT_PURCHASING_PRICE	FLOAT	00.00			The cost incurred to acquire a single product unit
COLOR	TEXT	XXXX			The color of the product
SIZE	TEXT				The size of the product
MARKUP	FLOAT	00.00			The markup of the product

A2. LLM Prompt Sequence




ChatGPT 3.5
You
give me the number of possible combination of 12 categories, 2 genders, 4 age groups
ChatGPT
To find the total number of possible combinations, you would multiply the number of options in each category together.
Given:
• 12 categories
• 2 genders
• 4 age groups
The total number of possible combinations would be:
$12 \times 2 \times 4 = 96$
So, there are 96 possible combinations when you consider these 12 categories, 2 genders, and 4 age groups.
You
give me a code to create all the combination in R
ChatGPT
Certainly! You can use the `expand.grid()` function in R to create all possible combinations of given
Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.


ChatGPT 3.5
ChatGPT
Certainly! You can use the `expand.grid()` function in R to create all possible combinations of given vectors. Here's a code snippet to generate all combinations for 12 categories, 2 genders, and 4 age groups:
R
Copy code
# Define vectors for categories, genders, and age groups
categories <- c("Category1", "Category2", "Category3", "Category4", "Category5", "Cat
genders <- c("Male", "Female")
age_groups <- c("Group1", "Group2", "Group3", "Group4")
# Generate all combinations
all_combinations <- expand.grid(categories, genders, age_groups)
# View the first few combinations
head(all_combinations)
This code will generate all possible combinations of categories, genders, and age groups and store them in the variable `all_combinations`. You can adjust the vectors `categories`, `genders`, and `age_groups` as per your specific requirements.
Ha
You
I want an index column as well
Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.



ChatGPT 3.5

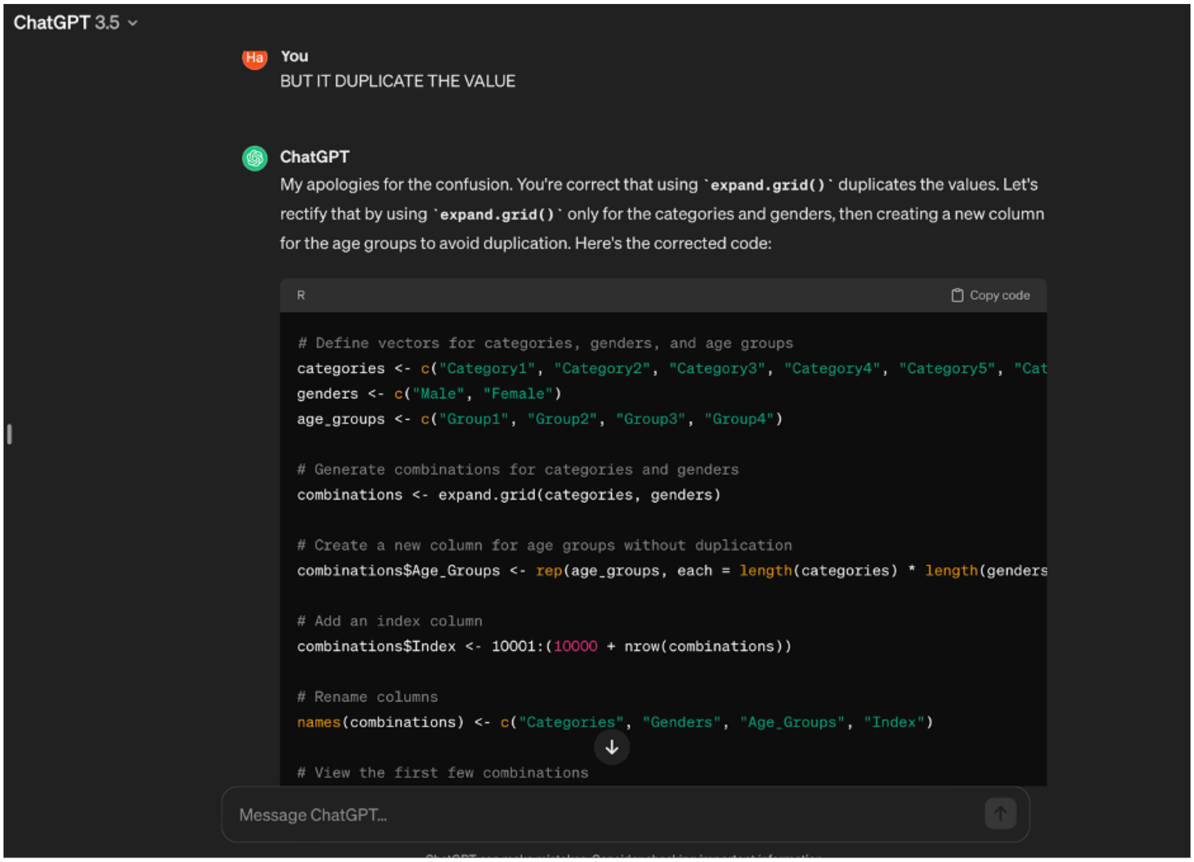
You
give me a column that generate number from 10001 to 10096 and add it to the code

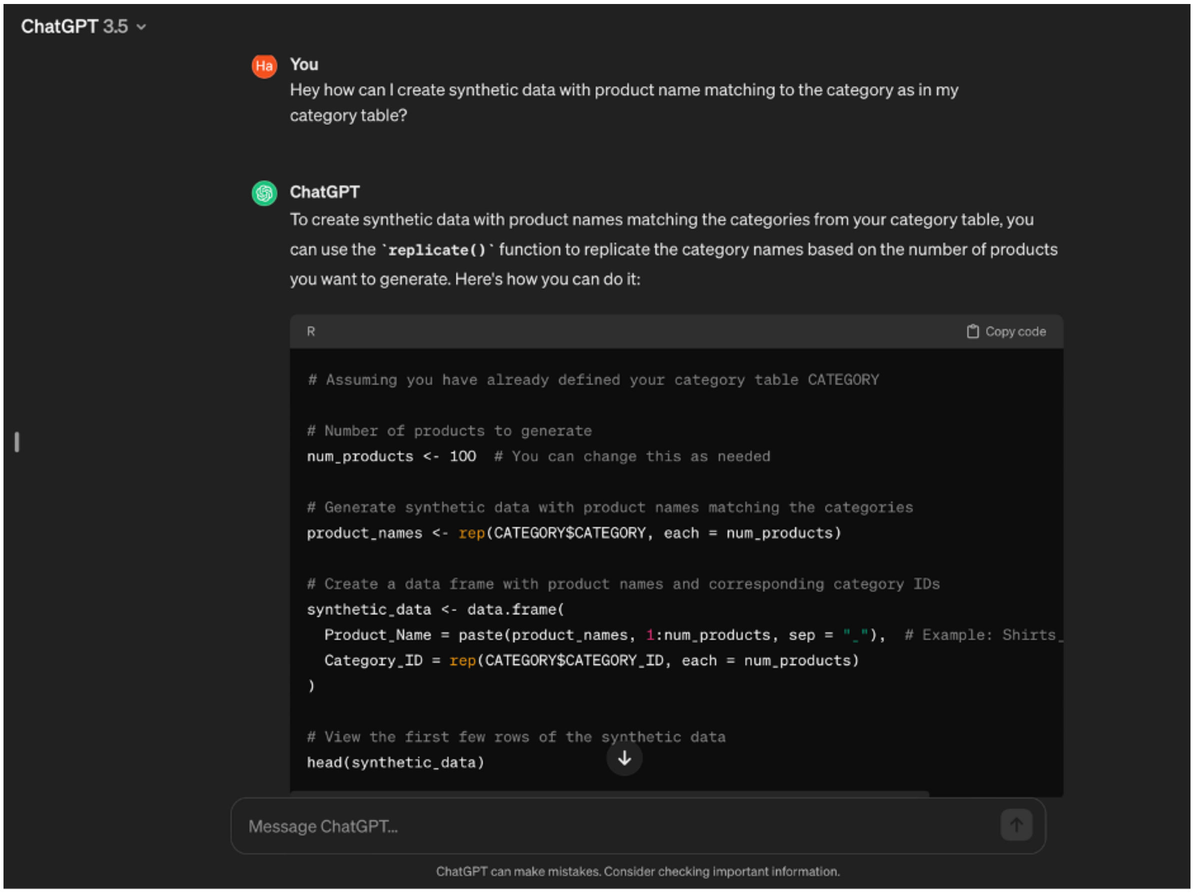
ChatGPT
You can generate a sequence of numbers from 10001 to 10096 and add it as a new column to your dataframe. Here's how you can modify the code to include this:

```R
# Define vectors for categories, genders, and age groups
categories <- c("Category1", "Category2", "Category3", "Category4", "Category5", "Cat
genders <- c("Male", "Female")
age_groups <- c("Group1", "Group2", "Group3", "Group4")

# Generate all combinations
all_combinations <- expand.grid(categories, genders, age_groups)

# Add an index column
all_combinations$Index <- 10001:10096

# View the first few combinations
head(all_combinations)
```

This will generate a sequence of numbers from 10001 to 10096 and add it as a new column named "Index" to your dataframe `all_combinati`. Each row will have a unique number from the sequence as its index.

Message ChatGPT...


ChatGPT 3.5
You
BUT IT DUPLICATE THE VALUE
ChatGPT
My apologies for the confusion. You're correct that using `expand.grid()` duplicates the values. Let's rectify that by using `expand.grid()` only for the categories and genders, then creating a new column for the age groups to avoid duplication. Here's the corrected code:
R
Copy code
# Define vectors for categories, genders, and age groups
categories <- c("Category1", "Category2", "Category3", "Category4", "Category5", "Cat
genders <- c("Male", "Female")
age_groups <- c("Group1", "Group2", "Group3", "Group4")
# Generate combinations for categories and genders
combinations <- expand.grid(categories, genders)
# Create a new column for age groups without duplication
combinations$Age_Groups <- rep(age_groups, each = length(categories) * length(genders
Add an index column
combinations$Index <- 10001:(10000 + nrow(combinations))
# Rename columns
names(combinations) <- c("Categories", "Genders", "Age_Groups", "Index")
# View the first few combinations
Message ChatGPT...



ChatGPT 3.5

You

Hey how can I create synthetic data with product name matching to the category as in my category table?

ChatGPT

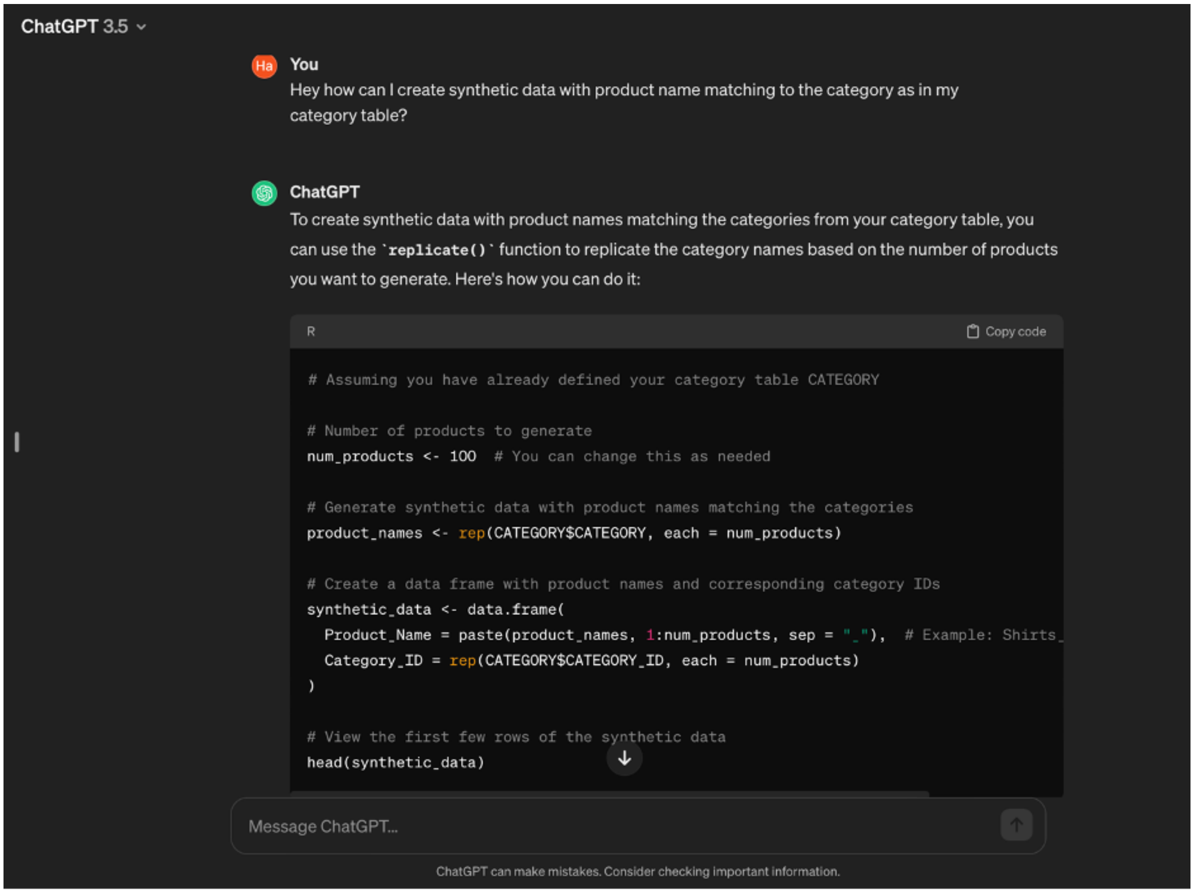
To create synthetic data with product names matching the categories from your category table, you can use the `replicate()` function to replicate the category names based on the number of products you want to generate. Here's how you can do it:

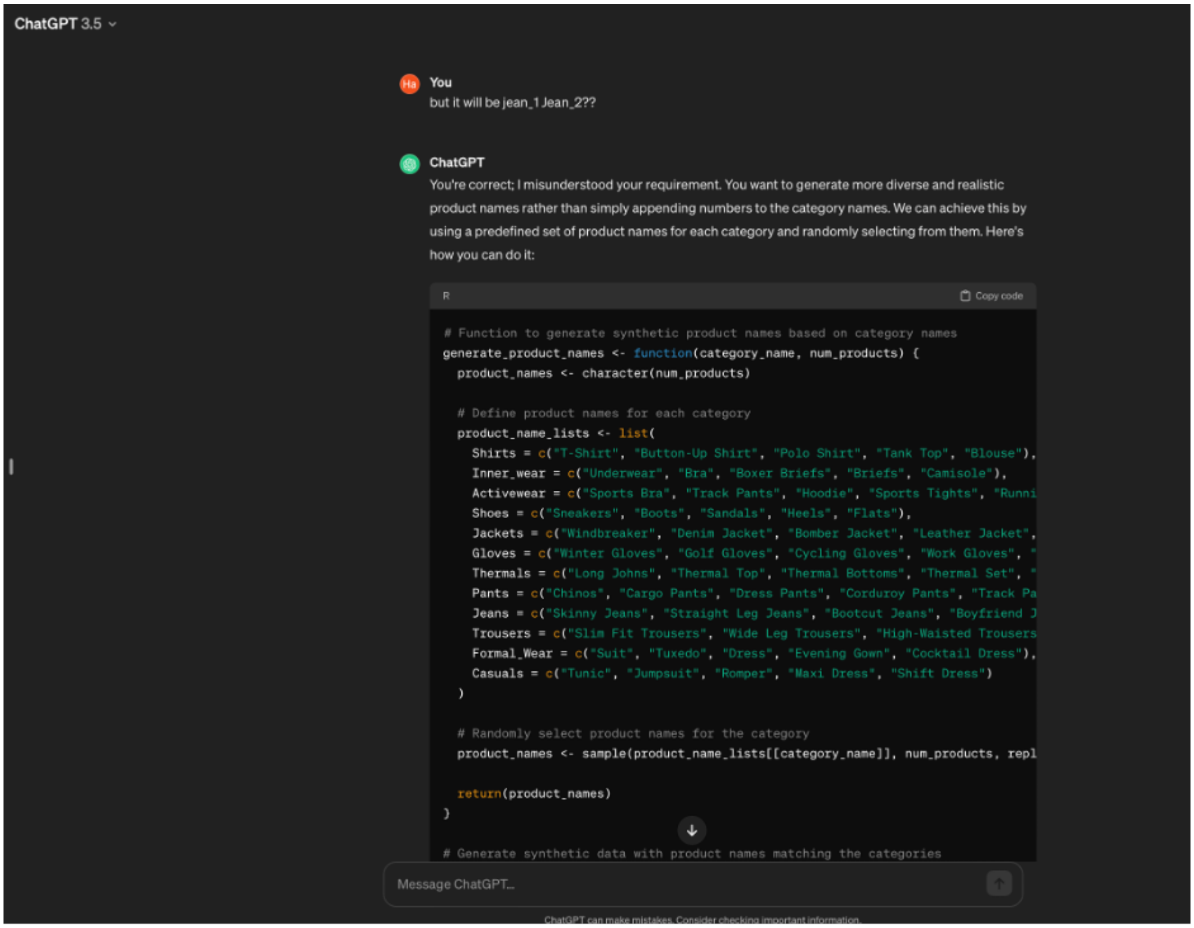
R

```r
# Assuming you have already defined your category table CATEGORY

# Number of products to generate
num_products <- 100  # You can change this as needed

# Generate synthetic data with product names matching the categories
product_names <- rep(CATEGORY$CATEGORY, each = num_products)

# Create a data frame with product names and corresponding category IDs
synthetic_data <- data.frame(
  Product_Name = paste(product_names, 1:num_products, sep = "_"),  # Example: Shirts_
  Category_ID = rep(CATEGORY$CATEGORY_ID, each = num_products)
)

# View the first few rows of the synthetic data
head(synthetic_data)
```

Message ChatGPT...

ChatGPT can make mistakes. Consider checking important information.


ChatGPT 3.5
You
Hey how can I create synthetic data with product name matching to the category as in my category table?
ChatGPT
To create synthetic data with product names matching the categories from your category table, you can use the `replicate()` function to replicate the category names based on the number of products you want to generate. Here's how you can do it:
R
Copy code
# Assuming you have already defined your category table CATEGORY
# Number of products to generate
num_products <- 100 # You can change this as needed
# Generate synthetic data with product names matching the categories
product_names <- rep(CATEGORY$CATEGORY, each = num_products)
# Create a data frame with product names and corresponding category IDs
synthetic_data <- data.frame(
Product_Name = paste(product_names, 1:num_products, sep = "_"), # Example: Shirts_
Category_ID = rep(CATEGORY$CATEGORY_ID, each = num_products)
)
# View the first few rows of the synthetic data
head(synthetic_data)
Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.


ChatGPT 3.5

You

but it will be jean_1 Jean_2??

ChatGPT

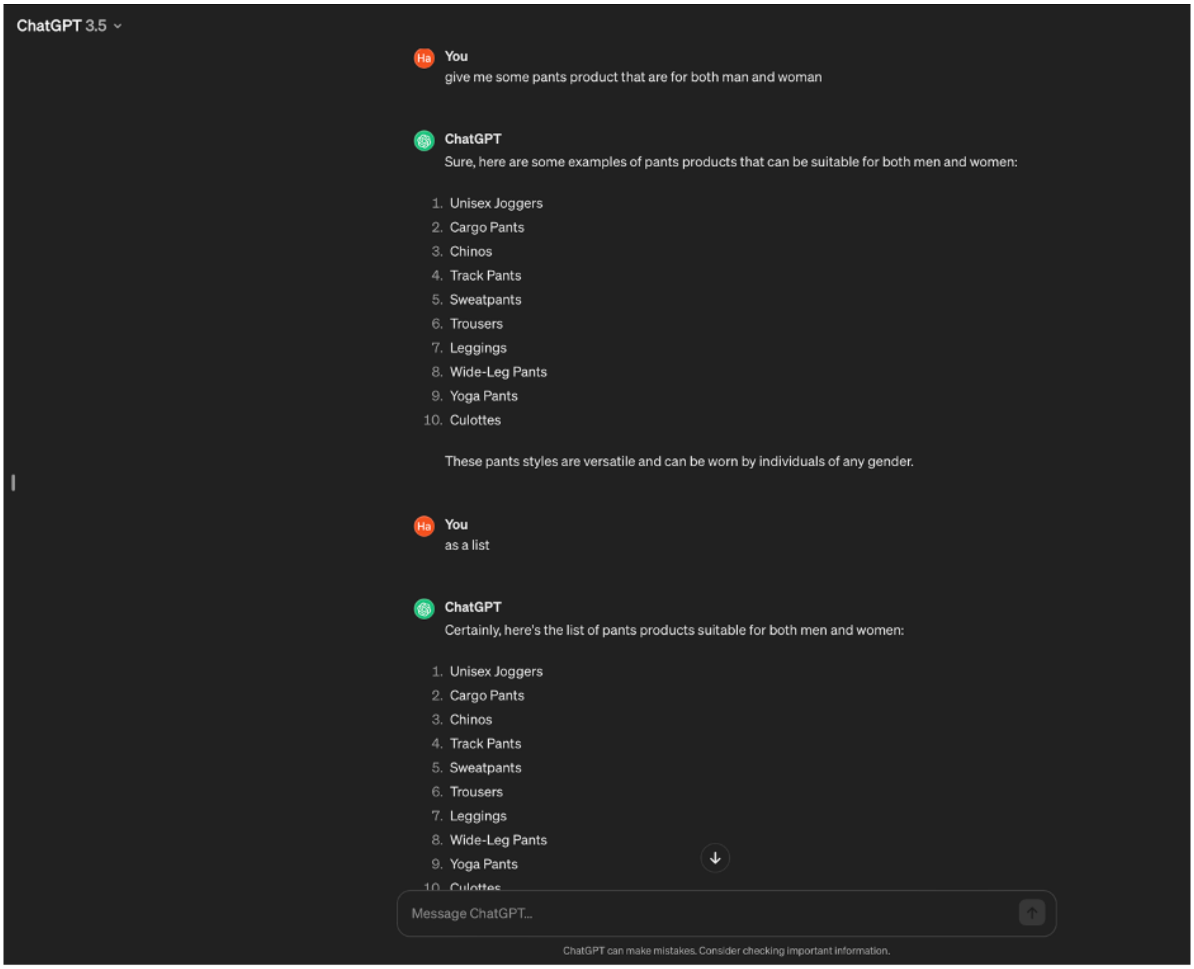
You're correct; I misunderstood your requirement. You want to generate more diverse and realistic product names rather than simply appending numbers to the category names. We can achieve this by using a predefined set of product names for each category and randomly selecting from them. Here's how you can do it:

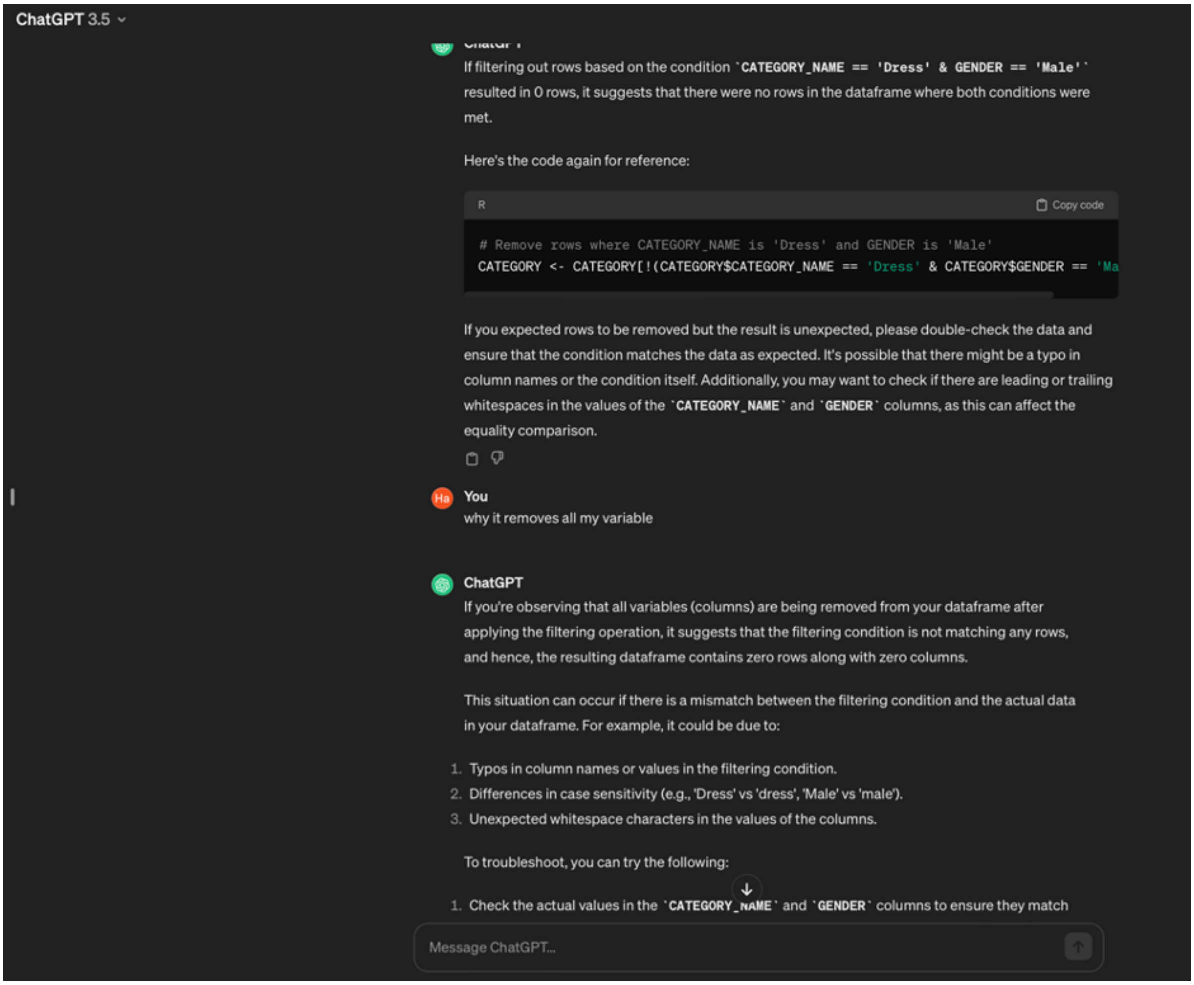
R Copy code

```r
# Function to generate synthetic product names based on category names
generate_product_names <- function(category_name, num_products) {
  product_names <- character(num_products)

  # Define product names for each category
  product_name_lists <- list(
    Shirts = c("T-Shirt", "Button-Up Shirt", "Polo Shirt", "Tank Top", "Blouse"),
    Inner_wear = c("Underwear", "Bra", "Boxer Briefs", "Briefs", "Camisole"),
    Activewear = c("Sports Bra", "Track Pants", "Hoodie", "Sports Tights", "Runni
    Shoes = c("Sneakers", "Boots", "Sandals", "Heels", "Flats"),
    Jackets = c("Windbreaker", "Denim Jacket", "Bomber Jacket", "Leather Jacket",
    Gloves = c("Winter Gloves", "Golf Gloves", "Cycling Gloves", "Work Gloves", "
    Thermals = c("Long Johns", "Thermal Top", "Thermal Bottoms", "Thermal Set", "
    Pants = c("Chinos", "Cargo Pants", "Dress Pants", "Corduroy Pants", "Track Pa
    Jeans = c("Skinny Jeans", "Straight Leg Jeans", "Bootcut Jeans", "Boyfriend J
    Trousers = c("Slim Fit Trousers", "Wide Leg Trousers", "High-Waisted Trousers
    Formal_Wear = c("Suit", "Tuxedo", "Dress", "Evening Gown", "Cocktail Dress"),
    Casuals = c("Tunic", "Jumpsuit", "Romper", "Maxi Dress", "Shift Dress")
  )

  # Randomly select product names for the category
  product_names <- sample(product_name_lists[[category_name]], num_products, repl

  return(product_names)
}

# Generate synthetic data with product names matching the categories
```

Message ChatGPT...

ChatGPT can make mistakes. Consider checking important information.


ChatGPT 3.5
You
give me some pants product that are for both man and woman
ChatGPT
Sure, here are some examples of pants products that can be suitable for both men and women:
1. Unisex Joggers
2. Cargo Pants
3. Chinos
4. Track Pants
5. Sweatpants
6. Trousers
7. Leggings
8. Wide-Leg Pants
9. Yoga Pants
10. Culottes
These pants styles are versatile and can be worn by individuals of any gender.
You
as a list
ChatGPT
Certainly, here's the list of pants products suitable for both men and women:
1. Unisex Joggers
2. Cargo Pants
3. Chinos
4. Track Pants
5. Sweatpants
6. Trousers
7. Leggings
8. Wide-Leg Pants
9. Yoga Pants
Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.


ChatGPT 3.5

ChatGPT

If filtering out rows based on the condition `CATEGORY_NAME == 'Dress' & GENDER == 'Male'` resulted in 0 rows, it suggests that there were no rows in the dataframe where both conditions were met.

Here's the code again for reference:

R Copy code

```r
# Remove rows where CATEGORY_NAME is 'Dress' and GENDER is 'Male'
CATEGORY <- CATEGORY[!(CATEGORY$CATEGORY_NAME == 'Dress' & CATEGORY$GENDER == 'Ma
```

If you expected rows to be removed but the result is unexpected, please double-check the data and ensure that the condition matches the data as expected. It's possible that there might be a typo in column names or the condition itself. Additionally, you may want to check if there are leading or trailing whitespaces in the values of the `CATEGORY_NAME` and `GENDER` columns, as this can affect the equality comparison.

You

why it removes all my variable

ChatGPT

If you're observing that all variables (columns) are being removed from your dataframe after applying the filtering operation, it suggests that the filtering condition is not matching any rows, and hence, the resulting dataframe contains zero rows along with zero columns.

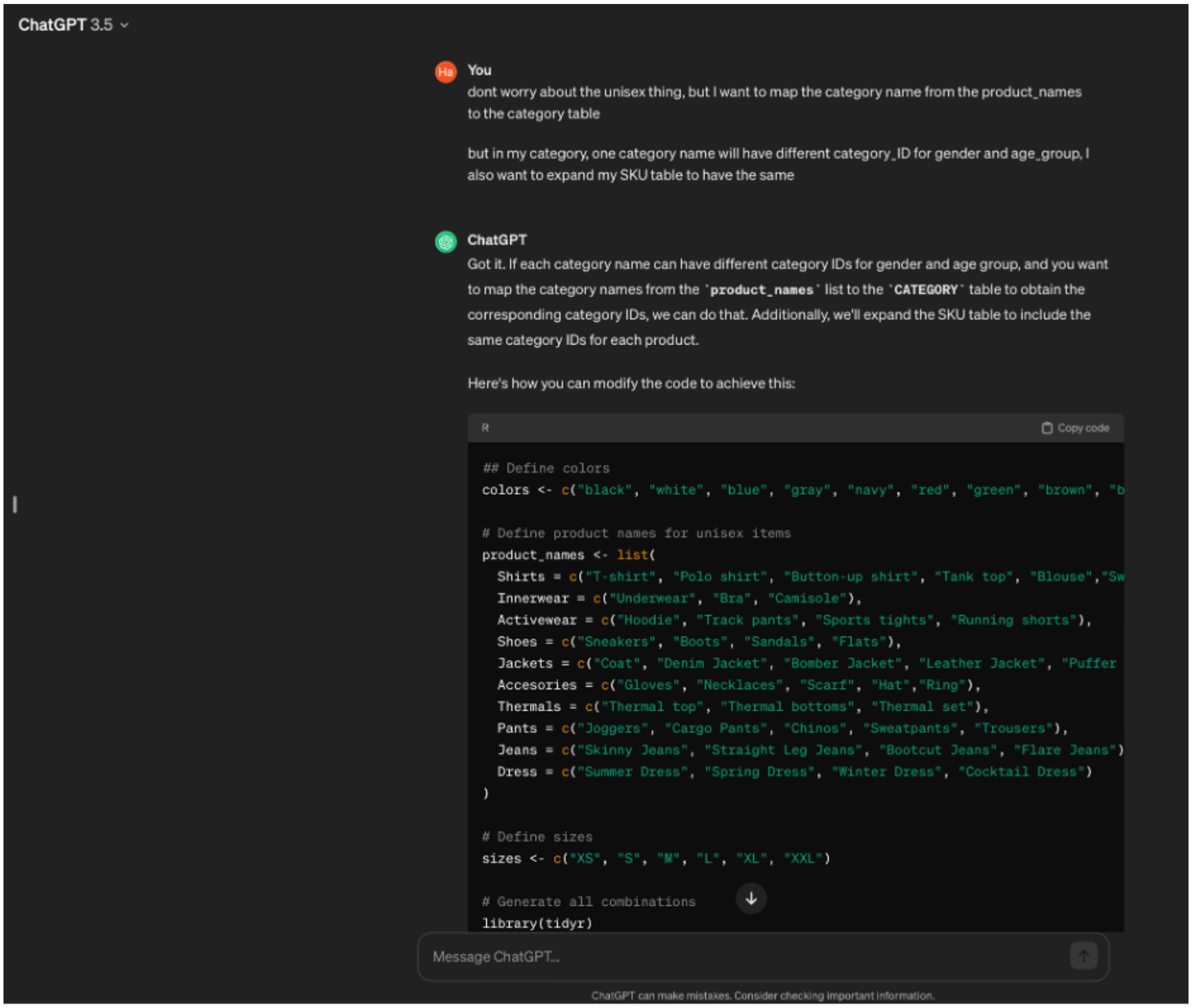
This situation can occur if there is a mismatch between the filtering condition and the actual data in your dataframe. For example, it could be due to:

1. Typos in column names or values in the filtering condition.
2. Differences in case sensitivity (e.g., 'Dress' vs 'dress', 'Male' vs 'male').
3. Unexpected whitespace characters in the values of the columns.

To troubleshoot, you can try the following:

1. Check the actual values in the `CATEGORY_NAME` and `GENDER` columns to ensure they match

Message ChatGPT...


ChatGPT 3.5

You
dont worry about the unisex thing, but I want to map the category name from the product_names to the category table

but in my category, one category name will have different category_ID for gender and age_group, I also want to expand my SKU table to have the same

ChatGPT
Got it. If each category name can have different category IDs for gender and age group, and you want to map the category names from the `product_names` list to the `CATEGORY` table to obtain the corresponding category IDs, we can do that. Additionally, we'll expand the SKU table to include the same category IDs for each product.

Here's how you can modify the code to achieve this:

R Copy code

```r
## Define colors
colors <- c("black", "white", "blue", "gray", "navy", "red", "green", "brown", "b

# Define product names for unisex items
product_names <- list(
  Shirts = c("T-shirt", "Polo shirt", "Button-up shirt", "Tank top", "Blouse","Sw
  Innerwear = c("Underwear", "Bra", "Camisole"),
  Activewear = c("Hoodie", "Track pants", "Sports tights", "Running shorts"),
  Shoes = c("Sneakers", "Boots", "Sandals", "Flats"),
  Jackets = c("Coat", "Denim Jacket", "Bomber Jacket", "Leather Jacket", "Puffer
  Accesories = c("Gloves", "Necklaces", "Scarf", "Hat","Ring"),
  Thermals = c("Thermal top", "Thermal bottoms", "Thermal set"),
  Pants = c("Joggers", "Cargo Pants", "Chinos", "Sweatpants", "Trousers"),
  Jeans = c("Skinny Jeans", "Straight Leg Jeans", "Bootcut Jeans", "Flare Jeans")
  Dress = c("Summer Dress", "Spring Dress", "Winter Dress", "Cocktail Dress")
)

# Define sizes
sizes <- c("XS", "S", "M", "L", "XL", "XXL")

# Generate all combinations
library(tidyr)
```

Message ChatGPT...

ChatGPT can make mistakes. Consider checking important information.

ChatGPT 3.5

You

again you miss the product_names???

ChatGPT

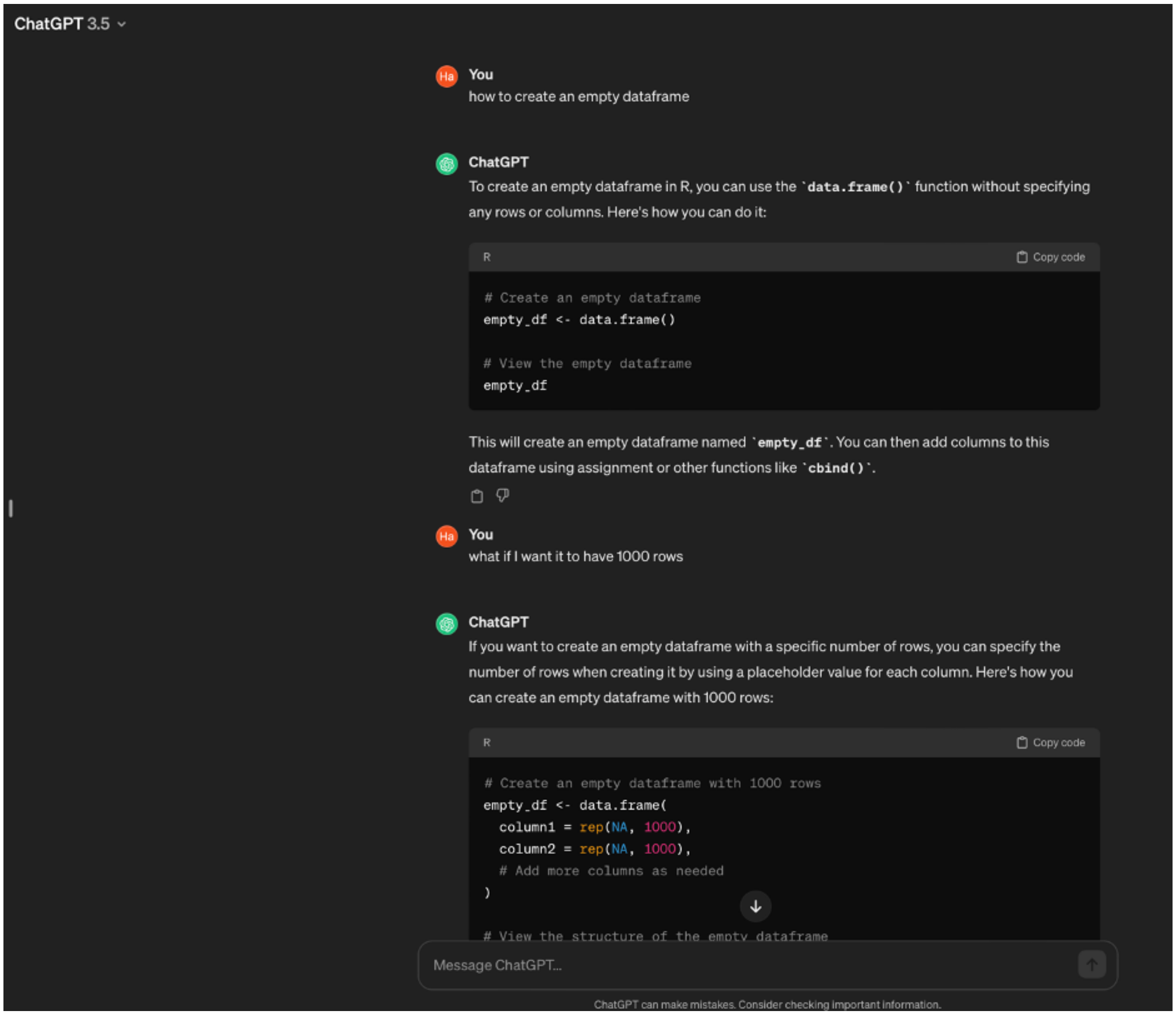
You're absolutely correct, and I apologize for the oversight. Let's correct that. We'll directly define the `product_names` within the code:

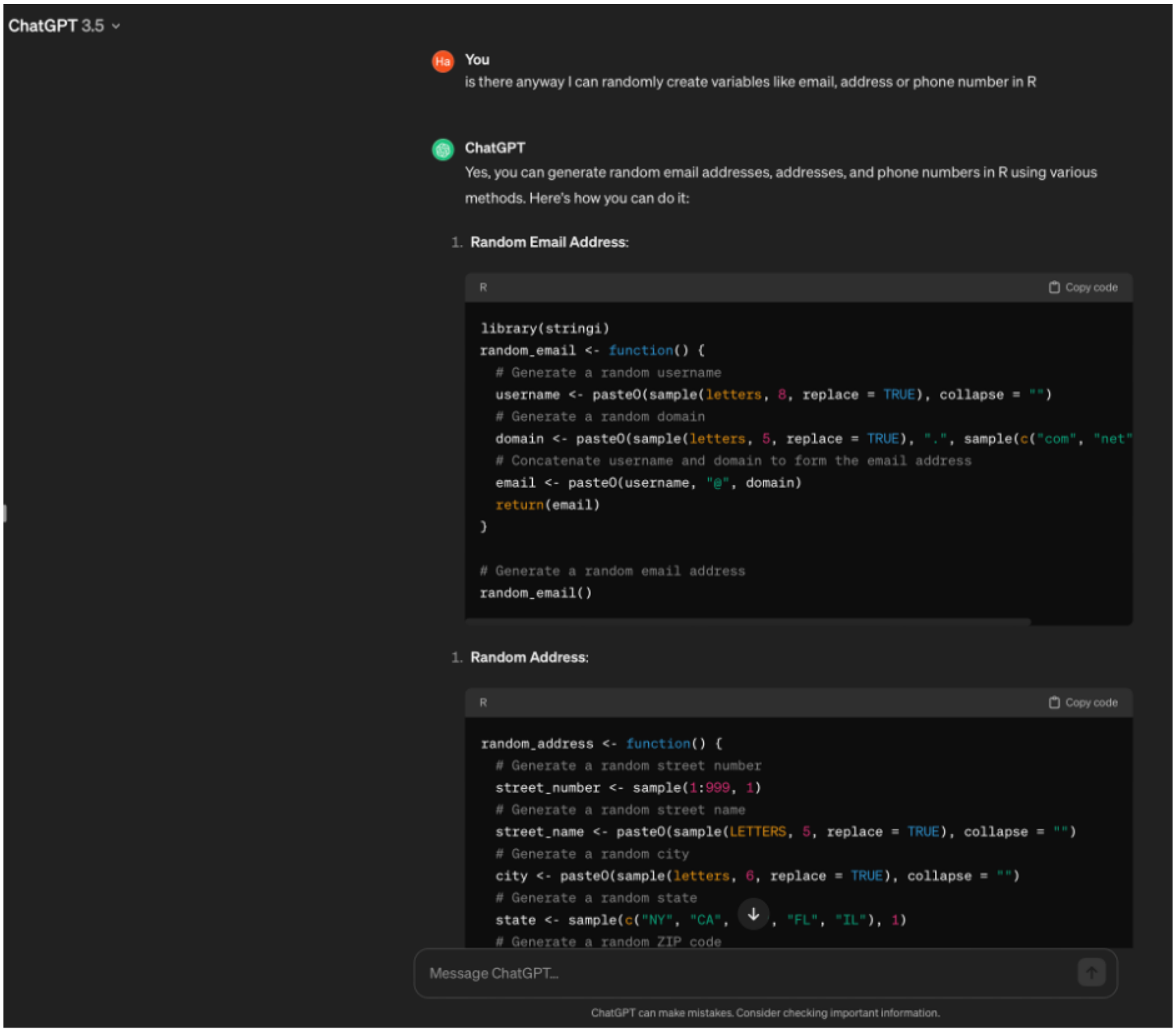
R   Copy code

```r
## Define colors
colors <- c("black", "white", "blue", "gray", "navy", "red", "green", "brown", "b

# Define sizes
sizes <- c("XS", "S", "M", "L", "XL", "XXL")

# Define product names for each category
product_names <- list(
  Shirts = c("T-shirt", "Polo shirt", "Button-up shirt", "Tank top", "Blouse","Sw
  Innerwear = c("Underwear", "Bra", "Camisole"),
  Activewear = c("Hoodie", "Track pants", "Sports tights", "Running shorts"),
  Shoes = c("Sneakers", "Boots", "Sandals", "Flats"),
  Jackets = c("Coat", "Denim Jacket", "Bomber Jacket", "Leather Jacket", "Puffer
  Accessories = c("Gloves", "Necklaces", "Scarf", "Hat","Ring"),
  Thermals = c("Thermal top", "Thermal bottoms", "Thermal set"),
  Pants = c("Joggers", "Cargo Pants", "Chinos", "Sweatpants", "Trousers"),
  Jeans = c("Skinny Jeans", "Straight Leg Jeans", "Bootcut Jeans", "Flare Jeans")
  Dress = c("Summer Dress", "Spring Dress", "Winter Dress", "Cocktail Dress")
)

# Create a dataframe to store products
products <- do.call(rbind, lapply(names(product_names), function(category) {
  expand.grid(
    CATEGORY_NAME = category,
    PRODUCT_NAME = product_names[[category]],
    COLOR = colors,
    SIZE = sizes,
    stringsAsFactors = FALSE
  )
}))
```

Message ChatGPT...

ChatGPT can make mistakes. Consider checking important information.


ChatGPT 3.5
You
how to create an empty dataframe
ChatGPT
To create an empty dataframe in R, you can use the `data.frame()` function without specifying any rows or columns. Here's how you can do it:
R
Copy code
# Create an empty dataframe
empty_df <- data.frame()
# View the empty dataframe
empty_df
This will create an empty dataframe named `empty_df`. You can then add columns to this dataframe using assignment or other functions like `cbind()`.
You
what if I want it to have 1000 rows
ChatGPT
If you want to create an empty dataframe with a specific number of rows, you can specify the number of rows when creating it by using a placeholder value for each column. Here's how you can create an empty dataframe with 1000 rows:
R
Copy code
# Create an empty dataframe with 1000 rows
empty_df <- data.frame(
column1 = rep(NA, 1000),
column2 = rep(NA, 1000),
# Add more columns as needed
)
View the structure of the empty dataframe
Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.



ChatGPT 3.5

You

is there anyway I can randomly create variables like email, address or phone number in R

ChatGPT

Yes, you can generate random email addresses, addresses, and phone numbers in R using various methods. Here's how you can do it:

1. **Random Email Address:**

R Copy code

```
library(stringi)
random_email <- function() {
  # Generate a random username
  username <- paste0(sample(letters, 8, replace = TRUE), collapse = "")
  # Generate a random domain
  domain <- paste0(sample(letters, 5, replace = TRUE), ".", sample(c("com", "net"
  # Concatenate username and domain to form the email address
  email <- paste0(username, "@", domain)
  return(email)
}

# Generate a random email address
random_email()
```

1. **Random Address:**

R Copy code

```
random_address <- function() {
  # Generate a random street number
  street_number <- sample(1:999, 1)
  # Generate a random street name
  street_name <- paste0(sample(LETTERS, 5, replace = TRUE), collapse = "")
  # Generate a random city
  city <- paste0(sample(letters, 6, replace = TRUE), collapse = "")
  # Generate a random state
  state <- sample(c("NY", "CA", , "FL", "IL"), 1)
  # Generate a random ZIP code
```

Message ChatGPT...

ChatGPT can make mistakes. Consider checking important information.