Ong Yeow Hwee, Glen

Singapore University of Social Science

ICT337: Big Data Computing in the Cloud

Prof. Hu He

November 07, 2023

**Question 1**

**(a)**

**(b)**

**Question 2**

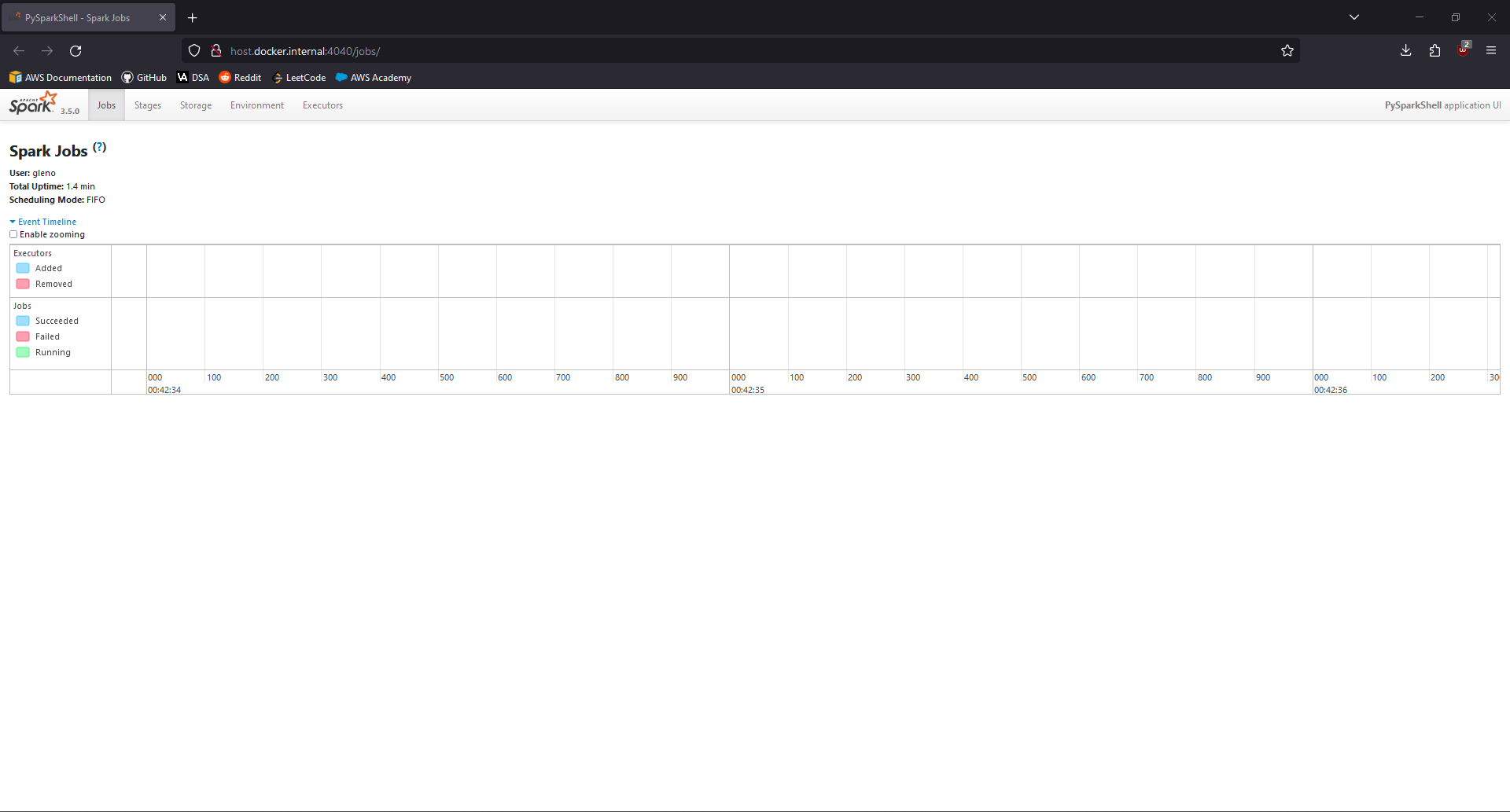
**Premise**

I have configured Hadoop, Java and Python in order to ensure that PySpark will work smoothly on my local machine and prepare it for future use. This framework will allow me to demonstration in my local environment and provide a detailed overview of the "Kmeans.py" script.

A screenshot of a computer

Description automatically generated

*Figure 1. Execution of PySpark in Windows PowerShell*



*Figure 2. Spark Web User Interface*

**Dependencies**

In the ‘kmeans.py’ example program, it begins by importing system-specific parameters and functions with ‘import sys’ This allows it to interact with the command prompt by reading and processing information from the command line. For example, it uses command line arguments such as the 'len(sys.argv)', ‘sys.stderr’, ‘sys.exit’, ‘sys.argv[1]’, ‘sys.argv[2]’ and ‘sys.argv[3]’.

Additionally, the ‘List’ type hint from the typing module is also imported by the 'from typing import List'. It is employed for annotating function arguments and variable types. The 'closestPoint' function in the 'kmeans.py' example program requires a list of NumPy arrays as its ‘centers’ parameter.

Moreover, to effectively handle numerical data, perform mathematical operations, and work with arrays, the NumPy library is also imported using the 'import numpy as np' command. It is used in the 'kmeans.py' example program for data parsing and transformation, distance calculations, and array manipulation.

Furthermore, the SparkSession class is imported from the pyspark.sql module by the line of code ‘from pyspark.sql import SparkSession’. This serves as an API or entry point for configuring, creating, and utilizing Spark functionalities in the "kmeans.py" example program.

**Initialization**

First, the function ‘parseVector()’ is defined to convert a string of numbers into an array of floating-point values. The line of code ‘parseVector(line: str) -> np.ndarray’ takes in a string of numbers separated by spaces. Within this define function, the ‘return np.array([float(x) for x in line.split(' ')])’ list comprehension splits it into a list of individual substrings by space, converts each substring into a floating-point number using the float() and returning the results as an array.

Second, the function 'closestPoint()' function determines the index of the closest centroid by computing the squared Euclidean distance between a data point and a group of cluster centroids. A NumPy array representing a data point (p) and a list of NumPy arrays representing cluster centroids (centers) are needed to call the function. It sets the closest centroid's index (bestIndex) to 0 and the closest distance (closest) at positive infinity. Subsequently, it calculates the squared Euclidean distance of each cluster centroid iteratively, updating the closest and best indexes upon finding a closer centroid. The function ultimately returns the index of the closest centroid.

Third, to prevent the 'kmeans.py' example program from being accidentally executed, the 'if \_name\_ = "\_\_main\_\_"' line of code checks to see if the script is being executed as the main program.

Fourth, the condition 'if len(sys.argv)!= 4:' within the code block 'if name == "main":' validates the existence of exactly four command-line arguments. The application warns the user and ends if this condition is not satisfied.

Finally, the 'SparkSession' class and '.builder' method are called to create and configure a spark instance, respectively, within the 'if \_name\_ = "\_\_main\_\_":' block of code. Furthermore, the ".appName("PythonKMeans")" method gives the "PythonKMeans" identifier to the spark job's application name. In addition, the '.getOrCreate()' method makes sure that a SparkSession is available for the 'kmeans.py' example program by determining whether one already exists and creating one in case it doesn't.

**Main Application Logic**

Prior to the execution of the K-means algorithm, the incoming data needs to be prepared (1), number of clusters needs to be assigned (2), converge distance needs to be assigned (3), random selection of data centroids (4) and the change in distance of the cluster centroid (5).

First, the line of code 'lines = spark.read.text(sys.argv[1]).map(lambda r: r[0])' is used before the K-means algorithm is executed. Data is loaded from the path given in the command line's first index. Next, this data is transformed from its original DataFrame format into an Resilient Distributed Dataset (RDD) of strings and stores it in the ‘line’ variable.

Subsequently, the line of code ‘data = lines.map(parseVector).cache()’ parses the RDD of strings stored in the ‘line’ variable into an array of floating-point values by calling the ‘parseVector()’ function defined earlier. The parsed data is then cached to optimize the ‘kmeans.py’ program’s performance and stores it in the ‘data’ variable. This parsed data would be used as input for the K-means algorithm.

Second, the line of code ‘k = int(sys.argv[2])’ converts the number given in the command line’s second index into an integer and stores it in the number of clusters variable (k). This variable would be used to determine the number of clusters for the K-means algorithm.

Third, the line of code ‘convergeDist = float(sys.argv[3])’ converts the number given in the command line’s third index into a float and stores it in the converge distance variable (convergeDist). This variable would be used to determine converge threshold for the iterative K-means algorithm.

Fourth, the line of code ‘kPoints = data.takeSample(False, K, 1)’ takes K samples out of the data pool without replacement with the seed of 1 to specify a starting point to generate the samples. This variable would be used as the initial cluster centroid for the iterative K-means algorithm.

Fifth, the line of code ‘tempDist = 1.0’ assigns an initial distance to 1.0. This variable monitors the distance change of the cluster centroid for the iterative K-means algorithm.

The code block inside the while-loop will be executed to start the K-means algorithm after the K-means parameters have been defined. The line of code ‘while tempDist > convergeDist:’ establishes the stopping criterion for the iterations. Whereby in the situation where the distance change of the cluster centroid is below the converge distance defined, the K-means algorithm is considered to have converged and stop iterating.

Subsequently, in each iteration of the K-means algorithm, the cluster centroid’s index closest to each data point is identified. The line of code ‘closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))’ maps each data point in the input RDD data by computing the closest cluster centroid of each data point based on the initial cluster centroid defined and stores the rdd in ‘(index of cluster centroid closest to data point, (data point, count))’ structure in the ‘closest’ variable.

Additionally, in the next line of code ‘pointStats = closest.reduceByKey(lambda p1\_c1, p2\_c2: (p1\_c1[0] + p2\_c2[0], p1\_c1[1] + p2\_c2[1]))’ within the K-means iteration (while-loop), it groups the datapoints by the identified closest clusters, adds the data points and adds the total count of the data points within each cluster. It then stores the aggregated RDD in ‘(index of cluster centroid closest to data point, (sum of data points, total count))’ structure in the ‘pointStats’ variable.

Moreover, in the following line of code ‘newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()’ within the k-means iteration (while-loop), it maps the aggregated RDD into ‘(index of cluster centroid closest to data point, average data point)’ structure and stores it into the ‘newPoints’ variable. It achieves this by dividing the sum of data points by the total data point.

Furthermore, in the following line of code ‘tempDist = sum(np.sum((kPoints[iK] - p) \*\* 2) for (iK, p) in newPoints)’, it uses the squared Euclidean distance to calculate the distance change between the current cluster centroid and the previous cluster centroid. The computed value is used to update distance change between the cluster centroids (tempDist) to determine if subsequent iterations of the K-means algorithm are required.

Finally, the cluster centroids (kPoints) are updated in the subsequent lines of code within the K-means iteration (while-loop). The algorithm ends by showing the user the final cluster centroids when the convergence criterion is satisfied, indicating that the change in cluster centroids is less than the given threshold.

**Demonstration**

To demonstrate the non-interactive execution of the build-in ‘kmeans.py’ example program, the Windows PowerShell is used and navigated into the Spark Hadoop Package directory with the ‘cd C:\apps\spark-3.5.0-bin-hadoop3’ command (see Figure 3). The ‘kmeans.py’ example program provided requires data input, number of clusters and converge distance in sequence. In this demonstration an example input ‘kmeans\_data\_mod.txt’, two clusters and a converge distance of 0.1 is used to test the ‘kmeans.py’ example program (see Figure 4). Therefore, the ‘./bin/spark-submit ./examples/src/main/python/kmeans.py 2 0.1’, would return the final cluster centroids of ‘Final centers: [array([0.1]), array([9.1])]’ (see Figure 5).

A screenshot of a computer

Description automatically generated

*Figure 3. Changing directory to the Spark Hadoop package file in Windows PowerShell.*

A screenshot of a computer program

Description automatically generated

*Figure 4. kmeans.py program execution with 2 clusters and 0.1 converge distance.*

A screenshot of a computer program

Description automatically generated

*Figure 5. kmeans.py program output with 2 clusters and 0.1 converge distance.*

**Question 3**

**Full PySpark Program**

import logging

import os

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, avg, min, max, when, split, lit, concat

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

VEHICLE\_MPG\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "vehicle\_mpg.tsv")

VEHICLE\_MANUFACTURERS\_DATA\_FILE\_PATH = os.path.join(

    DATA\_DIR, "vehicle\_manufacturers.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

# Configuration

numeric\_dtypes = ("int", "double", "float", "decimal")

mpg\_class\_config = {

    "low": {"max\_value": 20},

    "mid": {"min\_value": 20, "max\_value": 30},

    "high": {"min\_value": 30, "max\_value": 40},

    "very high": {"min\_value": 40}

}

def configure\_logging():

    """

    Configure logging settings and return a logger object.

    Returns

    -------

    logger : object

        Logger object for logging messages.

    Notes

    -----

    This function returns a logger object that may be used to log messages

    inside the program and initializes the logging parameters, including the logging level.

    The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    logging.basicConfig(*level*=LOGGING\_LEVEL)

    return logging.getLogger(\_\_name\_\_)

def create\_spark\_session(*app\_name*="TMA\_Data\_Analysis"):

    """

    Create and return a Spark session.

    Parameters

    ----------

    app\_name : str, optional

        The name of the Spark application, by default "TMA\_Data\_Analysis".

    Returns

    -------

    SparkSession

        The Spark session object.

    Notes

    -----

    This function initializes a Spark session, which is the entry point for working with Spark functionality.

    """

    return SparkSession.builder.appName(app\_name)\

        .config("spark.some.config.option", "some-value")\

        .getOrCreate()

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    """

    Show rows of a DataFrame with the option to limit the number of rows displayed.

    Parameters

    ----------

    df : DataFrame

        The DataFrame to be displayed.

    max\_rows : int, optional

        The maximum number of rows to display. Default is 100.

    show\_rows : int, optional

        The rows to display if records is above max rows. Default is 20.

    Returns

    -------

    None

    Notes

    -----

    This function shows the rows from the DataFrame input.

    The DataFrame will only display the first "show\_rows" rows if there are more rows than the specified "max\_rows".

    The DataFrame will display all available rows without truncation if the number of rows is less than "max\_rows".

    """

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def shape(*df*, *logger*):

    """

    Display the shape of a DataFrame.

    Parameters

    ----------

    df : DataFrame

        The DataFrame to be analyzed.

    logger : Logger

        Logger object for logging messages.

    Returns

    -------

    None

    Notes

    -----

    This function calculates the number of rows and columns in the input DataFrame.

    The 'logger' object is used for logging and displaying the total number of rows and columns.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    num\_rows = df.count()

    num\_columns = len(df.columns)

    logger.info(

        f"Number of Rows: {num\_rows}, Number of Columns: {num\_columns}")

def load\_data(*spark*, *logger*, *file\_path*, *delimiter*=","):

    """

    Load data from CSV file into a Spark DataFrame.

    Parameters

    ----------

    spark : SparkSession

        The Spark session.

    logger : Logger

        Logger object for logging messages.

    file\_path : str, optional

        The path to the CSV file to load.

    delimiter : str, optional

        The delimiter used in the CSV file. Default is ",".

    Returns

    -------

    DataFrame

        DataFrame containing the data.

    Raises

    ------

    FileNotFoundError

        If the specified file does not exist.

    Notes

    -----

    This function reads data from a CSV file and loads it into a Spark DataFrame.

    The data is assumed to be in CSV format, and the default delimiter is a comma (','), which can be customized using the 'delimiter' parameter.

    The 'logger' object is used for logging messages, and any error that occurs during the data loading process is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").option("delimiter", delimiter).csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def process\_missing\_data(*loaded\_df*, *logger*):

    """Process and analyze the loaded data.

    Parameters

    ----------

    loaded\_df : DataFrame

        DataFrame containing the data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the cleansed data.

    Raises

    ------

    Exception

        If an error occurs during data processing.

    Notes

    -----

    This function determines which rows in the supplied DataFrame have missing values,

    displays those rows, and removes them from the DataFrame. It details the total number of missing values,

    the cleaned DataFrame that is produced, and any problems that may have occurred.

    The 'logger' object is used for logging messages, and any error that occurs during the processing of missing data is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        columns\_to\_check = loaded\_df.columns

        filter\_condition = None

        # Loop through the list of columns and build a filter condition to check for null values.

        for column\_name in columns\_to\_check:

            if filter\_condition is None:

                filter\_condition = col(column\_name).isNull()

            else:

                filter\_condition = filter\_condition | col(column\_name).isNull()

        missing\_data\_df = loaded\_df.filter(filter\_condition)

        logger.info("Sample rows in the DataFrame with Missing Value:")

        show\_dataframe(missing\_data\_df)

        missing\_occurrence = missing\_data\_df.count()

        logger.info(

            f"There are {missing\_occurrence} rows with missing values in DataFrame.\n")

        clean\_data\_df = loaded\_df.filter(~filter\_condition)

        return clean\_data\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def numeric\_summary(*cleansed\_df*, *numeric\_dtypes*, *logger*):

    """

    Generate summary statistics for numeric columns in a DataFrame.

    Parameters

    ----------

    cleansed\_df : DataFrame

        The input DataFrame containing the cleansed data.

    numeric\_dtypes : str or list

        The data types associated with numeric columns, e.g., "int," "double," "float," or "decimal."

        You can also provide a list of data types.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame containing summary statistics for the numeric columns.

    Raises

    ------

    Exception

        If an error occurs while generating the summary statistics.

    Notes

    -----

    This function computes summary statistics for columns with data types typically associated with numeric values.

    It selects the numeric columns from the input DataFrame and computes summary statistics using the `summary()` method.

    The 'numeric\_dtypes' parameter specifies the data types that should be considered numeric for summary calculation.

    It can be a single data type or a list of data types.

    The 'logger' object is used for logging messages, and any error that occurs during the computation of summary is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        numeric\_columns = [

            col[0] for col in cleansed\_df.dtypes if col[1].startswith(numeric\_dtypes)]

        return cleansed\_df.select(numeric\_columns).summary()

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def count\_by(*cleansed\_df*, *grouped\_columns*, *logger*):

    """

    Count occurrences of rows by grouping columns.

    Parameters

    ----------

    cleansed\_df : DataFrame

        DataFrame containing the cleansed data.

    grouped\_columns : list

        List of columns to group by.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with counts, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function divides the data in the input DataFrame into groups according to the chosen columns,

    then counts the number of rows in each group. The outcome is a DataFrame with counts that is

    sorted in decreasing order according to the count values.

    The 'logger' object is used for logging messages, and any error that occurs during the count by process is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        count\_by\_col = cleansed\_df.groupby(\*grouped\_columns).count()

        sorted\_counts\_df = count\_by\_col.orderBy("count", *ascending*=False)

        return sorted\_counts\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def custom\_split(*cleansed\_df*, *column\_to\_split*, *seperator*, *value\_position*, *new\_column\_name*, *logger*):

    """

    Split a DataFrame column based on a separator and create a new column with the selected value position.

    Parameters

    ----------

    cleansed\_df : DataFrame

        The input DataFrame containing the cleansed data.

    column\_to\_split : str

        The name of the column to split.

    separator : str

        The separator used to split the column values.

    value\_position : int

        The position of the value to select after splitting (1-based index).

    new\_column\_name : str

        The name of the new column to store the selected values.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A new DataFrame with the added column containing the selected values.

    Raises

    ------

    Exception

        If an error occurs during the split and column creation.

    Notes

    -----

    This function takes an input DataFrame and splits a specified column using a given separator.

    It then creates a new column containing the selected value at the specified position after splitting.

    The resulting DataFrame includes the new column and the original data.

    The 'logger' object is used for logging messages, and any error that occurs during the custom split is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        clean\_data\_df = cleansed\_df.withColumn(new\_column\_name, split(

            col(column\_to\_split), seperator)[value\_position-1])

        return (clean\_data\_df)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def column\_class(*cleansed\_df*, *numeric\_column*, *new\_column\_name*, *config*: *dict*, *logger*):

    """

    Classify numeric values in a DataFrame based on a provided configuration and add the result as a new column.

    Parameters:

    -----------

    cleansed\_df : DataFrame

        The input DataFrame containing the cleansed data.

    numeric\_column : str

        The name of the numeric column in the DataFrame to be classified.

    new\_column\_name : str

        The name of the new column to store the classification results.

    config : dict

        A dictionary that defines the classification criteria.

        It should be in the format:

        {

            "class1": {"min\_value": min1, "max\_value": max1},

            "class2": {"min\_value": min2, "max\_value": max2},

            ...

        }

        where "min\_value" and "max\_value" define the range for each class.

    logger : object

        Logger object for logging messages.

    Returns:

    --------

    DataFrame

        A DataFrame with the new classification column added.

    Notes:

    ------

    This function takes a DataFrame, a numeric column to classify, a new column name to store the classification results,

    a configuration dictionary specifying the classification ranges, and a logger for error logging.

    It classifies the values in the specified numeric column based on the configuration and adds the classification results

    to the DataFrame as a new column.

    The 'logger' object is used for logging messages, and any error that occurs during the classification is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        classification = when(col(numeric\_column) <=

                              config["low"]["max\_value"], "low")

        for class\_name, class\_config in config.items():

            if class\_name != "unknown":

                min\_value = class\_config.get("min\_value", *float*("-inf"))

                max\_value = class\_config.get("max\_value", *float*("inf"))

                classification = classification.when(

                    (col(numeric\_column) > min\_value) & (

                        col(numeric\_column) <= max\_value), class\_name

                )

        return cleansed\_df.withColumn(new\_column\_name, classification)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def numeric\_stats(*cleansed\_df*, *group\_by\_column*, *numeric\_column*, *logger*):

    """

    Compute statistics for a numeric column grouped by another column.

    Parameters

    ----------

    cleansed\_df : DataFrame

        The input DataFrame containing the cleansed data.

    group\_by\_column : str

        The name of the column to group by.

    numeric\_column : str

        The name of the numeric column to compute statistics for.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame containing statistics (average, minimum, and maximum) for the numeric column

        grouped by the specified column, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates statistics (average, minimum, maximum) for a specified numeric column in the DataFrame.

    The statistics are computed based on groups formed by the values in the specified 'group\_by\_column.'

    The resulting DataFrame is ordered in descending order of the average of the numeric column.

    The 'logger' object is used for logging messages, and any error that occurs during the calculation of mathematical functions is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        col\_stats = cleansed\_df.groupBy(group\_by\_column).agg(

            avg(numeric\_column).alias(f"average\_{numeric\_column}"),

            min(numeric\_column).alias(f"min\_{numeric\_column}"),

            max(numeric\_column).alias(f"max\_{numeric\_column}")

        ).orderBy(f"average\_{numeric\_column}", *ascending*=False)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def numeric\_stats\_sql(*cleansed\_df*, *group\_by\_column*, *numeric\_column*, *spark*, *logger*):

    """

    Compute statistics for a numeric column grouped by another column using PySpark SQL.

    Parameters

    ----------

    cleansed\_df : DataFrame

        The input DataFrame containing the cleansed data.

    group\_by\_column : str

        The name of the column to group by.

    numeric\_column : str

        The name of the numeric column to compute statistics for.

    spark : SparkSession

        The Spark session.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame containing statistics (average, minimum, and maximum) for the numeric column

        grouped by the specified column, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates statistics (average, minimum, maximum) for a specified numeric column in the DataFrame.

    The statistics are computed based on groups formed by the values in the specified 'group\_by\_column.'

    The resulting DataFrame is ordered in descending order of the average of the numeric column.

    The 'logger' object is used for logging messages, and any error that occurs during the calculation of mathematical functions is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        cleansed\_df.createOrReplaceTempView("data")

        query = f"""

            SELECT {group\_by\_column},

                AVG({numeric\_column}) AS average\_{numeric\_column},

                MIN({numeric\_column}) AS min\_{numeric\_column},

                MAX({numeric\_column}) AS max\_{numeric\_column}

            FROM data

            GROUP BY {group\_by\_column}

            ORDER BY average\_{numeric\_column} DESC

        """

        col\_stats = spark.sql(query)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_by(*cleansed\_df*, *grouped\_columns*, *measurement*, *logger*):

    """

    Average occurrences of rows by grouping columns.

    Parameters

    ----------

    cleansed\_df : DataFrame

        DataFrame containing the cleansed data.

    grouped\_columns : list

        List of columns to group by.

    aggregate\_column : str

        Name of the column to aggregate.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with averages, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during averaging.

    Notes

    -----

    This function divides the data in the input DataFrame into groups according to the chosen columns,

    then averages the specified column in each group. The outcome is a DataFrame with averages that is

    sorted in decreasing order according to the average values.

    The 'logger' object is used for logging messages, and any error that occurs during the calculation of mathematical function is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        avg\_column = avg(col(measurement)).alias(f"avg\_{measurement}")

        avg\_by\_col = cleansed\_df.groupBy(grouped\_columns).agg(avg\_column)

        sorted\_average\_df = avg\_by\_col.orderBy(

            f"avg\_{measurement}", *ascending*=False)

        return sorted\_average\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_by\_sql(*cleansed\_df*, *grouped\_columns*, *measurement*, *spark*, *logger*):

    """

    Average occurrences of rows by grouping columns using PySpark SQL.

    Parameters

    ----------

    cleansed\_df : DataFrame

        DataFrame containing the cleansed data.

    grouped\_columns : list

        List of columns to group by.

    measurement : str

        Name of the column to aggregate.

    spark : SparkSession

        The Spark session.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with averages, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during averaging.

    Notes

    -----

    This function divides the data in the input DataFrame into groups according to the chosen columns,

    then averages the specified column in each group. The outcome is a DataFrame with averages that is

    sorted in decreasing order according to the average values.

    The 'logger' object is used for logging messages, and any error that occurs during the calculation of mathematical function is logged and raised as an exception.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        cleansed\_df.createOrReplaceTempView("data")

        query = f"""

            SELECT {",".join(grouped\_columns)},

                AVG({measurement}) AS avg\_{measurement}

            FROM data

            GROUP BY {",".join(grouped\_columns)}

            ORDER BY avg\_{measurement} DESC

        """

        avg\_by\_col = spark.sql(query)

        return avg\_by\_col

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def main():

    """Entry point of the script.

    Parameters

    ----------

    None

    Returns

    -------

    None

    Notes

    -----

    This function serves as the entry point of the script for processing flight data. It performs the following steps:

    1. Configures the logging settings and initializes a logger.

    2. Creates a Spark session for data processing.

    3. Loads flight data from a CSV file and processes it to handle missing values.

    4. Performs various data analyses.

    5. Displays and logs the analysis results.

    6. Stops the Spark session when processing is complete.

    """

    logger = configure\_logging()

    spark = create\_spark\_session()

    try:

        # Load tab delimited file

        vehicle\_mpg\_data\_frame = load\_data(

            spark, logger, VEHICLE\_MPG\_DATA\_FILE\_PATH, "\t")

        logger.info("Sample rows in the vehicle DataFrame:")

        show\_dataframe(vehicle\_mpg\_data\_frame)

        occurrence = vehicle\_mpg\_data\_frame.count()

        logger.info(f"There are {occurrence} occurrences in the DataFrame.\n")

        logger.info(vehicle\_mpg\_data\_frame.schema)

        shape(vehicle\_mpg\_data\_frame, logger)

        clean\_vehicle\_mpg\_df\_01 = process\_missing\_data(

            vehicle\_mpg\_data\_frame, logger)

        logger.info("Sample rows in the cleaned vehicle DataFrame:")

        show\_dataframe(clean\_vehicle\_mpg\_df\_01)

        clean\_occurrence = clean\_vehicle\_mpg\_df\_01.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

        stats\_num\_columns = numeric\_summary(

            clean\_vehicle\_mpg\_df\_01, numeric\_dtypes, logger)

        show\_dataframe(stats\_num\_columns)

        clean\_vehicle\_mpg\_df\_02 = custom\_split(

            clean\_vehicle\_mpg\_df\_01, "carname", " ", 1, "manufacturer", logger)

        show\_dataframe(clean\_vehicle\_mpg\_df\_02)

        manufacturer\_occurrence = count\_by(

            clean\_vehicle\_mpg\_df\_02, ["manufacturer"], logger)

        show\_dataframe(manufacturer\_occurrence)

        clean\_vehicle\_mpg\_df\_03 = clean\_vehicle\_mpg\_df\_02.withColumn(

            "modelyear", concat(lit("19"), col("modelyear")))

        show\_dataframe(clean\_vehicle\_mpg\_df\_03)

        model\_year\_occurrence = count\_by(

            clean\_vehicle\_mpg\_df\_03, ["modelyear"], logger)

        show\_dataframe(model\_year\_occurrence)

        clean\_vehicle\_mpg\_df\_mpg\_class = column\_class(

            clean\_vehicle\_mpg\_df\_03, "mpg", "mpg\_class", mpg\_class\_config, logger)

        show\_dataframe(clean\_vehicle\_mpg\_df\_mpg\_class)

        mpg\_class\_occurrence = count\_by(

            clean\_vehicle\_mpg\_df\_mpg\_class, ["mpg\_class"], logger)

        show\_dataframe(mpg\_class\_occurrence)

        vehicle\_manu\_data\_frame = load\_data(

            spark, logger, VEHICLE\_MANUFACTURERS\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the vehicle manufacturers DataFrame:")

        show\_dataframe(vehicle\_manu\_data\_frame)

        occurrence = vehicle\_manu\_data\_frame.count()

        logger.info(f"There are {occurrence} occurrences in the DataFrame.\n")

        vehicle\_full\_df = clean\_vehicle\_mpg\_df\_mpg\_class.join(

            vehicle\_manu\_data\_frame, *on*=["manufacturer"], *how*="inner")

        show\_dataframe(vehicle\_full\_df)

        occurrence = vehicle\_full\_df.count()

        logger.info(

            f"There are {occurrence} occurrences in the full vehicle DataFrame.\n")

        mpg\_stats\_by\_country = numeric\_stats(

            vehicle\_full\_df, "country", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_country)

        mpg\_stats\_by\_cylinders = numeric\_stats(

            vehicle\_full\_df, "cylinders", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_cylinders)

        mpg\_stats\_by\_modelyear = numeric\_stats(

            vehicle\_full\_df, "modelyear", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_modelyear)

        mpg\_stats\_by\_manufacturer = numeric\_stats(

            vehicle\_full\_df, "manufacturer", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_manufacturer)

        average\_mpg\_by\_carname\_manufacturer = average\_by(

            vehicle\_full\_df, ["carname", "manufacturer"], "mpg", logger)

        show\_dataframe(average\_mpg\_by\_carname\_manufacturer)

        mpg\_stats\_by\_country\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "country", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_country\_sql)

        mpg\_stats\_by\_cylinders\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "cylinders", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_cylinders\_sql)

        mpg\_stats\_by\_modelyear\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "modelyear", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_modelyear\_sql)

        mpg\_stats\_by\_manufacturer\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "manufacturer", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_manufacturer\_sql)

        average\_mpg\_by\_carname\_manufacturer\_sql = average\_by\_sql(

            vehicle\_full\_df, ["carname", "manufacturer"], "mpg", spark, logger)

        show\_dataframe(average\_mpg\_by\_carname\_manufacturer\_sql)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

    finally:

        if spark is not None:

            spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**(a)**

***Script Constants Snippet***

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

VEHICLE\_MPG\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "vehicle\_mpg.tsv")

VEHICLE\_MANUFACTURERS\_DATA\_FILE\_PATH = os.path.join(

    DATA\_DIR, "vehicle\_manufacturers.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def shape(*df*, *logger*):

    num\_rows = df.count()

    num\_columns = len(df.columns)

    logger.info(

        f"Number of Rows: {num\_rows}, Number of Columns: {num\_columns}")

def load\_data(*spark*, *logger*, *file\_path*, *delimiter*=","):

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").option("delimiter", delimiter).csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

***Script Main Function Snippet***

        # Load tab delimited file

        vehicle\_mpg\_data\_frame = load\_data(

            spark, logger, VEHICLE\_MPG\_DATA\_FILE\_PATH, "\t")

        logger.info("Sample rows in the vehicle DataFrame:")

        show\_dataframe(vehicle\_mpg\_data\_frame)

        occurrence = vehicle\_mpg\_data\_frame.count()

        logger.info(f"There are {occurrence} occurrences in the DataFrame.\n")

        logger.info(vehicle\_mpg\_data\_frame.schema)

        shape(vehicle\_mpg\_data\_frame, logger)

***Output Screenshot***

A building with many lights

Description automatically generated

*Figure 6. Content, number of occurrences and schema of vehicles mpg DataFrame.*

**(b)**

***Script Configuration Snippet***

# Configuration

numeric\_dtypes = ("int", "double", "float", "decimal")

mpg\_class\_config = {

    "low": {"max\_value": 20},

    "mid": {"min\_value": 20, "max\_value": 30},

    "high": {"min\_value": 30, "max\_value": 40},

    "very high": {"min\_value": 40}

}

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def process\_missing\_data(*loaded\_df*, *logger*):

    try:

        columns\_to\_check = loaded\_df.columns

        filter\_condition = None

        # Loop through the list of columns and build a filter condition to check for null values.

        for column\_name in columns\_to\_check:

            if filter\_condition is None:

                filter\_condition = col(column\_name).isNull()

            else:

                filter\_condition = filter\_condition | col(column\_name).isNull()

        missing\_data\_df = loaded\_df.filter(filter\_condition)

        logger.info("Sample rows in the DataFrame with Missing Value:")

        show\_dataframe(missing\_data\_df)

        missing\_occurrence = missing\_data\_df.count()

        logger.info(

            f"There are {missing\_occurrence} rows with missing values in DataFrame.\n")

        clean\_data\_df = loaded\_df.filter(~filter\_condition)

        return clean\_data\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def numeric\_summary(*cleansed\_df*, *numeric\_dtypes*, *logger*):

    try:

        numeric\_columns = [

            col[0] for col in cleansed\_df.dtypes if col[1].startswith(numeric\_dtypes)]

        return cleansed\_df.select(numeric\_columns).summary()

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        clean\_vehicle\_mpg\_df\_01 = process\_missing\_data(

            vehicle\_mpg\_data\_frame, logger)

        logger.info("Sample rows in the cleaned vehicle DataFrame:")

        show\_dataframe(clean\_vehicle\_mpg\_df\_01)

        clean\_occurrence = clean\_vehicle\_mpg\_df\_01.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

        stats\_num\_columns = numeric\_summary(clean\_vehicle\_mpg\_df\_01, logger)

        show\_dataframe(stats\_num\_columns)

***Output Screenshot***

A screenshot of a computer

Description automatically generated

*Figure 7. Content and number of occurrences of missing data.*

A screenshot of a computer

Description automatically generated

*Figure 8. Content and number of occurrences of the cleaned vehicle mpg DataFrame.*

A screen shot of a computer

Description automatically generated

*Figure 9. Statistics profile of numeric columns in the cleaned vehicle mpg DataFrame.*

**(c)**

***Script Configuration Snippet***

# Configuration

numeric\_dtypes = ("int", "double", "float", "decimal")

mpg\_class\_config = {

    "low": {"max\_value": 20},

    "mid": {"min\_value": 20, "max\_value": 30},

    "high": {"min\_value": 30, "max\_value": 40},

    "very high": {"min\_value": 40}

}

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def count\_by(*cleansed\_df*, *grouped\_columns*, *logger*):

    try:

        count\_by\_col = cleansed\_df.groupby(\*grouped\_columns).count()

        sorted\_counts\_df = count\_by\_col.orderBy("count", *ascending*=False)

        return sorted\_counts\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def custom\_split(*cleansed\_df*, *column\_to\_split*, *seperator*, *value\_position*, *new\_column\_name*, *logger*):

    try:

        clean\_data\_df = cleansed\_df.withColumn(new\_column\_name, split(

            col(column\_to\_split), seperator)[value\_position-1])

        return (clean\_data\_df)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def column\_class(*cleansed\_df*, *numeric\_column*, *new\_column\_name*, *config*: *dict*, *logger*):

    try:

        classification = when(col(numeric\_column) <=

                              config["low"]["max\_value"], "low")

        for class\_name, class\_config in config.items():

            if class\_name != "unknown":

                min\_value = class\_config.get("min\_value", *float*("-inf"))

                max\_value = class\_config.get("max\_value", *float*("inf"))

                classification = classification.when(

                    (col(numeric\_column) > min\_value) & (

                        col(numeric\_column) <= max\_value), class\_name

                )

        return cleansed\_df.withColumn(new\_column\_name, classification)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        clean\_vehicle\_mpg\_df\_02 = custom\_split(

            clean\_vehicle\_mpg\_df\_01, "carname", " ", 1, "manufacturer", logger)

        show\_dataframe(clean\_vehicle\_mpg\_df\_02)

        manufacturer\_occurrence = count\_by(

            clean\_vehicle\_mpg\_df\_02, ["manufacturer"], logger)

        show\_dataframe(manufacturer\_occurrence)

        clean\_vehicle\_mpg\_df\_03 = clean\_vehicle\_mpg\_df\_02.withColumn(

            "modelyear", concat(lit("19"), col("modelyear")))

        show\_dataframe(clean\_vehicle\_mpg\_df\_03)

        model\_year\_occurrence = count\_by(

            clean\_vehicle\_mpg\_df\_03, ["modelyear"], logger)

        show\_dataframe(model\_year\_occurrence)

        clean\_vehicle\_mpg\_df\_mpg\_class = column\_class(

            clean\_vehicle\_mpg\_df\_03, "mpg", "mpg\_class", mpg\_class\_config, logger)

        show\_dataframe(clean\_vehicle\_mpg\_df\_mpg\_class)

        mpg\_class\_occurrence = count\_by(

            clean\_vehicle\_mpg\_df\_mpg\_class, ["mpg\_class"], logger)

        show\_dataframe(mpg\_class\_occurrence)

***Output Screenshot***

A screenshot of a computer

Description automatically generated

*Figure 10. Content of the transformed vehicle mpg DataFrame with manufacturer.*

*A screenshot of a computer

Description automatically generated*

*Figure 11. Number of manufacturer occurrences in transformed vehicle mpg DataFrame.*

A screenshot of a computer

Description automatically generated

*Figure 12. Content of the transformed model year of vehicle mpg DataFrame.*

A screenshot of a computer screen

Description automatically generated

*Figure 13. Number of model year occurrences in transformed vehicle mpg DataFrame.*

A screenshot of a computer

Description automatically generated

*Figure 14. Content of the transformed vehicle mpg DataFrame with mpg class.*

A screenshot of a computer program

Description automatically generated

*Figure 15. Number of mpg class occurrences in transformed vehicle mpg DataFrame.*

**(d)**

***Script Constants Snippet***

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

VEHICLE\_MPG\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "vehicle\_mpg.tsv")

VEHICLE\_MANUFACTURERS\_DATA\_FILE\_PATH = os.path.join(

    DATA\_DIR, "vehicle\_manufacturers.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def load\_data(*spark*, *logger*, *file\_path*, *delimiter*=","):

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").option("delimiter", delimiter).csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

***Script Main Function Snippet***

        vehicle\_manu\_data\_frame = load\_data(

            spark, logger, VEHICLE\_MANUFACTURERS\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the vehicle manufacturers DataFrame:")

        show\_dataframe(vehicle\_manu\_data\_frame)

        occurrence = vehicle\_manu\_data\_frame.count()

        logger.info(f"There are {occurrence} occurrences in the DataFrame.\n")

        vehicle\_full\_df = clean\_vehicle\_mpg\_df\_mpg\_class.join(

            vehicle\_manu\_data\_frame, *on*=["manufacturer"], *how*="inner")

        show\_dataframe(vehicle\_full\_df)

        occurrence = vehicle\_full\_df.count()

        logger.info(

            f"There are {occurrence} occurrences in the full vehicle DataFrame.\n")

***Output Screenshot***

A screenshot of a computer program

Description automatically generated

*Figure 16. Content and number of occurrences of vehicle manufacturers DataFrame.*

A screenshot of a computer

Description automatically generated

*Figure 17. Content and number of occurrences of joined vehicles DataFrame.*

**(e)**

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def numeric\_stats(*cleansed\_df*, *group\_by\_column*, *numeric\_column*, *logger*):

    try:

        col\_stats = cleansed\_df.groupBy(group\_by\_column).agg(

            avg(numeric\_column).alias(f"average\_{numeric\_column}"),

            min(numeric\_column).alias(f"min\_{numeric\_column}"),

            max(numeric\_column).alias(f"max\_{numeric\_column}")

        ).orderBy(f"average\_{numeric\_column}", *ascending*=False)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_by(*cleansed\_df*, *grouped\_columns*, *measurement*, *logger*):

    try:

        avg\_column = avg(col(measurement)).alias(f"avg\_{measurement}")

        avg\_by\_col = cleansed\_df.groupBy(grouped\_columns).agg(avg\_column)

        sorted\_average\_df = avg\_by\_col.orderBy(

            f"avg\_{measurement}", *ascending*=False)

        return sorted\_average\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        mpg\_stats\_by\_country = numeric\_stats(

            vehicle\_full\_df, "country", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_country)

        mpg\_stats\_by\_cylinders = numeric\_stats(

            vehicle\_full\_df, "cylinders", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_cylinders)

        mpg\_stats\_by\_modelyear = numeric\_stats(

            vehicle\_full\_df, "modelyear", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_modelyear)

        mpg\_stats\_by\_manufacturer = numeric\_stats(

            vehicle\_full\_df, "manufacturer", "mpg", logger)

        show\_dataframe(mpg\_stats\_by\_manufacturer)

        average\_mpg\_by\_carname\_manufacturer = average\_by(

            vehicle\_full\_df, ["carname", "manufacturer"], "mpg", logger)

        show\_dataframe(average\_mpg\_by\_carname\_manufacturer)

***Output Screenshot***

A screen shot of a computer

Description automatically generated

*Figure 17. Average, minimum, and maximum mpg by country, sorted by average mpg.*

A screen shot of a computer

Description automatically generated

*Figure 18. Average, minimum, and maximum mpg by cylinders, sorted by average mpg.*

A screenshot of a computer program

Description automatically generated

*Figure 19. Average, minimum, and maximum mpg by model year, sorted by average mpg.*

A screenshot of a computer

Description automatically generated

*Figure 20. Average, minimum, and maximum mpg by manufacturer, sorted by average mpg.*

A screenshot of a computer

Description automatically generated

*Figure 21. Average mpg by car name and manufacturer, sorted by average mpg.*

**(f)**

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def numeric\_stats\_sql(*cleansed\_df*, *group\_by\_column*, *numeric\_column*, *spark*, *logger*):

    try:

        cleansed\_df.createOrReplaceTempView("data")

        query = f"""

            SELECT {group\_by\_column},

                AVG({numeric\_column}) AS average\_{numeric\_column},

                MIN({numeric\_column}) AS min\_{numeric\_column},

                MAX({numeric\_column}) AS max\_{numeric\_column}

            FROM data

            GROUP BY {group\_by\_column}

            ORDER BY average\_{numeric\_column} DESC

        """

        col\_stats = spark.sql(query)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_by\_sql(*cleansed\_df*, *grouped\_columns*, *measurement*, *spark*, *logger*):

    try:

        cleansed\_df.createOrReplaceTempView("data")

        query = f"""

            SELECT {",".join(grouped\_columns)},

                AVG({measurement}) AS avg\_{measurement}

            FROM data

            GROUP BY {",".join(grouped\_columns)}

            ORDER BY avg\_{measurement} DESC

        """

        avg\_by\_col = spark.sql(query)

        return avg\_by\_col

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        mpg\_stats\_by\_country\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "country", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_country\_sql)

        mpg\_stats\_by\_cylinders\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "cylinders", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_cylinders\_sql)

        mpg\_stats\_by\_modelyear\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "modelyear", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_modelyear\_sql)

        mpg\_stats\_by\_manufacturer\_sql = numeric\_stats\_sql(

            vehicle\_full\_df, "manufacturer", "mpg", spark, logger)

        show\_dataframe(mpg\_stats\_by\_manufacturer\_sql)

        average\_mpg\_by\_carname\_manufacturer\_sql = average\_by\_sql(

            vehicle\_full\_df, ["carname", "manufacturer"], "mpg", spark, logger)

        show\_dataframe(average\_mpg\_by\_carname\_manufacturer\_sql)

***Output Screenshot***

A screen shot of a computer

Description automatically generated

*Figure 22. Average, minimum, and maximum mpg by country, sorted by average mpg (SQL approach).*

A screenshot of a computer program

Description automatically generated

*Figure 23. Average, minimum, and maximum mpg by cylinders, sorted by average mpg (SQL approach).*

A screenshot of a computer program

Description automatically generated

*Figure 24. Average, minimum, and maximum mpg by model year, sorted by average mpg (SQL approach).*

A screenshot of a computer

Description automatically generated

*Figure 25. Average, minimum, and maximum mpg by manufacturer, sorted by average mpg (SQL approach).*

A screenshot of a computer

Description automatically generated

*Figure 26. Average mpg by car name and manufacturer, sorted by average mpg (SQL approach).*

**Question 4**

**Full PySpark Program**

**(a)**

***Script Constants Snippet***

***Script Functions Snippet***

***Script Main Function Snippet***

***Output Screenshot***

**(b)**

***Script Constants Snippet***

***Script Functions Snippet***

***Script Main Function Snippet***

***Output Screenshot***

**(c)**

***Script Constants Snippet***

***Script Functions Snippet***

***Script Main Function Snippet***

***Output Screenshot***

**(d)**

***Script Constants Snippet***

***Script Functions Snippet***

***Script Main Function Snippet***

***Output Screenshot***

**(e)**

***Script Constants Snippet***

***Script Functions Snippet***

***Script Main Function Snippet***

***Output Screenshot***

**(f)**

***Script Constants Snippet***

***Script Functions Snippet***

***Script Main Function Snippet***

***Output Screenshot***