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)**

|  |  |  |
| --- | --- | --- |
| Components | Apache Hadoop | Apache Spark |
| Performance | Iterative algorithms perform slower. data is processed using MapReduce and stored across various sources. | Faster because intermediate data is not read and written to disk; instead, processing is done in-memory using caching. |
| Cost | Uses any kind of disk storage for data processing, so it operates at a cheaper cost. | Operates more expensively because it uses large amounts of RAM to spin up nodes and relies on in-memory computations for real-time data processing. |
| Processing | Mostly for linear data processing and batch processing. | Suitable for handling real-time, unstructured data streams in real-time. |
| Processing Model | MapReduce-based. | Direct Acyclic Graph (DAG)-based. |
| Scalability | Uses the Hadoop Distributed File System to scale quickly to meet demand. | Depends on the fault-tolerant Hadoop File System to handle massive data sets. |
| Data Storage | Hadoop Distributed File System (HDFS). | Multiple storage system (HDFS, S3, local). |
| Security | Employs a variety of access control and authentication techniques. | Increases security by using shared secrets or event logging for authentication. |
| Machine Learning | Limited support for machine learning. | Adequate. support for external libraries and built-in machine learning libraries (MLlib). |
| Fault Tolerance | Replication (3x). | Lineage information for recovery. |
| Resource Management | Hadoop YARN. | Spark-build in cluster manager. |
| Ease of Use | Setup and configuration complexity. | Easier due to high level APIs. |
| APIs | Limited (MapReduce, Pig, Hive). | Rich APIs (RDDs, DataFrames and Datasets). |

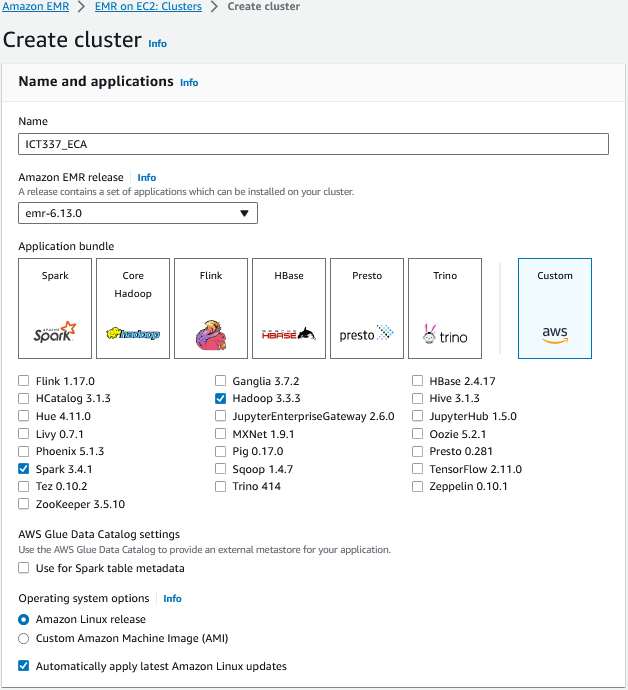
**(b)**

***Premise***

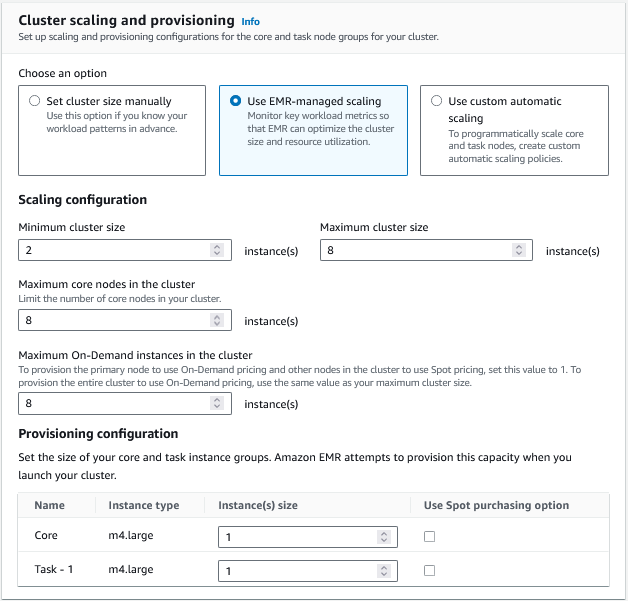
This explanation will focus on the Spark job execution process in a cluster computing setup within the context of cloud computing, using AWS EMR (Elastic MapReduce) as an example. A popular option for big data applications, Spark is a powerful open-source data processing framework that is carefully crafted for distributed computing. The steps that make up the Spark job execution process in this cluster computing environment will then be discussed in detail.

***Cluster Configuration***

Configuring the cluster is the first step. Users can launch a cluster with the desired number of instances, each with Spark and Hadoop pre-installed, in a cloud computing environment such as AWS EMR. Users can scale resources up or down as needed by configuring the cluster size to match the requirements for data processing. To ensure the smooth processing of the spark submit, the default configurations are used apart from the following to create an EMR cluster. It is crucial for users to select the spark and Hadoop application as well as creating a PEM key stored in the users local machine.



*Figure 1. Customized name and application for EMR cluster*

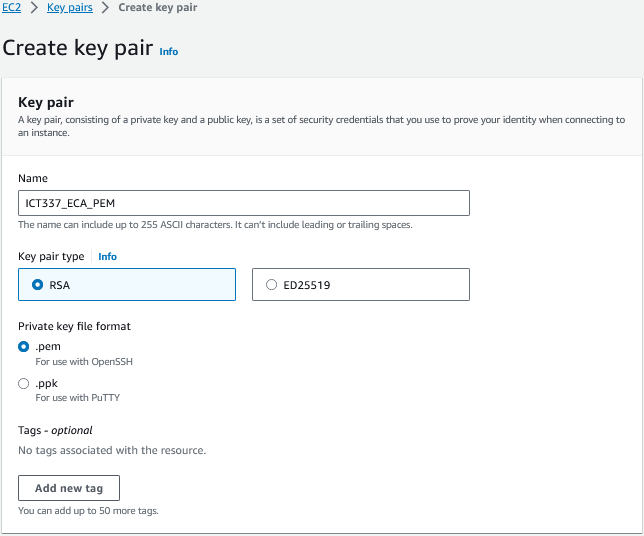


*Figure 2. Customized cluster scaling and provisioning*

A screenshot of a computer

Description automatically generated

*Figure 3. Customized cluster termination*

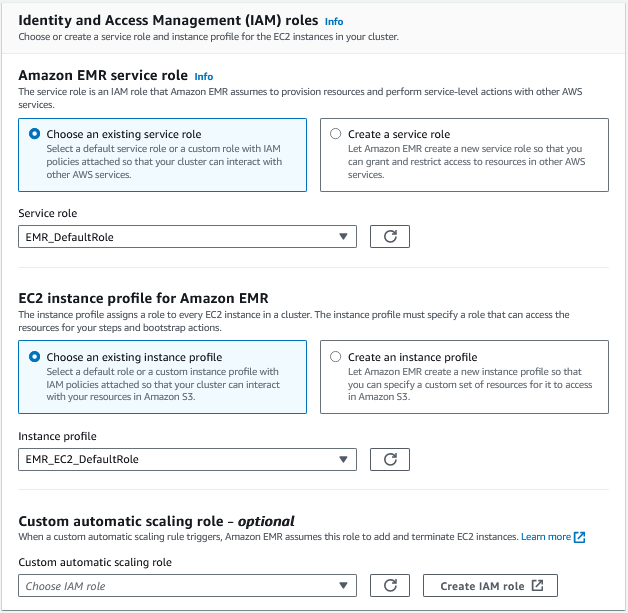


*Figure 4. Creation of PEM security key pair for spark-submit*

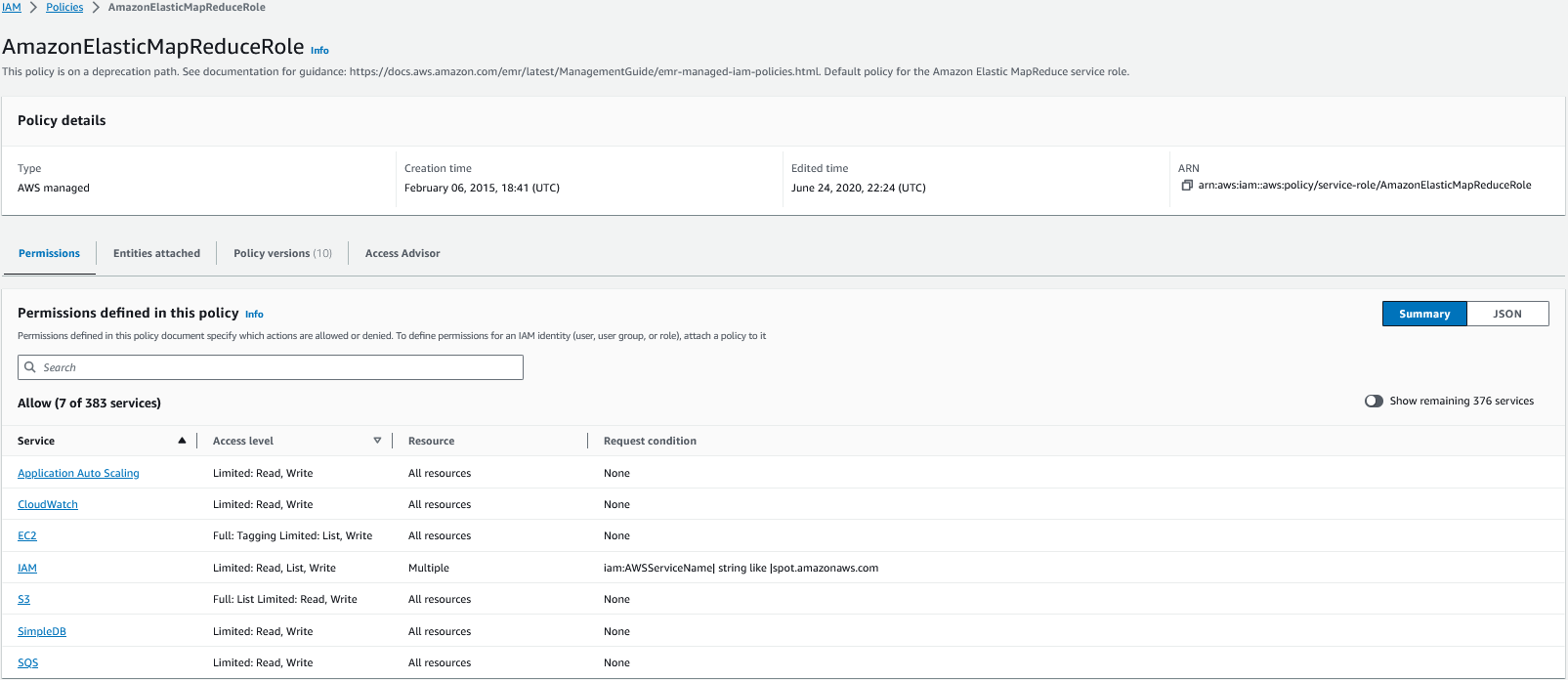
A screenshot of a computer program

Description automatically generated

*Figure 5. Customized Security configuration*



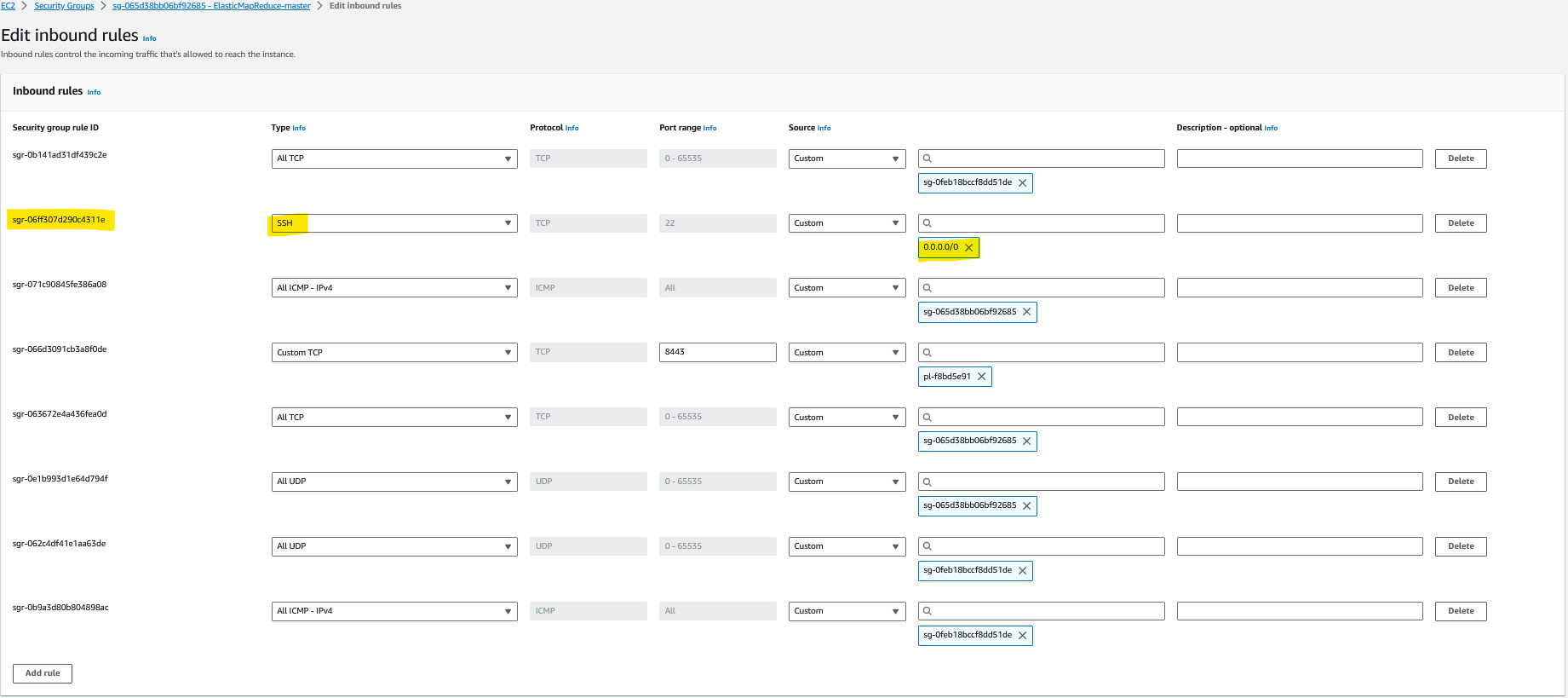
*Figure 6. Selection of EMR\_EC2\_DefaultRole*



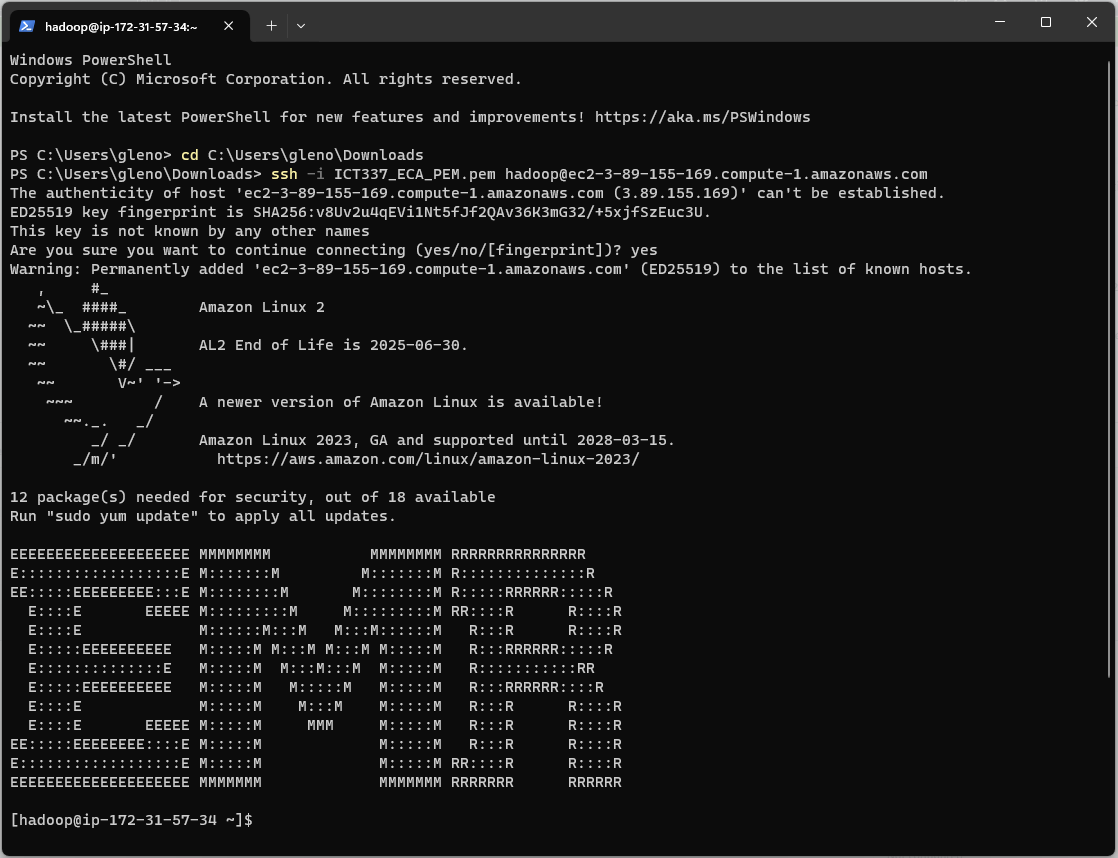
*Figure 7. Policy attached to EMR\_EC2\_DefaultRole*

***Application Submission***

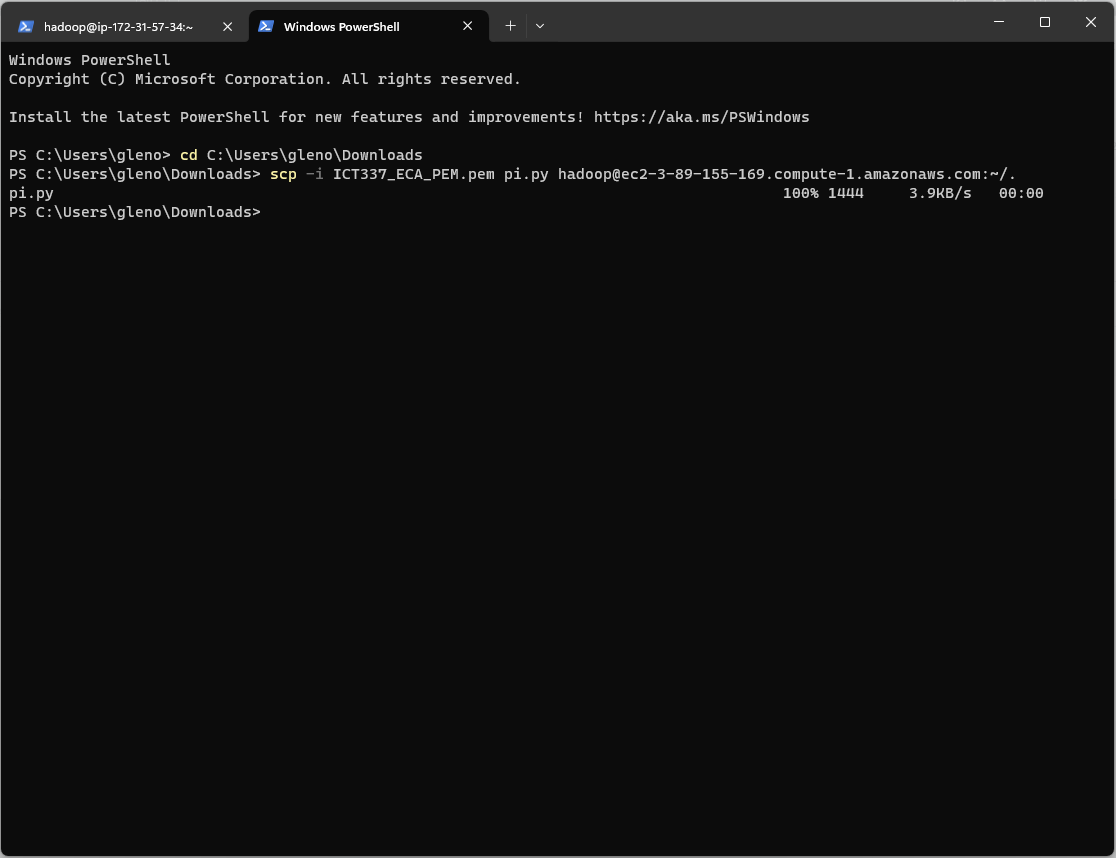
Users can submit their Spark application, which includes the code and dependencies, once the cluster is ready. The spark-submit command, which specifies the main class and any application-specific configuration options, is typically used to submit applications. To perform a spark-submit, users have to access the EMR cluster using the respective SSH command (see Figure 9), upload the PySpark application into the EMR master node (see Figure 10) before running the application (see Figure 11). To ensure the smooth running of the spark-submit, it is crucial for user to whitelist the EMR cluster of the EMR master node in the firewall configuration.



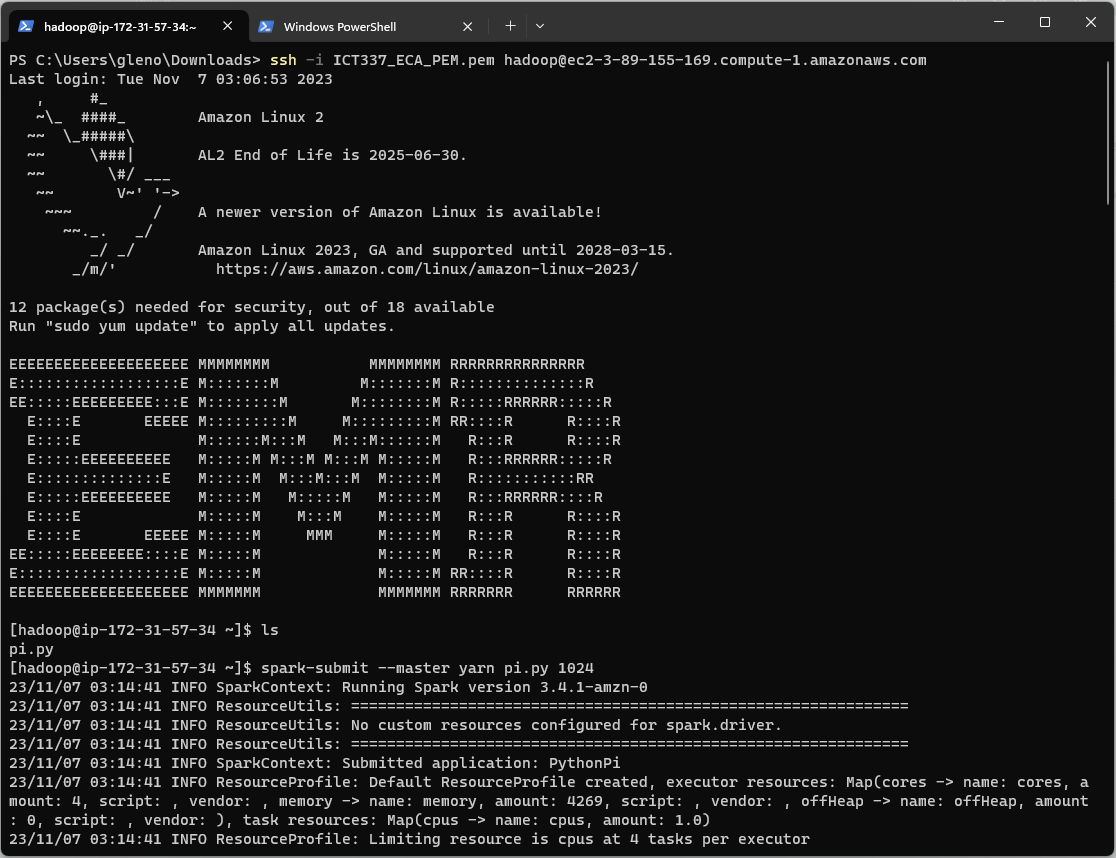
*Figure 8. Whitelist EMR cluster master node inbound rule*



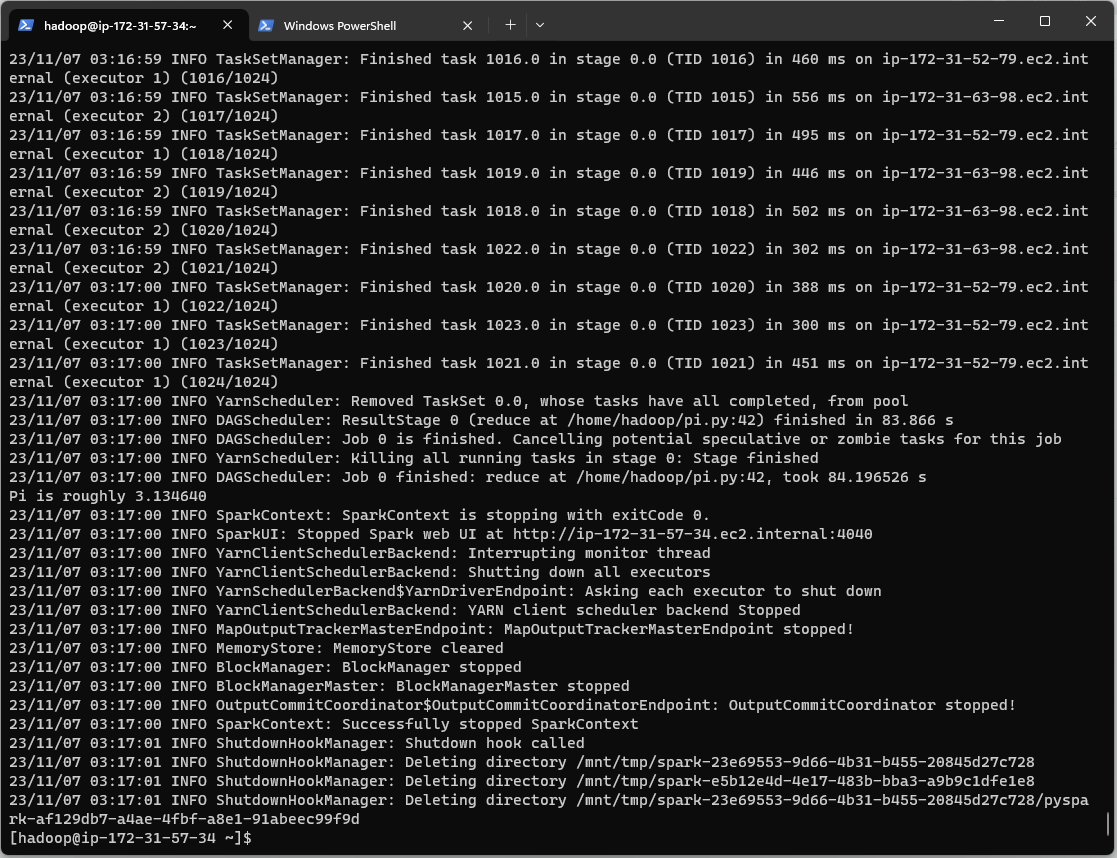
*Figure 9. Connect to primary node using SSH*



*Figure 10. Upload PySpark application into EMR cluster master node*



*Figure 11. spark-submit PySpark application in the EMR cluster master node*



*Figure 12. PySpark application output*

***Job Execution***

Spark jobs are organized into stages, which are the core work units in Spark's model of execution. Every step consists of a collection of concurrently executable data transformations. Tasks are the smallest units of parallelism and are separated into stages. Spark schedules these steps using a directed acyclic graph (DAG) scheduler.

A screenshot of a computer

Description automatically generated

*Figure 13. Spark executors’ history*

Task ***Scheduling***

To request resources and schedule tasks, the main application, Spark driver, interacts with the cluster manager (such as YARN in Hadoop or the EMR cluster manager in AWS). Next, the tasks are divided among the cluster's worker nodes. Every task processes a subset of the data, and Spark uses in-memory processing to expedite computation and data access.

***Data Shuffling***

Certain Spark operations, like groupBy or reduceByKey, might require worker nodes to exchange and shuffle data. This calls for the data to be transferred over a network, which can be a resource-intensive process. Spark streamlines this procedure to reduce data shuffles and boost efficiency.

***Fault Tolerance***

Fault tolerance mechanisms are provided by Spark to manage cluster failures. Spark can move tasks to other functional nodes to keep the job moving forward if a task or a node fails. To retrieve lost data and recalculate required transformations, checkpoints and lineage information are utilized.

***Job Completion***

The Spark driver gathers the finished output as soon as the tasks are finished and the results are combined. Depending on the needs of their application, users can store the results in databases, HDFS, Amazon S3, or other storage systems.

***Cluster Termination***

Users can terminate the cluster to stop paying after the task is completed. Users can simply scale and manage their cluster resources according to their workload requirements with cloud providers like AWS EMR.

A screenshot of a computer

Description automatically generated

*Figure 14. EMR cluster termination*

**Question 2**

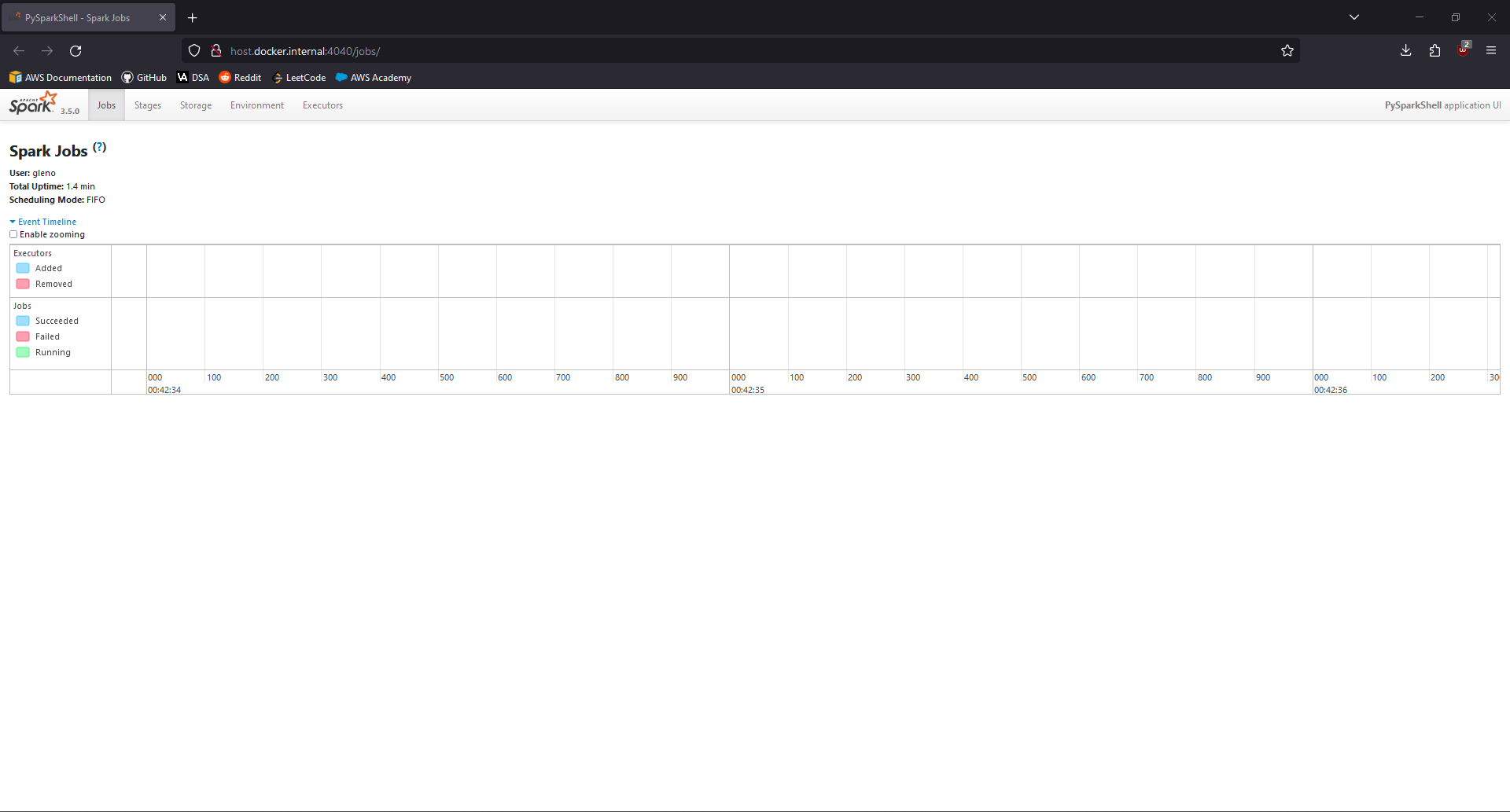
**Premise**

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 15. Execution of PySpark in Windows PowerShell*



*Figure 16. 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, 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 17). 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 18). 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 19).

A screenshot of a computer

Description automatically generated

*Figure 17. Changing directory to the Spark Hadoop package file in Windows PowerShell.*

A screenshot of a computer program

Description automatically generated

*Figure 18. kmeans.py program execution with 2 clusters and 0.1 converge distance.*

A screenshot of a computer program

Description automatically generated

*Figure 19. 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 20. 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 21. Content and number of occurrences of missing data.*

A screenshot of a computer

Description automatically generated

*Figure 22. Content and number of occurrences of the cleaned vehicle mpg DataFrame.*

A screen shot of a computer

Description automatically generated

*Figure 23. 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 24. Content of the transformed vehicle mpg DataFrame with manufacturer.*

*A screenshot of a computer

Description automatically generated*

*Figure 25. Number of manufacturer occurrences in transformed vehicle mpg DataFrame.*

A screenshot of a computer

Description automatically generated

*Figure 26. Content of the transformed model year of vehicle mpg DataFrame.*

A screenshot of a computer screen

Description automatically generated

*Figure 27. Number of model year occurrences in transformed vehicle mpg DataFrame.*

A screenshot of a computer

Description automatically generated

*Figure 28. Content of the transformed vehicle mpg DataFrame with mpg class.*

A screenshot of a computer program

Description automatically generated

*Figure 29. 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 30. Content and number of occurrences of vehicle manufacturers DataFrame.*

A screenshot of a computer

Description automatically generated

*Figure 31. 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 32. Average, minimum, and maximum mpg by country, sorted by average mpg.*

A screen shot of a computer

Description automatically generated

*Figure 33. Average, minimum, and maximum mpg by cylinders, sorted by average mpg.*

A screenshot of a computer program

Description automatically generated

*Figure 34. Average, minimum, and maximum mpg by model year, sorted by average mpg.*

A screenshot of a computer

Description automatically generated

*Figure 35. Average, minimum, and maximum mpg by manufacturer, sorted by average mpg.*

A screenshot of a computer

Description automatically generated

*Figure 36. 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 37. Average, minimum, and maximum mpg by country, sorted by average mpg (SQL approach).*

A screenshot of a computer program

Description automatically generated

*Figure 38. Average, minimum, and maximum mpg by cylinders, sorted by average mpg (SQL approach).*

A screenshot of a computer program

Description automatically generated

*Figure 39. Average, minimum, and maximum mpg by model year, sorted by average mpg (SQL approach).*

A screenshot of a computer

Description automatically generated

*Figure 40. Average, minimum, and maximum mpg by manufacturer, sorted by average mpg (SQL approach).*

A screenshot of a computer

Description automatically generated

*Figure 41. Average mpg by car name and manufacturer, sorted by average mpg (SQL approach).*

**Question 4**

**Full PySpark Program**

import logging

import os

from pyspark import SparkContext

from pyspark.mllib.recommendation import ALS

# Constants

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

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

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

MOVIE\_OCCUPATION\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_occupation.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

LOGGING\_LEVEL = logging.INFO

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

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

# Configurations

age\_groups = {(0, 6): "[0-6]", (7, 12): "(6-12]", (13, 18): "(12-18]",

              (19, 30): "(18-30]", (31, 50): "(30-50]", (51, *float*("inf")): "50+"}

summer = ["may", "jun", "jul"]

new\_user\_profiles = [

    [0, 50, 5, 881250949],

    [0, 172, 5, 881250949],

    [0, 181, 5, 881250949]

]

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.

    Note: 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\_context(*app\_name*="ECA\_Alternating\_Least\_Squares"):

    """

    Create and return a SparkContext.

    Parameters

    ----------

    app\_name : str, optional

        The name of the Spark application. Default is "ECA\_Alternating\_Least\_Squares".

    Returns

    -------

    SparkContext

        An instance of SparkContext.

    Notes

    -----

    This function initializes a SparkContext, which is the entry point for Spark operations.

    """

    sc = SparkContext("local", app\_name)

    return sc

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    """

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

    Parameters

    ----------

    df : rdd

        The rdd 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 displays rows of the input rdd. If the rdd contains more

    rows than the specified `max\_rows`, it will limit the display to the first `show\_rows`

    rows. If the rdd has fewer rows than `max\_rows`, it will display all available

    rows without truncation.

    """

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

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

    """

    Load data from a text file into an RDD using SparkContext.

    Parameters

    ----------

    sc : SparkContext

        The SparkContext for creating RDDs.

    logger : object

        Logger object for logging messages.

    file\_path : str

        The path to the text file to load.

    delimiter : str, optional

        The delimiter used to split data in the text file. Default is a comma (',').

    Returns

    -------

    RDD

        An RDD containing the loaded data.

    Raises

    ------

    FileNotFoundError

        If the specified file does not exist.

    Exception

        If an error occurs while loading the data.

    Notes

    -----

    This function uses the SparkContext 'sc' to load data from a text file into an RDD.

    The 'delimiter' parameter specifies how the data in the text file is split into columns.

    The 'logger' object is used for logging messages, and any error that occurs during the 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:

        rdd = sc.textFile(file\_path)

        rdd = rdd.map(lambda *line*: line.split(delimiter))

        return rdd

    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 count\_unique\_by(*loaded\_rdd*, *column\_index*, *logger*):

    """

    Count the number of unique values in an RDD based on a specified column index.

    Parameters

    ----------

    loaded\_rdd : RDD

        The input RDD containing data.

    column\_index : int

        The index of the column to consider for counting unique values.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    int

        The count of unique values in the specified column.

    Raises

    ------

    Exception

        If an error occurs while counting unique values.

    Notes

    -----

    This function processes an RDD to count the number of unique values in a specified column.

    The 'column\_index' parameter determines which column to consider, and uniqueness is determined based on distinct values.

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

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

    """

    try:

        unique\_reviewers\_count = loaded\_rdd.map(

            lambda *line*: line[column\_index-1]).distinct().count()

        return unique\_reviewers\_count

    except *Exception* as e:

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

        raise e

def top\_n\_counts\_by(*loaded\_rdd*, *column\_position*, *n*, *logger*):

    """

    Get the top 'n' counts of values in an RDD based on a specified column position.

    Parameters

    ----------

    loaded\_rdd : RDD

        The input RDD containing data.

    column\_position : int

        The position of the column to consider for counting values.

    n : int

        The number of top counts to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list of (value, count) tuples

        A list of the top 'n' counts and their corresponding values, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs while retrieving the top 'n' counts.

    Notes

    -----

    This function processes an RDD to retrieve the top 'n' counts and their corresponding values in a specified column.

    The 'column\_position' parameter determines which column to consider, and the values are counted using the `countByValue()` method.

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

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

    """

    try:

        review\_count = loaded\_rdd.map(

            lambda *line*: line[column\_position-1]).countByValue()

        sorted\_review\_count = sorted(

            review\_count.items(), *key*=lambda *item*: item[1], *reverse*=True)

        return sorted\_review\_count[:n]

    except *Exception* as e:

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

        raise e

def date\_parse(*movie\_item\_rdd*, *column\_index*, *logger*):

    """

    Extracts the release years from a specified column of an RDD.

    Parameters

    ----------

    movie\_item\_rdd : RDD

        The input RDD containing movie data.

    column\_index : int

        The index of the column to extract release years from.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        An RDD containing the extracted movie release years.

    Raises

    ------

    Exception

        If an error occurs during processing, it is logged and raised.

    Notes

    -----

    This function processes the input movie RDD to extract the release years from a specified column.

    It extracts the last 4 digits from strings that end with a 4-digit year and returns an RDD of release years.

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

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

    """

    try:

        # Extract last 4 string from the date column that ends with a 4 digit year

        movie\_release\_years = movie\_item\_rdd.map(

            lambda *line*: line[column\_index][-4:])

        return movie\_release\_years

    except *Exception* as e:

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

        raise e

def count\_movie\_release\_years(*movie\_release\_years*, *logger*):

    """

    Counts and sorts movie release years from an RDD.

    Parameters

    ----------

    movie\_release\_years : RDD

        An RDD containing movie release years.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    dict

        A dictionary with movie release years as keys and their counts as values,

        sorted in descending order of counts.

    Raises

    ------

    Exception

        If an error occurs during counting and sorting, it is logged and raised.

    Notes

    -----

    This function takes an RDD containing movie release years, counts the occurrences of each release year,

    and returns a dictionary with release years as keys and their counts as values.

    The dictionary is sorted in descending order based on counts.

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

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

    """

    try:

        release\_year\_counts = movie\_release\_years.countByValue()

        sorted\_release\_year\_counts = *dict*(

            sorted(release\_year\_counts.items(), *key*=lambda *x*: x[1], *reverse*=True))

        return sorted\_release\_year\_counts

    except *Exception* as e:

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

        raise e

def movie\_release\_year\_range(*movie\_release\_years*, *logger*):

    """

    Calculate the range of movie release years from an RDD.

    Parameters

    ----------

    movie\_release\_years : RDD

        An RDD containing movie release years.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    None

        This function does not return a value, but it logs the range of movie release years.

    Raises

    ------

    Exception

        If an error occurs during the calculating, it is logged and raised.

    Notes

    -----

    This function takes an RDD containing movie release years, filters out any empty values, and calculates the

    range of release years (from the minimum to the maximum). It logs the calculated range using the provided 'logger' object.

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

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

    """

    try:

        movie\_release\_years = movie\_release\_years.filter(

            lambda *year*: year != '')

        min\_year = movie\_release\_years.min()

        max\_year = movie\_release\_years.max()

        logger.info(f"Range of Movie Release Years: {min\_year} to {max\_year}")

    except *Exception* as e:

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

        raise e

def extract\_genres(*movies\_with\_genre*, *mov\_genre\_rdd*, *logger*):

    """

    Extract and map movie genres in the RDD 'movies\_with\_genre'.

    Parameters

    ----------

    movies\_with\_genre : RDD

        An RDD containing movie records with genre indicators.

    mov\_genre\_rdd : RDD

        An RDD containing genre mapping data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        An RDD with movie records, where genres are mapped to their names.

    Raises

    ------

    Exception

        If an error occurs during extracting, it is logged and raised.

    Notes

    -----

    This function performs the following steps:

    1. Extracts genre mapping information from 'mov\_genre\_rdd' and creates a dictionary for mapping genre indicators to genre names.

    2. Extracts and maps movie genres in 'movies\_with\_genre' RDD. For each movie record in 'movies\_with\_genre', this step involves:

        a. Creating a list of genres by identifying '1' values in the record's genre indicator.

        b. Mapping these genre indicators to their corresponding genre names using the genre\_mapping dictionary.

        c. Associating the movie ID and its title with the list of genres.

    3. Returns the transformed RDD with genres mapped to their names.

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

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

    """

    try:

        genre\_mapping = mov\_genre\_rdd.map(

            lambda *genre*: (genre[1], genre[0])).collectAsMap()

        # Extract and map movie genres in movies\_with\_genre RDD

        movies\_with\_genre = movies\_with\_genre.map(lambda *record*: ([genre\_mapping[*str*(

            index)] for index, value in enumerate(record[1][1]) if value == "1"], record[0], record[1][0]))

        transformed\_movies\_with\_genre = movies\_with\_genre.map(

            lambda *x*: (x[1], (x[0], x[2])))

        return transformed\_movies\_with\_genre

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_genre(*movie\_genre\_with\_avg\_rating*, *genre\_rdd*, *n*, *genre\_output\_path*, *logger*):

    """

    Write the top 'n' reviewed movies for each genre to separate text files.

    Parameters

    ----------

    movie\_genre\_with\_avg\_rating : RDD

        An RDD containing movie records with genres and average ratings.

    genre\_rdd : RDD

        An RDD containing genre information.

    n : int

        The number of top reviewed movies to retrieve for each genre.

    genre\_output\_path : str

        The base path for saving genre-specific text files.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    None

    Raises

    ------

    Exception

        If an error occurs during writing, it is logged and raised.

    Notes

    -----

    This function processes movie data and genre information to write the top 'n' reviewed movies for each genre to separate text files.

    It follows these steps:

    1. Loop through each genre in the 'genre\_rdd'.

    2. Filter movie records by the current genre and create an RDD of movies specific to that genre.

    3. Sort the movies by the number of reviews in descending order.

    4. Select the top 'n' reviewed movies for the current genre.

    5. Sort the selected movies by their average rating in descending order.

    6. Generate a file path specific to the current genre using 'genre\_output\_path'.

    7. Save the sorted movies to a text file with the genre-specific file path.

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

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

    """

    try:

        # Loop through each genre

        for genre\_name, genre\_id in genre\_rdd.collect():

            # Filter movies by the current genre

            filtered\_movie\_genre\_with\_avg\_rating = movie\_genre\_with\_avg\_rating.filter(

                lambda *x*: genre\_name in x[0][0]).map(

                lambda *x*: (x[0][0], x[0][1], x[0][2], x[1], x[2]))

            # Sort movies by number of reviews in descending order

            sorted\_movie\_genre\_with\_avg\_rating = filtered\_movie\_genre\_with\_avg\_rating.sortBy(

                lambda *x*: x[3], *ascending*=False).map(lambda *x*: (x[0], x[1], x[2], x[4]))

            top\_n\_movie\_by\_genre = sorted\_movie\_genre\_with\_avg\_rating.zipWithIndex().filter(

                lambda *x*: x[1] < n).keys()

            # Sort movies by average rating in descending order

            sorted\_top\_n\_movie\_by\_genre = top\_n\_movie\_by\_genre.sortBy(

                lambda *x*: x[3], *ascending*=False)

            genre\_output\_path\_for\_genre = f"{genre\_output\_path}\{genre\_name}"

            sorted\_top\_n\_movie\_by\_genre.coalesce(

                1).saveAsTextFile(genre\_output\_path\_for\_genre)

    except *Exception* as e:

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

        raise e

def assign\_age\_group(*age*, *age\_groups*: *dict*, *logger*):

    """

    Assign an age group label based on the provided age and age group definitions.

    Parameters

    ----------

    age : int

        The age of the user to be categorized into an age group.

    age\_groups : dict

        A dictionary containing age group definitions as key-value pairs.

        The keys are tuples representing age range boundaries, and the values are the corresponding age group labels.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    str

        The age group label to which the given age belongs. Returns 'Unknown' if the age doesn't fit into any defined age group.

    Raises

    ------

    Exception

        If an error occurs during the assignment process, it is logged and raised.

    Notes

    -----

    This function categorizes a user's age into an age group based on the provided age group definitions. It follows these steps:

    1. Converts the 'age' parameter to an integer.

    2. Iterates through the 'age\_groups' dictionary to find a matching age range.

    3. If the age falls within a defined age range, the corresponding age group label is returned.

    4. If no matching age range is found, 'Unknown' is returned as the age group label.

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

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

    """

    try:

        age = *int*(age)

        for age\_range, group in age\_groups.items():

            if age\_range[0] <= age <= age\_range[1]:

                return group

        return 'Unknown'

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_age\_group(*movie\_names\_review\_with\_user*, *age\_groups*: *dict*, *n*, *age\_group\_output\_path*, *logger*):

    """

    Write the top 'n' reviewed movies for each age group to separate text files.

    Parameters

    ----------

    movie\_names\_review\_with\_user : RDD

        An RDD containing movie records with age group information, including movie ID, movie name, age group, age, and rating.

    age\_groups : dict

        A dictionary containing age group definitions as key-value pairs.

        The keys are tuples representing age range boundaries, and the values are the corresponding age group labels.

    n : int

        The number of top reviewed movies to retrieve for each age group.

    age\_group\_output\_path : str

        The base path for saving age group-specific text files.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    None

    Raises

    ------

    Exception

        If an error occurs during writing, it is logged and raised.

    Notes

    -----

    This function processes movie data with age group information to write the top 'n' reviewed movies for each age group to separate text files.

    It follows these steps:

    1. Transform the input RDD to prepare it for calculations.

    2. Calculate the total count of movies per age group.

    3. Insert the total count into the RDD.

    4. Find the list of age groups per movie ID.

    5. Insert the list of age groups into the RDD.

    6. Loop through each age group and filter movies by the age group.

    7. Calculate the average rating per movie in the age group.

    8. Sort movies by the number of reviews in descending order.

    9. Select the top 'n' reviewed movies for the age group.

    10. Sort the selected movies by their average rating in descending order.

    11. Save the sorted movies to text files specific to each age group using 'age\_group\_output\_path'.

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

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

    """

    try:

        transformed\_movie\_names\_review\_with\_user = movie\_names\_review\_with\_user.map(

            lambda *x*: (x[0][0], (x[0][1], x[0][2], x[1][0], x[1][1])))

        # Calculate the total count of movies per age group

        total\_movie\_count\_by\_age\_group = transformed\_movie\_names\_review\_with\_user.map(

            lambda *x*: (x[0], 1)).reduceByKey(lambda *a*, *b*: a + b)

        # Insert total count

        total\_movie\_count\_by\_age\_group = transformed\_movie\_names\_review\_with\_user.join(

            total\_movie\_count\_by\_age\_group).map(lambda *x*: (

                x[1][0][0], (x[1][1], x[0], x[1][0][1], x[1][0][2], x[1][0][3])))

        movie\_age\_group = movie\_names\_review\_with\_user.map(

            lambda *x*: (x[0][1], x[0][0]))

        # Find list of age groups per movie id

        age\_group\_list = movie\_age\_group.distinct().groupByKey().map(

            lambda *x*: (x[0], (*list*(x[1]))))

        # Insert list of age groups

        total\_movie\_count\_by\_age\_group = total\_movie\_count\_by\_age\_group.join(age\_group\_list).map(

            lambda *x*: ((x[1][0][1], x[0], x[1][0][2]), (x[1][0][0], x[1][1], x[1][0][3], x[1][0][4])))

        # Loop through each genre

        for age\_range, age\_group in age\_groups.items():

            # Filter movies by the age\_group

            filtered\_movies\_in\_age\_group = total\_movie\_count\_by\_age\_group.filter(

                lambda *x*: x[0][0] == age\_group)

            # Calculate average rating per movie

            filtered\_movie\_names\_with\_avg\_rating = filtered\_movies\_in\_age\_group.groupByKey().map(lambda *x*: (len(x[1]), *list*(

                x[1])[0][0], *list*(x[1])[0][1], x[0][1], x[0][2], sum(*float*(item[3]) for item in x[1]) / len(x[1])))

            # Sort movies by number of reviews in descending order

            most\_reviewed\_movies = filtered\_movie\_names\_with\_avg\_rating.sortBy(

                lambda *x*: (x[0]), *ascending*=False).map(lambda *x*: (x[1], x[2], x[3], x[4], x[5]))

            top\_n\_movie\_by\_age\_group = most\_reviewed\_movies.zipWithIndex().filter(

                lambda *x*: x[1] < n).keys()

            # Sort movies by average rating in descending order

            sorted\_top\_n\_movie\_by\_age\_group = top\_n\_movie\_by\_age\_group.sortBy(

                lambda *x*: x[4], *ascending*=False)

            age\_group\_output\_path\_for\_age\_group = f"{age\_group\_output\_path}\{age\_group}"

            sorted\_top\_n\_movie\_by\_age\_group.coalesce(

                1).saveAsTextFile(age\_group\_output\_path\_for\_age\_group)

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_occupation\_genre(*movie\_genre\_with\_avg\_rating*, *occupation\_list*, *genre\_list*, *n*, *logger*):

    """

    Write the top 'n' reviewed movies for each occupation and genre to separate text files.

    Parameters

    ----------

    movie\_genre\_with\_avg\_rating : RDD

        An RDD containing movie records with genres and average ratings, where each record is a tuple in the format:

        ((occupation, genre), movie\_id, movie\_name, number\_of\_reviews, average\_rating).

    occupation\_list : list

        A list of occupations for which to calculate the top reviewed movies.

    genre\_list : list

        A list of genres for which to calculate the top reviewed movies.

    n : int

        The number of top reviewed movies to retrieve for each occupation and genre.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    None

    Raises

    ------

    Exception

        If an error occurs during writing, it is logged and raised.

    Notes

    -----

    This function processes movie data with occupation and genre information to write the top 'n' reviewed movies for each combination of occupation and genre to separate text files.

    It follows these steps:

    1. Loop through each occupation in the 'occupation\_list'.

    2. Loop through each genre in the 'genre\_list'.

    3. Filter movies by the current occupation and genre combination.

    4. Sort movies by the number of reviews in descending order.

    5. Select the top 'n' reviewed movies for the current occupation and genre combination.

    6. Sort the selected movies by their average rating in descending order.

    7. Construct a header indicating the current occupation and genre combination.

    8. Log the header and the top reviewed movies for that combination.

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

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

    """

    try:

        # Loop through each occupation

        for occupation in occupation\_list:

            # Loop through each genre

            for genre\_name in genre\_list:

                # Filter movies by the current occupation and genre

                filtered\_movies = movie\_genre\_with\_avg\_rating.filter(

                    lambda *x*: x[0][0] == occupation and genre\_name in x[0][1])

                # Sort movies by number of reviews in descending order

                filtered\_movie\_names\_with\_count = filtered\_movies.sortBy(

                    lambda *x*: (x[1]), *ascending*=False).map(lambda *x*: (x[0][0], x[0][1], x[0][2], x[0][3], x[2]))

                top\_n\_movie\_by\_genre = filtered\_movie\_names\_with\_count.zipWithIndex().filter(

                    lambda *x*: x[1] < n).keys()

                # Sort movies by average rating in descending order

                sorted\_top\_n\_movie\_by\_genre = top\_n\_movie\_by\_genre.sortBy(

                    lambda *x*: (x[4]), *ascending*=False)

                header = f"Occupation: {occupation}, Genre: {genre\_name}"

                logger.info(

                    f"{header}\n{sorted\_top\_n\_movie\_by\_genre.collect()}")

    except *Exception* as e:

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

        raise e

def add\_new\_user\_profiles(*existing\_reviews\_rdd*, *new\_user\_reviews*, *sc*, *logger*):

    """

    Add new user profiles to existing reviews in an RDD.

    Parameters

    ----------

    existing\_reviews\_rdd : RDD

        An RDD containing existing user reviews.

    new\_user\_reviews : list

        A list of new user reviews to be added.

    sc : SparkContext

        The SparkContext for parallelizing the new user reviews.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        An RDD containing updated reviews after adding the new user profiles.

    Raises

    ------

    Exception

        If an error occurs during the updating, it is logged and raised.

    Notes

    -----

    This function takes a list of new user reviews, converts it into an RDD, and adds the new user profiles to the existing reviews.

    The resulting RDD contains all the reviews, including the new user profiles.

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

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

    """

    try:

        new\_user\_reviews\_rdd = sc.parallelize(new\_user\_reviews)

        updated\_reviews = new\_user\_reviews\_rdd.union(existing\_reviews\_rdd)

        return updated\_reviews

    except *Exception* as e:

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

        raise e

def main():

    """

    Entry point of the script for processing movies data.

    Parameters

    ----------

    None

    Returns

    -------

    None

    Notes

    -----

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

    1. Configures the logging settings and initializes a logger.

    2. Creates a SparkContext for data processing.

    3. Loads movie ratings data from a CSV file and cleanses it from trailing spaces.

    4. Displays and logs various statistics and analysis results, such as transaction counts, unique reviewer and movie counts, top reviewers, top reviewed movies, and movie release years.

    5. Analyzes and logs top reviewed movies by genre and occupation.

    6. Adds new user profiles to the existing reviews.

    7. Builds and trains an ALS recommendation model using user ratings.

    8. Generates movie recommendations for a specific user.

    9. Stops the SparkContext when processing is complete.

    """

    logger = configure\_logging()

    sc = create\_spark\_context()

    try:

        # Load tab delimited file

        mov\_review\_rdd = load\_data(

            sc, logger, MOVIE\_RATINGS\_DATA\_FILE\_PATH, "\t")

        show\_rdd(mov\_review\_rdd, logger)

        rating\_count = mov\_review\_rdd.count()

        logger.info(f"There are {rating\_count} rating records in the rdd.\n")

        count\_unique\_reviewers = count\_unique\_by(mov\_review\_rdd, 1, logger)

        logger.info(

            f"There are {count\_unique\_reviewers} unique number of reviewers.")

        count\_unique\_mov = count\_unique\_by(mov\_review\_rdd, 2, logger)

        logger.info(

            f"There are {count\_unique\_mov} unique number of movies reviewed.")

        top\_10\_reviewers = top\_n\_counts\_by(mov\_review\_rdd, 1, 10, logger)

        logger.info(top\_10\_reviewers)

        top\_10\_mov\_reviewed = top\_n\_counts\_by(mov\_review\_rdd, 2, 10, logger)

        top\_10\_mov\_reviewed\_rdd = sc.parallelize(

            top\_10\_mov\_reviewed).map(lambda *x*: (x[0], x[1]))

        mov\_item\_rdd = load\_data(

            sc, logger, MOVIE\_ITEMS\_DATA\_FILE\_PATH, "|")

        top\_10\_mov\_reviewed\_rdd = top\_10\_mov\_reviewed\_rdd.join(

            mov\_item\_rdd)

        top\_10\_mov\_reviewed\_rdd = top\_10\_mov\_reviewed\_rdd.map(

            lambda *item*: (item[0], item[1][1], item[1][0]))

        top\_10\_mov\_reviewed\_rdd = top\_10\_mov\_reviewed\_rdd.sortBy(

            lambda *item*: item[2], *ascending*=False)

        show\_rdd(top\_10\_mov\_reviewed\_rdd, logger)

        mov\_release\_years = date\_parse(mov\_item\_rdd, 2, logger)

        sorted\_mov\_release\_years\_count = count\_movie\_release\_years(

            mov\_release\_years, logger)

        logger.info(sorted\_mov\_release\_years\_count)

        movie\_release\_year\_range(mov\_release\_years, logger)

        top\_year = *list*(sorted\_mov\_release\_years\_count)[0]

        mov\_in\_max\_year = mov\_item\_rdd.filter(

            lambda *line*: line[2][-4:] == top\_year)

        # Select movie id, movie name and list of genres

        top\_year\_mov\_with\_genre = mov\_in\_max\_year.map(lambda *line*: (

            line[0], (line[1], line[5:])))

        mov\_genre\_rdd = load\_data(

            sc, logger, MOVIE\_GENRE\_DATA\_FILE\_PATH, "|")

        top\_year\_mov\_with\_genre = extract\_genres(

            top\_year\_mov\_with\_genre, mov\_genre\_rdd, logger)

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[1], (x[2])))

        mov\_genre\_with\_rating = mov\_ratings.join(top\_year\_mov\_with\_genre)

        # Compute for average rating and number of reviews based on genre, movie id and movie name

        mov\_genre\_with\_avg\_rating = mov\_genre\_with\_rating.groupBy(lambda *x*: (*tuple*(x[1][1][0]), x[0], x[1][1][1])).map(

            lambda *x*: (x[0], len(x[1]), sum(*int*(item[1][0]) for item in x[1]) / len(x[1])))

        write\_top\_n\_reviewed\_movies\_by\_genre(

            mov\_genre\_with\_avg\_rating, mov\_genre\_rdd, 3, TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH, logger)

        mov\_user\_rdd = load\_data(

            sc, logger, MOVIE\_USER\_DATA\_FILE\_PATH, "|")

        mov\_user\_age = mov\_user\_rdd.map(lambda *x*: (x[0], (x[1])))

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[0], (x[1], x[2])))

        mov\_review\_with\_user = mov\_ratings.join(mov\_user\_age).map(

            lambda *x*: (x[1][0][0], (x[1][0][1], x[1][1])))

        mov\_name = mov\_item\_rdd.map(lambda *x*: (x[0], (x[1])))

        mov\_names\_review\_with\_user = mov\_review\_with\_user.join(mov\_name).map(

            lambda *x*: (x[0], x[1][1], x[1][0][0], x[1][0][1]))

        # Assign age group and outputs age group, movie id, movie name, age and rating

        mov\_names\_review\_with\_user = mov\_names\_review\_with\_user.map(lambda *x*: (

            (assign\_age\_group(x[3], age\_groups, logger), x[0], x[1]), (x[3], *float*(x[2]))))

        write\_top\_n\_reviewed\_movies\_by\_age\_group(

            mov\_names\_review\_with\_user, age\_groups, 30, TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH, logger)

        movies\_in\_summer = mov\_item\_rdd.filter(

            lambda *line*: line[2][3:-5].lower() in summer)

        # Split the movies by genre

        summer\_movies\_with\_genre = movies\_in\_summer.map(lambda *line*: (

            line[0], (line[1], line[5:])))

        summer\_movies\_with\_genre = extract\_genres(

            summer\_movies\_with\_genre, mov\_genre\_rdd, logger)

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[1], (x[2])))

        summer\_mov\_genre\_with\_rating = mov\_ratings.join(

            summer\_movies\_with\_genre)

        # Compute for average rating and number of reviews based on genre, movie id and movie name

        summer\_mov\_genre\_with\_avg\_rating = summer\_mov\_genre\_with\_rating.groupBy(lambda *x*: (*tuple*(x[1][1][0]), x[0], x[1][1][1])).map(

            lambda *x*: (x[0], len(x[1]), sum(*int*(item[1][0]) for item in x[1]) / len(x[1])))

        write\_top\_n\_reviewed\_movies\_by\_genre(

            summer\_mov\_genre\_with\_avg\_rating, mov\_genre\_rdd, 3, TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH, logger)

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[0], (x[1], x[2])))

        mov\_user\_occupation = mov\_user\_rdd.map(lambda *x*: (x[0], x[3]))

        mov\_review\_with\_user = mov\_ratings.join(mov\_user\_occupation).map(

            lambda *x*: (x[1][0][0], (x[1][1], x[1][0][1])))

        mov\_with\_genre = mov\_item\_rdd.map(lambda *line*: (

            line[0], (line[1], line[5:])))

        mov\_with\_genre = extract\_genres(

            mov\_with\_genre, mov\_genre\_rdd, logger)

        mov\_with\_genre = mov\_with\_genre.map(

            lambda *x*: (x[0], (x[1][0], x[1][1])))

        mov\_genre\_with\_rating = mov\_review\_with\_user.join(mov\_with\_genre).map(

            lambda *x*: (x[1][0][0], x[1][1][0], x[0], x[1][1][1], x[1][0][1]))

        # Compute for average rating and number of reviews based on occupation, genre, movie id and movie name

        mov\_genre\_with\_avg\_rating = mov\_genre\_with\_rating.groupBy(lambda *x*: (x[0], *tuple*(x[1]), x[2], x[3])).map(

            lambda *x*: (x[0], len(x[1]), sum(*int*(item[4]) for item in x[1]) / len(x[1])))

        occupation\_rdd = load\_data(

            sc, logger, MOVIE\_OCCUPATION\_DATA\_FILE\_PATH, "|")

        occupation\_list = occupation\_rdd.flatMap(lambda *x*: x).collect()

        genre\_list = mov\_genre\_rdd.map(lambda *x*: x[0]).collect()

        write\_top\_n\_reviewed\_movies\_by\_occupation\_genre(

            mov\_genre\_with\_avg\_rating, occupation\_list, genre\_list, 3, logger)

        write\_top\_n\_reviewed\_movies\_by\_occupation\_genre(

            mov\_genre\_with\_avg\_rating, ["administrator"], ["Action"], 3, logger)

        updated\_reviews = add\_new\_user\_profiles(

            mov\_review\_rdd, new\_user\_profiles, sc, logger)

        show\_rdd(updated\_reviews, logger)

        logger.info(updated\_reviews.count())

        # Selecting user/reviewer identifier, movie identifier, rating

        ratings = updated\_reviews.filter(lambda *row*: len(row) == 4).map(

            lambda *x*: (*int*(x[0]), *int*(x[1]), *float*(x[2])))

        # Define ALS model parameters

        rank = 20

        num\_iterations = 15

        mov\_ratings\_model = ALS.train(ratings, rank, num\_iterations)

        show\_rdd(mov\_ratings\_model.userFeatures(), logger)

        logger.info(mov\_ratings\_model.userFeatures().count())

        user\_id = 0

        num\_recommendations = 10

        # Use the model to recommend movies for the user

        mov\_recommendations = mov\_ratings\_model.recommendProducts(

            user\_id, num\_recommendations)

        print(mov\_recommendations)

        logger.info(len(mov\_recommendations))

    except *Exception* as e:

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

    finally:

        if sc is not None:

            sc.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")

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

MOVIE\_OCCUPATION\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_occupation.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

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\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

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

    try:

        rdd = sc.textFile(file\_path)

        rdd = rdd.map(lambda *line*: line.split(delimiter))

        return rdd

    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 count\_unique\_by(*loaded\_rdd*, *column\_index*, *logger*):

    try:

        unique\_reviewers\_count = loaded\_rdd.map(

            lambda *line*: line[column\_index-1]).distinct().count()

        return unique\_reviewers\_count

    except *Exception* as e:

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

        raise e

def top\_n\_counts\_by(*loaded\_rdd*, *column\_position*, *n*, *logger*):

    try:

        review\_count = loaded\_rdd.map(

            lambda *line*: line[column\_position-1]).countByValue()

        sorted\_review\_count = sorted(

            review\_count.items(), *key*=lambda *item*: item[1], *reverse*=True)

        return sorted\_review\_count[:n]

    except *Exception* as e:

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

        raise e

***Script Main Function Snippet***

        # Load tab delimited file

        mov\_review\_rdd = load\_data(

            sc, logger, MOVIE\_RATINGS\_DATA\_FILE\_PATH, "\t")

        show\_rdd(mov\_review\_rdd, logger)

        rating\_count = mov\_review\_rdd.count()

        logger.info(f"There are {rating\_count} rating records in the rdd.\n")

        count\_unique\_reviewers = count\_unique\_by(mov\_review\_rdd, 1, logger)

        logger.info(

            f"There are {count\_unique\_reviewers} unique number of reviewers.")

        count\_unique\_mov = count\_unique\_by(mov\_review\_rdd, 2, logger)

        logger.info(

            f"There are {count\_unique\_mov} unique number of movies reviewed.")

        top\_10\_reviewers = top\_n\_counts\_by(mov\_review\_rdd, 1, 10, logger)

        logger.info(top\_10\_reviewers)

        top\_10\_mov\_reviewed = top\_n\_counts\_by(mov\_review\_rdd, 2, 10, logger)

        top\_10\_mov\_reviewed\_rdd = sc.parallelize(

            top\_10\_mov\_reviewed).map(lambda *x*: (x[0], x[1]))

        mov\_item\_rdd = load\_data(

            sc, logger, MOVIE\_ITEMS\_DATA\_FILE\_PATH, "|")

        top\_10\_mov\_reviewed\_rdd = top\_10\_mov\_reviewed\_rdd.join(

            mov\_item\_rdd)

        top\_10\_mov\_reviewed\_rdd = top\_10\_mov\_reviewed\_rdd.map(

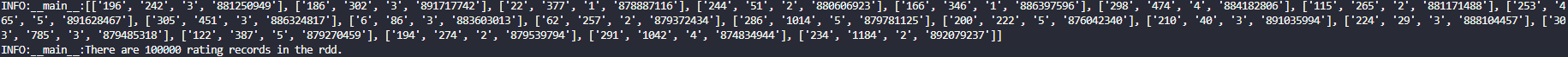
            lambda *item*: (item[0], item[1][1], item[1][0]))

        top\_10\_mov\_reviewed\_rdd = top\_10\_mov\_reviewed\_rdd.sortBy(

            lambda *item*: item[2], *ascending*=False)

        show\_rdd(top\_10\_mov\_reviewed\_rdd, logger)

***Output Screenshot***



*Figure 42. Content and number of movie reviews RDD.*



*Figure 43. Number of unique reviewers.*



*Figure 44. Number of unique movies reviewed.*



*Figure 45. Top 10 reviewers, sorted by number of movies reviewed.*



*Figure 46. Top 10 movies reviewed, sorted by number of reviews.*

**(b)**

***Script Constants Snippet***

# Constants

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

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

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

MOVIE\_OCCUPATION\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_occupation.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

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

    try:

        rdd = sc.textFile(file\_path)

        rdd = rdd.map(lambda *line*: line.split(delimiter))

        return rdd

    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 date\_parse(*movie\_item\_rdd*, *column\_index*, *logger*):

    try:

        # Extract last 4 string from the date column that ends with a 4 digit year

        movie\_release\_years = movie\_item\_rdd.map(

            lambda *line*: line[column\_index][-4:])

        return movie\_release\_years

    except *Exception* as e:

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

        raise e

def count\_movie\_release\_years(*movie\_release\_years*, *logger*):

    try:

        release\_year\_counts = movie\_release\_years.countByValue()

        sorted\_release\_year\_counts = *dict*(

            sorted(release\_year\_counts.items(), *key*=lambda *x*: x[1], *reverse*=True))

        return sorted\_release\_year\_counts

    except *Exception* as e:

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

        raise e

def movie\_release\_year\_range(*movie\_release\_years*, *logger*):

    try:

        movie\_release\_years = movie\_release\_years.filter(

            lambda *year*: year != '')

        min\_year = movie\_release\_years.min()

        max\_year = movie\_release\_years.max()

        logger.info(f"Range of Movie Release Years: {min\_year} to {max\_year}")

    except *Exception* as e:

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

        raise e

def extract\_genres(*movies\_with\_genre*, *mov\_genre\_rdd*, *logger*):

    try:

        genre\_mapping = mov\_genre\_rdd.map(

            lambda *genre*: (genre[1], genre[0])).collectAsMap()

        # Extract and map movie genres in movies\_with\_genre RDD

        movies\_with\_genre = movies\_with\_genre.map(lambda *record*: ([genre\_mapping[*str*(

            index)] for index, value in enumerate(record[1][1]) if value == "1"], record[0], record[1][0]))

        transformed\_movies\_with\_genre = movies\_with\_genre.map(

            lambda *x*: (x[1], (x[0], x[2])))

        return transformed\_movies\_with\_genre

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_genre(*movie\_genre\_with\_avg\_rating*, *genre\_rdd*, *n*, *genre\_output\_path*, *logger*):

    try:

        # Loop through each genre

        for genre\_name, genre\_id in genre\_rdd.collect():

            # Filter movies by the current genre

            filtered\_movie\_genre\_with\_avg\_rating = movie\_genre\_with\_avg\_rating.filter(

                lambda *x*: genre\_name in x[0][0]).map(

                lambda *x*: (x[0][0], x[0][1], x[0][2], x[1], x[2]))

            # Sort movies by number of reviews in descending order

            sorted\_movie\_genre\_with\_avg\_rating = filtered\_movie\_genre\_with\_avg\_rating.sortBy(

                lambda *x*: x[3], *ascending*=False).map(lambda *x*: (x[0], x[1], x[2], x[4]))

            top\_n\_movie\_by\_genre = sorted\_movie\_genre\_with\_avg\_rating.zipWithIndex().filter(

                lambda *x*: x[1] < n).keys()

            # Sort movies by average rating in descending order

            sorted\_top\_n\_movie\_by\_genre = top\_n\_movie\_by\_genre.sortBy(

                lambda *x*: x[3], *ascending*=False)

            genre\_output\_path\_for\_genre = f"{genre\_output\_path}\{genre\_name}"

            sorted\_top\_n\_movie\_by\_genre.coalesce(

                1).saveAsTextFile(genre\_output\_path\_for\_genre)

    except *Exception* as e:

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

        raise e

***Script Main Function Snippet***

        mov\_release\_years = date\_parse(mov\_item\_rdd, 2, logger)

        sorted\_mov\_release\_years\_count = count\_movie\_release\_years(

            mov\_release\_years, logger)

        logger.info(sorted\_mov\_release\_years\_count)

        movie\_release\_year\_range(mov\_release\_years, logger)

        top\_year = *list*(sorted\_mov\_release\_years\_count)[0]

        mov\_in\_max\_year = mov\_item\_rdd.filter(

            lambda *line*: line[2][-4:] == top\_year)

        # Select movie id, movie name and list of genres

        top\_year\_mov\_with\_genre = mov\_in\_max\_year.map(lambda *line*: (

            line[0], (line[1], line[5:])))

        mov\_genre\_rdd = load\_data(

            sc, logger, MOVIE\_GENRE\_DATA\_FILE\_PATH, "|")

        top\_year\_mov\_with\_genre = extract\_genres(

            top\_year\_mov\_with\_genre, mov\_genre\_rdd, logger)

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[1], (x[2])))

        mov\_genre\_with\_rating = mov\_ratings.join(top\_year\_mov\_with\_genre)

        # Compute for average rating and number of reviews based on genre, movie id and movie name

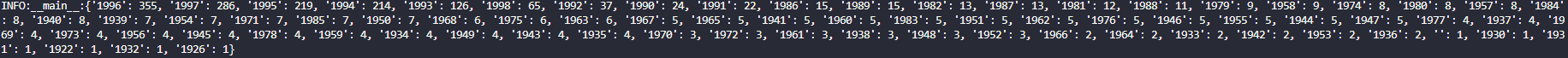
        mov\_genre\_with\_avg\_rating = mov\_genre\_with\_rating.groupBy(lambda *x*: (*tuple*(x[1][1][0]), x[0], x[1][1][1])).map(

            lambda *x*: (x[0], len(x[1]), sum(*int*(item[1][0]) for item in x[1]) / len(x[1])))

        write\_top\_n\_reviewed\_movies\_by\_genre(

            mov\_genre\_with\_avg\_rating, mov\_genre\_rdd, 3, TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH, logger)

***Output Screenshot***



*Figure 47. Number of movies released per released year.*



*Figure 48. Range of movie release years.*

A screenshot of a computer

Description automatically generated

*Figure 49. text output folder per genre.*

A screen shot of a computer

Description automatically generated

*Figure 50. Top 3 most reviewed Action movies.*

*A screen shot of a computer

Description automatically generated*

*Figure 51. Top 3 most reviewed Adventure movies.*

A screen shot of a computer

Description automatically generated

*Figure 52. Top 3 most reviewed Animation movies.*

A screen shot of a computer

Description automatically generated

*Figure 53. Top 3 most reviewed Children’s movies.*

A screenshot of a computer

Description automatically generated

*Figure 54. Top 3 most reviewed Comedy movies.*

A screenshot of a computer

Description automatically generated

*Figure 55. Top 3 most reviewed Crime movies.*

A screenshot of a computer

Description automatically generated

*Figure 56. Top 3 most reviewed Documentary movies.*

A screenshot of a computer

Description automatically generated

*Figure 57. Top 3 most reviewed Drama movies.*

A screen shot of a computer

Description automatically generated

*Figure 58. Top 3 most reviewed Fantasy movies.*

A black and white text

Description automatically generated

*Figure 59. Top 3 most reviewed Film-Noir movies.*

A screenshot of a computer

Description automatically generated

*Figure 60. Top 3 most reviewed Horror movies.*

A screen shot of a computer

Description automatically generated

*Figure 61. Top 3 most reviewed Musical movies.*

A screenshot of a computer

Description automatically generated

*Figure 62. Top 3 most reviewed Mystery movies.*

A screen shot of a computer

Description automatically generated

*Figure 63. Top 3 most reviewed Romance movies.*

A screen shot of a computer

Description automatically generated

*Figure 64. Top 3 most reviewed Sci-Fi movies.*

A screenshot of a computer

Description automatically generated

*Figure 65. Top 3 most reviewed Thriller movies.*

A black screen with white x and plus

Description automatically generated

*Figure 66. Top 3 most reviewed Unknown movies.*

A screen shot of a computer

Description automatically generated

*Figure 67. Top 3 most reviewed War movies.*

A screen shot of a computer

Description automatically generated

*Figure 68. Top 3 most reviewed Western movies.*

**(c)**

***Script Constants Snippet***

# Constants

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

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

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

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 Configurations Snippet***

# Configurations

age\_groups = {(0, 6): "[0-6]", (7, 12): "(6-12]", (13, 18): "(12-18]",

              (19, 30): "(18-30]", (31, 50): "(30-50]", (51, *float*("inf")): "50+"}

summer = ["may", "jun", "jul"]

***Script Functions Snippet***

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

    try:

        rdd = sc.textFile(file\_path)

        rdd = rdd.map(lambda *line*: line.split(delimiter))

        return rdd

    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 assign\_age\_group(*age*, *age\_groups*: *dict*, *logger*):

    try:

        age = *int*(age)

        for age\_range, group in age\_groups.items():

            if age\_range[0] <= age <= age\_range[1]:

                return group

        return 'Unknown'

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_age\_group(*movie\_names\_review\_with\_user*, *age\_groups*: *dict*, *n*, *age\_group\_output\_path*, *logger*):

    try:

        transformed\_movie\_names\_review\_with\_user = movie\_names\_review\_with\_user.map(

            lambda *x*: (x[0][0], (x[0][1], x[0][2], x[1][0], x[1][1])))

        # Calculate the total count of movies per age group

        total\_movie\_count\_by\_age\_group = transformed\_movie\_names\_review\_with\_user.map(

            lambda *x*: (x[0], 1)).reduceByKey(lambda *a*, *b*: a + b)

        # Insert total count

        total\_movie\_count\_by\_age\_group = transformed\_movie\_names\_review\_with\_user.join(

            total\_movie\_count\_by\_age\_group).map(lambda *x*: (

                x[1][0][0], (x[1][1], x[0], x[1][0][1], x[1][0][2], x[1][0][3])))

        movie\_age\_group = movie\_names\_review\_with\_user.map(

            lambda *x*: (x[0][1], x[0][0]))

        # Find list of age groups per movie id

        age\_group\_list = movie\_age\_group.distinct().groupByKey().map(

            lambda *x*: (x[0], (*list*(x[1]))))

        # Insert list of age groups

        total\_movie\_count\_by\_age\_group = total\_movie\_count\_by\_age\_group.join(age\_group\_list).map(

            lambda *x*: ((x[1][0][1], x[0], x[1][0][2]), (x[1][0][0], x[1][1], x[1][0][3], x[1][0][4])))

        # Loop through each genre

        for age\_range, age\_group in age\_groups.items():

            # Filter movies by the age\_group

            filtered\_movies\_in\_age\_group = total\_movie\_count\_by\_age\_group.filter(

                lambda *x*: x[0][0] == age\_group)

            # Calculate average rating per movie

            filtered\_movie\_names\_with\_avg\_rating = filtered\_movies\_in\_age\_group.groupByKey().map(lambda *x*: (len(x[1]), *list*(

                x[1])[0][0], *list*(x[1])[0][1], x[0][1], x[0][2], sum(*float*(item[3]) for item in x[1]) / len(x[1])))

            # Sort movies by number of reviews in descending order

            most\_reviewed\_movies = filtered\_movie\_names\_with\_avg\_rating.sortBy(

                lambda *x*: (x[0]), *ascending*=False).map(lambda *x*: (x[1], x[2], x[3], x[4], x[5]))

            top\_n\_movie\_by\_age\_group = most\_reviewed\_movies.zipWithIndex().filter(

                lambda *x*: x[1] < n).keys()

            # Sort movies by average rating in descending order

            sorted\_top\_n\_movie\_by\_age\_group = top\_n\_movie\_by\_age\_group.sortBy(

                lambda *x*: x[4], *ascending*=False)

            age\_group\_output\_path\_for\_age\_group = f"{age\_group\_output\_path}\{age\_group}"

            sorted\_top\_n\_movie\_by\_age\_group.coalesce(

                1).saveAsTextFile(age\_group\_output\_path\_for\_age\_group)

    except *Exception* as e:

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

        raise e

***Script Main Function Snippet***

        mov\_user\_rdd = load\_data(

            sc, logger, MOVIE\_USER\_DATA\_FILE\_PATH, "|")

        mov\_user\_age = mov\_user\_rdd.map(lambda *x*: (x[0], (x[1])))

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[0], (x[1], x[2])))

        mov\_review\_with\_user = mov\_ratings.join(mov\_user\_age).map(

            lambda *x*: (x[1][0][0], (x[1][0][1], x[1][1])))

        mov\_name = mov\_item\_rdd.map(lambda *x*: (x[0], (x[1])))

        mov\_names\_review\_with\_user = mov\_review\_with\_user.join(mov\_name).map(

            lambda *x*: (x[0], x[1][1], x[1][0][0], x[1][0][1]))

        # Assign age group and outputs age group, movie id, movie name, age and rating

        mov\_names\_review\_with\_user = mov\_names\_review\_with\_user.map(lambda *x*: (

            (assign\_age\_group(x[3], age\_groups, logger), x[0], x[1]), (x[3], *float*(x[2]))))

        write\_top\_n\_reviewed\_movies\_by\_age\_group(

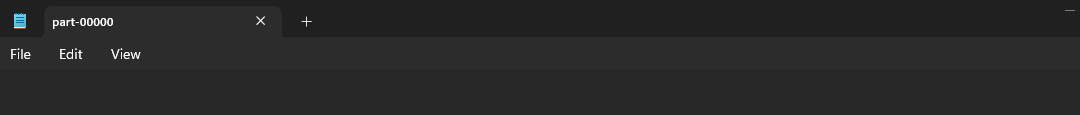
            mov\_names\_review\_with\_user, age\_groups, 30, TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH, logger)

***Output Screenshot***

*A screenshot of a computer

Description automatically generated*

*Figure 69. text output folder per age group.*

**

*Figure 70. Top 30 most reviewed for age group [0-6].*

*A screen shot of a computer

Description automatically generated*

*Figure 71. Top 30 most reviewed for age group* *(6-12].*

*A screen shot of a computer program

Description automatically generated  
Figure 72. Top 30 most reviewed for age group* *(12-18].*

*A screen shot of a computer screen

Description automatically generated*

*Figure 73. Top 30 most reviewed for age group* *(18-30].*

*A screen shot of a computer screen

Description automatically generated*

*Figure 74. Top 30 most reviewed for age group* *(30-50].*

*A screen shot of a computer screen

Description automatically generated*

*Figure 75. Top 30 most reviewed for age group* *50+.*

**(d)**

***Script Constants Snippet***

# Constants

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

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

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

MOVIE\_OCCUPATION\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_occupation.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

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 Configurations Snippet***

# Configurations

age\_groups = {(0, 6): "[0-6]", (7, 12): "(6-12]", (13, 18): "(12-18]",

              (19, 30): "(18-30]", (31, 50): "(30-50]", (51, *float*("inf")): "50+"}

summer = ["may", "jun", "jul"]

new\_user\_profiles = [

    [0, 50, 5, 881250949],

    [0, 172, 5, 881250949],

    [0, 181, 5, 881250949]

]

***Script Functions Snippet***

def extract\_genres(*movies\_with\_genre*, *mov\_genre\_rdd*, *logger*):

    try:

        genre\_mapping = mov\_genre\_rdd.map(

            lambda *genre*: (genre[1], genre[0])).collectAsMap()

        # Extract and map movie genres in movies\_with\_genre RDD

        movies\_with\_genre = movies\_with\_genre.map(lambda *record*: ([genre\_mapping[*str*(

            index)] for index, value in enumerate(record[1][1]) if value == "1"], record[0], record[1][0]))

        transformed\_movies\_with\_genre = movies\_with\_genre.map(

            lambda *x*: (x[1], (x[0], x[2])))

        return transformed\_movies\_with\_genre

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_genre(*movie\_genre\_with\_avg\_rating*, *genre\_rdd*, *n*, *genre\_output\_path*, *logger*):

    try:

        # Loop through each genre

        for genre\_name, genre\_id in genre\_rdd.collect():

            # Filter movies by the current genre

            filtered\_movie\_genre\_with\_avg\_rating = movie\_genre\_with\_avg\_rating.filter(

                lambda *x*: genre\_name in x[0][0]).map(

                lambda *x*: (x[0][0], x[0][1], x[0][2], x[1], x[2]))

            # Sort movies by number of reviews in descending order

            sorted\_movie\_genre\_with\_avg\_rating = filtered\_movie\_genre\_with\_avg\_rating.sortBy(

                lambda *x*: x[3], *ascending*=False).map(lambda *x*: (x[0], x[1], x[2], x[4]))

            top\_n\_movie\_by\_genre = sorted\_movie\_genre\_with\_avg\_rating.zipWithIndex().filter(

                lambda *x*: x[1] < n).keys()

            # Sort movies by average rating in descending order

            sorted\_top\_n\_movie\_by\_genre = top\_n\_movie\_by\_genre.sortBy(

                lambda *x*: x[3], *ascending*=False)

            genre\_output\_path\_for\_genre = f"{genre\_output\_path}\{genre\_name}"

            sorted\_top\_n\_movie\_by\_genre.coalesce(

                1).saveAsTextFile(genre\_output\_path\_for\_genre)

    except *Exception* as e:

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

        raise e

            top\_n\_movie\_by\_genre = sorted\_movie\_genre\_with\_avg\_rating.zipWithIndex().filter(

                lambda *x*: x[1] < n).keys()

            # Sort movies by average rating in descending order

            sorted\_top\_n\_movie\_by\_genre = top\_n\_movie\_by\_genre.sortBy(

                lambda *x*: x[3], *ascending*=False)

            genre\_output\_path\_for\_genre = f"{genre\_output\_path}\{genre\_name}"

            sorted\_top\_n\_movie\_by\_genre.coalesce(

                1).saveAsTextFile(genre\_output\_path\_for\_genre)

    except *Exception* as e:

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

        raise e

***Script Main Function Snippet***

        movies\_in\_summer = mov\_item\_rdd.filter(

            lambda *line*: line[2][3:-5].lower() in summer)

        # Split the movies by genre

        summer\_movies\_with\_genre = movies\_in\_summer.map(lambda *line*: (

            line[0], (line[1], line[5:])))

        summer\_movies\_with\_genre = extract\_genres(

            summer\_movies\_with\_genre, mov\_genre\_rdd, logger)

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[1], (x[2])))

        summer\_mov\_genre\_with\_rating = mov\_ratings.join(

            summer\_movies\_with\_genre)

        # Compute for average rating and number of reviews based on genre, movie id and movie name

        summer\_mov\_genre\_with\_avg\_rating = summer\_mov\_genre\_with\_rating.groupBy(lambda *x*: (*tuple*(x[1][1][0]), x[0], x[1][1][1])).map(

            lambda *x*: (x[0], len(x[1]), sum(*int*(item[1][0]) for item in x[1]) / len(x[1])))

        write\_top\_n\_reviewed\_movies\_by\_genre(

            summer\_mov\_genre\_with\_avg\_rating, mov\_genre\_rdd, 3, TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH, logger)

***Output Screenshot***

A screenshot of a computer

Description automatically generated

*Figure 76. text output folder per genre.*

A screenshot of a computer

Description automatically generated

*Figure 77. Top 3 most reviewed Action movies of summer.*

*A screenshot of a computer

Description automatically generated*

*Figure 78. Top 3 most reviewed Adventure movies of summer.*

A screen shot of a computer screen

Description automatically generated

*Figure 79. Top 3 most reviewed Animation movies of summer.*

A screen shot of a computer

Description automatically generated

*Figure 80. Top 3 most reviewed Children’s movies of summer.*

A screen shot of a computer

Description automatically generated

*Figure 81. Top 3 most reviewed Comedy movies of summer.*

A screen shot of a computer

Description automatically generated

*Figure 82. Top 3 most reviewed Crime movies of summer.*

A black and white screen with white text

Description automatically generated

*Figure 83. Top 3 most reviewed Documentary movies of summer.*

A screenshot of a computer

Description automatically generated

*Figure 84. Top 3 most reviewed Drama movies of summer.*

A screen shot of a computer

Description automatically generated

*Figure 85. Top 3 most reviewed Fantasy movies of summer.*

A black and white background with a white cross

Description automatically generated

*Figure 86. Top 3 most reviewed Film-Noir movies of summer.*

A screen shot of a computer

Description automatically generated

*Figure 87. Top 3 most reviewed Horror movies of summer.*

A screen shot of a computer

Description automatically generated

*Figure 88. Top 3 most reviewed Musical movies of summer.*

A screenshot of a computer

Description automatically generated

*Figure 89. Top 3 most reviewed Mystery movies of summer.*

A computer screen shot of a computer code

Description automatically generated

*Figure 90. Top 3 most reviewed Romance movies of summer.*

A screenshot of a computer

Description automatically generated

*Figure 91. Top 3 most reviewed Sci-Fi movies of summer.*

A screenshot of a computer

Description automatically generated

*Figure 92. Top 3 most reviewed Thriller movies of summer.*

A black and white background

Description automatically generated

*Figure 93. Top 3 most reviewed Unknown movies of summer.*

A black and white screen

Description automatically generated

*Figure 94. Top 3 most reviewed War movies of summer.*

A black screen with white text

Description automatically generated

*Figure 95. Top 3 most reviewed Western movies of summer.*

**(e)**

***Script Constants Snippet***

# Constants

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

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

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

MOVIE\_OCCUPATION\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_occupation.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

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

    try:

        rdd = sc.textFile(file\_path)

        rdd = rdd.map(lambda *line*: line.split(delimiter))

        return rdd

    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 extract\_genres(*movies\_with\_genre*, *mov\_genre\_rdd*, *logger*):

    try:

        genre\_mapping = mov\_genre\_rdd.map(

            lambda *genre*: (genre[1], genre[0])).collectAsMap()

        # Extract and map movie genres in movies\_with\_genre RDD

        movies\_with\_genre = movies\_with\_genre.map(lambda *record*: ([genre\_mapping[*str*(

            index)] for index, value in enumerate(record[1][1]) if value == "1"], record[0], record[1][0]))

        transformed\_movies\_with\_genre = movies\_with\_genre.map(

            lambda *x*: (x[1], (x[0], x[2])))

        return transformed\_movies\_with\_genre

    except *Exception* as e:

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

        raise e

def write\_top\_n\_reviewed\_movies\_by\_occupation\_genre(*movie\_genre\_with\_avg\_rating*, *occupation\_list*, *genre\_list*, *n*, *logger*):

    try:

        # Loop through each occupation

        for occupation in occupation\_list:

            # Loop through each genre

            for genre\_name in genre\_list:

                # Filter movies by the current occupation and genre

                filtered\_movies = movie\_genre\_with\_avg\_rating.filter(

                    lambda *x*: x[0][0] == occupation and genre\_name in x[0][1])

                # Sort movies by number of reviews in descending order

                filtered\_movie\_names\_with\_count = filtered\_movies.sortBy(

                    lambda *x*: (x[1]), *ascending*=False).map(lambda *x*: (x[0][0], x[0][1], x[0][2], x[0][3], x[2]))

                top\_n\_movie\_by\_genre = filtered\_movie\_names\_with\_count.zipWithIndex().filter(

                    lambda *x*: x[1] < n).keys()

                # Sort movies by average rating in descending order

                sorted\_top\_n\_movie\_by\_genre = top\_n\_movie\_by\_genre.sortBy(

                    lambda *x*: (x[4]), *ascending*=False)

                header = f"Occupation: {occupation}, Genre: {genre\_name}"

                logger.info(

                    f"{header}\n{sorted\_top\_n\_movie\_by\_genre.collect()}")

    except *Exception* as e:

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

        raise e

***Script Main Function Snippet***

        mov\_ratings = mov\_review\_rdd.map(lambda *x*: (x[0], (x[1], x[2])))

        mov\_user\_occupation = mov\_user\_rdd.map(lambda *x*: (x[0], x[3]))

        mov\_review\_with\_user = mov\_ratings.join(mov\_user\_occupation).map(

            lambda *x*: (x[1][0][0], (x[1][1], x[1][0][1])))

        mov\_with\_genre = mov\_item\_rdd.map(lambda *line*: (

            line[0], (line[1], line[5:])))

        mov\_with\_genre = extract\_genres(

            mov\_with\_genre, mov\_genre\_rdd, logger)

        mov\_with\_genre = mov\_with\_genre.map(

            lambda *x*: (x[0], (x[1][0], x[1][1])))

        mov\_genre\_with\_rating = mov\_review\_with\_user.join(mov\_with\_genre).map(

            lambda *x*: (x[1][0][0], x[1][1][0], x[0], x[1][1][1], x[1][0][1]))

        # Compute for average rating and number of reviews based on occupation, genre, movie id and movie name

        mov\_genre\_with\_avg\_rating = mov\_genre\_with\_rating.groupBy(lambda *x*: (x[0], *tuple*(x[1]), x[2], x[3])).map(

            lambda *x*: (x[0], len(x[1]), sum(*int*(item[4]) for item in x[1]) / len(x[1])))

        occupation\_rdd = load\_data(

            sc, logger, MOVIE\_OCCUPATION\_DATA\_FILE\_PATH, "|")

        occupation\_list = occupation\_rdd.flatMap(lambda *x*: x).collect()

        genre\_list = mov\_genre\_rdd.map(lambda *x*: x[0]).collect()

        write\_top\_n\_reviewed\_movies\_by\_occupation\_genre(

            mov\_genre\_with\_avg\_rating, occupation\_list, genre\_list, 3, logger)

        write\_top\_n\_reviewed\_movies\_by\_occupation\_genre(

            mov\_genre\_with\_avg\_rating, ["administrator"], ["Action"], 3, logger)

***Output Screenshot***

******

Figure 96. First 20 results for top 3 movies per occupation and genre category part 1

******

Figure 97. First 20 results for top 3 movies per occupation and genre category part 2



Figure 98. First 20 results for top 3 movies per occupation and genre category part 3



Figure 99. First 20 results for top 3 movies per occupation and genre category part 4



Figure 100. First 20 results for top 3 movies per occupation and genre category part 5



Figure 101. First 20 results for top 3 movies per occupation and genre category part 6



Figure 102. First 20 results for top 3 movies per occupation and genre category part 7



Figure 103. First 20 results for top 3 movies per occupation and genre category part 8



Figure 104. First 20 results for top 3 movies per occupation and genre category part 9



Figure 105. First 20 results for top 3 movies per occupation and genre category part 10



Figure 106. First 20 results for top 3 movies per occupation and genre category part 11



Figure 107. First 20 results for top 3 movies per occupation and genre category part 12



Figure 108. First 20 results for top 3 movies per occupation and genre category part 13



Figure 109. First 20 results for top 3 movies per occupation and genre category part 14



Figure 110. First 20 results for top 3 movies per occupation and genre category part 15



Figure 111. First 20 results for top 3 movies per occupation and genre category part 16



Figure 112. First 20 results for top 3 movies per occupation and genre category part 17



Figure 113. First 20 results for top 3 movies per occupation and genre category part 18



Figure 114. First 20 results for top 3 movies per occupation and genre category part 19



Figure 115. First 20 results for top 3 movies per occupation and genre category part 20



Figure 116. top 3 movies by administrator occupation and action genre

**(f)**

***Script Constants Snippet***

# Constants

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

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

MOVIE\_RATINGS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_rating.dat")

MOVIE\_ITEMS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_item.dat")

MOVIE\_GENRE\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_genre.dat")

MOVIE\_USER\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_user.dat")

MOVIE\_OCCUPATION\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "mov\_occupation.dat")

TOP\_3\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(DATA\_DIR, "top\_3\_mov\_by\_genre")

TOP\_30\_MOVIE\_BY\_AGE\_GROUP\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_30\_mov\_by\_age\_group")

TOP\_3\_SUMMER\_MOVIE\_BY\_GENRE\_OUTPUT\_PATH = os.path.join(

    DATA\_DIR, "top\_3\_summer\_mov\_by\_genre")

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 Configurations Snippet***

# Configurations

age\_groups = {(0, 6): "[0-6]", (7, 12): "(6-12]", (13, 18): "(12-18]",

              (19, 30): "(18-30]", (31, 50): "(30-50]", (51, *float*("inf")): "50+"}

summer = ["may", "jun", "jul"]

new\_user\_profiles = [

    [0, 50, 5, 881250949],

    [0, 172, 5, 881250949],

    [0, 181, 5, 881250949]

]

***Script Functions Snippet***

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

def add\_new\_user\_profiles(*existing\_reviews\_rdd*, *new\_user\_reviews*, *sc*, *logger*):

    try:

        new\_user\_reviews\_rdd = sc.parallelize(new\_user\_reviews)

        updated\_reviews = new\_user\_reviews\_rdd.union(existing\_reviews\_rdd)

        return updated\_reviews

    except *Exception* as e:

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

        raise e

***Script Main Function Snippet***

        updated\_reviews = add\_new\_user\_profiles(

            mov\_review\_rdd, new\_user\_profiles, sc, logger)

        show\_rdd(updated\_reviews, logger)

        logger.info(updated\_reviews.count())

        # Selecting user/reviewer identifier, movie identifier, rating

        ratings = updated\_reviews.filter(lambda *row*: len(row) == 4).map(

            lambda *x*: (*int*(x[0]), *int*(x[1]), *float*(x[2])))

        # Define ALS model parameters

        rank = 20

        num\_iterations = 15

        mov\_ratings\_model = ALS.train(ratings, rank, num\_iterations)

        show\_rdd(mov\_ratings\_model.userFeatures(), logger)

        logger.info(mov\_ratings\_model.userFeatures().count())

        user\_id = 0

        num\_recommendations = 10

        # Use the model to recommend movies for the user

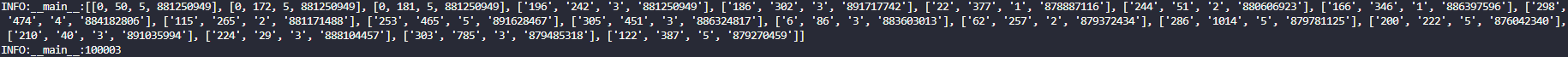
        mov\_recommendations = mov\_ratings\_model.recommendProducts(

            user\_id, num\_recommendations)

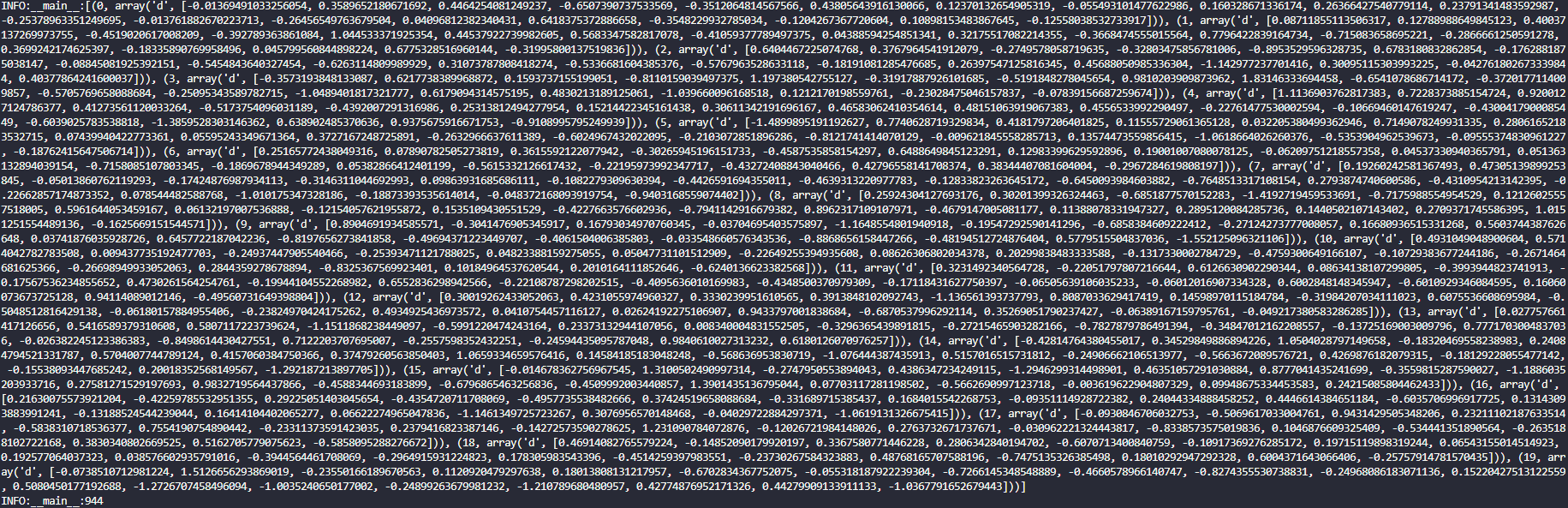
        print(mov\_recommendations)

        logger.info(len(mov\_recommendations))

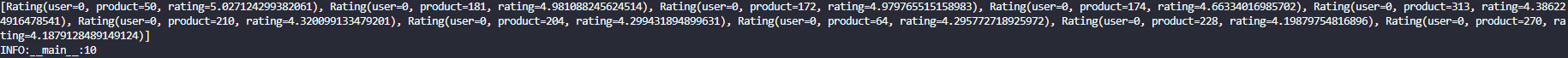
***Output Screenshot***

**

*Figure 117. Content and number of movie reviews RDD.*

**

*Figure 118. Trained ALS model output and number of movies RDD.*

**

*Figure 119. Top 10 movie recommendations for user id = 1.*