**Collaborative Filtering for Netflix Movie Recommendations**

**Group members:**

Ashish Kumar - G23AI1005

Samir Kumar Mishra - G23AI1052

Prudhivi Rachan Kumar Sai - G23AI1030

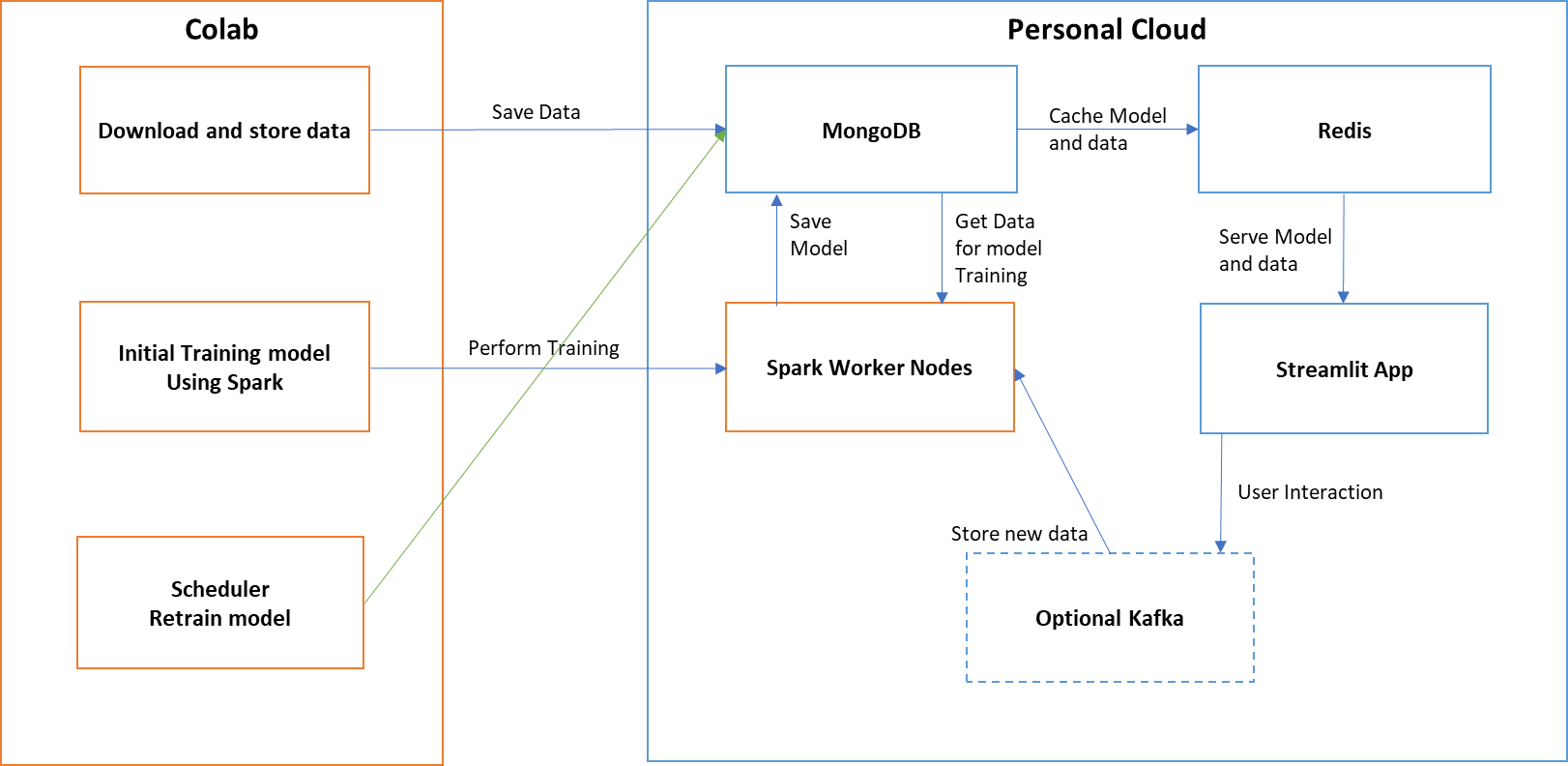
**Overview:**

Collaborative filtering relies on the concept that people who liked something in the past would also like the same experience in future. This project aims to develop a movie recommendation system for Netflix using collaborative filtering techniques to predict user preferences based on their past ratings and behaviour. The primary goal is to enhance the user experience by providing personalized movie

recommendations. This project will utilize the Netflix dataset, which contains a large volume of user

rating data, and will apply big data analytics to derive meaningful insights and accurate predictions.

**Overall Architecture Diagram:**



**Steps and Implementation:**

1. Data Ingestion:
   1. Apache Kafka on Docker (Optional)
      1. Purpose: Stream user interactions and new ratings in real-time.
      2. Setup: Use Docker to run Kafka locally or on a cloud VM with free credits (like Google Cloud or AWS Free Tier).
   2. Spark connector for MongoDB
      1. Purpose: Stream existing user rating data to process.
      2. Setup: Ran Spark on Google Colab.
2. Data Storage:
   1. Personal MongoDB Cloud:
      1. Purpose: Store historical data, Batch data, Movie data, Model data.
      2. Setup: Connected the spark application to MongoDB instance hosted on personal cloud.
   2. Personal Redis Cloud:
      1. Purpose: Cache stored Model data, Movie data and IMDB images(optional) and serve low-latency recommendations.
      2. Setup: Connected spark application to the Redis instance hosted on personal cloud.
3. Data Processing and Model Creation:
   1. Apache Spark on Google Colab:
      1. Purpose: Process historical ratings to create model for prediction and to batch process updated rating to update model.
      2. Setup: Used Google Colab to run Spark jobs.
4. Model serving:
   1. Streamlit Application:
      1. Purpose: Serve the real-time recommendation based on Model prediction.
      2. Setup: Hosted application on the personal cloud.

**Individual Details:**

1. Gather Netflix Historical data:

We download the Netflix Prize dataset from Kaggle and extract it in the Colab environment.

!kaggle datasets download -d netflix-inc/netflix-prize-data -p /content

import zipfile

zip\_path = '/content/netflix-prize-data.zip'

extract\_path = '/content/netflix-prize-data'

with zipfile.ZipFile(zip\_path, 'r') as zip\_ref:

    zip\_ref.extractall(extract\_path)

1. Connect to MongoDB:

We establish a connection to MongoDB using pymongo and specify the database and collections for storing movie titles and ratings.

!pip install pymongo

from pymongo import MongoClient

client = MongoClient('mongodb://xxx.xxx.xxx.xxx:8081/')

db = client['netflix']

1. Store Movie Titles:

We read the movie titles from the extracted CSV file and store them in MongoDB. Batching is used to handle large volumes of data efficiently.

import csv

movies\_collection = db['movies']

batch\_size = 10000

batch = []

with open('/content/netflix-prize-data/movie\_titles.csv', 'r', encoding='ISO-8859-1') as file:

    reader = csv.reader(file, delimiter=',')

    for row in reader:

        movie\_id = int(row[0])

        year = row[1]

        title = row[2]

        # Handle 'NULL' year

        year = int(year) if year != 'NULL' else None

        movie\_doc = {

            "movie\_id": movie\_id,

            "year\_of\_release": year,

            "title": title

        }

        batch.append(movie\_doc)

        if len(batch) >= batch\_size:

            movies\_collection.insert\_many(batch)

            batch.clear()

    # Insert remaining documents in the batch

    if batch:

        movies\_collection.insert\_many(batch)

1. Store Ratings:

Similarly, we read the ratings data from multiple text files and store it in MongoDB using batching.

import os

from tqdm import tqdm

ratings\_collection = db['ratings']

ratings\_files = ['combined\_data\_1.txt', 'combined\_data\_2.txt', 'combined\_data\_3.txt', 'combined\_data\_4.txt']

batch\_size = 100000

batch = []

for filename in tqdm(ratings\_files):

    with open(os.path.join('/content/netflix-prize-data/', filename), 'r') as file:

        lines = file.readlines()

        current\_movie\_id = None

        for line in lines:

            if line.endswith(':\n'):  # Movie ID line

                current\_movie\_id = int(line.strip().replace(':', ''))

            else:

                try:

                    customer\_id, rating, date = line.strip().split(',')

                    rating\_doc = {

                        "movie\_id": current\_movie\_id,

                        "customer\_id": int(customer\_id),

                        "rating": int(rating),

                        "date": date

                    }

                    batch.append(rating\_doc)

                except ValueError as e:

                    print(f"Skipping line due to parsing error: {line} - Error: {e}")

                if len(batch) >= batch\_size:

                    ratings\_collection.insert\_many(batch)

                    batch.clear()

        # Insert remaining documents in the batch

        if batch:

            ratings\_collection.insert\_many(batch)

            batch.clear()

1. Set Up Spark Environment:

We configure and initialize a Spark session to use more memory and connect to our MongoDB instance.

import pyspark

from pyspark import SparkConf, SparkSession

conf = SparkConf() \

    .setAppName("NetflixRecommendation") \

    .set("spark.driver.memory", "4g") \

    .set("spark.executor.memory", "4g") \

    .set("spark.mongodb.read.connection.uri", "mongodb://xxx.xxx.xxx.xxx:8081/netflix") \

    .set("spark.mongodb.write.connection.uri", "mongodb://xxx.xxx.xxx.xxx:8081/netflix") \

    .set("spark.jars.packages", "org.mongodb.spark:mongo-spark-connector:10.0.2") \

    .set("spark.ui.port", "4050")

spark = SparkSession.builder.config(conf=conf).getOrCreate()

1. Load and Preprocess Data with Spark:

We read the ratings data from MongoDB into a Spark DataFrame and preprocess it for training.

ratings\_df = spark.read \

    .format('mongodb') \

    .option("uri", "mongodb://xxx.xxx.xxx.xxx:8081/netflix.ratings") \

    .load()

ratings\_df = ratings\_df.withColumn("customer\_id", ratings\_df["customer\_id"].cast("integer"))

ratings\_df = ratings\_df.withColumn("movie\_id", ratings\_df["movie\_id"].cast("integer"))

ratings\_df = ratings\_df.withColumn("rating", ratings\_df["rating"].cast("float"))

1. Train and save ALS Model:

We use the ALS (Alternating Least Squares) algorithm to train a recommendation model on the ratings data.

from pyspark.ml.recommendation import ALS

als = ALS(

    maxIter=10,

    regParam=0.01,

    userCol="customer\_id",

    itemCol="movie\_id",

    ratingCol="rating",

    coldStartStrategy="drop",

    nonnegative=True

)

model = als.fit(ratings\_df)

# Save user and item factors to MongoDB

user\_factors = model.userFactors

item\_factors = model.itemFactors

user\_factors.write \

    .format("mongodb") \

    .mode("overwrite") \

    .option("uri", "mongodb://xxx.xxx.xxx.xxx:8081/netflix.user\_factors") \

    .save()

item\_factors.write \

    .format("mongodb") \

    .mode("overwrite") \

    .option("uri", "mongodb://xxx.xxx.xxx.xxx:8081/netflix.item\_factors") \

    .save()

1. Cache Data in Redis:

We cache the user factors, item factors, and movie details in Redis for efficient retrieval.

import redis

import pymongo

import json

redis\_client = redis.StrictRedis(host='xxx.xxx.xxx.xxx', port=8088, db=0)

redis\_client.flushdb()

# Cache user factors

def cache\_user\_factors():

    user\_factors\_collection = db['user\_factors']

    user\_factors = list(user\_factors\_collection.find({}))

    pipeline = redis\_client.pipeline()

    for user in user\_factors:

        user\_id = str(user['id'])

        user\_factors\_data = {

            'features': json.dumps(user['features'])

        }

        pipeline.hset(f'user\_factors:{user\_id}', mapping=user\_factors\_data)

    pipeline.execute()

cache\_user\_factors()

# Cache item factors

def cache\_item\_factors():

    item\_factors\_collection = db['item\_factors']

    item\_factors = list(item\_factors\_collection.find({}))

    item\_factors\_dict = {str(item['id']): item['features'] for item in item\_factors}

    redis\_client.set('item\_factors', json.dumps(item\_factors\_dict))

cache\_item\_factors()

# Cache movies

def cache\_movies():

    movies\_collection = db['movies']

    movies = list(movies\_collection.find({}))

    pipeline = redis\_client.pipeline()

    for movie in movies:

        movie\_id = str(movie['movie\_id'])

        title = movie.get('title', '')

        year\_of\_release = movie.get('year\_of\_release', 'None')

        movie\_data = {

            'title': title,

            'year\_of\_release': year\_of\_release

        }

        pipeline.hset(f'movie:{movie\_id}', mapping=movie\_data)

    pipeline.execute()

cache\_movies()

# Cache user IDs

def cache\_user\_ids():

    user\_factors\_collection = db['user\_factors']

    user\_factors = list(user\_factors\_collection.find({}))

    user\_ids = [str(user['id']) for user in user\_factors]

    redis\_client.sadd('user\_ids', \*user\_ids)

cache\_user\_ids()

1. Serve using streamlit:

We cache the user factors, item factors, and movie details in Redis for efficient retrieval.