**Collaborative Filtering for Netflix Movie Recommendations**

**Group members:**

Ashish Kumar - G23AI1005

Samir Kumar Mishra - G23AI1052

Prudhivi Rachan Kumar Sai - G23AI1030

**Project Github Repo:**

Link: <https://github.com/ursamir/iitj_big_data_project>

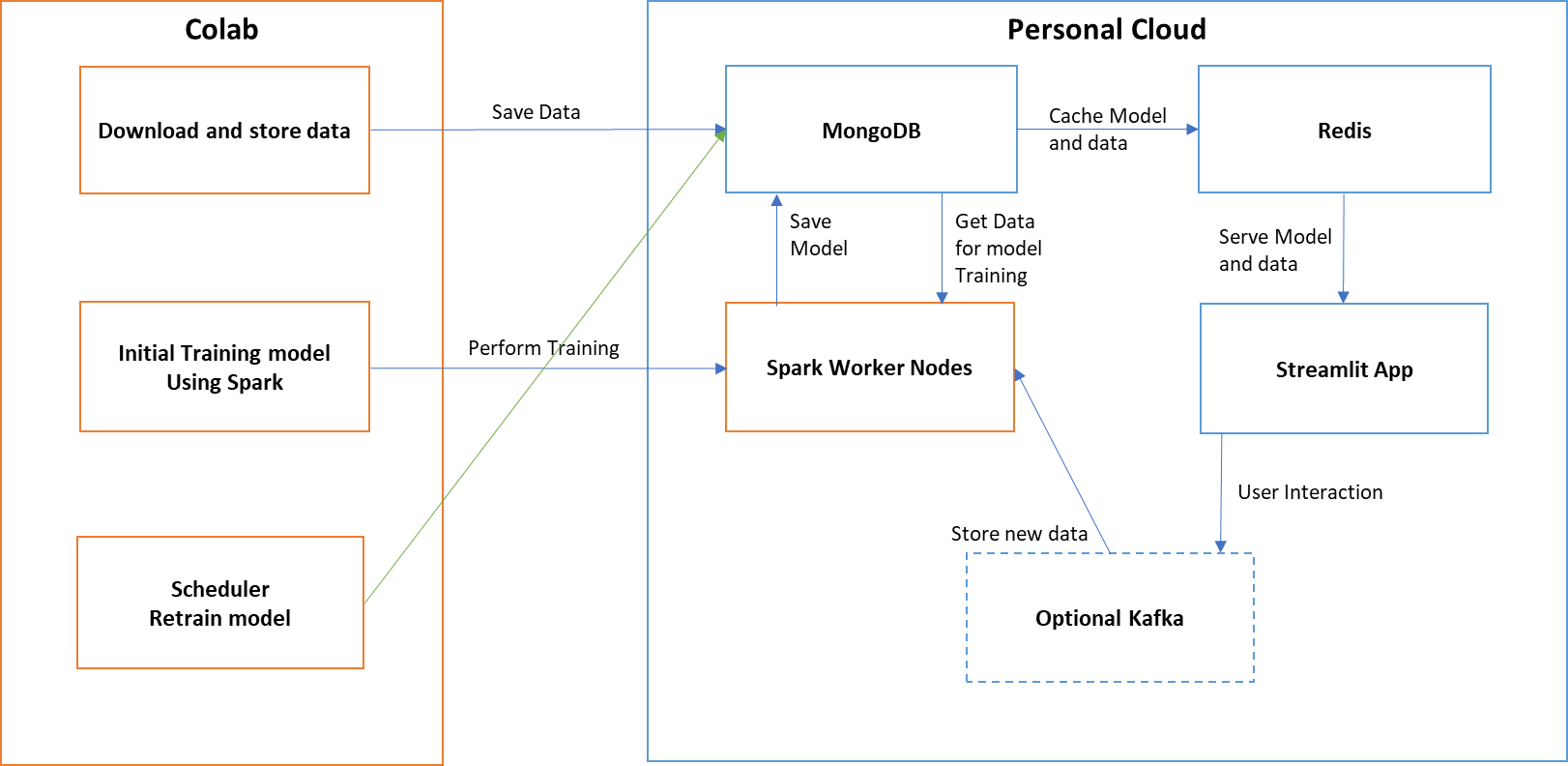
**Overview:**

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.

import streamlit as st

import redis

import json

import pymongo

import time

from imdb import IMDb

# Connect to MongoDB

mongo\_client = pymongo.MongoClient('mongodb://xxx.xxx.xxx.xxx:8081/')

mongo\_db = mongo\_client["netflix"]

mongo\_ratings\_collection = mongo\_db["ratings"]

# Connect to Redis

redis\_url = "xxx.xxx.xxx.xxx"

redis\_port = 8088

redis\_client = redis.StrictRedis(host=redis\_url, port=redis\_port, db=0)

# Load user IDs and item factors data once

g\_user\_ids = [user\_id.decode('utf-8') for user\_id in redis\_client.smembers('user\_ids')]

item\_factors\_data = redis\_client.get('item\_factors')

if item\_factors\_data:

    item\_factors = json.loads(item\_factors\_data)

else:

    item\_factors = None

# Initialize IMDb instance

ia = IMDb()

# Function to fetch user IDs from Redis

@st.cache\_data()

def fetch\_user\_ids(offset, limit):

    return g\_user\_ids[offset:offset+limit]

# Function to fetch user factors from Redis based on user\_id

def fetch\_user\_factors(user\_id):

    user\_factors\_data = redis\_client.hgetall(f'user\_factors:{user\_id}')

    if user\_factors\_data:

        user\_factors = {key.decode('utf-8'): json.loads(value) for key, value in user\_factors\_data.items()}

        return user\_factors

    else:

        return None

# Function to fetch movie details from Redis based on movie\_id

@st.cache\_data()

def fetch\_movie\_details(movie\_id):

    movie\_data = redis\_client.hgetall(f'movie:{movie\_id}')

    if movie\_data:

        # Convert bytes to string

        movie\_details = {key.decode('utf-8'): value.decode('utf-8') for key, value in movie\_data.items()}

        return movie\_details

    else:

        return None

# Function to fetch movie poster from IMDb

def fetch\_movie\_poster(title):

    search\_results = ia.search\_movie(title)

    if search\_results:

        movie = search\_results[0]

        ia.update(movie, info=['main'])

        if 'cover url' in movie:

            return movie['cover url']

    return None

# Function to recommend movies based on user factors and item factors

@st.cache\_data()

def recommend\_movies(user\_id, user\_factors, item\_factors, top\_n=10):

    if not item\_factors:

        return []

    # Fetch user's already rated movie\_ids from MongoDB

    start\_time = time.time()

    already\_rated\_movies = set()

    ratings = mongo\_ratings\_collection.find({"customer\_id": int(user\_id)})

    for rating in ratings:

        already\_rated\_movies.add(rating["movie\_id"])

    end\_time = time.time()

    print(f"Time taken to fetch already rated movies: {end\_time - start\_time:.4f} seconds")

    recommendations = []

    # Perform recommendation logic based on user and item factors (example logic)

    for movie\_id, item\_features in item\_factors.items():

        # Skip movie if already rated by the user

        if movie\_id in already\_rated\_movies:

            continue

        # Perform dot product of user factors and item factors

        user\_features = user\_factors['features']

        dot\_product = sum(u \* i for u, i in zip(user\_features, item\_features))

        # Append movie ID and dot product as a tuple

        recommendations.append((movie\_id, dot\_product))

    # Sort recommendations based on dot product (higher is better)

    recommendations.sort(key=lambda x: x[1], reverse=True)

    # Select top N recommendations

    top\_recommendations = recommendations[:top\_n]

    # Fetch movie details for the top recommendations

    recommended\_movies = []

    for movie\_id, score in top\_recommendations:

        movie\_details = fetch\_movie\_details(movie\_id)

        if movie\_details:

            movie\_details['movie\_id'] = movie\_id

            movie\_details['score'] = score

            recommended\_movies.append(movie\_details)

    return recommended\_movies

# Function to fetch top rated movies by user ID from MongoDB

def fetch\_top\_rated\_movies(user\_id, top\_n=10):

    start\_time = time.time()

    top\_rated\_movies = []

    ratings = mongo\_ratings\_collection.find({"customer\_id": int(user\_id)}).sort("rating", pymongo.DESCENDING).limit(top\_n)

    for rating in ratings:

        movie\_details = fetch\_movie\_details(rating["movie\_id"])

        if movie\_details:

            movie\_details['movie\_id'] = rating["movie\_id"]

            movie\_details['rating'] = rating["rating"]

            top\_rated\_movies.append(movie\_details)

    end\_time = time.time()

    print(f"Time taken to fetch top rated movies: {end\_time - start\_time:.4f} seconds")

    return top\_rated\_movies

# Streamlit app

def main():

    st.title('Movie Recommendation System')

    # Load user IDs in batches of 50 items

    offset = st.slider('Select offset', 0, len(g\_user\_ids), 0, 50)

    user\_ids = fetch\_user\_ids(offset, 50)

    # Allow manual input of user ID

    user\_id\_input = st.text\_input('Enter User ID manually')

    if user\_id\_input:

        user\_id = user\_id\_input.strip()

    else:

        user\_id = st.selectbox('Select User ID', list(user\_ids)) if user\_ids else None

    if st.button('Get Recommendations'):

        if user\_id:

            # Fetch user factors from Redis

            user\_factors = fetch\_user\_factors(user\_id)

            if user\_factors and item\_factors:

                with st.spinner('Fetching recommendations...'):

                    st.subheader(f'Recommendations for User {user\_id}')

                    recommendations = recommend\_movies(user\_id, user\_factors, item\_factors)

                    if recommendations:

                        # Prepare data for table

                        table\_data = []

                        for movie in recommendations:

                            poster\_url = fetch\_movie\_poster(movie["title"])

                            table\_data.append({

                                "Title": movie["title"],

                                "Movie ID": movie["movie\_id"],

                                "Year": movie["year\_of\_release"],

                                "Score": f"{movie['score']:.2f}",

                                "Poster": poster\_url

                            })

                        # Display table with posters

                        for movie in table\_data:

                            cols = st.columns([1, 2])

                            if movie["Poster"]:

                                cols[0].image(movie["Poster"], width=100)

                            cols[1].write(f"\*\*{movie['Title']}\*\* ({movie['Year']})")

                            cols[1].write(f"\*\*Score\*\*: {movie['Score']}")

                            cols[1].write(f"\*\*Movie ID\*\*: {movie['Movie ID']}")

                            st.markdown("---")

                    else:

                        st.write('No recommendations found.')

                # Fetch top rated movies by the user

                with st.spinner('Fetching top rated movies...'):

                    st.subheader(f'Top Rated Movies by User {user\_id}')

                    top\_rated\_movies = fetch\_top\_rated\_movies(user\_id)

                    if top\_rated\_movies:

                        # Prepare data for table

                        top\_rated\_table\_data = []

                        for movie in top\_rated\_movies:

                            poster\_url = fetch\_movie\_poster(movie["title"])

                            top\_rated\_table\_data.append({

                                "Title": movie["title"],

                                "Movie ID": movie["movie\_id"],

                                "Year": movie["year\_of\_release"],

                                "Rating": movie['rating'],

                                "Poster": poster\_url

                            })

                        # Display table with posters

                        for movie in top\_rated\_table\_data:

                            cols = st.columns([1, 2])

                            if movie["Poster"]:

                                cols[0].image(movie["Poster"], width=100)

                            cols[1].write(f"\*\*{movie['Title']}\*\* ({movie['Year']})")

                            cols[1].write(f"\*\*Rating\*\*: {movie['Rating']}")

                            cols[1].write(f"\*\*Movie ID\*\*: {movie['Movie ID']}")

                            st.markdown("---")

                    else:

                        st.write(f'No top rated movies found for User {user\_id}.')

            elif not user\_factors:

                st.write(f'User {user\_id} not found or no factors available.')

            else:

                st.write('Item factors not found.')

        else:

            st.write('Please select or enter a User ID.')

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Application Screenshot**

