

**UNIVERSIDAD DE SONORA**  
**DIVISIÓN CIENCIAS EXACTAS Y NATURALES**



**Tema:**

Sistema de recomendación con Apache Spark

**Materia:**

Tópicos avanzados de ciencias computacionales

**Profesor:**

Juan Pablo Soto Barrera

**Alumno:**

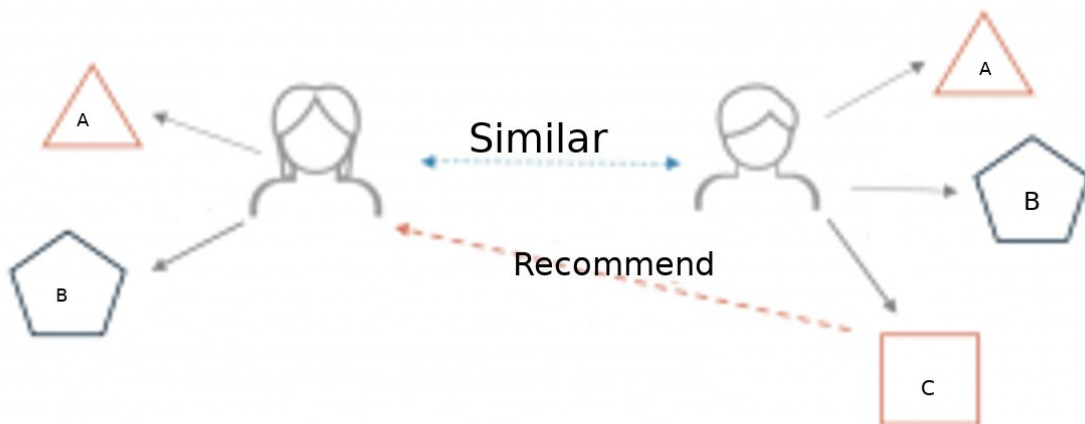
Raúl Francisco Pérez Rodríguez

# Introducción

El contenido de este documento es la explicación y implementación de un sistema de recomendación de tipo filtro colaborativo, donde se recomendarán películas a un nuevo usuario dependiendo del conjunto de películas que haya valorado previamente. Primero se verá una breve explicación de que es un sistema de recomendación y los tipos de sistemas de recomendación, por último se realizará la implementación del mismo utilizando Apache Spark con el API de python.

## Sistema de recomendación

Un sistema de recomendación es un algoritmo cuyo objetivo es proporcionar la información más relevante a un usuario mediante el descubrimiento de patrones en un conjunto de datos. El algoritmo clasifica los elementos y muestra al usuario los elementos que calificaría con buena valoración.



Por ejemplo, dos usuarios califican con buena valoración las películas A y B. Cuando esto sucede, se calcula el índice de similitud de estos dos usuarios y dependiendo de la puntuación, el sistema puede recomendar la película C al otro usuario, ya que el sistema detectó que esos dos usuarios son similares.

## Diferentes tipos de sistemas de recomendación

Los tipos de sistemas de recomendación más comunes son los sistemas de recomendación **basados en contenido** (content based) y de **filtro colaborativo** (collaborative filtering).

En el de filtro colaborativo, el comportamiento de un grupo de usuarios se utiliza para hacer recomendaciones a otros usuarios. La recomendación se basa en la preferencia de otros usuarios. Un ejemplo sería recomendar una película a un usuario basándose en el hecho de que un amigo le gusta la película.

Hay dos tipos de modelos colaborativos. Métodos basados en memoria y métodos basados en modelo. La ventaja de las técnicas basadas en memorias es que son fáciles de implementar y las recomendaciones resultantes son a menudo más fáciles de explicar. Se dividen en dos:

**Filtro colaborativo basado en el usuario** (User-based collaborative filtering): En este modelo, los productos se recomiendan a un usuario en función de que productos le gustan y han gustado a usuarios similares. Por ejemplo, si a A y B le gustan las mismas películas y sale una nueva película que le gusta a A, entonces se puede recomendar la película a B.

**Filtro colaborativo basado en elementos** (Item-based collaborative filtering): Estos sistemas identifican elementos similares según las calificaciones de los usuarios. Por ejemplo, si los usuarios A, B y C otorgaron una calificación de 5 estrellas a los libros X e Y, luego cuando un usuario D compra el libro X, también obtienen una recomendación para comprar el libro Y, porque el sistema identifica que los libros X e Y son similares según las calificaciones de usuarios A, B y C.

Los **métodos basados en modelos** se basan en la factorización matricial y son mejores para tratar la escasez. Se desarrollan utilizando minería de datos, algoritmos de aprendizaje automático para predecir la calificación de los usuarios de elementos sin calificación.

Los sistemas **basados en contenido** utilizan metadatos como género, productor, actor, músico para recomendar artículos como películas o música. Por ejemplo, recomendar Infinity War, porque actúa Vin Diesel y a alguien le gusta Rápido y furioso. Del mismo modo, puedes obtener recomendaciones de música de ciertos artistas porque te gustó su música. Los sistemas basados en contenido se basan en la idea de que si le gustó un determinado artículo es probable que le guste algo que sea similar a él.

# Implementación

Se utilizará el conjunto de datos de MovieLens para implementar un sistema de recomendación de filtro colaborativo. El conjunto de datos de MovieLens tiene dos opciones, la pequeña (100 mil ratings) o la completa (21 millones de ratings). Se utilizará la completa para probar el rendimiento de Spark.

Primero que todo hay que importar pyspark y inicializar el contexto para poder usar sus funciones.

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()
```

Después hay que cargar el archivo de ratings, el archivo de ratings contiene los siguientes campos: userId, movieId, rating y timestamp. Donde se eliminará el timestamp ya que no se necesitará para el recomendador.

```
complete_ratings_raw_data = sc.textFile('data/full/ratings.csv')
complete_ratings_raw_data_header = complete_ratings_raw_data.take(1)[0]

# Parse
complete_ratings_data = complete_ratings_raw_data \
    .filter(lambda line: line != complete_ratings_raw_data_header) \
    .map(lambda line: line.split(",")) \
    .map(lambda tokens:(int(tokens[0]),int(tokens[1]),float(tokens[2]))).cache()

print("Hay %s recomendaciones en el conjunto de datos completo" %
      (complete_ratings_data.count()))
-----
Hay 27753444 recomendaciones en el conjunto de datos completo
```

Hacemos lo mismo para el archivo de movies, el cual tiene los siguientes campos movieId, title y genres, donde se eliminará el campo de genres, ya que no se utilizara para el recomendador.

```
complete_movies_raw_data = sc.textFile('data/full/movies.csv')
complete_movies_raw_data_header = complete_movies_raw_data.take(1)[0]

# Parse
complete_movies_data = complete_movies_raw_data \
    .filter(lambda line: line!=complete_movies_raw_data_header) \
```

```

.map(lambda line: line.split(",")) \
.map(lambda tokens: (int(tokens[0]),tokens[1],tokens[2])).cache()

complete_movies_titles = complete_movies_data.map(lambda x: (int(x[0]),x[1]))

print("Hay %s películas en el conjunto de datos completo" %
      (complete_movies_titles.count()))
-----
Hay 58098 películas en el conjunto de datos completo

```

Lo siguiente será entrenar el conjunto de datos utilizando ALS (Alternating Least Squares) que se encuentra en la biblioteca MLLIB de Spark.

El conjunto de datos se separaran en entrenamiento y en prueba, 70% y 30% respectivamente. Utilizando la función train, se entrenará el modelo para el conjunto de datos. Una vez el modelo termine de entrenar, se probará la precisión del modelo usando los datos de prueba para ello se usará RMSE para ver el error cometido por el modelo.

```

from pyspark.mllib.recommendation import ALS
import math

iterations = 10
regularization_parameter = 0.1
best_rank = 4

training_RDD, test_RDD = complete_ratings_data.randomSplit([7, 3])

complete_model = ALS.train(training_RDD, best_rank, iterations=iterations,
                             lambda_=regularization_parameter)

test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))

predictions = complete_model.predictAll(test_for_predict_RDD) \
    .map(lambda r: ((r[0], r[1]), r[2]))
rates_and_preds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])),
      float(r[2]))).join(predictions)
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print( 'El error para los datos de prueba con RMSE es %s' % (error))
-----
El error para los datos de prueba con RMSE es 0.8302071628940553

```

Otra cosa que hay que tener en cuenta es dar recomendaciones de películas con un número mínimo de calificaciones. Para eso, necesitamos contar el número de clasificaciones por película realizadas por los usuarios.

```
def get_counts_and_averages(ID_and_ratings_tuple):
    nratings = len(ID_and_ratings_tuple[1])
    return ID_and_ratings_tuple[0], (nratings, float(sum(x for x in
        ID_and_ratings_tuple[1]))/nratings)

movie_ID_with_ratings_RDD = (complete_ratings_data.map(lambda x: (x[1],
    x[2])).groupByKey())

movie_ID_with_avg_ratings_RDD = movie_ID_with_ratings_RDD \
    .map(get_counts_and_averages)

movie_rating_counts_RDD = movie_ID_with_avg_ratings_RDD.map(lambda x: (x[0],
    x[1][0]))
```

Lo siguiente será hacer una recomendación de películas a un nuevo usuario, por lo tanto, se creará un conjunto de películas valoradas.

```
new_user_ID = 0

# The format of each line is (userID, movieID, rating)
new_user_ratings = [
    (0,260,4), # Star Wars: Episode IV - A New Hope (1977)
    (0,1196,4), # Star Wars: Episode V - The Empire Strikes Back (1980)
    (0,1210,4), # Star Wars: Episode VI - Return of the Jedi (1983)
    (0,88140,3), # Captain America: The First Avenger (2011)
    (0,110102,3), # Captain America: The Winter Soldier (2014)
    (0,122920,4), # Captain America: Civil War (2016)
    (0,122892,3), # Avengers: Age of Ultron (2015)
    (0,122910,4), # Avengers: Infinity War - Part I (2018)
    (0,89745,4), # Avengers, The (2012)
    (0,122916,3) # Thor: Ragnarok (2017)
]
new_user_ratings_RDD = sc.parallelize(new_user_ratings)
print( 'Ratings del nuevo usuario: %s' % new_user_ratings_RDD.take(10))
-----
Ratings del nuevo usuario: [(0, 260, 4), (0, 1196, 4), (0, 1210, 4), (0, 88140,
3), (0, 110102, 3), (0, 122920, 4), (0, 122892, 3), (0, 122910, 4), (0, 89745,
4), (0, 122916, 3)]
```

Una vez que tengamos las puntuaciones del nuevo usuario, las uniremos con el conjunto de datos completo para reentrenar el modelo con las nuevas valoraciones.

```
complete_data_with_new_ratings_RDD = complete_ratings_data \
    .union(new_user_ratings_RDD)

new_ratings_model = ALS.train(complete_data_with_new_ratings_RDD, best_rank,
    iterations=iterations, lambda_=regularization_parameter)
```

Cuando se termine de entrenar, ya se pueden predecir los valores de las películas que el nuevo usuario no tiene valoradas.

```
new_user_ratings_ids = map(lambda x: x[1], new_user_ratings)

new_user_unrated_movies_RDD = (complete_movies_data
    .filter(lambda x: x[0] not in new_user_ratings_ids).map(lambda
    x:(new_user_ID, x[0])))

new_user_recommendations_RDD = new_ratings_model
    .predictAll(new_user_unrated_movies_RDD)
```

Una vez que se tienen las valoraciones, se buscan los títulos de las películas verificando que cumpla el mínimo de los votos permitido por película, y se ordenan las películas por la valoración. Por último se da un top de las películas que pueden gustarle al nuevo usuario.

```
new_user_recommendations_rating_RDD = new_user_recommendations_RDD.map(lambda x:
    (x.product, x.rating))
new_user_recommendations_rating_title_and_count_RDD=
new_user_recommendations_rating_RDD
    .join(complete_movies_titles)
    .join(movie_rating_counts_RDD)

new_user_recommendations_rating_title_and_count_RDD = \
    new_user_recommendations_rating_title_and_count_RDD.map(lambda r:
        (r[1][0][1], r[1][0][0], r[1][1]))

top_movies = new_user_recommendations_rating_title_and_count_RDD
    .filter(lambda r: r[2]>=25).takeOrdered(10, key=lambda x: -x[1])

print('TOP 10 de películas recomendadas:\n%s' %
    '\n'.join(map(str, top_movies)))
-----
TOP 10 de películas recomendadas:
```

```
('Louis C.K.: One Night Stand (2005)', 3.993158993606868, 38)
('DMB (2000)', 3.9907390325178205, 29)
('Hitman Hart: Wrestling with Shadows (1998)', 3.9807323152545466, 33)
('The Second Renaissance Part II (2003)', 3.9279417381965596, 61)
('Down House (2001)', 3.921930985056261, 26)
('Monty Python's Fliegender Zirkus (1971)', 3.9167775760983945, 28)
('Jimmy Carr: Comedian (2007)', 3.8873387247304603, 30)
('Saturday Night Live: The Best of Will Ferrell (2002)', 3.8810049334224272, 26)
('Rick and Morty: State of Georgia Vs. Denver Fenton Allen (2016)',
3.874377809692846, 32)
('Gurren Lagann: Childhood's End (Gekijô ban Tengen toppa guren ragan: Guren
hen) (2008)', 3.8451507719254225, 35)
```

## Conclusión

El uso de un sistema de recomendación necesita mucho poder de cómputo y Apache Spark es una buena opción para eso, ya que el modelo de un sistema de recomendación de filtro colaborativo necesita estarse actualizando con las nuevas valoraciones de los usuarios y Apache Spark al poder hacer los cálculos en paralelo con distintos clusters es una gran opción para las grandes empresas que necesitan realizar los cálculos en el menor tiempo posible. Utilizar la librería de MLLIB de Apache Spark hizo el proceso rápido, fácil y entendible, al tener una buena cantidad de funciones que facilitan a la hora de programar.

## Referencias

Mwiti, D. (2018). *How to build a Simple Recommender System in Python*. [online] Towards Data Science. Available at: <https://towardsdatascience.com/how-to-build-a-simple-recommender-system-in-python-375093c3fb7d> [Accessed 4 Dec. 2018].

Dianes, J. (2018). *Building a Movie Recommendation Service with Apache Spark | Codementor*. [online] Codementor.io. Available at: <https://www.codementor.io/jadianes/building-a-recommender-with-apache-spark-python-example-app-part1-du1083qbw> [Accessed 4 Dec. 2018].

Spark.apache.org. (2018). *Collaborative Filtering - RDD-based API - Spark 2.1.0 Documentation*. [online] Available at: <https://spark.apache.org/docs/2.1.0/mllib-collaborative-filtering.html> [Accessed 4 Dec. 2018].