Apache Spark - Machine Learning

Competencias

- Conocerá los casos de uso de la librería MLlib y ML.
- Conocerá los principales tipos de datos asociados para trabajar con Machine Learning en Spark.
- Implementará algoritmos de Machine Learning en Spark.
- Aprenderá a generar búsqueda de hiper parámetros vía grilla en Spark.

Motivación

Hasta el momento, ya entendemos cómo funciona a grandes rasgos la API de SparkEn esta sesión aprenderemos a implementar modelos de recomendación basados en la factorización de matrices con PySpark. La implementación se realiza mediante el algoritmo Alternating Least Squares para generar recomendaciones en un conjunto de usuarios e ítems.

De manera adicional, en esta lectura se aprenderá a implementar búsqueda de hiper parámetros vía grilla cruzada y se entregarán lineamientos para la creación de pipelines.

Una breve introducción a los sistemas de recomendación

El objetivo de los sistemas de recomendación es generar sugerencias a un usuario en base a su comportamiento pasado. A grandes rasgos existen dos tipos de sistemas de recomendación: aquellos basados en el contenido generado por los usuarios, y aquellos basados en el filtro colaborativo entre pares. Este último se puede subdividir en otras dos categorías: aquellos basados en modelos estadísticos y aquellos basados en técnicas aglomerativas. Los modelos basados en estadística buscan aprender representaciones latentes sobre los atributos asociados a nivel de usuario e ítem. En contraste, los métodos basados en técnicas aglomerativas utilizan ratings a nivel de usuario e item almacenados en el sistema para predecir ratings.

El principal objetivo establecido es generar predicciones correctas de los ratings a nivel de usuario e ítem. La manera más común de implementar un sistema es representar los datos como una matriz de datos de rating a nivel de usuario e ítem con forma $R \in \Re^{u0+i0}$, donde $u(0)$ representa el total de usuarios e $i(0)$ representa el total de ítems. Por lo general esta matrix R tenderá a presentar una alta cantidad de valores perdidos, por lo que su representación más natural será mediante una matriz dispersa. Un sistema de recomendación buscará predecir los ratings faltantes en esta matriz dispersa y generará una recomendación.

Alternating Least Squares

Spark implementa un algoritmo de recomendación en atributos explícitos conocido como **Alternating Least Squares (ALS)**. Este algoritmo busca encontrar un vector de atributos con k dimensiones para cada usuario e ítem, para posteriormente reconstruir una matriz de ratings. Para obtener esta nueva matriz de ratings, ALS implementa métodos de factorización de matrices para reducir las dimensiones. Mediante esta reducción, vamos a generar un modelo que permita generar una rotación del sistema de coordenadas de manera tal de reducir las correlaciones entre dimensiones. Una vez que ya tenemos las correlaciones canceladas y nuestra representación de los datos es la adecuada, es posible reexpresar las observaciones faltantes.

ALS necesita de un conjunto de ratings existentes entre usuario e ítem con una estructura similar a la siguiente:

	Manzana	**Pera**	**Lechuga**	**Tomate**	**Durazno**	**Cebolla**	**Palta**
u_1	5	4	4			3	2
u_2	3	5		1	3		1
u_3			3				1
u_4		5			5		
u_5	2					3	5

Tabla 1. Estructura de ejemplo.

A grandes rasgos el algoritmo implementa la factorización de la matriz entre usuarios e ítems. En este proceso, vamos a fijar un número relativamente bajo $k \approx 10$ y resumir el comportamiento de cada usuario i con un vector $k-dimensional \quad xi$ y cada ítem j con un vector $k-dimensional \quad yi$. Estos vectores xi e yi se conocen como factores. En base a estos podemos predecir el ranking específico como $rij \approx xi^T y_j$.

Evaluación del método Alternating Least Squares

El objetivo es desarrollar una matriz completa de rankings $R \approx X^T$ mediante la estimación del producto punto entre ambos vectores. Este problema se puede formular como una optimización donde minimizamos una función objetivo y encontraremos un X e Y óptimos. Para evitar sobreajuste en las predicciones, Alternating Least Square implementa un parámetro de regularización en norma ℓ_2 en los parámetros.

$$f[i] = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{j \in N_{(i)}} (r_{ij} - w^{\mathsf{T}} f[j])^2 + \lambda \|w\|_2^2$$

Por lo general la matriz de datos dispersa se divide en dos muestras de entrenamiento y validación, donde en el conjunto de entrenamiento se entrenan los modelos y calibran los hiper parámetros, y el conjunto de validación sirve para evaluar el desempeño mediante métricas. Asumamos que tenemos una matriz dispersa predicha $\widehat{R} \in \Re^{u0+i0}$, las métricas de evaluación tendrán forma:

- Raíz del error cuadrático promedio.

$$\text{RMSE} = \sqrt{\frac{\sum_{(u,i) \in \mathcal{R}_{\text{test}}} \left(\mathbf{R}_{u,i} - \widehat{\mathbf{R}_{u,i}} \right)^2}{\left| \mathcal{R}_{\text{test}} \right|}}$$

- Media del error absoluto.

$$\text{MAE} = \frac{\sum_{(u,i) \in \mathcal{R}_{\text{test}}} \left| \mathbf{R}_{u,i} - \widehat{\mathbf{R}_{u,i}} \right|}{\left| \mathcal{R}_{\text{test}} \right|}$$

Si bien se puede conceptualizar el problema al nivel de una métrica continua generalizable, también se puede generar soluciones localizadas a un puntaje de rating específico. Ante este tipo de situaciones, por lo general preferiremos utilizar métricas asociadas a un problema de clasificación, como recall y precisión. Su conceptualización en este tipo de problemas se resume a:

- Recall en los top k sugerencias:

$$\text{Recall}(k) = \frac{\sum_u N(k,u)}{\sum_u N(u)}$$

- Precisión en los top k sugerencias para usuario u:

$$\text{Precision}(k,u) = \frac{N(k,u)}{k}$$

- Precisión generalizada al conjunto de validación:

$$\text{Precision} = \frac{1}{u_0}\sum_u \frac{N(k,u)}{k}$$

Implementando un algoritmo de recomendación con `pyspark.ml.recommendation.ALS`

Comencemos por incorporar nuestro objeto `SparkSession` para poder trabajar con la base de datos `MovieLens`. De manera adicional vamos a incorporar el módulo para implementar Alternating Least Squares con `pyspark.ml.recommendation.ALS`. Una vez que creamos nuestra conexión e importamos la librería a utilizar, podemos procesar los datos.

```
from pyspark.sql import SparkSession
from pyspark.ml.recommendation import ALS, ALSModel
```

```
# crear objeto SparkSession
spark = SparkSession\
          .builder\
          .master('yarn')\
          .appName('u5lec2')\
          .enableHiveSupport()\
          .getOrCreate()
```

Nuestros datos se van a contener en un archivo `csv`, para lo cual utilizaremos el método `.read.csv` dentro de nuestra conexión con `SparkSession`. Para generar la separación y coerción correcta de cada dato del registro, vamos a implementar una pequeña variante de la lógica de RDD, orientada a objetos `pyspark.sql.dataframe.DataFrame`. Si solicitamos un registro a nuestro objeto, veremos que está correctamente separado, pero sin etiquetas a nivel de campo. De manera adicional, se aprecia que todos los datos ingresados tienen el tipo de dato string.

```
movies_rating =
spark.read.csv('s3://bigdata-desafio/challenges/u3act1/ratings.csv')
```

```
movies_rating.take(1)
```

```
[Row(_c0='1', _c1='2', _c2='3.5', _c3='1112486027')]
```

Así, nuestro objetivo va a ser asignar un nombre a cada campo y corregir su tipo de dato. Para ello vamos a utilizar en nuestro `DataFrame` la función `.selectExpr` que permite proyectar expresiones de SQL/HQL en un conjunto de datos y devolver otro objeto `DataFrame`. Dentro de esta expresión, pasaremos un string con la instrucción cast de `Hive`, que cambiará el tipo de dato, y posteriormente la renombraremos. Esta operación la repetiremos para cada una de las columnas en nuestro DataFrame. El objeto `process_movies_rating` presentará una estructura de datos adecuada para trabajar.

```
process_movies_rating = movies_rating\
                      .selectExpr(
                       "cast(_c0 as int) as user_id",
                       "cast(_c1 as int) as movie_id",
                       "cast(_c2 as float) as rating",
                       "cast(_c3 as long) as timestamp")
```

```
process_movies_rating.take(1)
```

```
[Row(user_id=1, movie_id=2, rating=3.5, timestamp=1112486027)]
```

La implementación de Alternating Least Squares no difiere del flujo clásico de Machine Learning, por lo que el siguiente paso es la división en conjuntos de entrenamiento y validación. Este punto lo lograremos con el método `randomSplit` dentro de nuestro `DataFrame` ya procesado. Posterior a la división de muestras, vamos a instanciar nuestro modelo ALS. Observamos que nuestros datos disponibles corresponden a 20 millones de observaciones, divididas en un esquema .7/.3, teniendo 14 millones de observaciones para entrenar nuestro modelo y 5 millones para conjunto de validación.

```
train_vector, test_vector = process_movies_rating.randomSplit([0.7,
0.3])
```

```
print("Train sample size: ", train_vector.count())
print("Test sample size: ", test_vector.count())
```

```
Train sample size:  14001610
Test sample size:  5998653
```

El primer punto a considerar es la cantidad de recomendaciones que deseamos generar. Este parámetro se gobierna con `rank`, que determinará la dimensionalidad de los vectores de atributos a aprender por la máquina. También debemos incorporar una serie de etiquetas para que Spark pueda identificarlas dentro de los objetos separados. En específico debemos declarar la columna con los id de los usuarios (`userCol`), los items (`itemCol`) y el rating del ítem por el usuario (`ratingCol`). Cabe destacar que la clase también permite una serie de hiper parámetros (en específico, el coeficiente de regularización) que hablaremos después.

Un problema que hemos abordado de manera tangencial es la dispersión de la matriz de datos. Esto es problemático dado que el algoritmo tenderá a favorecer ítems que presentan una mayor tasa de ocurrencia o que presentan mayor grado de información. Si eventualmente ingresamos un nuevo producto al sistema, el algoritmo no lo recomendará. Este problema se conoce como **Cold Start Problem**. `pyspark.ml.recommendation.ALS` implementa dos variantes para solucionar este problema: `'drop'` que eliminará los registros vacíos, generando una matriz densa; y `'nan'`, que permitirá poblar las celdas vacías con objetos `np.nan`.

```
alternate_ls_model = ALS(rank = 5,
                         userCol='user_id',
                         itemCol='movie_id',
                         ratingCol='rating',
                         coldStartStrategy='drop')
```

Para entrenar el modelo, debemos implementar el método `fit` en la instancia ya creada. Recuerden siempre asignar el `fit` en un nuevo objeto. En este caso, el modelo ya entrenado se encontrará en el objeto `train_als_model`. Este nuevo objeto tendrá información relevante sobre cómo se generó la matriz de recomendación en los objetos `train_als_model.userFactors` y `train_als_model.itemFactors`. La función `get_als_factors_information` nos permitirá identificar con cuántos individuos estamos trabajando y cuántos items fueron evaluados. Para evitar problemas en la integración de scripts externos dentro de nuestro kernel de `PySpark3`, implementaremos la función de manera manual.

```
# entrenamos el modelo
train_als_model = alternate_ls_model.fit(train_vector)
```

```
def get_als_factors_information(model, n_users = 3, n_items = 10):
    """Report summary on factor loadings at item and user level

    :model: An pyspark.ml.recommendation.ALSModel object.
    :n_users: An integer. Quantity of users to report.
    :n_items: An integer. Quantity of items to report.
    :returns: A printed summary.

    """
    if isinstance(model, ALSModel) or isinstance(model, ALS):
        tmp_user_factors, tmp_item_factors = model.userFactors,
model.itemFactors
        print(f"Número de usuarios en entrenamiento:
{tmp_user_factors.count()}")
        print(f"Producto punto para los primeros {n_users} usuarios:")

        for i in tmp_user_factors.take(n_users):
            print(f"Usuario: {i.id} -> Producto punto: {[round(j, 3) for
j in i.features]}" )

        print("\n")
        print(f"Número de items en entrenamiento:
{tmp_item_factors.count()}")
        print(f"Producto punto para los primeros {n_items} items:")

        for i in tmp_item_factors.take(n_items):
            print(f"Item: {i.id} -> Producto punto: {[round(j, 3) for j
in i.features]}")
```

Mediante la función `get_als_factors_information`, observamos que el modelo se entrenó con 138 mil usuarios que evaluaron 25 mil películas, lo cual no es una cantidad despreciable de datos. Dentro de cada usuario e ítem, vamos a extraer los productos punto existentes a nivel de factor latente solicitado. Dado que solicitamos 5 factores latentes en nuestro modelo con el método `rank`, los factores representados serán 5.

```
get_als_factors_information(train_als_model)
```

```
Número de usuarios en entrenamiento: 138493
Producto punto para los primeros 3 usuarios:
Usuario: 10 -> Producto punto: [-0.545, 0.918, 0.673, -0.767, -1.189]
Usuario: 20 -> Producto punto: [-1.147, 0.526, 0.104, -0.304, -1.524]
Usuario: 30 -> Producto punto: [-0.672, 0.927, 0.606, -0.564, -0.958]
```

```
    Número de items en entrenamiento: 25287
    Producto punto para los primeros 10 items:
    Item: 10 -> Producto punto: [-0.45, 0.358, 0.309, -1.075, -1.289]
    Item: 20 -> Producto punto: [0.097, 0.511, -0.026, -0.829, -1.258]
    Item: 30 -> Producto punto: [-0.905, 1.143, 0.55, -0.464, -1.049]
    Item: 40 -> Producto punto: [-0.432, 1.28, 0.775, -0.57, -1.031]
    Item: 50 -> Producto punto: [-0.914, 0.527, 0.814, -0.639, -1.706]
    Item: 60 -> Producto punto: [-0.172, 1.042, 0.527, -0.89, -0.837]
    Item: 70 -> Producto punto: [-0.843, 0.391, -0.206, -0.584, -1.456]
    Item: 80 -> Producto punto: [-1.022, 1.307, 0.58, -0.479, -0.883]
    Item: 90 -> Producto punto: [-0.54, 1.218, 0.212, -0.527, -1.071]
    Item: 100 -> Producto punto: [-0.303, 0.621, 0.477, -0.669, -1.252]
```

Ahora podemos darle la principal utilidad a nuestro modelo Alternating Least Squares: **Hacer predicciones y dar recomendaciones**. Dado que el objetivo de generar representaciones aproximadas para celdas no existentes, se puede considerar una operación de transformación. Para ello vamos a utilizar el método `transform` y generar predicciones en nuestro conjunto de validación. Como siempre, asegúrense de generar un nuevo objeto para esto.

```
get_predictions = train_als_model.transform(test_vector)
```

Al implementar el método `transform`, obtendremos un nuevo objeto `pyspark.sql.dataframe.DataFrame` con todas las columnas originales, más una columna con los rating predichos para cada combinación entre usuario y película. Supongamos que estamos interesados en ver cuál es el comportamiento de nuestro modelo para predecir el rating en todas las películas evaluadas por el usuario 7. A grandes rasgos observamos que el modelo tiende a predecir de mejor manera el rating cuando éste es bajo. El modelo tiende a producir predicciones bajas por el hecho que se ha entrenado de manera substancial con ratings entre 2 y 3. Podemos corroborar esto preguntando por la frecuencia de ocurrencia de cada clasificación.

```
get_predictions.filter(get_predictions.user_id == 7).show()
```

```
+-------+--------+------+----------+----------+
|user_id|movie_id|rating| timestamp|prediction|
+-------+--------+------+----------+----------+
|      7|    3179|   5.0|1011208971| 3.0632024|
|      7|    1270|   4.0|1011206698| 3.8164206|
|      7|     362|   3.0|1011208344| 3.2676897|
|      7|    3844|   3.0|1011206022| 3.2661734|
|      7|    3235|   3.0|1011205675| 2.5607698|
|      7|    2018|   4.0|1011205698| 3.2967641|
|      7|    1307|   3.0|1011207700| 3.5569854|
|      7|    2752|   4.0|1011205422|  2.895252|
|      7|     271|   3.0|1011206217| 3.0013263|
|      7|    3528|   2.0|1011208366| 2.9065127|
|      7|    1210|   5.0|1011204654| 3.9004083|
|      7|    4317|   3.0|1011208675| 2.7553058|
|      7|    3037|   4.0|1011206346| 3.4426894|
|      7|    1077|   5.0|1011206898|   3.13728|
|      7|    1777|   4.0|1011207938| 3.2652586|
|      7|    3108|   3.0|1011208263| 3.0538998|
|      7|     122|   2.0|1011208569|  2.746234|
|      7|    4018|   3.0|1011208220| 3.2431793|
|      7|    1206|   2.0|1011206732| 2.9121718|
|      7|     355|   3.0|1011205262| 2.5143986|
+-------+--------+------+----------+----------+
only showing top 20 rows
```

```
# verificamos la frecuencia de rating
train_vector\
    .groupBy('rating')\
    .agg({'rating':'count'})\
    .show()
```

```
+------+-------------+
|rating|count(rating)|
+------+-------------+
|   5.0|      2030119|
|   2.5|       618384|
|   2.0|      1001786|
|   3.0|      3003279|
|   1.5|       195408|
|   0.5|       167407|
|   3.5|      1540024|
|   1.0|       476635|
|   4.5|      1074700|
|   4.0|      3893868|
+------+-------------+
```

Resulta que el algoritmo Alternating Least Squares se puede entender como un problema de regresión. Así, podemos evaluar el comportamiento predictivo del modelo mediante el método `pyspark.ml.evaluation.RegressionEvaluator`. Si bien la naturaleza ordinal de los ratings sugiere la implementación de métricas más sofisticadas, métricas como el RMSE tienden a ser buenos puntos de partida. `RegressionEvaluator` funciona de manera similar a como se construye un modelo en `pyspark.ml`. Debemos definir cuáles son las columnas con las predicciones y puntajes verdaderos en el `DataFrame` de las predicciones. De manera adicional, podemos definir el tipo de métrica, en este caso el RMSE. En este caso, el modelo tiende a sobreestimar las predicciones en aproximadamente .8 puntos en la escala del RMSE.

```
from pyspark.ml.evaluation import RegressionEvaluator

evaluate_als =
RegressionEvaluator(predictionCol='prediction',labelCol='rating',
metricName='rmse')
get_rmse = evaluate_als.evaluate(get_predictions)
print(f"El RMSE promedio en el conjunto de entrenamiento es de:
{get_rmse}")
```

```
El RMSE promedio en el conjunto de entrenamiento es de:
0.816026170246165
```

De manera adicional, `pyspark` ofrece el método `pyspark.mllib.evaluation.RegressionMetrics` que contiene más métricas de evaluación. El método requiere de un RDD entre lo observado y lo predicho, para lo cual vamos a generar un nuevo objeto llamado `user_level_prediction` que preserve tuplas a nivel de registro con la observación exacta y la predicha. Posterior a eso, podemos pasar este RDD en el método.

```
from pyspark.mllib.evaluation import RegressionMetrics

user_level_prediction = get_predictions\
                        .select('prediction','rating')\
                        .rdd\
                        .map(lambda x: (x[0], x[1]))
regression_metrics_output = RegressionMetrics(user_level_prediction)
```

El objeto creado generará los reportes para la varianza explicada (con la instancia `.explainedVariance`), el error cuadrático promedio (con la instancia `.meanSquaredError`), el error absoluto promedio (con la instancia `.meanAbsoluteError`) y la raíz del error cuadrático promedio (con la instancia `.rootMeanSquaredError`). Podemos extraer todas estas métricas con la función `report_reg_metrics` definida a continuación.

Observamos que el modelo generado explica un .4 de la varianza generada dentro del modelo. De manera adicional, el modelo tiende a sobreestimar en aproximadamente .6 puntos los ratings de cualquier usuario para cualquier película. Este hallazgo se apoya por los valores para el Error Cuadrático Promedio y la Raíz del Error cuadrático promedio.

```
def report_reg_metrics(metrics):
    """Report metrics from a RegressionMetrics object

    :metrics: a RegressionMetrics object
    :returns: a printed report.

    """

    if isinstance(metrics, RegressionMetrics) is True:
        print(f"Varianza Explicada: {round(metrics.explainedVariance,
3)}")
        print(f"Error cuadrático promedio:
{round(metrics.meanSquaredError, 3)}")
        print(f"Error absoluto promedio:
{round(metrics.meanAbsoluteError, 3)}")
        print(f"Raíz del error cuadrático promedio:
{round(metrics.rootMeanSquaredError, 3)}")
    else:
        raise TypeError("metrics argument must be a
pyspark.mllib.evaluation.RegressionMetrics object.")
```

```
report_reg_metrics(regression_metrics_output)
```

```
Varianza Explicada: 0.413
Error cuadrático promedio: 0.666
Error absoluto promedio: 0.633
Raíz del error cuadrático promedio: 0.816
```

Calibración de hiper parámetros

Ahora generemos una búsqueda de hiper parámetros para ver si podemos mejorar las predicciones/recomendaciones de nuestro modelo. Para poder implementar el método `pyspark.ml.tuning.CrossValidator`, deberemos generar tres instancias de modelo. Una correspondiente al modelo en sí, que estará en el objeto `als_instance`; una instancia que represente el tipo de evaluador, que estará en el objeto `reg_eval`; y una instancia correspondiente a la grilla de hiper parámetros, que estará en el objeto `hyperparam_grid`.

Existen tres hiper parámetros en el algoritmo Alternating Least Squares. El primero es el rango (`rank`) que hace referencia a la cantidad de factores a extraer a nivel de matriz de usuario e ítem. Se puede entender como un análogo al parámetro `n_components` existente en el método `sklearn.decomposition.PCA`. El segundo corresponde a la regularización (`regParam`) que debe implementar el modelo, la cual corresponde a la norma ℓ_2 (también conocida como regularización de Tikhonov). Mediante la regularización vamos a evitar la prevalencia de ciertas cargas en los factores inferidos a nivel de items o usuarios.

Toda tarea de Machine Learning para presentar un modelo correcto se realiza mediante el desarrollo de la calibración de hiper parámetros. El proceso de calibración en Spark se puede implementar para algún modelo específico o para Pipelines de múltiples pasos, afectando algoritmos, proceso de ingeniería de atributos entre otros.

MLlib genera calibración de modelos mediante herramientas como `CrossValidator` y `TrainValidationSplit`, que necesitan por lo bajo de los siguientes elementos:

- Un estimador.
- Una grilla de valores
- Un evaluador.

El proceso de implementación de este modelo es:

- Generar una división de los datos en conjuntos de entrenamiento y validación.

- Para cada par de (`entrenamiento`, `validación`), se iteran los siguientes pasos:
    - Para cada grilla de valores generadas con `ParamMaps`, se implementa un `Estimador` con la combinación específica de hiper parámetros, y evaluamos el desempeño del modelo utilizando un Evaluador.

- Seleccionamos el mejor modelo producido.

El método evaluador puede ser alguna variante de los métodos disponibles en el módulo `pyspark.ml.evaluation` como `RegressionEvaluator` para problemas de regresión, `BinaryClassificationEvaluator` para problemas de clasificación binaria y `MulticlassClassificationEvaluator` para problemas de clasificación multiclase. Por defecto la métrica por defecto implementada para elegir la mejor combinación de hiper parámetros en `ParamMap` se puede sobreescribir mediante el método `metricsName` en cada evaluador específico.

Digresión: Sobre las variantes de validación cruzada en `pyspark`

Método CrossValidation

El método `pyspark.ml.tuning.CrossValidator` permite realizar el proceso iterativo de implementar un mismo algoritmo en un conjunto de datos, donde se realizarán k particiones internas donde cada observación podrá ser parte de un conjunto de entrenamiento y validación. Supongamos que estamos implementando un `CrossValidator` con $k = 5$, esto significa que generaremos 5 tuplas de objetos (`entrenamiento`, `validación`). En base a una configuración específica de hiper parámetros especificados en el `ParamMap`, el método `CrossValidator` calculará el promedio de la métrica en base a los resultados de cada Modelo.

Método TrainValidation

El método `pyspark.ml.tuning.TrainValidationSplit` permite realizar el proceso de búsqueda de grilla de hiper parámetros, pero sin la fase de validación cruzada en cada combinación de hiper parámetros definidos en el `ParamMap`. De esta manera, esta es una opción computacionalmente eficiente cuando deseamos evaluar desempeño en conjuntos de datos grandes. Ante el caso de tener pocos casos, por lo general no entregará resultados confiables. Con este método, se genera solo una división de muestra entre (`entrenamiento`, `validación`).

Construyamos nuestro método para realizar una búsqueda de grilla con este algoritmo. Por motivos pedagógicos, vamos a importar de manera explícita cada elemento a utilizar. Partamos por generar una nueva instancia del algoritmo ALS. En este vamos a declarar los elementos de usuario, item, rating y estrategia de manejo ante el problema de Cold Start. El segundo elemento es la definición de la grilla de hiper parámetros. Acá el comportamiento dista de la implementación ya conocida en `scikit-learn`.

Debemos generar un nuevo objeto con el método `ParamGridBuilder`, donde concatenamos el rango de hiper parámetros con la opción `.addGrid`. Dentro de `.addGrid` debemos incorporar el acceso específico al hiper parámetro, y el segundo parámetro que corresponderá al rango de valores. En esta etapa podemos agregar tanto hiper parámetro y rango que deseemos. Finalmente, debemos construir el objeto con la opción `.build`.

Por último, vamos a crear un objeto evaluador con el método `RegressionEvaluator`, donde debemos indicarle al modelo cómo se llamará la columna con los predichos, y la columna con los datos observados. Terminamos declarando el tipo de métrica a incorporar con el argumento `metricName`.

Con estos elementos definidos, podemos ingresarlos en el método `CrossValidator` para generar una búsqueda de grilla. El último elemento a considerar es la cantidad de validaciones cruzadas a incorporar. Por motivos expositivos y de brevedad en la ejecución, vamos a restringirla a 2 validaciones. Cabe destacar y recordar que en aplicaciones serias, este modelo debería tener entre 5 y 10 validaciones cruzadas para tener buenos resultados.

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS

# definimos el modelo
als_instance = ALS(userCol='user_id', itemCol='movie_id',
ratingCol='rating', coldStartStrategy='drop')

# definimos una grilla que tenga
hyperparam_grid = ParamGridBuilder()\
                    .addGrid(als_instance.rank,[5, 7, 10])\
                    .addGrid(als_instance.regParam, [0.01, 0.5, 1])\
                    .build()
# definimos el evaluador
reg_eval =
RegressionEvaluator(predictionCol='prediction',labelCol='rating',
metricName='rmse')
```

```
# definimos la búsqueda de grilla
grid_search_als = CrossValidator(estimator=als_instance,
                                 estimatorParamMaps = hyperparam_grid,
                                 evaluator = reg_eval,
                                 numFolds = 2)

# entrenamos el modelo
grid_search_als = grid_search_als.fit(train_vector)
```

Ya con nuestro modelo validado vía grilla, podemos extraer el mejor modelo candidato con la opción `grid_search_als.bestModel`. Con este modelo procederemos a generar predicciones válidas utilizando el método `transform`.

```
# preservamos el mejor modelo en la grilla
best_als_model = grid_search_als.bestModel
# generamos las predicciones en el conjunto de validación
get_predictions = best_als_model.transform(test_vector)
```

Volvamos a evaluar el comportamiento de este mejor modelo respecto a las métricas RMSE. Podemos evaluar que nuestro modelo calibrado con hiper parámetros no tuvo un gran desempeño en cuanto a RMSE. Para todo el tiempo que demoró en entrenarse, aparentemente el comportamiento es similar. Aún así, nuestro reporte de varianza explicada sugiere que este modelo mejoró en aproximadamente .9, lo cual es indicativo que aprendió más sobre el conjunto de datos.

```
evaluate_als =
RegressionEvaluator(predictionCol='prediction',labelCol='rating',
metricName='rmse')
get_rmse = evaluate_als.evaluate(get_predictions)
print(f"El RMSE promedio en el conjunto de entrenamiento es de:
{get_rmse}")
user_level_prediction = get_predictions\
                        .select('prediction','rating')\
                        .rdd\
                        .map(lambda x: (x[0], x[1]))
regression_metrics_output = RegressionMetrics(user_level_prediction)
```

```
El RMSE promedio en el conjunto de entrenamiento es de:
0.8182623700198136
```

```
user_level_prediction = get_predictions\
                        .select('prediction','rating')\
                        .rdd\
                        .map(lambda x: (x[0], x[1]))
regression_metrics_output = RegressionMetrics(user_level_prediction)
```

```
report_reg_metrics(regression_metrics_output)
```

```
Varianza Explicada: 0.503
Error cuadrático promedio: 0.67
Error absoluto promedio: 0.629
Raíz del error cuadrático promedio: 0.818
```

Asumamos por el momento que nuestro modelo es el óptimo, y aterricemos de lleno en las recomendaciones para usuarios. Para ello vamos a hacer uso del método `recommendForAllUsers` que nos permitirá generar un número finito de recomendaciones de películas para cada usuario. De manera adicional, podemos hacer uso del método `recommendForAllItems` para poder generar una lista de usuarios a los cuales recomendar una película en específico.

```
get_recommendations_for_users = best_als_model.recommendForAllUsers(10)
```

```
get_recommendations_for_users.show(10)
```

```
+-------+--------------------+
|user_id|     recommendations|
+-------+--------------------+
|    148|[[84792, 19.98748...|
|    463|[[109953, 12.9154...|
|    471|[[84792, 13.13536...|
|    496|[[74159, 11.16245...|
|    833|[[116821, 22.9646...|
|   1088|[[74159, 17.16883...|
|   1238|[[109953, 10.1759...|
|   1342|[[74159, 10.12761...|
|   1580|[[116821, 18.5193...|
|   1591|[[109953, 12.7542...|
+-------+--------------------+
only showing top 10 rows
```

```
get_recommendations_on_items = best_als_model.recommendForAllItems(10)
get_recommendations_on_items.show(10)
```

```
+--------+--------------------+
|movie_id|     recommendations|
+--------+--------------------+
|     148|[[65918, 5.645903...|
|     463|[[14403, 6.175496...|
|     471|[[123082, 6.37906...|
|     496|[[56145, 7.99611]...|
|     833|[[111596, 6.95661...|
|    1088|[[14403, 8.0037],...|
|    1238|[[61315, 7.368344...|
|    1342|[[65918, 7.997752...|
|    1580|[[108993, 5.74440...|
|    1591|[[96611, 6.117304...|
+--------+--------------------+
only showing top 10 rows
```

En ambos métodos, el punto en común es que devuelve tanto el `id` de la recomendación, así como un rating predicho que muchas veces puede estar fuera del rango plausible. En este ejemplo, observamos que existen recomendaciones cuyo puntaje predicho se encontrará más arriba del límite lógico de 5. Cabe destacar que independiente de lo poco plausible de la medida, es una buena aproximación para el caso de confianza asociada a la recomendación.

Nuestro siguiente objetivo es extraer sólo los `movie_id` de aquellos registros donde el rating de los usuarios haya sido igual o superior a 3.5. Vamos a segmentar nuestro objeto de predicciones `get_predictions` con la opción `where`. Posterior a eso, vamos a agrupar los registros a nivel de usuario y finalmente vamos a agregar y seleccionar sólo con los `movie_id` mediante el argumento `collect_set` de Hive.

```
recommend_most_likely_movies = get_predictions\
                                .where('rating > 3.5')\
                                .groupBy('user_id')\
                                .agg(expr('collect_set(movie_id)'))
```

```
recommend_most_likely_movies.show(10)
```

```
+-------+---------------------+
|user_id|collect_set(movie_id)|
+-------+---------------------+
|    148| [915, 916, 4016, ...|
|    463| [161, 45, 590, 39...|
|    471| [3296, 69849, 180...|
|    496| [1242, 2208, 590,...|
|    833| [350, 527, 165, 4...|
|   1088| [628, 1407, 376, ...|
|   1238| [931, 800, 2871, ...|
|   1342| [2019, 4226, 3552...|
|   1580| [69406, 2369, 231...|
|   1591| [356, 150, 1643, ...|
+-------+---------------------+
only showing top 10 rows
```

Ahora realicemos un ejercicio similar, y ordenemos los `movie_id` de manera descendente acorde a la predicción generada por el modelo. Para ello vamos a reordenar las columnas de usuario y predicción, siendo esta última la que comandará el orden. Posterior a esto, podemos seguir con los pasos de agrupar y agregar con `collect_list`.

```
from pyspark.sql.functions import col, expr

user_level_predictions = get_predictions\
                         .orderBy(col('user_id'), expr('prediction
DESC'))\
                         .groupBy('user_id')\
                         .agg(expr('collect_list(movie_id)'))
```

```
user_level_predictions.show(10)
+-------+----------------------+
|user_id|collect_list(movie_id)|
+-------+----------------------+
|    148|  [25, 163, 5171, 5...|
|    463|  [434, 10, 48, 349...|
|    471|  [231, 5419, 1388,...|
|    496|  [317, 95, 852, 19...|
|    833|  [410, 34, 595, 43...|
|   1088|  [102, 891, 1183, ...|
|   1238|  [2126, 1644, 648,...|
|   1342|  [1882, 1831, 849,...|
|   1580|  [858, 750, 1203, ...|
|   1591|  [1499, 1515, 1479...|
+-------+----------------------+
only showing top 10 rows
```

Evaluación de ALS mediante métricas de Ranking

Al inicio de la lectura señalamos el hecho de que existen dos modos de evaluar este tipo de modelos. La primera forma está orientada a evaluar el comportamiento macro del modelo, respecto a métricas continuas como el RMSE y el MAE (Error medio absoluto, no Error mediano absoluto). La segunda forma es más específica, orientada a evaluar las precisiones a nivel de rating específico. Para incorporar este tipo de métricas, vamos a importar el módulo `pyspark.mllib.evaluation.RankingMetrics`. En base a los dos objetos creados arriba (`recommend_most_likely_movies` y `user_level_predicttions`), vamos a generar un nuevo `DataFrame` que nos concatene las divergencias entre lo predicho y lo observado para los registros en el conjunto de evaluación.

Posterior a la creación de este objeto, podremos incorporarlo a nuestro método `RankingMetrics`. Con éste vamos a poder extraer una cantidad substancial de indicadores. El primero va a ser la precisión promedio del modelo con la opción `.meanAveragePrecision`. La métrica no es muy alentadora, dado que nos dice que hay una probabilidad 17% menor a un recomendador aleatorio de asignar un ranking correcto. Ahora, nuestro contrapunto a esta observación es el hecho que estamos intentando generar nuevas recomendaciones, no predecir el patrón específico.

```
from pyspark.mllib.evaluation import RankingMetrics

user_recommendation_divergence = recommend_most_likely_movies\
                                        .join(user_level_predictions,
['user_id'])\
                                        .rdd.map(lambda x: (x[1],
x[2][:15]))

rank_metrics = RankingMetrics(user_recommendation_divergence)
rank_metrics.meanAveragePrecision
```

```
0.3346879289508089
```

Veamos cómo se comporta el modelo a nivel específico de rating. Vamos a utilizar el método `precisionAt` que nos entregará una representación local de la precisión de las recomendaciones. Este método lo vamos a solicitar para los rating de 1 a 5. A grandes rasgos apreciamos que el comportamiento de las predicciones generadas por el modelo para cada categoría no son inferiores al 50. El segundo punto a considerar es el hecho que nuestro algoritmo recomendador tiende a generar mejores métricas cuando las recomendaciones son bajas. Así, el modelo tiende a tener un buen comportamiento con malas películas por sobre el resto.

```
for rating in range(1, 6):
    print(f"Recomendaciones correctas con {rating} estrella:
{rank_metrics.precisionAt(rating)}")
```

```
Recomendaciones correctas con 1 estrella: 0.5702889869386087
Recomendaciones correctas con 2 estrella: 0.5611759635332965
Recomendaciones correctas con 3 estrella: 0.5461336917667452
Recomendaciones correctas con 4 estrella: 0.5259036320603104
Recomendaciones correctas con 5 estrella: 0.5029833737544924
```

Vinculando las predicciones con el listado de películas

Por último, podemos solicitar los nombres de todas las películas recomendadas para un usuario en específico. Para ello vamos a ingestar el archivo `movies.csv` que se encuentra en el bucket S3 del curso. De similar manera, vamos a coercionar los datos para que tengan un tipo de dato válido y un nombre de columna informativa mediante el método `selectExpr`. El resultado mostrado con `show` nos entrega cómo se comportan los registros en esta nueva tabla.

```
movies_data =
spark.read.csv('s3://bigdata-desafio/challenges/u3act1/movies.csv')
```

```
movies_data = movies_data.selectExpr(
                "cast(_c0 as int) as movie_id",
                "cast(_c1 as string) as movie_name",
                "cast(_c2 as string) as tagged_genre")
```

```
movies_data.show()
```

```
+--------+--------------------+--------------------+
|movie_id|          movie_name|        tagged_genre|
+--------+--------------------+--------------------+
|       1|    Toy Story (1995)|Adventure|Animati...|
|       2|      Jumanji (1995)|Adventure|Childre...|
|       3|Grumpier Old Men ...|      Comedy|Romance|
|       4|Waiting to Exhale...|Comedy|Drama|Romance|
|       5|Father of the Bri...|              Comedy|
|       6|         Heat (1995)|Action|Crime|Thri...|
|       7|      Sabrina (1995)|      Comedy|Romance|
|       8| Tom and Huck (1995)|  Adventure|Children|
|       9| Sudden Death (1995)|              Action|
|      10|    GoldenEye (1995)|Action|Adventure|...|
|      11|American Presiden...|Comedy|Drama|Romance|
|      12|Dracula: Dead and...|       Comedy|Horror|
|      13|        Balto (1995)|Adventure|Animati...|
|      14|        Nixon (1995)|               Drama|
|      15|Cutthroat Island ...|Action|Adventure|...|
|      16|       Casino (1995)|         Crime|Drama|
|      17|Sense and Sensibi...|       Drama|Romance|
|      18|   Four Rooms (1995)|              Comedy|
|      19|Ace Ventura: When...|              Comedy|
|      20|  Money Train (1995)|Action|Comedy|Cri...|
+--------+--------------------+--------------------+
only showing top 20 rows
```

Para poder extraer los nombres de las películas recomendadas para un usuario, debemos separar los elementos conformantes. Esto lo logramos mediante la selección de todos los `movie_id` con el método `collect_list`. Posteriormente deberemos limpiar el registro específico, lo cual lograremos mediante el acceso a un elemento específico con `[0]` y posteriormente entrando a la columna con nombre `collect_list(movie_id)`. El resultado de este procedimiento será una lista con todos los `movie_id`.

Esta lista la pasaremos dentro del argumento `where`, donde preservaremos aquellos registros que sí contengan los valores contenidos en la lista. Posteriormente vamos a seleccionar sólo la columna `movie_name` de la tabla y la imprimiremos.

```
extract_user = user_level_predictions.select('collect_list(movie_id)').take(1)
```

```
clean_collect_list = extract_user[0]['collect_list(movie_id)']

movies_data\
    .where(movies_data.movie_id.isin(clean_collect_list))\
    .select('movie_name')\
    .show()
```

```
+--------------------+
|          movie_name|
+--------------------+
|Leaving Las Vegas...|
|     Clueless (1995)|
| Bed of Roses (1996)|
|    Desperado (1995)|
|Circle of Friends...|
|Don Juan DeMarco ...|
|         I.Q. (1994)|
|  Love Affair (1994)|
|     Only You (1994)|
|North by Northwes...|
|   Casablanca (1942)|
|      Sabrina (1954)|
|Roman Holiday (1953)|
|To Catch a Thief ...|
|Everyone Says I L...|
|William Shakespea...|
|Strictly Ballroom...|
|Star Wars: Episod...|
|    Manhattan (1979)|
|Room with a View,...|
+--------------------+
only showing top 20 rows
```

Aspectos adicionales

Implementación de Pipelines

En la versión actual de MLlib, la API hereda buenas prácticas de `scikit-learn` para combinar múltiples algoritmos en un flujo de trabajo único. Existen una serie de conceptos asociados al proceso de Pipeline, los cuales ya hemos visto anteriormente:

- **`DataFrame`**: La API hace uso de una estructura de datos `pyspark.sql.dataframe.DataFrame` para contener múltiples formas de datos.

- **`Transformer`**: Un Transformer es un algoritmo que permite re-expresar un `pyspark.sql.dataframe.DataFrame` en otro objeto `DataFrame`. Por ejemplo, un modelo `pyspark.ml` es un transformador dado que permite reexpresar un `DataFrame` en otro con una columna de predicciones.

- **`Estimator`**: Un Estimator es un algoritmo que puede ser ajustado en un `DataFrame` para generar un transformador. Por ejemplo, un algoritmo de aprendizaje que retorna un Modelo que se entrena en un conjunto de Datos y genera otro conjunto de datos.

- **`Pipeline`**: Un Pipeline concatena múltiples Transformadores y Estimadores en un flujo de trabajo.

Un pipeline está especificado como una secuencia de etapas, donde cada etapa puede ser un transformador o un estimador. Estas etapas se ejecutan de manera secuencia, y el `DataFrame` que se consume se transforma en la medida que pasa por cada etapa. Para las etapas de Transformación, se ejecuta el método `transform` en cada `DataFrame`. Para los Estimadores, se implementa el método `fit` que se llama para producir un método `Transformer` (que posteriormente se incluye como una etapa dentro de nuestro pipeline).

Pipelines y Grafos Acíclicos Dirigidos

Por defecto, el comportamiento de los Pipelines es una secuencia lineal. Resulta que es posible crear Pipelines que operen de forma intuitiva con estructuras de Grafos Acíclicos Dirigidos. Para trabajar con un Pipeline desde este flujo, es necesario especificar todos los inputs y outputs de cada etapa en el Pipeline. Una vez que se definen los vértices, es necesario generar las conexiones mediante orden topológico del Grafo.

Pipelines e instancias únicas

Las etapas dentro de un pipeline deben considerarse como instancias únicas para su posterior compilación en un GAD de Spark. Si nos encontramos en la eventualidad de implementar dos veces el mismo modelo (como un meta clasificador donde un modelo basal es una regresión logística y el deliberador es otra), podemos generar instancias específicas con distintos nombres.

Pipelines y persistencia

A veces resulta conveniente el guardar el modelo/pipeline para posterior uso. A partir de Spark 2.3, toda la API basada en `DataFrame` tiene cobertura completa para persistir modelos. Por lo general, la librería MLlib presenta compatibilidad inversa, de manera tal de poder implementar modelos en una versión específica y volver a servirlos en versiones actualizadas de Spark. Existen un par de excepciones:

- **Persistencia del modelo:** Hay que asegurarse que el modelo persistido en la versión `X` tenga su análogo en la versión `Y` de Spark.

- **Comportamiento del modelo:** ¿El modelo se comporta de manera similar entre ambas versiones?

Persistencia de modelos

En la medida que comencemos a ingestar conjuntos de datos más y más grandes, probablemente necesitaremos que nuestro modelo persista para un posterior uso.

`pyspark.mllib` hace uso de un formato interno y un formato conocido como Predictive Model Markup Language para serializar el modelo. Dentro `PMML`, en el contexto de modelos con estructura de pipeline todavía no se migra del `pyspark.mllib` a `pyspark.ml`, por lo que es necesario implementar proyectos externos como `MLeap` que permite exportar modelos a formato PMML.

Referencias

- Aggarwal, Charu. 2016. Recommender Systems. Springer.
- Chambers, Bill; Zaharia, Matei. 2018. Spark, The Definitive Guide: Big Data Processing Made Simple. Sebastopol, CA: O'Reilly Media.
- Karau, Holden; Warren, R. 2017. High Performance Spark: Best practices for scaling and optimizing Apache Spark. Sebastopol, CA: O'Reilly Media.
- Yang, Xiwang; Guo, Yang; Liu, Yong; Steck, Harald. 2014. A survey of collaborative filtering based social recommender systems. Computer Communications 41: 1-10.