Taller introductorio a Spark

Contenido General

El lenguaje a utilizar es Python 3.x, principalmente en ambientes Windows y Linux (y puede aplicarse a macOS).

- Visión general de Spark
- Ambiente de trabajo e instalación de software
- Modelo de ejecución
- Spark SQL
- MLib
- Aplicación en clasificación de Texto

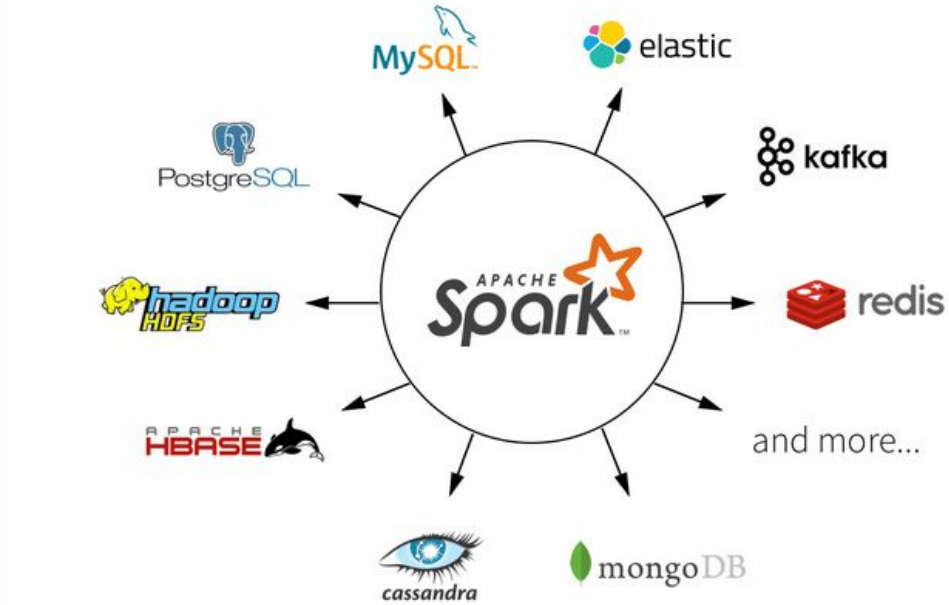
¿Que es Spark?

Apache Spark

Apache Spark se puede considerar un sistema de computación en clúster de propósito general y orientado a la velocidad. Es ampliamente considerado como el sucesor de MapReduce para el procesamiento de datos en clústeres Apache Hadoop.

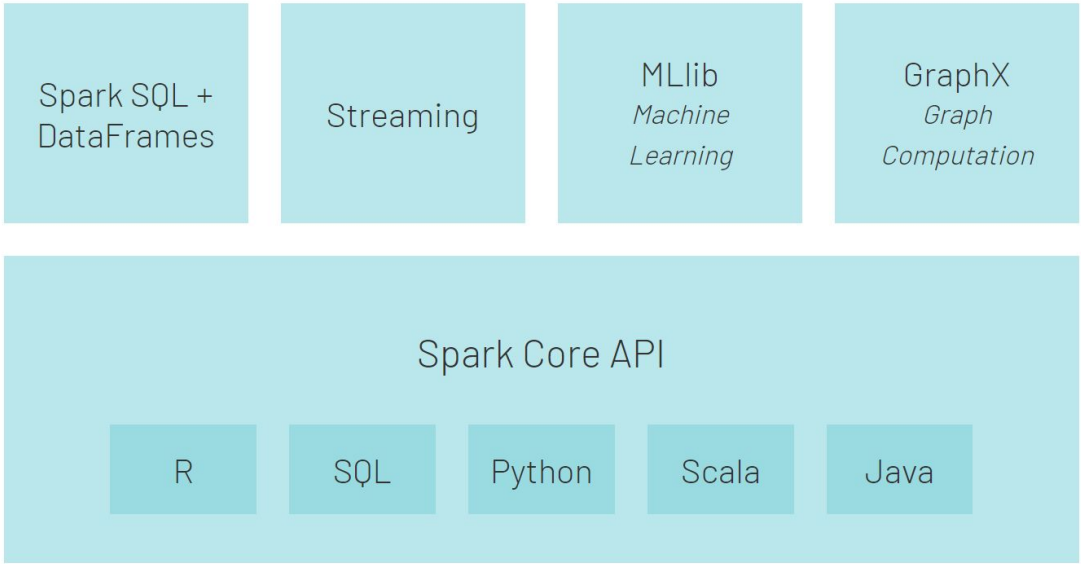
Apache Spark Ecosystem

Spark puede correr en modo cluster standalone, EC2, Hadoop YARN, Mesos, o Kubernetes. Puede acceder datos HDFS, Alluxio, Apache Cassandra, Apache HBase, Apache Hive, entre otros.



Apache Spark Ecosystem

Proporciona APIs de alto nivel en Java, Scala, Python y R, y un motor optimizado que admite grafos de ejecución general. Es compatible con un amplio conjunto de herramientas de alto nivel que incluyen Spark SQL para SQL y procesamiento de datos estructurados, MLlib para aprendizaje automático, GraphX para procesamiento de grafos y Spark Streaming.



Spark SQL

Spark SQL permite consultar datos estructurados utilizando SQL o la API DataFrame. Utilizable en Java, Scala, Python y R.

Spark SQL puede usar Hive metastores, SerDes (Serializador/Deserializador), y UDFs (Funciones definidas por el usuario). Así mismo se pueden integrar herramientas existentes de BI (business intelligence)

Spark Streaming

Muchas aplicaciones necesitan la capacidad de procesar y analizar no solo datos por lotes, sino también flujos de datos nuevos en tiempo real. Spark Streaming permite potentes aplicaciones analíticas interactivas tanto en stream como en datos históricos. Se integra fácilmente con una amplia variedad de fuentes de datos populares, incluyendo HDFS, Flume, Kafka y Twitter.

Spark Streaming

Muchas aplicaciones necesitan la capacidad de procesar y analizar no solo datos por lotes, sino también flujos de datos nuevos en tiempo real. Spark Streaming permite potentes aplicaciones analíticas interactivas tanto en stream como en datos históricos. Se integra fácilmente con una amplia variedad de fuentes de datos populares, incluyendo HDFS, Flume, Kafka y Twitter.

Spark Machine Learning

MLlib es la biblioteca de aprendizaje automático (ML) de Spark. Su objetivo es hacer que el aprendizaje automático práctico sea escalable y fácil. En un nivel alto, proporciona herramientas como:

- ML Algorithms: algoritmos de aprendizaje comunes como clasificación, regresión, agrupamiento y filtrado colaborativo
- Featurization: extracción de características, transformación, reducción de dimensionalidad y selección
- Pipes: herramientas para construir, evaluar y ajustar pipes de ML
- Persistence: guardar y cargar algoritmos, modelos y pipes
- Utilities: álgebra lineal, estadísticas, manejo de datos, etc.

Spark GraphX

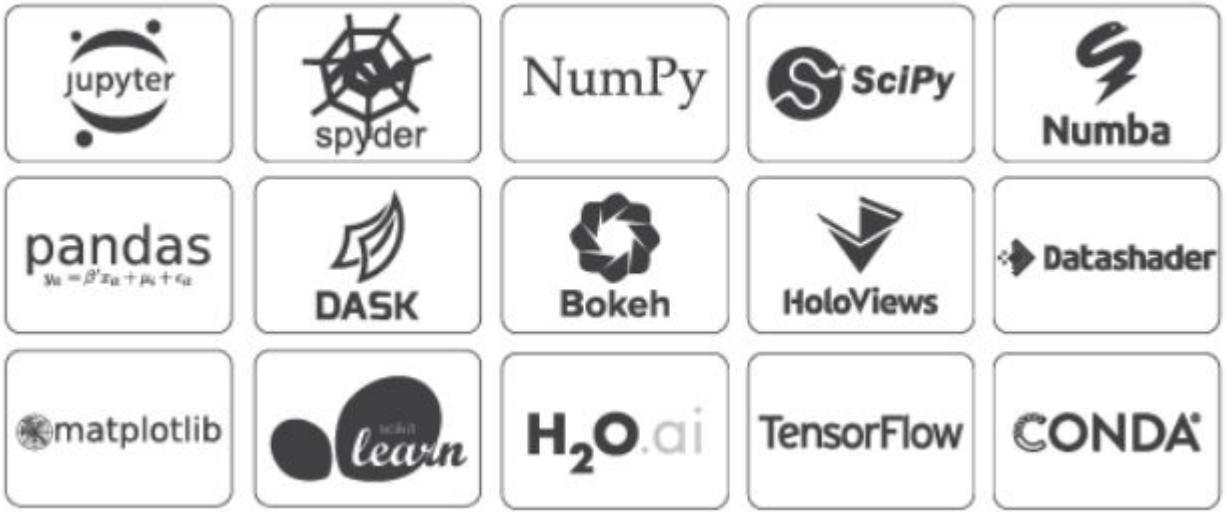
GraphX es un motor de cálculo de grafos que permite construir, transformar y analizar interactivamente sobre datos estructurados. Viene con una biblioteca completa de algoritmos comunes entre los que se encuentran:

- PageRank
- Connected components
- Label propagation
- SVD++
- Strongly connected components
- Triangle count

Ambiente de Trabajo

Anaconda Distribution

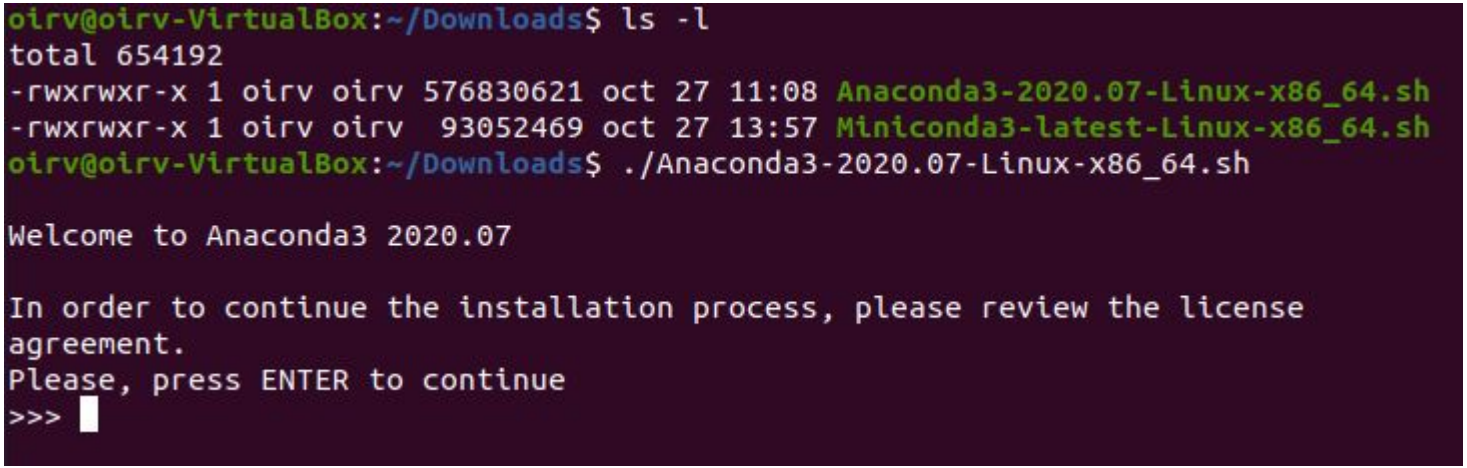
La distribución de código abierto Anaconda es la forma más fácil de realizar ciencia de datos Python / R y aprendizaje automático en Linux, Windows y Mac OS X.



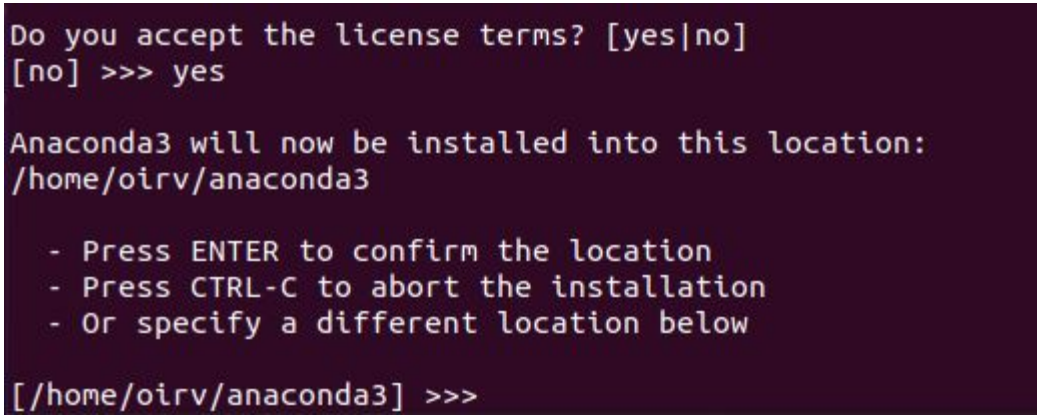
Instalación

Instalación

bash Anaconda3-2019.07-Linux-x86_64.sh



Instalación



Instalación

```
installation finished.

Do you wish the installer to prepend the Anaconda3 install
location

to PATH in your /home/user/.bashrc ? [yes|no]
```

Si se elige “no”, se puede activar anaconda en linux de la siguiente manera:

```
source /home/user/anaconda3/bin/activate
```

Instalación

Use el comando ***conda*** para probar la instalación:

```
conda info
```

Para actualizar

```
conda update conda
```

Crear y activar entornos de anaconda

Cree un entorno de Python 3 llamado ***spark*** haciendo lo siguiente:

```
conda create --name spark python=3.8
```

Activar el entorno:

```
conda activate spark
```

Instalar anaconda el entorno:

```
Conda install anaconda
```

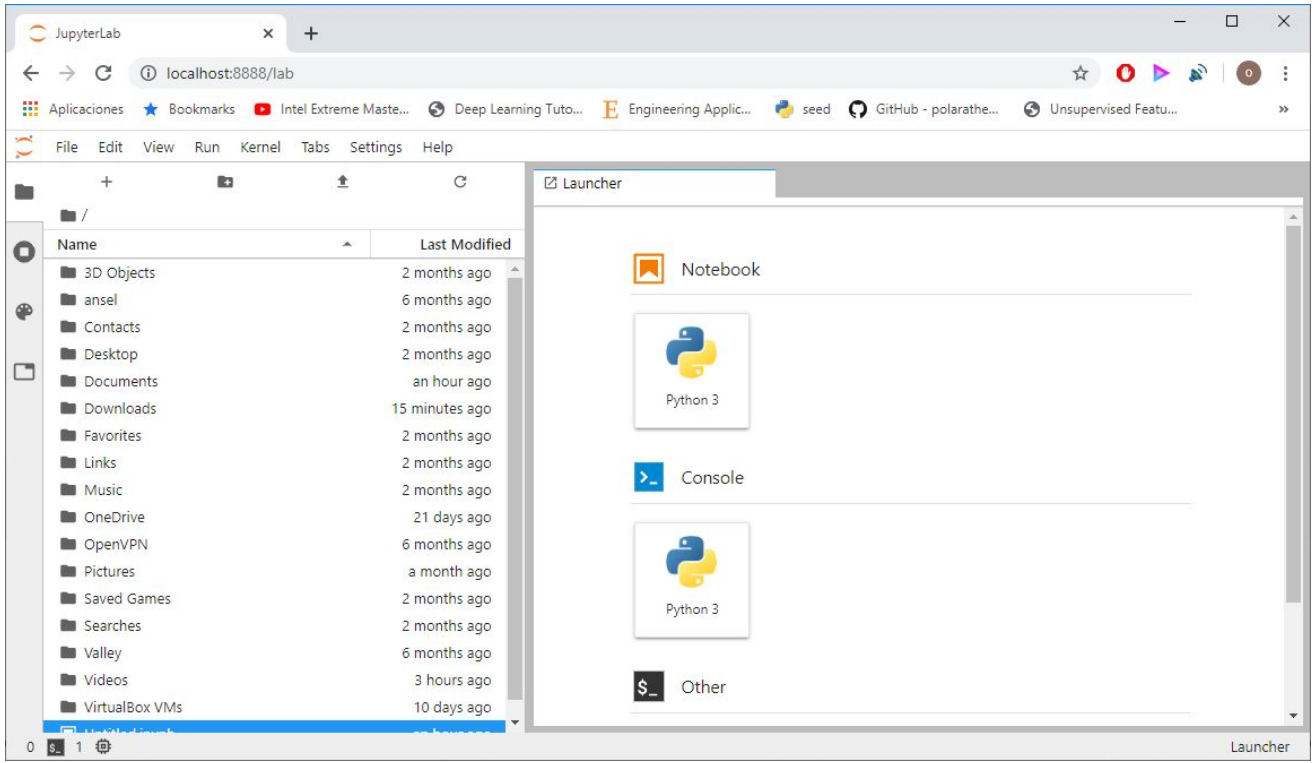
JupyterLab

JupyterLab es un entorno de desarrollo interactivo basado en web para notebooks de Jupyter, código y datos.

JupyterLab

Ejecutar en consola:

```
jupyter lab
```



Instalación Spark

Anaconda

Aunque Spark fue diseñado para correr en clusters, es fácil de ejecutarlo localmente en una máquina; solo se necesita tener Java instalado y la variable de entorno JAVA_HOME que apunta a la instalación de Java.

Para usar en JupyterLab:

- Descargue e instale el JDK de Java versión 8
  - En ubuntu: `sudo apt install openjdk-8-jdk`
  - En ubuntu: `export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64`
- Ejecutar en consola anaconda e instalar Pyspark
  - conda install -c conda-forge pyspark
- Exportar variable de entorno JAVA_HOME

Externa

Si se desea utilizar la versión más nueva disponible en la pagina oficial de spark.

Para usar en JupyterLab:

- Descargue e instale el JDK de Java versión 8
  - En ubuntu: `sudo apt install openjdk-8-jdk`
  - En ubuntu: `export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64`
- Ejecutar en consola anaconda e instalar Py4J
  - conda install -c conda-forge py4j
- Descargar spark https://spark.apache.org/downloads.html
- Descomprimir spark en alguna carpeta
- Agregar ruta de pyspark
- Exportar variable de entorno JAVA_HOME

Rutas

Se mencionó que se debe crear una variable de entorno JAVA_HOME así como agregar la ruta a la instalación de spark (en caso que se desee conectar remotamente), dentro de python se puede hacer fácilmente:

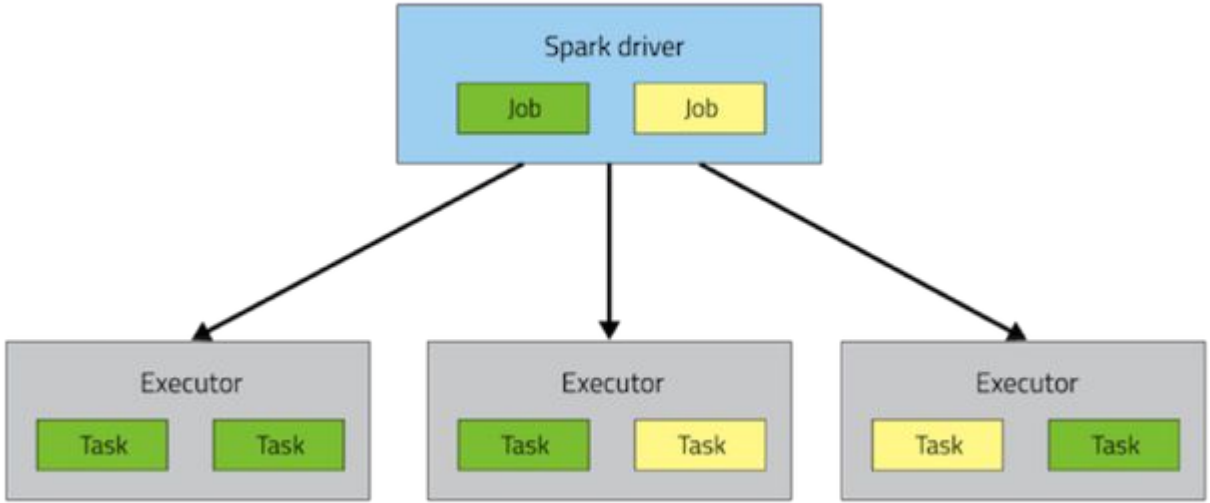
```
import os
# en windows la diagonal es \\ en linux y mac es /
os.environ["JAVA_HOME"] = "C:\\Program Files\\Java\\jdk1.8.0_241"


import sys
# es la ruta donde se descomprime spark
sys.path.append("/home/octavio.renteria/spark/python")
```

Modelo de ejecución

Spark execution model

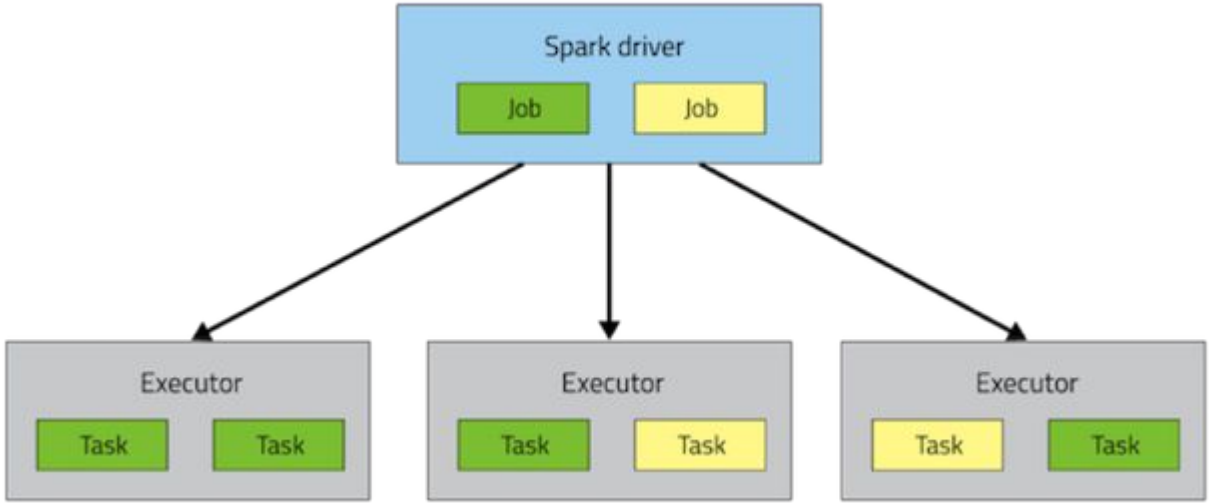
La ejecución de una aplicación Spark implica conceptos como **driver, executor, task, job,** y **stage**.



Spark execution model

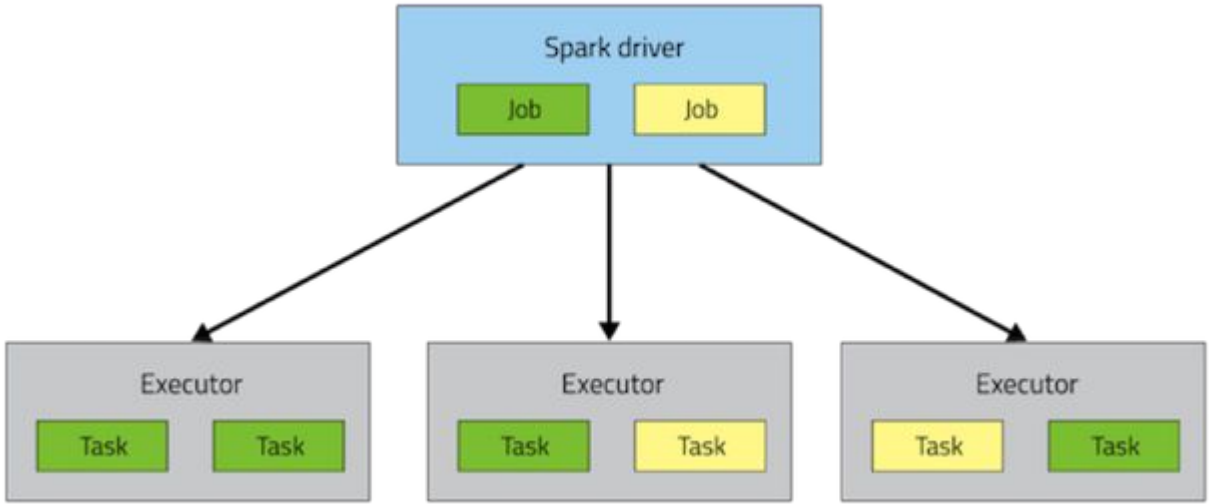
En tiempo de ejecución una aplicación Spark se mapea a un **driver** único y a un conjunto de procesos de ejecución distribuidos entre los nodos de un clúster.

El **driver** de procesos gestiona el flujo de **jobs** y **tasks**, el driver está disponible todo el tiempo que se ejecuta la aplicación.



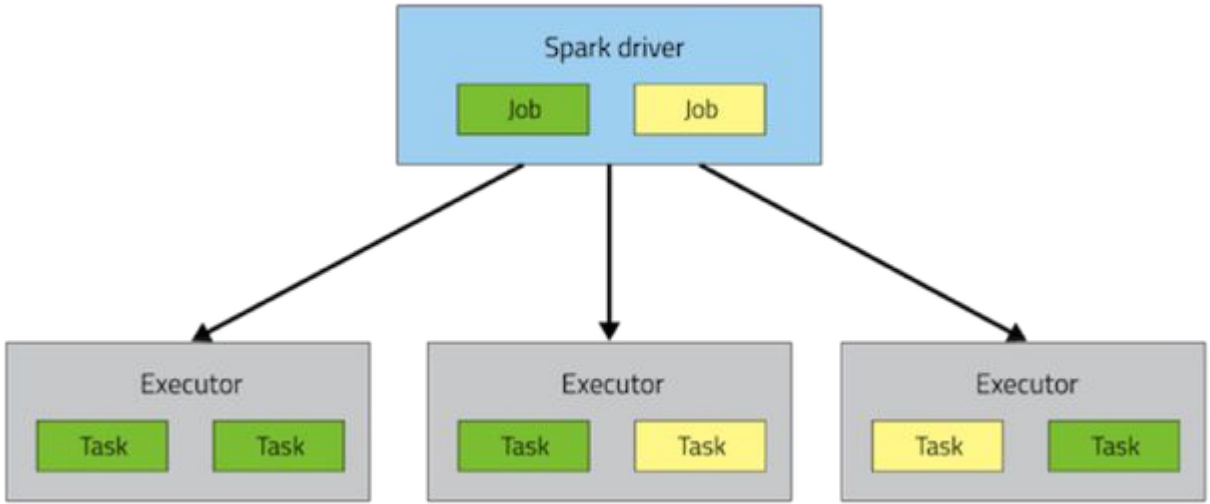
Spark execution model

Los **executors** son responsables de realizar el **job**, en forma de **tasks**, así como de guardar en caché cualquier información generada.



Spark execution model

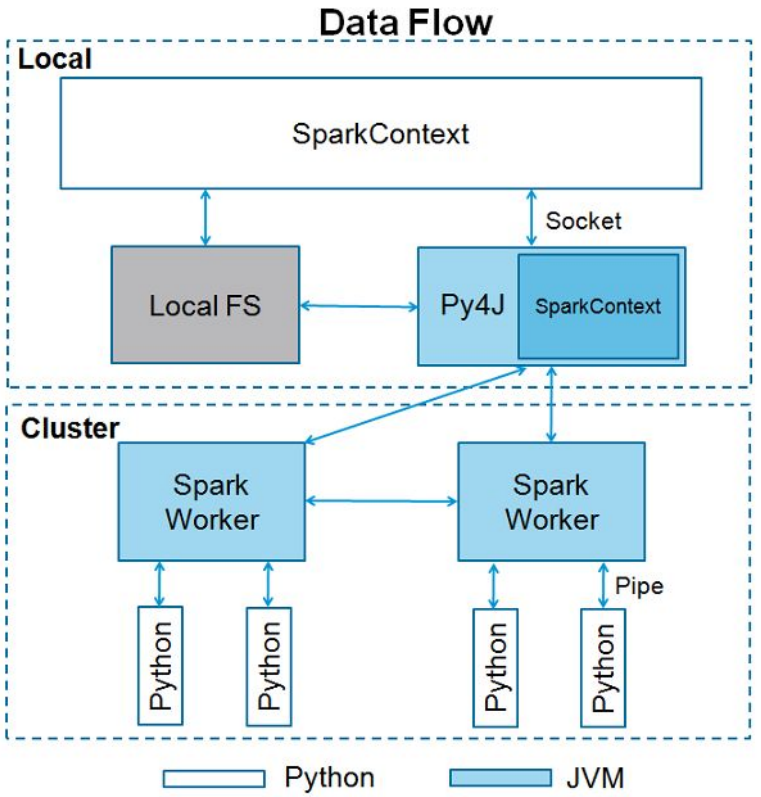
Invocar una acción dentro de una aplicación Spark desencadena el inicio de un **job**. Spark examina el conjunto de datos del que depende esa acción y formula un plan de ejecución. El plan de ejecución ensambla las transformaciones del conjunto de datos en **stages**. Un **stage** es una colección de tareas que ejecutan el mismo código, cada una en un subconjunto diferente de datos.



Spark Context

SparkContext

Es el punto de entrada a cualquier funcionalidad de Spark. Cuando ejecutamos cualquier aplicación, se inicia un programa **driver**, que contiene la función main y el **SparkContext** se inicia aquí.



SparkContext Class

```
class pyspark.SparkContext (
   master = None,
   appName = None,
   sparkHome = None,
   pyFiles = None,
   environment = None,
   batchSize = 0,
   serializer = PickleSerializer(),
   conf = None,
   gateway = None,
   jsc = None,
   profiler_cls = <class 'pyspark.profiler.BasicProfiler'> )
```

SparkContext Class

Algunos de los parámetros más relevantes:

- ***master***: Es la URL del clúster al que se conecta (remota o localmente).
- **appName**: nombre de la aplicación con que se identifica en el cluster
- **conf**: lista de opciones de spark

SparkConf Class

Un objeto Lista {SparkConf} para establecer propiedades de Spark, puede ver la lista completa en https://spark.apache.org/docs/latest/configuration.html

- **spark.executor.cores**: El número de núcleos a usar en cada ejecutor
- **spark.cores.max**: Cantidad máxima de núcleos de CPU que puede solicitar la aplicación al cluster
- **spark.executor.memory**: Cantidad de memoria a utilizar por proceso ejecutor, soporta sufijos de unidad de tamaño ("k", "m", "g" o "t", por ejemplo, 512m, 2g).

SparkConf Class

Ejemplo, se solicitan 4 núcleos y memoria de 4 gigabytes:

```
conf = pyspark.SparkConf()
conf.set('spark.executor.cores', '4')
conf.set('spark.cores.max', '4')
conf.set('spark.executor.memory', '4g')
```

Iniciando un contexto

```
# crear el contexto de spark
conf = pyspark.SparkConf()
conf.set('spark.executor.cores', '4')
conf.set('spark.cores.max', '4')
conf.set('spark.executor.memory', '4g')
# local
sc = pyspark.SparkContext(master="local",appName="MyApp",conf=conf)
# remoto
sc = pyspark.SparkContext(master="spark://10.10.22.162:7077",appName="MyApp",
                          conf=conf)
```

Spark SQL

Spark SQL, DataFrames and Datasets

Spark SQL es un módulo de para el procesamiento de datos estructurados. Trabaja sobre **Datasets**, que son un conjunto de datos distribuidos.

Un **DataFrame** es un **Dataset** organizado en columnas con nombre. Es conceptualmente equivalente a una tabla en una base de datos relacional. Los **DataFrames** se pueden construir a partir de una amplia variedad de fuentes, como archivos de datos estructurados (csv, json, xml, etc), tablas en Hive, bases de datos externas o **RDD** existentes. En este tipo de datos se pueden realizar consultas **SQL**.

Los **RDD (Resilient Distributed Datasets)** son un conjunto de objetos Java o Scala que representan una colección de elementos particionados en los nodos del clúster que se pueden operar en paralelo.

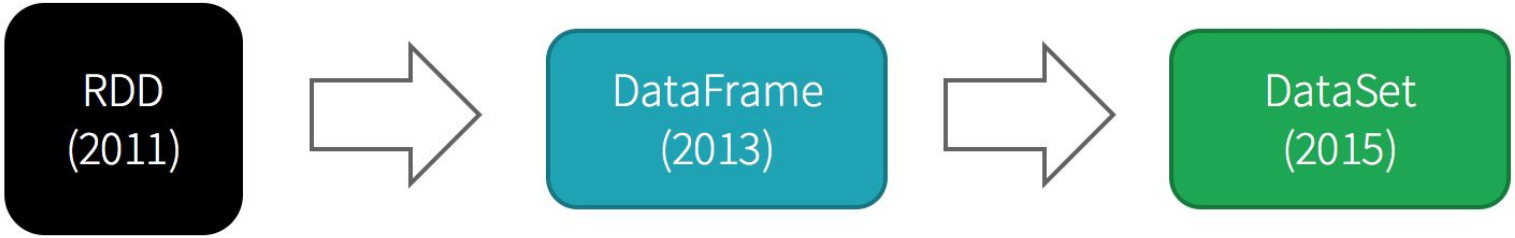
Características de los DataFrame/RDD

- **De naturaleza inmutable**: podemos crear DataFrames/RDD pero no podemos cambiarlos. Se les pueden aplicar transformaciones.
- **Evaluaciones perezosas**: lo que significa que una tarea no se ejecuta hasta que se realiza una acción.
- **Distribuido**: RDD y DataFrame son distribuidos por naturaleza.

Ventajas de DataFrame

- Están diseñados para procesar una gran colección de datos estructurados o semiestructurados.
- Los datos se organizan en columnas con nombre, lo que ayuda a Spark a comprender el esquema de un DataFrame para optimizar el plan de ejecución en sus consultas.
- Tienen la capacidad de manejar petabytes de datos.
- Tiene soporte para una amplia variedad de fuentes y formatos de datos.
- Tiene soporte para APIs de diferentes lenguajes como Python, R, Scala, Java.

Historia de las APIs



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the “Best of both worlds”: **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

Cargando archivos

sql.SparkSession.read

Interfaz utilizada para cargar un DataFrame desde sistemas de almacenamiento externo (archivos locales o remotos).

Este ejemplo muestra cómo cargar un archivo csv, se pueden exportar archivos de varios orígenes:

```
# archivo local
input_data = sqlContext.read.csv("my_file.csv")
# desde un servidor hadoop de archivos
input_data = sqlContext.read.csv("hdfs://10.10.22.162:9000/user/hadoop/my_file.csv")
# desde Amazon Simple Storage Service S3 (requiere configuración de llaves)
input_data = sqlContext.read.csv("s3n://my_file.csv")
```

sql.SparkSession.read

Interfaz utilizada para cargar un DataFrame desde sistemas de almacenamiento externo (archivos locales o remotos).

Este ejemplo muestra cómo cargar un archivo json, se pueden exportar archivos de varios orígenes:

```
# archivo local
input_data = sqlContext.read.json("my_file.json")
# desde un servidor hadoop de archivos
input_data = sqlContext.read.json("hdfs://10.10.22.162:9000/user/hadoop/my_file.json")
# desde Amazon Simple Storage Service S3 (requiere configuración de llaves)
input_data = sqlContext.read.json("s3n://my_file.json")
```

DataFrame

Detalles del dataframe cargado

```
print( type(input_data) )
input_data.printSchema()
input_data.show()
```

Crear una vista para manejar los datos:

```
input_data.createOrReplaceTempView("zonas")
```

DataFrame SQL

Se pueden hacer consultas con sentencias SQL sobre los DataFrames, la función sql ejecuta la sentencia y regresa un DataFrame con los datos de la consulta:

```
# seleccionar las columnas CVE_ZM, NOM_ZM y POB_2015, en caso de que el nombre de la columna
# Tenga caracteres conflictivos (como espacios o caracteres especiales), se pueden usar
# comillas invertidas ` `
result = sqlContext.sql("SELECT CVE_ZM, NOM_ZM, POB_2015 from zonas")
result.show()
```

El resultado siempre es un DataFrame.

Funciones DataFrame

Algunas funciones relevantes del DataFrame:

- **select**: genera un DataFrame con las columnas requeridas
- **rdd**: convierte el DataFrame a RDD
- **limit**: genera un DataFrame con los primeros “N” renglones especificados.
- **show**: Imprime en pantalla los primeros 20 renglones especificados.
- **filter**: genera un DataFrame con los renglones donde una condición es verdadera.
- **where**: es un alias de **filter**.
- **collect**: Copia al host todo el DataFrame en forma de lista de renglones.
- **withColumn**: Modifica los valores de una columna o agrega una nueva columna.

MLlib

General

MLlib es la biblioteca de aprendizaje automático (Machine Learning) de Spark. Su objetivo es hacer que el aprendizaje automático sea práctico, escalable y fácil.

MLlib utiliza el paquete de álgebra lineal ***Breeze***, que depende de ***netlib-java*** para optimizar el procesamiento numérico. Si las bibliotecas nativas no están disponibles en el sistema, aparecerá un mensaje de advertencia y en su lugar se utilizará una implementación de **JVM** pura.

LSA

El análisis semántico latente (LSA), es un método matemático que intenta resaltar las relaciones latentes dentro de una colección de documentos. Se basa en la suposición de que las palabras de significado cercano aparecerán en fragmentos de texto similares.

Este algoritmo se compone de estos pasos generales:

- Generar una matriz de términos-documentos

	D1	D2	D3	D4	D5	D6	D7	D8	D9
measur	1	0	0	2	1	0	1	0	0
effici	1	0	1	0	0	0	1	0	0
machin	1	0	0	0	0	0	0	0	0
factori	1	0	0	0	0	0	0	0	0
system	1	0	0	0	0	0	0	0	0
input	1	0	0	2	0	0	1	0	1
output	1	1	0	2	0	0	1	0	1
averag	0	1	0	0	0	0	1	0	0
cost	0	1	1	0	0	0	0	0	0
resourc	0	1	0	0	0	0	0	0	0
consum	0	1	0	0	0	0	0	1	0

LSA

El análisis semántico latente (LSA), es un método matemático que intenta resaltar las relaciones latentes dentro de una colección de documentos. Se basa en la suposición de que las palabras de significado cercano aparecerán en fragmentos de texto similares.

Este algoritmo se compone de estos pasos generales:

- Generar una matriz de términos-documentos
- De matriz anterior, generar una matriz TF-IDF

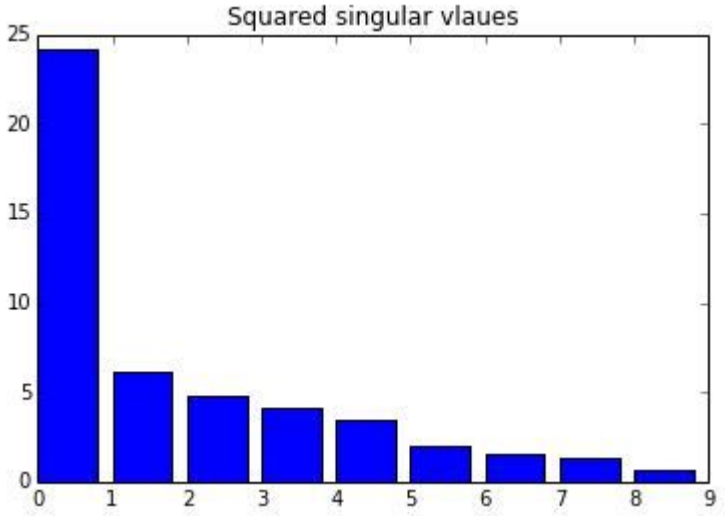
	D1	D2	D3	D4	D5	D6	D7
measur	0.237917	0.000000	0.000000	0.475834	0.305211	0.000000	0.3052
effici	0.396528	0.000000	0.814116	0.000000	0.000000	0.000000	0.5086
machin	1.189584	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
factori	1.189584	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
system	1.189584	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
input	0.237917	0.000000	0.000000	0.475834	0.000000	0.000000	0.3052
output	0.198264	0.254343	0.000000	0.396528	0.000000	0.000000	0.2543
averag	0.000000	0.763028	0.000000	0.000000	0.000000	0.000000	0.7630
cost	0.000000	0.763028	1.221174	0.000000	0.000000	0.000000	0.0000
resourc	0.000000	1.526056	0.000000	0.000000	0.000000	0.000000	0.0000
consum	0.000000	0.763028	0.000000	0.000000	0.000000	0.000000	0.0000

LSA

El análisis semántico latente (LSA), es un método matemático que intenta resaltar las relaciones latentes dentro de una colección de documentos. Se basa en la suposición de que las palabras de significado cercano aparecerán en fragmentos de texto similares.

Este algoritmo se compone de estos pasos generales:

- Generar una matriz de términos-documentos
- De matriz anterior, generar una matriz TF-IDF
- Reducir dimensionalidad a la matriz TF-IDF con SVD

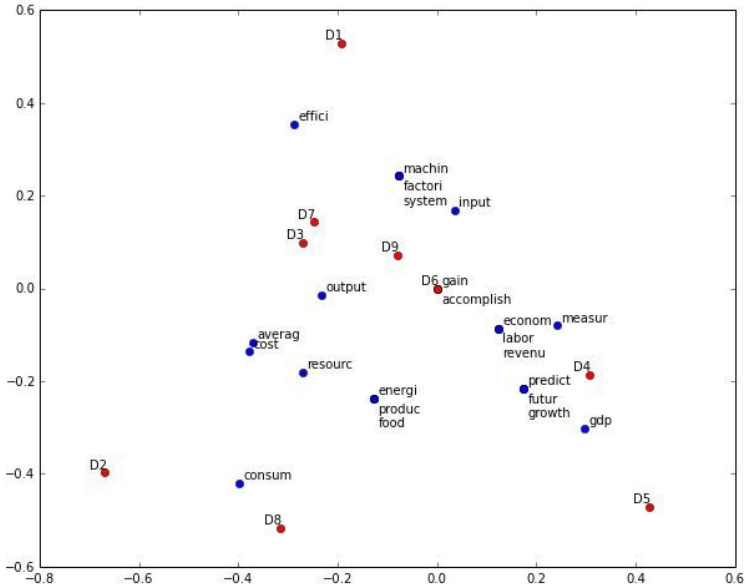


LSA

El análisis semántico latente (LSA), es un método matemático que intenta resaltar las relaciones latentes dentro de una colección de documentos. Se basa en la suposición de que las palabras de significado cercano aparecerán en fragmentos de texto similares.

Este algoritmo se compone de estos pasos generales:

- Generar una matriz de términos-documentos
- De matriz anterior, generar una matriz TF-IDF
- Reducir dimensionalidad a la matriz TF-IDF con SVD
- Medir distancias entre documentos con la matriz U



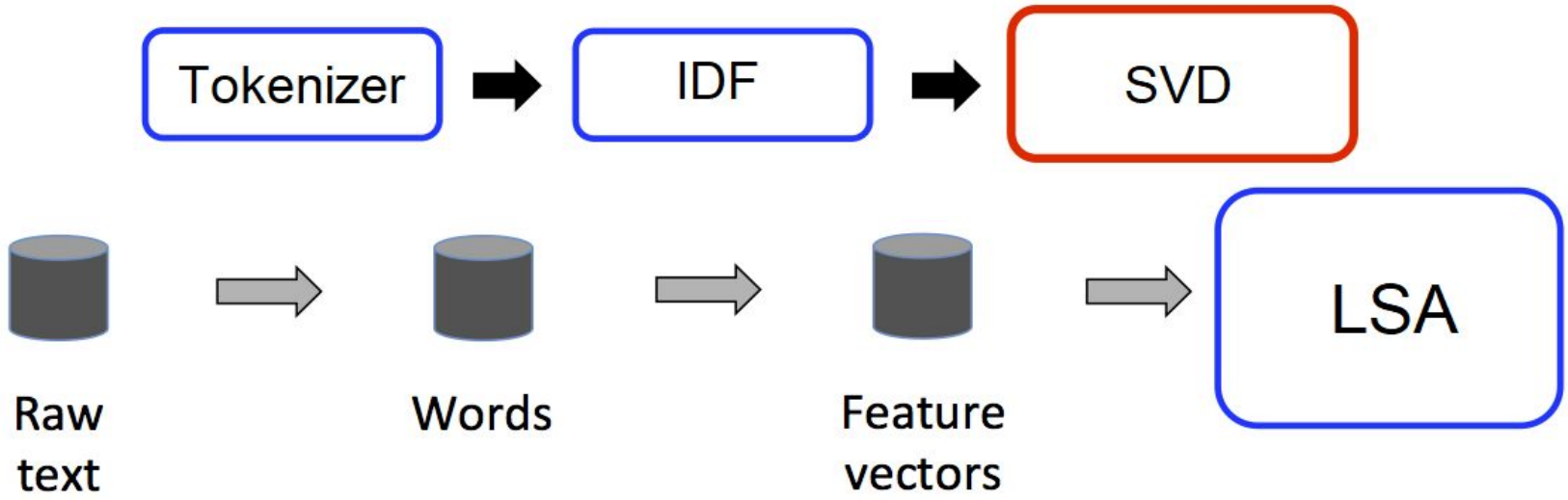
LSA

LSA se basa en las siguientes simplificaciones:

- Los documentos de texto se representan como "bolsas de palabras", donde el orden de las palabras no es importante. Sin embargo, lo que cuenta es cuántas veces aparece cada palabra.
- Los conceptos se representan como patrones de palabras que generalmente aparecen juntas en un documento, como "correa", "perro" y "obedecer" que aparecen juntas en un documento sobre adiestramiento canino.
- Un conjunto de palabras conocidas como "steam words" son excluidas del análisis por que no aportan mucho. Ejemplos de estas palabras son "y", "o", "para", "en", "de", "el", "a", etc.
- Las palabras se derivan, es decir, se reducen a la raíz de la palabra, como 'biblio' en 'biblioteca', 'bibliotecario' o 'bibliografía', etc.

LSA en Spark

En esta parte se realizarán transformaciones en una colección de documentos para mostrar algunas de las funciones y clases de MLlib aplicadas a NLP (procesamiento de lenguaje natural), se convertirá el texto a ***features*** para hacer comparación de documentos.



NLP en Spark

En esta parte se realizarán transformaciones al DataSet para mostrar algunas de las funciones y clases de MLlib aplicadas a NLP (procesamiento de lenguaje natural), se convertirá el texto a ***features*** para hacer comparación de documentos. Las clases a importar son “Transformadores”.

```
# empezar importando las clases relevantes para nuestro caso.
from pyspark.ml.feature import Tokenizer, IDF, StopWordsRemover, CountVectorizer
```

Tokenizer

La tokenización es el proceso de tomar texto (como una oración) y dividirlo en términos individuales (generalmente palabras). La clase **Tokenizer** proporciona esta funcionalidad.

Algunos de sus argumentos relevantes:

- inputCol, columna de un DataFrame que serán los datos de entrada.
- outputCol, nombre de la columna que se creará con los datos transformados de salida

```
# La columna de interes en el DataFrame es texto, se creará una columna words con los resultados.
tokenizer = Tokenizer(inputCol="texto", outputCol="words")
result = tokenizer.transform(result)
result.show(1,True)
```

StopWordsRemover

Stop Words son palabras que se deben excluir de la entrada, generalmente porque las palabras aparecen con frecuencia y tienen poca relevancia. StopWordsRemover toma como entrada una secuencia de cadenas (por ejemplo, la salida de un Tokenizer) y elimina todas las Stop Words de las secuencias de entrada.

Se puede acceder a la lista de Stop Words predeterminadas para algunos idiomas llamando a la función **StopWordsRemover.loadDefaultStopWords** (idioma)

```
remover = StopWordsRemover(inputCol="words",
                           outputCol="filtered",
                           stopWords=StopWordsRemover.loadDefaultStopWords("spanish"))
result = remover.transform(result)
result.show(1,True)
```

CountVectorizer

CountVectorizer y **CountVectorizerModel** tienen como objetivo ayudar a convertir una colección de documentos de texto en vectores de contadores de tokens. Cuando un diccionario a priori no está disponible, **CountVectorizer** se puede usar como un Estimador para extraer el vocabulario y genera un **CountVectorizerModel.**

```
vectorizer = CountVectorizer(inputCol="filtered", outputCol="vectorized")

vectorizer_model = vectorizer.fit(result)

result = vectorizer_model.transform(result)

result.show(1,True)
```

IDF

IDF es un Estimador que se ajusta a un conjunto de datos y produce un **IDFModel**. IDFModel toma vectores de características (generalmente creados a partir de **HashingTF** o **CountVectorizer**) y escala cada característica.

El objetivo de esta operación es ajustar los pesos de las palabras aparecen con menos frecuencia en un corpus son más relevantes.

```
idf = IDF(inputCol="vectorized", outputCol="idf")
idf_model = idf.fit(result)
result = idf_model.transform(result)
result.show(1,True)
```

Distributed matrix

Distributed matrix

En algún momento, los datos se requieren en otra forma para realizar operaciones, un tipo especial son Matrices Distribuidas. Una matriz distribuida tiene índices de fila y columna de tipo long y valores de tipo doble, almacenados distributivamente en uno o más RDD. Spark implementa cuatro tipos de matrices distribuidas:

- RowMatrix
- IndexedRowMatrix
- CoordinateMatrix
- BlockMatrix

RowMatrix

Resenta una matriz distribuida orientada a renglones sin índices de renglones significativos.

```
from pyspark.mllib.linalg.distributed import RowMatrix
from pyspark.mllib.linalg import Vectors

data = result.select("idf").rdd.map(lambda row:
                        Vectors.sparse(row.idf.size, row.idf.indices, row.idf.values )
                        )
mat = RowMatrix(data)
```

RowMatrix Functions

Algunas de las operaciones que se pueden hacer en Matrices.

- **multiply**: Multiplica esta matriz por una matriz local densa a la derecha.
- **computeCovariance**: Calcula la matriz de covarianza, interpretando cada fila como una observación.
- **computeGramianMatrix**: Calcula la matriz de Gram
- **computeSVD**: Calcula los factores de descomposición de valores singulares (SVD)
- **computePrincipalComponents**: Calcula los ***K*** componentes principales de la matriz dada

computeSVD

En nuestro ejemplo, se aplicad SVD a nuestra matriz, reduciendo la dimensión (columnas de características) a solo 100 elementos.

```
# Aquí solo trabajamos con la matriz U
svd = mat.computeSVD(100, computeU=True)
U = svd.U
s = svd.s
V = svd.V
```

computeSVD

En nuestro ejemplo, se aplicad SVD a nuestra matriz, reduciendo la dimensión (columnas de características) a solo 100 elementos.

```
# Aquí solo trabajamos con la matriz U
svd = mat.computeSVD(100, computeU=True)
U = svd.U
s = svd.s
V = svd.V
```

Convertir de Matrix a DataFrame

Todos los manejadores de Datos en Spark tienen la función **map**, **filter** y **reduce**. Se pueden utilizar para realizar operacione sobre los items.

En este ejemplo convertimos la matriz a un Dataframe

```
u = U.rows.map(lambda x: (x, )).toDF(['features'])
```

Buscar dato

Ahora buscaré los registros que tengan la palabra “matrimonio”, se imprimen los primeros 5 y elegiré el id del registro que me interese:

```
sqlContext.sql("select `registro`,index as id, rubro from documentos").where(
 F.col("texto").contains('matrimonio') ).show(5,False)
```

```
id_target = 253
```

zipWithIndex

Ya que se necesita el vector de features del documento a analizar, se requiere extraerlo de la matriz **U,** desafortunadamente las Matrices distribuidas no son indexable por default, se le puede agregar un índice con la función **zipWithIndex,** que agrega un índice de cada fila (el índice lo agrega al final de cada fila).

```
vectors = U.rows.zipWithIndex()
# bajar los primeros 5 elementos para imprimir
vectors.take(5)
```

map, filter, reduce

Los **DataFrame** y **RDD** poseen funciones tales como ***map***, ***filter*** y ***reduce***. Que son útiles para manipular los datos, en este ejemplo extraemos el vector que corresponde al id del registro que nos interesa, el vector es de tipo **VectorSparse**, por lo que lo convertimos al numpy array con la función **toArray()**.

Finalmente se “bajan” los datos con **collect**:

```
target = vectors.filter(lambda x:x[1]==id_target).map(lambda x:x[0].toArray()).collect()
# notar que collect regresa una lista, en este caso de 1 elemento
target
```

map, filter, reduce

Finalmente se calcula la distancia entre el vector elegido contra el resto de los vectores, para esto se calcula la distancia coseno con **cosine**, que es parte de scipy.

```
from scipy.spatial.distance import cosine
import numpy as np

distances = vectors.map(lambda x:  cosine(target[0], x[0].toArray()) ).collect()
# se ordenan las distancias de menor a mayor
distances = np.array(distances)
# se usan los índices que son relativos a los ids de los registros
idx = np.argsort(distances)
```

map, filter, reduce

Finalmente se extraen los top 5 registros, basados en las distancias. En esta sentencia tambien se agrega el valor de distancia:

```
rows = sqlContext.sql(
        "select `registro`,index as id, texto from documentos"
    ).rdd.filter(
        lambda x: x.id in idx[:5]
    ).map(
        lambda x: (distances[x.id], x.registro, x.texto)
    ).collect()
```

map, filter, reduce

Ordenar resultados e imprimir.

```
rows = sorted(rows, key = lambda x: x[0])
for row in rows:
    print(row,'\n')
```