**UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO**

**FACULTAD DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA, INFORMÁTICA Y MECÁNICA**

**ESCUELA PROFESIONAL DE INGENIERÍA INFORMÁTICA Y DE SISTEMAS**



**“ Implementación Algoritmo Apriori PYSPARK”**

**ASIGNATURA :** Minería de Datos

**DOCENTE :** PhD. Carlos Fernando Montoya Cubas

**PRESENTADO POR :**

**NOMBRES** **CÓDIGOS**

Bustamante Flores Erick Andrew 171943

Dorado Torres Diego Alonso 170594

Huancara Ccolqque Alex Helder 174911

Huillca Herrera Victor Pool 163845

Sarco Jacinto Daniel Eduardo 174452

Quispe Yahuira Ronaldo 171866

Vega Centeno Olivera Ronaldinho 140934

**Cusco – Perú**

**2022**

**Índice**

Algoritmo Apriori PySpark . . . . . . . . . . . .. . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . .2

1. Algoritmo Apriori . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . .2

2. Apriori en Spark . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3

3. Implementación del algoritmo . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . .4

4. Ejecución del algoritmo . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . .12

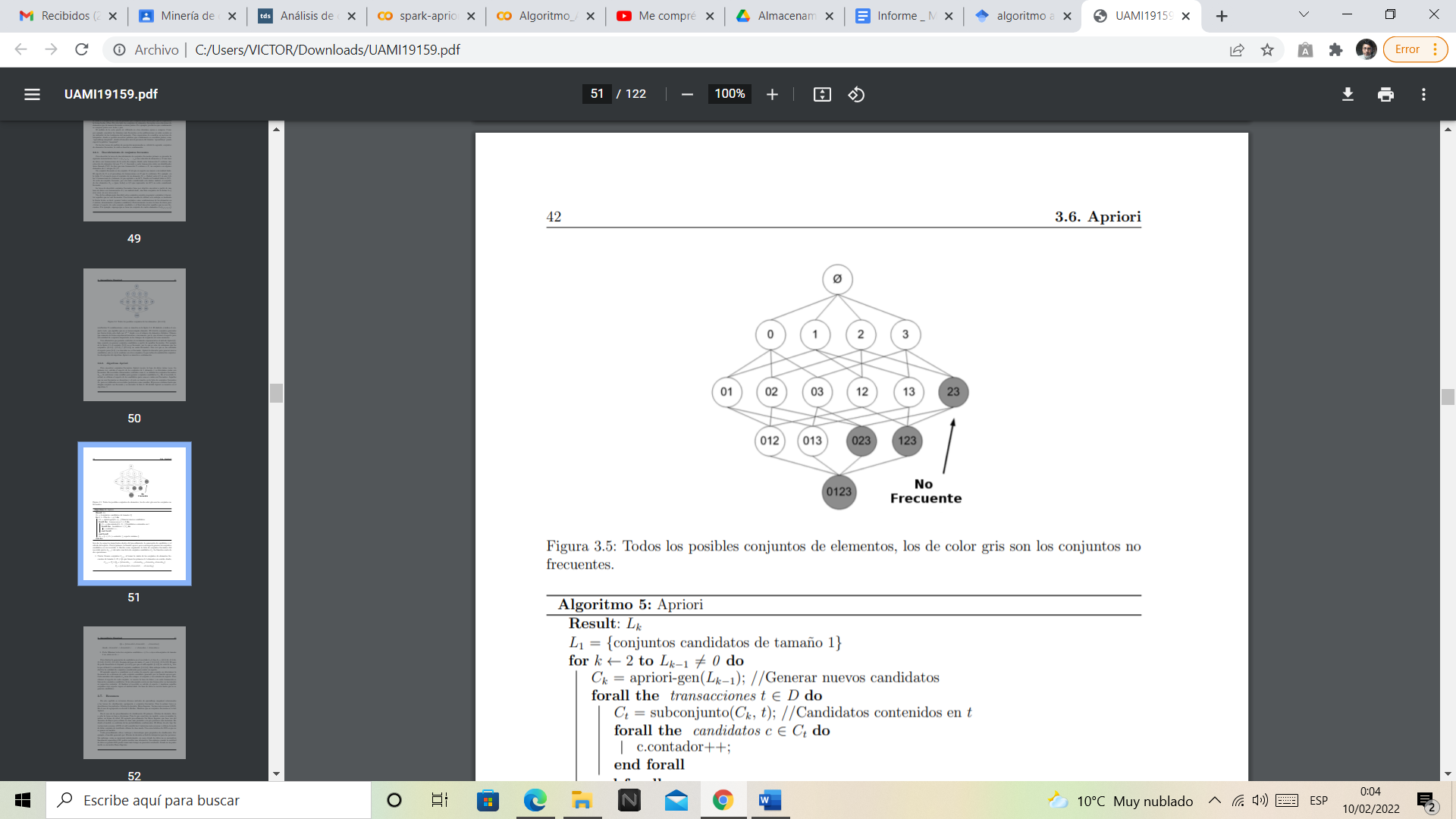
5. Conclusiones . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .12

6. Bibliografía . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .12

**ALGORITMO APRIORI PYSPARK**

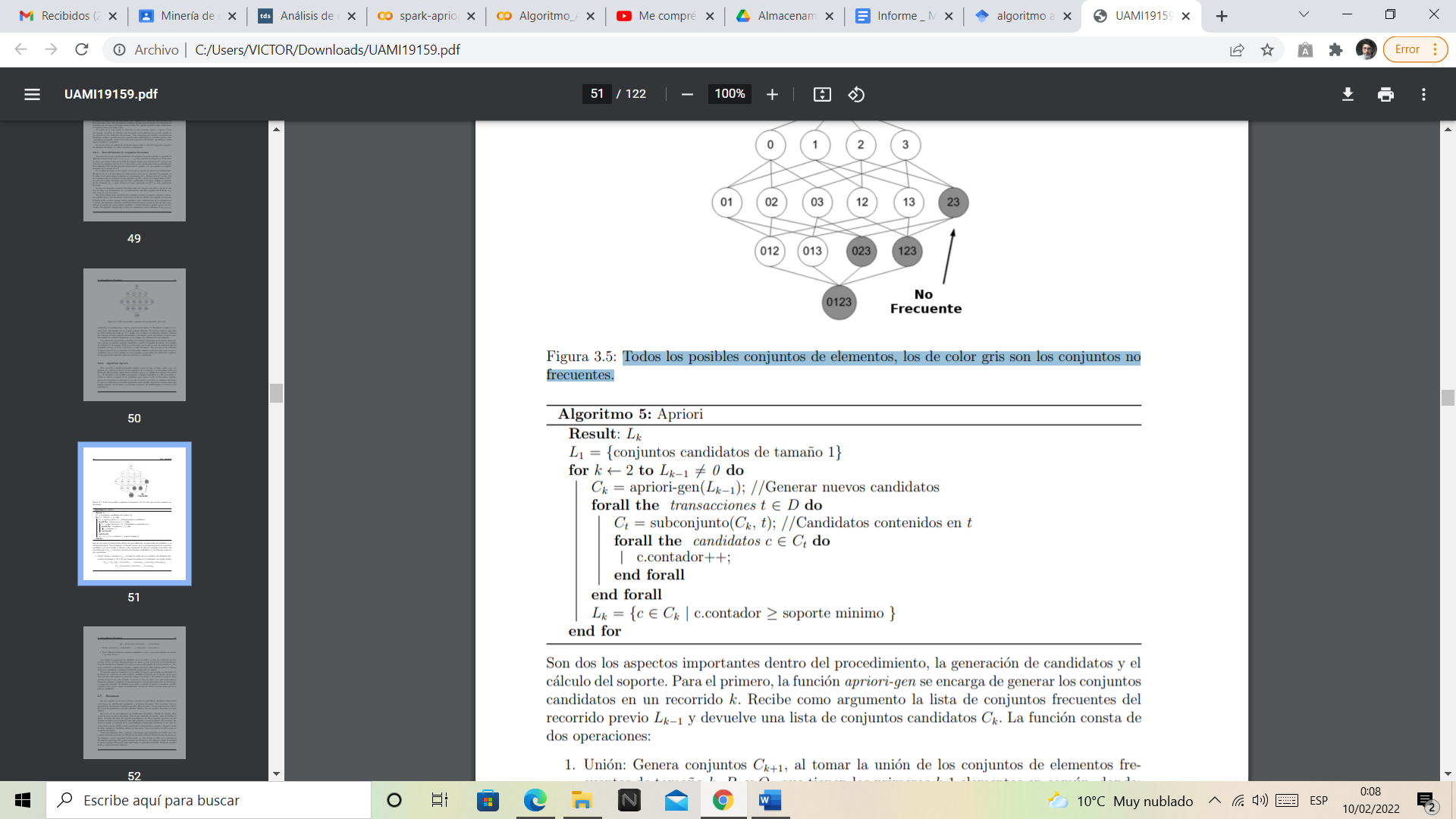
1. **Algoritmo Apriori**

Para encontrar conjuntos frecuentes Apriori recorre la base de datos varias veces. La primera vez, calcula el soporte de los conjuntos de 1 elemento y se determina cuales son frecuentes. En recorridos subsecuentes, referidos como *k*, se utilizan los conjuntos frecuentes L*k*-1 de anteriores como semillas para generar conjuntos candidatos C*k*. En el recorrido *k*, actual, se obtiene el soporte de los candidatos para conocer cuáles son frecuentes. Aquellos que no son frecuentes se descartan y el resto se inserta en la lista de conjuntos frecuentes L*k*, para ser utilizados en recorridos posteriores como semillas. El proceso continúa hasta que ningún conjunto sea frecuente y se devuelve la lista L.



En la figura observamos todos los posibles conjuntos de elementos, los de color gris son los conjuntos no frecuentes.

**Algoritmo**: Apriori

****

Son dos los aspectos importantes dentro del procedimiento, la generación de candidatos y el cálculo del soporte. Para el primero, la función *apriori-gen* se encarga de generar los conjuntos candidatos en un recorrido *k*. Recibe como argumento la lista de conjuntos frecuentes del recorrido previo L*k*-1 y devuelve una lista de conjuntos candidatos Ck. La función consta de dos operaciones:

**1. Unión**: Genera conjuntos C*k*+1, al tomar la unión de los conjuntos de elementos frecuentes de tamaño *k*, P*k* y Q*k* que tienen los primeros *k*-1 elementos en común, donde;

C*k*+1 = P*k* ∪ Q*k* = {elemento1, . . . , elemento*k*-1, elementok, elemento*k*}

P*k* = {elemento1, elemento2, . . . , elemento*k*}

Q*k* = {elemento1, elemento2, . . . , elemento*k*}

donde, elemento1 < elemento2 < . . . < elemento*k* < elemento*k*+i

**2. Poda**: Eliminar todos los conjuntos candidatos c ∈ C*k*+1 cuyos subconjuntos de tamaño *k* no estén en L*k*-1.

Para ilustrar la generación de candidatos en el recorrido k=4. Sea L3 = {{1,2,3}, {1,2,4}, {1,3,4}, {1,3,5}, {2,3,4}}. Después del paso de unión, C4 será { {1,2,3,4}, {1,3,4,5}}. El paso de poda descartara el conjunto {1,3,4,5} porque el subconjunto {1,4,5} no est´a en L3. Por lo que al final C4 contendrá al conjunto candidato {1,2,3,4}. Este enfoque reduce de manera efectiva la cantidad de conjuntos considerados para contar su soporte. El segundo aspecto a considerar es el conteo de soporte, que consiste en determinar la frecuencia de ocurrencia de cada conjunto candidato generado por la función *apriori-gen*. Cada miembro del conjunto C*k* tiene dos campos: el conjunto y un contador de soporte. Para obtener el soporte de cada conjunto, se recorre la base de datos y en cada transacción se buscan los conjuntos candidatos. Si un subconjunto ocurre en una transaccion i se incrementa su respectivo contador. Al finalizar el recorrido se calcula el soporte y mantiene aquellos conjuntos cuyo soporte supere al umbral dado. La base de datos se recorre hasta que no se generen candidatos.

1. **Apriori en Spark**

La implementación en Spark de Apriori se basa en una versión de MapReduce de [25]. Que consiste en dividir la base de datos, en este caso, el conjunto de entrenamiento D en bloques y por cada bloque realizar los conteos locales de los conjuntos. Posteriormente resumir los conteos locales para obtener el total global de cada conjunto. Uno de los inconvenientes que presenta el uso de MapReduce para este algoritmo, se encuentra en que se deben ejecutar tantos trabajos como recorridos a la base de datos se realicen. Lo que implica que se debe cargar los registros de la base desde el disco para realizarlos, impactando en los tiempos para obtener resultados. Una de las ventajas de utilizar Spark se encuentra en que permite persistir en memoria dicha base de datos para acceder a ella de forma iterativa. Llevar a cabo el procedimiento en Spark radica en crear un RDDBD con las transacciones de la base de datos con la forma de la pareja (id,transacción) y mantener dicho RDDBD en memoria para facilitar el acceso para futuras consultas. RDDBD se divide en particiones por Spark y por cada una se realizan los conteos de los conjuntos, para lo que se recorre la respectiva partición. Por cada transacción se obtiene una lista de los conjuntos que ocurren en esta, en esta a lista se emite la pareja (conjunto,1) posteriormente se reducen las parejas y se obtienen los conteos globales. A partir de estos conteos se realiza el cálculo del soporte para seleccionar aquellos que superen el umbral. Siguiente, con base en los conjuntos frecuentes de tamaño *k* se generan los candidatos de tamaño *k*+1, dicha lista es enviada a los nodos para recorrer la partición que mantienen de RDDBD  y realizar los conteos de los nuevos candidatos. El proceso continúa hasta que ya no se generen más conjuntos candidatos.

1. **Implementación del algoritmo**

## **INSTALACIÓN DE PSYPARK**

!pip install pyspark

## **FUNCIONES UTILIZADAS DE PYSPARK**

1. .map() es una transformación RDD que se usa para aplicar la función de transformación (lambda) en cada elemento de RDD/DataFrame y devuelve un nuevo RDD.
2. .flatMap() Une todos los datos en un solo RDD/DataFrame después de aplicar la función en cada elemento y devuelve un nuevo RDD/DataFrame respectivamente.
3. .distinct() Lo utilizamos para eliminar los datos repetidos.
4. .reduceByKey() se utiliza para fusionar los valores de cada clave mediante una función de reducción asociativa.
5. .min() Se utiliza para recuperar el menor valor del RDD.
6. .filter() Se utiliza para filtrar los datos del RDD por medio de una definicion.
7. .cartesian() Se utiliza para realizar el producto cartesiano de ambos RDD.

**MÓDULOS**

**#Suma que nos permite reducir por medio de una clave en el RDD.**

**def Suma(x, y):**

**'''**

**Realiza la suma de 2 valores (x + y)**

**Parameters**

**=========**

**`x :` Primer valor a sumar (type: int)**

**`y :` Segundo valor a sumar (type: int)**

**Returns**

**=========**

**`x+y :` La suma de los datos de entrada (type: int)**

**'''**

**return x + y**

**def EliminarRepetidos(record):**

**"""**

**Elimina los elementos repetidos, ya que apriori reconoce A,B y B,A como iguales**

**Parameters**

**=========**

**`record :` Conjunto de datos en un arreglo**

**Returns**

**=========**

**`result o x1 :` Conjunto de datos en un arreglo**

**"""**

**if (isinstance(record[0], tuple)):**

**x1 = record[0]**

**x2 = record[1]**

**else:**

**x1 = [record[0]]**

**x2 = record[1]**

**if (any(x == x2 for x in x1) == False):**

**a = list(x1)**

**a.append(x2)**

**a.sort()**

**result = tuple(a)**

**return result**

**else:**

**return x1**

**def FiltroConf(item):**

**"""**

**Se encarga de realizar el filtro de los datos verificando la existencia donde por lo menos 1 es diferente.**

**Parameters**

**=========**

**`item :` Conjunto de datos (type:tupla)**

**Returns**

**=========**

**`item :` Devuelve el item donde conf x es mayor al conf (type:tupla)**

**"""**

**if len(item[0][0]) > len(item[1][0]):**

**if not Verificar(item[0][0], item[1][0]):**

**pass**

**else:**

**return item**

**else:**

**pass**

**def Verificar(item\_1, item\_2):**

**"""**

**Se encarga de comparar que al menos tengan un elemento en comun.**

**Parameters**

**=========**

**`item\_1 :` Conjunto de datos (type:Lista)**

**`item\_2 :` Conjunto de datos (type:Tupla)**

**Returns**

**=========**

**Booleano**

**"""**

**if len(item\_1) > len(item\_2): # confianza de x mayor a y**

**return all(any(k == l for k in item\_1) for l in item\_2)**

**else: #confianza de y mayor a x**

**return all(any(k == l for k in item\_2) for l in item\_1)**

**def Confidence(item):**

**"""**

**Calcula el nivel de confianza de cada datos previamente filtrado**

**Parameters**

**=========**

**`item :` Conjunto de datos (type:tupla)**

**Returns**

**=========**

**Conjunto de datos (type: lista)**

**"""**

**parent = set(item[0][0])**

**if isinstance(item[1][0], str):**

**child = set([item[1][0]])**

**else:**

**child = set(item[1][0])**

**parentSupport = item[0][1]**

**childSupport = item[1][1]**

**support = (parentSupport / childSupport) \* 100**

**return list([list(child), list(parent.difference(child)), support])**

## **DEFINICIÓN DE LA CLASE APRIORI**

**class Apriori:**

**def \_\_init\_\_(self, path, sc, minSupport=2):**

**#Definimos nuestras variables iniciales.**

**self.confidences = None**

**self.path = path**

**# Definimos spark context**

**self.sc = sc**

**# DEfinimos nustro soporte minimo**

**self.minSupport = minSupport**

**self.raw = self.sc.textFile(self.path) #Creamos un modelo RDD para utilizar las funciones de spark.**

**self.lblitems = self.raw.map(lambda x: x.split(','))#Map, Aplicamos un split a cada x del RDD para asi tokenizarlor y ponerlos en una lista por medio de las ','.**

**self.wlitems = self.raw.flatMap(lambda x: x.split(','))#flatMap, Aplicamos un split a cada x del RDD para asi tokenizarlo por medio de las ',' y tenerlos todo en una lista.**

**self.uniqueItems = self.wlitems.distinct() #distinct(), eliminamos todos aquellos datos que sean repetidos para asi tener nuestros datos unicos.**

**def fit(self):**

**"""**

**Se encarga de entrenar el modelo apriori para asi poder determinar el confidence de cada conjunto de datos**

**Parameters**

**=========**

**Returns**

**=========**

**`confidences :` Conjunto de datos (type: lista)**

**"""**

**supportRdd = self.wlitems.map(lambda x: (x, 1)) #Map, Convertimos nuestros datos en tuplas donde le asignamos a cada una un 1 el cual representa la vez que aparece en el conjunto de datos.**

**supportRdd = supportRdd.reduceByKey(Suma) #reduceByKey, aplicamos la funcion suma la cual se encargar de sumar las frecuencias donde tengan la misma clave para asi tener nuestra frecuencia de datos.**

**supports = supportRdd.map(lambda x: x[1]) #map, creamos una variable la cual tendra unicamente todos los soportes.**

**# Definir valor mínimo de soporte**

**if self.minSupport is 'auto': #ponemos un valor por defecto el cual sera 2 ya que queremos evitar aquellos datos que su frecuencia sea solo 1**

**minSupport = supports.min() #Si vemos que se utiliza este dato por defecto entonces calculamos el soporte minimo de un conjunto de datos**

**else:**

**minSupport = self.minSupport #Si vemos que se utiliza un dato distinto entonces pasamos a definirlo como el soporte minimo**

**# Si el soporte mínimo es 1, se tomara el valor de 2 ya que no se requiere aplicar el algoritmo apriori a frencuencias 1**

**minSupport = 2 if minSupport < 2 else minSupport**

**supportRdd = supportRdd.filter(lambda x: x[1] >= minSupport) #Filter, filtramos nuestros datos con respecto a su soporte en donde solo se toman aquellos que son mayores al soporte minimo**

**baseRdd = supportRdd.map(lambda x: ([x[0]], x[1])) #Map, Creamos una tupla la cual tendra un lista con los datos y los soportes respectivamente.**

**supportRdd = supportRdd.map(lambda x: x[0]) #Map, Creamos una variable donde solo tendra los datos que cumplen con el soporte**

**#ya que tenemos definido nuestra combinada de 1 [a] pasamos a definir la siguiente que seria combianda de 2 [a,b].**

**c = 2**

**#Creamos un while para crear las combinada correspondiendte**

**while not supportRdd.isEmpty():**

**combined = supportRdd.cartesian(self.uniqueItems) #Cartesian, realizamos el producto cartesiano entre los datos que cumple el soporte con el conjunto de datos unico.**

**combined = combined.map(lambda x: EliminarRepetidos(x)) #Map, aplicamos la funcion EliminarRepetidos ya que apriori considera A,B igual a B,A**

**combined = combined.filter(lambda x: len(x) == c) #Filter, aplocamos la comparacion del tamaño del dato con el tamaño de la combinada lo cual nos permitira descartae aquillas combinadas mayores o menores a c**

**combined = combined.distinct()#Aplicamos distinct para eliminar los repetidos.**

**combined\_2 = combined.cartesian(self.lblitems) #Cartesian, realizamos el producto cartesiano entre todo el conjunto de datos qie cumplen la combianda de c con de datos existentes en el input**

**#nos permite verificar que la combinada existe en por lo menos en alguno de los datos de entrada**

**combined\_2 = combined\_2.filter(lambda item: all(x in item[1] for x in item[0])) #Filter, filtramos todos aquellos datos que cumplan la funcion all.**

**combined\_2 = combined\_2.map(lambda x: x[0]) #Map, creamos una variable con todos los datos combinados.**

**combined\_2 = combined\_2.map(lambda x: (x, 1)) #map, creamos una tupla la cual nos permitira saber la frecuencia de dicho dato.**

**combined\_2 = combined\_2.reduceByKey(Suma) #reduceByKey, aplicamos la funcion de suma y asi saber la frecuencia de cada combinada.**

**combined\_2 = combined\_2.filter(lambda x: x[1] >= minSupport) #filter, filtramos tos aquellos datos que tiene un soporte mayor o igual al de entrada.**

**baseRdd = baseRdd.union(combined\_2) #Realizamos la combinada de el conjunto de las frecuencias con la combinada correspondiente.**

**combined\_2 = combined\_2.map(lambda x: x[0]) #Realizamos la funcion map para asi guardas todos los datos que cumplen con el soporte.**

**supportRdd = combined\_2**

**c = c + 1 #oncrementamos c hasta que se cumplan todas las iteraciones.**

**sets = baseRdd.cartesian(baseRdd) #Cartesian, realizamos el producto cartesiano de baseRDD con BaseRDD.**

**filtered = sets.filter(lambda x: FiltroConf(x))#Aplicamos el filter a sets el cual utilizare la funcion filtroConf.**

**confidences = filtered.map(lambda x: Confidence(x)) #Aplicamos la funcion map y a este mismo le pasamos la funcion confidence la cual nos permitira ver si se cumple la confidencia o no**

**self.confidences = confidences #Guardamos la informacion**

**return confidences #Retornamos la confidencia**

**def BusquedaEspecifica(self, set, confidence):**

**"""**

**Se encarga de mostrar los datos que cumplen con el nivel de confianza y el dato set**

**Parameters**

**=========**

**`set :` Conjunto de datos para busqueda (type: Lista)**

**`confidences :` Valor de confidences a comparar (type:float)**

**Returns**

**=========**

**`filtered :` Conjunto de datos (type: lista)**

**"""**

**if not isinstance(set, list): #Verificamos que el dato de entrada no sea una lista**

**raise ValueError('For prediction "set" argument should be a list')**

**\_confidences = self.confidences**

**\_FiltraBusqueda = self.\_FiltraBusqueda**

**filtered = \_confidences.filter(lambda x: \_FiltraBusqueda(x, set, confidence)) #Filterer aplicamos la funcion \_FiltrarBusqueda la cual nos permitira recuperar si el confidence se cumple o no.**

**return filtered #Retronamos los datos filtrados**

**@staticmethod**

**def \_FiltraBusqueda(item, set, confidence):**

**"""**

**Se encarga de buscar los datos que cumplen con el confidence y tiene el mismo valor de set dentro de los confidence hallados anteriormente.**

**=========**

**`ítem :` Conjunto de datos (type: Lista)**

**`set :` Conjunto de datos para comparación (type: Lista)**

**`confidences :` Valor de confidences a comparar (type:float)**

**Returns**

**=========**

**`filtered :` Conjunto de datos (type: lista)**

**"""**

**it = item[0] #Recuperamos el primer dato de item**

**it.sort() #PAsamos a ordenarlo**

**set.sort() #Ordenmos los datos de set**

**if it == set and item[2] >= confidence: #Verificamos que cumpla 2 condiciones las cuales son que it == a set y que el conficende del dato sea mayor al confidence pasado como entrada**

**return item**

**else:**

**pass**

## **DEFINICIÓN DE SPARKCONTEXT**

**#Importamos SparkContext.**

**from pyspark import SparkContext**

**sc = SparkContext()**

## **FILTRAR DATOS**

#Pasamos nuestro conjunto de datos, que habiando visto la clase de apriori permite el ingreso de datos separados por comas y sin encabezado

path = "data.txt"

# Creamos el objeto apriori

#Pasamos el conjunto de datos

#PAsamos el sparkcontext creado

#PAsamos minSupport en auto.

apriori = Apriori(path, sc, minSupport="auto")

#Entrenamos nuestro modelo de apriori.

apriori.fit()

#for i in range(1,6):

#Pasamos a hacer la busqueda especifica.

#PAsamos el valor de la informacion que queremos buscar

#PAsamos el valor del confidence que debe cumplir

# pr = apriori.BusquedaEspecifica(['I' + str(i)], 60)

#Guardamos todos los datos de la lista en una variable

# result = pr.collect()

#Mostramos los resultados.

# print(result)

pr = apriori.BusquedaEspecifica(['I2'], **40**)

result = pr.collect()

**print**(result)

**4. Ejecución del programa**

**5. Conclusiones**

El algoritmo Apriori es un algoritmo eficiente que escanea la base de datos solo una vez.

Reduce considerablemente el tamaño de los conjuntos de elementos en la base de datos proporcionando un buen rendimiento. Por lo tanto, la minería de datos ayuda mejor a los consumidores y las industrias en el proceso de toma de decisiones.

Apriori es un algoritmo sencillo que aprende rápidamente las reglas de asociación entre elementos (puntos de datos). Si bien se expuso a través de su uso para el análisis de la canasta de mercado, también hay muchas otras aplicaciones prácticas, incluida la bioinformática (secuenciación de proteínas), el diagnóstico médico (relación entre los síntomas y la enfermedad) o el análisis de datos del censo.

Una característica para usar el algoritmo Apriori en grandes conjuntos de datos es la elección del umbral mínimo de soporte. Si no tiene cuidado, puede quedarse sin memoria rápidamente con una cantidad potencialmente enorme de conjuntos de elementos de tamaño 2.

**6. Bibliografía**

<https://towardsdatascience.com/big-data-market-basket-analysis-with-apriori-algorithm-on-spark-9ab094b5ac2c>

<https://towardsdatascience.com/apriori-algorithm-for-association-rule-learning-how-to-find-clear-links-between-transactions-bf7ebc22cf0a>

<https://www.softwaretestinghelp.com/apriori-algorithm/>