

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 del Algoritmo Apriori en PYSPARK”

ASIGNATURA : Minería de Datos

DOCENTE : PhD. Carlos Fernando Montoya Cubas

INTEGRANTES :

- | | |
|------------------------------------|--------|
| • Bustamante Flores, Erick Andrew | 171943 |
| • Huancara Ccolque, Alex Helder | 174911 |
| • Sarco Jacinto, Daniel Eduardo | 174452 |
| • Quispe Yahaira, Ronaldo | 171866 |
| • Vega Centeno Olivera, Ronaldinho | 140934 |

Cusco – Perú

2022

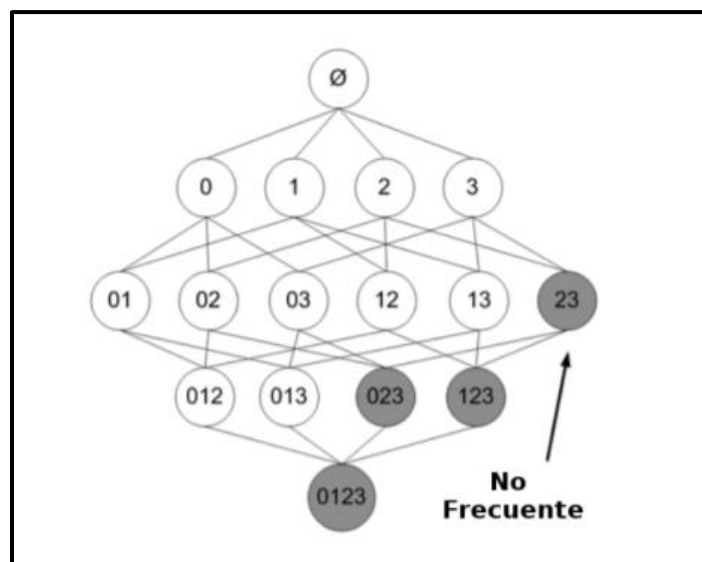
CONTENIDO

ALGORITMO APRIORI PYSPARK-----	2
1. Algoritmo Apriori-----	2
2. Apriori en Spark-----	3
3. Implementación del algoritmo -----	4
3.1. Instalación de PySpark -----	4
3.2. Funciones utilizadas de PySpark -----	4
3.3. Módulos -----	4
3.4. Clase Apriori-----	7
3.5. Definición de SparkContext-----	11
3.6. Ejecución del algoritmo-----	11
3.7. Detalle de la ejecución -----	11
3.8. Crear DF con las reglas y guardar -----	12
3.9. Resultados-----	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 cuáles 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:

```
Result:  $L_k$ 
 $L_1 = \{\text{conjuntos candidatos de tamaño 1}\}$ 
for  $k \leftarrow 2$  to  $L_{k-1} \neq \emptyset$  do
     $C_k = \text{apriori-gen}(L_{k-1})$ ; //Generar nuevos candidatos
    forall the transacciones  $t \in D$  do
         $C_t = \text{subconjunto}(C_k, t)$ ; //Candidatos contenidos en  $t$ 
        forall the candidatos  $c \in C_t$  do
            |  $c.\text{contador}++$ ;
        end forall
    end forall
     $L_k = \{c \in C_k \mid c.\text{contador} \geq \text{soporte minimo}\}$ 
end for
```

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 C_k . La función consta de dos operaciones:

- A. 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;

$$C_{k+1} = P_k \cup Q_k = \{\text{elemento}_1, \dots, \text{elemento}_{k-1}, \text{elemento}_k, \text{elemento}_k\}$$

$$P_k = \{\text{elemento}_1, \text{elemento}_2, \dots, \text{elemento}_k\}$$

$$Q_k = \{\text{elemento}_1, \text{elemento}_2, \dots, \text{elemento}_k\}$$

donde, $\text{elemento}_1 < \text{elemento}_2 < \dots < \text{elemento}_k < \text{elemento}_{k+1}$

- B. Poda:** Eliminar todos los conjuntos candidatos $c \in C_{k+1}$ cuyos subconjuntos de tamaño k no estén en L_{k-1} .

Para ilustrar la generación de candidatos en el recorrido $k=4$. Sea $L_3 = \{\{1,2,3\}, \{1,2,4\}, \{1,3,4\}, \{1,3,5\}, \{2,3,4\}\}$. Después del paso de unión, C_4 será $\{\{1,2,3,4\}, \{1,3,4,5\}\}$. El paso de poda descartará el conjunto $\{1,3,4,5\}$ porque el subconjunto $\{1,4,5\}$ no está en L_3 . Por lo que al final C_4 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 transacción 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.

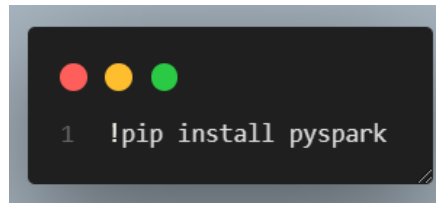
2. Apriori en Spark

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 RDD_{BD} con las transacciones de la base de datos con la forma de la pareja (id,transacción) y mantener dicho RDD_{BD} en memoria para facilitar el acceso para futuras consultas. RDD_{BD} 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. Siguiendo, 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 RDD y realizar los conteos de los nuevos candidatos. El proceso continúa hasta que ya no se generen más conjuntos candidatos.

3. Implementación del algoritmo

3.1. Instalación de PySpark

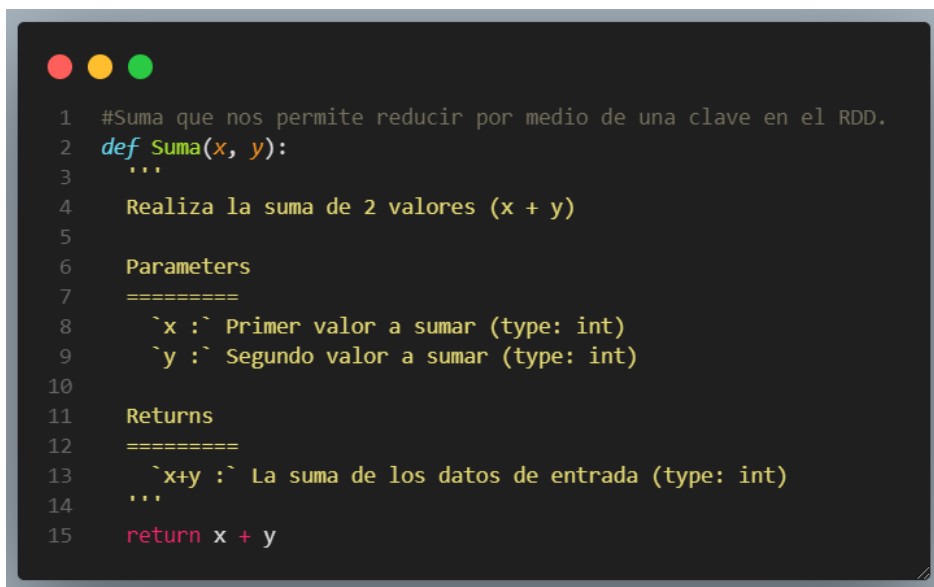


```
1 !pip install pyspark
```


3.2. 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 definición.
7. **.cartesian()** Se utiliza para realizar el producto cartesiano de ambos RDD.

3.3. Módulos




```
1 #Suma que nos permite reducir por medio de una clave en el RDD.
2 def Suma(x, y):
3     ...
4     Realiza la suma de 2 valores (x + y)
5
6     Parameters
7     =====
8     `x` :` Primer valor a sumar (type: int)
9     `y` :` Segundo valor a sumar (type: int)
10
11     Returns
12     =====
13     `x+y` :` La suma de los datos de entrada (type: int)
14     ...
15     return x + y
```



```

1 def EliminarRepetidos(record):
2     """
3     Elimina los elementos repetidos, ya que apriori reconoce A,B y B,A como iguales
4
5     Parameters
6     =====
7     `record :` Conjunto de datos en un arreglo
8
9     Returns
10    =====
11    `result o x1 :` Conjutno de datos en un arreglo
12
13    """
14    if (isinstance(record[0], tuple)):
15        x1 = record[0]
16        x2 = record[1]
17    else:
18        x1 = [record[0]]
19        x2 = record[1]
20
21    if (any(x == x2 for x in x1) == False):
22        a = list(x1)
23        a.append(x2)
24        a.sort()
25        result = tuple(a)
26        return result
27    else:
28        return x1

```



```

1 def FiltroConf(item):
2     """
3     Se encarga de realizar el filtro de los datos verificando la existencia
4     donde por lo menos 1 es diferente.
5
6     Parameters
7     =====
8     `item :` Conjunto de datos (type:tupla)
9
10    Returns
11    =====
12    `item :` Devuelve el item donde conf x es mayor al conf (type:tupla)
13
14    """
15    if len(item[0][0]) > len(item[1][0]):
16        if not Verificar(item[0][0], item[1][0]):
17            pass
18        else:
19            return item
20    else:
21        pass

```



```
1 def Verificar(item_1, item_2):
2     """
3     Se encarga de comparar que al menos tengan un elemento en comun.
4     Parameters
5     =====
6     `item_1` : Conjunto de datos (type:Lista)
7     `item_2` : Conjunto de datos (type:Tupla)
8
9     Returns
10    =====
11    Booleano
12
13    """
14    if len(item_1) > len(item_2): # confianza de x mayor a y
15        return all(any(k == l for k in item_1 for l in item_2)
16    else: #confianza de y mayor a x
17        return all(any(k == l for k in item_2 for l in item_1)
```



```
1 def Confidence(item):
2     """
3     Calcula el nivel de confianza de cada datos previamente filtrado
4     Parameters
5     =====
6     `item` : Conjunto de datos (type:tupla)
7
8     Returns
9     =====
10    Conjunto de datos (type: lista)
11
12    """
13    parent = set(item[0][0])
14
15    if isinstance(item[1][0], str):
16        child = set([item[1][0]])
17    else:
18        child = set(item[1][0])
19    parentSupport = item[0][1]
20    childSupport = item[1][1]
21
22    support = (parentSupport / childSupport) * 100
23
24    return list([list(child), list(parent.difference(child)), support])
```

```

1  def print_reglas(reglas):
2      """
3      Imprime las reglas de asociacion, confidence
4
5      ej. :
6
7          REGLA N° 1
8          Regla      : [1] => [2, 3]
9          Confianza   : 200
10
11      -----
12
13      """
14      cont=1
15      for i in reglas:
16          print('REGLA N°',cont)
17          print('Regla      : ',set(i[0]),"=>",set(i[1]))
18          print('Confianza   : ',i[2]*100)
19          print("_____")
20          cont+=1

```

3.4. Clase Apriori

```

1  class Apriori:
2      def __init__(self, path, sc, minSupport=2):
3          #Definimos nuestras variables iniciales.
4          self.confidences = None
5          self.path = path
6
7          # Definimos spark context
8          self.sc = sc
9
10         # Definimos nuestro soporte minimo
11         self.minSupport = minSupport
12         #Creamos un modelo RDD para utilizar las funciones de spark.
13         self.raw = self.sc.textFile(self.path)
14         #Map, Aplicamos un split a cada x del RDD para asi
15         # tokenizarlor y ponerlos en una lista por medio de las ','.
16         self.lblitems = self.raw.map(lambda line: line.split(','))
17
18         #flatMap, Aplicamos un split a cada x del RDD para asi
19         #tokenizarlo por medio de las ',' y tenerlos todo en una lista.
20         self.wlitems = self.raw.flatMap(lambda line: line.split(','))
21
22         #distinct(), eliminamos todos aquellos datos que
23         #sean repetidos para asi tener nuestros datos unicos.
24         self.uniqueItems = self.wlitems.distinct()

```




```
1 def fit(self):
2     """
3     Se encarga de entrenar el modelo apriori para asi poder determinar el
4     confidence de cada conjunto de datos
5
6     Parameters
7     =====
8
9     Returns
10    =====
11    `confidences` : Conjunto de datos (type: lista)
12
13    """
14    #Map, Convertimos nuestros datos en tuplas donde le asignamos a cada
15    #una un 1 el cual representa la vez que aparece en el conjunto de datos.
16    supportRdd = self.wlitems.map(lambda item: (item, 1))
17    #reduceByKey, aplicamos la funcion suma la cual se encargara
18    # de sumar las frecuencias donde tengan la misma clave para
19    # asi tener nuestra frecuencia de datos.
20    supportRdd = supportRdd.reduceByKey(Suma)
21    #map, creamos una variable la cual tendra unicamente todos los soportes.
22    supports = supportRdd.map(lambda item: item[1])
23
24    # Definir valor mínimo de soporte
25
26    #ponemos un valor por defecto el cual sera 2 ya que queremos
27    # evitar aquellos datos que su frecuencia sea solo 1
28    if self.minSupport is 'auto':
29        #Si vemos que se utiliza este dato por defecto entonces calculamos
30        #el soporte minimo de un conjunto de datos
31        minSupport = supports.min()
32    else:
33        #Si vemos que se utiliza un dato distinto entonces pasamos
34        #a definirlo como el soporte minimo
35        minSupport = self.minSupport
36
37    # Si el soporte mínimo es 1, se tomara el valor de 2 ya que
38    #no se requiere aplicar el algoritmo apriori a frecuencias 1
39    minSupport = 2 if minSupport < 2 else minSupport
40    #Filter, filtramos nuestros datos con respecto a su soporte en donde
41    # solo se toman aquellos que son mayores al soporte minimo
42    supportRdd = supportRdd.filter(lambda item: item[1] >= minSupport)
43    #Map, Creamos una tupla la cual tendra una lista
44    #con los datos y los soportes respectivamente.
45    baseRdd = supportRdd.map(lambda item: ([item[0]], item[1]))
46    #Map, Creamos una variable donde solo tendra
47    #los datos que cumplen con el soporte
48    supportRdd = supportRdd.map(lambda item: item[0])
49
```

```

50     #ya que tenemos definido nuestra combinada de 1 [a] pasamos
51     #a definir la siguiente que seria combianda de 2 [a,b].
52     c = 2
53     #Creamos un while para crear las combinada correspondiente
54     while not supportRdd.isEmpty():
55         #Cartesian, realizamos el producto cartesiano entre
56         # los datos que cumple el soporte con el conjunto de datos unico.
57         combined = supportRdd.cartesian(self.uniqueItems)
58         #Map, aplicamos la funcion EliminarRepetidos ya que
59         # apriori considera A,B igual a B,A
60         combined = combined.map(lambda item: EliminarRepetidos(item))
61         #Filter, aplicamos la comparacion del tamaño del dato
62         # con el tamaño de la combinada lo cual nos permitira
63         #descartar aquellas combinadas mayores o menores a c
64         combined = combined.filter(lambda item: len(item) == c)
65         #Aplicamos distinct para eliminar los repetidos.
66         combined = combined.distinct()
67         #Cartesian, realizamos el producto cartesiano entre
68         #todo el conjunto de datos que cumplen la combianda
69         # de c con de datos existentes en el input
70         #nos permite verificar que la combinada existe en
71         # por lo menos en alguno de los datos de entrada
72         combined_2 = combined.cartesian(self.lblitems)
73         #filter, filtramos todos aquellos datos que cumplan la funcion all.
74         combined_2 = combined_2.filter(lambda item: all(x in item[1] for x in item[0]))
75
76         #Map, creamos una variable con todos los datos combinados.
77         combined_2 = combined_2.map(lambda item: item[0])
78         #map, creamos una tupla la cual nos permitira
79         #saber la frecuencia de dicho dato.
80         combined_2 = combined_2.map(lambda item: (item, 1))
81         #reduceByKey, aplicamos la funcion de suma
82         # y así saber la frecuencia de cada combinada.
83         combined_2 = combined_2.reduceByKey(Suma)
84         #filter, filtramos los aquellos datos que
85         #tiene un soporte mayor o igual al de entrada.
86         combined_2 = combined_2.filter(lambda item: item[1] >= minSupport)
87
88         #Realizamos la combinada de el conjunto de las
89         #frecuencias con la combinada correspondiente.
90         baseRdd = baseRdd.union(combined_2)
91         #Realizamos la funcion map para así guardas todos
92         #los datos que cumplen con el soporte.
93         combined_2 = combined_2.map(lambda item: item[0])
94         supportRdd = combined_2
95         #oncrementamos c hasta que se cumplan todas las iteraciones.
96         c = c + 1
97         #Cartesian, realizamos el producto cartesiano de baseRDD con BaseRDD.
98         sets = baseRdd.cartesian(baseRdd)
99         #Aplicamos el filter a sets el cual utilizare la funcion filtroConf.
100        filtered = sets.filter(lambda item: FiltroConf(item))
101        #Aplicamos la funcion map y a este mismo le pasamos la funcion
102        #confidence la cual nos permitira ver si se cumple la confidencia o no
103        confidences = filtered.map(lambda item: Confidence(item))
104        #Guardamos la informacion
105        self.confidences = confidences
106
107        #Retornamos la confidencia
108        return confidences

```

```

1  def BusquedaEspecifica(self, set, confidence):
2      """
3          Se encarga de mostrar los datos que cumplen con el nivel de confianza
4          y el dataset
5
6          Parameters
7          =====
8          `set` :` Conjunto de datos para busqueda (type: Lista)
9          `confidences` :` Valor de confidences a comparar (type:float)
10
11          Returns
12          =====
13          `filtered` :` Conjunto de datos (type: lista)
14
15          """
16          #Verificamos que el dato de entrada no sea una lista
17          if not isinstance(set, list):
18              raise ValueError('For prediction "set" argument should be a list')
19
20          _confidences = self.confidences
21          _FiltroBusqueda = self._FiltroBusqueda
22          #Filterer aplicamos la funcion _FiltrarBusqueda la cual nos permitira
23          #recuperar si el confidence se cumple o no.
24          filtered = _confidences.filter(lambda x: _FiltroBusqueda(x, set, confidence))
25          #Retronamos los datos filtrados
26          return filtered

```

```

1  @staticmethod
2  def _FiltroBusqueda(item, set, confidence):
3      """
4          Se encarga de buscar los datos que cumplen con el confidence y tiene el
5          mismo valor de set dentro de los confidence hallados anteriormente.
6
7          Parameters:
8          =====
9          `item` :` Conjunto de datos (type: Lista)
10          `set` :` Conjunto de datos para comparacion (type: Lista)
11          `confidences` :` Valor de confidences a comparar (type:float)
12
13          Returns
14          =====
15          `filtered` :` Conjunto de datos (type: lista)
16
17          """
18          #Recuperamos el primer dato de item
19          it = item[0]
20          #PASamos a ordenarlo
21          it.sort()
22          #Ordenmos los datos de set
23          set.sort()
24          #Verificamos que cumpla 2 condiciones las cuales son que it == a set y
25          #que el conficende del dato sea mayor al confidence pasado como entrada
26          if it == set and item[2] >= confidence:
27              return item
28          else:
29              pass

```

3.5. Definición de SparkContext

```
1 #Importamos SparkContext.
2 from pyspark import SparkContext
3 sc = SparkContext("local", "First App")
```

3.6. Ejecución del algoritmo

```
1 import time
2 #Pasamos nuestro conjunto de datos, que habiendo visto la clase de apriori
3 #permite el ingreso de datos separados por comas y sin encabezado
4 path = "comestibles2.txt"
5 # Creamos el objeto apriori
6 #Pasamos el conjunto de datos
7 #PASamos el sparkcontext creado
8 #PASamos minSupport en auto.
9 inicio=time.time()
10 apriori = Apriori(path, sc, minSupport=100)
11 #Entrenamos nuestro modelo de apriori.
12 apriori.fit()
13 fin=time.time()
14 tiempo = fin-inicio
```

3.7. Detalle de la ejecución

```
1 print('-----')
2 print('                DETALLES                ')
3 print('-----')
4 Productos=apriori.uniqueItems.collect()
5 listas=apriori.lblitems.collect()
6 reglas=apriori.confidences.collect()
7 print('| Total de productos distintas           :', len(Productos))
8 print('| Total de lista de compras(transacciones) :', len(listas))
9 print('| Total de reglas generadas                :',len(reglas))
10 print('| Soporte minimo           :',apriori.minSupport)
11 print('| Tiempo de ejecucion ----> ',round(tiempo,4),'seg.')
12 print('-----')
```

3.8. Crear DF con las reglas y guardar

```
1 import pandas as pd
2 data_array=apriori.confidences.collect()
3 column_names = ["Antecedente", "Consecuente", "Confidence"]
4 data_df = pd.DataFrame(data_array, columns=column_names)
5 data_df
```

3.9. Resultados

	Antecedente	Consecuente	Confidence
0	[citrus fruit]	[whole milk]	39.016393
1	[whole milk]	[citrus fruit]	13.492063
2	[citrus fruit]	[other vegetables]	35.409836
3	[yogurt]	[whole milk]	42.738589
4	[whole milk]	[yogurt]	23.356009
5	[other vegetables]	[citrus fruit]	16.023739
6	[tropical fruit]	[whole milk]	42.307692
7	[whole milk]	[tropical fruit]	16.213152
8	[whole milk]	[other vegetables]	28.684807
9	[tropical fruit]	[other vegetables]	37.278107
10	[whole milk]	[bottled water]	14.512472
11	[other vegetables]	[whole milk]	37.537092
12	[other vegetables]	[tropical fruit]	18.694362
13	[bottled water]	[whole milk]	30.117647
14	[whole milk]	[curd]	11.337868

REGLA N° 1
Regla : {'whole milk'} => {'citrus fruit'}
Confianza : 13.492063492063492
REGLA N° 2
Regla : {'whole milk'} => {'yogurt'}
Confianza : 23.356009070294785
REGLA N° 3
Regla : {'whole milk'} => {'tropical fruit'}
Confianza : 16.213151927437643
REGLA N° 4
Regla : {'whole milk'} => {'other vegetables'}
Confianza : 28.68480725623583
REGLA N° 5
Regla : {'whole milk'} => {'bottled water'}
Confianza : 14.512471655328799
REGLA N° 6
Regla : {'whole milk'} => {'curd'}
Confianza : 11.337868480725625
REGLA N° 7
Regla : {'whole milk'} => {'soda'}
Confianza : 16.666666666666664

4. 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.

5. Bibliografía

Frequent Pattern Mining. Spark 2.4.0 Documentation. Retrieved 2019-01-31.

Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1), 55–86. doi:10.1007/s10618-006-0059-1

<https://spark.apache.org/>

Rakesh Agrawal, Sakti P. Ghosh, Tomasz Imielinski, Balakrishna R. Iyer, Arun N. Swami: An Interval Classifier for Database Mining Applications. *VLDB* 1992: 560-573

6. Linkografía

<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/>