



Facultad de Ciencias

Reescribir slatt en Python 3

Rewrite slatt in Python 3

Trabajo de fin de grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Alejandro Martínez Floranes

Director: Domingo Gómez Pérez

julio de 2021

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Agradecimientos

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Resumen

palabras clave: reglas de asociación, hipergrafos, slatt, refactorizar

Las reglas de asociación son objetos matemáticos empleados de forma extensa en disciplinas como la minería de datos, aprendizaje automático y representación del conocimiento, entre otros campos. Slatt es un proyecto de software libre desarrollado por José Luis Balcázar (Universidad Politécnica de Barcelona). Ofrece funcionalidades para el cálculo de reglas de asociación. Para ello, se apoya en implementaciones del algoritmo a priori para el cálculo de clausuras, el retículo de las clausuras y, entre otras funcionalidades, devuelve las reglas representativas para cualquier elección de los parámetros de soporte y confianza.

En este proyecto, se ha mejorado este software utilizando como base las implementaciones disponibles en Slatt aplicadas a:

- hipergrafos y algoritmos con aplicación a estos objetos;
- cálculo de clausuras y retículos (lattices);

Este trabajo ha requerido la búsqueda y análisis de algoritmos propuestos en la literatura científica sobre los puntos anteriores. El lenguaje de desarrollo será Python3, encontrándose la implementación anterior en Python 2.7

Rewrite slatt in Python 3

Abstract

keywords: association rules, hypergraph, slatt, refactor

Association rules are mathematical objects used extensively in disciplines such as data mining, machine learning, and knowledge representation, among other fields. Slatt is a free software project developed by José Luis Balcázar (Polytechnic University of Barcelona). It offers functionalities for the calculation of association rules. To do this, it relies on implementations of the a priori algorithm for the calculation of closures, the lattice of closures and, among others functionalities, returns the representative rules for any choice of support and trust parameters.

In this project, this software has been improved using as a basis the implementations available in Slatt applied to:

- hypergraphs and algorithms with application to these objects;
- calculation of closures and lattices (lattices);

This work has required the search and analysis of algorithms proposed in the scientific literature on the previous points. The development language will be Python3, the previous implementation being in Python 2.7

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Índice

1. Introducción	1
1.1. Reglas de asociación	1
1.2. Metodología y Requisitos	3
1.3. Diagrama de clases UML	4
1.4. Diagrama de Gantt	6
2. Metaprogramación	7
2.1. Conceptos básicos	7
2.2. Charla metaprogramación MIT	8
2.2.1. Metaclases	8
2.2.2. Clase Type	9
2.2.3. Usos de la Metaprogramación	9
2.2.4. Construir sistemas	9
3. Hipergrafo y problemas relacionados	11
3.1. Conceptos básicos	11
3.2. Problemas relacionados con SIMPLE-H-SAT en la teoría de bases de datos	12
3.3. Problemas relacionados con las transversales de los hipergrafos y su saturación	13
3.4. Axiomas de Armstrong	13
3.4.1. Ejemplo de axiomas de Armstrong	13
4. KD Trees	15
4.1. Conceptos básicos	15
4.2. Implementación en Reglas de Asociación	15
4.3. Implementación KD-Trees en Slatt	15
4.4. Ejemplo gráfico KD Tree	16
5. Desarrollo Software	17
5.1. Ejecución y funcionamiento de Slatt	17
5.2. Test unitarios	17
5.3. Mejoras del código	18
5.3.1. Diferencias entre python2 y python3	18
5.3.2. Cambios realizados en las clases	18
5.4. Discusión y resultados de la mejora	21
6. Conclusiones	23
Referencias	25

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

1 Introducción

1.1. Reglas de asociación

Este proyecto se basa principalmente en reescribir **slatt** en Python 3 e incluir diferentes mejoras en el código para mejorar tanto su eficiencia en cuanto a rendimiento, como en reducción de líneas de código, y realizar una mejora en la legibilidad del propio código.

Slatt es un proyecto de software libre que ofrece funcionalidades para el cálculo de reglas de asociación.

Las reglas de asociación son declaraciones 'if-then' que ayudan a mostrar la probabilidad de relaciones entre elementos de datos, dentro de grandes conjuntos de datos. Las medidas de evaluación de las reglas de asociación son: *el soporte, la confianza y el lift*.

- El **soporte** es la fracción de las transacciones que contiene tanto a X como a Y.
- La **confianza** es la fracción de las transacciones en las que aparece X que también incluyen a Y; esto es, la confianza mide con que frecuencia aparece Y en las transacciones que incluyen X.
- El **lift** es la confianza de la reglas dividido por el cociente del consecuente.

$$\begin{array}{c} \text{Rule: } X \Rightarrow Y \begin{cases} \nearrow \text{Support} = \frac{\text{frq}(X,Y)}{N} \\ \rightarrow \text{Confidence} = \frac{\text{frq}(X,Y)}{\text{frq}(X)} \\ \searrow \text{Lift} = \frac{\text{Support}}{\text{Supp}(X) \times \text{Supp}(Y)} \end{cases} \end{array}$$

Ilustración 1: Association Rules

A su vez, a la hora de plantear un problema utilizando reglas de asociación necesitamos disponer de diferentes elementos:

1. **Itemset.-** Conjunto de uno o más items (artículos).
2. **K-itemset.-** Itemset con k elementos.
3. **Soporte de un itemset.-** Fracción de las transacciones que contienen el itemset. **Itemset Frecuente** Itemset con soporte igual o superior a un umbral de soporte establecido por el usuario.
4. **Umbral mínimo de confianza y de soporte.-** estos umbrales se utilizan para coger las reglas cuyo soporte o confianza sean mayores que las de umbral mínimo.

Ejemplo sencillo de como utilizar las reglas de asociación para un problema general (problema de la lista de la compra). [Berzal, 2016]

TID	Artículos
1	Pan, leche, huevos
2	Pan, pañales, cerveza
3	Leche, pañales, cerveza
4	Pan, leche, pañales, cerveza
5	Pan, leche, huevos, cerveza

Cuadro 1: Problema reglas de asociación

- El **soporte** de la cerveza, $\text{supp}(\text{cerveza}) = 4/5 = 0.8$
- El **soporte** de los pañales, $\text{supp}(\text{pañales}) = 3/5 = 0.6$
- El **soporte** de la cerveza y los pañales, $\text{supp}(\text{cerveza}, \text{pañales}) = 3/5 = 0.6$
- La **confianza** entre la cerveza y los pañales seria, $\text{supp}(\text{cerveza}, \text{pañales}) / \text{supp}(\text{cerveza}) = 0.6 / 0.8 = 0.75$

Para implementar este tipo de operaciones en código python, Slatt se basa principalmente en el algoritmo *Apriori*, el proceso de este algoritmo sigue dos pasos: [Morales y Gonzales, 2013]

1. Genera los itemsets.

- Genera todos los itemsets con un elemento.
- Usa estos para generar los de dos elementos, y así sucesivamente.
- Toma todos los que cumplen con el mínimo soporte (esto permite eliminar posibles combinaciones).

2. Genera las reglas revisando que cumplan con el criterio mínimo de confianza.

```

1 def apriori(dataSet,minSupport=0.5):
2     '''
3         Función total
4     '''
5     C1 = createC1(dataSet)
6     D = map(set, dataSet)
7     L1, supportData = scanD(D, C1, minSupport)
8     L = [L1]
9     k = 2
10    while (len(L[k-2]) > 0):
11        Ck = aprioriGen(L[k-2], k)
12        Lk, supK = scanD(D, Ck, minSupport)
13        supportData.update(supK)
14        L.append(Lk)
15        k += 1
16    return L,supportData

```

Listing 1.1: Apriori Algorithm python version

1.2. Metodología y Requisitos

Para la realización del proyecto se ha utilizado una metodología incremental, debido a que dicho proyecto esta dividido en diferentes partes, las cuales unas no tienen relación directa con otras, es por esto, que he decidió usar este tipo de metodología.

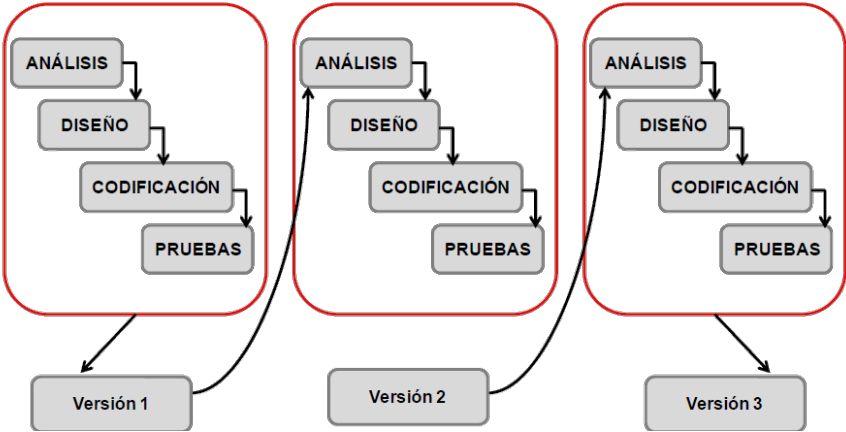


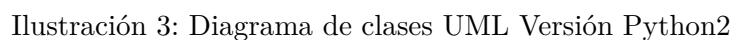
Ilustración 2: Metodología iterativa incremental

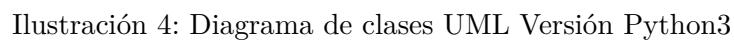
A continuación, defino los requisitos funcionales y no funcionales del proyecto software, al no ser una aplicación destinada hacia un usuario(cliente), los requisitos no funcionales, se basan en el correcto funcionamiento de la aplicación.

Identificador	Descripción
RNF01	La aplicación deberá tener la misma funcionalidad tanto en python2 como en python3
RNF02	La mejora en python3 deberá ser mas eficiente que la de su anterior versión
RNF03	La mejora en python3 deberá tener menos lineas de código que su anterior versión
RNF04	La refactorización de los métodos no supondrá un cambio en los resultados

Cuadro 2: Requisitos no funcionales del proyecto

En este apartado indico el diagrama de clases UML inicial de Slatt escrito en python2 [3] y el resultado final del proyecto al reescribirlo a python3 [4] con las nuevas clases, herencias, etc.





1.4. Diagrama de Gantt

En esta otra sección indico como ha sido dividido el proyecto en función del tiempo con un diagrama de Gantt 5 en el que se ve que tareas se han ido realizando durante el tiempo que el proyecto a estado activo.

	Semana 1	Semana 2	Semana 3	Semanas 4 y 5	Semana 5	Semanas 6 y 7	Semanas 7 y 8	Semanas 9 10 11
Revisión del codigo								
Reescritura del código parte 1								
Investigación hipergrafos y problemas relacionados								
Investigación Metaprogramación								
Reescritura del código parte 2								
Investigación KD trees								
Test unitarios								
Corrección del código								
Finalizar memoria								

Ilustración 5: Diagrama de clases UML Versión Python3

2 Metaprogramación

2.1. Conceptos básicos

En este proyecto se van a utilizar diferentes herramientas relacionadas con la metaprogramación en el entorno de desarrollo.

La metaprogramación consiste en escribir programas que manipulan otros programas, es decir, cuando un programa es capaz de recibir una entrada, modificar el contenido de dicha entrada durante su ejecución y producir como salida otro programa, es un metaprograma.

La metaprogramación se base en estos conceptos principales:

- **Reflexión.-** La capacidad de un programa para observar y modificar su estructura.

Un ejemplo de reflexión seria cuando el código fuente de un programa se compila, se suele perder información sobre la estructura del programa, pero si un sistema permite reflexión, se mantiene dicha estructura como metadatos en el programa generado.

La reflexión se puede descomponer en dos partes:

1. **Introspección.-** La capacidad de un lenguaje de obtener información sobre si mismo.

Un ejemplo de introspección seria java, en java se puede saber hasta nivel del método cuantos atributos, que tipos utiliza, etc.

2. **Intercesión.-** La capacidad de modificar el propio comportamiento/ significado de la estructura del programa.

- **Reificación.-** La capacidad de convertir algo abstracto en un dato explícito. Ejemplos de reificación serian:

- Crear un tipo de satos para acceder a una posición de memoria
- Crear estructuras de datos que representan tipos abstractos de datos

En conclusión la metaprogramación es la programación que utiliza los programas como datos y permite modificar su propia estructura. [Silva Galiana, 2018], aunque otra definición valida seria el conjunto de cosas que tienen más que ver con el proceso que con escribir código o trabajar de manera más eficiente.

2.2. Charla metaprogramación MIT

David Bleazley en esta charla explica uno de los grandes cambios que existen en Python3 respecto de Python2, la metaprogramación.[Bleazley, 2013]

En concreto, explica como funcionan los decoradores, la clase *decorators*, los descriptores y las metaclasses, estos elementos son usados en este proyecto para mejorar el rendimiento de slatt y reducir a su vez lineas de código.

2.2.1. Metaclasses

Una clase es un objeto que se utiliza para crear instancias de nuevos objetos. Una metaclass está por encima de una clase. Agrupa un conjunto de clases, es decir, podemos tener metainformación sobre clases dentro de una metaclass. Es importante saber que en Python todo es un objeto, incluidas las clases, y cada una de las clases es creada por una metaclass.

Por ejemplo:

```
def prueba_metaprogramacion():
class Prueba_met:
    pass
return Prueba_met

print(type(prueba_metaprogramacion))
print(type(prueba_metaprogramacion()))
```

En el primer print la terminal no devuelve: `<class 'function'>`, debido a que *prueba_metaprogramacion* pertenece a la clase función. Mientras que en el segundo de los print la terminal devuelve: `<class 'type'>`, debido a que el retorno de *prueba_metaprogramacion()* pertenece a la clase tipo.

La clase type es una metaclass. Esta clase se usa para crear otras clases en Python. El constructor de dicha clase se usa para crear clases que pueden crear instancias de dicha clase.

##El constructor de la clase type tiene esta estructura

```
type(cls, what, bases=None, dict=None)
El primer parametro es el nombre de la clase
El segundo parametro es una tuple de clases base
El tercer parametro es un diccionario de llaves y valores,
es decir, la implementación de la clase.
```

Cuando creamos una metaclass que hereda la clase tipo, esta tiene acceso al nombre de la clase, sus padres y todos sus atributos.

2.2.2. Clase Type

`__new__()` es otro ejemplo de metaprogramación. Crea nuevas instancias de clase y no está vinculado a una instancia de la clase. Este método se llama antes de llamar a `__init__()`. Podemos anular el `__new__()` de la superclase. El retorno del método `__new__()` es la instancia de la clase. Esto es útil cuando queremos modificar la creación de tipos de datos inmutables como las tuplas.

En este proyecto utilizamos este tipo de funcionalidad en la clase `decorator.py` la cual tiene una clase llamada `StructMeta` la cual hereda de `type` e implementa el funcionamiento de el metodo `__new__()`.

```
class StructMeta(type):
    def __new__(cls, name, bases, clsdict):
        clsobj = super().__new__(cls, name, bases, clsdict)
        sig = make_signature(clsobj._fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj
```

2.2.3. Usos de la Metaprogramación

1. Si queremos comprobar si una clase esta definida correctamente, podemos usar metaclasses.
2. Podemos usar metaclasses para generar errores durante la importación de los módulos de la clase.
3. Podemos usar la metaprogramación si queremos que nuestras clases tengan una convención especifica de métodos y atributos.
4. Las metaclasses se pueden utilizar para modificar el atributo de clase. [Malik, 2020]

2.2.4. Construir sistemas

Otra funcionalidad que nos ofrece la metaprogramación es la construcción de sistemas, ya que para la mayoría de los proyectos como este, existe un "proceso de construcción", es decir, se necesita una secuencia de operaciones para transformar todas las entradas que metemos en salidas. Por lo general este proceso suele ser bastante odioso ya que solemos tener una cantidad elevada de entradas, como por ejemplo, ejecuto un código para que funcione el programa, otro código para los benchmarks, otro código para los gráficos, es por ello que utilizamos la metaprogramación en este ámbito también. [./missing semester, 2020]

Básicamente la funcionalidad es la siguiente, se define una serie de dependencias, una serie de objetivos y reglas para ir de uno a otro. Le indicas al sistema de compilación que objetivo desear ejecutar en particular, y el trabajo de este es buscar y encontrar todas las dependencias de ese objetivo, después aplicar las reglas que este contiene y llegar al objetivo final.

En este proyecto he usado uno de los sistemas de compilación más conocidos en el ámbito de la programación **make**, debido a que el proyecto a sido totalmente desarrollado en linux y make esta instalado en cualquier maquina que utilice UNIX como sistema operativo. En este caso he usado dicho sistema para generar la memoria en latex y para la ejecución de los test unitarios y benchmarks, un ejemplo de ello son los siguientes objetivos marcados:

```
paper.pdf: paper.tex plot-data.png
pdflatex paper.tex
```

La directiva `paper.pdf` tiene una regla sobre cómo producir el lado izquierdo usando el lado derecho. O, dicho de otra manera, las cosas nombradas en el lado derecho son dependencias y el lado izquierdo es el objetivo. En `make`, la primera directiva también define el objetivo predeterminado. Si ejecuta `makes` sin argumentos, este es el objetivo que construirá.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

3 *Hipergrafo y problemas relacionados*

3.1. Conceptos básicos

En dicho proyecto también se van a tratar elementos como los hipergrafos y algoritmos con aplicación a estos, para ello primero he realizado un primer apartado con los conceptos básicos de que es o para que sirve un hipergrafo y más adelante me dedico a una explicación más técnica y profunda en el tema.

Un hipergrafo H es una familia de subconjuntos (aristas) de un conjunto finito de vértices. Un hipergrafo es simple si ninguno de sus bordes está contenido en ningún otro de sus bordes. Decimos que un hipergrafo está saturado si cada subconjunto del conjunto de vértices está contenido en un borde o contiene un borde de el hipergrafo. [Eiter y Gottlob, 1995]

Se toma como algoritmo principal *The algorithm of Berge*, al cual se le añadirán diferentes mejoras analizadas y comprobadas en el paper desarrollado por [Kavvadias y Stavropoulos, 2005] el cual implementa un algoritmo eficiente para la generación transversal de hipergrafos.

```
for i = 2, . . . , m do
  Find  $\text{Tr}(-H_{i1})$ 
  Compute  $\text{Tr}(H_i) = \text{Min}(\text{Tr}(-H_{i1}) \cup \{v\}, v \in E_i)$ 
end for
Return  $\text{Tr}(H_m)$ 
```

Listing 3.1: Original Berge Algorithm

```
for k = 0, . . . , m - 1 do
  Add  $E_{k+1}$ 
  Update the set of generalized nodes
  Express  $\text{Tr}(H^k_g)$  and  $E_{k+1}$  as sets of generalized nodes of level  $k + 1$ 
  Compute  $\text{Tr}(H^{k+1}_g) = \text{Min}(\text{Tr}(H^k_g) \cup \{v_X\} : v \in E^{k+1}_g)$ 
end for
Output  $\text{Tr}(H_m)$ 
```

Listing 3.2: Efficient Berge Algorithm

3.2. Problemas relacionados con SIMPLE-H-SAT en la teoría de bases de datos

Seguendo la lectura del paper [Eiter y Gottlob, 1995] en el apartado séptimo habla sobre los diferentes problemas relacionados con el SIMPLE-H-SAT en la teoría de bases de datos.

Inicialmente, habla sobre las bases de datos relaciones y hace una explicación sobre como esta formada una relación, básicamente una Una relación es una tabla de tuplas distintas de pares de componente que son valores de los dominios de los atributos. Por ejemplo, $U = \{A_0, A_1, \dots, A_n\}$ es un conjunto de atributos, cada uno de los cuales está asociado a un dominio D . Entonces $D(A_i)$ es el dominio de un atributo A_i . Entonces una relación sobre U es un subconjunto de $\prod_{i=1}^n D(A_i)$, y el los elementos de una relación se denominan tuplas.

El paper también habla sobre las relaciones funcionales, es decir, esas reglas de asociación que nos dan una información de unas variables frente a otras, y lo explica de la siguiente manera:

”Dados los conjuntos X, Y de atributos, la dependencia funcional (FD) $X \rightarrow Y$ se mantiene en la relación R si en cada tupla los valores de los atributos en Y están determinados únicamente por los valores de los atributos en X , para tuplas $t_1, t_2 \in R$; $t_1[X] = t_2[X]$ implica que $t_1[Y] = t_2[Y]$: Dado que posiblemente todas las tuplas de una relación tienen los mismos valores para algún atributo, el lado izquierdo X de un FD $X \rightarrow Y$ puede estar vacío. Si $Y \subseteq X$, entonces se llama dependencia funcional trivial.”

Y en general, explica como conseguir relaciones funcionales válidas, para ello se tienen que definir diferentes términos.

- **Conjuntos cerrados.-** Son los que no puedes ampliar sin perder soporte. Formalmente hablando, un conjunto $X \subseteq I$ es cerrado si $X = X$, donde X es la clausura de X y se calcula con la formula: $X = \{a \in I \mid s(X \cup \{a\}) = s(X)\}$
- **conjuntos frecuentes.-** $F_\tau = \{X \subseteq I \mid s(X) \geq \tau\}$
- **conjuntos frecuentes cerrados.-** $FC_\tau = \{X \in F_\tau \mid \forall Z \supset X, s(Z) < s(X)\}$
- **generadores minimales frecuentes.-** $\{X \in F_\tau \mid \forall Y \subset X, s(Y) > s(X)\}$

Con estos elementos definimos si una regla es redundante, sirve para no llenar la base de datos de una misma regla de asociación que se repite durante el tiempo, ocupa espacio innecesario y no ofrece ningún tipo de información valiosa.

Después de la definición de los anteriores conceptos, podemos decir que una regla de asociación es: $AR_{\tau, \gamma} = \{X \rightarrow Y \mid s(X \rightarrow Y) \geq \tau, c(X \rightarrow Y) \geq \gamma\}$, siendo τ y γ umbrales de soporte y confianza respectivamente.

También podemos definir la reglas representativas como:

$$RR_{\tau, \gamma} = \{r \in AR_{\tau, \gamma} \mid \nexists r' \in AR_{\tau, \gamma} \mid \gamma\{r\} \text{ tal que } r \in C(r')\} \text{ [Cristina Tirnauca, 2020]}$$

Con esta definición de regla representativa no redundante podemos conseguir una cantidad de reglas de asociación menor con un mismo valor de contenido, es decir, nos ahorraríamos mucho espacio en una base de datos gracias a esta definición de regla representativa.

3.3. Problemas relacionados con las transversales de los hipergrafos y su saturación

El segundo problema que nos encontramos después del SIMPLE-H-SAT es el reconocimiento de las transversales mínimas de un hipergrafo, ahora indicaré cual es la complejidad de dicho problema y luego demostrare la equivalencia que tiene con SIMPLE-H-SAT y el reconocimiento transversal en tiempo polinomial, basándome en la definiciones del paper anteriormente mencionado [Eiter y Gottlob, 1995].

Definición 1. Teniendo dos hipergrafos H, G con vértices V . Entonces $H \geq G$ si y solo si $\forall h \in H : \exists g \in G : H \supseteq G$. Esto a su vez define la siguiente relación, $H \leq G$ si y solo si $\forall h \in H : \exists g \in G : H \subseteq G$.

Decir que $H = G$ si y solo si $H \geq G$ y $G \geq H$, y $H \geq G$ no implica que $G \leq H$ y viceversa. Ahora indico una caracterización para la saturación de los hipergrafos:

Teorema 1. $H \neq \emptyset$ puede ser un hipergrafo. Entonces H esta saturado si y solo si $Tr(max(H)) \geq min(H)$.

Prueba: Asumimos que H esta saturado, y consideramos una arista T de $Tr(max(H))$. Tenemos $T \in Cov(H)$, pero T es una transversal de $max(H)$, $\forall E \in H$ se mantiene que $T \not\subseteq E$. Pero entonces $E' \subseteq T$ para algún $E' \in H$ lo que implica que $T \supseteq E''$ para algún $E'' \in min(H)$. Esto implica que $Tr(max(H)) \geq min(H)$. Asumimos ahora que H no esta saturado, $X \supseteq V, X \notin Cov(H)$.

Corolario 1. Un hipergrafo $H \neq \emptyset$ es saturado si y solo si $Tr(\overline{H}) \geq H$.

Prueba: Es facilmente demostrable que $\overline{max(H)} = min(\overline{H})$, sacado directamente del Teorema 1

Corolario 2. Para demostrar si $Tr(H) \geq H$ se sostiene para un hipergrafo H es co-NP-completo, incluso si H se auto completa y contiene solo aristas de tamaño 3 y $n-3$, donde $n = |V(H)|$

3.4. Axiomas de Armstrong

Los Axiomas de Armstrong son en su esencia reglas de inferencia mencionadas anteriormente. Estas reglas permiten deducir todas las dependencias funcionales que tienen lugar entre un conjunto dado de atributos $A = \{A_1, A_2, \dots, A_n\}$, como consecuencia de las dependencias dadas, es decir, se deducen todas las dependencias que se asumen ciertas a partir del conocimiento del problema.

3.4.1. Ejemplo de axiomas de Armstrong

Tenemos una relación R con tres atributos (A, B, C) , ahora, después de realizar la función de dependencias, nos devuelve las dos siguientes reglas $F = \{A \rightarrow B, B \rightarrow C\}$, con estas dos reglas podemos inferir el conjunto completo de funciones de dependencias que serian: $\{A \rightarrow C, A \rightarrow A, B \rightarrow B, C \rightarrow C, \}$.

Ahora tenemos un set F con las dos reglas de dependencias y un superset F^+ el cual es un set de funciones lógicamente inferidas de F . A continuación, muestro una tabla de la relación entre el numero de atributos combinados y las posibles keys que podemos elegir.

	# de atributos combinados		
Posibles keys	1	2	3
	A	AB	ABC
	B	AC	
	C	BC	
Total	3	3	1

En este caso es sencillo debido a que únicamente tenemos 3 atributos posibles, pero en el caso que por ejemplo tengamos un numero elevado de ellos $A = 100$ se puede utilizar una formula matemática que calcula el numero de posibilidades $\frac{n!}{(n-r)!(r)!}$ donde r es número total de atributos y n cuantos atributos por key tenemos.

Una vez tenemos esto necesitamos un mecanismo que nos permita conocer si las reglas de dependencia están dentro o fuera del F^+ , es entonces cuando utilizamos los axiomas de Armstrong.

Para ello primero definiré que es un axioma, según la tercera definición del diccionario, un axioma es una proposición que se asume sin prueba, con el fin de estudiar las consecuencias que se derivan de ella. Los axiomas de armstrong tienen tres acciones reflexividad, aumentatividad y transitividad, que son en términos matemáticos definidas de la siguiente manera:

1. **Reflexividad** $\forall X, X \rightarrow X \text{ ex. } (ABC \rightarrow AB)$
2. **Aumentatividad** $\{X \rightarrow Y, Z \supseteq X\} \Rightarrow Z \rightarrow Y \text{ ex. } (AC \rightarrow CB)$
3. **Transitividad** $\{X \rightarrow Z, Z \rightarrow Y\} \Rightarrow X \rightarrow Y \text{ ex. } (A \rightarrow B, B \rightarrow C) \Rightarrow (A \rightarrow C)$

Si tenemos en F^+ un set de todas las funciones con dependencias las cuales hemos inferido de F, si aplicamos las acciones de armstrong a F no vamos a encontrar ningún set de funciones con dependencias fuera de F solo podremos encontrar estas funciones dentro de F.

otro aspecto de los axiomas de armstrong es que es completo esto quiere decir que si aplicamos acciones de armstrong a F vamos a encontrar todas las funciones con dependencias existentes en dicho set.

4 *KD Trees*

4.1. Conceptos básicos

Un KD Tree (también conocido como K-Dimensional Tree) es un árbol de búsqueda binario donde los datos en cada nodo son un punto k-dimensional en el espacio. Básicamente, es una estructura de datos que utiliza una partición en el espacio para organizar puntos en un espacio K-Dimensional.

1. Cada nodo tiene como clave multidimensional un vector de tamaño k que contiene los valores de las k claves unidimensionales y tiene asociado un discriminante con valor entero entre 1 y k .
2. Para cada nodo con clave multidimensional x y discriminante j se cumple que cualquier nodo del subárbol izquierdo con clave multidimensional y cumple que su n -ésima componente es menor que la n -ésima componente de x , es decir ($y_n < x_n$). Para el subárbol derecho se cumple que la n -ésima componente de la clave multidimensional z de cualquier nodo de este subárbol es mayor que la n -ésima componente de la clave x , es decir ($z_n > x_n$).
3. La raíz del kd-tree tiene profundidad 0 y discriminante 1. Cualquier nodo a una profundidad p tiene discriminante de valor: $(p \bmod k) + 1$. [Martí Fuentesauco, 2010]

Otra de las aplicaciones es la de representar un conjunto de puntos y la consulta más frecuente sobre este conjunto es la de localizar el punto o puntos más cercanos a uno dado. Esta consulta se conoce con el nombre de búsqueda del vecino más cercano.

4.2. Implementación en Reglas de Asociación

Los kd-trees son frecuentemente utilizados en bases de datos para satisfacer consultas que incluyan valores de varios campos. Esto se puede extrapolar a las reglas de asociación, ya que se podrían realizar consultas como se ha explicado en el ejemplo introductorio de reglas de asociación.

Una consulta podría ser por ejemplo, saber todos los registros de personas que compren pan, leche y huevos, y además tengan entre 25 y 45 años, y que vivan en España, todo esto se podría realizar en una única consulta utilizando los *KD trees* como algoritmo de búsqueda ya que permite realizar diferentes consultas simultaneas, en lugar del algoritmo *apriori* actualmente utilizado, en el que al añadir múltiples consultas, su coste computacional hace que no tenga sentido dicho calculo, debido a su lento proceso.

4.3. Implementación KD-Trees en Slatt

No he incluido la implementación de KD-Trees en Slatt debido a que la forma en la que he intentado implementarlo hacia más lento el código y no conseguía una mejora real en el programa, es por esto que decidí no incluirla y usar otros algoritmos y clases para la mejora, como he mencionado a lo largo de la memoria.

4.4. Ejemplo gráfico KD Tree

Ejemplo gráfico de un KD-tree para el siguiente ItemSet:
 $(3,6), (17,15), (13,15), (6,12), (9,1), (2,7), (10,19)$

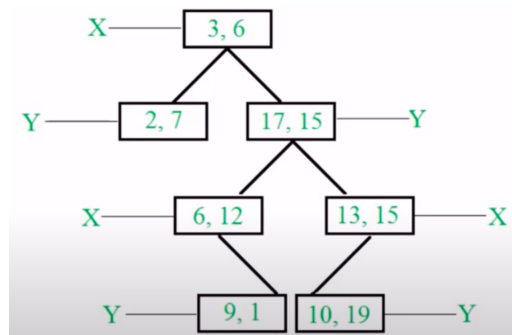


Ilustración 6: Árbol del ItemSet

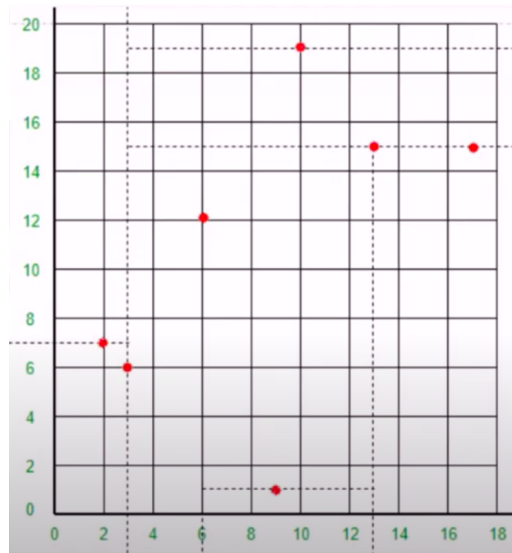


Ilustración 7: Gráfica del ItemSet

Desde un punto de vista geométrico como se ve en la Ilustración 6, cada nodo del kd tree hace una partición del plano en dos “subplanos”. En la Ilustración 7 todos los puntos en el “subplano” de la izquierda corresponden al subárbol izquierdo del nodo raíz, y los que quedan a la derecha son los puntos del subárbol derecho.

5 *Desarrollo Software*

En este apartado del proyecto se muestran que cambios han sido necesarios en el código para la correcta ejecución de Slatt pasando todo su código de python2 a python3, los test necesarios para su correcto funcionamiento y además se explica que mejoras se han llevado a cabo tanto de rendimiento, como en la reducción del número de líneas de código.

5.1. Ejecución y funcionamiento de Slatt

Slatt es una herramienta matemática que se encarga de mediante una entrada de datos en este caso, por ejemplo, un dataset sobre los problemas de visión de ciertas personas, el cual dispone de 4 atributos, la edad, la prescripción (myope, hypermetrope, etc), si tiene o no astigmatismo y la cantidad de lágrimas que genera.

Se introduce dicho dataset en el programa principal y la herramienta devuelve diferentes datos como son la confianza, el soporte y con esto generar las diferentes reglas de asociación que existen y cuales de estas son reglas de inferencia.

5.2. Test unitarios

En esta sección se especifica como se realizan los test unitarios, para comprobar si las dos versiones del código mantienen la misma funcionalidad basándose en la salida de estos test.

```
##MAKEFILE##
##AUTHOR: ALEJANDRO MARTINEZ
##DATE: 26/03/2021

#Nombre_fich test python2 to python3
Nombre_fich :

    python3 Python3/Nombre_fich.py > /TFG/tests/nombre_fich3.txt

    python2 Python2/Nombre_fich.py > /TFG/tests/nombre_fich2.txt

    diff /TFG/tests/nombre_fich3.txt TFG/tests/nombre_fich3.txt >
    /TFG/tests/diff_nombre_fich.txt
```

Listing 5.1: Makefile

Utilizando este **makefile** se realiza una ejecución simultanea de ambas versiones, guardando dicha salida en un fichero.txt y a continuación, se hace un **diff** entre los dos archivos, en el caso en el que el **diff** devuelva un archivo vacío, sabremos que las dos versiones tienen la misma funcionalidad.

5.3. Mejoras del código

En esta sección se va a discutir cuales han sido los cambios y mejoras en cada una de las clases que se han modificado en el código:

5.3.1. Diferencias entre python2 y python3

En la actualidad, existen dos tipos de versiones de python, las cuales son 2.x en la que está inicialmente escrita Slatt y 3.x en la cual se ha estado desarrollado este proyecto.

Las principales diferencias que he encontrado en el proyecto han sido las siguientes [Miscelánea, 2020]:

1. Sentencia print

Esta posiblemente sea la diferencia que más veces ha aparecido en el proyecto, ya que en prácticamente todas y cada una de las clases aparece algún print en el main o en algún método de la propia clase. Se soluciona de una manera muy sencilla, añadiendo el el argumento de la sentencia print dos paréntesis. (`print x -> print(x)`)

2. Comparación de tipos

En **python2** se permiten las comparaciones entre objetos de distintos tipos, mientras que en **python3** nos lanza una excepción del tipo *TypeError*, esto a sido necesario tenerlo en cuenta en el código porque había bastantes comparaciones de tipos distintos a lo largo del código en clase como `brulattice.py` o `closminer.py`.

3. División de números enteros

En **python2** la división entre números enteros da un numero entero y si quieres conseguir un resultado con decimales necesitas que uno de los dos números de la operación tengan un decimal, en este caso en **python3** se hace mediante un truncado de la división usando el `//` en lugar de `/`.

4. Iterar un diccionario

La ultima diferencia que se ha detectado en código a la hora de hacer la reescritura a **python3** a sido la iteración de diccionarios ya que en **python2** se pueden iterar los elementos clave-valor de un diccionario con el método `iteritems()` o `items()`, mientras que en python3 únicamente se puede usar el segundo de estos dos métodos, ya que con el primero de ellos se produce una excepción de tipo *AttributeError*. A su vez, los métodos `iterkeys()` e `itervalues()` para iterar las claves y los valores de un diccionario respectivamente no existen en Python 3. En su lugar tenemos que usar los métodos `keys()` y `values()`. Estos métodos tanto `keys()`, `values()` como `items()` se han visto afectados en clases como `brulattice.py` o `cboost.py`

5.3.2. Cambios realizados en las clases

■ `brulattice.py`

En la clase `brulattice.py` no se han realizado mejoras ni cambios en el código salvo lo necesario para la ejecución en python3.

■ `cboost.py`

En la clase `cboost.py` inicialmente he modificado el método `allsubsets` para que sea mas corto y eficiente utilizando la librería de combinations procedente de `itertools` la cual me permite realizar fors anidados de una manera más eficiente y con un número de líneas reducido.

```

def allsubsets(givenset):
    aset = givenset.copy()
    for e in aset:
        aset.remove(e)
        p = allsubsets(aset)
        q = []
        for st in p:
            s = st.copy()
            s.add(e)
            q.append(s)
        return p+q
    return [ set([]) ]

#### Se sustituye por: ####

def allsubsets(givenset):
    s = len(givenset)
    q=[]
    for i in range(s):
        for e in combinations(givenset,i):
            q.append(e)
    return q

```

Dentro del método *filt()* también se han cambiado alguno de los fors anidados que tenía debido a que aumentaban la carga de trabajo innecesariamente, para ello en vez de utilizar el método *keys()* a la hora de iterar, se ha decidido utilizar dos parámetros en el for e iterarlo sobre el método *items()*, con lo que mejoramos tanto la legibilidad del código como el rendimiento ya que nos eliminamos un for anidado por cada vez que lo sustituimos, el único inconveniente encontrado en este paso fue que al usar este tipo de for no puede haber dependencias entre ambos parámetros ya que entonces no funciona correctamente el programa. (# líneas antes = 60, # líneas después = 52)

```

for cn2 in rrseconf.keys():
    for an2 in rrseconf[cn2]:

#### Se sustituye por: ####

for cn2,values in rrseconf.items():
    for an2 in values:

```

■ clattice.py

En la clase *clattice.py* utilice la variable *__processing* para eliminar el *__init__* el cual era muy largo y costoso definiéndolo en un método fuera de la clase el cual se ejecuta una vez y no es necesario volver a ejecutar con lo que a la larga nos ahorramos código en tiempo de ejecución, por lo demás no he incluido ninguna mejora más respecto a python2.

■ closminer.py

En la clase *closminer.py* se ha realizado un cambio similar al de *clattice.py* introduciendo una método *pos__init__* que funciona como método auxiliar para la creación de una instancia de *closminer*, además de introducir la herencia de la clase *structure* de *decorator* para poder eliminar el método *__init__* y así acelerar el proceso como hemos hecho en las demás clases del programa.

■ **corr.py**

La clase `corr.py` es una clase con pocas líneas de código y difícil de optimizar más ya que no contiene método muy densos, por que únicamente he realiza la herencia de `Structure` y he eliminado el constructor ineficiente que teníamos en `python2`.

■ **decorator.py**

La clase `decorator.py` es una nueva clase implementada para **python3** la cual nos permite utilizar diferentes funcionalidades de la metaprogramación mencionadas anteriormente, como es la creación de constructores mediante una simple línea de código gracias a las clase `Structure` y la variable `__fields` o incluso introducir partes del código (funciones, métodos) con otra variable llamada `__processing`.

También se introducen un método que nos permite saber quienes son las clases padre de una clase u objeto, gracias al modulo *Signature* de la clase `inspect.py`.

■ **hypergraph.py**

En la clase `hypergraph` se realiza una herencia de la clase `Structure`, para eliminar su `__init__` sustituyéndolo por la variable `__fields` y con ello reducir el número de líneas de código. También se ha diseñado un patrón decorator el cual sustituye la mayoría de las funciones que se encuentran en dicha clase: [**addel,added,addhg,remel,remed**], este cambio se llevó a cabo debido a que se vio un claro patrón de repetición en todas las funciones de la clase, el patrón es el siguiente:

```
def decorator(self,a1,a2,a3,a4,a5):
    if a1:
        a1()
    for e in a2:
        a3(e)
    if a4:
        a4(a2)
    if a5:
        a5()
```

Con este patrón se consigue resumir todas y cada una de las funciones de la clase en una única línea de código, para ello tuve que usar funcionalidades como *lambda* la cual permite generar una función dentro del patrón decorator. (# líneas antes = 142, # líneas después = 134)

■ **rerulattice.py**

En la clase `rerulattice.py` no se han realizado mejoras ni cambios en el código salvo lo necesario para la ejecución en `python3`.

■ **slanode.py**

En esta clase `slanode.py` se ha eliminado la clase *auxitset* con todos sus métodos, introduciéndola dentro de la propia clase *slanode*, en concreto se han introducido los dos métodos *str2node* y *set2node* en la clase *slanode* y también se ha creado el `__init__` de *auxitset* y *slanode* con un solo método llamado `__new__` que introduce como parámetros los atributos de la clase y los inicializa sin ser necesario usar una clase auxiliar.

■ **slarule.py**

En esta otra clase se añade también al herencia de la clase `decorator` para poder minimizar la creación del constructor `__init__`, se eliminan todas la ocurrencias de igualdad con 0, ya que en este caso siempre es verdadero y por lo tanto, las reglas `if x = 0` son inútiles un ejemplo de ello seria:

```

1
2
3
4
5
6     if self.an.supp == 0:
7         self.confval = float(self.cn.supp)/self.an.supp
8
9     ##### Se sustituye por: #####
10
11     if not self.an.supp:
12         self.confval = float(self.cn.supp)/self.an.supp
13
14

```

Con este simple cambio se genera un pequeña mejora en el rendimiento del programa y a su vez facilita la compresión del mismo.

■ slattice.py

En la clase slattice.py no se han realizado mejoras ni cambios en el código salvo lo necesario para la ejecución en python3.

■ verbosity.py

En esta clase se añade una herencia de la clase decorator.py previamente analizada, verbosity utiliza una de las clases de decorator llamada Structure la cual permite eliminar el `__init__` de la clase añadiendo una única variable llamada `_fields`, en la cual se definen cada uno de los parámetros definidos en el `__init__`, tanto argumentos como los propios atributos de la clase.

En código se ve así:

```

30
31     def __init__(self,verb=True):
32         """
33         verbosity;
34         lim and count for the progress-reporting ticks;
35         """
36
37         self.verb = verb
38         self.lim = 0
39
40         self.count = 0
41
42
43     ##### Se sustituye por: #####
44
45     _fields = [('verb', True), ('lim', 0), ('count', 0)]
46

```

Con este cambio conseguimos una mejora en cuanto al número de líneas y simplicidad de código enorme, ya que realizar el constructor de la clase antes nos costaba 7 líneas donde ahora lo podemos conseguir en una. Por lo demás, los demás cambios han sido simplemente en la sintaxis ya que python2 y python3 tienen diferencias en el lexico. (# líneas antes = 47, # líneas después = 35)

5.4. Discusión y resultados de la mejora

DUDA SI METER ESTE APARTADO

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

6 Conclusiones

El propósito de este trabajo de fin de grado es su inicio fue la reescritura de Slatt (programa matemático que calcula reglas de asociación) del lenguaje de programación python2 a su nueva versión de python3, además de esto se pretendía mejorar dicho programa añadiendo diversas mejoras con algoritmos basados en hipergrafos y cálculos de clausuras y retículos.

Para esto se realizó inicialmente una búsqueda de información sobre dichas estructuras de datos, tanto hipergrafos como kd-trees para ver su funcionamiento y llegar a la conclusión que son una buena herramienta para nuestro problema ya que nos permiten guardar las diferentes variables de las reglas de una forma más eficiente que como se estaban guardando ahora en simple diccionarios o listas de elementos, es por ello, que se decidió implementar parte del programa usando estos dos tipos de estructuras.

Comentar que era difícil la mejora en el propio código de python2, ya que no había mucha mejora posible en el propio código que venía de serie, es por esto que se buscó realizar clases auxiliares que proporcionasen algo de mejora al código, siempre buscando una mejora tanto de rendimiento como en líneas de código, este fue el objetivo principal a la hora de la reescritura.

Mi conclusión personal sobre el trabajo realizado es que las reglas de asociación son una herramienta muy útil en el ámbito de las ciencias de la computación, como por el ejemplo en el *Big Data*, y que todavía queda mucho desarrollo de estas por delante, no tanto en lo que ya conocemos como objeto matemático sino en la mejora de su rendimiento añadiendo nuevos algoritmos que nos permitan tener una producción de reglas de asociación más rápida y efectiva, ya que gracias a ellas se pueden obtener información muy valiosa sobre un dataset muy extenso que a priori no tenga mucho valor y con esto generar una ventaja ya sea en el ámbito de los negocios como una simple mejora en la industria de la investigación científica.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Referencias

F. Berzal. Reglas de asociación, 2016.

<http://elvex.ugr.es/idbis/dm/slides/2%20Association.pdf>

D. Bleazley. Python 3 metaprogramming. 2013.

https://www.youtube.com/watch?v=sPiWg5jSoZI&ab_channel=NextDayVideo

U. d. C. Cristina Tirnauca, Dept. Matesco. Reglas de asociación. 2020.

T. Eiter y G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6), 1995.

<https://epubs.siam.org/doi/10.1137/S0097539793250299>

D. J. Kavvadias y E. C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

<http://jgaa.info/accepted/2005/KavvadiasStavropoulos2005.9.2.pdf>

F. Malik. Advanced python: Metaprogramming. 2020.

<https://medium.com/fintechexplained/advanced-python-metaprogramming-980da1be0c7d>

S. Martí Fuentesauco. Copy-based kd-trees y quad-trees. 2010.

https://upcommons.upc.edu/bitstream/handle/2099.1/9078/PFC_Copybased.pdf?sequence=1&isAllowed=y

Miscelánea. Diferencias entre python 2 y 3: ¿qué versión debería aprender? 2020.

<https://www.programaenpython.com/miscelanea/diferencias-entre-python-2-y-3>

./missing semester. metaprogramming ./missing-semester. 2020.

<https://missing.csail.mit.edu/2020/metaprogramming/>

E. Morales y J. Gonzales. Reglas de asociación, 2013.

<http://ccc.inaoep.mx/~emorales/Cursos/NvoAprend/Acetatos/reglasAsociacion.pdf>

J. F. Silva Galiana. Metaprogramación, ¿qué es y para qué sirve? 2018.

<https://riunet.upv.es/handle/10251/103846>