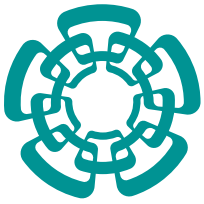

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco
Departamento de Computación

Inferencia de dependencias funcionales mediante funciones de similitud en minería de datos

Tesis que presenta
Zelzin Marcela Márquez Navarrete
para obtener el grado de
Maestra en Ciencias en Computación

Director de la Tesis
Dr. Guillermo Benito Morales Luna

Ciudad de México Noviembre 2019

Resumen

Dado un conjunto de atributos en una bases de datos relacionales, obtuvimos de forma correcta todas las dependencias funcionales mínimas y no triviales de dicha relación mediante una modificación al algoritmo TANE para hacer uso de funciones de similitud. Con el fin de hallar nuevas relaciones para el descubrimiento automático de conocimiento en distintas bases de datos, realizamos experimentos sobre ellas, variando los umbrales de las funciones de similitud implementadas.

La metodología implementada fue capaz de hallar nuevas relaciones de dependencia en las bases de datos que, con un algoritmo de búsqueda de dependencias funcionales exactas no puede ser obtenido debido a las restricciones de este tipo de dependencia. Adicionalmente esta metodología es útil para el proceso de limpieza y diseño de la base de datos puesto que utilizando funciones de similitud es posible determinar cuando existen tuplas que no cumplen con alguna dependencia funcional que deber ser válida en la relación. En un caso como este nuestra metodología tiene la ventaja de que no requiere modificar los datos antes de determinar si la dependencia funcional será válida.

Abstract

Given a set of attributes in a relational database, we correctly found all minimal non-trivial functional dependencies of a database using a modified TANE algorithm that employs similarity functions. In order to find new relations to perform automated knowledge discovery, we perform several experiments with different thresholds for each similarity function implemented.

The proposed methodology is able to find relations that would be impossible to find using traditional algorithms for the inference of exact functional dependencies. Furthermore, using our methodology it is possible to determine if some tuples prevent a functional dependency from being valid. If such dependency must be valid then we know that there are anomalies in the data that must be corrected. Such results are useful for data cleaning, or for the design of relational databases. One advantage of our methodology is that data does not need to be modified to verify if a functional dependency holds in the database.

Agradecimientos

Agradezco al CONACyT por proporcionarme dos años de beca de posgrado. Al Dr. Guillermo Benito Morales Luna por permitirme colaborar en el proyecto del laboratorio digital de violencia y paz. Al CINVESTAV por darme la oportunidad de adquirir conocimiento en este programa de posgrado. A Mario por su orientación y apoyo. A Ulises quien me obsequió la computadora que me permitió realizar esta tesis.

Zelzin Marcela Márquez Navarrete
CINVESTAV, Noviembre 2019

Índice general

Resumen I

Abstract III

Agradecimientos V

Índice general VII

Índice de figuras XI

Índice de tablas XIII

1 Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Estudio del caso: Laboratorio Digital de Violencia y Paz . . . 3
  - 1.2.1. Modelos sociodemográficos e indicadores urbanos . . . 4
  - 1.2.2. Los municipios de Nezahualcóyotl y Ecatepec de Morelos . . . 4
- 1.3. Problema . . . 5
- 1.4. Contribuciones . . . 6
  - 1.4.1. Publicaciones . . . 6
- 1.5. Organización de la tesis . . . 6

I Definiciones y descripción del problema 7

2 Bases de datos relacionales 9
- 2.1. Introducción . . . 9
- 2.2. El modelo relacional de bases de datos . . . 9
- 2.3. Atributos y relaciones . . . 9
- 2.4. Dependencias funcionales . . . 10
- 2.5. Esquema relacional . . . 11
  - 2.5.1. Proyección y junta . . . 11
- 2.6. Forma normal de Boyce-Codd . . . 12
  - 2.6.1. Limpieza de datos . . . 12
- 2.7. Dependencias basadas en DFs . . . 13
  - 2.7.1. Dependencias multivaluadas . . . 14
  - 2.7.2. Dependencias funcionales aproximadas . . . 14
  - 2.7.3. Dependencias funcionales condicionales . . . 15
- 2.8. Dependencias funcionales basadas en funciones de similitud . . . 15
  - 2.8.1. Funciones de similitud . . . 16

2.8.2. Dependencias funcionales basadas en funciones de distancia 17
2.8.3. Dependencias de acoplamiento 17
2.8.4. Dependencias funcionales métricas 19
2.8.5. Dependencias de diferencias 20
2.8.6. Dependencias de vecindad 21
2.8.7. Dependencias secuenciales 21
2.9. Descubrimiento de conocimiento en bases de datos 22
2.9.1. Minería de datos 22

3 Inferencia de dependencias funcionales 23
3.1. Introducción 23
3.2. Complejidad de la inferencia de dependencias funcionales 24
3.3. Algoritmos para la inferencia de dependencias funcionales 24
3.3.1. Particiones 24
3.3.2. Algoritmos de retícula 25
3.3.3. Algoritmos basados en conjuntos 25
3.4. TANE 27
3.4.1. Refinamiento de las particiones 27
3.4.2. Poda a través de conjuntos candidatos 27
3.4.3. Poda a través de llaves 29
3.4.4. Algoritmos 29

II Metodología y experimentación 33

4 Metodología 35
4.1. Introducción 35
4.2. Limpieza de la base de datos 35
4.2.1. Datos faltantes 36
4.2.2. Errores léxicos 36
4.2.3. Errores de formato 36
4.2.4. Manejo de información redundante 37
4.2.5. Concatenación de la latitud y longitud 38
4.3. Cálculo de clases de equivalencia mediante funciones de similitud 38
4.4. Algoritmo para el cálculo de clases de equivalencia usando funciones de similitud 40

5 Experimentos y resultados 43
5.1. Introducción 43
5.2. Bases de datos de prueba 43
5.3. Pruebas de correctitud 44
5.4. Descubrimiento automático de conocimiento con funciones de similitud . . . 45
5.5. Detección de anomalías con funciones de similitud 47
5.6. Análisis de la base de datos CRÍMENES 49

III Conclusiones **53**

6 Conclusiones y trabajo futuro **55**
6.1. Conclusiones 55
6.2. Trabajo futuro 56

Apéndices **58**

A Código fuente **59**
A.1. Limpieza de datos 59
A.1.1. Datos faltantes 59
A.1.2. Errores léxicos 59
A.1.3. Errores de formato 59
A.1.4. Concatenación de la latitud y longitud 60
A.2. Cálculo de clases de equivalencia usando funciones de similitud 60
A.3. Funciones de similitud 64
A.3.1. Distancia de Minkowski 64
A.3.2. Distancia de Levenshtein 64
A.3.3. Distancia entre dos fechas en días 65
A.3.4. Distancia entre dos horas en minutos 66

B Dependencias funcionales **67**

C Manual de usuario **73**

Bibliografía **77**

Índice de figuras

2.1. Cada punto en la bola de diámetro d representa una tupla en el lado derecho de una dependencia funcional métrica. 20

3.1. Retícula de elementos del conjunto potencia de los atributos {A, B, C, D}, que debido a que se eliminó B los subsecuentes conjuntos que lo contienen no son revisados por el algoritmo. 26

4.1. Clases de equivalencia. 41

5.1. Información de la base de datos IRIS después de ser agrupada. 45

Índice de tablas

1.1. Municipios con la mayor población en el Estado de México. 5
1.2. Incidencia delictiva entre enero y abril de 2016 en los municipios de Ecatepec de Morelos y Nezahuacóyotl [1] 5

2.1. Una tabla es una relación sobre un conjunto de atributos. 10
2.2. Un subconjunto del conjunto de datos iris. 18

3.1. Una relación para mostrar un ejemplo de particiones respecto a sus atributos. . . . 25
3.2. Particiones de los atributos en la tabla 3.3.1 25

4.1. Errores de formato. 37
4.2. Errores de formato corregidos. 37
4.3. Datos de prueba y clases de equivalencia. 39
4.4. Datos de prueba y clases de equivalencia. 39

5.1. Resultados de las experimentos de inferencia de FDs exactas. 44
5.2. Resultados para los atributos LP y LS usando la distancia de Minkowski. 46
5.3. DFs encontradas usando el algoritmo propuesto para calcular clases de equivalencia. 46
5.4. Resultados usando la distancia de Levenshtein y $\delta = 1$ en el atributo `phone`. . . . 48
5.5. DFs obtenidas usando la distancia de Levenshtein sobre `addr` y `city`. 49
5.6. Ejemplos de tuplas para los atributos COORDENADA y CUADRANTE. 50
5.7. Ejemplos de tuplas para los atributos CUADRANTE y SECTOR. 51
5.8. Tuplas duplicadas para el par (HORA_FIN, COORDENADA). 52

Capítulo 1

Introducción

1.1. Motivación

La inferencia de dependencias funcionales es una herramienta utilizada para el análisis de bases de datos relacionales. Debido a su amplia gama de aplicaciones, ha sido de gran interés para áreas como la minería de datos, el aprendizaje automatizado, la inteligencia artificial, entre otras [2]. Una dependencia funcional establece una relación entre dos atributos de la base de datos, de manera tal, que a cada valor en el dominio del primer atributo corresponda un valor único en el dominio del segundo.

A través de la inferencia de dependencias funcionales es posible establecer relaciones entre los datos que lleven al descubrimiento automatizado de conocimiento. Por ejemplo, consideremos una base de datos donde cada atributo representa alguna característica de la taxonomía de varias especies de plantas, e. g, tamaño del tallo, forma de la hoja, tamaño de la flor, etc. En ella, es de esperar que existan dependencias funcionales que relacionan para una misma especie de planta el tamaño de la flor con la forma de la hoja.

En otros casos es posible verificar la consistencia de los datos a través de la inferencia de dependencias funcionales. Por ejemplo, si en una base de datos se tiene la certeza de que un par de atributos están relacionados y dicha relación no es encontrada cuando se calculan las dependencias funcionales, entonces es posible que los datos presenten alguna anomalía, *e. g.*, datos faltantes en algún atributo que provoque no encontrar la relación [3].

Además de los ejemplos anteriores, las dependencias funcionales han sido aplicadas con éxito al diseño de bases de datos relacionales. En particular, es posible eliminar la redundancia de datos a través de la inferencia de dependencias funcionales mediante un proceso conocido como normalización [4]. Debido a lo anterior, ha existido un gran interés en el desarrollo de algoritmos que sean capaces de determinar todas las dependencias funcionales de una base de datos de manera eficiente.

En algunos casos la definición usual de dependencia funcional no es lo suficientemente robusta como para encontrar dichas dependencias en los atributos incluso si estas son obvias o esperadas. Lo anterior ocurre debido errores durante la recolección de los datos, inconsistencias en el formato usado para representar los valores de un atributo, o a que los datos provengan de distintas fuentes.

Es importante notar que a menudo la misma naturaleza de los datos hace que sea imposible establecer una dependencia funcional entre dos atributos. Por ejemplo, en una base de datos de empleados sería fácil establecer una relación entre el puesto que tiene dicho empleado y su sueldo. Esto debido a que el dominio para ambos atributos es discreto con valores bien definidos. En contraste, consideremos una base de datos que contiene información del clima de diversas ciudades, no obstante que es razonable esperar que exista una relación entre una ciudad y la temperatura esperada en cierta época del año, en realidad, las pequeñas diferencias que existirán en los registros de temperatura en una ciudad, dificultarán la inferencia de una dependencia funcional. Para resolver el problema anterior se han propuesto modificaciones a la definición de una dependencia funcional que relajan los requerimientos necesarios para establecer una relación entre dos atributos.

Uno de los primeros trabajos que aborda las limitaciones de una relación de equivalencia para inferir relaciones en una base de datos aparece en [5], en él, Buckles y Petry analizan el problema de definir relaciones entre los atributos de una base de datos cuando existen datos faltantes en la misma. Al igual que sucede con los problemas antes descritos, los datos faltantes también impiden inferir las dependencias funcionales de una base de datos. En ese sentido, los autores proponen utilizar relaciones de similitud entre los atributos de la base de datos en lugar de una relación de equivalencia. Es importante notar que en este trabajo, una relación de similitud debe entenderse como una correspondencia reflexiva y simétrica entre cualquier par de elementos de un conjunto de atributos y el intervalo $[0, 1]$. Los autores muestran que estructurar una base de datos en torno a relaciones de similitud apropiadas para cada atributo permite realizar consultas que determinan que tan bien un conjunto de atributos cumple con cierta propiedad.

Otra alternativa está dada por las llamadas dependencias funcionales aproximadas [6]. En este tipo de dependencias, se utiliza una medida de error que determina que tan lejos está de ser válida una posible dependencia funcional. Por lo tanto, una dependencia con un error bajo es indicativa de una posible relación entre los atributos correspondientes. El criterio más común para definir la medida de error es el número de tuplas que impiden que una posible dependencia sea válida. Tradicionalmente las dependencias aproximadas son utilizadas para verificar la correctitud y consistencias de los datos, así como herramienta de apoyo para el proceso de limpieza de datos. En general, para calcular dependencias funcionales aproximadas solo se modifica alguno de los algoritmos que calculan dependencias funcionales exactas [7]. No obstante existen implementaciones dedicadas exclusivamente a obtener este tipo de dependencias [8]. El principal problema de estas dependencias radica en definir que tan grande puede ser la media de error antes establecer que una dependencia ya no es válida. Una posible solución es dada en [9], donde Mandros, Boley y Vreeken utilizan estimadores de error basados en la información mutua.

De especial interés para este trabajo de tesis son las dependencias funcionales en las cuales se utiliza una función de similitud en lugar de una relación de equivalencia para establecer cuando un atributo depende de un conjunto de ellos en la base de datos. Un primer ejemplo está dado por las dependencias funcionales acopladas [10] [11], mismas que especifican una restricción que debe ser cumplida por dos conjuntos de atributos para poder establecer una dependencia funcional entre ellos. Por ejemplo, para que un atributo *ciudad* dependa de un atributo *calle*, es necesario que cualesquiera dos tuplas de *calle* tengan al menos una similitud α, y que las correspondientes tuplas de *ciudad* tengan al menos una similitud β; $\alpha, \beta \in [0, 1]$. Para establecer la similitud entre cualesquiera dos tuplas de un atributo dado, es necesario utilizar una función de similitud apropiada, *e. g.*, la distancia de Levenshtein y la distancia del coseno para texto o la distancia de Minkowski para atributos numéricos, en [12] es posible consultar una reseña sobre diversas funciones de similitud y sus principales características.

En bases de datos relacionales no siempre se busca establecer relaciones a partir de una relación de equivalencia, *i. e.*, hay relaciones en los datos que son válidas porque los valores de un atributo son diferentes. Este tipo de relaciones no pueden ser inferidas a través de dependencias funcionales exactas.

En [13] se utiliza un tipo de dependencia funcional llamado de diferencias, que parte de las diferencias en los datos para establecer una relación. Una dependencia funcional de diferencias especifica una restricción, a través de una función de similitud, que debe ser cumplida para que una dependencia funcional sea válida. Por ejemplo, en una base de datos de transacciones bancarias con atributos `Tarjeta`, `Ubicación` y `Hora`, si dos tuplas tienen la misma tarjeta (la distancia es cero) y la diferencia entre las ubicaciones de ambas tuplas es mayor a 60, entonces la hora a la

que se llevaron a cabo ambas transacciones debe tener una diferencia mayor a 20 minutos. Al igual que en las dependencia funcionales acopladas, para determinar la diferencia entre dos tuplas de algún conjunto de atributos es necesario utilizar una función de similitud apropiada.

Otro ejemplo de dependencias funcionales definidas a partir de funciones de similitud aparece en [14]. En este trabajo, Koudas, Saha, Srivastava y Venkatasubramanian introducen el concepto de dependencia funcional métrica. A diferencia de las dependencias funcionales de diferencias y las dependencias funcionales acopladas, una dependencia funcional métrica estable una relación entre dos conjuntos de atributos, de manera tal, que para cada conjunto de tuplas con el mismo valor en el primer conjunto corresponden tuplas que comparten valores similares en el segundo conjunto de atributos. Es decir, dada una función de similitud simétrica d que cumple con la desigualdad del triángulo, se dice que dos conjuntos de atributos X y Y establecen una dependencia funcional métrica si el conjunto de todas las tuplas con un valor x en el conjunto X al ser proyectado a Y yace dentro de una bola de diámetro δ. Es necesario destacar que este trabajo depende fuertemente de visualizar los datos como puntos en un espacio métrico altamente dimensional y que por lo tanto es particularmente apropiado para datos numéricos y texto. Partiendo de esta consideración, los autores desarrollaron un algoritmo para la inferencia de dependencias funcionales métricas a partir de algoritmos de geometría computacional.

No obstante que el uso de funciones de similitud en la inferencia de dependencias funcionales ha tenido como principal campo de aplicación el diseño de bases de datos relacionales y la limpieza de datos, también existen ejemplos de esta aproximación en áreas como la minería de datos. Debido a la importancia que tiene el descubrimiento automatizado de este tipo de relaciones, se han propuesto diversos tipos de dependencias que tienen como objetivo superar las limitaciones de una relación de equivalencia en la definición clásica de dependencia funcional. En el capítulo 2 hacemos una revisión de diversos artículos en los que los autores utilizan medidas de similitud para el calculo de dependencias con las que se realiza obtención automática de conocimiento, como el caso de las dependencias de vecindario [15] y las dependencias funcionales métricas [14].

De acuerdo a las ideas anteriores, existen diversas condiciones en una base de datos donde el cálculo de dependencias funcionales exactas no permite inferir de manera adecuada las relaciones existentes entre los datos. Además, es posible encontrar relaciones válidas que no están en función de una relación de equivalencia. No obstante, debido a su importancia para un gran número de disciplinas existe un gran interés en diseñar algoritmos que calculen estas relaciones. Es importante recordar que la inferencia de dependencias funcionales exactas es un problema difícil, exponencial en el número de atributos en una base de datos. Por supuesto, el cálculo de dependencias funcionales no exactas requiere procesamiento adicional para determinar la similitud entre dos tuplas de algún conjunto de atributos. En consecuencia, se requieren de algoritmos que procesen de manera adecuada el gran número de combinaciones de atributos que existen en algunas bases de datos.

Como fue mencionado en los trabajos anteriores, la inferencia de dependencias funcionales es una útil herramienta para la limpieza de bases de datos y el descubrimiento automatizado de conocimiento, por este motivo decidimos aplicar esta técnica para complementar el desarrollo del Laboratorio Digital de Violencia y Paz.

1.2. Estudio del caso: Laboratorio Digital de Violencia y Paz

El Laboratorio Digital de Violencia y Paz surge en el año 2017 tras haberse firmado un acuerdo de colaboración entre El Colegio de México (COLMEX), el Centro de Investigación y de Estudios

Avanzados del IPN (CINVESTAV) y el Instituto Nacional de Salud Pública (INSP) como un proyecto del seminario de violencia y paz; este seminario es un proyecto del COLMEX en el cual participan miembros de diversas instituciones como el Centro de Estudios Demográficos, Urbanos y Ambientales, el Centro de Estudios Históricos y el Centro de Estudios Sociológicos. La finalidad de este Laboratorio es crear un sistema de información geográfica, es decir, un sistema de software diseñado para capturar, almacenar, manipular, analizar, administrar y visualizar todo tipo de datos de naturaleza geográfica; para hechos de violencia y paz a nivel municipal en los Estados Unidos Mexicanos [16], el cual provea la información necesaria para proponer modelos sociodemográficos y hacer diagnóstico y pronóstico de políticas de gobierno.

De acuerdo el informe anual sobre la situación de pobreza y rezago social [17], para el año 2014, los indicadores de rezago social y acceso a servicios de salud fueron encabezados por los municipios de Ecatepec y Nezahualcóyotl. Por tal motivo es necesario tener herramientas basadas en el conocimiento que ayuden a mejorar la toma de decisiones respecto al establecimiento de los indicadores urbanos específicos para estas zonas, de tal forma que permitan evaluar de forma más acertada su bienestar social.

1.2.1. Modelos sociodemográficos e indicadores urbanos

Una variable sociodemográfica es aquella variable que se define en términos de un conjunto de características sociales y demográficas. Ejemplo de características demográficas son el sexo, la edad, el lugar de residencia, el nivel educativo, etc. Por otra parte, las características sociales están dadas por factores como pertenencia a un grupo social en particular [18]. En ese sentido un modelo sociodemográfico es un modelo matemático que incorpora una o más variables sociodemográficas.

Un indicador urbano es un instrumento cualitativo o cuantitativo que permite determinar el nivel de habitabilidad de una zona urbana. Para lograr lo anterior, un indicador puede ser definido a través de una o más variables sociodemográficas, o utilizar una variable sociodemográfica para describir el nivel de habitabilidad. A través de los indicadores urbanos es posible formular políticas, programas de desarrollo y proyectos que mejoren la habitabilidad de manera continua y sustentable. Para medir la efectividad de estos programas de desarrollo se pueden diseñar modelos sociodemográficos a partir de un conjunto de indicadores urbanos.

El programa ONU-HÁBITAT contempla los siguientes indicadores relacionados con los niveles de violencia y criminalidad en una zona urbana [19]:

- Violencia Urbana. Políticas existentes y nivel de aplicación para combatir la violencia urbana.
- Homicidios. Número de homicidios reportados anualmente por sexo por cada diez mil habitantes.

1.2.2. Los municipios de Nezahualcóyotl y Ecatepec de Morelos

De acuerdo a los datos del censo de población y vivienda del año 2010 [20] la población del Estado de México asciende a más de quince millones de personas, con una tasa de crecimiento del 1.4 por ciento. De entre todos los municipios del estado, Ecatepec y Nezahualcóyotl tiene la mayor cantidad de habitantes albergando cada uno de ellos a más de un millón de personas (ver Tabla 1.1).

Más del 50 % de la población del estado se concentra tan solo en diez municipios, y de estos, nueve se encuentran en el Valle de México. Ecatepec y Nezahualcóyotl, ubicados al oriente de la Ciudad de México concentran a más de 2.7 millones de personas.

Municipio	Número de habitantes
Ecatepec de Morelos	1656107
Nezahualcóyotl	1110565
Naucalpan de Juárez	833779
Toluca	819561
Tlalnepantla de Baz	664561

Tabla 1.1: Municipios con la mayor población en el Estado de México.

Por supuesto, el proceso de urbanización que se da en el estado es consecuencia de la cercanía de varios municipios con la Ciudad de México. Los procesos migratorios también contribuyen al proceso de urbanización, los datos muestran que el 58.4 % de la población del municipio de Nezahualcóyotl es nacida en otra entidad federativa. Por otra parte Ecatepec posee la mayor superficie de territorio urbanizado de la Zona Metropolitana de la Ciudad de México [21].

A partir de esta información se puede observar que la población de los municipios de Ecatepec y Nezahualcóyotl está compuesta por personas que habitan en ubicaciones densamente pobladas y con un alto nivel de urbanización. De ahí que resulte crítico el diseño de indicadores urbanos que permitan tomar decisiones de índole gubernamental para mejorar la calidad de vida de estas personas, en particular dada la realidad social de estos dos municipios.

Sin embargo, los datos de los planes de desarrollo municipal de Ecatepec y Nezahualcóyotl muestran que el índice delictivo en estos municipios ha ido en aumento. Siendo los delitos más frecuentes en ambos municipios los robos a vehículos y transeúntes (ver Tabla 1.2). Las autoridades municipales señalan que detrás de los altos índices de criminalidad hay problemas como el rezago social, la desintegración familiar, acceso a la educación, la corrupción y la impunidad [22, 23].

Delito	Ecatepec de Morelos	Nezahualcóyotl
Robo a vehículo	3645	928
Robo a transeúnte	199	66
Homicidios	122	101

Tabla 1.2: Incidencia delictiva entre enero y abril de 2016 en los municipios de Ecatepec de Morelos y Nezahuacóyotl [1]

De acuerdo el informe anual sobre la situación de pobreza y rezago social [17], para el año 2014, los indicadores de rezago social y acceso a servicios de salud fueron encabezados por los municipios de Ecatepec y Nezahualcóyotl.

1.3. Problema

Dado un conjunto de atributos en una base de datos relacional, encontrar todas las dependencias funcionales mínimas y no triviales de dicha relación que, usando funciones de similitud, permitan hallar nuevas relaciones para el descubrimiento automatizado de conocimiento.

1.4. Contribuciones

En el presente trabajo de tesis se propone una extensión al algoritmo TANE, para el cálculo de dependencias funcionales exactas, que permite inferir dependencias funcionales a partir del cálculo de clases de equivalencia con funciones de similitud. Estas funciones fueron escritas en C++ sobre una implementación existente del algoritmo TANE.

Los resultados muestran que esta aproximación permite inferir relaciones que no pueden ser halladas a través de la inferencia de dependencias funcionales exactas, dichas relaciones pueden ser de utilidad para el descubrimiento automatizado de conocimiento o tareas de minería de datos. Por otra parte, estos resultados también pueden ser utilizados para verificar la consistencia de los datos al revisar si una dependencia es válida o no en una relación. Procesos como el anterior son comúnmente utilizados en tareas de limpieza de datos y diseño de bases de datos relacionales.

Es importante destacar que nuestra aproximación tiene la ventaja de ser fácil de implementar ya que sólo se requiere modificar un algoritmo bien conocido. De acuerdo a la revisión hecha de la literatura, el tipo de dependencias obtenidas mediante nuestra aproximación no ha sido reportado en trabajos previos, sin embargo, son similares tanto en su objetivo como en su definición a dependencias como las métricas o las de diferencias.

Para validar el desempeño y la correctitud de la modificación del algoritmo se realizaron experimentos con distintos conjuntos de datos obtenidos del *UCI Machine Learning Repository* [24].

1.4.1. Publicaciones

Este trabajo dio lugar al artículo "Inferring functional dependencies through similarity functions in a crime database", presentado en la *16th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE 2019)* del 11 al 13 de septiembre en la CDMX.

1.5. Organización de la tesis

El presente trabajo de tesis está organizado como sigue: en el capítulo 2 realizamos una revisión de la teoría de bases de datos relacionales y elementos importantes para comprender las dependencias funcionales, así como su extensión a través de funciones de similitud. Adicionalmente hacemos una revisión de los trabajos relacionados en los que se calculan distintos tipos de dependencias, y damos una breve introducción a la minería de datos. En el capítulo 3 abordamos el problema del cálculo de dependencias funcionales y su complejidad. Además, se describen los algoritmos y heurísticas utilizados para resolver este problema, haciendo especial énfasis en algoritmo TANE. En el capítulo 4 realizamos la descripción de la metodología empleada para la inferencia de dependencias funcionales en distintos conjuntos de datos y se detallan las modificaciones realizadas al algoritmos TANE para inferir dependencias funcionales usando funciones de similitud. En el capítulo 5 presentamos los resultados relevantes de esta modificación al algoritmo, comparaciones con distintos conjuntos de datos y pruebas de desempeño para el mismo. Finalmente, en el capítulo 6 presentamos las conclusiones y trabajo futuro.

Parte I

Definiciones y descripción del problema

Capítulo 2

Bases de datos relacionales

2.1. Introducción

En este capítulo presentaremos las definiciones necesarias para comprender el modelo relacional de bases de datos, el esquema relacional y qué es una dependencia funcional y sus características. También explicamos de forma breve características importantes para llevar a cabo un adecuado proceso de limpieza en una base de datos.

Realizamos una revisión literaria de distintos artículos en los que se aborda el problema de encontrar dependencias que no pueden ser encontradas mediante una relación de equivalencia.

Finalmente damos una descripción breve de qué es la minería de datos y algunas de las técnicas utilizadas para ella.

2.2. El modelo relacional de bases de datos

El modelo relacional de bases de datos fue concebido a finales de la década de 1960 por Edgar F. Codd [25]. En dicho modelo, una base de datos está organizada mediante una colección de tablas, cada una de ellas representan una relación. Cada una está compuesta por un conjunto de tuplas que comúnmente son utilizadas para capturar información acerca del mundo real. Por lo que es utilizada para estructurar información a partir de las relaciones existentes entre los datos capturados. Asimismo, Codd [26] introdujo el término dependencia funcional en el año 1972.

Hoy en día el modelo relacional es el modelo de datos más utilizado para implementar bases de datos digitales, esto debido a su capacidad para representar información de manera eficiente y a que tiene un fundamento matemático sólido.

2.3. Atributos y relaciones

Sea $\mathscr{U} = \{A_1, A_2, \ldots, A_n\}$ el conjunto de finito de todos los atributos que aparecen en todas las tablas de la base de datos, a los atributos individuales los denotamos por $A, B, \ldots, E$ y a conjuntos de atributos como $R, S, \ldots, X, Y, Z$. Por conveniencia, no hacemos distinción entre $\{A\}$ y A. Para cada atributo tenemos un conjunto finito o infinito, $dom(A)$, de valores válidos para dicho atributo. Una relación $r = \{h_1, h_2, \ldots, h_n\} \in \mathscr{U}$ es un conjunto de tuplas, donde cada h_i está dada por:

$$h_i : \mathscr{U} \to \bigcup_{A \in \mathscr{U}} dom(A), \quad h_i(A) \in dom(A), \quad i = 1, \ldots, m,$$

donde $dom(A)$ es una correspondencia que define el dominio de cada atributo $A \in \mathscr{U}$.

Utilizaremos las letras t, u, v para denotar tuplas y T, U, V para un conjunto de ellas. Si t es una tupla en X, entonces denotamos a $t(X)$ como la tupla t restringida a los atributos de X.

2.4. Dependencias funcionales

Una *dependencia funcional* (DF[1]) es una relación de la forma $X \to A$, $X \subseteq \mathscr{U}$ y $A \in X$, entonces [27]:

- $X \to A$ denota que X determina funcionalmente al atributo A, *i. e.*, dado el valor de una tupla en X es posible determinar de manera única el valor de A.
- Sea f la DF $X \to A$, el lado izquierdo de f se denota por $izq(f) = X$ y el lado derecho como $der(f) = A$.
- Una DF $X \to A$ es válida en la relación r ($r \vDash X \to A$), si para cualquier par de tuplas t, $u \in r$, $t(B) = u(B), \forall B \in X \implies t(A) = u(A)$, $t \neq u$ (decimos que t y u *coinciden* en X y A). Definida de esta forma se le llama dependencia funcional *exacta*.
- Una DF $X \to A$ es mínima si A no depende funcionalmente de ningún subconjunto propio de X, *i. e.*, si al remover algún atributo del conjunto X la dependencia se vuelve inválida.
- Una DF $X \to A$ es trivial si $A \in X$.

En este contexto, una DF es una dependencia funcional exacta, *i. e.*, es una dependencia que está dada por una relación de equivalencia. El concepto de DF se puede generalizar a conjuntos, $X \to Y \Leftrightarrow \forall A \in X, X \to A$.

Para ejemplificar estas definiciones, consideremos la información contenida en la tabla 2.1. Esta constituye una relación sobre el siguiente conjunto de atributos $\mathscr{U} = \{\texttt{Nombre}, \texttt{Puesto}, \texttt{Edad}\}$, donde {Josué, Profesor, 37} es una tupla y el *dom*(Edad) va de 0 a 130. Además, en la tabla $\texttt{Nombre} \to \texttt{Puesto}$ y $\{\texttt{Puesto}, \texttt{Edad}\} \to \texttt{Nombre}$ son dependencias funcionales. Por otro lado $\{\texttt{Nombre}, \texttt{Puesto}\} \to \texttt{Nombre}$ es una DF trivial y $\{\texttt{Puesto}, \texttt{Edad}\} \to \texttt{Nombre}$ es una DF mínima.

Nombre	Puesto	Edad
David	Ayudante	23
Josué	Profesor	37
Francisco	Profesor	43
Ana	Secretaria	37

Tabla 2.1: Una tabla es una relación sobre un conjunto de atributos.

Estudiado por W. W. Armstrong en 1974, un sistema formal para DFs comprende un conjunto de reglas y axiomas. Sea F el conjunto de todas las dependencias funcionales que son válidas en una relación r. Se tiene un *sistema de Armstrong* si para todo $W, X, Y, Z \subseteq \mathscr{U}$ se cumplen un axioma y tres reglas de inferencia [26]:

1. **Reflexividad.** $\vdash X \to X \in F$
2. **Transitividad.** $(X \to Y \in F, Y \to Z \in F) \vdash X \to Z \in F$

[1] FD en la literatura en inglés.

3. **Acumulación y proyección.** $(X \to Y, X \subseteq W, Z \subseteq Y) \vdash W \to Z$

4. **Unión.** $(X \to Y \in F, Z \to W \in F) \vdash (X \cup Z \to Y \cup W \in F)$

Dado un conjunto F de DFs y una DF f, denotamos una *implicación* como $F \vDash f$, decimos que el conjunto F de DFs implica a la DF f, *i. e.*, cualquier relación que satisfaga F debe satisfacer. f. Una *derivación* $F \vdash f$, de $f = f_1, f_2, \ldots, f_n$ significa que f_i $(1 \leq i \leq n)$ es ya sea una instancia de un esquema axiomático o es consecuencia de las DFs anteriores en la secuencia, a través de una de las reglas de inferencia.

En un sistema de Armstrong definimos la solidez y completitud como

- **Sólido** si $F \vDash f$ es una consecuencia necesaria de $F \vdash f$.
- **Completo** si $F \vdash f$ es una consecuencia necesaria de $F \vDash f$.

El sistema formal de DFs, que es sólido y completo, consiste de un axioma y dos reglas de inferencia [28]:

- **Reflexividad.** $\vdash X \to \emptyset \in F$
- **Transitividad.** $(X \to Y \in F, Y \to Z \in F) \vdash X \to Z \in F$
- **Acumulación.** $X \to Y \in F \vdash (X \cup Z \to Y \cup Z)$

2.5. Esquema relacional

En una base de datos el esquema relacional representa el diseño lógico de la misma [29]. Por lo tanto, un esquema relacional no solo define cuál es el conjunto de atributos en una relación, sino que también debe especificar cuales son las relaciones existentes entre los atributos.

De manera formal un *esquema de relación* es un par (R, F), $R \subseteq \mathscr{U}$ donde R es un conjunto finito de atributos y F es un conjunto de dependencias funcionales sobre R [27] [30] . Un *esquema relacional* $\mathcal{D}$, es un conjunto de esquemas de relación $\mathcal{D} = \{(R_1, F_1), (R_2, F_2), \ldots, (R_k, F_k)\}$. Una relación r sobre $\mathscr{U}$ es llamada una instancia de $(\mathscr{U}, F)$, si cada dependencia funcional en F es válida en r.

El objetivo del esquema relacional para el diseño de la base de datos es evitar la existencia de redundancia y anomalías en las tuplas al realizar operaciones sobre la misma, tales como, escritura, borrado o actualizaciones. Es por ello que se establecen ciertas restricciones para el diseño de la base de datos, para facilitar el mantenimiento y administración de la misma. Estas restricciones las conocemos con el nombre de formas normales.

2.5.1. Proyección y junta

Sea (X, F) un esquema de relación, donde $X \subseteq \mathscr{U}$. Definimos la *proyección* de r en X como $\Pi_X(r) = \{t(X) | t \in r\}$. Sean $r_1, r_2, \ldots, r_k$ relaciones, definimos la *junta* de dichas relaciones como $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_k = \bowtie_{j=1}^{k} r_j = \{t | t(X_j) \in r_j \forall j \leq 1 \leq k\}$.

La proyección nos permite obtener una nueva relación a partir de un conjunto de atributos, mientras que la junta combina las tuplas de dos o más relaciones que concuerdan en atributos en

común para hacerlas una sola, por lo que son operaciones que guardan una estrecha relación y podrían pensarse como operaciones inversas, sin embargo, no siempre es el caso.

Sea r una relación, a la proyección de r sobre cada elemento de X le llamamos *descomposición*, $r \subseteq \bowtie_{j=1}^{m} \Pi_{X_j}$. Esta descomposición será sin pérdida cuando $r = \bowtie_{j=1}^{m} \Pi_{X_j}$, de otra forma tendrá pérdida.

2.6. Forma normal de Boyce-Codd

Dado un esquema de relación (R, F), un conjunto de atributos $X \subseteq R$ es un *determinante* de R si existe al menos un atributo $A \in R \backslash X$ tal que $F \vDash X \to A$. Si para toda $A \in R$ se tiene que $F \vDash X \to A$ entonces X es una *llave* de R. Si X es una llave y además si para alguna $B \in X$, $F \vDash X - B \nrightarrow A$ entonces X es llamada *llave mínima* de R [26].

Un esquema relacional $\mathcal{D}$ se encuentra en forma normal de Boyce-Codd (BCNF, por sus siglas en inglés) si cada vez que X es determinante de R entonces X es una llave de R, donde R es parte del esquema relacional $\mathcal{D}$. Lo cual implica que para un $A \in R \backslash X$ siempre que $X \to A$ es válida, necesariamente X es una llave mínima.

Es posible realizar la normalización de una base de datos hasta la BCNF a partir de (R, F) realizando una descomposición de la misma. Para ello, supongamos que un esquema relacional $\mathcal{D}$ no se encuentra en BCNF, sin pérdida de generalidad, asumamos que el esquema de relación (R, F) es el que causa la violación, *i. e.*, existe una X tal que es determinante de R pero no es una llave de R. Entonces existe un atributo $A \in R \backslash X$ tal que $F \vDash X \to A$, por tanto descompongamos el esquema de relación en dos, $D_1 = (\Pi_{X \cup A}(R_j, F_j^1))$ y $D_2 = (\Pi_{R_j \backslash A}(R_j, F_j^2))$, donde $F_j^1 = \Pi_{X \cup A}(F_j)$ y $F_j^2 = \Pi_{R_j \backslash A}(F_j)$, esta descomposición no tiene pérdida, ya que la presencia de dependencias funcionales garantiza descomposiciones sin pérdida[26]. El proceso de descomposición de forma continua de manera recursiva si D_2 no se encuentra en BCNF. Es un problema NP completo verificar si existe una violación de la BCNF en $\mathcal{D}$.

2.6.1. Limpieza de datos

Cualquier proceso de recolección de datos es propenso a errores, ya sea porque son recolectados manualmente, o porque no existe un proceso estandarizado para llenar los registros de una base de datos. Algunas de las fuentes de error más comunes incluyen errores ortográficos, registros con valores equivocados, valores faltantes y valores repetidos. La mayoría de las fuentes de error pueden ser catalogadas en alguna de las siguientes categorías [31]:

- **Errores de llenado de datos.** Causados por humanos al momento de registrar datos.
- **Errores de medición.** Se presentan cuando alguna unidad física no es registrada adecuadamente por problemas de diseño, calidad o calibración.
- **Errores de destilación.** Producidos cuando los datos originales son resumidos o preprocesados antes de formar parte de la base de datos.
- **Errores de integración de los datos.** Ocasionados al juntar datos de distintas fuentes, sobre todo causan inconsistencias en la representación de los datos.

Ya que la inferencia de dependencias funcionales depende de poder corroborar que dos tuplas tengan valores idénticos de acuerdo a uno o más atributos, es necesario garantizar la calidad de los datos antes de que los mismos puedan ser analizados. En particular es necesario garantizar que los datos cumplan con los siguientes requerimientos [32]:

- **Precisión.** Los datos son consistentes con los valores reales.
- **Temporalidad.** Los datos están actualizados.
- **Completitud.** Todos los registros correspondientes a un atributo han sido llenados.
- **Consistencia.** La representación de los datos para un mismo atributo es uniforme.
- **Unicidad.** No existen datos duplicados.

De acuerdo con lo anterior, se dice que los datos tienen una anomalía si cualquiera de estos requerimientos no ha sido satisfecho. Por lo tanto, es necesario un proceso que garantice que, de existir alguna de estas anomalías, las mismas puedan ser corregidan. Dicho proceso es llamado *limpieza de datos*. Cada una de las anomalías que pueden existir en un conjunto de datos puede ser clasificada como sigue[32]:

- **Sintácticas.** Se refiere a errores léxicos y de formato, *i. e.*, valores que no cumplen con el formato especificado para un atributo en particular.
- **Semánticas.** En esta categoría se violan las restricciones impuestas sobre los valores de un atributo, tuplas duplicadas y datos inválidos.
- **De cobertura.** En esta categoría están comprendidos los errores debido a datos faltantes.

Debido a la enorme cantidad de datos que pueden existir en una base de datos, es necesario automatizar este proceso. En general un proceso de limpieza de datos debe tener los siguientes elementos [33]:

1. Encontrar todas las anomalías en el conjunto de datos.
2. Determinar el tipo de cada una.
3. Corregirlas.
4. Verificar que cada anomalía haya sido corregida.

2.7. Dependencias basadas en DFs

Como observamos, las DF nos permiten establecer relaciones de dependencia entre los atributos, sin embargo, estas dependencias pueden no ser lo suficientemente generales para establecer muchas de las relaciones existentes en las bases de datos.

2.7.1. Dependencias multivaluadas

Tomemos como ejemplo el caso de una base de datos de empleados, en la cual cada uno de ellos tiene asociado su conjunto de hijos. Es evidente que esta relación entre los empleados y sus hijos existe pero una dependencia funcional no es capaz de establecer esta relación ya que cada empleado puede tener más de un hijo.

En un esquema de relación (R, F), una *dependencia multivaluada* [26] (mvd, por sus siglas en inglés) ocurre cuando los valores en X determinan a un conjunto de valores en Y independientes del conjunto de valores en $R \backslash Y$, *i. e.*, si $X, Y \in R$, $X \twoheadrightarrow Y$ (X *multidetermina* Y) si por cada relación $r \in R$, para todas las tuplas $u, v \in r$, si $u(X) = v(X)$, entonces existe una tupla $w \in r$ tal que

$$\begin{aligned} w(X) &= u(X) = v(X), \\ w(Y) &= v(Y), \\ w(R \backslash \{X \cup Y\}) &= v(R \backslash \{X \cup Y\}) \end{aligned}$$

Se cumple que en las relaciones de r, $X \twoheadrightarrow Y$ cuando r se puede descomponer en las proyecciones sobre $X \cup Y$ y $R \backslash X \cup Y$ sin tener pérdida de información. Lo que podemos expresar como

$$r = \Pi_{X \cup Y}(r) \bowtie \Pi_{X \cup (R \backslash Y)}(r).$$

La cuarta forma normal (4NF, por sus siglas en inglés) se encuentra asociada a las dependencias multivaluadas; ya que estas son una generalización de las dependencias funcionales, en consecuencia, cualquier dependencia multivaluada se encuentra automáticamente en BCNF. Definimos la 4NF de forma análoga a la BCNF pero cada dependencia debe ser formada como consecuencia de una llave mínima.

Dado un esquema de relación $\mathcal{D} = (R, F)$ que obedece solo aquellas DFs y mdvs que son consecuencia lógica de un conjunto de DF, entonces la BCNF coincide con la 4NF [26].

2.7.2. Dependencias funcionales aproximadas

Este tipo de dependencias funcionales son llamadas de esta manera porque se requiere que una dependencia funcional $X \to Y$ sea satisfecha al menos de manera parcial, *i. e.*, existe una cierto número de tuplas para las cuales la dependencia no es válida.

Con el objetivo de definir de manera precisa cuando una *dependencia funcional aproximada* es considerada como válida se utiliza una función de *error* $g(f, r)$, donde f es alguna dependencia funcional candidata y r una relación. Si el valor de $g(f, r)$ es menor o igual que el del parámetro ϵ, entonces se dice que la dependencia f es aproximadamente satisfecha por r o que es ϵ-buena, de lo contrario se dice que $g(f, r)$ es aproximadamente violada por r o que es ϵ-mala [34]. Debe tenerse en cuenta que la elección de la función de error es de importancia crítica para la inferencia de dependencias funcionales aproximadas, distintas funciones de error darán resultados distintos. Como consecuencia, es necesario tener en cuenta las características de los datos y el campo de aplicación para seleccionar una función de error adecuada. En [6] se proponen las siguientes funciones de error:

$$\begin{aligned} G_1(X \to Y, r) &= |\{(u, v) : u, v \in r, u(X) = v(X), u(Y) \neq v(Y)\}|, \\ G_2(X \to Y, r) &= |\{u : u \in r, \exists v \in \mathscr{U}, u(X) = v(X), u(Y) \neq v(Y)\}|, \\ G_3(X \to Y, r) &= |r| - \text{máx}\{|s| : s \subseteq r, s \vDash X \to Y\}. \end{aligned}$$

Claramente, G_1 representan el número de pares que violan una dependencia funcional, G_2 el número de tuplas que la violan, y G_3 el número de tuplas que deben ser borradas para que $X \rightarrow Y$ sea válida.

Es importante notar que varios de los algoritmos para la inferencia de dependencias funcionales exactas pueden ser adaptados para calcular dependencias funcionales aproximadas [7] [35]. En ese sentido, la complejidad del problema de inferir todas dependencias aproximadas mínimas y no triviales está dada por la complejidad del algoritmo utilizado además del costo del cálculo de la función $g(f, r)$. Cada una de estas medidas puede ser calculada en tiempo $O(n|r| \log |r|)$[6].

2.7.3. Dependencias funcionales condicionales

Consideradas como una extensión a las DFs [36], este tipo de dependencias funcionales tienen el objetivo de imponer una restricción sobre un conjunto de datos de manera que sea posible verificar la consistencias de los datos. Como consecuencia, este tipo de dependencias son principalmente utilizadas durante el proceso de limpieza de datos, además de que son más efectivas para detectar y reparar inconsistencias en los datos.

Dado un esquema de relación (R, F), una *dependencia funcional condicional*, denotada por ϕ, sobre el conjunto de atributos R es un par $(X \rightarrow Y, t_p)$[37], donde:

- $X, Y \in R$.
- $X \rightarrow Y$ es una dependencia funcional exacta sobre R.
- t_p es una *tupla patrón* con todos los atributos en X y Y. Por cada atributo A en X o Y, $t(A)$ es una constante $a \in dom(A)$, o una variable sin nombre '_' que toma valores del $dom(A)$.

Para una tupla t_c se define una correspondencia ρ de t_c a una tupla de datos sin variables, tal que para cada atributo $A \in X \cup Y$ si $t_c(A) =$ _, entonces ρ es la imagen de esta tupla a una constante en $dom(A)$. Por otra parte si $t_c(A) = a$ entonces ρ es la imagen de t_c a la misma constante a. De acuerdo a lo anterior, se dice que una tupla de datos t cumple con un patrón dado por t_c, lo cual se denota con $t \asymp t_c$, si existe una ρ tal que $\rho(t_c) = t$.

Una relación r sobre R satisface la dependencia condicional ϕ, lo cual se denota por $r \vDash \phi$ si para cada par de tuplas $t_1, t_2 \in r$ y para cada tupla t_c de ϕ, si $t_1(X) = t_2(X) \asymp t_c(X)$ entonces $t_1(Y) = t_2(Y) \asymp t_c(Y)$. Es decir si $t_1(X)$ y $t_2(X)$ son iguales y ambos cumplen con el patrón $t_c(X)$, entonces $t_1(Y)$ y $t_2(Y)$ deben ser iguales y ambas deben cumplir con el patrón $t_c(Y)$.

2.8. Dependencias funcionales basadas en funciones de similitud

Las dependencias presentadas anteriormente presentan algún parecido o extensión a las DFs, generalizando o solucionando restricciones que las DF exactas tienen. Otra forma de relajar las restricciones de las dependencias funcionales con el fin de encontrar dependencias que aunque obvias no se presentan en cierta base de datos, es utilizar funciones de similitud.

2.8.1. Funciones de similitud

Una *función de similitud* es una función $s : X \times X \to \mathbb{R}^+$ que asocia a cada par de elementos en X un número real, que determina la similitud entre sus elementos. La función tiene un valor más grande cuanto más similares sean los elementos. En general, una función de similitud tiene las siguientes propiedades, para toda $x, y \in X$

1. **No negatividad:** $s(x, y) \geq 0$.
2. **Maximalidad:** $s(x, x) \geq s(x, y)$.
3. **Simetría:** $s(x, y) = s(y, x)$.

Se dice que una función de similitud está normalizada si su codominio es el intervalo $[0, 1] \in \mathbb{R}$.

Una función de similitud puede ser utilizada durante las siguientes tareas:

- Identificar datos con valores distintos debido a errores ortográficos.
- Determinar instancias equivalentes de un atributo en conjuntos de datos distintos. Por ejemplo, nombres o direcciones que son idénticas pero contienen errores ortográficos.
- Identificar grupos de datos con características similares, *i. e.*, agrupamiento.

De forma análoga, es posible definir el concepto de función de *función de distancia*, o *métrica* como

$$d : X \times X \to \mathbb{R}^+,$$

donde $\mathcal{R}^+$ representa el conjunto de los números reales no negativos. Satisfacen las siguientes propiedades para toda $x, y, z \in X$

1. **No negatividad:** $d(x, y) \geq 0$
2. **Identidad de los indiscernibles:** $d(x, y) = 0 \Leftrightarrow x = y$
3. **Simetría:** $d(x, y) = d(y, x)$
4. **Desigualdad del triángulo:** $d(x, y) \leq d(x, z) + d(z, y)$

También podemos llamarle *función de disimilitud*, ya que es una función que expresa que tan diferentes son dos elementos. Tanto las funciones de similitud como las de disimilitud constituyen medidas de proximidad. Algunas de las funciones de distancia comúnmente utilizadas son las siguientes:

1. **Distancia euclidiana.** Esta función está definida como:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2},$$

donde n es el número de dimensiones (atributos), y x_k y y_k son los i-ésimos atributos de x y y.

2. **Distancia de Minkowski.** Esta función es una generalización de la distancia euclidiana.

$$d(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^r \right)^{1/r},$$

donde $r = 2$ en el caso de la distancia euclidiana.

3. **Distancia de Levenshtein.** Esta función define la distancia entre dos cadenas de texto como el número mínimo de transformaciones que es necesario realizar sobre la primera para convertirla en la segunda. Needleman y Wunsch [38] definen esta función de distancia a través de la siguiente ecuación de recurrencia:

$$D(s[1:i], t[1:j]) = \text{mín} \begin{cases} D(s[1:i-1] & , t[1:j-1]) + c(s[i], t[j]) \\ D(s[1:i-1] & , t[1:j]) + c(s[i], \epsilon) \\ D(s[1:i-1] & , t[1:j]) + c(\epsilon, t[j]) \end{cases},$$

donde $D(s[i:i], t[1:j])$ es la distancia entre los prefijos de longitud i y j en las cadenas s y t respectivamente, $c(s[i], t[j])$ es el costo de sustituir el i-ésimo caracter de s con el j-ésimo caracter de t, $c(s[i], \epsilon)$ es el costo de borrar el i-ésimo caracter de s y $c(\epsilon, t[j])$ es el costo de insertar el j-ésimo caracter de t. Es importante notar que también existen funciones de similitud para vectores booleanos, así como funciones basadas en entropía, densidad, etc [12]. Es crítico tomar en cuenta la naturaleza de un atributo para elegir una función de similitud apropiada [39].

2.8.2. Dependencias funcionales basadas en funciones de distancia

Una DF tal y como se definió en la sección 2.3 no siempre es apropiada para determinar si los atributos de una base de datos están relacionados de manera significativa. Para clarificar esta idea consideremos la tabla 2.2, la cual muestra un pequeño subconjunto del conjunto de datos `iris`. La tabla contiene datos taxonómicos para tres especies distintas de plantas de la familia iris. Es bien sabido que este conjunto de datos solo contiene 4 dependencias funcionales mínimas $X \rightarrow A$, $|X| = 3$, *i. e.*, es necesario conocer al menos tres de las características para poder determinar la especie de la planta. Es importante notar que en la tabla 2.2 la tupla $(4.8, 1.8)$ en las filas 5 y 7 impide que la DF siguiente sea válida:

$$(\texttt{Longitud del petalo}, \texttt{Ancho del petalo}) \rightarrow \texttt{Clase}$$

Ya que es poco razonable que características como la longitud o el ancho del pétalo de una flor sean siempre idénticos cuando se recolectan datos de distintos especímenes, es claro que una dependencia que capture esta relación debe tomar en cuenta de manera explicita dichas diferencias.

2.8.3. Dependencias de acoplamiento

Consideremos una relación r sobre un esquema de relación (R, F), una *dependencia de acoplamiento* es una expresión de la forma

$$(X, Y, \lambda),$$

donde

Tabla 2.2: Un subconjunto del conjunto de datos iris.

Longitud sépalo	Ancho sépalo	Longitud pétalo	Ancho pétalo	Clase
5.1	3.5	1.4	0.2	Setosa
4.9	3.0	1.4	0.2	Setosa
4.7	3.2	1.3	0.2	Setosa
6.5	2.8	4.6	1.5	Versicolor
5.9	3.2	4.8	1.8	Versicolor
6.1	2.8	4.0	1.3	Versicolor
6	3	4.8	1.8	Virginica
6.9	3.1	5.4	2.1	Virginica
6.7	3.1	5.6	2.4	Virginica

- $X, Y \subseteq R$
- $\lambda \in \mathbb{R}^+$ es un umbral de similitud para acoplamiento para algún atributo $A \in X \cup Y$.

De manera formal una dependencia de acoplamiento (ϕ) se define como [11]:

$$\phi = \bigwedge_{A_i \in X} t_1(A_i) \approx_{\lambda(A_i)} t_2(A_i) \implies \bigwedge_{A_j \in Y} t_1(A_j) \approx_{\lambda(A_j)} t_2(A_j), \tag{2.1}$$

donde

- $\lambda(A_i)$ y $\lambda(A_j)$ son los umbrales de similitud para acoplamiento de los atributos A_i y A_j respectivamente.
- $\approx_\delta$ es un operador de similitud basado en una función de distancia δ.

Una dependencia definida de acuerdo a la Ecuación 2.1 específica una restricción que debe ser cumplida por un par de tuplas sobre dos conjuntos de atributos X y Y.

Una manera de evaluar el nivel de acoplamiento entre dos conjuntos de atributos es utilizar de medidas de soporte y confianza.

En [10], los autores determinan el nivel de acoplamiento definiendo una distribución estadística $\mathscr{D} = (A_1, \ldots, A_m, P)$, donde cada atributo A_i registra la similitud entre todos los pares de tuplas sobre el atributo A_i en r, y P es un parámetro estadístico. Si s es una tupla estadística para la calidad del acoplamiento en $\mathscr{D}$, entonces cada $s(A_i)$ denota la similitud entre dos tuplas sobre el atributo $A_i \in R$. Por otro lado, $s(P)$ denota la probabilidad de que un par de tuplas, t_1 y t_2 en r, tengan similitud $s(A_i)$, $\forall A_i \in r$.

Una vez obtenido $\mathscr{D}$ es posible obtener las medidas de soporte y confianza para los atributos X y Y a partir de la distribución estadística de $\mathscr{D}$. Sean λ_X y λ_Y las proyecciones del patrón de los umbrales de similitud λ sobre los atributos X y Y respectivamente, en una dependencia acoplada ϕ. Además, sea Z el conjunto de atributos que no aparecen en ϕ, *i. e.*, $R \setminus (X \cup Y)$. Las medidas

de soporte y confianza están dadas por:

$$\begin{aligned} soporte(\phi) &= P(X \vDash \lambda_X, Y \vDash \lambda_Y) \\ &= \sum_Z P(X \vDash \lambda_X, Y \vDash \lambda_Y, Z), \\ confianza(\phi) &= P(Y \vDash \lambda_Y \mid X \vDash \lambda_X) \\ &= \frac{\sum_Z P(X \vDash \lambda_X, Y \vDash \lambda_Y, Z)}{\sum_{Y,Z} P(X \vDash \lambda_X, Y, Z)}, \end{aligned}$$

donde el operador $\vDash$ denota que los valores de similitud en todos los atributos de X satisfacen los umbrales de similitud para de acoplamiento dados por λ_X. Dadas dos tuplas t_1 y t_2 en r, la medida $soporte(\phi)$ estima la probabilidad de que la similitud de t_1 y t_2 sobre X y Y satisfagan los umbrales de acoplamiento dados por λ_X y λ_Y, respectivamente. De manera similar, la $confianza(\phi)$ determina la probabilidad de que la similitud entre t_1 y t_2 sobre Y satisface los umbrales especificados por λ_Y dada la condición de que ambas son similares sobre los atributos de X, *i. e.*, $X \vDash \lambda_X$, pero no similares sobre los atributos de Y. En aplicaciones donde se desea detectar inconsistencias en los datos se prefieren dependencias acopladas con una una medida de confianza alta. Mientras que en otro tipo de aplicaciones se prefiere que la medida de soporte sea alta.

El cálculo de las dependencias de acoplamiento depende entonces de las medidas de soporte (η_s) y confianza (η_c) deseadas. En general se desea obtener todas las dependencias de acoplamiento que tengan al menos η_s y η_c. Aunque esto puede cambiar dependiendo de la aplicación. Los algoritmos que calculan umbrales de acoplamiento de manera que se satisfagan los requerimientos η_s y η_c tienen complejidad lineal en el tamaño de $\mathscr{D}$.

2.8.4. Dependencias funcionales métricas

Una *dependencia funcional métrica* es una generalización de una dependencia funcional exacta que parte de la idea de remplazar la relación de equivalencia en el lado derecho de una dependencia funcional exacta por una métrica d. Sea (R, F) un esquema de relación y r una relación sobre dicho esquema, una dependencia funcional métrica es una relación de la forma

$$X \xrightarrow{\delta} Y,$$

donde

- $X, Y \subseteq \mathscr{U}$.
- δ es un parámetro que representa la distancia máxima que puede existir entre dos tuplas para considerar que son similares. Para calcular δ se recurre a una métrica $d : dom(Y) \times dom(Y) \to \mathbb{R}^+$. Es decir, solo se revisa la similitud entre tuplas en el lado derecho de la dependencia, el lado izquierdo sigue estando sujeto a una relación de equivalencia.
- La métrica d cumple con las propiedades mencionadas en la subsección 2.8.1.

Sea P un conjunto de puntos en algún espacio métrico, el diámetro de P denotado como Δ_d representa la distancia máxima que existe entre un par de puntos en P bajo la métrica d. De acuerdo a lo anterior se dice que una dependencia funcional métrica es válida si:

$$\max_{T \in \pi_x} \Delta_d(T(Y)) \leq \delta,$$

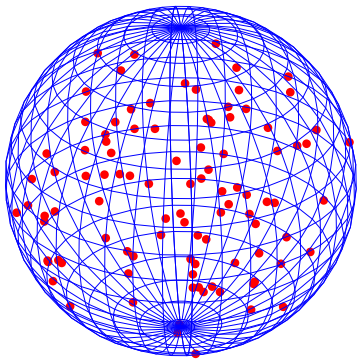

Figura 2.1: Cada punto en la bola de diámetro d representa una tupla en el lado derecho de una dependencia funcional métrica.

donde $T(Y)$ es el conjunto de todas las tuplas del conjunto de atributos Y. Dicho de otra manera, la dependencia es válida si $\forall x \in dom(X)$, el conjunto de tuplas donde $T(X) = x$, al ser proyectado a Y, yace dentro de una bola de diámetro d (ver Figura 2.1).

Como consecuencia de la discusión anterior, es fácil ver que para verificar si una dependencia funcional métrica $X \xrightarrow{\delta} Y$ es válida en una relación r basta con calcular el diámetro de un conjunto de puntos en un espacio métrico definido por d. Para datos de naturaleza numérica y de dimensionalidad baja basta con calcular la envolvente convexa de un conjunto de puntos y posteriormente obtener el diámetro deseado. En dos o tres dimensiones el cálculo de la envolvente convexa puede realizarse en $O(n \log n)$, sin embargo, para dimensiones mayores este procedimiento no siempre es factible. Debido a lo anterior, en [14] se propone utilizar un algoritmo que garantiza una aproximación al diámetro deseado en $O(n + 1/\alpha^{\frac{3(\delta-1)}{2}})$. Cabe destacar, que si no es posible encontrar una función de similitud de manera que sea posible construir un espacio métrico, entonces este tipo de dependencias no pueden ser obtenidas.

2.8.5. Dependencias de diferencias

Con un objetivo similar al de las dependencias de acoplamiento y las dependencias funcionales métricas, en [13] se introduce el concepto de dependencia de diferencia. En lugar de simplemente reemplazar la relación de equivalencia en la definición de una DF exacta, los autores utilizan funciones de similitud para calcular la diferencia entre los valores de dos tuplas con respecto de un atributo en particular. Si dicha diferencia cumple con las restricciones impuestas sobre la dependencias, entonces la dependencia será válida. Al igual que en el caso de las dependencias funcionales métricas, por cada atributo X en una relación r se establece una métrica de distancia $\delta : dom(X) \times dom(X) \to \mathbb{R}^+$, esta métrica debe cumplir con las propiedades descritas en 2.8.1.

Una vez dada δ es posible definir la función de diferencias Φ_X, esta función establece una restricción sobre el atributo X. De acuerdo a los autores, Φ_X está dada por:

$$\Phi_X = \begin{cases} verdadero, \text{ si } (t_1, t_2) \asymp d \\ falso, \text{en cualquier otro caso} \end{cases},$$

donde $(t_1, t_2) \asymp d$ significa que la distancia entre el par de tuplas $t_1, t_2 \in r$, bajo la métrica δ, cumple la restricción impuesta por Φ_X de acuerdo a algún operador y un parámetro d apropiados.

Por ejemplo, $\Phi_B = B(\leq 0.1)$ implica que para que Φ_B tenga un valor verdadero, entonces la distancia entre dos tuplas t_i y t_j con respecto del atributo B debe ser menor o igual que 0.1, en cuyo caso decimos que $(t_1, t_2) \asymp 0.1$.

Usando las definiciones anteriores, es posible definir una función de diferencias sobre un conjunto de atributos Z de la manera siguiente:

$$\Phi_Z = \bigwedge_{A_i \in Z} \Phi_{A_i}.$$

Esto es, para cualesquiera dos tuplas t_1, t_2 la función de diferencias Φ_Z tiene el valor $verdadero$, si la distancia correspondiente para cada atributo en Z cumple con la restricción impuesta por la métrica δ. De acuerdo a lo anterior una dependencia diferencial es una expresión de la forma:

$$\Phi_X \rightarrow \Phi_Y,$$

donde X y Y son conjuntos de atributos, y Φ_X, Φ_Y son las funciones de diferencias sobre los atributos X y Y. La dependencia será válida si cada Φ_X es verdadero y cada Φ_Y es verdadero.

Cabe destacar que al igual que con otros tipos de dependencias, el cálculo de las dependencias diferenciales involucra un problema cuya complejidad es exponencial en el número de atributos en la relación. En [13] se presenta un algoritmo que utiliza diversos métodos de poda en el espacio de soluciones para la inferencia de dependencias diferenciales, la complejidad de este algoritmo es $O\left(|r|^2|\mathscr{U}|s^{|\mathscr{U}|}\right)$, donde s es el tamaño máximo del conjunto de funciones diferenciales definidas para cada atributo.

2.8.6. Dependencias de vecindad

No obstante que el uso de funciones de similitud en la inferencia de dependencias funcionales ha tenido como principal campo de aplicación el diseño de bases de datos relacionales y la limpieza de datos, también existen ejemplos de esta aproximación en áreas como la minería de datos. En [15] los autores presentan una metodología para la predicción de variables objetivo a través de un nuevo tipo de dependencias funcionales conocidas como dependencias de vecindad. La idea principal detrás de la metodología es que si dos tuplas tienen valores cercanos con respecto de un conjunto de variables predictoras, entonces también deberían tener un valor cercano con respecto de la variable objetivo. En una base de datos las variables (tanto las variables predictoras como la variable objetivo) están dadas por los atributos, tomando en cuenta lo anterior a cada atributo le es asignado una función f_i que determina el grado de similitud entre un par de tuplas. A diferencia de otros trabajos, la única restricción impuesta sobre cada f_i es que su dominio debe ser el intervalo $[0, 1]$. Usando las funciones de similitud correspondientes un predicado de vecindad determina si dos o más tuplas son vecinas. El método propuesto por los autores es particularmente interesante porque permite determinar que tan bueno es un predicado para predecir una variable objetivo, esto debido a que una vez determinada una vecindad, la misma puede ser utilizada para medir la confianza del predicado, i.e., la probabilidad de que dos o más tuplas sean vecinas.

2.8.7. Dependencias secuenciales

Para descubrir las relaciones existentes en los datos, es conveniente tomar en cuenta cualquier característica intrínseca que brinde información acerca de su naturaleza. Ejemplo de esto son las

dependencias secuenciales [40], este tipo de dependencias tienen como objetivo expresar las relaciones existentes en atributos cuyos valores pueden ser ordenados, *e. g.*, número de ventas, fechas, horas, temperatura, etc. Dado un intervalo g, una dependencia secuencial sobre un par de atributos X y Y establece que la distancia entre dos tuplas consecutivas del atributo Y, cuando los datos son ordenados con respecto de X, está dentro de g. Relaciones como está pueden ser utilizadas para encontrar inconsistencias en los datos, *e. g.*, datos faltantes que se supone deben de aparecer con cierta frecuencia. Al igual que en las dependencias funcionales de vecindad, los autores utilizan una medida de confianza para determinar el grado de satisfacción de una dependencia secuencial. Como caso de prueba, los autores muestran como obtener el conjunto mínimo de tuplas donde una dependencia es válida, dicha información es utilizada en el análisis semántico de bases de datos, así como para verificar la consistencia de los datos.

2.9. Descubrimiento de conocimiento en bases de datos

El *descubrimiento de conocimiento en bases de datos* (KDD, por sus siglas en inglés) es el proceso mediante el cual se obtiene conocimiento, que aunque presente en una base de datos, carece de una representación explicita en la misma [41]. Por lo tanto, cualquier sistema que lleve a cabo descubrimiento de conocimiento debe ser capaz de encontrar, de manera eficiente, patrones y relaciones en los datos que sean útiles con respecto a un dominio de conocimiento en particular. Es deseable que los sistemas que lleven a cabo KDD necesiten la menor intervención humana posible, e idealmente deben ser completamente automatizados. Para lograr lo anterior el proceso de KDD hace uso de herramientas como el aprendizaje de máquina y la minería de datos.

2.9.1. Minería de datos

La minería de datos es el conjunto de procesos mediante los cuales, se organiza, analizan y combinan grandes conjuntos de datos con el objetivo de identificar patrones y extraer información. Para lograr lo anterior, la minería de datos hace uso, entre otras, de la siguientes técnicas [42]:

- Aprendizaje de máquina. Esta disciplina investiga cómo una computadora puede mejorar su desempeño en una tarea en particular a partir de un conjunto de datos de entrenamiento. Para lograr lo anterior se recurren a técnicas como el aprendizaje supervisado, donde una computadora es entrenada con un conjunto de datos preclasificados que ayuda a la computadora a inferir los resultados correctos. Por otra parte, en las técnicas de aprendizaje no supervisado los datos de entrada no están clasificados y deben de usarse métodos de agrupamiento para realizar la clasificación.

- Métodos de visualización científica. La visualización es el conjunto de técnicas que permiten comprender o extraer conocimiento a partir de la representación gráfica de un conjunto de datos. Dicho conocimiento puede ayudar a interpretar de mejor manera el resultado de mediciones o simulaciones de un modelo matemático.

- Análisis estadístico. Este tipo de modelos consiste en un conjunto de funciones matemáticas definidas en términos de una o más variables aleatorias y sus distribuciones de probabilidad correspondientes. En minería de datos, los modelos estadísticos pueden ser utilizados para caracterizar y clasificar un conjunto de datos, para construir un modelo de predicción.

Capítulo 3

Inferencia de dependencias funcionales

3.1. Introducción

Con base en las definiciones presentadas en el capítulo 2 es posible enunciar el problema de la inferencia de dependencias funcionales de la manera siguiente, tomando las definiciones de Mannila y Räihä [43]:

Dada una relación r sobre un esquema de relación (R, F), si F es un conjunto de dependencias funcionales, entonces $r \vDash F$ significa que todas las dependencias de F son válidas en r. Si $X \to Y$ es una sola dependencia, $r \vDash X \to Y$ significa que $r \vDash \{X \to Y\}$. Al conjunto de las dependencias válidas en r se le denota como $dep(r)$, *i. e.*,

$$dep(r) = \{X \backslash A \to A | X, Y \subseteq R, r \vDash X \backslash \to A\}$$

La dependencia $X \to Y$ es una consecuencia de F se denota $F \vDash X \to Y$, si $r \vDash F$ implica que $r \vDash X \to Y$ para todas las relaciones r.

Si F y G son conjuntos de equivalentes de dependencias, entonces todas las dependencias de G son consecuencia de F y *vice versa*, decimos que F es una cubierta de G (y G es una cubierta de F). En general $dep(r)$ tiene varias cubiertas equivalentes de distintos tamaños, el *problema de la inferencia de dependencias funcionales* es encontrar una cubierta pequeña para $dep(r)$. Este problema tiene aplicación en el diseño de bases de datos, optimización de consultas y limpieza de datos [3].

Es posible obtener todas las dependencias funcionales exactas de una relación utilizando un algoritmo de fuerza bruta:

Algoritmo 3.1 Algoritmo de fuerza bruta para la inferencia de FDs.

ENTRADA: Una relación $r \in R$.
SALIDA: Un conjunto F que contiene todas las FD's válidas en r.

```
1:  procedimiento ObtenerFDs(r)
2:      F ← ∅
3:      para X ∈ P(R) hacer
4:          para A ∈ R \ X hacer
5:              si r ⊨ X → A entonces
6:                  F ← F ∪ {X → A}
7:              fin si
8:          fin para
9:      fin para
10: fin procedimiento
```

El algoritmo 3.1 analiza todos los elementos en el conjunto potencia de R. Cada conjunto de atributos $X \in \mathcal{P}(R)$ representa un dependencia funcional candidata para algún conjunto de atributos $A \in R$, por lo que el algoritmo simplemente verifica una a una todos las posibles dependencias candidatas.

3.2. Complejidad de la inferencia de dependencias funcionales

La complejidad de el algoritmo 3.1 es $O(n^2 2^n |r| \log |r|)$ [43]. Por supuesto un algoritmo como este es poco adecuado para la mayoría de las aplicaciones del mundo real. Se ha demostrado que el problema de inferir las dependencias funcionales de una relación r requiere al menos $\Omega(|r| \log |r|)$ incluso para un esquema de relación con solo dos atributos. Además por cada atributo n existe una relación r por lo que al crecer el número de atributos crece el número de relaciones de forma lineal $|r| = O(n)$, sin embargo, cada cubierta para $dep(r)$ tiene $\Omega(2^{n/2})$ dependencias por lo que no puede existir un algoritmo que resuelva este problema en tiempo polinomial[6].

Lo que es posible, es encontrar un algoritmo que sea polinomial respecto al tamaño de la relación y al tamaño de la cubierta más pequeña de $dep(r)$. De hecho, se ha demostrado que la existencia de un algoritmo polinomial para resolver el problema de la inferencia de dependencias funcionales implicaría la existencia de algoritmos similares para varios problemas abiertos.

Debido a la amplia gama de aplicaciones de las dependencias funcionales se han desarrollado varios algoritmos, que de alguna manera mejoran la complejidad respecto al algoritmo 3.1, para la inferencia de este tipo de relaciones.

3.3. Algoritmos para la inferencia de dependencias funcionales

En esta sección revisaremos distintos algoritmos propuestos en la literatura para la inferencia de dependencias funcionales así como algunos de los conceptos principales para entender los mismos. Estos algoritmos son clasificados como ascendentes y descendentes [34], nosotros les llamaremos algoritmos de retícula y basados en conjuntos. Ambos tipos de algoritmos hacen uso de las particiones para calcular las dependencias funcionales.

3.3.1. Particiones

Decimos que dos tuplas t y u son *equivalentes* respecto a un conjunto de atributos X si $\forall A \in X, t(A) = u(A)$. Cualquier conjunto de atributos X hace particiones de las tuplas de la relación en clases de equivalencia. Denotamos la *clase de equivalencia* de una tupla $t \in r$ respecto a un conjunto $X \subseteq R$ como $Q_X(t)$, *i. e.*, $Q_X(t) = \{\forall A \in X, u \in r | t(A) = u(A)\}$. Al conjunto $\pi_X = Q_X(t) | t \in r$ de clases de equivalencia le llamamos una *partición* de r sobre X. Esto es, π_X es una colección de conjuntos disjuntos (clases de equivalencia) de tuplas, tales que cada conjunto tiene un valor único para el conjunto de atributos X y la unión de tales conjuntos es la relación r. El *rango* $|\pi|$ de una partición pi es el número de clases de equivalencia que hay en π.

Ejemplo 3.3.1. Consideremos la relación mostrada en la tabla 3.2, el atributo A tiene el valor 1 solo en las tuplas 1 y 2, así que forman una clase de equivalencia $Q_A(1) = Q_A(2) = \{1, 2\}$ (identificamos a la tupla con su identificador, mostrado de la lado izquierdo). Observemos que la partición completa respecto a A es $\pi_{\{A\}} = \{\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}\}$. La partición respecto a $\{B, C\}$ es $\pi_{\{B,C\}} = \{\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6\}, \{7\}, \{8\}\}$.

	A	B	C	D
1	1	a	$	Pato
2	1	A	*	Perro
3	2	A	$	Gato
4	2	A	$	Pato
5	2	b	*	Vaca
6	3	b	$	Pez
7	3	C	*	Pato
8	3	C	#	Caballo

Tabla 3.1: Una relación para mostrar un ejemplo de particiones respecto a sus atributos.

$$
\begin{aligned}
\pi_{\{A\}} &= \{\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\} \\
\pi_{\{B\}} &= \{\{1\}, \{2, 3, 4\}, \{5, 6\}, \{7, 8\}\} \\
\pi_{\{C\}} &= \{\{1, 3, 4, 6\}, \{2, 5, 7\}, \{8\}\} \\
\pi_{\{D\}} &= \{\{1, 4, 7\}, \{2\}, \{3\}, \{5\}, \{6\}, \{8\}\}
\end{aligned}
$$

Tabla 3.2: Particiones de los atributos en la tabla 3.3.1

3.3.2. Algoritmos de retícula

Al igual que el algoritmo de fuerza bruta, en este tipo de algoritmos el espacio de soluciones está dado por el conjunto potencia de todos los atributos en r. Sin embargo, en lugar de explorar de manera exhaustiva dicho espacio, estos algoritmos emplean una retícula donde cada nodo corresponde a un elemento del espacio de búsqueda. Para podar este espacio, la retícula es recorrida de manera ascendente al mismo tiempo que se generan y verifican FDs candidatas.

En la figura 3.1 podemos observar un ejemplo de una retícula en la que se arreglan todos los elementos del conjunto potencia de de cuatro atributos de forma ascendente en la cantidad de atributos, lo cual garantiza que se obtengan solo DFs mínimas. Se poda esta retícula, comenzando por los atributos individuales, verifica nivel a nivel la validez de las dependencias funcionales; en caso de no ser válido, ese elemento y sus descendientes son eliminados.

TANE [7], *FD_mine* [44] [45] y *DFD* [35] son ejemplos de algoritmos de retícula. En la sección 3.4 hablaremos del algoritmo TANE de forma más detallada puesto que es de especial interés para este trabajo.

3.3.3. Algoritmos basados en conjuntos

A diferencia de los algoritmos de retícula donde una DF candidata es verificada usando la información de la relación para verificar su validez, en este tipo de algoritmos cada DF es verificada

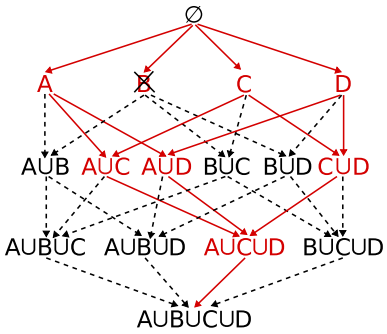

Figura 3.1: Retícula de elementos del conjunto potencia de los atributos {A, B, C, D}, que debido a que se eliminó B los subsecuentes conjuntos que lo contienen no son revisados por el algoritmo.

usando la información contenida en alguno de dos conjuntos conocidos como *conjunto diferencia* y el otro como *conjunto acuerdo*.

Un conjunto acuerdo de dos tuplas t_i y t_j es el conjunto X más grande de atributos para el cual se cumple que $t_i(X) = t_j(X)$ [34]. El conjunto acuerdo de dos tuplas se denota como $ag(t_i, t_j)$, mientras que el conjunto de todos los conjuntos acuerdo en una relación r se denota como $ag(r)$. Para calcular un conjunto acuerdo se utilizan las particiones de t_i y t_j de la manera siguiente: el atributo $A \in X$ si existe algún subconjunto $q \in \pi_A$ tal que $t_i, t_j \in q$. La propiedad más importante de los conjuntos acuerdo es que si $ag(t_i, t_j) = X$, entonces para cualquier atributo $A \in (\mathscr{U} \setminus X)$ se cumple que $t_i(A) \neq t_j(A)$, es decir t_i y t_j impiden que la DF $X \rightarrow A$ sea válida.

El conjunto max de un atributo A, denotado por $max(A)$, contiene conjuntos acuerdo que no incluyen al conjunto A, *i. e.*,

$$max(A) = \{X : X \in ag(r) \ \wedge \ A \notin X \wedge \nexists\, Y \in ag(r), X \subset Y\}$$

Los conjuntos max de cada atributo $A \in \mathscr{U}$ forman una cubierta para la cerradura negativa de la relación, *i. e.*, el conjunto de todas las DF que son violadas por la relación. Una vez que se tienen los conjuntos max de cada atributo en la relación, estos pueden ser utilizados para obtener las DFs que son válidas en r, tal que A se encuentra el lado derecho de la DF, estas dependencias se denotan por $DF(A)$ y se obtienen como sigue:

$$DF_1(A) = \{X \rightarrow A : X \in (\mathscr{U} \setminus A) \wedge \nexists\, Y \in max(A), X \subseteq A\} \tag{3.1}$$

$$DF(A) = \{f : f \in DF_1(A) \wedge \nexists\, g \in DF_1(A), izq(g) \subseteq izq(f)\} \tag{3.2}$$

Observemos que para cualquier $Y \in max(A)$, la DF $Y \rightarrow A$ debe ser violada por al menos un par de tuplas de r. Si algún atributo B es agregado a Y tal que $Y \cup V$ no está $max(A)$, entonces $Y \cup A$ es válida. Ahora, ya que X no es un subconjunto de ningún Y, la DF $X \rightarrow A$ debe ser válida. Las condiciones de la ecuación 3.2 garantizan que $FD(A)$ solo contenga DFs mínimas.

Sea L el conjunto de todos los atributos en $max(A)$, primero es necesario revisar cada conjunto que contiene un solo atributo $B \in L$, si dicho conjunto no se encuentra en $max(A)$, se agrega la DF $B \rightarrow A$ a $DF(A)$. A continuación se revisan combinaciones de dos atributos en el conjunto L, si alguna combinación, *e. g.*, $B \cup C$ no se encuentra en algún subconjunto de $max(A)$ y no contiene el lado izquierdo de ninguna dependencia en $DF(A)$, entonces $\{B \cup C\} \rightarrow A$ es agregado a $DF(A)$. Este proceso continua usando en cada paso combinaciones más grandes de atributos en L.

La complejidad del algoritmo anterior es exponencial en el número de atributos en $\mathscr{U}$, en particular $O(|\mathscr{U}||r|^2+|\mathscr{U}||ag(r)|^2+|\mathscr{U}||max(A)|\frac{|L|}{2}2^{|L|})$. Ejemplo de este tipo de algoritmos son *FUN* y *Dep-Miner* [46].

3.4. TANE

El algoritmo TANE encuentra todas las DFs mínimas y no triviales de una base de datos. Este considera de la coincidencia de un conjunto tuplas en algún conjunto de atributos para calcular las DFs, revisando si estas coinciden tanto del lado derecho como del lado izquierdo y verifica fácilmente si no es una dependencia funcional puesto que no coinciden del lado derecho, reduciendo el conjunto de las tuplas sobre las que se realizará la búsqueda. Esto se puede denotar formalmente con las definiciones de particiones y clases de equivalencia:

$$Q_X(t) = \{\forall A \in X, u, t \in r | t(A) = u(A)\}$$

3.4.1. Refinamiento de las particiones

Una partición π *refina* a otra partición π' si cada clase de equivalencia en π es un subconjunto de alguna clase de equivalencia de π'. El refinamiento de partición da casi directamente dependencias funcionales.

Lema 3.4.1. *Una dependencia funcional $X \to A$ es válida si y solo si π_X refina a $\pi_{\{A\}}$.*

Existe una prueba mucho más simple para saber si $X \to A$ es válida, esto se realiza revisando si $|\pi_X| = |\pi_{X\cup\{A\}}|$. Ya que si π_X refina a $\pi_{\{A\}}$, entonces $\pi_{X\cup\{A\}} = \pi_X$. Además como $\pi_{X\cup\{A\}}$ siempre refina a π_X, $\pi_{X\cup\{A\}}$ no puede tener el mismo número de clases de equivalencia que π_X a menos que $\pi_{X\cup\{A\}}$. De aquí obtenemos el siguiente lema [7].

Lema 3.4.2. *$X \to A$ es válida si y solo si $|\pi_X| = |\pi_{X\cup\{A\}}|$.*

Ejemplo 3.4.3. De la tabla 3.2 se pueden obtener las particiones

$$\pi_{\{A,C\}} = \{\{1\},\{2\},\{3,4\},\{5\},\{6\},\{7\},\{8\}\}$$
$$\pi_{\{A,B,C\}} = \{\{1\},\{2\},\{3,4\},\{5\},\{6\},\{7\},\{8\}\}$$

Observemos que $\pi_{\{A,B,C\}}$ refina a $\pi_{\{A,C\}}$, *i. e.*, $\pi_{\{A,B,C\}} = \pi_{\{A,C\}} \implies |\pi_{\{A,B,C\}}| = |\pi_{\{A,C\}}|$.

3.4.2. Poda a través de conjuntos candidatos

TANE realiza la búsqueda de las dependencias funcionales utilizando la retícula de atributos como se explica en la sección 3.3.2. Verifica las dependencias de un conjunto X de tal forma que no sean consideradas las dependencias triviales, $X\backslash A \to A$ y se reduce el tiempo de cómputo ya que se reutilizan la información obtenida en los niveles anteriores.

Si para algún conjunto X, $A \in \mathcal{C}(X)$ significa que no se ha encontrado que A sea dependiente de algún conjunto propio de X, *i. e.*, el conjunto de *candidatos iniciales del lado derecho* de $X \subseteq R$, $\mathcal{C}(X) = R\backslash\overline{\mathcal{C}(X)}$, donde $\overline{\mathcal{C}(X)} = \{A \in X | r \vDash X\backslash\{A\} \to A\}$. Para verificar que las DFs obtenidas son mínimas solo es necesario saber si existe $X\backslash\{A\} \to A$, $A \in X$ y $A \in \mathcal{C}(X\backslash\{B\}), \forall B \subset X$, *i. e.*, A está en todos los conjuntos candidatos de todos los subconjuntos.

Ejemplo 3.4.4. Sea $R = \{A, B, C\}$ y asuma que de algún paso anterior, sabemos que $C \to A$, por lo que necesitamos verificar las dependencias $\{A, B\} \to C, \{A, C\} \to B$ y $\{B, C\} \to A$.

$$\begin{aligned}\mathcal{C}(\{A, B\}) &= \{A, B, C\}\\ \mathcal{C}(\{A, C\}) &= \{B, C\}\\ \mathcal{C}(\{B, C\}) &= \{A, B.C\}\end{aligned}$$

No hay necesidad de revisar si $\{B, C\} \to A$ puesto que $A \notin \mathcal{C}(\{A, B\}) \cap \mathcal{C}(\{A, C\}) \cap \mathcal{C}(\{B, C\})$.

Ya que $r \vDash C \to A \implies A \notin \mathcal{C}(\{A, C\}) = \mathcal{C}(R \backslash \{B\})$, lo que le hace saber al TANE que $\{B, C\} \to A$ no es una DF mínima.

Ejemplo 3.4.5. Sea $R = \{A, B, C, D\}, \quad dep(r) = \{A \to C, \{C, D\} \to B\}$,

$$\begin{aligned}\mathcal{C}(A) = \mathcal{C}(B) = \mathcal{C}(C) = \mathcal{C}(D) &= \{A, B, C, D\} \backslash \{\}\\ \mathcal{C}(\{A, B\}) &= \{A, B, C, D\} \backslash \{\}\\ \mathcal{C}(\{A, C\}) &= \{A, B, C, D\} \backslash \{C\} = \{A, B, D\}\\ \mathcal{C}(\{C, D\}) &= \{A, B, C, D\} \backslash \{\}\\ \mathcal{C}(\{B, C, D\}) &= \{A, B, C, D\} \backslash \{B\} = \{A, C, D\}\end{aligned}$$

Ya que $r \vDash A \to C \implies C \notin \mathcal{C}(\{A, C\}) = \mathcal{C}(R \backslash \{B, D\})$ y $r \vDash \{C, D\} \to B \implies \mathcal{C}(\{B, C, D\}) = \mathcal{C}(R \backslash \{A\})$ por lo que TANE sabe que tanto $\{A, B, D\} \to C$ como $\{A, C, D\} \to B$ no son DFs mínimas.

La idea de poda del espacio de búsqueda en TANE se basa en que si el conjunto $\mathcal{C}(X) = \emptyset$ entonces para cualquier subconjunto $Y \subset X$ también $\mathcal{C}(Y) = \emptyset$, por lo que no puede haber una dependencia de la forma $Y \backslash A \to A$ que sea mínima lo que significa que el subconjunto Y aún debe ser procesado.

Aunque los candidatos del lado derecho permiten garantizar DFs mínimas, utilizar el conjunto de *candidatos del lado derecho mejorados* $C^+(X) = \{A \in R | \forall B \in X : r \nvDash X\{A, B\} \to B\}$ permite podar el espacio de búsqueda de forma más efectiva. Note que A puede ser igual a B, en tal caso $\mathcal{C}^+(X) \equiv \mathcal{C}(X)$.

Lema 3.4.6. *Sea $A \in X$ y $r \vDash X \backslash \{A\}$. La DF $X \backslash \{A\}$ es mínima si y solo si $\forall B \in X, A \in \mathcal{C}^+(X \backslash \{B\})$.*

Es posible remplazar $\mathcal{C}^+(X \backslash \{B\})$ por $\mathcal{C}(X \backslash \{B\})$ y el lema 3.4.2 seguirá siendo válido, sin embargo, usar al conjunto $C^+(X \backslash \{B\})$ tiene dos ventajas. La primera, es que podríamos encontrar una B para la cual $A \notin \mathcal{C}^+(X \backslash \{B\})$ y el algoritmo detendría la verificación antes, ahorrando tiempo. La segunda, es que puede haber alguna B para que que $A \in \mathcal{C}^+(X \backslash \{B\})$ esté vacío y $A \in \mathcal{C}(X \backslash \{B\})$ no. De esta forma no tenemos que procesar al conjunto X.

Lema 3.4.7. *Sea $B \in X$ y $r \vDash X \backslash \{B\}$. Si $r \vDash X \backslash A \implies r \vDash X \backslash \{B\} \to A$.*

Esto nos permite remover atributos adicionales del conjunto de candidatos iniciales del lado derecho $\mathcal{C}(X)$. Puesto que si $\exists B \in X | r \vDash X \backslash \{B\}$, por el lema 3.4.2, una dependencia que tenga a X del lado izquierdo no puede ser mínima ya que B puede ser removido sin alterar la validez de la DF.

Ejemplo 3.4.8. Sea $r \vDash A \to B$, si $r \vDash \{A, B\} \to C$ entonces $r \vDash A \to C$.

Lo que nos permite remover de $\mathcal{C}(X)$ el siguiente conjunto:

$$\overline{C'(X)} = \begin{cases} R \setminus X & \text{si } \exists B \in X : r \models X\backslash\{B\} \to B \\ \emptyset & \text{en otro caso} \end{cases}$$

Ejemplo 3.4.9. Sea $X = \{A, B, C\}$ y $r \vDash C \to B$.
Entonces $A \in \mathcal{C}'(B, C)$ lo que le da información al TANE para saber que $X\backslash\{A\} \to A$ no es mínima y no necesita saber si es válida o no.

La idea anterior aplica también para subconjuntos, por lo que también podemos remover:

$$\overline{\mathcal{C}''(X)} = \{A \in X | \exists B \in X\backslash\{A\} : r \vDash X\backslash\{A, B\} \to B\}$$

Entonces la definición de $\mathcal{C}^+(X)$ remueve tres tipos de candidatos:

$$\begin{aligned} \mathcal{C}(X) &= \{A \in X | r \vDash X\backslash\{A\} \to A\} \\ \mathcal{C}'(X) &= \{R\backslash X\} \text{ si } \exists B \in X : r \vDash X\backslash\{B\} \to B \\ \mathcal{C}''(X) &= \{A \in X | \exists B \in X\backslash\{A\} : r \vDash X\backslash\{A, B\} \to B\} \end{aligned}$$

Podemos definir el conjunto de los candidatos del lado derecho mejorados en términos de $\mathcal{C}(X)$, $\mathcal{C}'(X)$ y $\mathcal{C}''(X)$.

Lema 3.4.10. $\mathcal{C}^+(X) = ((R\backslash\overline{\mathcal{C}(X)})\backslash\overline{\mathcal{C}'(X)}\backslash\overline{\mathcal{C}''(X)}$

3.4.3. Poda a través de llaves

Los conceptos de llave y superllave también pueden ser utilizados para podar el espacio de búsqueda. Decimos que un conjunto de atributos X es una superllave si no existen dos tuplas con la misma clase de equivalencia, *i. e.*, π_X contiene solo conjuntos de un solo elemento. Además X será una llave, si es una superllave y no existe un subconjunto propio de X que sea una superllave. De acuerdo al siguiente lema, si se encuentra una llave durante la búsqueda es posible realizar una poda adicional.

Lema 3.4.11. *Sea $B \in X$ y sea $r \vDash X\backslash\{B\} \to B$. Si X es una superllave, entonces $X\backslash\{B\}$ es una superllave.*

Generalmente para revisar si una dependencia $X \to A$ es válida se necesita el conjunto $X \cup \{A\}$, lo anterior debido a que se requiere $\pi_{X\cup\{A\}}$ para verificar la dependencia. Sin embargo, cuando X es una superllave $X \to A$ siempre es válida. En consecuencia el conjunto $X \cup \{A\}$ no es necesario. Cuando X es una superllave, pero no una llave una DF como $X \to A$ no es mínima para ningún $A \notin X$. Si $A \in X$ y $X\backslash\{A\} \to A$ es válida, por el lema 3.4.3, $X\backslash\{A\}$ es una superllave. Por lo tanto, no se necesita π_X para determinar la validez de la DF $X\backslash\{A\} \to A$. Por lo tanto es posible podar todas los conjuntos de atributos X que sean llaves y todas sus superllaves.

3.4.4. Algoritmos

Como se dijo anteriormente, TANE es un algoritmo de retícula que explora la misma nivel a nivel de manera ascendente, *i. e.*, se comienza con conjuntos de un solo atributo y se procede con conjuntos cada vez más grandes. El algoritmo 3.2 detalla este proceso. Cada nivel de la retícula es

representado mediante un conjunto L_l de tamaño l. El algoritmo comienza con cada conjunto $L_1 = \{\{A\} : A \in R\}$ y de acuerdo a las DFs halladas calculará los conjuntos $L_2, L_3, \ldots$. En cada nivel el algoritmo CalcularDependencias determina las DFs mínimas y validas tales que L_{l-1} aparece en el lado izquierdo de la dependencia. Por otro lado el algoritmo Podar se encarga de la poda de la retícula eliminando conjuntos de cada nivel L_l. Finalmente el algoritmo GenerarSiguienteNivel se encarga de construir el siguiente nivel L_{l+1} de la retícula.

Algoritmo 3.2 Algoritmo TANE.

Entrada: Una relación r sobre R.
Salida: Las DFs exactas mínimas y no triviales que son válidas en r.

1: $L_0 \leftarrow \{\emptyset\}$
2: $\mathcal{C}^+(\emptyset) \leftarrow R$
3: $L_1 \leftarrow \{\{A\} : A \in R\}$
4: $l \leftarrow 1$
5: **mientras** $L_l \neq \emptyset$ **hacer**
6: CalcularDependencias(L_l)
7: Podar(L_l)
8: $L_{l+1} \leftarrow$ GenerarSiguienteNivel(L_l)
9: $l \leftarrow l + 1$
10: **fin mientras**

El algoritmo 3.3 se encarga de calcular las DFs mínimas en cada nivel de la retícula de acuerdo al lema 3.4.2. Para realizar lo anterior, este algoritmo calcula los conjuntos $\mathcal{C}^+(X)$ para cada conjunto de atributos X en el nivel L_l.

Algoritmo 3.3 Algoritmo para calcular dependencias.

1: **procedimiento** CalcularDependencias(L_l)
2: **para cada** $X \in L_l$ **hacer**
3: $\mathcal{C}^+(X) \leftarrow \bigcap_{A \in X} \mathcal{C}^+(X \backslash \{A\})$
4: **para cada** $A \in X \cup (C)^+(X)$ **hacer**
5: **si** $X \backslash \{A\} \rightarrow A$ es válida **entonces**
6: Eliminar A de $\mathcal{C}^+(X)$
7: Eliminar toda B en $R \backslash X$ de $\mathcal{C}^+(X)$
8: **regresar** $X \backslash \{A\} \rightarrow A$
9: **fin si**
10: **fin para**
11: **fin para**
12: **fin procedimiento**

Para podar la retícula se siguen la estrategias descritas en las secciones 3.4.2 y 3.4.3. Si $\mathcal{C}^+(X) = \emptyset$, entonces el conjunto de atributos X es eliminado del nivel L_l. Después, por el lema 3.4.3, si el conjunto X es una llave o superllave entonces también es eliminado del nivel L_l.

Para general el nivel L_{l+1} de la retícula, el algoritmo 3.5 debe particionar el nivel L_l en bloques disjuntos. Dicho proceso se lleva a cabo en el procedimiento Prefijos de la manera siguiente: considere al conjunto $X \in L_l$ como un conjunto ordenado de atributos. Diremos que dos conjuntos $X, Y \in L_l$ pertenecen al mismo bloque si tiene un prefijo común de longitud $l - 1$, *i. e.*, difieren a

Algoritmo 3.4 Algoritmo de poda.

```
1: procedimiento PODA(L_l)
2:     para cada X ∈ L_l hacer
3:         si C⁺(X) = ∅ entonces
4:             Eliminar X de L_l
5:         fin si
6:         si X es una llave o superllave entonces
7:             para A ∈ C⁺(X)\X hacer
8:                 si A ∈ ⋃_{B∈X} C⁺(X ∪ {A}\{B}) entonces
9:                     regresar X → A
10:                fin si
11:            fin para
12:            Eliminar X de L_l
13:        fin si
14:    fin para
15: fin procedimiento
```

lo más en el último atributo. Cada prefijo forma un bloque consecutivo en orden lexicográfico en L_l. De esta manera es fácil calcular los bloques que forman L_{l+1} a partir de L_l.

Algoritmo 3.5 Algoritmo para generar cada nivel de la lattice.

```
1: procedimiento GENERARSIGUIENTENIVEL(L_l)
2:     L_{l+1} ← ∅
3:     para cada K ∈ PREFIJOS(L_l) hacer
4:         para cada {Y, Z} ⊆ K, Y ≠ Z hacer
5:             X ← Y ∪ Z
6:             si ∀A ∈ X, X\{A} ∈ L_l entonces
7:                 L_{l+1} ← L_{l+1} ∪ {X}
8:             fin si
9:         fin para
10:    fin para
11: fin procedimiento
```

Parte II

Metodología y experimentación

Capítulo 4

Metodología

4.1. Introducción

En el capítulo 3 presentamos el problema de la inferencia de dependencias funcionales, así como los algoritmos que resuelven dicho problema. En el presente capítulo retomamos dichas ideas, así como lo expuesto en el capítulo 2 acerca de limpieza de datos y funciones de similitud, para presentar la aportación principal de este trabajo de tesis, *i. e.*, una metodología para la inferencia de DFs mediante funciones de similitud. Además de presentar nuestro proceso de limpieza de datos, este capítulo describe de manera detallada la extensión realizada al algoritmo TANE que nos permite inferir DFs a partir de funciones de similitud. Cómo se expuso en la sección 3.4, este algoritmo infiere DFs haciendo uso de particiones y clases de equivalencia. Esta característica hace de TANE uno de los algoritmos que además de calcular DFs exactas también calcula DFs aproximadas (ver sección 2.7.2). Debido a lo anterior elegimos a TANE como punto de partida para desarrollar un algoritmo para la inferencia de DFs con funciones de similitud.

4.2. Limpieza de la base de datos

Antes de poder realizar la inferencia de dependencias funcionales en una base de datos la información contenida en la misma debe estar libre de cualquier anomalía.

Como caso de estudio, esta sección se centra en el análisis de una base de datos, que de aquí en adelante será conocida como CRÍMENES. Esta contiene información acerca de crímenes ocurridos en el municipio de Nezahualcóyotl, México entre los años de 2016 y 2017. Esta base de datos comprende 13 atributos ($n = 13$) y 16425 tuplas ($|r| = 16425$). Cada crimen es descrito a través de de los siguientes atributos:

- ID. Un entero que identifica cada tupla en la base de datos.
- HORA. Determina el intervalo de tiempo en el que fue cometido un delito.
- FECHA. Contiene la fecha en la que fue cometido el delito.
- MES. Determina el mes en el que fue cometido un crimen.
- DELITO. Describe el tipo de delito que fue cometido.
- CALLE. Es el nombre de la calle donde se cometió el delito.
- NO . . Determina el número en la calle donde fue cometido el delito.
- ENTRE_CALLE_1 y ENTRE_CALLE_2. Los nombres de las calles entre las cuales fue cometido el delito.
- COLONIA. La colonia donde fue cometido el delito.

- CUADRANTE. Un cuadrante es un territorio al que se le asigna un estado de fuerza de personal (policías), vehículos (patrullas) y equipo de comunicación (radios y teléfonos celulares). Cada cuadrante es representado con una cadena de texto.
- SECTOR. Un sector es un conjunto de cuadrantes, cada sector es representado por el caracter C seguido de un entero.
- TURNO. Cada cuadrante cuenta con un encargado por turno (jefe de cuadrante), el cual es el responsable en alguno de los tres turnos posibles.
- X y Y. La latitud y longitud donde se cometió el delito.

La mayoría de la información que conforma una base de datos sociodemográfica proviene de estudios de campo que reúnen información acerca de áreas urbanas. Ya que comúnmente los datos son recolectados manualmente, son propensos a anomalías como errores léxicos, datos faltantes, y errores de formato. Cada una de estas anomalías hace que la inferencia de dependencias funcionales sea una tarea complicada.

Las secciones siguientes detallan el proceso de limpieza de datos que se llevo a cabo para eliminar las anomalías en la base de datos CRÍMENES.

4.2.1. Datos faltantes

Una de las anomalías más comunes son los datos faltantes. En el caso de las bases de datos sociodemográficas no siempre se es posible recolectar los datos correspondientes a algún indicador urbano. Por ejemplo, no siempre es posible conocer la calle y número en la que fue cometido un delito. Para eliminar esta anomalía de la base de datos CRÍMENES se inserto la cadena SIN_ATRIBUTO, donde ATRIBUTO es el nombre del atributo correspondiente, *e. g.*, en el caso de una tupla que no tiene un valor para el atributo NO. se insertaría la cadena SIN_NO.

4.2.2. Errores léxicos

Como ejemplo de este tipo de errores consideremos el atributo NO. que representa la dirección en particular donde ocurrió un crimen. Un ejemplo de un valor asociado a este atributo es la cadena AV, PANTITLAN. Dado que nuestras bases de datos están almacenadas en formato CSV, no se permite que un valor contenga el caracter coma.

Para corregir esta anomalía simplemente se realiza un reemplazo de cada coma en los valores de los atributos por un caracter adecuado, por ejemplo, un punto. De esta manera la cadena AV, PANTITLAN es reemplazada por la cadena AV. PANTITLAN.

4.2.3. Errores de formato

Este tipo de anomalías están caracterizadas por valores que no cumplen con el formato asociado a un atributo. Consideremos las tuplas y sus valores correspondientes mostradas en la tabla 4.1, en esta tabla se observa que los valores asociados al atributo FECHA están especificados en diferentes formatos, esta diferencia hace imposible inferir alguna dependencia funcional con el atributo FECHA. Para resolver este problema utilizamos una expresión regular apropiada para cada uno de los formatos. Con dicha expresión regular es posible extraer la información de cada

campo que compone la fecha y finalmente se produce el formato unificado DIA/MES/AÑO para todas las tuplas.

Tupla	FECHA	HORA
2	1/1/2016	6:40
3519	14/05/16	17:00 A 20:15
15130	1-Jul	14:00

Tabla 4.1: Errores de formato.

En el caso del atributo HORA, la tabla 4.1 muestra que algunas tuplas contienen un solo valor que representa la hora en la que fue cometido un delito. Sin embargo, la entrada 3519 contiene dos horas que representan el intervalo de tiempo durante el cual ocurrió el delito. Para corregir esta anomalía optamos por separar el atributo HORA en dos nuevos atributos HORA_INICIO y HORA_FIN. Estos atributos representan la hora en la que inicia y concluye un crimen respectivamente. En el caso de aquellas tuplas que solo tienen un solo valor para el atributo HORA los valores para los atributos HORA_INICIO y HORA_FIN serán los mismos. Esta transformación tiene la ventaja de que es posible operar con los valores de HORA_INICIO y HORA_FIN para obtener una estimación del tiempo de duración de un delito. La tabla 4.2 muestra los resultados de estas transformaciones.

Tupla	FECHA	HORA_INICIO	HORA_FIN
2	01/01/2016	6:40	6:40
3519	14/05/2016	17:00	20:15
15130	01/07/2017	14:00	14:00

Tabla 4.2: Errores de formato corregidos.

4.2.4. Manejo de información redundante

Aunque no se considera como tal una anomalía, la información redundante complica la inferencia de dependencias funcionales. Como se mencionó en la sección 3.3 los algoritmos de retícula como TANE son sensibles al número de atributos en una base de datos, por lo tanto resulta apropiado eliminar aquellos atributos que contienen información innecesaria. En el caso de la base de datos CRÍMENES el atributo MES contiene información que puede ser deducida a partir del atributo FECHA. Por lo tanto, eliminamos dicho atributo de la base de datos.

Además de MES, el atributo ID no proporciona información relevante acerca de la ocurrencia de un crimen. Es importante notar que dado que cada tupla tiene un valor distinto para este atributo, al realizar la inferencia de dependencias funcionales se tendrán múltiples del tipo ID $\rightarrow$ ATRIBUTO, este tipo de dependencias describen una relación obvia entre ID y cada uno de los atributos restantes, en consecuencia decidimos eliminar este atributo de la base de datos.

4.2.5. Concatenación de la latitud y longitud

Los atributos X y Y determinan la latitud y longitud de la ubicación donde fue cometido un crimen. Ya que es de nuestro interés poder determinar relaciones a partir de funciones de similitud, se decidió por crear un nuevo atributo llamada COORDENADA a partir de los atributos X y Y. De esta manera es posible utilizar la distancia de Minkowsky en dos dimensiones para determinar cuando dos tuplas del atributo COORDENADA son similares dado el parámetro δ.

4.3. Cálculo de clases de equivalencia mediante funciones de similitud

Sea $\mathscr{U}$ un conjunto de atributos, y r una relación sobre $\mathscr{U}$. Para cada $X \in \mathscr{U}$, con dominio $dom(X)$, consideremos una función de similitud s_X dada por:

$$s_X : dom(X) \times dom(X) \to \mathbb{R}^+$$

Como es usual, la función s_X satisface las siguiente propiedades:

- No negatividad, $s_X(x, y) \geq 0$.
- Identidad de los indiscernibles, $s_x(x, y) = 0 \iff x = y$.
- Simetría $s_X(x, y) = s_X(y, x)$.

El objetivo de la función s_X es generalizar una relación de equivalencia, *e. g.*, si $s(x, y) = 0 \implies x = y$. A través de esta generalización, es posible extender la definición de una DF exacta como sigue: una DF es una expresión de la forma $X \to A$, donde $X \subseteq \mathscr{U}$ y $A \in \mathscr{U}$. La DF será válida en r si para cada par de tuplas t_i y t_j en r se tiene que:

$$Q_B(t_i) = Q_B(t_j), \forall B \in X \implies Q_A(t_i) = Q_A(t_j) \tag{4.1}$$

donde $Q_B(t_i)$ denota la clase de equivalencia de la tupla t_i con respecto del atributo B. $Q_B(t_i)$ está dada por:

$$Q_B(t_i) = \{t_j \in r : s_B(t_i(B), t_j(B)) \leq \delta\} \tag{4.2}$$

Es decir, t_i y t_j pertenecen a la misma clase de equivalencia del atributo B si sus valores para este atributo son lo suficientemente similares bajo la función de similitud s_B y el parámetro δ. De acuerdo a lo anterior, un atributo B particiona a la relación r en un conjunto de clases de equivalencia, *i. e.*:

$$\pi_B = \{Q_B(t_i) : t_i \in r\}$$

Al igual que en el caso de una relación de equivalencia, diremos que una partición π refina a otra partición π' si cada clase de equivalencia en π es un subconjunto de alguna clase de equivalencia de π'.

Para ejemplificar lo anterior consideremos los datos y clases de equivalencia correspondientes mostrados en la tabla 4.3. Es fácil observar que en dicha tabla una DF exacta como C $\to$ D debe ser válida. Si se analizan las clases de equivalencia de los atributos C y D se observa que ambas cumplen, aunque trivialmente, con lo establecido por la ecuación 4.1, en consecuencia la DF es

A	B	C	D	π_A	π_B	π_C	π_D
DEW	22:30	1/1/2016	-99.051253 19.488201	0	0	0	0
RKP	0:05	1/2/2016	-99.033186 19.405685	1	1	1	1
EVR	0:00	1/10/2016	-99.037534 19.401316	2	2	2	2
DEW	21:30	11/3/2016	-99.024387 19.408049	0	3	3	3

Tabla 4.3: Datos de prueba y clases de equivalencia.

válida. De la tabla 4.3 también se observa que la DF exacta A $\rightarrow$ B no es válida en la relación, esto debido a que las particiones π_A y π_B no cumplen con lo establecido por la ecuación 4.1.

Si ahora se calculan las clases de equivalencia de π_B usando como función de similitud la diferencia en minutos entre dos horas, y un parámetro $\delta = 60$ se obtienen las clases de equivalencia mostradas en la tabla 4.4. Ya que la diferencia en minutos en los valores para el de las tuplas 0 y 3 para el atributo B es menor o igual a 60 ambas tuplas pertenecerán a la misma clase de equivalencia. Usando un argumento similar es fácil ver que las tuplas 1 y 2 también pertenecen a la misma clase de equivalencia. Por lo tanto, la DF A $\rightarrow$ B es ahora válida en la relación.

Tupla	π_A	π_B	π_C	π_D
0	0	0	0	0
1	1	1	1	1
2	2	1	2	2
3	0	0	3	3

Tabla 4.4: Datos de prueba y clases de equivalencia.

Cómo se muestra en [47] es posible utilizar el los conceptos de clase de equivalencia y partición para verificar si una DF es válida en una relación a través del siguiente lema:

Lema 4.3.1. *Una dependencia funcional $X \rightarrow A$ es válida en una relación r si y solamente si π_X refina π_A.*

Demostración. Si $X \rightarrow A$ es válida en r, entonces para cada par de tuplas t_i y t_j se cumple lo expresado por la ecuación 4.1, *i. e.*, si las tuplas t_i y t_j tiene las mismas clases de equivalencia con respecto de cada atributo $B \in X$ entonces también tendrán la misma clase de equivalencia con respecto del atributo A. Debido a lo anterior, tuplas que estén en el mismo conjunto en el lado izquierdo de la DF también estarán en el mismo conjunto del lado derecho. En el peor de los casos el conjunto del lado derecho tendrá una cardinalidad mayor y se cumplirá que una clase de equivalencia en el lado izquierdo será subconjunto de alguna clase de equivalencia del lado derecho. Por lo tanto π_X refina a π_A.

Por otro lado si π_X refina a π_A entonces cada clase de equivalencia con respecto de X es un subconjunto de una clase de equivalencia con respecto de A. En consecuencia cualesquiera dos tuplas que tengan la misma clase de equivalencia con respecto de X tendrán la misma clase de equivalencia con respecto de A, por lo tanto de acuerdo a la ecuación 4.1 la DF será válida. ■

A partir del lema anterior se deriva el siguiente resultado que da una manera simple de verificar si una DF es válida.

Lema 4.3.2. *Una dependencia funcional $X \rightarrow A$ es válida en una relación r si y solamente si $|\pi_X| = |\pi_{X \cup A}|$*

Demostración. Si la DF $X \rightarrow A$ es válida en r, entonces por el lema 4.3, π_X refina a π_A. Observemos que si se toma $X \cup A$ entonces las clases de equivalencia en $\pi_{X \cup A}$ son idénticas a las clases de equivalencia en π_X, *i. e.*, agregar A no puede generar clases de equivalencia adicionales, de ahí se sigue que $|\pi_X| = |\pi_{X \cup A}|$.

Por otra parte, observemos que dado que $\pi_{X \cup A}$ siempre refina a π_X la única manera en la que $|\pi_X| = |\pi_{X \cup A}|$ es que ambas particiones tengas las mismas clases de equivalencia, *i. e.*, π_X refina a π_A y por lo tanto la DF $X \rightarrow A$ es válida en r. ■

4.4. Algoritmo para el cálculo de clases de equivalencia usando funciones de similitud

De acuerdo a la ecuación 4.2, una clase de equivalencia con respecto a un atributo B consiste de todas aquellas tuplas que son similares de acuerdo a una función s_B y un parámetro δ. Una manera de obtener la clase de equivalencia de una tupla t_i con respecto del atributo B es simplemente asignar la clase de equivalencia de la primer tupla para la cual se cumple que $s_B(t_i(B), t_j(B)) \leq \delta$. Si no se encuentra una tupla tal que dicha condición sea satisfecha entonces simplemente se asigna una nueva clase de equivalencia a dicha tupla. Lo anterior puede realizarse en tiempo $O(|r|)$ (exceptuando el tiempo que toma evaluar s_B). Sin embargo, esta aproximación tiene la desventaja de no garantizar que se cumpla lo establecido por la ecuación 4.2. Por ejemplo, consideremos una tupla t_1 la cual pertenece a una clase de equivalencia q con respecto de un atributo B, y una tupla t_2 tal que $s_B(t_1(B), t_2(B)) \leq \delta$ por lo que $Q_B(t_2) = q$. Sea ahora una tupla t_3 con $s_B(t_1(B), t_3(B)) \leq \delta$, por lo tanto $Q_B(t_3) = q$. Sin embargo, no hay manera de asegurar que estas tres tuplas sean de hecho similares, esto debido a que puede ocurrir que $s_B(t_2(B), t_3(B)) > \delta$.

Una manera de corregir este problema es considerar cada tupla como un punto en el espacio métrico correspondiente, y por lo tanto una clase de equivalencia estará dada por aquellas tuplas que se encuentren dentro de una bola de diámetro δ (ver figura 4.1).

Para lograr lo anterior se pueden utilizar algunas de las tuplas como puntos de referencia (centros) con respecto de los cuales se decide a que clase de equivalencia pertenece alguna tupla en particular, usando la función s correspondiente. Si una tupla no es similar a ninguna de las clases de equivalencia existentes, *i. e.*, la distancia que la separa es mayor que δ, entonces dicha tupla será utilizada como centro de una nueva clase de equivalencia. Este proceso se repite hasta que todas las tuplas de un atributo hayan sido asignadas a alguna clase de equivalencia. Para llevar a cabo lo anterior, el algoritmo 4.1 utiliza dos conjuntos L y Q que inicialmente están vacíos. El conjunto L contiene las tuplas que serán utilizadas como referencia para agrupar todas las tuplas en clases de equivalencia. El conjunto Q contendrá la clase de equivalencia correspondiente para cada tupla con respecto de un atributo B, es importante notar que en el algoritmo cada clase de equivalencia es identificada con un entero no negativo. Si la función s_B toma un valor menor o igual al del parámetro δ entonces una tupla t tiene la misma clase de equivalencia que la tupla u usada como referencia. De lo contrario, la tupla t se usará como referencia para definir una nueva clase de equivalencia y es por lo tanto agregada al conjunto L.

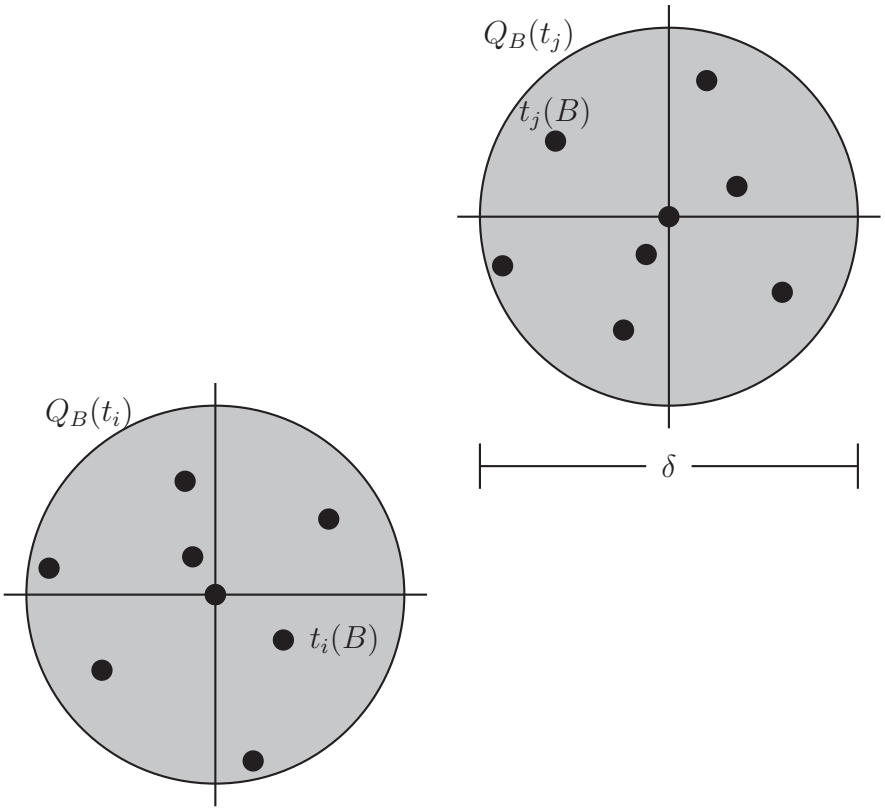

Figura 4.1: Clases de equivalencia.

La complejidad del algoritmo 4.1 es como sigue: encontrar la clase de equivalencia a la que pertenece una tupla en particular en tiempo $O(|L|)$, es importante resaltar que el algoritmo no asume a priori el número de clases de equivalencia que existen para un atributo B, dicho número depende enteramente del valor de cada tupla y del parámetro δ. Claramente encontrar la clase de equivalencia tomará tiempo $O(|r||L|)$ en el peor de los casos.

Algoritmo 4.1 Algoritmo para calcular clases de equivalencia.

ENTRADA: Una relación r, un atributo B y el parámetro δ.
SALIDA: Las clases de equivalencia con respecto del atributo B.

```
1:  procedimiento CalcularClasesDeEquivalencia(r)
2:      Q ← ∅
3:      L ← ∅
4:      para t ∈ r hacer
5:          si Q = ∅ entonces
6:              Agregar t a L
7:              Agregar 0 a Q
8:              romper
9:          en otro caso
10:             e ← −1
11:             q ← 0
12:             m ← ∞
13:             para u ∈ L hacer
14:                 d ← s_B(t(B), u(B))
15:                 si d ≤ δ entonces
16:                     si d < m entonces
17:                         m ← d
18:                         e ← q
19:                     fin si
20:                 fin si
21:                 q ← q + 1
22:             fin para
23:             si e = −1 entonces
24:                 Agregar t a L
25:                 Agregar q a Q
26:             en otro caso
27:                 Agregar e a Q
28:             fin si
29:         fin si
30:     fin para
31: fin procedimiento
```

Capítulo 5

Experimentos y resultados

5.1. Introducción

En esta sección se reportan los resultados de la inferencia de dependencias funcionales a través de la metodología descrita en el capítulo 4. Todos los experimentos reportados en este capítulo fueron realizados usando una implementación del algoritmo correspondiente en el lenguaje de programación C++, en una computadora de escritorio con un procesador a 2.0 GHz y 8 GB de RAM.

5.2. Bases de datos de prueba

Para validar los resultados presentados en esta tesis, además del análisis de la base de datos de CRÍMENES, se realizó el análisis de varias bases de datos conocidas en la literatura. Algunas de estas bases de datos tienen características que son deseables cuando se evalúa un algoritmo para la inferencia de FDs, *e. g.*, un número grande de atributos o un número grande de tuplas. En otros casos estás bases de datos corresponden a casos de uso reales, por lo cuales pueden ser utilizadas para evaluar la correctitud y desempeño en condiciones que no necesariamente son ideales, por ejemplo los datos pueden contener anomalías. Finalmente, el uso de estas bases de datos permite realizar un análisis comparativo con algoritmos desarrollados anteriormente para la inferencia de FDs exactas y aquellas basadas en funciones de similitud.

Las bases de datos usadas en nuestros experimentos son las siguientes:

- CRÍMENES. Esta base de datos contiene información acerca de los crímenes ocurridos en la municipalidad de Nezahualcóyotl, México entre los años 2016 y 2017. La ocurrencia de cada crimen es descrita a través de atributos como: `Fecha`, `Tipo`, `Colonia`, etc.

- CHESS[24]. Los registros de esta base de datos representan los movimientos en un tablero de ajedrez que realizan un rey y una torre blancos para vencer a un rey negro. Para describir dichos movimientos se utilizan 7 atributos que representan las posiciones de cada pieza en el tablero de ajedrez.

- ABALONE [24]. La información de esta base de datos tiene el objetivo de predecir la edad del molusco *abalone* a partir de medir las dimensiones y sexo de varios especímenes. Para lograr lo anterior se utilizan atributos como `Sexo`, `Diámetro`, `Peso`, etc.

- NURSERY[24]. La información de esta base de datos fue utilizada para clasificar las solicitudes de varios padres para ingresar a sus hijos a una guardería. Cada solicitud es caracterizada mediante atributos como `Empleo`, `Estado social`, `Estado financiero`, etc.

- IRIS [24]. Contiene datos taxonómicos acerca de tres especies distintas de la planta iris. Los atributos representan las siguientes características de la planta: ancho del pétalo (AP), lon-

gitud del pétalo (LP), longitud del sépalo (LS), ancho del sépalo (AS), así como la especie de la planta (`Clase`).

- CORA [48] (Disponible en [49]). Cada registro en esta base de datos contiene información de citas de artículos científicos obtenidas a partir del motor de búsqueda *CORA*. Para describir cada registro se utilizan atributos como `Autor`, `Título`, etc.

- RESTAURANT [50] (Disponible en [49]). Esta base de datos contiene 864 registros de la guía de restaurantes *Fodor and Zagat*. Cada registro contiene atributos que describen un restaurante, *e. g.*, `Nombre`, `Dirección`, `Teléfono`, `Tipo`, etc.

5.3. Pruebas de correctitud

Con el objetivo de corroborar que la implementación del algoritmo TANE utilizada en esta tesis sea correcta, se realizaron experimentos para inferir todas las FDs exactas mínimas y no triviales de las bases de datos descritas en la sección 5.2. Para obtener una estimación precisa del tiempo de ejecución del algoritmo se efectuaron 100 repeticiones con cada base de datos y el tiempo total fue promediado. Para medir el tiempo de cada ejecución se utilizo el comando `time` de *UNIX*. La tabla 5.1 muestra los resultados correspondientes, en esta tabla N representa el número de FDs exactas que fueron encontradas para cada base de datos analizada.

Base de datos	n	$\|r\|$	N	Tiempo (s)	Memoria
ABALONE	9	4177	137	019446	27.2 MB
CHESS	7	28056	1	0.5353	60.8 MB
CORA	14	1295	40	2.9131	182.3 MB
CRÍMENES	13	16425	163	42.2969	136.8 MB
IRIS	5	150	4	0.03656	0.8 MB
NURSERY	9	12960	1	0.88912	90.8 MB
RESTAURANT	7	864	16	0.0523	6.5 MB

Tabla 5.1: Resultados de las experimentos de inferencia de FDs exactas.

Los resultados de nuestros experimentos son consistentes con lo reportado en trabajos anteriores [51] [52], *i. e.*, las FDs exactas de las bases de datos ABALONE, CHESS, IRIS y NURSERY son inferidas correctamente. En el caso de las bases de datos RESTAURANT y CORA, a pesar de haber realizado una exhaustiva búsqueda, no pudimos encontrar referencia alguna acerca del número de FDs exactas de estas bases de datos. No obstante, los resultados obtenidos en nuestros experimentos son reportados en el caso de que sean de utilidad para futuros trabajos.

También es interesante notar que de todas las bases de datos analizadas, son CRÍMENES y CORA las bases de datos que mayor tiempo requieren para que sus FDs sean inferidas. Cómo se explicó en el capítulo 3, los algoritmos de retícula son sensibles al número de atributos en una relación, en consecuencia es de esperar que siendo estas bases de datos las que tienen el mayor número de atributos se necesite mayor tiempo de procesamiento para obtener el resultado deseado.

5.4. Descubrimiento automático de conocimiento con funciones de similitud

Como ejemplo de este tipo de aplicaciones consideremos la base de datos IRIS, como se menciono en la sección 5.2 esta contiene información acerca de las características físicas de varios individuos de la especie iris. Como se mencionó en la sección 2.8.2 esta base de datos contiene cuatro FDs exactas que indican que es necesario conocer al menos tres características de una planta para poder determinar la especie (*Setosa*, *Versicolor* o *Virginicca*) a la que pertenece un individuo en particular. Sin embargo, cuando se analiza está información usando métodos de agrupación (ver figura 5.1) es posible observar que es suficiente usar los atributos AP o LP para determinar la especie a la que pertenece una planta. La figura 5.1 también muestra que algunos de los individuos más grandes de la especie *Versicolor* tienen dimensiones similares a los individuos más pequeños de la especie *Virginica*. Debido a esta similitud, resulta imposible establecer una relación a través de una FD exacta.

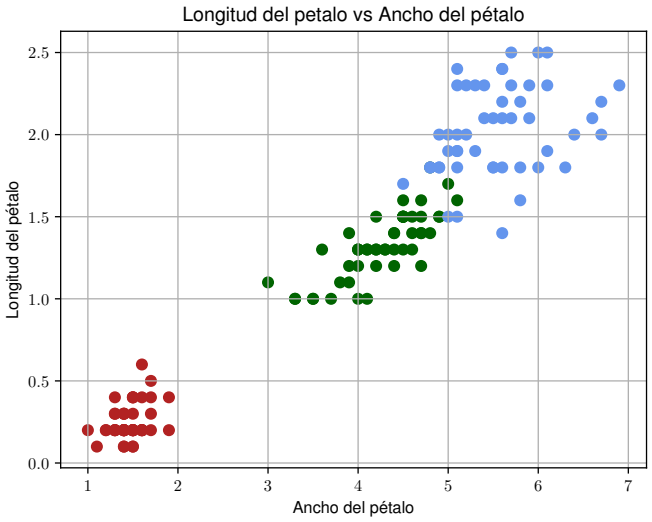

Figura 5.1: Información de la base de datos IRIS después de ser agrupada.

Para clarificar esta idea, consideremos la tabla 2.2 la cual muestra algunos de los datos contenidos en esta base de datos. Es fácil ver que en esta tabla la tupla $(4.8, 1.8)$ en las filas 5 y 7 del atributo `Ancho del pétalo` evita que la siguiente dependencia funcional sea válida:

$$(\texttt{LP},\ \texttt{AP}) \rightarrow \texttt{Clase}$$

Para encontrar esta relación es necesario que aquellas tuplas con una longitud o ancho del pétalo similares pertenezcan a la misma clase de equivalencia. Para lograr esto es necesario elegir una función de similitud y un valor de δ adecuados de acuerdo a la naturaleza del atributo así como el dominio de los valores correspondientes. Tomando en cuenta estas consideraciones realizamos experimentos adicionales con las siguientes características:

- Dado que los atributos LP y AP son numéricos, se eligió la distancia de Minkowski como función de similitud.

- Los valores de δ utilizados fueron determinados a partir de la información contenida en la base de datos, i.e., del dominio de cada atributo.
- Para determinar la clase de equivalencia de una tupla se utilizó la heurística descrita en la sección 4.4, es decir a cada tupla se le asigna la clase de equivalencia de la primer tupla para la cual se cumple que la distancia entre ambas es menor que δ.

Los resultados de estos experimentos se resumen en la tabla 5.2, por claridad se han omitido aquellos resultados que consisten en DFs idénticas a una dependencia funcional exacta. Como puede observarse la DF buscada que relaciona la longitud y el ancho del pétalo con la especie a la que pertenece un individuo es correctamente inferida usando un valor de $\delta = 0.25$ para la longitud del pétalo. Note, sin embargo, que para este valor de δ solo se infieren tres DFs, lo anterior se debe a que al agrupar ciertas tuplas con valores similares se reduce el número total de clases de equivalencia para cada atributo.

Atributo(s)	δ	DF(s)
LP	0.25	$\{\texttt{LP}, \texttt{AP}\} \rightarrow \texttt{Clase}$
LS	1.90	$\{\texttt{AS}, \texttt{LP}\} \rightarrow \texttt{LS}$
LS, LP	1.9, 0.25	$\{\texttt{LP}, \texttt{AP}\} \rightarrow \texttt{Clase}$ $\{\texttt{LP}, \texttt{Clase}\} \rightarrow \texttt{LS}$ $\{\texttt{LP}, \texttt{AP}\} \rightarrow \texttt{LS}$ $\{\texttt{AS}, \texttt{LP}\} \rightarrow \texttt{LS}$

Tabla 5.2: Resultados para los atributos LP y LS usando la distancia de Minkowski.

Como se mencionó en la sección 4.4, esta heurística no garantiza que las tuplas que pertenecen a una misma clase de equivalencia sean similares. Debido a esto, realizamos nuevos experimentos con el algoritmo 4.1, utilizando un valor fijo de $\delta = 0.1$ para la longitud del pétalo y valores de δ que van de 0.1 a 1.6 para el ancho del pétalo. Los resultados de este experimento se muestra en la tabla 5.3, al igual que antes, se omiten los resultados que coinciden con DFs exactas. Cómo puede observarse la DF $\{\texttt{LP}, \texttt{AP}\} \rightarrow \texttt{Clase}$ es encontrada cuando $\delta = 1.6$.

δ	DF(s)
1.1	$\{\texttt{Clase}\} \rightarrow \texttt{AP}$ $\{\texttt{LP}\} \rightarrow \texttt{AP}$ $\{\texttt{LS}, \texttt{AS}\} \rightarrow \texttt{AP}$
1.2	$\{\texttt{AS}, \texttt{LP}\} \rightarrow \texttt{AP}$ $\{\texttt{LS}, \texttt{Clase}\} \rightarrow \texttt{AP}$ $\{\texttt{LS}, \texttt{LP}\} \rightarrow \texttt{AP}$ $\{\texttt{LS}, \texttt{AS}\} \rightarrow \texttt{AP}$
1.6	$\{\texttt{LP}, \texttt{AP}\} \rightarrow \texttt{Clase}$ $\{\texttt{Clase}\} \rightarrow \texttt{AP}$

Tabla 5.3: DFs encontradas usando el algoritmo propuesto para calcular clases de equivalencia.

Finalmente cabe destacar que el resultado anterior puede expresarse a través de la siguiente dependencia diferencial:

$$[\text{LP}(\leq 0.1) \wedge \text{AP}(\leq 1.6)] \rightarrow \texttt{Clase}(= 0)$$

En ese sentido, a través de la metodología propuesta en esta tesis es posible inferir dependencias donde una restricción sobre los atributos de una relación debe ser satisfecha. En la siguiente sección se utiliza dicha característica para encontrar anomalías en una base de datos, lo cual resulta de gran ayuda en el proceso de limpieza de datos.

5.5. Detección de anomalías con funciones de similitud

Una manera de verificar la consistencia de los datos es imponer una restricción sobre algún atributo, de manera que todas las tuplas deban cumplir con dicha restricción. Por ejemplo, se puede verificar que los valores de un atributo estén limitados a cierto intervalo, o que los datos tengan cierto formato. Cuando en una relación se establece una dependencia funcional que para ser válida, requiere que dicha restricción sea cumplida por todas las tuplas, entonces si al analizar la relación la dependencia no es inferida se puede determinar que existen ciertas tuplas con anomalías.

Como primer ejemplo de este tipo de aplicaciones consideremos la base de datos RESTAURANT, donde el el atributo `name` determina el nombre de un restaurante, el atributo `addr` determina su dirección, `city` la ciudad donde está ubicado y el atributo `phone` el número telefónico del restaurante. Dada esta información, resulta razonable esperar que exista en conjunto de atributos X que determine de manera única el número telefónico de un restaurante, i.e.,

$$X \rightarrow \{\texttt{phone}\}$$

Al obtener las DFs exactas de RESTAURANT solo $\{\texttt{id}\} \rightarrow \texttt{phone}$ cumple con esta relación. Dado que el atributo `id` es un entero que identifica de manera única cada tupla, resulta obvio que la DF $\{\texttt{id}\} \rightarrow \texttt{phone}$ sea válida en la base de datos. Cuando se analiza la información contenida en esta base de datos se observa que existen tuplas cuyos valores representan la misma dirección no obstante que dichos valores son distintos. Por ejemplo, la dirección del restaurante `arnie morton's of chicago` aparece por primera vez con la cadena `435 s. la cienega blv.`, esta misma dirección aparece una segunda vez con la cadena `435 s. la cienega blvd.`; como se observa estas dos cadenas, aunque difieren en un caracter, representan la misma dirección. Problemas similares son encontrados en los valores que pertenecen al atributo `name`, donde existen cadenas con pequeñas diferencias que representan el mismo nombre de un restaurante. En el caso del atributo `phone` existen inconsistencias en el formato de los datos, en algunas ocasiones se utiliza el caracter `/` para separar el primer grupo de dígitos y en otras ocasiones se utiliza el caracter `-`. Por supuesto, es posible usar una función de similitud como la distancia de Levenshtein con un valor apropiado de δ para inferir las dependencias que involucran a los atributos `name` y `addr`. Sin embargo, nuestro propósito no es inferir estas dependencias funcionales, ya que para esta aplicación se asume que se sabe cuales dependencias deben ser válidas en una base de datos. En su lugar, la metodología propuesta es utilizada para detectar cuando una base de datos presenta inconsistencias.

Como se mencionó antes, en RESTAURANT una DF de la forma $X \rightarrow \{\texttt{phone}\}$ debe ser válida. Una causa de que este dependencia no sea encontrada es debido a los errores de formato en el

atributo phone. Por ejemplo, el restaurante con nombre campanile aparece en una tupla con el teléfono 213/938-1447 y en otra tupla con el teléfono 213-938-1447. Para verificar lo anterior realizamos experimentos utilizando el algoritmo 4.1 con $\delta = 1$, y usando como función de similitud la distancia de Levenshtein. Los resultados de la tabla 5.4 muestran tres dependencias funcionales de la forma $X \rightarrow \{\texttt{phone}\}$, observe que en cada caso $|X| = 3$. Es importante notar que debido a que un restaurante puede tener más de un teléfono no es posible que dependencias como $\{\texttt{name}\} \rightarrow \{\texttt{phone}\}$ o $\{\texttt{addr, city}\} \rightarrow \{\texttt{phone}\}$ sean válidas. Una manera alternativa de establecer estas dependencias es observar que este tipo de relaciones pueden ser descritas a través de dependencias multivaluadas, o usando dependencias aproximadas con un valor apropiado de la función de error. Este resultado muestra que la metodología propuesta puede ser utilizada para verificar si una anomalía en un atributo es responsable de que una DF no sea válida en la relación. Lo anterior tienen la ventaja de que los datos no necesitan ser modificados para verificar que la DF existe, solo es necesario utilizar una función de similitud apropiada.

δ	DF(s)		
1	$\{\texttt{city}, \texttt{type}, \texttt{class}\}$	$\rightarrow$	phone
	$\{\texttt{name}, \texttt{addr}, \texttt{city}\}$	$\rightarrow$	phone
	$\{\texttt{name}, \texttt{city}, \texttt{type}\}$	$\rightarrow$	phone

Tabla 5.4: Resultados usando la distancia de Levenshtein y $\delta = 1$ en el atributo phone.

En una base de datos como RESTAURANT es de esperar que DFs como $\{\texttt{addr}\} \rightarrow \{\texttt{name}\}$ o $\{\texttt{addr}\} \rightarrow \{\texttt{city}\}$ sean válidas. Sin embargo, cuando se infieren las DFs exactas ninguna de estas dependencias es encontrada, esto puede deberse a anomalías en los valores correspondientes a estos atributos. Para verificar lo anterior y finalizar el análisis de esta base de datos usamos la distancia de Levenshtein sobre los atributos addr y city. Dado que las cadenas de estos atributos tienen una longitud que puede superar los 20 caracteres, realizamos experimentos con valores de δ en el intervalo $[1, 10]$ para ambos atributos. Los resultados de la tabla 5.5 muestran que se obtienen 4 nuevas DFs, tres de ellas permiten determinar la dirección de un restaurante a partir de atributos como name, phone y class. Es importante notar que una de las DFs exactas de RESTAURANT es $\{\texttt{phone}\} \rightarrow \{\texttt{address}\}$, por lo tanto se puede concluir que la única razón de que las DFs de la tabla 5.4 sean válidas es debido a que el atributo phone está en el lado izquierdo de la dependencia. La dependencia $\{\texttt{name}, \texttt{city}, \texttt{type}\} \rightarrow \{\texttt{class}\}$ tampoco es útil para verificar la consistencia de los datos, es válida por que la combinación de estos tres atributos hace que los valores de cada tupla sean distintos para cada valor del atributo clase.

Los resultados de este experimento son particularmente interesantes, no obstante que se usaron valores grandes de δ (comparados con la longitud de las cadenas), no fue posible inferir la DF $\{\texttt{addr}\} \rightarrow \{\texttt{city}\}$. Lo anterior se debe a que una vez que el parámetro δ se vuelve lo suficientemente grande, establecer la similitud entre dos valores se vuelve complicado. Es necesario recordar que elegir un valor grande de δ tiene como objetivo agrupar en la misma clase de equivalencia cadenas como 8358 sunset blvd. west y 8358 sunset blvd.. No obstante, lo anterior provoca que cadenas como 903 n. la cienega blvd. y 435 s. la cienega blvd., que claramente representan restaurantes distintos, también sean asignadas a la misma clase de de equivalencia. Como consecuencia de lo anterior no se puede inferir de manera

δ_{addr}	δ_{city}	DF(s)		
2	2	{name, phone}	→	addr
		{phone, class}	→	addr
		{phone, type}	→	addr
3	3	{name, phone}	→	addr
		{phone, class}	→	addr
		{phone, type}	→	addr
6	6	{name, phone}	→	addr
		{phone, class}	→	addr
		{phone, type}	→	addr
8	8	{name, city, type}	→	class
9	9	{name, phone}	→	addr
		{phone, class}	→	addr
		{phone, type}	→	addr

Tabla 5.5: DFs obtenidas usando la distancia de Levenshtein sobre addr y city.

correcta una DF a partir de estos atributos. En ese sentido, es necesario explorar el uso de funciones de similitud distintas a la distancia de Levenshtein, *e. g.*, la similitud del coseno.

5.6. Análisis de la base de datos CRÍMENES

En esta sección se realiza un análisis de las DF exactas presentes en la base de datos CRÍMENES. Esto tiene como objetivo determinar si existen anomalías en los datos que no hayan sido corregidas por las transformaciones descritas en la sección 4.2, así como evaluar si existe alguna relación entre los atributos de la base de datos que describa las condiciones bajo las cuales ocurre un delito. De acuerdo a la sección 5.3, la base de datos CRÍMENES tiene 163 DFs exactas que pueden ser consultadas en el apéndice B.

No obstante el gran número de DFs exactas que son inferidas de esta base de datos, establecer una relación significativa entre los atributos de la base de datos es complicado. Consideremos por ejemplo la DF exacta siguiente:

$$\{\text{COORDENADA}\} \rightarrow \{\text{CUADRANTE}\}$$

Ya que ambos atributos describen la ubicación geográfica donde fue cometido un delito, resulta razonable esperar que ambos estén relacionados, *i. e.*, que a partir del valor del atributo COORDENADA sea posible determinar el valor del atributo CUADRANTE. Sin embargo, al analizar las clases de equivalencia para ambos atributos se observa que en general cada tupla tiene una clase de equivalencia distinta con respecto del atributo COORDENADA. Es decir, la DF es válida ya que en general a cada valor distinto de coordenada le corresponde algún valor del atributo CUADARANTE, ejemplos de esto se muestran en la tabla 5.6. Es importante señalar, que existen algunos casos donde a un par de coordenadas idénticas les corresponde el mismo cuadrante, pero esto no es suficiente para establecer una relación significativa entre estos atributos más allá de lo especificado

por una DF exacta. Este resultado solo debe ser interpretado como que los datos son consistentes en el sentido de que esta dependencia funcional es válida.

Tupla	COORDENADA	CUADRANTE
1	$(-99.051253, 19.488201)$	C013
2	$(-98.991762, 19.409349)$	C112
3	$(-99.006214, 19.404228)$	C108
4	$(-99.033186, 19.405685)$	C095
5	$(-98.996179, 19.402763)$	C116
6	$(-99.021046, 19.4053)$	C092
7	$(-99.05085, 19.48838)$	C013
8	$(-99.032195, 19.401072)$	C081
9	$(-99.011126, 19.400301)$	C094
10269	$(-99.046165, 19.42831)$	C052
10270	$(-99.046165, 19.42831)$	C052

Tabla 5.6: Ejemplos de tuplas para los atributos COORDENADA y CUADRANTE.

Por supuesto, no es suficiente analizar un par de atributos para asegurar que los datos sean consistentes. Ejemplo de esto es la siguiente DF exacta:

$$\{\text{CUADRANTE}\} \rightarrow \{\text{SECTOR}\} \tag{5.1}$$

La tabla 5.7 muestra algunas tuplas y sus valores para estos atributos. Claramente el valor del atributo SECTOR está determinado por el valor del atributo CUADRANTE, *i. e.*, el sector está dado por los primeros dos dígitos que siguen al caracter C. De acuerdo a lo anterior, si los datos son consistentes entonces la DF exacta de la ecuación 5.1 debe ser válida en la base de datos. No obstante, en las FDs exactas mostradas en el apéndice B dicha DF no existe. Esto indica la presencia de anomalías en una o más tuplas de estos atributos. Dado que conocemos el formato que los datos de estos atributos deben cumplir, basta con utilizar las funciones de filtrado de una hoja de cálculo para encontrar la tupla con anomalías. Los resultados de este proceso mostraron que la tupla número 4673 tiene el valor C096 para el atributo CUADRANTE y el valor 8 para el atributo SECTOR, esto claramente es una violación del formato establecido para estos atributos. Después de corregir esta anomalía la DF de la ecuación 5.1 es válida en la base de datos. La discusión anterior ilustra como es posible utilizar las DFs exactas para verificar que la información contenida en una base de datos sea correcta con respecto de dichas relaciones

Para finalizar el análisis de las DF exactas de la base de datos CRÍMENES consideremos las siguientes DFs exactas que son válidas en la base de datos:

1. $\{\text{FECHA}, \text{CALLE}, \text{COORDENADA}\} \rightarrow \{\text{DELITO}\}$
2. $\{\text{FECHA}, \text{ENTRE\_CALLE\_1}, \text{COORDENADA}\} \rightarrow \{\text{DELITO}\}$
3. $\{\text{FECHA}, \text{TURNO}, \text{COORDENADA}\} \rightarrow \{\text{DELITO}\}$
4. $\{\text{HORA\_FIN}, \text{COORDENADA}\} \rightarrow \{\text{DELITO}\}$

Tupla	CUADRANTE	SECTOR
1	C013	1
2	C112	11
3	C108	10
4	C095	9
5	C116	11

Tabla 5.7: Ejemplos de tuplas para los atributos CUADRANTE y SECTOR.

5. $\{\texttt{HORA\_INICIO}, \text{COORDENADA}\} \rightarrow \{\texttt{DELITO}\}$

En cada caso se tiene un conjunto X de atributos de lado izquierdo de la DF mediante el cual se puede determinar el valor para el atributo DELITO. Por ejemplo, consideremos la DF $\{\text{HORA\_FIN, COORDENADA}\} \rightarrow \{\texttt{DELITO}\}$ que relaciona la hora y la coordenada en la que ocurrió un delito con el tipo de delito cometido. Para el atributo COORDENADA se tienen 16215 clases de equivalencia y para el atributo HORA_FIN se tienen 945 clases de equivalencia distintos. Tomando en cuenta que en total se tienen 16425 tuplas en la base de datos es fácil concluir que la mayoría de los valores para el par $(\text{HORA\_FIN}, \text{COORDENADA})$ son distintos y que por lo tanto esta DF es en general obvia. Sin embargo, existen algunas tuplas que tienen valores duplicados, dichas tuplas son mostradas en la tabla 5.8. La única razón por la cual las tuplas 10167 a 15402 son duplicados es debido a que el atributo COORDENADA toma el valor $(0.0, 0.0)$, estas tuplas son aquellas que originalmente no tenían valor alguno para el atributo COORDENADA y que como parte del proceso de limpieza de datos les fue asignado un valor por defecto. Por lo tanto estas tuplas no aportan información alguna que pueda ser utilizada para determinar las condiciones de un delito. El caso de la tupla 8651 y 8652 corresponde a tuplas duplicadas, aunque en la tabla se han omitido los valores de la mayoría de los atributos, estas dos tuplas contienen exactamente la misma información. Finalmente solo las tuplas 242, 1088, 1109 y 3963 representan casos en donde una misma hora y coordenada corresponde a la ocurrencia de dos delitos distintos. De este análisis se puede concluir que no obstante que las DFs exactas son herramientas valiosas para verificar la consistencia de los datos y como herramienta de apoyo en el diseño de bases de datos relacionales, no pueden ser utilizadas para establecer relaciones causa - efecto como las condiciones que pueden llevar a la ocurrencia de un delito.

Tupla	HORA_FIN	DELITO	COORDENADA
242	17:10	ROBO EN VIA PUBLICA	-99.036657, 19.403078
1088	17:10	ROBO EN VIA PUBLICA	-99.036657, 19.403078
1109	19:14	ROBO EN VIA PUBLICA	-99.037049, 19.403258
3963	19:14	ROBO EN VIA PUBLICA	-99.037049, 19.403258
8651	9:15	ROBO A COMERCIO	-99.046531, 19.47813
8652	9:15	ROBO A COMERCIO	-99.046531, 19.47813
10167	14:00	ROBO DE VEHICULO	0.0, 0.0
11599	13:00	ROBO DE VEHICULO	0.0, 0.0
12778	22:30	ROBO DE VEHICULO	0.0, 0.0
13710	13:00	ROBO DE VEHICULO	0.0, 0.0
13825	22:30	ROBO DE VEHICULO	0.0, 0.0
14065	20:00	ROBO DE VEHICULO	0.0, 0.0
14090	20:00	ROBO DE VEHICULO	0.0, 0.0
14258	14:00	ROBO DE VEHICULO	0.0, 0.0
14561	13:00	ROBO DE VEHICULO	0.0, 0.0
15349	13:00	ROBO DE VEHICULO	0.0, 0.0
15402	14:00	ROBO DE VEHICULO	0.0, 0.0

Tabla 5.8: Tuplas duplicadas para el par (HORA_FIN, COORDENADA).

Parte III

Conclusiones

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Los resultados del presente trabajo de tesis muestran que el uso de dependencias funcionales exactas para analizar la base de datos CRÍMENES no arroja ninguna dependencia funcional útil entre los atributos que caracterizan la ocurrencia de un crimen. Debido a la naturaleza espacial de este conjunto de datos, solo fue posible obtener dependencias que pueden ser consideradas como obvias. Para realizar descubrimiento automatizado de conocimiento mediante el uso de dependencias funcionales en una base de datos relacional es necesario que esta cuente con un conjunto de atributos posiblemente relacionados, de tal forma que este método nos pueda ayudar a confirmar ciertas sospechas sobre la relación de los atributos. Sin embargo, mediante el uso de funciones de similitud fue posible encontrar dependencias funcionales que aunque obvias, resaltan anomalías residuales durante el proceso de limpieza de datos y por lo tanto pueden ser utilizadas para complementar dicho proceso.

Los experimentos realizados sobre las bases de dato RESTAURANT muestran que si bien la metodología propuesta puede ser utilizada para detectar anomalías, no es posible establecer una DF que claramente debe ser válida en una relación si un reducido número de tuplas no cumplen con la misma. De manera similar cuando se tienen relaciones que asocian a cada valor de un conjunto de atributos uno o más valores de otro conjunto de atributos, tampoco es posible inferir relación alguna. Para inferir este tipo de relaciones, comúnmente conocidas "como uno a muchos", es necesario recurrir a otro tipo de dependencias que describan adecuadamente dichas relaciones, *e. g.*, dependencias multivaluadas.

De los resultados de la base de datos de CRÍMENES podemos concluir que las dependencias funcionales exactas no son aptas para ser utilizadas para el descubrimiento automático de conocimiento en bases de datos, ya que estas solo establecen una relación de correspondencia matemática entre un conjunto de atributos. Por lo tanto, no se puede establecer una relación semántica entre ambos conjuntos, sino solo para consistencia y diseño de la base de datos.

No obstante lo anterior, la modificación propuesta al algoritmo TANE es capaz de inferir de manera correcta relaciones significativas en otros conjuntos de datos. Estas relaciones no pueden ser halladas cuando se consideran solamente dependencias funcionales exactas, lo que resalta la importancia del uso de funciones de similitud en el análisis de bases de datos. Las pruebas de desempeño muestran además que el algoritmo propuesto puede obtener resultados correctos en un tiempo aceptable (algunos segundos o minutos, según el número de atributos) tomando en cuenta que se esta trabajando en un espacio de soluciones exponencial y que el uso de funciones de similitud requiere tiempo de computo adicional. Los experimentos realizados sobre múltiples conjuntos de datos muestran que los resultados son altamente sensibles a la función de similitud particular que se elija para un atributo dado y al límite de similitud correspondiente.

6.2. Trabajo futuro

En su forma actual nuestra metodología no es adecuada para realizar descubrimiento de conocimiento en la base de datos CRÍMENES. Sin embargo, se pueden considerar modificaciones adicionales a nuestra versión del algoritmo TANE para calcular dependencias funcionales aproximadas o multivaluadas basadas en funciones de similitud. Al hacer esto, podemos descartar un cierto número de tuplas que evitan que una dependencia sea válida, y también tomar en cuenta anomalías residuales en los datos, o datos que contienen tuplas con valores similares.

En trabajos recientes se han propuesto algunas modificaciones adicionales al algoritmo TANE que mejoran el tiempo en el que encuentras las dependencias funcionales [53][54], estas podrían ser implementadas para disminuir el tiempo de ejecución de los experimentos.

Adicionalmente, con la obtención de las dependencias funcionales en la base de datos, es posible realizar de forma inmediata la normalización a la forma normal de Boyce-Codd. Por lo que se podría proponer un diseño para la base de datos CRÍMENES de tal forma que ayude en el proceso de consulta para el Laboratorio Digital de Violencia y Paz.

Apéndices

Apéndice A

Código fuente

A.1. Limpieza de datos

A continuación se incluye la implementación en Python de las transformaciones descritas en la sección 4.2 para la limpieza de datos en la bases de datos `Crímenes`. Para implementar estas transformaciones se hace uso de la biblioteca de análisis de datos para Python *pandas*. El código fuente correspondiente se encuentra disponible en [55].

A.1.1. Datos faltantes

En la base de datos existen múltiples tuplas que no tiene un valor para cierto atributo. Para eliminar esta anomalía, cada tupla con un dato faltante será llenado con la cadena `Sin NOMBRE_ATRIBUTO`. Donde NOMBRE_ATRIBUTO es el nombre del atributo para la cual la tupla correspondiente tiene un valor faltante. Para realizar esta transformación basta con utilizar el método `fillna()` de una serie de Pandas como sigue:

```
for column in df.columns:
    df[column] = df[column].fillna('Sin␣' + column)
```

A.1.2. Errores léxicos

Esta transformación tiene como objetivo eliminar el caracter que representa una coma (`,`) con el caracter que representa un punto (`.`). Para realizar esta transformación basta con utilizar el método `replace()` de una cadena en Python.

```
for column in df.columns:
    df[column] = df[column].str.replace(',', '.', regex=False)
```

A.1.3. Errores de formato

Existen tuplas que contienen una hora de inicio y una hora de fin. Con el objetivo de tener tuplas con formatos consistentes, los datos de esta atributo son divididos. El atributo HORA_INICIO tendrá la hora en la que comenzó el delito, y el atributo HORA_FIN tendrá la hora en la que terminó el delito. Para aquellas tuplas que solo tengan una hora, la hora de inicio y la hora de fin serán la misma. Para realizar esta transformación utilizamos el método `split()` de una cadena y el método `where()` de NumPy.

```
df['HORA_INICIO'], df['HORA_FIN'] = df['HORA'].str.split('␣A␣', 1).str
df['HORA_FIN'] = np.where(pd.isna(df['HORA_FIN']), df['HORA_INICIO'], df['
    HORA_FIN'])
```

El caso del atributo FECHA es similar, en este caso los datos serán transformados de manera que cada fecha cumpla con el formato `DIA/MES/AÑO`. Para realizar dicha información se hace

uso de expresiones regulares de manera que se pueda extraer cada campo de una fecha y después se construye una cadena con el formato correcto.

```
import datetime as dt

# Elimina las filas donde el mes está dado con nombre
df['FECHA'] = df['FECHA'].str.replace('Jul', '07', regex=False)
df['FECHA'] = df['FECHA'].str.replace('ago', '08', regex=False)

# Unifica el separador de dia mes y año
df['FECHA'] = df['FECHA'].str.replace('-', '/', regex=False)

# Agrega el año a las tuplas que no lo tienen
dates = df['FECHA'].str.extract(r'^((\d{2}|\d{1})/(\d{2}|\d{1})/(\d{4}|\d{2}))'
    , expand=False)

df['FECHA'] = np.where(dates[0].isnull(), df['FECHA'] + '/2017', df['FECHA'])

dates = df['FECHA'].str.extract(r'(/(\d{2}))$', expand=False)
dates2 = df['FECHA'].str.extract(r'^((\d{2}|\d{1})/(\d{2}|\d{1})/)', expand=
    False)
df['FECHA'] = np.where(dates[0].notnull(), dates2[0] + '20' + dates[1], df['
    FECHA'])
```

A.1.4. Concatenación de la latitud y longitud

Como mencionamos en la sección 4.2.5, los valores para los atributos X y Y son utilizados para crear un nuevo atributo llamado COORDENADA. Esto tiene el objetivo de poder utilizar la distancia de Minkowski en dos dimensiones para inferir alguna DF que involucre a este atributo. Esta transformación se sencilla solo se concatenan los datos de los atributos X y Y y se separan con un espacio, finalmente se eliminan las columnas de estos atributos.

```
df['COORDENADA'] = df['X'] + ' ' + df['Y']
df = df.drop('X', 1)
df = df.drop('Y', 1)
```

A.2. Cálculo de clases de equivalencia usando funciones de similitud

En esta sección se presenta la implementación en C++ del algoritmo 4.1 para el cálculo de clases de equivalencia usando funciones de similitud. El código fuente presentado en este apéndice se encuentra disponible en [56]. Con el objetivo de tener una implementación clara, en lugar de recibir como parámetros una tupla y un atributo B, la función `getEquivalenceClass()` recibe una cadena `value` que representa el valor de una tupla con respecto de un atributo que es identificado a través del entero `col_idx` que también es un parámetro de entrada.

```
int getEquivalenceClass(std::string& value, int col_idx)
{
    if (col_idx >= value_map.size()) {
        value_map.emplace_back(std::unordered_map<std::string, int>());
```

```
        value_map[value_map.size() - 1].reserve(100000);
    }

    if(_indices.count(col_idx) > 0)
    {
        if(centersMap.count(col_idx) == 0)
        {
            centersMap[col_idx] = std::vector<std::string>();
            centersMap[col_idx].push_back(value);
            return 0;
        }
        else
        {
            std::string metric = _indices[col_idx].first;

            int eqClass = -1;
            float minDistance = FLT_MAX;

            for(int j = 0; j < centersMap[col_idx].size(); j++)
            {
                if(metric == "w")
                {
                    // Parse strings into numeric vector
                    std::regex re("[␣]+");
                    std::sregex_token_iterator it1(centersMap[col_idx][j].
                       begin(), centersMap[col_idx][j].end(), re, -1);
                    std::sregex_token_iterator it2(value.begin(), value.end()
                       , re, -1);
                    std::sregex_token_iterator reg_end;

                    std::vector<float> x;
                    std::vector<float> y;

                    for(; it1 != reg_end && it2 != reg_end; ++it1, ++it2)
                    {
                        x.push_back(std::stof(it1->str()));
                        y.push_back(std::stof(it2->str()));
                    }

                    float d = getDifference(x, y);
                    float delta = std::stof(_indices[col_idx].second);

                    if(d <= delta)
                    {
                        if(d < minDistance)
                        {
                            minDistance = d;
                            eqClass = j;
                        }
                    }
                }
                else if(metric == "d")
                {
                    // Parsea las cadenas de la fecha
                    std::regex re("[/]+");
```

```
            std::sregex_token_iterator it1(centersMap[col_idx][j].
               begin(), centersMap[col_idx][j].end(), re, -1);
            std::sregex_token_iterator it2(value.begin(), value.end()
                , re, -1);
            std::sregex_token_iterator reg_end;

            int d1[3] = { 0, 0, 0 };
            int d2[3] = { 0, 0, 0 };

            for(int i = 0; it1 != reg_end && it2 != reg_end; ++it1,
               ++it2, ++i)
            {
                d1[i] = std::stoi(it1->str());
                d2[i] = std::stoi(it2->str());
            }

            Date date1 = { d1[0], d1[1], d1[2] };
            Date date2 = { d2[0], d2[1], d2[2] };

            int d = getDifference(date1, date2);
            long delta = std::stol(_indices[col_idx].second);

            if(d <= delta)
            {
                if(d < minDistance)
                {
                    minDistance = d;
                    eqClass = j;
                }
            }
        }
        else if(metric == "t")
        {
            // Parse las cadenas de la hora
            std::regex re("[:]");
            std::sregex_token_iterator it1(centersMap[col_idx][j].
               begin(), centersMap[col_idx][j].end(), re, -1);
            std::sregex_token_iterator it2(value.begin(), value.end()
                , re, -1);
            std::sregex_token_iterator reg_end;

            unsigned t1[2] = { 0, 0 };
            unsigned t2[2] = { 0, 0 };

            for(int i = 0; it1 != reg_end && it2 != reg_end; ++it1,
               ++it2, ++i)
            {
                t1[i] = std::stoi(it1->str());
                t2[i] = std::stoi(it2->str());
            }

            Time time1 = { t1[0], t1[1] };
            Time time2 = { t2[0], t2[1] };

            long d = getDifference(time1, time2);
```

```
                    long threshold = std::stol(_indices[col_idx].second);

                    if(d <= threshold)
                    {
                        if(d < minDistance)
                        {
                            minDistance = d;
                            eqClass = j;
                        }
                    }
                }
                else if(metric == "1")
                {
                    size_t d = getDifference(centersMap[col_idx][j], value);
                    float threshold = std::stoi(_indices[col_idx].second);

                    if(d <= threshold)
                    {
                        if(d < minDistance)
                        {
                            minDistance = d;
                            eqClass = j;
                        }
                    }
                }
            }

            if(eqClass == -1)
            {
                // Agrega un nuevo centro
                int code = centersMap[col_idx].size();
                centersMap[col_idx].push_back(value);
                return code;
            }

            return eqClass;
        }
    }

    auto iter = value_map[col_idx].find(value);

    if (iter != value_map[col_idx].end())
    {
        return iter->second;
    }
    else
    {
        int code = value_map[col_idx].size();
        value_map[col_idx][value] = code;
        return code;
    }
}
```

A.3. Funciones de similitud

En esta sección se presentan las implementaciones en C++ de las siguientes funciones de similitud:

- Distancia de Minkowski.
- Distancia de Levenshtein.
- Distancia entre dos fechas en días.
- Distancia entre dos horas en minutos.

La sección 2.8.1 describe con mayor detalle cada una de estas funciones. El código fuente correspondiente se encuentra disponible en [56].

A.3.1. Distancia de Minkowski

```
float getDifference(const std::vector<float>& x, const std::vector<float>& y)
{
    float sum = 0.0f;

    for(int i = 0; i < x.size(); i++)
    {
        sum += ((x[i] - y[i]) * (x[i] - y[i]));
    }

    return sqrtf(sum);
}
```

A.3.2. Distancia de Levenshtein

```
size_t getDifference(const std::string &s1, const std::string &s2)
{
    const size_t m(s1.size());
    const size_t n(s2.size());

    if( m==0 ) return n;
    if( n==0 ) return m;

    size_t *costs = new size_t[n + 1];

    for( size_t k=0; k<=n; k++ ) costs[k] = k;

    size_t i = 0;
    for ( std::string::const_iterator it1 = s1.begin(); it1 != s1.end(); ++it1,
        ++i )
    {
        costs[0] = i+1;
        size_t corner = i;

        size_t j = 0;
        for ( std::string::const_iterator it2 = s2.begin(); it2 != s2.end(); ++
            it2, ++j )
```

```
        {
            size_t upper = costs[j+1];
            if( *it1 == *it2 )
            {
                costs[j+1] = corner;
            }
            else
            {
                size_t t(upper<corner?upper:corner);
                costs[j+1] = (costs[j]<t?costs[j]:t)+1;
            }

            corner = upper;
        }
    }

    size_t result = costs[n];
    delete [] costs;

    return result;
}
```

A.3.3. Distancia entre dos fechas en días

```
struct Date
{
    int d, m, y;
};

// This function counts number of leap years before the given date
int countLeapYears(Date d)
{
    int years = d.y;

    // Check if the current year needs to be considered needs to be considered
    // for the count of leap years or not
    if(d.m <= 2)
    {
        years--;
    }

    // A year is a leap year if it is a multiple of 4, multiple of 400 and not
    // a multiple of 100
    return years / 4 - years / 100 + years / 400;
}

// This function returns the number of days between two given dates
long getDifference(const Date& dt1, const Date& dt2)
{
    // To store the number of days in each month
    const int monthDays[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    // Count total number of days before first date

    // Initialize count using day and years
```

```
    long int n1 = dt1.y * 365 + dt1.d;

    // Add days form months in given date
    for(int i = 0; i < dt1.m - 1; i++)
    {
        n1 += monthDays[i];
    }

    // Since very leap year is off 366 days. Add a day for eavery leap day
    n1 += countLeapYears(dt1);

    // Similary, count total number of days before dt2
    long int n2 = dt2.y * 365 + dt2.d;

    for(int i = 0; i < dt2.m - 1; i++)
    {
        n2 += monthDays[i];
    }

    n2 += countLeapYears(dt2);

    // return difference between two counts
    return abs(n2 - n1);
}
```

A.3.4. Distancia entre dos horas en minutos

```
struct Time
{
    unsigned int h, m;
};

long getDifference(const Time& t1, const Time& t2)
{
    long m1 = t1.h * 60 + t1.m;
    long m2 = t2.h * 60 + t2.m;

    return abs(m1 - m2);
}
```

Apéndice B

Dependencias funcionales

A continuación se listan todas las DFs exactas de la base de datos Crímenes.

1. {FECHA DELITO No. COLONIA COORDENADA } → {CALLE}
2. {FECHA ENTRE_CALLE_1 COORDENADA } → {CALLE}
3. {FECHA No. TURNO COORDENADA } → {CALLE}
4. {HORA_FIN ENTRE_CALLE_1 COLONIA COORDENADA } → {CALLE}
5. {HORA_FIN FECHA COORDENADA } → {CALLE}
6. {HORA_FIN FECHA DELITO ENTRE_CALLE_1 ENTRE_CALLE_2 CUADRANTE } → {CALLE}
7. {HORA_FIN FECHA DELITO No. ENTRE_CALLE_1 CUADRANTE } → {CALLE}
8. {HORA_INICIO ENTRE_CALLE_1 COLONIA COORDENADA } → {CALLE}
9. {HORA_INICIO FECHA COORDENADA } → {CALLE}
10. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 CUADRANTE } → {CALLE}
11. {CALLE ENTRE_CALLE_1 COORDENADA } → {COLONIA}
12. {FECHA CALLE COORDENADA } → {COLONIA}
13. {FECHA ENTRE_CALLE_1 COORDENADA } → {COLONIA}
14. {FECHA TURNO COORDENADA } → {COLONIA}
15. {HORA_FIN CALLE COORDENADA } → {COLONIA}
16. {HORA_FIN FECHA CALLE ENTRE_CALLE_1 } → {COLONIA}
17. {HORA_FIN FECHA COORDENADA } → {COLONIA}
18. {HORA_FIN FECHA DELITO CALLE CUADRANTE } → {COLONIA}
19. {HORA_FIN FECHA DELITO CALLE ENTRE_CALLE_2 } → {COLONIA}
20. {HORA_FIN FECHA DELITO CALLE No. } → {COLONIA}
21. {HORA_FIN FECHA DELITO ENTRE_CALLE_1 } → {COLONIA}
22. {HORA_INICIO CALLE COORDENADA } → {COLONIA}
23. {HORA_INICIO FECHA CALLE ENTRE_CALLE_1 } → {COLONIA}
24. {HORA_INICIO FECHA COORDENADA } → {COLONIA}
25. {HORA_INICIO FECHA DELITO CALLE CUADRANTE } → {COLONIA}
26. {HORA_INICIO FECHA DELITO CALLE ENTRE_CALLE_2 } → {COLONIA}
27. {HORA_INICIO FECHA DELITO CALLE No. } → {COLONIA}
28. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 CUADRANTE } → {COLONIA}

29. {COORDENADA } → {CUADRANTE}
30. {HORA_FIN FECHA DELITO CALLE COLONIA } → {CUADRANTE}
31. {HORA_FIN FECHA DELITO CALLE ENTRE_CALLE_1 } → {CUADRANTE}
32. {HORA_FIN FECHA DELITO CALLE ENTRE_CALLE_2 } → {CUADRANTE}
33. {HORA_FIN FECHA DELITO CALLE No. } → {CUADRANTE}
34. {HORA_INICIO FECHA DELITO CALLE COLONIA } → {CUADRANTE}
35. {HORA_INICIO FECHA DELITO CALLE ENTRE_CALLE_1 } → {CUADRANTE}
36. {HORA_INICIO FECHA DELITO CALLE ENTRE_CALLE_2 } → {CUADRANTE}
37. {HORA_INICIO FECHA DELITO CALLE No. } → {CUADRANTE}
38. {FECHA CALLE COORDENADA } → {DELITO}
39. {FECHA ENTRE_CALLE_1 COORDENADA } → {DELITO}
40. {FECHA TURNO COORDENADA } → {DELITO}
41. {HORA_FIN COORDENADA } → {DELITO}
42. {HORA_INICIO COORDENADA } → {DELITO}
43. {CALLE ENTRE_CALLE_2 COLONIA TURNO COORDENADA } → {ENTRE_CALLE_1}
44. {FECHA CALLE COORDENADA } → {ENTRE_CALLE_1}
45. {FECHA DELITO No. COLONIA COORDENADA } → {ENTRE_CALLE_1}
46. {FECHA No. TURNO COORDENADA } → {ENTRE_CALLE_1}
47. {HORA_FIN CALLE COORDENADA } → {ENTRE_CALLE_1}
48. {HORA_FIN FECHA COORDENADA } → {ENTRE_CALLE_1}
49. {HORA_FIN FECHA DELITO CALLE ENTRE_CALLE_2 } → {ENTRE_CALLE_1}
50. {HORA_INICIO CALLE COORDENADA } → {ENTRE_CALLE_1}
51. {HORA_INICIO FECHA COORDENADA } → {ENTRE_CALLE_1}
52. {HORA_INICIO FECHA DELITO CALLE ENTRE_CALLE_2 } → {ENTRE_CALLE_1}
53. {DELITO ENTRE_CALLE_1 COORDENADA } → {ENTRE_CALLE_2}
54. {ENTRE_CALLE_1 TURNO COORDENADA } → {ENTRE_CALLE_2}
55. {FECHA COORDENADA } → {ENTRE_CALLE_2}
56. {HORA_FIN COORDENADA } → {ENTRE_CALLE_2}
57. {HORA_FIN FECHA CALLE ENTRE_CALLE_1 } → {ENTRE_CALLE_2}
58. {HORA_FIN FECHA No. ENTRE_CALLE_1 } → {ENTRE_CALLE_2}
59. {HORA_INICIO CALLE No. ENTRE_CALLE_1 CUADRANTE TURNO } → {ENTRE_CALLE_2}
60. {HORA_INICIO COORDENADA } → {ENTRE_CALLE_2}
61. {HORA_INICIO FECHA CALLE ENTRE_CALLE_1 } → {ENTRE_CALLE_2}
62. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 CUADRANTE } → {ENTRE_CALLE_2}
63. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 SECTOR } → {ENTRE_CALLE_2}

64. {HORA_INICIO FECHA ENTRE_CALLE_1 COLONIA CUADRANTE } → {ENTRE_CALLE_2}

65. {HORA_INICIO FECHA ENTRE_CALLE_1 COLONIA SECTOR } → {ENTRE_CALLE_2}

66. {HORA_INICIO FECHA No. ENTRE_CALLE_1 CUADRANTE } → {ENTRE_CALLE_2}

67. {HORA_INICIO FECHA No. ENTRE_CALLE_1 SECTOR } → {ENTRE_CALLE_2}

68. {HORA_INICIO HORA_FIN DELITO CALLE No. ENTRE_CALLE_1 TURNO } → {ENTRE_CALLE_2}

69. {HORA_INICIO HORA_FIN FECHA DELITO ENTRE_CALLE_1 } → {ENTRE_CALLE_2}

70. {HORA_INICIO HORA_FIN FECHA ENTRE_CALLE_1 COLONIA } → {ENTRE_CALLE_2}

71. {No. ENTRE_CALLE_1 COORDENADA } → {ENTRE_CALLE_2}

72. {HORA_FIN CALLE TURNO COORDENADA } → {FECHA}

73. {HORA_FIN ENTRE_CALLE_1 COLONIA TURNO COORDENADA } → {FECHA}

74. {HORA_INICIO CALLE TURNO COORDENADA } → {FECHA}

75. {HORA_INICIO ENTRE_CALLE_1 COLONIA TURNO COORDENADA } → {FECHA}

76. {FECHA CALLE TURNO COORDENADA } → {HORA_FIN}

77. {FECHA ENTRE_CALLE_1 TURNO COORDENADA } → {HORA_FIN}

78. {FECHA No. TURNO COORDENADA } → {HORA_FIN}

79. {HORA_INICIO CALLE COORDENADA } → {HORA_FIN}

80. {HORA_INICIO COLONIA COORDENADA } → {HORA_FIN}

81. {HORA_INICIO ENTRE_CALLE_1 COORDENADA } → {HORA_FIN}

82. {HORA_INICIO FECHA CALLE CUADRANTE } → {HORA_FIN}

83. {HORA_INICIO FECHA CALLE ENTRE_CALLE_1 } → {HORA_FIN}

84. {HORA_INICIO FECHA CALLE ENTRE_CALLE_2 COLONIA } → {HORA_FIN}

85. {HORA_INICIO FECHA CALLE No. } → {HORA_FIN}

86. {HORA_INICIO FECHA CALLE SECTOR } → {HORA_FIN}

87. {HORA_INICIO FECHA COORDENADA } → {HORA_FIN}

88. {HORA_INICIO FECHA DELITO CALLE } → {HORA_FIN}

89. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 COLONIA SECTOR } → {HORA_FIN}

90. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 ENTRE_CALLE_2 COLONIA } → {HORA_FIN}

91. {HORA_INICIO FECHA ENTRE_CALLE_1 CUADRANTE } → {HORA_FIN}

92. {HORA_INICIO FECHA No. ENTRE_CALLE_1 COLONIA SECTOR } → {HORA_FIN}

93. {HORA_INICIO FECHA No. ENTRE_CALLE_1 ENTRE_CALLE_2 COLONIA } → {HORA_FIN}

94. {HORA_INICIO TURNO COORDENADA } → {HORA_FIN}

95. {FECHA CALLE TURNO COORDENADA } → {HORA_INICIO}

96. {FECHA ENTRE_CALLE_1 TURNO COORDENADA } → {HORA_INICIO}

97. {FECHA No. TURNO COORDENADA } → {HORA_INICIO}

98. {HORA_FIN CALLE COORDENADA } → {HORA_INICIO}

99. {HORA_FIN COLONIA COORDENADA } → {HORA_INICIO}
100. {HORA_FIN ENTRE_CALLE_1 COORDENADA } → {HORA_INICIO}
101. {HORA_FIN FECHA CALLE ENTRE_CALLE_1 TURNO } → {HORA_INICIO}
102. {HORA_FIN FECHA CALLE ENTRE_CALLE_2 COLONIA TURNO } → {HORA_INICIO}
103. {HORA_FIN FECHA CALLE ENTRE_CALLE_2 SECTOR TURNO } → {HORA_INICIO}
104. {HORA_FIN FECHA CALLE No. COLONIA TURNO } → {HORA_INICIO}
105. {HORA_FIN FECHA CALLE No. SECTOR TURNO } → {HORA_INICIO}
106. {HORA_FIN FECHA COORDENADA } → {HORA_INICIO}
107. {HORA_FIN FECHA DELITO CALLE } → {HORA_INICIO}
108. {HORA_FIN FECHA DELITO ENTRE_CALLE_1 ENTRE_CALLE_2 } → {HORA_INICIO}
109. {HORA_FIN FECHA DELITO ENTRE_CALLE_2 CUADRANTE } → {HORA_INICIO}
110. {HORA_FIN FECHA DELITO No. CUADRANTE } → {HORA_INICIO}
111. {HORA_FIN FECHA DELITO No. ENTRE_CALLE_1 } → {HORA_INICIO}
112. {HORA_FIN FECHA ENTRE_CALLE_1 ENTRE_CALLE_2 TURNO } → {HORA_INICIO}
113. {HORA_FIN FECHA ENTRE_CALLE_2 CUADRANTE TURNO } → {HORA_INICIO}
114. {HORA_FIN FECHA No. CUADRANTE TURNO } → {HORA_INICIO}
115. {HORA_FIN FECHA No. ENTRE_CALLE_1 TURNO } → {HORA_INICIO}
116. {FECHA CALLE COORDENADA } → {No.}
117. {FECHA ENTRE_CALLE_1 COORDENADA } → {No.}
118. {HORA_FIN COORDENADA } → {No.}
119. {HORA_INICIO COORDENADA } → {No.}
120. {COORDENADA } → {SECTOR}
121. {DELITO CALLE CUADRANTE TURNO } → {SECTOR}
122. {DELITO ENTRE_CALLE_1 CUADRANTE TURNO } → {SECTOR}
123. {FECHA CALLE CUADRANTE } → {SECTOR}
124. {FECHA CUADRANTE TURNO } → {SECTOR}
125. {FECHA DELITO CALLE ENTRE_CALLE_1 COLONIA TURNO } → {SECTOR}
126. {FECHA DELITO CALLE ENTRE_CALLE_1 ENTRE_CALLE_2 COLONIA } → {SECTOR}
127. {FECHA DELITO CUADRANTE } → {SECTOR}
128. {HORA_FIN CUADRANTE } → {SECTOR}
129. {HORA_FIN DELITO CALLE ENTRE_CALLE_1 ENTRE_CALLE_2 COLONIA TURNO } → {SECTOR}
130. {HORA_FIN DELITO CALLE No. ENTRE_CALLE_1 COLONIA TURNO } → {SECTOR}
131. {HORA_FIN FECHA DELITO CALLE } → {SECTOR}
132. {HORA_FIN FECHA DELITO COLONIA } → {SECTOR}
133. {HORA_FIN FECHA DELITO ENTRE_CALLE_1 } → {SECTOR}

134. {HORA_INICIO CUADRANTE } → {SECTOR}

135. {HORA_INICIO DELITO CALLE No. ENTRE_CALLE_1 COLONIA TURNO } → {SECTOR}

136. {HORA_INICIO FECHA DELITO CALLE } → {SECTOR}

137. {HORA_INICIO FECHA DELITO ENTRE_CALLE_1 ENTRE_CALLE_2 } → {SECTOR}

138. {HORA_INICIO FECHA DELITO ENTRE_CALLE_2 COLONIA } → {SECTOR}

139. {HORA_INICIO HORA_FIN DELITO CALLE ENTRE_CALLE_1 COLONIA TURNO } → {SECTOR}

140. {HORA_FIN FECHA COORDENADA } → {TURNO}

141. {HORA_FIN FECHA DELITO CALLE } → {TURNO}

142. {HORA_FIN FECHA DELITO COLONIA } → {TURNO}

143. {HORA_FIN FECHA DELITO CUADRANTE } → {TURNO}

144. {HORA_FIN FECHA DELITO ENTRE_CALLE_1 } → {TURNO}

145. {HORA_FIN FECHA DELITO SECTOR } → {TURNO}

146. {HORA_INICIO FECHA CALLE } → {TURNO}

147. {HORA_INICIO FECHA COLONIA CUADRANTE } → {TURNO}

148. {HORA_INICIO FECHA COLONIA SECTOR } → {TURNO}

149. {HORA_INICIO FECHA COORDENADA } → {TURNO}

150. {HORA_INICIO FECHA ENTRE_CALLE_1 COLONIA } → {TURNO}

151. {HORA_INICIO FECHA ENTRE_CALLE_1 CUADRANTE } → {TURNO}

152. {HORA_INICIO FECHA ENTRE_CALLE_1 ENTRE_CALLE_2 } → {TURNO}

153. {HORA_INICIO FECHA ENTRE_CALLE_1 SECTOR } → {TURNO}

154. {HORA_INICIO FECHA ENTRE_CALLE_2 COLONIA } → {TURNO}

155. {HORA_INICIO FECHA No. COLONIA } → {TURNO}

156. {HORA_INICIO FECHA No. ENTRE_CALLE_1 } → {TURNO}

157. {HORA_INICIO HORA_FIN FECHA COLONIA } → {TURNO}

158. {HORA_INICIO HORA_FIN FECHA CUADRANTE } → {TURNO}

159. {HORA_INICIO HORA_FIN FECHA DELITO } → {TURNO}

160. {HORA_INICIO HORA_FIN FECHA ENTRE_CALLE_1 } → {TURNO}

161. {HORA_INICIO HORA_FIN FECHA ENTRE_CALLE_2 } → {TURNO}

162. {HORA_INICIO HORA_FIN FECHA No. } → {TURNO}

163. {HORA_INICIO HORA_FIN FECHA SECTOR } → {TURNO}

Apéndice C

Manual de usuario

En este apéndice se describe la operación del programa que fue utilizado para realizar los experimentos descritos en el capítulo 5. Nuestra implementación del algoritmo TANE y el algoritmo 4.1 fue realizada en el lenguaje de programación C++ y ha sido ejecutada tanto en Windows 10, como en varias distribuciones de Linux, *e. g.*, Ubuntu 19.04.

Nuestro programa es una aplicación de consola con las siguientes características.

1. Nombre del programa y comando de ejecución: `DependencyMiner`
2. Sinopsis: Realiza la inferencia de todas la dependencias funcionales exactas, o basadas en funciones de similitud de una relación dada en forma de un archivo con formato de valores separados por comas (`*.csv`). El programa asume que la primera línea del archivo contiene los nombres de cada atributo en la relación. Como salida el programa produce tres archivos de texto descritos a continuación:
   - `<prefijo_salida>.txt`. Este archivo contiene las dependencias funcionales inferidas para el archivo de entrada. En este archivo cada atributo aparece con un identificador numérico en el intervalo $[0, n-1]$ donde n es el número de atributos en el archivo de entrada.
   - `<prefijo_salida>_unsorted.txt`. Este archivo contiene las dependencias funcionales inferidas para el archivo de entrada. Cada atributo aparece con el nombre dado por la primera línea del archivo de entrada.
   - `<prefijo_salida>_sorted.txt`. En este archivo, las dependencias funcionales inferidas aparecen en orden lexicográfico de acuerdo al lado derecho de la dependencia. Cada atributo aparece con el nombre dado por la primera línea del archivo de entrada.

   La cadena de caracteres `<prefijo_salida>` representa el prefijo que será agregado a estos archivos de salida y puede ser especificado por el usuario como un argumento al ejecutar el programa.
3. Sintaxis:

```
./DependencyMiner -f <nombre_archivo> -o <prefijo_salida>
                  [-s <similitud>] [-a <lista_atributos>]
                  [-m <lista_funciones>] [-v <lista_deltas>]
```

4. Argumentos.
   - `-f <nombre_archivo`. Con esta argumento se especifica el archivo `*.csv` cuyas DFs serán inferidas, `nombre_archivo` es una cadena de caracteres que representa la ruta a dicho archivo incluyendo el nombre. Este argumento no puede ser omitido.

- `-o <prefijo_salida>`. Este argumento y la cadena `prefijo_salida`, especifica el prefijo que será utilizado para los nombres de cada archivo de salida producido por el programa. Por ejemplo, si `prefijo_salida` es la cadena `prueba`, los archivos de salida tendrán los nombres `prueba.txt`, `prueba_unsorted.txt` y `prueba_sorted.txt`. Este argumento no puede ser omitido.
- `-s <similitud>`. Si `<similitud>` toma el valor `1` entonces el programa utilizará las funciones de similitud especificadas por el argumento `m` sobre los atributos especificados por el argumento `a`, usando los valores de delta dados por el argumento `v`. En otro caso se realiza la inferencia de dependencias funcionales exactas.
- `-a <lista_atributos>`. Cuando el argumento `-s` indica que deben usarse funciones de similitud, el argumento `-a` junto con la cadena `<lista_atributos>` indican sobre que atributos deben utilizarse las funciones de similitud. La cadena `<lista_atributos>` es una lista separada por comas de enteros que representan los índices de los atributos en el archivo `*.csv` de entrada.
- `-m <lista_funciones>`. Cuando el argumento `-s` indica que deben usarse funciones de similitud, el argumento `-m` junto con la cadena `<lista_funciones>` se usan para indicar sobre cuales son las funciones de similitud que deben ser utilizadas. La cadena `<lista_funciones>` es una lista separada por comas de caracteres que determinan la función de similitud para el atributo correspondiente en `lista_atributos`. El caracter `l` indica que se debe usar la distancia de Levenshtein, el caracter `w` representa la distancia de Minkowski, el caracter `d` indica el uso de la diferencia en días entre dos fechas. Finalmente el caracter `t` representa la diferencia en minutos entre dos horas.
- `-v <lista_deltas>`. Cuando se usan funciones de similitud, el argumento `-v` se usa para indicar cuales son los valores de cada δ que deben ser utilizados. En ese sentido, `<lista_deltas>` es una lista separada por comas de reales, cada real representa la delta que corresponden a los atributos y funciones dadas por `lista_atributos` y `lista_funciones` respectivamente.

5. Ejemplos. Para los siguientes ejemplos, asuma un archivo de entrada llamado `test.csv` con los siguientes datos:

```
String,Time,Date,Coordinate
DEW,22:30,1/1/2016,-99.051253 19.488201
RKP,0:05,1/2/2016,-99.033186 19.405685
EVR,0:00,1/10/2016,-99.037534 19.401316
KQC,21:30,11/3/2016,-99.024387 19.408049
```

   En dicho archivo el nombre de cada atributo es indexado con un entero que empieza en cero, *i. e.*, el atributo `String` tiene el índice 0, y el atributo `Coordinate` tiene el índice 3.

   - Inferencia de dependencias funcionales exactas:

```
./DependencyMiner -f test.csv -o salida
```

Este comando producirá los archivos `test.txt`, `test_unsorted.txt` y `test_sorted.txt`.

- Inferencia usando una función de similitud:

```
./DependencyMiner -f test.csv -o salida -a 3 -m w -v 1.25
```

El siguiente comando realizará la inferencia de dependencias funcionales usando como función de similitud la distancia de Minkowski para el atributo `Coordinate` usando $\delta = 1.25$. Para los demás atributos se usará una relación de equivalencia.

- Inferencia usando varias funciones de simlitud:

```
./DependencyMiner -f test.csv -o salida -a 0,2,3
                  -m l,d,w -v 1,5,1.25
```

El comando anterior realizará la inferencia de DFs usando tres funciones de similitud. Para el atributo `String` se usará la distancia de Levenshtein con $\delta = 1$. Para el atributo `Date` se usará la diferencia en días entre dos fechas con $\delta = 5$. Finalmente para el atributo `Coordinate` se usará la distancia de Minkowski con $\delta = 1.25$.

Bibliografía

[1] K. Guadalupe-Álvarez, V. Ortega-Palma, S. Urbano-Serrano, J. Mercado-Vilchis, M. Alcántara-Valencia, V. M. Soto-Gómez, J. Hilario-Valencia, G. Román-Hernández, M. M. Garfias-González, y D. A. Vega-Martínez, "Informe de Indicadores Delictivos en el Estado de Mexico Abril-Junio 2016," Instituto de Estudios Legislativos, Toluca, Estado de México, Rep. Téc., 2016.

[2] J. C. Schlimmer, "Using learned dependencies to automatically construct sufficient and sensible editing views," en *KDD93: Workshop on Knowledge Discovery in Databases. AAAI.* Menlo Park, California: AAAI Press, 1993, pp. 186–196. [En línea]. Disponible: https://www.aaai.org/Library/Workshops/1993/ws93-02-017.php

[3] L. Berti-Équille, H. Harmouch, F. Naumann, N. Novelli, y S. Thirumuruganathan, "Discovery of genuine functional dependencies from relational data with missing values," *Proceedings of the VLDB Endowment*, Vol. 11, no. 8, pp. 880–892, Abr. 2018. [En línea]. Disponible: http://dl.acm.org/citation.cfm?doid=3204028.3228390

[4] M. W. Vincent, J. Liu, y C. Liu, "Redundancy Free Mappings from Relations to XML," en *Advances in Web-Age Information Management*, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, Q. Li, G. Wang, y L. Feng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, Vol. 3129, pp. 346–356. [En línea]. Disponible: http://link.springer.com/10.1007/978-3-540-27772-9_35

[5] B. P. Buckles y F. E. Petry, "A fuzzy representation of data for relational databases," *Fuzzy Sets and Systems*, Vol. 7, no. 3, pp. 213–226, May 1982. [En línea]. Disponible: https://linkinghub.elsevier.com/retrieve/pii/0165011482900525

[6] J. Kivinen y H. Mannila, "Approximate inference of functional dependencies from relations," *Theoretical Computer Science*, Vol. 149, no. 1, pp. 129–149, Sep. 1995. [En línea]. Disponible: https://linkinghub.elsevier.com/retrieve/pii/030439759500028U

[7] Y. Huhtala, J. Kärkkäinen, P. Porkka, y H. Toivonen, "Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies," *The Computer Journal*, Vol. 42, no. 2, pp. 100–111, Ene. 1999. [En línea]. Disponible: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/42.2.100

[8] S. Kruse y F. Naumann, "Efficient discovery of approximate dependencies," *Proceedings of the VLDB Endowment*, Vol. 11, no. 7, pp. 759–772, Mar. 2018. [En línea]. Disponible: http://dl.acm.org/citation.cfm?doid=3192965.3228335

[9] P. Mandros, M. Boley, y J. Vreeken, "Discovering Reliable Approximate Functional Dependencies," en *Proceedings of the 23rd ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining - KDD '17. Halifax, NS, Canada: ACM Press, 2017, pp. 355–363. [En línea]. Disponible: http://dl.acm.org/citation.cfm?doid=3097983.3098062

[10] S. Song y L. Chen, "Efficient discovery of similarity constraints for matching dependencies," *Data & Knowledge Engineering*, Vol. 87, pp. 146–166, Sep. 2013. [En línea]. Disponible: https://linkinghub.elsevier.com/retrieve/pii/S0169023X13000700

[11] W. Fan, "Dependencies revisited for improving data quality," en *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '08. Vancouver, Canada: ACM Press, 2008, pp. 159–170. [En línea]. Disponible: http://portal.acm.org/citation.cfm?doid=1376916.1376940

[12] S. Cha, "Comprehensive Survey on Distance/Similarity Measures between Probability Density Functions," *International Journal of Mathematical models and Methods in Applied Sciences*, Vol. 1, no. 4, pp. 300–307, 2007. [En línea]. Disponible: http://www.naun.org/main/NAUN/ijmmas/mmmas-49.pdf

[13] S. Song y L. Chen, "Differential dependencies: Reasoning and discovery," *ACM Transactions on Database Systems*, Vol. 36, no. 3, pp. 1–41, Ago. 2011. [En línea]. Disponible: http://dl.acm.org/citation.cfm?doid=2000824.2000826

[14] N. Koudas, A. Saha, D. Srivastava, y S. Venkatasubramanian, "Metric Functional Dependencies," en *2009 IEEE 25th International Conference on Data Engineering*. Shanghai, China: IEEE, Mar. 2009, pp. 1275–1278. [En línea]. Disponible: http://ieeexplore.ieee.org/document/4812519/

[15] R. Bassée y J. Wijsen, "Neighborhood Dependencies for Prediction," en *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, D. Cheung, G. J. Williams, y Q. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, Vol. 2035, pp. 562–567. [En línea]. Disponible: http://link.springer.com/10.1007/3-540-45357-1_59

[16] "¿Quiénes somos?" [En línea]. Disponible: https://violenciaypaz.colmex.mx/index.php/presentacion/quienes-somos

[17] SEDESOL, "Informe anual sobre la situación de pobreza y rezago social. México, 2014," Subsecretaría de prospectiva, planeación y evaluación, Rep. Téc., 2014.

[18] "What Does Sociodemographic Mean? | Reference.com." [En línea]. Disponible: https://www.reference.com/world-view/sociodemographic-mean-cc3b7119d0e7aefe

[19] C. Hernández-Alavez y R. Murcio-Villanueva, "Estudio de Indicadores ONU-HÁBITAT para los Observatorios Urbanos Locales de las Ciudades Mexicanas," SEDESOL, ONU-HÁBITAT México, Rep. Téc., 2004.

[20] INEGI, "Principales resultados del Censo de Población y Vivienda 2010. México," Instituto Nacional de Estadística y Geografía, Aguscaliente, Aguascalientes, Rep. Téc., 2011.

[21] M. Bassols-Ricardez y M. Espinoza-Castillo, "Construcción social del espacio urbano: Ecatepec y Nezahualcóyotl. Dos gigantes del oriente," *Revista Polis*, Vol. 7, no. 2, pp. 181–212, 2011.

[22] R. S. Ríos-Ortega y A. Varela-Ramos, "Plan de Desarrollo Municipal 2016 - 2018 Ecatepec de Morelos, Estado de México," H. Ayuntamiento Constitucional de Ecatepec de Morelos, Rep. Téc. 01238/10, Mar. 2016.

[23] H. Ayuntamiento Constitucional de Nezahualcóyotl, "Plan de Desarrollo Municipal 2016 - 2018 Nezahualcóyotl, México," H. Ayuntamiento Constitucional de Ecatepec de Morelos, Nezahualcóyotl, Estado de México, Rep. Téc., Mar. 2016.

[24] D. Dua y C. Graff, "UCI Machine Learning Repository," 2017. [En línea]. Disponible: http://archive.ics.uci.edu/ml

[25] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, Vol. 13, no. 6, pp. 377–387, Jun. 1970.

[26] K. V. Iyer, "An Introduction to Functional dependency in Relational Databases," *arXiv:1103.4979 [cs]*, Vol. abs/1103.4979, Mar. 2011, arXiv: 1103.4979. [En línea]. Disponible: http://arxiv.org/abs/1103.4979

[27] J. Demetrovics, L. Libkin, y I. B. Muchnik, "Functional dependencies in relational databases: A lattice point of view," *Discrete Applied Mathematics*, Vol. 40, no. 2, pp. 155 – 185, 1992. [En línea]. Disponible: http://www.sciencedirect.com/science/article/pii/0166218X92900289

[28] M. Y. Vardi, *Fundamentals of dependency theory*. California, USA: Comp. Sci. Press, 1987. [En línea]. Disponible: https://www.cs.rice.edu/~vardi/papers/ttcs87.pdf

[29] A. Silberschatz, H. F. Korth, y S. Sudarshan, *Database System Concepts*, 6.ª ed. New York, New York: McGraw-Hill Education, 2010.

[30] C. H. LeDoux y D. S. Parker, Jr., "Reflections on Boyce-Codd Normal Form," en *Proceedings of the 8th International Conference on Very Large Data Bases*, ser. VLDB '82. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1982, pp. 131–141.

[31] J. M. Hellerstein, "Quantitative Data Cleaning for Large Databases," 2008, UC Berkeley. [En línea]. Disponible: http://db.cs.berkeley.edu/jmh/papers/cleaning-unece.pdf

[32] L. Li, "Data quality and data cleaning in database applications," Tesis doctoral, Edinburgh Napier University, Edinburgh, Scotland, Sep. 2012.

[33] J. I. Maletic y A. Marcus, "Data Cleansing: A Prelude to Knowledge Discovery," en *Data Mining and Knowledge Discovery Handbook*, 2.ª ed., O. Maimon y L. Rokach, Eds. Boston, MA: Springer US, 2009, pp. 19–32. [En línea]. Disponible: http://link.springer.com/10.1007/978-0-387-09823-4_2

[34] J. Liu, J. Li, C. Liu, y Y. Chen, "Discover Dependencies from Data—A Review," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 24, no. 2, pp. 251–264, Feb. 2012. [En línea]. Disponible: http://ieeexplore.ieee.org/document/5601725/

[35] Z. Abedjan, P. Schulze, y F. Naumann, "DFD: Efficient Functional Dependency Discovery," en *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management - CIKM '14*. Shanghai, China: ACM Press, 2014, pp. 949–958. [En línea]. Disponible: http://dl.acm.org/citation.cfm?doid=2661829.2661884

[36] W. Fan, F. Geerts, J. Li, y M. Xiong, "Discovering Conditional Functional Dependencies," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 23, no. 5, pp. 683–698, May 2011. [En línea]. Disponible: http://ieeexplore.ieee.org/document/5560658/

[37] P. Bohannon, W. Fan, F. Geerts, X. Jia, y A. Kementsietsidis, "Conditional Functional Dependencies for Data Cleaning," en *2007 IEEE 23rd International Conference on Data Engineering*. Istanbul: IEEE, Abr. 2007, pp. 746–755. [En línea]. Disponible: http://ieeexplore.ieee.org/document/4221723/

[38] M. Y. Bilenko, "Learnable similarity functions and their applications to record linkage and clustering." University of Texas at Austin, Texas, US, Rep. Téc. UT-AI-TR-03-305, 2003. [En línea]. Disponible: https://repositories.lib.utexas.edu/bitstream/handle/2152/2681/bilenkom45298.pdf

[39] L. Rhodes, "Similarity and Dissimilarity," Dic. 2018, juniata College. [En línea]. Disponible: http://jcsites.juniata.edu/faculty/rhodes/ml/simdissim.htm

[40] L. Golab, H. Karloff, F. Korn, A. Saha, y D. Srivastava, "Sequential dependencies," *Proceedings of the VLDB Endowment*, Vol. 2, no. 1, pp. 574–585, Ago. 2009. [En línea]. Disponible: http://dl.acm.org/citation.cfm?doid=1687627.1687693

[41] C. Matheus, P. Chan, y G. Piatetsky-Shapiro, "Systems for knowledge discovery in databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, no. 6, pp. 903–913, Dic. 1993. [En línea]. Disponible: http://ieeexplore.ieee.org/document/250073/

[42] I. H. Witten, E. Frank, M. A. Hall, y C. J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*, 4.ª ed., ser. The Morgan Kaufmann Series in Data Management Systems. San Francisco, California: Elsevier Science, Nov. 2016.

[43] H. Mannila y K.-J. Räihä, "On the complexity of inferring functional dependencies," *Discrete Applied Mathematics*, Vol. 40, no. 2, pp. 237 – 243, 1992. [En línea]. Disponible: http://www.sciencedirect.com/science/article/pii/0166218X92900315

[44] H. Yao y H. J. Hamilton, "Mining functional dependencies from data," *Data Mining and Knowledge Discovery*, Vol. 16, no. 2, pp. 197–219, Abr. 2008. [En línea]. Disponible: http://link.springer.com/10.1007/s10618-007-0083-9

[45] H. Yao, H. Hamilton, y C. Butz, "FD_mine: discovering functional dependencies in a database using equivalences," en *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* Maebashi City, Japan: IEEE Comput. Soc, 2002, pp. 729–732. [En línea]. Disponible: http://ieeexplore.ieee.org/document/1184040/

[46] A. A. Chavan y V. K. Verma, "Mining Functional Dependency in Relational Databases using FUN and Dep-Miner: A Comparative Study," *International Journal of Computer Applications*, Vol. 78, no. 15, pp. 34–36, Sep. 2013. [En línea]. Disponible: https://research.ijcaonline.org/volume78/number15/pxc3891377.pdf

[47] Y. Huhtala, J. Karkkainen, P. Porkka, y H. Toivonen, "Efficient discovery of functional and approximate dependencies using partitions," en *Proceedings 14th International Conference on*

Data Engineering. Orlando, FL, USA: IEEE Comput. Soc, 1998, pp. 392–401. [En línea]. Disponible: http://ieeexplore.ieee.org/document/655802/

[48] J. Motl y O. Schulte, "The CTU Prague Relational Learning Repository," *arXiv:1511.03086 [cs]*, Nov. 2015, arXiv: 1511.03086. [En línea]. Disponible: http://arxiv.org/abs/1511.03086

[49] M. Bilenko, "Duplicate Detection, Record Linkage, and Identity Uncertainty: Datasets," Ago. 2003. [En línea]. Disponible: https://www.cs.utexas.edu/users/ml/riddle/data.html

[50] S. Tejada, C. A. Knoblock, y S. Minton, "Learning object identification rules for information integration," *Information Systems*, Vol. 26, no. 8, pp. 607–633, Dic. 2001. [En línea]. Disponible: https://linkinghub.elsevier.com/retrieve/pii/S0306437901000424

[51] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, y F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," en *Proceedings of the VLDB Endowment*, Vol. 8. Kohala Coast,Hawaii: VLDB Endowment Inc., 2015, pp. 1082–1093. [En línea]. Disponible: http://www.vldb.org/pvldb/vol8/p1082-papenbrock.pdf

[52] N. Asghar y A. Ghenai, "Automatic Discovery of Functional Dependencies and Conditional Functional Dependencies: A Comparative Study," Abr. 2015, university of Waterloo. [En línea]. Disponible: https://cs.uwaterloo.ca/~nasghar/848.pdf

[53] R. Prajapati y V. K. Verma, "Improving TANE Algorithm to Reduce Dependency and Search Space," *International Journal of Science Technology Managementand Research*, Vol. 3, no. 6, pp. 31–34, 2018. [En línea]. Disponible: http://www.ijstmr.com/wp-content/uploads/2018/06/IJSTMR_V3I6_08181.pdf

[54] L. Bohong y J. Xinyuan, "An Improved DFD Based on Attribute Partition Information Entropy," en *2018 4th International Conference on Systems, Computing, and Big Data (ICSCBD 2018)*. Guangzhou, China: Francis Academic Press, UK, 2018, pp. 66–69. [En línea]. Disponible: https://www.webofproceedings.org/proceedings_series/ECS/ICSCBD%202018/ICSCBD22011.pdf

[55] Z. M. Marquez-Navarrete, "DataCleaning," 2018. [En línea]. Disponible: https://github.com/TsukiZombina/DataCleaning

[56] ——, "DependencyMiner," 2019. [En línea]. Disponible: https://github.com/TsukiZombina/DependencyMiner