IIC2440 – Procesamiento de Datos Masivos

Técnicas Probabilísticas

1. Motivación

Hasta ahora, para trabajar con datos, hemos usado bases de datos relacionales, almacenes de datos, y bases de datos distribuídas, bajo la suposición de que el trabajo constante con índices o con múltiples particiones nos da buenos resultados en cuanto a eficiencia.

Hoy, abandonamos esa suposición. Vamos a conocer dos técnicas adicionales para el trabajo de datos: *sampling*, y *hashing*. Estas técnicas son probabilistas: van a tener una cierta probabilidad de no funcionar (veremos en un momento lo que significa no funcionar). A cambio, nos entregan una velocidad de procesamiento muchísimo mayor.

2. Sampling

La idea es sencilla: si no puedo trabajar con todos los datos, mejor tomo un subconjunto de ellos, los trabajo, y después espero que la respuesta que obtuve generalize a todos los datos. El truco, por supuesto, consiste en diseñar una estructura de sampling que me permita una buena generalización, o bien que tenga una probabilidad baja de fallo.

Vamos a ver esta técnica aplicada a un algoritmo para minear *reglas de asociación*, un tema que es interesante también por si mismo.

2.1. Reglas de Asociación

Imaginemos una base de datos de compras de supermercado. Tenemos un conjunto E de elementos. Y registramos cada compra $C \subseteq E$ que hace cada persona (por ahora no identificamos quién hace la compra).

Lo que nos interesa es poder descubrir reglas de asociación

$$I \to j,$$

para un conjunto de elementos $I \subseteq E$ y un elemento $j \in E$. Estas reglas nos indican que, en las compras que tenemos registradas, al observar la presencia de todos los elementos de I en una compra, la presencia del elemento j es altamente problable.

Por ejemplo, una regla de asociación clásica es que la compra de pañales es bien probable que se asocie con cervezas, es decir, que tenemos la regla

$$\text{pañales} \to \text{cerveza}.$$

Veamos esto más formal. Para un conjunto E de elementos y canastas $C_1, \ldots, C_n, C_i \subseteq E$, definimos el *soporte* de un conjunto S como la cantidad de canastas que contienen a los elementos S, es decir, $|\{C_i \mid S \subseteq C_i\}|$. El *soporte de una regla* $I \to j$ es el soporte de $I \cup \{j\}$.

La *confianza* de una regla $I \to j$ es el soporte de $I \cup \{j\}$ dividido por el soporte de I. La confianza nos dice cuál es la proporción de canastas en las que aparece I y también j y las que aparece I. Una confianza de 1 indica que cada vez que aparece I aparece también j, una confianza de $0,5$ nos dice que solo en la mitad de las canastas en las que aparece I aparece j, y así.

El problema de mineo de reglas de asociación es el siguiente:

> Dados un conjunto de elementos E, canastas $C_1, \ldots, C_n$, $C_i \subseteq E$ y valores s, c en $[0, 1]$, encontrar todas las reglas de asociación que tengan soporte al menos sn y confianza al menos c.

2.2. Algoritmo Apriori

Este algoritmo nos sirve para minear todas las reglas de asociación con una cierta confianza y cierto soporte. Es un algoritmo lento, por que minear todas las reglas tiene que y debe de ser un proceso lento, pero está bien pensado: las reglas se van mineando de forma incremental de manera de asegurar que nunca se realiza una operación que sabemos que no va a ser necesaria.

Para un conjunto E de elementos, canastas $C_1, \ldots, C_n$ de elementos en E, confianza c y soporte s, el algoritmo funciona así.

Pasada inicial:

1. Comenzar por calcular el soporte de todos los conjuntos e_j de un solo elemento de E. Esto se almacena en una tabla[1].

2. Se descartan todos los conjuntos que tienen soporte menor a sn.

Pasada $\ell + 1$, asumiendo que el output de la pasada ℓ es una tabla con el soporte de todos los conjuntos de ℓ elementos:

1. Calcular el soporte para todos los conjuntos de $\ell+1$ elementos que puedo hacer uniendo dos conjuntos de ℓ elementos que estén en el resultado de la pasada ℓ. Es decir, si la pasada ℓ tiene conjuntos S, S' tal que $|S \cup S'| = \ell + 1$, entonces se calcula el soporte para $S \cup S'$.

2. Calcular la confianza de todas las reglas de la forma $S \cup S' \to e$.

3. Descartar todos los conjuntos que no alcanzan el soporte sn, y todas las reglas que no alcanzan la confianza c. El output es una nueva tabla con todos los soportes de conjuntos de $\ell + 1$ elementos, y todas las reglas que fueron computadas a partir de estos conjuntos.

[1]Para hacer más eficiente el almacenamiento, uno debería partir por construir un diccionario que le asigne un entero entre 1 y $|E|$ a cada elemento, y luego esta tabla va a ser simplemente un arreglo con $|E|$ entradas, la entrada j es el soporte del elemento j

El algoritmo apriori es correcto y completo. Hace el menor esfuerzo posible, por que una vez que un conjunto S no alcanza el soporte sn, nunca vuelve a computar el soporte para un superconjunto de S (lo que esta bien, por que nunca va a poder tener soporte mayor a sn). Pero es lento. En cada iteración ℓ del algoritmo debemos pasar por completo por todas las canastas $C_1, \ldots, C_n$.

2.3. Sampling al rescate

Una primera estrategia Una estrategia de sampling simple para realizar apriori consiste en lo siguiente. Tomamos un sample de n/k canastas (por ejemplo, si tenemos $n = 10^8$ canastas podemos tomar un sample de 10^6 canastas haciendo $k = 100$, lo que equivale a un $1\,\%$). Notar que el soporte que buscamos debe reducirse por que tenemos menos elementos. Si antes buscamos regla con soporte mayor a sn, ahora buscaremos reglas con soporte mayor a sn/k, simplemente por que nuestro nuevo dataset es k veces mas pequeño.

Ahora calculamos apriori con este sample. El resultado serán reglas de asociación que pueden tener errores, a saber:

- Falsos positivos, que son reglas $I \to j$ que tienen suficiente soporte y confianza en el sample, pero no así en la base de datos entera de n canastas. Entonces, son reglas en el output que no deberían estar ahí.
- Falsos negativos, que son reglas $I \to j$ que no tienen suficiente soporte y confianza en el sample, pero sí en la base de datos entera de n canastas. Son reglas que no aparecen en el output, a pesar de que deberían aparecer.

Ejercicio. Piensa en como podríamos reducir o eliminar estos errores. Entrega una respuesta distinta para falsos positivos y falsos negativos: ¿es posible eliminarlos realizando operaciones posteriores? Si no, ¿cómo podríamos mitigarlos?

Algoritmo de Toivonen. Este algoritmo no produce ni falsos positivos ni negativos, pero tiene una posibilidad de error. Partimos por computar el borde positivo y el borde negativo de una regla.
Borde positivo. Si buscamos soporte sn, el borde positivo de un sample de tamaño n/k son todas las reglas que tienen soporte psn/k, para un $p \leq 1$. Al bajar el soporte a una proporción menor (usando p), estamos tratando de conseguir más reglas (el soporte límite es menor por lo tanto hay más reglas que potenciamente tienen soporte mayor a nuestro límite).
Borde negativo. El borde negativo son todos los elementos o reglas que no están en el borde positivo, pero que todos sus subconjuntos o sub reglas inmediatas si lo están. Es decir, reglas tipo $I \to j$ en donde todos los subconjuntos de $I \cup \{j\}$ con un elemento menos si pertenecen al borde positivo.

Ahora damos una pasada por toda la base de datos de Canastas, buscando el soporte de todos los elementos en el borde positivo y el borde negativo. Nuestro output van a ser todas las reglas del soporte positivo que tengan soporte mayor a sn en el dataset general. Pero, pueden pasar dos cosas.

- No existe ningún conjunto o regla del borde negativo con soporte mayor a sn en todas las canastas. En este caso, el algoritmo funciona correctamente y encuentra todas las reglas, sin errores falsos positivos ni falsos negativos.
- Existe al menos un elemento en el borde negativo que con soporte mayor a sn cuando lo calcuamos en todo el dataset de canastas. En este caso, no podemos estar seguros de nuestra respuesta, y debemos repetir el algoritmo con otro sample aleatorio.

Ejercicio. ¿Por qué funciona el algoritmo de Toivonen?

Ejercicio. Volvamos a pensar en distribución. Imaginemonos que no podemos almacenar cada canasta en una máquina, si no que almacenamos todo en k máquinas distintas. En ese contexto, podríamos pedirle a cada máquina que procesara cada una de esas n/k canastas. Diseña un algoritmo tipo apriori que agrupe el resultado de cada una de esas máquinas, y entregue un conjunto de reglas de asociación que asegure que no hay falsos negativos.

3. Hashing

Hashing es una forma de trabajo distinta al sampleo. En sampling, en vez de trabajar con una base de datos D, trabajamos con un sample $d \subsetneq D$ mucho más chico. En hashing, usamos funciones para transformar todo el espacio de datos D y mapearlo a un espacio más pequeño.

3.1. Funciones de hash

En términos generales, una función de hash mapea inputs de tamaño arbitrario a outputs de tamaño fijo.

Así, considerando nuestros outputs como palabras de b bits, podemos pensar que una función de hash *indexa* cada input w, asignándole una palabra específica de b bits.

Ejemplo. Tal vez la función de hash más conocida es la función *módulo* $f(x) = x \mod n$. Esta función mapea números arbitrariamente grandes al espacio $[0, n-1]$. Por supuesto, el principio del palomar implica que, al procesar números mayores a n, necesariamente deben existir números m_1, m_2 tal que $f(m_1) = f(m_2)$. A esto le llamamos una *colisión*.

Ejemplo. Python tiene un módulo *hash()* que entrega un entero de 64 bits para cada objeto distinto. Python, además, tiene una librería llamada hashlib con una gran variedad de funciones de hash con propiedades bastante mejores.

En este curso no vamos a preocuparnos de la naturaleza de las funciones de hash que usemos. Nos basta con asumir que las colisiones están bien esparcidas en el espacio de datos.

3.2. Aplicación: Filtrado en streams

En *streaming* , nos interesa generar estadísticas sobre mensajes o requests que nos lleguen a nuestro sistema. Pero asumimos que la cantidad de información que nos llega es de tal magnitud, que con suerte tenemos para almacenar un bajo porcentaje de esta.

Ejercicio. Imagina que gestionas una página web, que recibe millones de requests. En cada día, solo puedes almacenar al mismo tiempo un 10 % de los requests que te llegan. Te gustaría poder razonar sobre la siguiente estadística: es verdad que si un usuario ejecuta un request r, posteriormente en el mismo día va a ejecutar un request r'? ¿Cómo podrías intentar resolver esto?

Filtros de Bloom. Un problema importante en la práctica consiste en filtrar datos dentro de un stream, ya sea por que tenemos un firewall que solo deja pasar ciertos datos, o por que para algunos de estos datos vamos a iniciar un cierto proceso, pero para la mayoría no.

Consideremos, por ejemplo, un sistema con nombres de usuario (usernames) y contraseñas. Como somos populares, nos llegan cientos de intentos de logins por minuto. Los usuarios y sus passwords están almacenados en una tabla (que debería estar encriptada, pero para efectos de esta clase eso no importa). Por cada request de login que nos llega, tenemos que ir a esa tabla a verificar si el usar corresponde con el password. Pero, el solo hecho de ver si un usuario está en nuestra base de datos nos toma tiempo!

Para poner números, supongamos que tenemos N igual a mil millones de usuarios, alrededor de 2^{30}. Si nuestra tabla está bien indexada, necesitamos alrededor de $\log(N)$ operaciones, o unas 30, para comprobar si un username está o no en la tabla. Realizar 30 operaciones está ok en muchos contextos, pero es demasiado para un contexto donde llegan cientos de logins al segundo.

Una buena solución es usar un filtro de Bloom. Usando un espacio de 1 gb, o alrededor de 8 mil millones de bits, hasheamos cada uno de los N usuarios usando una función de hash h que nos lleve cada string a un número i entre 0 y 8 mil millones. Luego construimos un arreglo donde la i-esima posición tiene un 1 si y solo si ese i resulta ser la imagen $h(s)$ de un hash de uno de nuestros usuarios, un 0 en las otras posiciones.

Cuando llega un usuario tentativo, digamos $u =$ `jreutter@ing.puc.cl`, realizamos lo siguiente. Tomamos $h(u)$, y eso nos da un número i entre 0 y 8 mil millones.

- Si a ese número le fue asignado un 0 en nuestro arreglo, entonces sabemos que ese usuario no está en nuestra base de datos.
- Si a ese número le fue asignado un 1, lo tenemos que ir a buscar a nuestra base de datos, por que podría estar (podría también no estar, ¿puedes ver por que?).

¿Qué hemos ganado con todo esto? De forma aproximada, si `jreutter@ing.puc.cl` no era parte de nuestros N usuarios, vamos a tener que ir a buscarlo igual con una probabilidad de 1/8: son 8 mil millones de entradas, y solo mil millones de usuarios, por lo que hay a lo mas mil millones de unos en nuestro arreglo: la probabilidad de que la función de hash nos

lleve a un 1 es, por ende, 1/8. Eso significa que con una pura operación de hash ya hemos descartado 7/8 de los usuarios que no están!

Podemos hacer más, pero veamos primero un mejor análisis de la probabilidad de que un usuario que no está sea asignado un 1. No es exactamente 1/8, pues los hashes pueden colisionar: dos usuarios en nuestro grupo N podrían ir a parar a la misma posición del arreglo.

Ahora, si tenemos B baldes o posiciones en nuestro arreglo, y tomamos un usuario fijo, la posibilidad que uno de los B baldes sea la imagen del hash de ese usuario es $1/B$, y que no sea la imagen de un usuario es $1 - 1/B$. Como cada hash es independiente del anterior la probabilidad que un balde no sea la imagen de alguno de los N usuarios es $(1 - 1/B)^N = (1 - 1/B)^{\frac{BN}{B}}$. Usando que $(1 - 1/\epsilon)^\epsilon = 1/e$, cuando ϵ es pequeño, la probabilidad nos queda como $e^{-\frac{N}{B}}$.

En nuestro caso, la probabilidad que alguno de los bits si sea seteado a 1 es $1 - e^{-\frac{N}{8N}} = 1 - e^{-\frac{1}{8}} \approx 0{,}118$

Filtros de Bloom anidados. A pesar de nuestro avance, 1/8 o incluso 0,118 significa que para más del 10 % de usuarios que no están en N vamos a tener que ir a buscarlos a la base de datos igual, lo que todavía puede ser costoso. Pero podemos reducir esa priobabilidad usando varios filtros de Bloom!

> Un filtro de bloom para N strings con H hashes y B baldes funciona de la siguiente forma. Para cada string s en N, y para cada h en H, setear en 1 el balde correspondiente a $h(s)$, es decir, hacer $B[h(s)] = 1$.
>
> Cuando llega un nuevo string, el filtro lo marca como un potencial elemento en N cuando para cada h en H se tiene que $B[h(s)]$ es 1.

Volviendo al ejemplo anterior, si ahora tenemos dos funciones de hash mapeando al mismo conjunto de B baldes, la probabilidad que un balde no tenga un 1 ahora es $e^{-\frac{2N}{B}}$, simplemente por que la segunda función de hash es independiente de la primera, y entonces para ver que baldes tienen un 1, equivalente a tener el doble de usuarios.

Pero ahora cuando nos llega un usuario nuevo, vamos a ir a mirarlo si y solo si *ambas* funciones de hash nos mandan a un 1! En este caso, la probabilidad de que ambas funciones de hash entreguen un uno para un nuevo string corresponde a la probabilidad que ambos baldes a los que se hashean esos strings entreguen un 1. Esto se aproxima como $(1 - e^{-\frac{2N}{B}})^2$, o $(1 - e^{-\frac{1}{4}})^2$ en nuestro caso, lo que es aproximadamente 0,05. Usando una segunda función de hash ¡reducimos el número de usuarios que tenemos que mirar a más de la mitad!