Práctica 1b
Técnicas de Búsqueda Local y Algoritmos Greedy
Problema de Agrupamiento con Restricciones

Sergio Quijano Rey - 72103503k
4º Doble Grado Ingeniería Informática y Matemáticas
Grupo de prácticas 2 - Viernes 17.30h a 19.30h
sergioquijano@correo.ugr.es

10 de abril de 2021

Índice

1. Descripción del problema

Vamos a trabajar el problema del agrupamiento con restricciones (***PAR***). Consiste en una generalización del problema de agrupamiento clásico, al que añadimos restricciones sobre los datos.

El problema de agrupamiento clásico consiste en, dados unos datos de entrada sin etiquetar X de tamaño n, agruparlos en k grupos (o en inglés, *clusters*) diferentes, formando una partición C de X, de forma que se optimice alguna métrica. Normalmente, se busca minimizar la distancia *intra_cluster* (que más tarde se definirá).

La diferencia con el problema de agrupamiento clásico, por tanto, es la inclusión de restricciones. En nuestro caso concreto, trabajermos con restricciones entre pares de puntos, que además serán de dos tipos:

- Restricción tipo *Must Link*: los dos puntos afectados por esta restricción deberán pertenecer al mismo cluster
- Restricción tipo *Cannot Link*: los dos puntos afectados por esta restricción no deben pertenecer al mismo cluster

Consideraremos de forma débil estas restricciones, es decir, podemos incumplir algunas restricciones. Pero supondrá que la solución será de peor calidad. Para especificar mejor esta noción, definimos la función de *fitness* que buscamos minimizar:

$$fitness(sol) := distancia_{intra-cluster}(sol) + \lambda * infeasibility(sol)$$

donde $infeasibility$ es el número de restricciones que se incumplen. Esta función de $fitness$ nos permite pasar de intentar optimizar dos funciones objetivo a solo tener que optimizar un objetivo. El valor de λ se especifica más adelante.

Como los datos no están etiquetados a priori, podríamos considerar este problema como un problema de aprendizaje no supervisado. Sin embargo, se puede considerar que las restricciones nos dan un tipo de etiquetado, por lo que es más correcto pensar que estamos ante una tarea de aprendizaje

semi-supervisado. La principal utilidad de resolver estos problemas es que normalmente estamos reduciendo la dimensionalidad de los datos a analizar, y de este modo, es más sencillo extraer conocimiento sobre dichos datos. También es útil para realizar

2. Descripción de la aplicación de los algoritmos empleados

2.1. Representación del conjunto de datos

Los datos vienen dados en una matrix de tamaño nxd donde n es el número de puntos y d es la dimensión de cada uno de los puntos.

2.1.1. Representación del conjunto de datos en código

El conjunto de datos viene representado en problem_datatypes::DataPoints que contiene un vector de otro tipo de dato: problem_datatypes::Point. El tipo de dato Point tiene un campo que es de tipo ndarray::Array1<f64> que representa un vector (usamos una librería para trabajar con matrices y vectores). Por tanto, hemos pasado de trabajar con una matriz de datos a trabajar con un vector de puntos. Esto nos permite trabajar de forma más expresiva y sencilla con el problema. Por ejemplo, podemos calcular con métodos del struct distancia entre dos puntos, centroide de un conjunto de puntos, ...

2.2. Representación de las restricciones

Las restricciones vienen dadas en un fichero de datos que representa una matriz que codifica las restricciones de la siguiente forma:

- El elemento en la fila i-ésima y columna j-ésima representa las restricciones que hay entre el punto i y el punto j
- Como la restricción que tenga el punto i con el punto j implica que el punto j tiene la misma restricción con el punto i, es claro que dicha matriz debe ser simétrica
- Un valor 0 significa que no hay restricciones. Un valor 1 significa que hay una restricción tipo *Must Link*. Un valor -1 implica una restricción *Cannot Link*
- Además, la matriz tiene la diagonal de 1s

2.2.1. Representación de las restricciones en código

El struct problem_datatypes::Constraints junto al enumerado problem_datatypes::ConstraintType representan en el código las restricciones. El código es el siguiente:

```
pub enum ConstraintType {
    MustLink,
    CannotLink,
}

pub struct Constraints{
    data: HashMap<(i32, i32), ConstraintType>,
}
```

Es claro que guardamos pares de enteros, que marcan los índices de los puntos, y la restricción entre el par de puntos representados, en un HashMap. Esta elección viene motivada por:

- Podemos acceder a las restricciones entre dos puntos en tiempo constante
- Podemos iterar sobre todas las restricciones, gracias a los métodos proporcionados por el lenguaje de programación, en un tiempo más que razonable. Así iteramos solo sobre una lista de r restricciones, en vez de sobre una matriz cuadrada de dimensión n^2
- En cierto modo, estamos combinando los beneficios de tener acceso directo a elementos concretos y los beneficios de poder iterar sobre una lista (aunque iterar sobre un Hash puede ser algo más lento que iterar sobre una lista o un array)
- Es fácil de implementar métodos para operar con restricciones con este tipo de dato

La implementación de los métodos que permiten manipular el struct aseguran que:

- No guardamos la restricción (i, j) y junto a la (j, i). Solo guardamos una de las dos restricciones, ahorrando memoria

- De hecho, el criterio es guardar como índices el par (i, j) donde $i \leq j$
- Tampoco guardamos las restricciones $(i, i), MustLink$ pues son restricciones triviales

2.3. Representación de la solución

Una representación de la solución será un vector de tamaño n con valores en $\{0, \ldots, k-1\}$ donde n es el número de puntos y k es el número de clusters en los que dividimos los datos. Este vector representa la partición de los datos en los k clusters. En la posición i-ésima del vector, guardamos el cluster al que pertecene el punto i-ésimo de nuestro conjunto de datos.

Las soluciones deben cumplir las siguientes restricciones:

- No pueden quedar clusters vacíos. Es decir, clusters a los que no haya ningún punto asignado. Esto puede verse viendo que $\forall i \in \{0, \ldots, k-1\}$ $\exists pos \in \{0, \ldots, n-1\}$ tal que $solution[pos] = i$, es decir, el vector de soluciones tiene al menos una vez cada valor posible de los clusters
- Cada punto solo puede pertenecer a un único cluster. Por la forma vectorial en la que representamos la partición, esta restricción se verifica forzosamente, y por tanto no nos tenemos que preocupar de realizar comprobaciones
- La unión de los puntos de los clusters debe ser todo el conjunto de datos, es decir, $X = \bigcup c_i$. De nuevo, nuestra representación vectorial fuerza a que esta restricción se verifique

Por ejemplo, si tenemos 5 puntos y 3 clusters, una posible solución sería $\{3, 1, 2, 3, 0\}$. Y por otro lado, la solución $\{3, 1, 2, 3, 2\}$ no es válida pues el cluster 0 está vacío.

Para cada solución podemos calcular algunas métricas necesarias para conocer el valor de $fitness$ de la solución que estamos representando. Para comenzar, por cada cluster podemos calcular el **centroide** del cluster:

$$\vec{\mu_i} := \frac{1}{|c_i|} \sum_{x_i \in c_i} \vec{x_i}$$

Definimos para cada cluster su **distancia media intra-cluster** como:

$$\bar{c_i} := \frac{1}{|c_i|} \sum_{x_i \in c_i} ||\vec{x_i} - \vec{\mu_i}||_2$$

Y con ello podemos definir la **desviación general de la partición** como:

$$\bar{c} := \frac{1}{k} \sum_{i \in 1,...k} \bar{c_i}$$

Definimos $infeasibility$ como el número de restricciones, tanto del tipo *Must Link* como del tipo *Cannot Link*, que se violan.

Con ello, ya podemos definir el valor de λ como $\lambda := \frac{D}{|R|}$ donde $|R|$ es el número total de restricciones y D la distancia máxima entre dos puntos de X.

2.3.1. Representación de la solución en código

La solución se representa en la clase problem_datatypes::Solution. El código de los campos del struct desarrollado es:

```
pub struct Solution<'a, 'b> {
    cluster_indexes: Vec<u32>,
    data_points: &'a DataPoints,
    constraints: &'b Constraints,
    number_of_clusters: i32,

    /// Representa el peso de infeasibility en el calculo de fitness
    /// Solo se calcula una vez al invocar a Solution::new
    lambda: f64,

    // Para cachear el valor de fitness pues es un calculo costoso de realizar
    // Como los datos del struct no cambian, podemos hacer el cacheo sin miedo
    // Usamos RefCell para tener un patron de mutabilidad interior
    fitness: RefCell<Option<f64>>,
}
```

Los campos del struct representan:

- cluster_indixes : el vector solución que representa la asignación de puntos a clusters
- data_points: referencia al conjunto de datos (sirve para calcular métricas como el $fitness$ de la solución que se representa)
- constraints: referencia al conjunto de restricciones sobre los datos (sirve para calcular métricas como el $fitness$ de la solución que se representa)
- number_of_clusters: número de clusters en los que se agrupan los datos (sirve para comprobar que una solución sea válida)
- lambda: valor de λ, necesario para calcular el $fitness$
- fitness : valor de $fitness$. Está incluida en un RefCell<Option<f64>> para poder cachear su valor, puesto que los atributos de una instancia nunca cambian y el cálculo del valor λ es muy costoso (implica calcular restricciones violadas y distancias entre puntos)

La comprobación de que no tenemos clusters sin puntos asignados se hace en el método Solution :: is_valid . La distancia media intracluster se calcula en Solution :: intra_cluster_distance . Mientras que la desviación general se calcula en Solution :: global_cluster_mean_distance. El valor de $infeasibility$ se calcula en Solution :: infeasibility . El cálculo de λ se realiza en el *constructor* del struct.

3. Descripción de los algoritmos empleados - Máximo 2 páginas por algorimto

* Se hace en pseudocódigo

4. Pseudocódigo de los algoritmos de comparación - Máximo sin especificar

5. Explicación del procedimiento considerado para desarrollar la práctica - Máximo sin especificar

6. Experimentos y análisis realizados - Máximo sin especificar

6.1. Descripción de los casos del problema empleados

6.2. Resultados obtenidos según el formato especificado

6.3. Análisis de resultados

7. Bibliografía - Máximo sin especificar