Estructuras de datos y algoritmos

PROP Grupo 33
Víctor Muñoz, Alex Moa, Artur Farriols, Mélanie Scaténa
Cuatrimestre otoño
Curso 21-22

Introducción

Este documento incluye una sucinta descripción y justificación del uso de los algoritmos y las principales estructuras de datos empleadas en las clases que conforman la capa de dominio. Al tratarse de un trabajo iterativo e incremental, la información aquí proporcionada está sujeta al cambio en función de las modificaciones aconsejadas por el tutor o aquéllas decididas por los integrantes del grupo.

Algoritmos

Este apartado contiene la explicación y justificación de las decisiones tomadas para implementar los algoritmos. En la próxima entrega, se incluirá pseudocódigo con tal de facilitar su comprensión.

Collaborative Filtering

Técnica de filtrado consistente en predecir la valoración que un determinado sujeto realizaría sobre un objeto o ítem en base a las opiniones de otros individuos con gustos similares.

En nuestro sistema software, esta técnica consta de dos partes (*kmeans* y *Slope One*). Una vez cargados todos los datos necesarios para ejecutar el algoritmo (usuarios e ítems), es necesario crear un número *k* de usuarios que actúen a modo centroides, que contendrán unas valoraciones realizadas aleatoriamente. Posteriormente, será preciso calcular la distancia euclidiana entre los usuarios almacenados en el sistema y los centroides. Tras realizar dichos cálculos, cada usuario será asignado al centroide más próximo creando, de este modo, un conjunto de usuarios que recibirá el nombre de clúster. Entonces, se recalcularán las puntuaciones del centroide empleando, para ello, las valoraciones de los usuarios asociados al clúster. En caso de no disponer de ninguna valoración para un determinado ítem, se mantendrá el valor aleatorio original. Llegados a este punto, simplemente será necesario repetir este proceso (cálculo de distancias y asignación de usuarios) hasta que no se produzcan más cambios.

Finalizada la ejecución del algoritmo *kmeans*, dará inicio la del *Slope One*. Primeramente, se asignará el usuario para el que se realiza la recomendación a uno

de los clústers resultantes. Se proseguirá entonces calculando la valoración esperada en función de las puntuaciones del resto de usuarios que conforman el clúster. Finalmente, solo restará ordenar los valores obtenidos.

PseudoCodigo:

El algoritmo contiene 2 funciones públicas (preprocessing y query):

```
preprocessing(itemMap, userMap){ //kmeans
  initializeDataStructures();

  while (change) {
    for (users) {
      for (centroides) {
        calculateEuclideanDistance(user, centroid);
      }
      //Se asigna el usuario al centroide más próximo
    }
    if (UserHasChangedOfClúster == 0) change = false;
    else recalculateCentroids() //recalcula las valoraciones de los centroides.
  }
}

query( user, unknownItems, Q){ //Slope One
  for (centroids) {
    calculateEuclideanDistance(cluster, user);
  }
  //Se asigna el usuario user al clúster más próximo
  for (unknownItems) {
    subgroupOfUsers = getSubgroupOfUsers(unknownItem, cluster);
    rating = calculateRating(knownRatings, subgroupOfUsers, unknownItem, user);
    recommendedItems.add(unknownItem, rating);
  }
```

```
  searchNullValues(recommendedItems);
  return limitRecommendedItems(recommendedItems, Q);
}

initializeDataStructures (){
  initializeK(3); //Número de clústers
  initializeItems(ItemMap); //Conjunto de ítems del dataset
  initializeUsers(UserMap); //Conjunto de usuarios y sus respectivas valoraciones
   initializeClusters(); //Crea los centroides y clústers de usuarios. Cada centroide se corresponderá con un determinado clúster.
}

calculateEuclideanDistance(user, centroid){
//Calculates the Euclidean distance
}

recalculateCentroids(){
  for (centroids) {
    for (users) { //usuarios pertenecientes al clúster
      for (items) { //ítems pertenecientes al usuario
        itemAppearances += 1;
        itemRating += itemRate;
          //Pese a que ambas variables se encontrarían incluidas en un mapa, con el fin de facilitar la comprensión del pseudocódigo, se representan como variables simples.
      }
    }
    for (items) { //Conjunto de ítems del dataset
      centroidItemRating = itemRating / itemAppearances;
    }
  }
}

getSubgroupOfUsers(unknownItem, cluster){
```

//Obtiene el subconjunto de usuarios pertenecientes al clúster que han valorado el ítem unknown pasado como parámetro
}
calculateRating(knownRatings, subgroupOfUsers, unknownItem, user){
//Aplica la fórmula del algoritmo *Slope One* para obtener la valoración esperada
}

searchNullValues(recommendedItems){
//En caso de que un ítem no tenga ninguna valoración esperada asignada, se le asignará un valor 0
}

limitRecommendedItems(recommendedItems, Q){
//limita el número de elementos del mapa recommendedItems al valor especificado por el parámetro Q.
}

Content-Based Filtering

Técnica de filtrado consistente en recomendar un nuevo objeto en base a los ítems previamente valorados por un usuario.
En nuestro sistema software, emplearemos el algoritmo KNN (K Nearest Neighbours). Una vez cargados todos los datos necesarios para ejecutar el algoritmo (usuario, ítems valorados e ítems a valorar), calcularemos la distancia entre objetos en base a los atributos de cada uno. Distinguiremos entre cuatro tipos de atributos (*Integer*, *Double*, *String* y *Set<String>*).
Con tal de calcular la distancia entre atributos de tipo *Integer* y *Double*, emplearemos la distancia Euclidiana. Por otra parte, para calcular la distancia entre aquellos de tipo *String*, nos limitaremos a comprobar que los valores de sendos atributos se correspondan. Finalmente, la distancia entre los atributos *Set<String>* se obtendrá dividiendo el número de valores distintos entre la cantidad de valores totales.
Una vez calculadas las distancias entre los ítems, únicamente restará comprobar la proximidad entre los ítems y ordenarlos en función de dichos valores.

PseudoCodigo:

El algoritmo contiene 2 funciones públicas (preprocessing y query):

```
preprocessing(itemMap, userMap) {
  //No hace nada
}

query( user, unknownItems, Q) {
  filteredKnownItems = filterKnownItems(user);
  neighbourhood = Empty Map;
  for(item : filteredKnownItems) {
    neighbour = empty list //listOfDistancesBetweenKnownItemAndUnknownItems
    for(unknownItem : unknownItems) {
      neighbour.add(unknownItem,calculateDistance(unknownItem,item));
    }
    neighbourhood.add(neighbour);
  }
  return filterNeighbourhood(neighbourhood);
}

filterKnownItems(user){
  //Obtiene el subconjunto de known items cuyo rating es superior a la media
}

calculateDistance(unknownItem,item){
  return distanceStringAttributes(unknownItem.stringAttributes,item.stringAttributes)
      + distanceIntAttributes(unknownItem.intAttributes,item.intAttributes)
      + distanceDoubleAttributes(unknownItem.doubleAttributes,item.doubleAttribute)
      + distanceSetAttributes(unknownItem.setAttributes,item.setAttributes);
 }
```

```
 distanceStringAttributes(unknownAttributes,knownAttributes){
          distance = 0;
  for(unknownAttribute :unknownAttributes) {
    if(knownAttributes.contains(unknownAttribute) {
      if(unknownAttribute.value != knownAttributes.get(unknownAttribute)
        ++distance;
    }
  }
  return distance;
}

 distanceIntAttributes(unknownAttributes,knownAttributes){
  distance = 0;
  for(unknownAttribute :unknownAttributes) {
    if(knownAttributes.contains(unknownAttribute) {
      distance += absoluteValue (unknownNormalized - knownNormalized);
    }
  }
  return distance;
}
 distanceDoubleAttributes(unknownAttributes,knownAttributes){
  distance = 0;
  for(unknownAttribute :unknownAttributes) {
    if(knownAttributes.contains(unknownAttribute) {
      distance += absoluteValue (unknownNormalized - knownNormalized);
    }
  }
  return distance;
 }


 distanceSetAttributes(unknownAttributes,knownAttributes){
```

```
  distance = 0;
  for(unknownAttribute :unknownAttributes) {
    if(knownAttributes.contains(unknownAttribute) and unknownAttribute.size > 0) {
          distance += (unknownSize - intersectionBetweenUnknownAndKnwon) / unknownSize
    }
  }
  return distance;
}
```

Hybrid Approach

Técnica de filtrado consistente en predecir la valoración que un determinado sujeto realizaría sobre un objeto o ítem empleando distintas técnicas.
En nuestro sistema software, nos limitaremos a realizar la unión del resultado obtenido por el resto de algoritmos (Content-Based y Collaborative Filtering). En caso de que ambos algoritmos recomendasen un mismo ítem, se calcularía la media de las valoraciones.

PseudoCodigo:

El algoritmo contiene 2 funciones públicas (preprocessing y query):

```
preprocessing(itemMap, userMap){
  //No hace nada
}

query( user, unknownItems, Q){
  queryContent = contentBasedFiltering.query(user,unknownItems,Q)
  queryCollaborative = collaborativeFiltering.query(user,unknownItems,Q)
```

```
      result = queryContent
  for( Item, rate : queryCollaborative) {
    if(result.contains(item)
      result.replace(item , (rate + result.get(item) ) / 2
    else
      result.put(item, rate)
  }
  result.sort()
  return result
}
```

Estructuras de datos

Este apartado contiene la descripción y justificación del uso de las distintas estructuras de datos agrupadas por clases.

Collaborative Filtering

centroids (Map<String, User>): contienen los usuarios centroides de los clústers de usuarios. Pese a que el emplear un diccionario suponga un gasto adicional de memoria, facilita el acceso a los datos.
clusters (Map<String, UserGroup>): contiene los clústers de usuarios generados por el algoritmo. Pese a que el emplear un diccionario suponga un gasto adicional de memoria, facilita el acceso a los datos.
users (Map<String, User>): contiene todos los usuarios del sistema necesarios para generar los clústers. Pese a que el emplear un diccionario suponga un gasto adicional de memoria, facilita el acceso a los datos.
items (Map<String, Item>): contiene todos los ítems del sistema. Pese a que el emplear un diccionario suponga un gasto adicional de memoria, facilita el acceso a los datos.

Content-Based

users (Map<String, User>): contiene todos los usuarios del sistema necesarios para generar los clústers. Pese a que el emplear un diccionario suponga un gasto adicional de memoria, facilita el acceso a los datos.
items (Map<String, Item>): contiene todos los ítems del sistema. Pese a que el emplear un diccionario suponga un gasto adicional de memoria, facilita el acceso a los datos.

Hybrid

contentBasedFiltering (ContentBasedFiltering): contiene el algoritmo de Content Based Filtering.
collaborativeFiltering (CollaborativeFiltering): contiene el algoritmo de Collaborative Filtering.

Item

stringAttributes (Map<String, String>): contiene los atributos de tipo *String* de un determinado ítem. Hemos empleado un mapa ya que permite almacenar un identificador (nombre del atributo) y un valor (valor del atributo).
intAttributes (Map<String, Integer>): contiene los atributos de tipo *Integer* de un determinado ítem. Hemos empleado un mapa ya que permite almacenar un identificador (nombre del atributo) y un valor (valor del atributo).
doubleAttributes (Map<String, Double>): contiene los atributos de tipo *Double* de un determinado ítem. Hemos empleado un mapa ya que permite almacenar un identificador (nombre del atributo) y un valor (valor del atributo).
setAttributes (Map<String, Set<String>>): contiene los atributos conformados por un conjunto de *Strings* de un determinado ítem. Hemos empleado un mapa ya que permite almacenar un identificador (nombre del atributo) y un valor (valor del atributo).

Recommendation

recommendedItems (Map<String, Double>): contiene el conjunto de ítems recomendados. Se ha empleado un mapa para facilitar el acceso y corresponderse con los diccionarios utilizados por los algoritmos.

User

items (Map<String, Item>): contiene el conjunto de ítems valorados por un usuario. Si bien es cierto que la utilización de un mapa conlleva un gasto adicional de memoria al almacenar información redundante (la *key* se corresponde con el identificador del ítem), facilita considerablemente el acceso a los datos, minimizando, de este modo, el encapsulamiento y favoreciendo la reusabilidad y mantenimiento de la clase.
ratings (Map<String, Double>): contiene el conjunto de valoraciones de un usuario. Al contrario que en el caso anterior, este mapa no almacena información redundante ya que la *key* se corresponde con el identificador del ítem y el valor con la puntuación proporcionada por el usuario.

Usergroup

users (Map<String, User>): contiene el conjunto de usuarios que conforman un grupo. Al igual que en el caso de *items*, el mapa, con tal de facilitar el acceso, almacena información redundante (la *key* se corresponde con el identificador del usuario).