UNIVERSIDADE DO MINHO

Mini-teste 3

Mestrado Integrado em Engenharia Informática

Mineração de Dados
(1.º Semestre / 2020-2021)

A84003 Beatriz Rocha

Braga,
Janeiro 2021

1. Considere o *dataset* "unbalanced". Compare o desempenho dos algoritmos *k-means* e *EM* quando o número de partições é determinado pelo algoritmo *EM*. Considerar o número de partições (*clusters*) obtidas. Deve "desligar" o atributo classe para tornar mais real a avaliação. Compare estes resultados com os obtidos pelo algoritmo *DBSCAN*.

Antes de mais, é importante começar por explicar cada um dos algoritmos mencionados:

- *EM*: o algoritmo *Expectation Maximization* trata-se de uma abordagem para obter a estimativa de máxima verosimilhança na presença de variáveis latentes. Este algoritmo executa esse procedimento, começando por estimar os valores das variáveis latentes, passando para a otimização do modelo e repetindo estes dois passos até convergir;

- *K-means*: o algoritmo *k-means* armazena os k centróides (pontos que são o centro de um *cluster*) que usa para definir os *clusters*. Considera-se que um ponto está num *cluster* particular se está mais perto do centróide desse *cluster* do que qualquer outro centróide. Este algoritmo encontra os melhores centróides alternando entre atribuir pontos a *clusters* com base nos centróides atuais e escolher centróides com base na atribuição atual dos pontos aos *clusters*;

- *DBSCAN*: dado um conjunto de pontos no espaço, o algoritmo *Density-based spatial clustering of applications with noise* cria grupos de pontos que estão muito próximos (pontos com muitos vizinhos próximos), marcando como *outliers* aqueles que ficam isolados em regiões de baixa densidade (cujos vizinhos mais próximos se encontram muito longe).

Começando por testar o algoritmo *EM* através de validação cruzada e ignorando o atributo `Outcome`, são necessárias 2 iterações para completar a sua execução e são devolvidos 18 *clusters*, tal como podemos ver de seguida:

```
Number of clusters selected by cross validation: 18
Number of iterations performed: 2
```

A distribuição das instâncias da classe `Outcome` pode ser vista na figura que se segue:

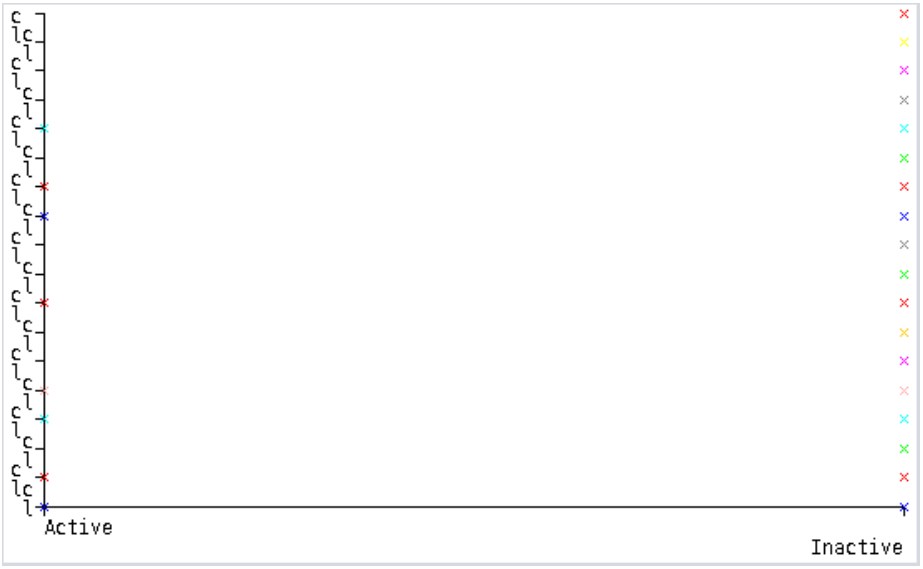

Figura 1: Distribuição das instâncias da classe `Outcome` para o algoritmo *EM*

Daqui, podemos constatar que o número 18, deve-se ao facto de o algoritmo *EM* dividir as instâncias `Active` da classe `Outcome` em múltiplos *clusters* em vez de apenas 1, enquanto as instâncias `Inactive` da mesma classe estão bem distribuídas.

Apesar de este algoritmo determinar o número de partições, há desvantagens associadas a esse processo, nomeadamente o tempo de execução. De facto, são necessários 46.38 segundos para calcular o número ideal de *clusters*, tal como podemos ver de seguida:

```
Time taken to build model (full training data) : 46.38 seconds
```

Já se este número de *clusters* for fornecido, o tempo de execução decresce significativamente (para 0.45 segundos):

```
Time taken to build model (full training data) : 0.45 seconds
```

De seguida, podemos ver a colocação das instâncias do conjunto de dados em cada um dos *clusters*:

```
Clustered Instances

0       45 (  5%)
1       25 (  3%)
2      133 ( 16%)
3       42 (  5%)
4       26 (  3%)
5       56 (  7%)
6       46 (  5%)
7       38 (  4%)
```

```
 8        22 (  3%)
 9        54 (  6%)
10        69 (  8%)
11        37 (  4%)
12        55 (  6%)
13        58 (  7%)
14        13 (  2%)
15        61 (  7%)
16        48 (  6%)
17        28 (  3%)
```

Definido o número de *clusters*, podemos agora executar o algoritmo *k-means* que se demonstra bastante mais rápido (0.09 segundos):

```
Time taken to build model (full training data) : 0.09 seconds
```

Para além disso, podemos constatar que este algoritmo demora 20 iterações a completar a sua execução e apresenta uma soma dos erros quadrados igual a, aproximadamente, 321.32:

```
Number of iterations: 20
Within cluster sum of squared errors: 321.3168661336296
```

A distribuição das instâncias da classe `Outcome` pode ser vista na figura seguinte:

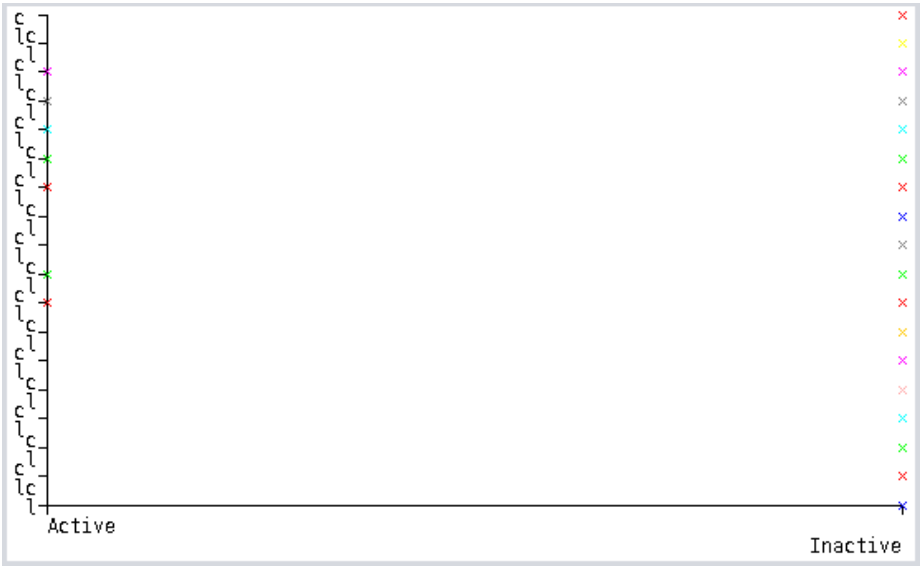

Figura 2: Distribuição das instâncias da classe `Outcome` para o algoritmo *k-means*

Por último, apresenta-se a colocação das instâncias do *dataset* em cada um dos *clusters*:

```
Clustered Instances
```

```
 0        22 (  3%)
 1        19 (  2%)
 2        75 (  9%)
 3        27 (  3%)
 4        52 (  6%)
 5       122 ( 14%)
 6        39 (  5%)
 7       102 ( 12%)
 8        52 (  6%)
 9        21 (  2%)
10        10 (  1%)
11        72 (  8%)
12        45 (  5%)
13        37 (  4%)
14        38 (  4%)
15        30 (  4%)
16        57 (  7%)
17        36 (  4%)
```

Executando agora o algoritmo *DBSCAN* sem a modificação de nenhum dos parâmetros, começamos por ver que este é executado em 0.18 segundos:

```
Time taken to build model (full training data) : 0.18 seconds
```

De seguida, podemos observar que este algoritmo devolve 2 *clusters* e a respetiva colocação das instâncias do *dataset* em cada um dos mesmos:

```
Clustered Instances

0      233 ( 28%)
1      612 ( 72%)
```

Por último, podemos ver a distribuição das instâncias da classe `Outcome` na figura seguinte:

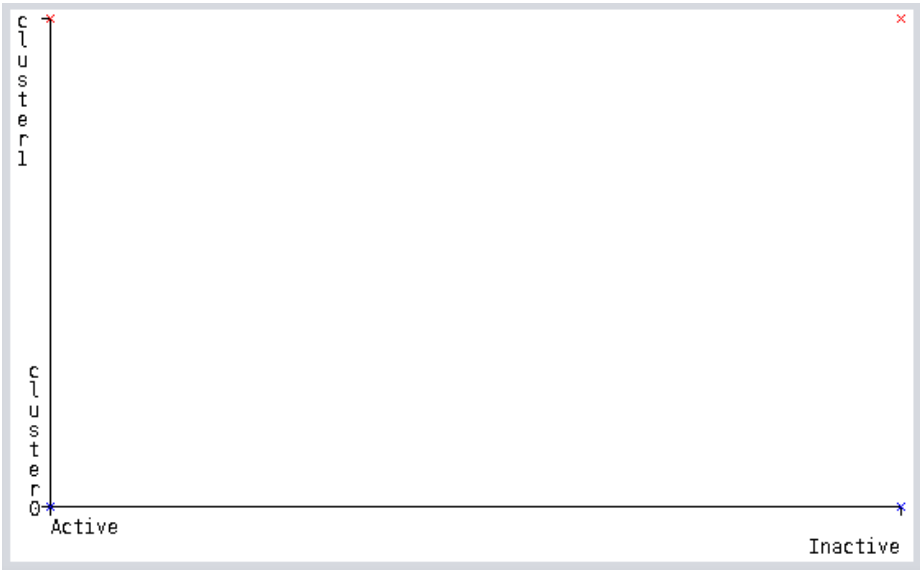

Figura 3: Distribuição das instâncias da classe `Outcome` para o algoritmo *DBSCAN*

Contudo, se diminuirmos o valor de *epsilon* para 0.5, ou seja, se diminuirmos o raio da vizinhança para 0.5, obtemos novamente os 18 *clusters*, mas um maior número de pontos é considerado como ruído.

De seguida, podemos ver a colocação das instâncias do conjunto de dados em cada um dos *clusters* e, por último, a distribuição das instâncias da classe `Outcome`:

```
Clustered Instances

 0       33 (  6%)
 1       29 (  5%)
 2       20 (  4%)
 3       34 (  6%)
 4       79 ( 14%)
 5       46 (  8%)
 6       45 (  8%)
 7      173 ( 30%)
 8       11 (  2%)
 9       21 (  4%)
10       10 (  2%)
11       11 (  2%)
12        8 (  1%)
13        6 (  1%)
14       11 (  2%)
15       20 (  4%)
16        8 (  1%)
17        6 (  1%)
```

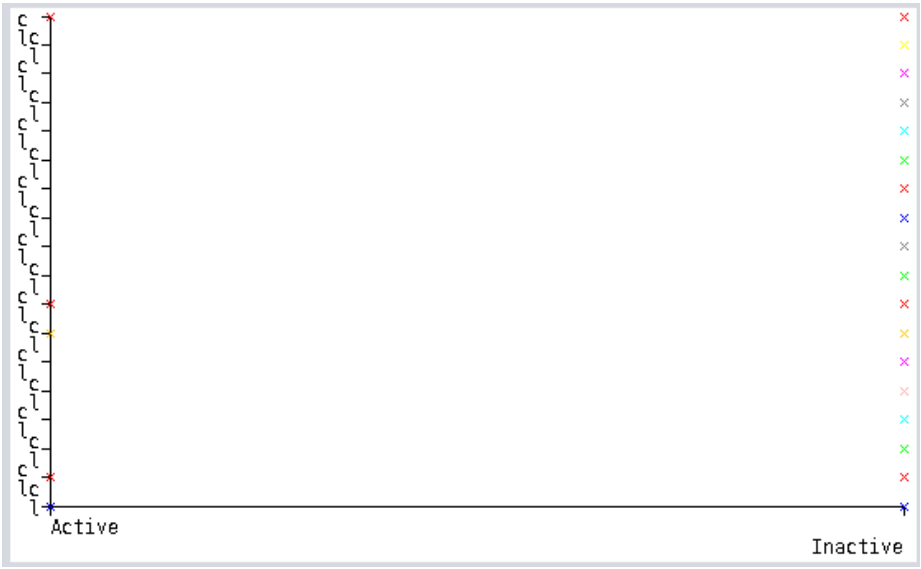

Figura 4: Distribuição das instâncias da classe `Outcome` para o algoritmo *DBSCAN* com *epsilon* igual a 0.5

2. Considere o *dataset* de disciplinas (*student_courses.bas*, descarregar do *blackboard*). Apresente para este *dataset* exemplos de regras redundantes, produtivas e não produtivas e regras significativas. Comente os exemplos obtidos/usados.

Comecemos por explicar cada uma das regras mencionadas:

- Regras redundantes: diz-se que uma regra é redundante se a sua generalização tem o mesmo valor de suporte;
- Regras produtivas: diz-se que uma regra é produtiva se o seu *improvement* for positivo, isto é, uma regra mais específica produz uma mais valia em termos de valor de medida de interesse;
- Regras não produtivas: diz-se que uma regra é não produtiva se, ao contrário de uma regra produtiva, o seu *improvement* for negativo, ou seja, se uma regra mais específica não produz uma mais valia em termos de valor de medida de interesse;
- Regras significativas: diz-se que uma regra é significativa se é estatisticamente significante relativamente às suas generalizações.

Para obter todas as regras de associação, podemos recorrer ao sistema CAREN com o comando seguinte:

```
java caren ~/4ano1sem/MD/Mini_teste_3/student_courses.bas 0.1 0.5
-s, -d
```

Feito isto e observando o *output*, rapidamente identificamos algumas regras redundantes, nomeadamente:

```
Sup = 0.11374  Conf = 1.00000   GEST400    <--    CC410  &  CC432
&  CC442  &  CC412  &  DIP463

Sup = 0.11374  Conf = 1.00000   GEST400    <--    CC410  &  CC432
&  CC442  &  CC412  &  DIP463  &  DIP461
```

Efetivamente, a segunda regra é redundante, visto que apresenta mais um atributo (`DIP461`), o que resulta em maior complexidade, mas o seu valor de suporte é igual ao da sua generalização (0.11374).

O sistema CAREN permite filtrar as regras produtivas com a *flag* `-imp` que indica um valor mínimo para o valor de *improvement*, tal como podemos ver de seguida:

```
java caren ~/4ano1sem/MD/Mini_teste_3/student_courses.bas 0.1 0.5
-s, -d -imp0.0001
```

Visto que, para uma regra de associação ser produtiva, é necessário que o seu *improvement* seja positivo, o valor de `-imp` tem de ser superior a 0, daí o valor 0.0001. É de notar que o número de regras reduziu significativamente (de 36601 para 3402). Alguns exemplos de regras produtivas podem ser vistos de seguida:

```
Sup = 0.39336  Conf = 0.63359   CC421    <--    CC411  &  CC412

Sup = 0.29384  Conf = 0.63265   CC421    <--    CC442  &  DIP461

Sup = 0.30332  Conf = 0.62745   CC421    <--    CC442  &  CC411

Sup = 0.34123  Conf = 0.62609   CC421    <--    DIP463  &  CC411
```

Para que uma regra seja não produtiva, é necessário que haja uma regra mais geral, cujo valor de confiança seja superior. Ora, isto pode ser visto de seguida, onde o valor de confiança da regra mais geral (0.96667) é superior ao valor de confiança da regra mais específica (0.96552):

```
Sup = 0.13744  Conf = 0.96667   CC421    <--    CC583  &  CC422  &
CC412  &  CC413

Sup = 0.13270  Conf = 0.96552   CC421    <--    CC583  &  CC422  &
CC412  &  CC413  &  CC420
```

Por último, o sistema CAREN ainda permite filtrar as regras significativas e, para isso, basta recorrer à *flag* `-fisher`, tal como se apresenta de seguida:

```
java caren ~/4ano1sem/MD/Mini_teste_3/student_courses.bas 0.1 0.5
-s, -fisher -d
```

Esta *flag* usa, por omissão, o valor de α a 0.05 para rejeição da hipótese nula. Mais uma vez, observou-se um grande decréscimo no número de regras não só relativamente a todas as regras de associação (de 36601 para 671), mas também relativamente às regras produtivas (de 3402 para 671). Alguns exemplos de regras significativas podem ser vistos de seguida:

```
Sup = 0.26066  Conf = 0.79710   CC442    <--    CC432  &  CC412

Sup = 0.30806  Conf = 0.79268   CC442    <--    CC421  &  DIP463

Sup = 0.39810  Conf = 0.78505   CC442    <--    CC420  &  CC411

Sup = 0.29384  Conf = 0.78481   CC442    <--    CC421  &  GEST400
```

3. Considere o *dataset adult* (descarregar do *blackboard*). Usando o sistema CAREN e os seus vários métodos para *Subgroup Mining* apresente e comente regras que denunciam discriminação de género e idade em termos de rendimentos anuais. Notar: o atributo classe deste *dataset Adult* caracteriza os rendimentos anuais dos indivíduos registados i.e. <=50K indica rendimento anual menor que 50 000 *USD* (e o seu oposto).

Para responder a esta questão, comecei por executar o seguinte comando que, por sua vez, irá gerar os grupos de contraste `class=<=50K` e `class=>50K`:

```
java -cp . caren ../../4ano1sem/MD/Mini_teste_3/adult.wd-fi.csv.test
0.01 0.5 -s, -Att -h"class=<=50K","class=>50K" -CS -ovrt -null?
```

Daqui, podemos observar regras que denunciam discriminação de género em termos de rendimentos anuais, nomeadamente:

```
Obs =  004831 | 000590  Gsup  = 0.38850 | 0.15341  p = 2.2578981927E-177
phi =  0.21189   class=<=50K >> class=>50K
Sup(CS) =  0.33296                                  <---     sex=Female

Obs =  003256 | 007604  Gsup  = 0.84659 | 0.61150  p = 2.2578981928E-177
phi =  0.21189   class=>50K >> class=<=50K
Sup(CS) =  0.66704                                    <---     sex=Male
```

De facto, podemos ver que o género feminino é discriminado em relação ao género masculino, pois a maior parte da população feminina (`sex=Female`) deste *dataset* recebe menos ou exatamente 50K *USD* por ano (`class=<=50K >> class=>50K`), enquanto a maior parte da população masculina (`sex=Male`) deste *dataset* recebe mais de 50K *USD* anualmente (`class=>50K >> class=<=50K`).

A partir do mesmo comando, podemos também observar regras que denunciam discriminação de idade em termos de rendimentos anuais:

```
Obs =  001584 | 000005  Gsup  = 0.12738 | 0.00130  p = 2.7638719927E-186
phi =  0.18046   class=<=50K >> class=>50K
Sup(CS) =  0.09760                               <---     age=(-inf-21.5]

Obs =  000851 | 000014  Gsup  = 0.06844 | 0.00364  p = 9.8297179458E-082
phi =  0.12271   class=<=50K >> class=>50K
Sup(CS) =  0.05313                               <---     age=(21.5-23.5]

Obs =  000395 | 000013  Gsup  = 0.03177 | 0.00338  p = 1.6192042870E-031
```

```
phi =  0.07714   class=<=50K >> class=>50K
Sup(CS) =  0.02506                              <---     age=(23.5-24.5]

Obs =  001040 | 000079  Gsup  = 0.08363 | 0.02054  p = 7.6387884205E-052
phi =  0.10593   class=<=50K >> class=>50K
Sup(CS) =  0.06873                              <---     age=(24.5-27.5]

Obs =  001040 | 000200  Gsup  = 0.08363 | 0.05200  p = 1.1981841362E-011
phi =  0.05065   class=<=50K >> class=>50K
Sup(CS) =  0.07616                              <---     age=(27.5-30.5]

Obs =  000818 | 001701  Gsup  = 0.21269 | 0.13679  p = 1.5089631813E-028
phi =  0.08915   class=>50K >> class=<=50K
Sup(CS) =  0.15472                              <---     age=(35.5-41.5]

Obs =  001522 | 002458  Gsup  = 0.39574 | 0.19767  p = 8.9643185008E-129
phi =  0.19576   class=>50K >> class=<=50K
Sup(CS) =  0.24446                              <---     age=(41.5-54.5]

Obs =  000408 | 000829  Gsup  = 0.10608 | 0.06667  p = 4.4082432499E-015
phi =  0.06319   class=>50K >> class=<=50K
Sup(CS) =  0.07598                              <---     age=(54.5-61.5]
```

Efetivamente, podemos constatar que até aos 30.5 anos (inclusive), os indivíduos são desfavorecidos no que toca ao salário anual, uma vez que a maior parte da população do *dataset* com idade inferior a 30.5 anos recebe menos ou exatamente 50K *USD* anualmente. Já a maior parte da população do *dataset* com idade superior a 35.5 anos recebe mais do que 50K *USD* anualmente, verificando-se, assim, uma discriminação nesse sentido, ou seja, os indivíduos mais jovens são prejudicados em comparação aos mais velhos.