

# Weka na Munheca

Um guia para uso do Weka em *scripts* e  
integração com aplicações em Java

Rafael Santos ([www.lac.inpe.br/~rafael.santos](http://www.lac.inpe.br/~rafael.santos))

## Sumário

<b>Sobre este documento</b>	<b>1</b>
Notações usadas neste documento	1
<b>1 Introdução</b>	<b>2</b>
1.1 Sobre o Weka	2
1.2 Ambiente de desenvolvimento	2
<b>2 Usando aplicações do Weka em <i>scripts</i></b>	<b>2</b>
2.1 Formato <code>.ARFF</code>	2
2.2 Executando aplicações a partir da linha de comando (classificação com o algoritmo J4.8)	3
2.3 Classificando dados com o algoritmo <i>K-Means</i>	7
2.4 Encontrando regras de associação com o algoritmo <i>Apriori</i>	8
2.4.1 Discretizando os dados contínuos	10
<b>3 Incluindo classes do Weka em aplicações em Java</b>	<b>13</b>

## Lista de Listagens

2.1 Conteúdo do arquivo <code>weather.arff</code>	3
2.2 Executando o classificador J4.8 pela linha de comando (com <i>classpath</i> )	4
2.3 Declarando a variável de ambiente <code>CLASSPATH</code>	4
2.4 Executando o classificador J4.8 pela linha de comando (sem <i>classpath</i> )	4
2.5 Resultado da execução do comando da listagem 2.2	5
2.6 Parte do arquivo <code>iris.arff</code> modificado	7
2.7 Executando o algoritmo <i>K-Means</i> pela linha de comando	7
2.8 Resultado da execução do algoritmo <i>K-Means</i> com os dados de íris	7
2.9 O arquivo <code>compras.arff</code>	8
2.10 Executando o algoritmo <i>Apriori</i> pela linha de comando	9
2.11 Resultado da execução do algoritmo <i>Apriori</i>	9
2.12 Comando para discretizar os atributos do arquivo <code>iris.arff</code>	11
2.13 Parte do arquivo <code>iris-disc.arff</code> (discretizado)	11

2.14	Executando o algoritmo <i>Apriori</i> pela linha de comando (arquivo <code>iris-disc.arff</code> ) . . .	12
2.15	Resultado da execução do algoritmo <i>Apriori</i> (arquivo <code>iris-disc.arff</code> ) . . . . .	12
3.1	Lendo o conteúdo de um arquivo <code>.arff</code> . . . . .	13
3.2	Criando vetores de dados e armazenando em um arquivo <code>.arff</code> . . . . .	14
3.3	Criando vetores de dados e armazenando em um arquivo <code>.arff</code> . . . . .	15
3.4	Executando o algoritmo J4.8 em uma aplicação (e avaliando o resultado) . . . . .	16
3.5	Executando o algoritmo KMeans em uma aplicação (duas vezes) . . . . .	17
3.6	Executando o algoritmo KMeans em uma aplicação (e classificando instâncias dos dados)	17

## Sobre este documento

**Antes de imprimir este documento por favor leia esta seção.**

Este documento é um guia para o desenvolvimento de aplicações usando o software Weka [1]. “Weka na Munheca” indica o foco do documento: ele descreve como usar as classes e aplicativos do pacote Weka em pequenos *scripts* e aplicações em Java ao invés de usar as interfaces gráficas do pacote (Explorer, KnowledgeFlow).

O documento foi escrito para servir de guia para algumas das aulas da disciplina *Princípios e Aplicações de Mineração de Dados* (CAP-359-3), oferecido pelo Programa de Pós-Graduação em Computação Aplicada do Instituto Nacional de Pesquisas Espaciais. Veja o site <http://www.lac.inpe.br/cap/> para mais informações sobre o programa e <http://www.lac.inpe.br/~rafael.santos/cap359/> para mais informações sobre a disciplina.

Este documento está em construção, e provavelmente será escrito e reescrito nos próximos meses. Ao imprimir este documento por favor considere que outras versões do mesmo estarão disponíveis no futuro. Versões do documento podem ser identificadas pela sua data, presente no rodapé. Em novas versões darei prioridade para adicionar conteúdo e exemplos e não para o formato do documento.

## Notações usadas neste documento

Comandos, listagens, resultados de execução e opções para programas são mostrados com fonte proporcional.

*Links* para sites de interesse são mostrados em azul. O documento ainda pode ter algumas notas de interesse do autor (basicamente “tenho que terminar de escrever isso!”), estas anotações aparecem em **violeta negro**.

Trechos envoltos em caixas com linhas azuis correspondem a comandos que devem ser executados em uma linha de comando. Trechos envoltos em caixas com linhas verdes correspondem a resultados exibidos no terminal pelos programas executados. Programas que devem ser digitados, compilados e executados aparecem envoltos em caixas com linhas vermelhas. Exercícios propostos aparecem dentro de caixas com linhas pretas.

# 1 Introdução

## 1.1 Sobre o Weka

## 1.2 Ambiente de desenvolvimento

Para desenvolver aplicações e *scripts* com Weka, você deve ter o ambiente de desenvolvimento Java (*J2SE*, *Java 2 Standard Edition*) instalado. Note que não basta ter somente a máquina virtual (*J2SE Runtime Environment*), é necessário ter instalado o *J2SE Development Kit*. Se você não tiver o ambiente instalado, visite <http://java.sun.com> para obter o software e instruções para instalação.

O ambiente de desenvolvimento sugerido é o Eclipse (<http://www.eclipse.org>). A versão sugerida é a 3.1. Tanto o Eclipse quanto o *J2SE Development Kit* podem ser instalados em computadores executando os sistemas operacionais Windows e Linux. Todos os exemplos mostrados neste documento foram criados usando Linux, mas a execução em Windows não deve apresentar problemas.

Para organizar melhor o código que usa o Weka devemos criar um projeto novo no Eclipse. Para isso use a opção `File - New - Project` do menu do Eclipse. Escolha `Java Project` e use um nome adequado para seu projeto. Para que o Weka possa ser incluído nas classes do nosso projeto, devemos avançar (use o botão `Next` ao invés de `Finish`) e selecione a `tab Libraries` no diálogo `Java Settings`. Use o botão `Add External JARs` para adicionar o arquivo `weka.jar` (a localização depende de onde você instalou o Weka). Clique no botão `Finish` para criar o projeto.

Para criar uma classe em seu projeto use a opção `File - New - Class`. Para simplificar usaremos o pacote `default` embora o Eclipse informe que isso não é aconselhável.

Mais detalhes sobre a criação de classes serão mostrados na seção 3.

## 2 Usando aplicações do Weka em *scripts*

Nesta seção veremos como usar as aplicações do Weka a partir da linha de comando e em *scripts* simples. Antes de executar a aplicação, precisamos formatar dados no formato `.ARFF` (*Attribute-Relation File Format*), que é o formato esperado pelos componentes do software Weka.

### 2.1 Formato `.ARFF`

Um arquivo no formato `.ARFF` é um arquivo de texto puro, composto de três partes:

- **Relação**, a primeira linha do arquivo, que deve ser igual a `@relation` seguida de uma palavra-chave que identifique a relação ou tarefa sendo estudada.
- **Atributos**, um conjunto de linhas onde cada uma inicia com `@attribute` seguida do nome do atributo e seguida do seu tipo, que pode ser nominal (neste caso as alternativas devem aparecer como uma lista separada por vírgulas e cercadas por chaves) ou numérico (neste caso o nome deve ser seguido da palavra-chave `real`). Geralmente, em uma tarefa de classificação supervisionada (onde conhecemos as classes das instâncias usadas para treinamento) o último atributo é a classe para as instâncias.

- **Dados**, depois de uma linha contendo @data. Cada linha deve corresponder a uma instância e deve ter valores separados por vírgula correspondentes (e na mesma ordem) dos atributos da seção Atributos.

O arquivo também pode conter linhas iniciadas com o sinal de percentagem (%). Estas linhas serão consideradas comentários e não serão processadas.

Para referência, o conteúdo do arquivo `weather.arff` é mostrado na listagem 2.1.

Listagem 2.1: Conteúdo do arquivo `weather.arff`

```
1 @relation weather
2
3 @attribute outlook {sunny, overcast, rainy}
4 @attribute temperature real
5 @attribute humidity real
6 @attribute windy {TRUE, FALSE}
7 @attribute play {yes, no}
8
9 @data
10 sunny,85,85,FALSE,no
11 sunny,80,90,TRUE,no
12 overcast,83,86,FALSE,yes
13 rainy,70,96,FALSE,yes
14 rainy,68,80,FALSE,yes
15 rainy,65,70,TRUE,no
16 overcast,64,65,TRUE,yes
17 sunny,72,95,FALSE,no
18 sunny,69,70,FALSE,yes
19 rainy,75,80,FALSE,yes
20 sunny,75,70,TRUE,yes
21 overcast,72,90,TRUE,yes
22 overcast,81,75,FALSE,yes
23 rainy,71,91,TRUE,no
```

## 2.2 Executando aplicações a partir da linha de comando (classificação com o algoritmo J4.8)

Esta subseção usa o mesmo exemplo mostrado no capítulo 8 de Witten e Frank [2]. O exemplo foi traduzido, corrigido para a última versão do Weka e ligeiramente modificado.

Vamos usar o algoritmo J4.8, que cria uma árvore de decisão, com os dados `weather` (previsão de jogar golfe ou não baseado em algumas informações meteorológicas simples, mostrado na figura 2.1). A árvore pode ser criada com o comando mostrado na listagem 2.2.

### Listagem 2.2: Executando o classificador J4.8 pela linha de comando (com *classpath*)

```
1 java -cp /usr/local/weka-3-4-4/weka.jar  
2     weka.classifiers.trees.J48  
3     -t /usr/local/weka-3-4-4/data/weather.arff
```

Algumas observações sobre o comando na listagem 2.2:

- Todo o conteúdo da listagem acima deve ser digitado em uma única linha de comando. A linha está quebrada em três para facilitar a visualização e explicação.
- Usamos a opção `-cp` para indicar que usaremos um pacote adicional para a máquina virtual Java (linha 1). O argumento para a opção `-cp` deve ser o caminho **completo** para a localização do arquivo `weka.jar`. O exemplo assume que o pacote foi instalado no diretório `/usr/local/weka-3-4-4/`.
- O nome da classe que implementa o classificador é `weka.classifiers.trees.J48` (linha 2). Este nome inclui o nome dos pacotes e subpacotes onde as classes estão organizadas.
- A opção `-t` (linha 3) indica o nome do arquivo que será usado como entrada (arquivo de treinamento). Vários arquivos no formato `.arff` são distribuídos como parte do Weka (no exemplo, os arquivos estão instalados no diretório `/usr/local/weka-3-4-4/data/`).

Uma forma alternativa de execução de comandos que dispensa a opção `-cp` é declarar a variável de ambiente `CLASSPATH`, o que é feito diferentemente para o Windows e para diversos *shells* do Linux. Como exemplo, para o *shell* TCSH no Linux podemos usar o comando mostrado na listagem 2.3.

### Listagem 2.3: Declarando a variável de ambiente `CLASSPATH`

```
1 setenv CLASSPATH /usr/local/weka-3-4-4/weka.jar
```

Com a variável de ambiente `CLASSPATH` definida, podemos executar o algoritmo J4.8 com o comando mostrado na listagem 2.4.

### Listagem 2.4: Executando o classificador J4.8 pela linha de comando (sem *classpath*)

```
1 java weka.classifiers.trees.J48  
2     -t /usr/local/weka-3-4-4/data/weather.arff
```

Enquanto a sessão durar (isto é, enquanto a variável de ambiente `CLASSPATH` estiver definida) podemos usar a forma acima, sem a opção `-cp`. Existem formas de manter a variável de ambiente constante entre sessões, que também são diferentes para os diferentes *shells* e que não serão cobertas neste documento.

Independente da forma usada para executar o comando, o resultado será exibido no terminal. É interessante paginar o resultado para melhor visualização, isso pode ser feito terminando o comando com `| less` para pagnar de forma simples ou redirecionando o resultado para um arquivo, terminando o comando com `> res.txt` (onde o arquivo `res.txt` conterá o resultado da execução, sendo criado ou sobrescrito se necessário).

O resultado da execução do comando mostrado na listagem 2.2 é mostrado na listagem 2.5.

Listagem 2.5: Resultado da execução do comando da listagem 2.2

```
1 J48 pruned tree
2 -----
3
4 outlook = sunny
5 |   humidity <= 75: yes (2.0)
6 |   humidity > 75: no (3.0)
7 outlook = overcast: yes (4.0)
8 outlook = rainy
9 |   windy = TRUE: no (2.0)
10 |  windy = FALSE: yes (3.0)
11
12 Number of Leaves   :    5
13
14 Size of the tree   :    8
15
16
17 Time taken to build model: 0.03 seconds
18 Time taken to test model on training data: 0 seconds
19
20 === Error on training data ===
21
22 Correctly Classified Instances      14           100      %
23 Incorrectly Classified Instances    0            0      %
24 Kappa statistic                     1
25 Mean absolute error                 0
26 Root mean squared error             0
27 Relative absolute error             0           %
28 Root relative squared error         0           %
29 Total Number of Instances          14
30
31
32 === Confusion Matrix ===
33
34  a b  <-- classified as
35  9 0 | a = yes
36  0 5 | b = no
37
```

```

38
39 === Stratified cross-validation ===
40
41 Correctly Classified Instances          9          64.2857 %
42 Incorrectly Classified Instances       5          35.7143 %
43 Kappa statistic                        0.186
44 Mean absolute error                    0.2857
45 Root mean squared error                0.4818
46 Relative absolute error                 60          %
47 Root relative squared error            97.6586 %
48 Total Number of Instances              14
49
50
51 === Confusion Matrix ===
52
53  a b  <-- classified as
54  7 2 | a = yes
55  3 2 | b = no

```

Alguns pontos de interesse no resultado mostrado na listagem 2.5 são:

- A linha 1 indica que o algoritmo usado foi o J4.8 e que a árvore foi podada (comportamento *default* para este algoritmo).
- As linhas 4 a 10 mostram a árvore de decisão de forma textual. Ao final de cada folha da árvore (que apresenta o valor possível para cada classe, no exemplo, ou *no* ou *yes*) um valor indica quantas instâncias do arquivo de treinamento foram classificadas por aquela folha. A árvore de decisão pode ser facilmente interpretada como um conjunto de regras de classificação, de forma análoga a um sistema especialista.
- As linhas 22 e 23 indicam a performance do classificador com os dados de treinamento (treinamos o algoritmo com os dados e depois verificamos ele com os próprios dados de treinamento).
- As linhas 32 a 36 mostram uma matriz de confusão que indica que instâncias foram classificadas de forma correta e incorreta. Se houve 100% de classificação correta podemos esperar uma matriz de confusão onde todo elemento fora das diagonais é igual a zero.
- O resultado de validação cruzada (dados de treinamento são misturados e reamostrados para classificação com a árvore criada, a experiência é repetida 10 vezes) é mostrado entre as linhas 39 e 55. Podemos observar que com validação cruzada alguns erros aparecem na classificação.

**Exercício 2.1:** Podemos usar o parâmetro `-M` (com um valor numérico inteiro) para indicar o número mínimo de instâncias nas folhas para o classificador J4.8. Repita a classificação com o conjunto de dados `weather.arff` e com diferentes valores para o argumento `-M` e analise os resultados.

## 2.3 Classificando dados com o algoritmo *K-Means*

O exemplo nesta subseção mostra como agrupar dados com o algoritmo *K-Means*. Os dados usados como exemplo são os das flores Íris, mas com os rótulos (que identificam as flores) removido – o arquivo original da distribuição do Weka contém os rótulos. Parte do arquivo modificado é mostrado na listagem 2.6.

Listagem 2.6: Parte do arquivo `iris.arff` modificado

```
1 @RELATION iris
2
3 @ATTRIBUTE sepallength REAL
4 @ATTRIBUTE sepalwidth REAL
5 @ATTRIBUTE petallength REAL
6 @ATTRIBUTE petalwidth REAL
7
8 @DATA
9 5.1,3.5,1.4,0.2
10 4.9,3.0,1.4,0.2
11 4.7,3.2,1.3,0.2
12 4.6,3.1,1.5,0.2
```

Supondo que a variável de ambiente `CLASSPATH` tenha sido criada, podemos executar o algoritmo de agrupamento com o comando mostrado na listagem 2.7.

Listagem 2.7: Executando o algoritmo *K-Means* pela linha de comando

```
1 java weka.clusterers.SimpleKMeans
2   -t /home/rafael/InpeHP/javadm/datasets/iris-unlabeled.arff
3   -N 3
```

Um dos parâmetros importantes para a execução do algoritmo *K-Means* é o número de grupos (*clusters*) a ser formado, neste caso, 3.

O resultado da execução do comando pode ser visto na listagem 2.8.

Listagem 2.8: Resultado da execução do algoritmo *K-Means* com os dados de íris

```
1 kMeans
2 =====
3
4 Number of iterations: 6
5 Within cluster sum of squared errors: 6.9981140048267605
6
7 Cluster centroids:
```

```

8
9 Cluster 0
10     Mean/Mode:  5.8885  2.7377  4.3967  1.418
11     Std Devs:   0.4487  0.2934  0.5269  0.2723
12 Cluster 1
13     Mean/Mode:  5.006   3.418   1.464   0.244
14     Std Devs:   0.3525  0.381   0.1735  0.1072
15 Cluster 2
16     Mean/Mode:  6.8462  3.0821  5.7026  2.0795
17     Std Devs:   0.5025  0.2799  0.5194  0.2811
18
19 === Clustering stats for training data ===
20
21 Clustered Instances
22 0         61 ( 41%)
23 1         50 ( 33%)
24 2         39 ( 26%)

```

Alguns pontos de interesse na listagem 2.8 são:

- Nas linhas 4 e 5 são mostrados alguns resultados do processo de agrupamento: número de iterações até convergência e medida objetiva minimizada (soma dos erros quadráticos entre *clusters*).
- Os centróides e desvios padrões de cada *cluster* são mostrados a seguir.
- O número de instâncias atribuídas para cada *cluster* é mostrado no final.

**Exercício 2.2:** Crie uma rotina em qualquer linguagem que receba como entrada o resultado do algoritmo *K-Means* e o arquivo com as instâncias, e classifique as instâncias como sendo pertencentes a um grupo qualquer usando como critério a distância euclidiana até o centróide do grupo mais próximo.

## 2.4 Encontrando regras de associação com o algoritmo *Apriori*

O próximo exemplo demonstra como obter regras de associação (do tipo “se A e C estão presentes então C também está presente”) usando o Weka. Para demonstrar este exemplo, vamos considerar o arquivo `compras.arff`, mostrado na listagem 2.9.

Listagem 2.9: O arquivo `compras.arff`

```

1 % Compras em um mercado (completamente simulado)
2 @RELATION compras
3
4 @attribute leite {sim, não}
5 @attribute ovos {sim, não}
6 @attribute café {sim, não}
7 @attribute açúcar {sim, não}

```

```

8 @attribute fraldas {sim, não}
9 @attribute manteiga {sim, não}
10 @attribute farinha {sim, não}
11 @attribute cerveja {sim, não}
12
13 @data
14 sim,sim,sim,sim,sim,sim,não,não
15 sim,não,sim,não,não,não,sim,não
16 sim,sim,não,sim,não,não,não,não
17 não,não,sim,sim,não,não,não,não
18 não,não,não,não,sim,não,não,não
19 sim,sim,não,não,não,sim,não,não
20 não,sim,não,não,não,sim,sim,não
21 sim,sim,sim,sim,não,sim,não,não
22 não,não,sim,não,sim,não,não,sim

```

O arquivo `compras.arff` contém valores puramente discretos ou nominais para seus atributos, no caso, indicando se um determinado produto foi comprado ou não.

Para encontrar as regras com o algoritmo *Apriori* podemos usar o comando mostrado na listagem 2.10.

#### Listagem 2.10: Executando o algoritmo *Apriori* pela linha de comando

```

1 java weka.associations.Apriori -t compras.arff

```

O resultado da execução do algoritmo *Apriori* é mostrado na listagem 2.11.

#### Listagem 2.11: Resultado da execução do algoritmo *Apriori*

```

1 Apriori
2 =====
3
4 Minimum support: 0.5
5 Minimum metric <confidence>: 0.9
6 Number of cycles performed: 10
7
8 Generated sets of large itemsets:
9
10 Size of set of large itemsets L(1): 13
11
12 Size of set of large itemsets L(2): 21
13
14 Size of set of large itemsets L(3): 9
15

```

```

16 Size of set of large itemsets L(4): 1
17
18 Best rules found:
19
20 1. fraldas=não 6 ==> cerveja=não 6    conf:(1)
21 2. ovos=sim 5 ==> cerveja=não 5    conf:(1)
22 3. leite=sim 5 ==> cerveja=não 5    conf:(1)
23 4. leite=sim ovos=sim 4 ==> farinha=não cerveja=não 4    conf:(1)
24 5. leite=sim farinha=não 4 ==> ovos=sim cerveja=não 4    conf:(1)
25 6. ovos=sim farinha=não 4 ==> leite=sim cerveja=não 4    conf:(1)
26 7. leite=sim ovos=sim farinha=não 4 ==> cerveja=não 4    conf:(1)
27 8. leite=sim ovos=sim cerveja=não 4 ==> farinha=não 4    conf:(1)
28 9. leite=sim farinha=não cerveja=não 4 ==> ovos=sim 4    conf:(1)
29 10. ovos=sim farinha=não cerveja=não 4 ==> leite=sim 4    conf:(1)

```

Alguns pontos de interesse da listagem 2.11 são:

- O resultado mostra alguns dos valores *default* para os parâmetros do algoritmo.
- Entre as linhas 10 e 16 os tamanhos dos conjuntos de *itemsets* com suporte mínimo são mostrados.
- A partir da linha 20 as melhores regras de associação são mostradas. Os valores depois dos antecedentes e consequentes das regras são o número de instâncias ou ocorrências para as quais a associação ocorre.  
É interessante observar que o algoritmo usa tanto atributos positivos (“sim”) como negativos (“não”), encontrando muitas regras significativas onde a associação é “Se A não foi comprado então B também não foi comprado”.

Por *default* a execução do algoritmo tenta criar 10 regras de associação, ordenadas por confiança.

**Exercício 2.3:** Execute novamente o algoritmo *Apriori* modificando o suporte mínimo para consideração dos itemsets (argumento *-M*).

**Exercício 2.4:** Execute novamente o algoritmo *Apriori* modificando o número de regras listadas (argumento *-N*). Qual é o número máximo de regras que pode ser criado, com qualquer suporte ou confiança?

**Exercício 2.5:** Execute novamente o algoritmo *Apriori* listando o conjunto de itemsets (argumento *-I*).

### 2.4.1 Discretizando os dados contínuos

Em alguns casos não poderemos executar o algoritmo *Apriori* diretamente em um conjunto de dados. O caso mais freqüente é quando temos dados numéricos, pois o algoritmo só trabalha com dados discretos

ou nominais.

Podemos usar uma rotina do Weka para discretizar dados numéricos. A rotina mais simples é a que discretiza dados usando intervalos iguais, e trocando a categoria numérica por uma nominal correspondente ao intervalo. Esta rotina é implementada no aplicativo `weka.filters.unsupervised.attribute.Discretize`.

Um exemplo de discretização é mostrado a seguir. Consideremos o problema de discretizar o arquivo de dados das flores íris, que tem quatro atributos numéricos. Vamos discretizar estes atributos de forma que cada um dos atributos tenha cinco valores nominais ou discretos possíveis. O comando para discretização é mostrado na listagem 2.12.

Listagem 2.12: Comando para discretizar os atributos do arquivo `iris.arff`

```
1 java weka.filters.unsupervised.attribute.Discretize
2   -R 1,2,3,4 -B 5
3   -i iris.arff -o iris-disc.arff
```

A discretização é feita pela aplicação `Discretize`. A opção `-R` indica quais são os índices dos atributos que devem ser discretizados (vários podem ser indicados, separados por vírgulas). A opção `-B` indica quantos valores discretos teremos no máximo. A opção `-i` indica o arquivo de entrada e a opção `-o` o de saída.

Para comparação, parte do arquivo processado é mostrado na listagem 2.13. O arquivo foi reformatado para melhor visualização.

Listagem 2.13: Parte do arquivo `iris-disc.arff` (discretizado)

```
1 @relation 'iris-weka.filters.unsupervised.attribute.
2   Discretize-B5-M-1.0-R1,2,3,4'
3
4 @attribute sepallength {(-inf-5.02],[5.02-5.74],[5.74-6.46],
5   (6.46-7.18],[7.18-inf)}
6 @attribute sepalwidth {(-inf-2.48],[2.48-2.96],[2.96-3.44],
7   (3.44-3.92],[3.92-inf)}
8 @attribute petallength {(-inf-2.18],[2.18-3.36],[3.36-4.54],
9   (4.54-5.72],[5.72-inf)}
10 @attribute petalwidth {(-inf-0.58],[0.58-1.06],[1.06-1.54],
11   (1.54-2.02],[2.02-inf)}
12 @attribute class {Iris-setosa,Iris-versicolor,Iris-virginica}
13
14 @data
15
16 (5.02-5.74],[3.44-3.92],[(-inf-2.18],[(-inf-0.58],[Iris-setosa
```

```
17 (-inf-5.02],(2.96-3.44],(-inf-2.18],(-inf-0.58],Iris-setosa
18 (-inf-5.02],(2.96-3.44],(-inf-2.18],(-inf-0.58],Iris-setosa
19 (-inf-5.02],(2.96-3.44],(-inf-2.18],(-inf-0.58],Iris-setosa
```

Com os atributos discretizados podemos executar o algoritmo *Apriori* nos dados das íris. O comando está mostrado na listagem 2.14.

Listagem 2.14: Executando o algoritmo *Apriori* pela linha de comando (arquivo `iris-disc.arff`)

```
1 java weka.associations.Apriori -t iris-disc.arff
```

O resultado da execução é mostrado na listagem 2.14

Listagem 2.15: Resultado da execução do algoritmo *Apriori* (arquivo `iris-disc.arff`)

```
1 Apriori
2 =====
3
4 Minimum support: 0.3
5 Minimum metric <confidence>: 0.9
6 Number of cycles performed: 14
7
8 Generated sets of large itemsets:
9
10 Size of set of large itemsets L(1): 8
11
12 Size of set of large itemsets L(2): 3
13
14 Size of set of large itemsets L(3): 1
15
16 Best rules found:
17
18 1. petallength='(-inf-2.18]' 50 ==>
19     class=Iris-setosa 50     conf:(1)
20 2. class=Iris-setosa 50 ==>
21     petallength='(-inf-2.18]' 50     conf:(1)
22 3. petalwidth='(-inf-0.58]' 49 ==>
23     petallength='(-inf-2.18]' class=Iris-setosa 49     conf:(1)
24 4. petallength='(-inf-2.18]' petalwidth='(-inf-0.58]' 49 ==>
25     class=Iris-setosa 49     conf:(1)
26 5. petalwidth='(-inf-0.58]' class=Iris-setosa 49 ==>
27     petallength='(-inf-2.18]' 49     conf:(1)
28 6. petalwidth='(-inf-0.58]' 49 ==>
29     class=Iris-setosa 49     conf:(1)
```

```

30 7. petalwidth='(-inf-0.58]' 49 ==>
31     petallength='(-inf-2.18]' 49     conf:(1)
32 8. petallength='(-inf-2.18]' 50 ==>
33     petalwidth='(-inf-0.58]' class=Iris-setosa 49 conf:(0.98)
34 9. class=Iris-setosa 50 ==>
35     petallength='(-inf-2.18]' petalwidth='(-inf-0.58]' 49 conf:(0.98)
36 10. petallength='(-inf-2.18]' class=Iris-setosa 50 ==>
37     petalwidth='(-inf-0.58]' 49     conf:(0.98)

```

**Exercício 2.6:** Compare os resultados da classificação das íris obtido com o algoritmo *Apriori* com o resultado obtido com o algoritmo J4.8. Existe alguma regra correlata?

### 3 Incluindo classes do Weka em aplicações em Java

Nesta seção veremos como incluir classes do Weka em aplicações. Não vou escrever muito, vou mais colocar o código como exemplo. **Falta completar esta parte.**

Antes de mais nada, vejamos algumas classes interessantes do Weka:

- `FastVector` é uma implementação rápida de vetores (*arrays* de tamanho mutável).
- `Instances` é o mapeamento para memória de um arquivo `.ARFF`. Contém informações sobre a relação, atributos e dados.
- `Instance` contém informações sobre uma instância (vetor) de dados.
- `Attribute` contém informações sobre um atributo de uma base de dados qualquer.

Listagem 3.1: Lendo o conteúdo de um arquivo `.arff`

```

1  import java.io.FileReader;
2  import java.io.IOException;
3  import weka.core.Attribute;
4  import weka.core.Instance;
5  import weka.core.Instances;
6
7  public class LeArff
8  {
9      public static void main(String[] args) throws IOException
10     {
11         // Read the instances from a file.
12         FileReader reader =
13             new FileReader("/home/rafael/InpeHP/javadm/datasets/iris-missing.arff");
14         Instances instances = new Instances(reader);
15         // Get the relation name.
16         System.out.println(instances.relationName());
17         // Get the number of attributes.
18         System.out.println(instances.numAttributes()+" attributes");
19         // Show the attributes.
20         for(int i=0;i<instances.numAttributes();i++)
21             {

```

```

22     String name = instances.attribute(i).name();
23     int type = instances.attribute(i).type();
24     String typeName = "";
25     switch(type)
26     {
27         case Attribute.NUMERIC: typeName = "Numeric"; break;
28         case Attribute.NOMINAL: typeName = "Nominal"; break;
29         case Attribute.STRING: typeName = "String"; break;
30     }
31     System.out.println(name+" type "+typeName);
32 }
33 // Show the data.
34 for(int i=0;i<instances.numInstances();i++)
35 {
36     Instance instance = instances.instance(i); // ugh
37     System.out.println((i+1)+": "+instance+
38         " missing? "+instance.hasMissingValue());
39 }
40 } // end main
41 } // end class

```

Devemos lembrar que para executar uma aplicação com Weka a partir da linha de comando devemos indicar o caminho dos pacotes usados ou usar a variável de ambiente CLASSPATH (veja listagens 2.2 ou 2.3), **mas o path para a aplicação deve também ser indicado**. Para executar a aplicação na listagem 3.1 podemos usar o comando mostrado na listagem 3.2.

Listagem 3.2: Criando vetores de dados e armazenando em um arquivo .arff

```

1 java -cp ./usr/local/weka-3-4-4/weka.jar LeArff

```

Para criar um arquivo .ARFF com dados obtidos de outras fontes devemos primeiro criar a estrutura do arquivo, depois populá-lo com os vetores de dados, como mostrado no exemplo da listagem 3.3. Os dados serão aproximadamente correspondentes ao que vemos na figura 1.

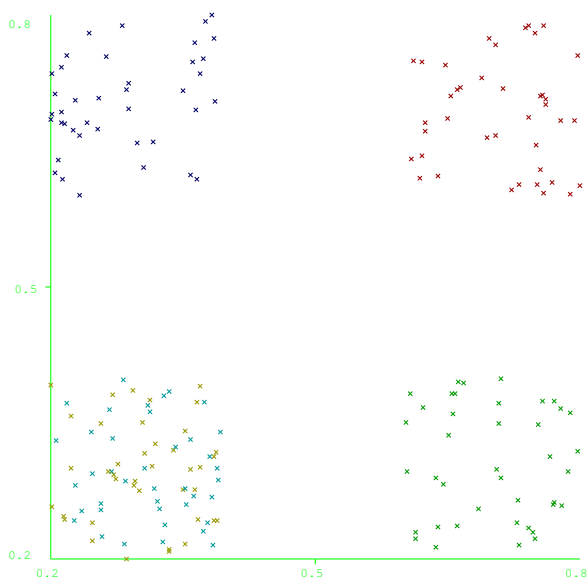


Figura 1: Dados aleatórios para cinco classes

### Listagem 3.3: Criando vetores de dados e armazenando em um arquivo .arff

```
1 import weka.core.Attribute;
2 import weka.core.FastVector;
3 import weka.core.Instance;
4 import weka.core.Instances;
5
6 public class CriaArff
7 {
8     public static void main(String[] args)
9     {
10        // First let's create the attributes.
11        Attribute x = new Attribute("x");
12        Attribute y = new Attribute("y");
13        // Third attribute is nominal.
14        FastVector classesLabels = new FastVector(5);
15        classesLabels.addElement("A");
16        classesLabels.addElement("B");
17        classesLabels.addElement("C");
18        classesLabels.addElement("D");
19        classesLabels.addElement("E");
20        Attribute classes = new Attribute("classes",classesLabels);
21        // Create a vector of attributes information.
22        FastVector attributes = new FastVector(3);
23        attributes.addElement(x);
24        attributes.addElement(y);
25        attributes.addElement(classes);
26        // Create an empty set of instances.
27        Instances instances = new Instances("random",attributes,0);
28        // Create 200 instances of more-or-less random data.
29        for(int i=0;i<40;i++)
30        {
31            Instance inst = new Instance(3);
32            // For class A.
33            inst.setValue(x,0.2+0.2*Math.random());
34            inst.setValue(y,0.6+0.2*Math.random());
35            inst.setValue(classes,"A");
36            instances.add(inst);
37            // For class B.
38            inst.setValue(x,0.6+0.2*Math.random());
39            inst.setValue(y,0.6+0.2*Math.random());
40            inst.setValue(classes,"B");
41            instances.add(inst);
42            // For class C.
43            inst.setValue(x,0.6+0.2*Math.random());
44            inst.setValue(y,0.2+0.2*Math.random());
45            inst.setValue(classes,"C");
46            instances.add(inst);
47            // For classes D and E.
48            inst.setValue(x,0.2+0.2*Math.random());
49            inst.setValue(y,0.2+0.2*Math.random());
50            inst.setValue(classes,"D");
51            instances.add(inst);
52            inst.setValue(x,0.2+0.2*Math.random());
53            inst.setValue(y,0.2+0.2*Math.random());
54            inst.setValue(classes,"E");
55            instances.add(inst);
56        }
57        // Here is the ARFF file contents. Redirect it or use
58        // BufferedWriter.write() to save it to a file.
59        System.out.println(instances);
60    }
}
```

}

Para o próximo exemplo vamos ver como executar o algoritmo J4.8 de dentro de uma aplicação. Podemos observar alguns problemas de *design* das classes do Weka.

#### Listagem 3.4: Executando o algoritmo J4.8 em uma aplicação (e avaliando o resultado)

```

1  import java.io.FileReader;
2  import weka.classifiers.Classifier;
3  import weka.classifiers.Evaluation;
4  import weka.classifiers.trees.j48.C45ModelSelection;
5  import weka.classifiers.trees.j48.C45PruneableClassifierTree;
6  import weka.core.Instance;
7  import weka.core.Instances;
8
9  public class ManualJ48 extends Classifier
10 {
11     // The tree.
12     private C45PruneableClassifierTree c45tree;
13
14     // This method must be written since we extends the abstract
15     // class Classifier.
16     public void buildClassifier(Instances instances) throws Exception
17     {
18         // Create a model selection for the tree (allowing the creation of subsets
19         // with at least three instances).
20         C45ModelSelection model = new C45ModelSelection(3,instances);
21         // Create the classifier and build the tree using those instances.
22         // The tree will be unpruned (therefore the confidence factor will be zero),
23         // Subtree raising will not be performed (since tree will be unpruned).
24         // What its cleanup data, anyway? Documentation is weak :-(
25         c45tree = new C45PruneableClassifierTree(model,false,0,false,true);
26         c45tree.buildClassifier(instances);
27     }
28
29     // Prints the tree.
30     public void printTree()
31     {
32         System.out.println(c45tree);
33     }
34
35     // We *must* implement this method otherwise the Evaluation instance will
36     // freak out.
37     public double classifyInstance(Instance instance) throws Exception
38     {
39         return c45tree.classifyInstance(instance);
40     }
41
42     // The entry point on the app.
43     public static void main(String[] args)
44     {
45         try
46         {
47             // Read the instances from a file.
48             FileReader reader =
49                 new FileReader("/usr/local/weka-3-4-4/data/weather.arff");
50             Instances instances = new Instances(reader);
51             instances.setClassIndex(4);
52             // Create the tree and classifier.
53             ManualJ48 thisClassifier = new ManualJ48();

```

```

54     thisClassifier.buildClassifier(instances);
55     // Print the tree.
56     thisClassifier.printTree();
57     // Let's evaluate the tree.
58     Evaluation evaluation = new Evaluation(instances);
59     evaluation.evaluateModel(thisClassifier,instances);
60     // How many correct instances?
61     System.out.print(evaluation.correct()+"/");
62     // How many instances?
63     System.out.println(instances.numInstances());
64 }
65 catch (Exception e)
66 {
67     System.err.println(e.getMessage());
68 }
69
70 }
71 }

```

### Listagem 3.5: Executando o algoritmo KMeans em uma aplicação (duas vezes)

```

1  import java.io.FileReader;
2  import weka.clusterers.SimpleKMeans;
3  import weka.core.Instances;
4
5  // This time we will extend the SimpleKMeans algorithm, it is so much
6  // simpler this way!
7  public class ManualKMeans extends SimpleKMeans
8  {
9
10     public static void main(String[] args)
11     {
12         try
13         {
14             // Read the instances from a file.
15             FileReader reader = new FileReader("2dim.arff");
16             Instances instances = new Instances(reader);
17             // We want to ignore the class of the data vectors.
18             instances.deleteAttributeAt(2);
19             // Create the clusterer.
20             ManualKMeans thisClusterer = new ManualKMeans();
21             thisClusterer.setNumClusters(4);
22             thisClusterer.buildClusterer(instances);
23             System.out.println(thisClusterer.getSquaredError());
24             // Try again with a different number of clusters.
25             thisClusterer.setNumClusters(5);
26             thisClusterer.buildClusterer(instances);
27             System.out.println(thisClusterer.getSquaredError());
28         }
29         catch (Exception e)
30         {
31             System.out.println(e.getMessage());
32         }
33     } // end main
34 } // end class

```

### Listagem 3.6: Executando o algoritmo KMeans em uma aplicação (e classificando instâncias dos dados)

```

1 import java.io.FileReader;
2 import weka.clusterers.SimpleKMeans;
3 import weka.core.Attribute;
4 import weka.core.Instance;
5 import weka.core.Instances;
6
7 // This time we will extend the SimpleKMeans algorithm, it is so much
8 // simpler this way!
9 public class KMeansClassifier extends SimpleKMeans
10 {
11
12     public static void main(String[] args)
13     {
14         try
15         {
16             // Read the instances from a file.
17             FileReader reader = new FileReader("2dim.arff");
18             Instances instances = new Instances(reader);
19             // We'll create a deep copy of the instances.
20             Instances originalInstances = new Instances(instances);
21             // We want to ignore the class of the data vectors for
22             // the clustering.
23             instances.deleteAttributeAt(2);
24             // Create the clusterer.
25             KMeansClassifier thisClusterer = new KMeansClassifier();
26             int nclusters = 4;
27             thisClusterer.setNumClusters(nclusters);
28             thisClusterer.buildClusterer(instances);
29             // Now we will classify each instance on the dataset and verify its class.
30             // We will need the possible values for the class.
31             Attribute classes = originalInstances.attribute(2);
32             int nclasses = classes.numValues();
33             // First we create a sort-of confusion matrix of KxK elements.
34             int[][] matrix = new int[nclasses][nclusters];
35             for(int i=0;i<originalInstances.numInstances();i++)
36             {
37                 // Get an instance of the original dataset.
38                 Instance instance = originalInstances.instance(i);
39                 // Get the class as a single character.
40                 char itsClass = classes.value((int)instance.value(2)).charAt(0);
41                 // Cut the relation between the dataset and the instance (!)
42                 instance.setDataset(null);
43                 // Cut the class.
44                 instance.deleteAttributeAt(2);
45                 // Get its cluster.
46                 int c = thisClusterer.clusterInstance(instance);
47                 System.out.println("Instance "+(i+1)+" class "+itsClass+
48                     " assigned to cluster "+c);
49                 matrix[itsClass-'A'][c]++;
50             }
51             // Print the confusion matrix.
52             System.out.print(" ");
53             for(int c2=0;c2<nclusters;c2++)
54                 System.out.print(" "+c2+" ");
55             System.out.println();
56             for(int c1=0;c1<nclasses;c1++)
57             {
58                 System.out.print((char)('A'+c1)+" ");
59                 for(int c2=0;c2<nclusters;c2++)
60                     System.out.print(matrix[c1][c2]+" ");
61                 System.out.println();

```

```
62     }
63   }
64   catch (Exception e)
65   {
66     System.out.println("Exception: "+e.getMessage());
67     e.printStackTrace();
68   }
69 } // end main
70 } // end class
```

## Referências

- [1] Weka 3: Data Mining Software in Java (<http://www.cs.waikato.ac.nz/ml/weka/index.html>, visitado em Abril de 2005).
- [2] Ian H. Witten, Eibe Frank, *Data Mining – Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 2000.