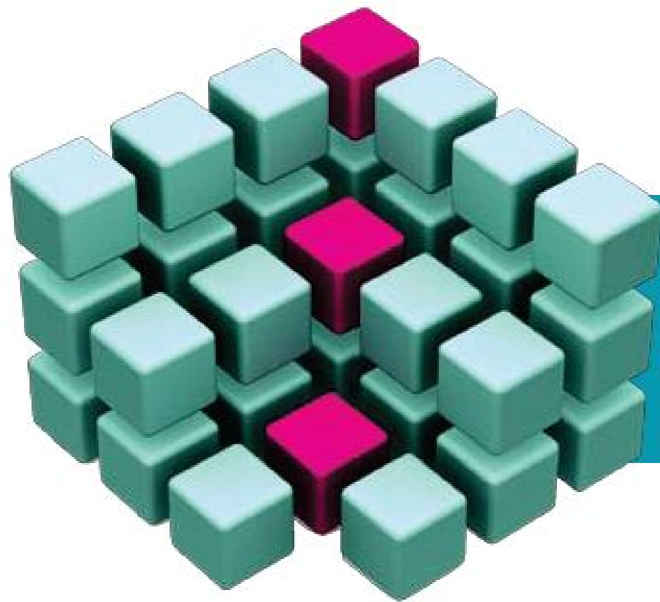


Mineração de Dados em Java: Weka

Como embutir algoritmos de mineração de dados em suas aplicações em Java.

Rafael Santos
(rafael.santos@lac.inpe.br)
É doutor em Inteligência Artificial pelo Instituto Tecnológico de Kyushu, Japão. É tecnologista do Instituto Nacional de Pesquisas Espaciais, atuando em pesquisa e desenvolvimento de técnicas de mineração de dados e processamento de imagens. É autor do livro "Introdução à Programação Orientada a Objetos usando Java" e de várias palestras e tutoriais sobre Java e aplicações científicas.

Mineração de dados (ou Data Mining) é o nome dado a um conjunto de processos e algoritmos cujo objetivo é extrair conhecimento de alto nível a partir de um grande volume de dados. O conhecimento obtido pode ser usado para fazer previsões e análises, com aplicações em muitas áreas comerciais, científicas e de segurança.

O software Weka (Waikato Environment for Knowledge Analysis), desenvolvido por pesquisadores da Universidade de Waikato na Nova Zelândia, é composto de um ambiente gráfico para testes e avaliação de algoritmos de mineração de dados. Este software também contém uma API (Application Programming Interface) que pode ser usada em aplicações em Java para embutir algoritmos de mineração de dados em suas aplicações. Neste artigo, mostraremos como alguns algoritmos de mineração de dados podem ser facilmente embutidos em aplicações em Java.

Mineração de dados é o nome genérico dado a técnicas e algoritmos que, a partir de uma massa de dados, extraem conhecimentos que permitem a tomada de decisão em relação a estes dados. Mineração de dados (também conhecida popularmente como Data Mining) não pode ser confundida com técnicas de criação de relatórios através de consulta (SQL e OLAP), sendo bem mais complexa e possibilitando análises bem mais interessantes. Técnicas e aplicações de mineração de dados têm se tornado popular recentemente por causa do barateamento do custo de computadores, em particular, do aumento na capacidade de coleta e armazenamento de dados. Bancos de dados de aplicações comerciais que contêm terabytes armazenados não são raros, e várias aplicações científicas têm ou planejam ter capacidades de armazenamento medidas em petabytes. Para estas aplicações, o simples armazenamento dos dados não basta, é necessário ter técnicas e mecanismos de análise dos mesmos.

Embora alguns algoritmos que podem ser usados para mineração de dados sejam relativamente clássicos e bem conhecidos, sua implementação correta pode ser muito complexa ou custosa. Pesquisadores da Universidade de Waikato, na Nova Zelândia, desenvolveram um pacote de software chamado Weka (Waikato Environment for Knowledge Analysis) que contém uma aplicação com interface gráfica em que um usuário pode experimentar com os diversos algoritmos presentes no pacote para tentar extrair

conhecimento de seus dados. Além da interface gráfica para exploração, os algoritmos do pacote podem ser executados a partir de linha de comando e embutidos em aplicações em Java.

Este artigo apresenta conceitos introdutórios de mineração de dados e uma breve descrição de um dos algoritmos mais interessantes e flexíveis para extração de conhecimento a partir de dados. Como este algoritmo é implementado na API do software Weka, usaremos esta implementação para criar uma aplicação com dados simulados. Os exemplos apresentados podem ser facilmente estendidos para aplicação em outros problemas.

O que é mineração de dados?

Mineração de dados é um dos componentes do processo de descoberta de conhecimento em bancos de dados (KDD, Knowledge Discovery in Databases). Este processo consiste em vários passos iterativos para levantamento, seleção, análise, mineração e sumarização de dados para obter conhecimento a partir dos mesmos. A parte de mineração dos dados é justamente a que usa vários algoritmos baseados em estatística, lógica, inteligência artificial, etc. para obter informações sumarizadas sobre os dados.

Mineração de dados não é uma técnica única: existem várias abordagens para a extração de conhecimentos a partir de bases de dados, e estas técnicas são solidamente fundamentadas em conceitos de estatística, matemática e computação aplicada (inteligência artificial, reconhecimento de padrões, aprendizado por máquina, computação de alto desempenho, bancos de dados, computação gráfica e visualização, etc.). Por usar técnicas e algoritmos que utilizam dados para obter conhecimento, é importante reconhecer que mineração de dados não é uma técnica mágica: os dados a serem minerados devem ser adequadamente levantados, selecionados, filtrados e pré-processados, caso contrário o resultado será comprometido (lei Garbage In, Garbage Out, ou seja, se entra lixo, sai lixo). Uma conseqüência direta deste requisito é que mineração de dados não deve ser "feita às cegas" é imprescindível o conhecimento sobre a natureza dos dados para minerá-los.

Para facilitar a compreensão de alguns exemplos e conceitos, consideraremos que um banco ou base de dados a ser analisado por técnicas e algoritmos de mineração de dados é composto de uma única tabela (ou relação). Cada coluna nesta tabela é um atributo, e cada linha é um dado ou instância. Na tabela, podemos ter atributos de diversos tipos, os mais freqüentes sendo numérico e nominal (ou discreto). Podemos imaginar que em algumas aplicações teremos dados com atributos incompletos, e que em alguns casos teremos informações sobre alguns (mas não todos) dados que indicam qual é a categoria a qual estes dados pertencem por exemplo, em uma tabela sobre clientes de um banco podemos ter um atributo que indica se o cliente foi avaliado por um gerente como sendo bom candidato a um serviço, mas como nem todos os clientes podem ter sido avaliados, nem todos terão valores conhecidos para aquele atributo.

Algumas categorias de técnicas de mineração de dados são sumarizadas a seguir:

- **Classificação:** predição de uma classe ou categoria discreta a partir de atributos de entrada. Algoritmos de classificação analisam dados para os quais as categorias discretas são conhecidas, tentando criar funções que separem as diferentes categorias usando os valores dos atributos. Depois da criação das funções de separação, as mesmas são usadas para determinar a classe de dados sem categoria explícita. Exemplos de classificação são previsão de categorias discretas como usadas em sistemas de detecção de intrusão em redes ou sistemas de tomada de decisões categóricas em instituições financeiras;

- **Associação:** descoberta de co-ocorrências entre elementos em grandes conjuntos de dados. Algoritmos de descoberta de regras de associação analisam os atributos dos dados procurando conjuntos de atributos que ocorram freqüentemente de acordo com uma métrica definida. Se considerarmos cada dado na nossa relação como sendo correspondente a um evento qualquer, podemos verificar que valores de atributos ocorrem em conjunto. Um exemplo clássico de descoberta de regras de associação é a chamada análise de carrinhos de compras: cada dado na relação corresponde à lista de itens que foram comprados em uma determinada transação, e a análise pode determinar que produtos são comprados em conjunto com alguma freqüência;

- **Agrupamento:** descoberta de grupos naturais em que dados que estão em um mesmo grupo são considerados semelhantes e dados em grupos distintos são considerados diferentes. Algoritmos de agrupamento tentam criar grupos de dados usando diferentes métricas de similaridade, geralmente de forma iterativa até que os grupos criados sejam considerados adequados, e é usado principalmente com dados puramente numéricos. Exemplos de aplicação destes tipos de algoritmos são os usados para segmentação de clientes em diferentes categorias para a exploração de similaridades ainda não conhecidas. Também associada a esta técnica é a possibilidade de descoberta de dados que não se assemelham a nenhum dos grupos criados (ou grupos que foram criados, mas apresentam características especiais), que podem ser considerados exceções ou raridades e investigados adequadamente;

- **Regressão ou predição numérica:** descoberta de um valor associado à cada dado e que pode ser calculado ou inferido a partir dos valores de seus atributos. É similar ao conceito de classificação, mas ao invés de tentar descobrir qual categoria discreta está associada ao dado, usa-se uma categoria numérica. Um exemplo clássico é o de previsão de séries temporais: a partir de um conjunto de dados coletados ao longo do tempo, pode tentar se prever o comportamento futuro dos dados. Um exemplo bem interessante (e complexo, e sujeito a erros) é a aplicação de técnicas de mineração de dados para previsão de indicadores econômicos e financeiros.

Existem vários exemplos de sucesso de aplicação de técnicas de mineração de dados. Alguns conhecidos publicamente são da Verizon Wireless, que reduziu o número de clientes desistentes de seus serviços através da análise dos perfis dos desistentes e modificação dos planos para assinantes; e o da rede Casino de supermercados franceses, que criou cartões de fidelidade para coletar dados de consumo de seus clientes para análise de perfis de consumo (é bem possível que você faça compras em um supermercado desta rede sem saber!). Existem também vários casos que são relacionados com segmentação de clientes em grupos para criação de marketing dirigido, promoções especiais, simulação de mudanças no comportamento de consumidores, etc.

É interessante notar que empresas de software e prestadores de consultoria em mineração de dados apresentam resultados interessantes em seus sites, evidenciando o sucesso das técnicas, mas somente com informações mínimas, sem comentar sobre exatamente como os resultados foram obtidos e qual foi o custo de implementação da solução. Também interessante é observar que existem raros casos documentados de fracasso na aplicação de técnicas de mineração de dados a problemas reais, embora seja fácil imaginar esta possibilidade.

Um breve exemplo

Para melhor compreensão dos conceitos, vejamos um exemplo simples de tarefa para a qual técnicas de mineração de dados podem obter resultados interessantes. Consideremos um shopping center localizado próximo a algumas instituições de ensino superior. Os principais consumidores da praça de alimentação deste shopping center são, na sua maioria, estudantes das instituições. O responsável pela praça de alimentação quer fazer uma análise do perfil dos estudantes para tentar verificar que tipo de restaurante é mais popular para cada tipo de perfil para ver possíveis melhorias e campanhas de marketing dirigido. Embora este exemplo seja completamente fictício (e bastante simplificado para melhor compreensão), não é difícil imaginar uma aplicação real com características semelhantes.

Uma pesquisa simples levantou, para um grupo de 30 alunos, os cursos, sexo, tipo de comida preferida e gasto mensal com alimentação no shopping center. A tarefa é, então, tentar descobrir que características determinam o perfil dos consumidores e que informações são possíveis extrair do banco de dados criado com o resultado da pesquisa. Os dados coletados pela pesquisa são mostrados na figura 1.

Uma pergunta que pode ser respondida com técnicas de mineração de dados é "que atributos determinam a preferência por algum tipo de restaurante? em outras palavras, é possível determinar, a partir do perfil do cliente, qual tipo de comida ele prefere? É possível adivinhar o tipo de comida preferido de, por exemplo, uma estudante de computação que pretende gastar 200 reais por mês com alimentação ou de um estudante de direito que pensa em gastar 300 reais?

Esta tarefa de predição de categorias pode ser feita usando-se um algoritmo de classificação, cuja tarefa seria justamente criar um conjunto de regras ou fórmulas a partir dos dados. Algoritmos de classificação têm duas etapas: na etapa de treinamento, as regras ou fórmulas serão criadas através da análise dos dados que têm categorias conhecidas (no exemplo, os da pesquisa). Para treinar o algoritmo, é preciso identificar qual dos atributos dos dados será usado para classificação. Este atributo deve ser discreto e corresponder à pergunta que queremos fazer sobre os dados.

Depois da criação das regras ou fórmulas elas podem ser aplicadas para descobrir qual é a categoria para dados cujas categorias ainda são desconhecidas – esta é a etapa de classificação propriamente dita.

Espera-se que seja possível deduzir as regras e fórmulas com um conjunto limitado de dados com classes conhecidas, e que com as regras ou fórmulas adequadas seja possível a classificação de um grande número de dados para os quais a classe é desconhecida (caso contrário seria mais vantajoso refazer a pesquisa com mais entrevistados).

Para adiantar, vejamos o resultado da classificação dos dados mostrados na figura 1 com um algoritmo que cria uma árvore de decisão. Uma árvore de decisão é um conjunto de regras, estruturadas de forma hierárquica, onde cada nó é uma decisão a ser tomada sobre os valores dos atributos e cada folha é uma classe. Árvores de decisão são similares também a sistemas especialistas, que usam regras para determinar classes, e são criadas através da análise estatística recursiva do conjunto de dados, tentando escolher atributos que separam os conjuntos de dados de forma mais homogênea. Um algoritmo bastante conhecido e implementado no Weka é o J4.8 (baseado em um algoritmo clássico chamado C4.5), que pode ser usado a partir do ambiente Explorer do Weka, a partir da linha de comando ou (o que nos interessa mais) embutido em uma aplicação em Java.

A figura 2 mostra de forma gráfica a árvore de decisão criada usando os dados de entrada (figura 1) e selecionando o atributo Comida como resultante.

Curso	Sexo	Comida	Gasto
Computação	M	Fast-food	180
Computação	M	Fast-food	220
Computação	M	Fast-food	240
Computação	M	Fast-food	210
Computação	M	Fast-food	190
Computação	M	Mineira	340
Computação	F	Italiana	320
Computação	F	Italiana	340
Computação	F	Japonesa	280
Computação	F	Italiana	330
Economia	M	Fast-food	170
Economia	M	Fast-food	200
Economia	M	Mineira	330
Economia	M	Japonesa	270
Economia	M	Japonesa	290
Economia	F	Italiana	290
Economia	F	Italiana	280
Economia	F	Italiana	260
Economia	F	Fast-food	190
Economia	F	Fast-food	170
Direito	M	Fast-food	260
Direito	M	Italiana	350
Direito	M	Japonesa	310
Direito	M	Mineira	360
Direito	M	Mineira	370
Direito	F	Japonesa	320
Direito	F	Japonesa	350
Direito	F	Japonesa	360
Direito	F	Japonesa	380
Direito	F	Mineira	370

Figura 1. Tabela relacional contendo dados para mineração.

A interpretação de uma árvore de decisão e seu uso para classificar dados com categoria ainda desconhecida é simples: os nós (representados por elipses) são condições que devem ser aplicadas aos atributos correspondentes dos dados, e as classes (representadas por retângulos) são as conclusões alcançadas. Para decidir sobre uma categoria para uma classe, basta percorrer a árvore, iniciando no primeiro nó, seguindo o caminho correspondente às decisões tomadas em função dos valores dos atributos e terminando em uma folha da árvore.

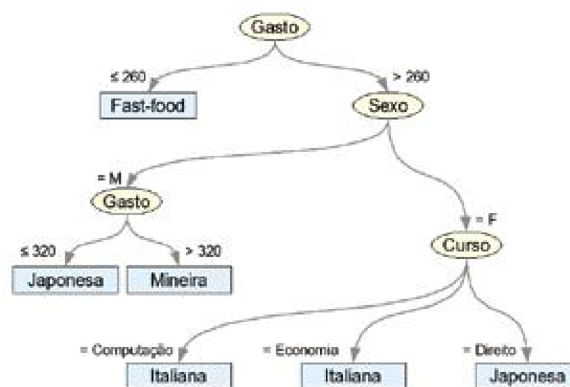


Figura 2. Árvore de decisão.

Como exemplo, considerando a árvore de decisão mostrada na figura 2 e a pergunta "qual é o tipo preferido de comida de uma estudante de computação que pretende gastar 200 reais por mês com alimentação? podemos ver que o caminho a ser percorrido passa pelo nó "Gasto", em que a decisão é "<= 260", o que indica que a comida preferida provavelmente será Fast-food. Em outro exemplo, usando a pergunta "qual será o tipo de comida preferido de um estudante de direito que pensa em gastar 300 reais?" seguiremos a árvore começando novamente no nó "Gasto , mas usando a decisão "> 260", o que nos leva ao nó "Sexo", em que tomamos a decisão "=M" e finalmente a um outro nó "Gasto no qual usamos a decisão "<= 320" para chegar à conclusão de que a comida preferida deverá ser japonesa.

Alguns comentários relevantes sobre árvores de decisão:

• alguns atributos podem não ser usados na construção da árvore (por não serem relevantes para a discriminação das classes);

• decisões sobre alguns atributos podem ser feitas em mais de um ponto da árvore (como ocorre com o atributo "Gasto no exemplo dado);

• uma árvore de decisão pode conter erros de classificação, que podem ser aceitáveis dependendo da tarefa. Como erro de classificação consideremos dois dados com valores de atributos idênticos, mas exceto pelo valor do atributo considerado para classificação (no exemplo dado, poderíamos ter dois estudantes do mesmo curso, sexo, com o mesmo orçamento, mas com preferências de tipo de comida diferentes). Estes erros de classificação podem ser informados pelo algoritmo de criação da árvore;

• parâmetros usados para a construção da árvore podem influenciar no número de nós criados. Uma árvore com mais nós pode dar resultados mais precisos na classificação, mas em contrapartida ser de interpretação mais complexa.

Criando a árvore de decisão usando Java e a API do Weka

Com os conceitos vistos, podemos ter um primeiro exemplo de criação de aplicação em Java que use a API do Weka. Para simplificar, usaremos os dados do exemplo mostrado na seção anterior.

A instalação da API do Weka é simples, basta copiar o software (o link é mostrado nas referências), instalá-lo e anotar onde o arquivo weka.jar foi instalado. Inclua este arquivo no CLASSPATH da sua IDE.

A API do Weka tem classes que facilitam muito a importação de dados de diversas fontes. O arquivo-padrão para o Weka tem formato .arff (Attribute-Relation File Format) ou .xrff (eXtensible attribute-Relation File Format) em versões mais novas. Ambos formatos são texto, mas o Weka permite uso de bancos de dados como fontes de dados. O formato .arff é mais adequado para aplicações simples. Um arquivo .arff tem as seguintes características:

• um nome de relação, identificado com a palavra-chave @RELATION seguido de um nome simples para a relação;

• um ou mais atributos, identificados com a palavra-chave @ATTRIBUTE, seguido do nome do atributo, seguido da palavra-chave NUMERIC (para atributos numéricos) ou da lista de valores aceitáveis para atributos discretos, entre chaves;

• uma seção de dados, identificada com a palavra-chave @DATA, seguido de várias linhas, em que cada linha corresponde a um dado. Cada linha deve conter os valores dos atributos na ordem em que foram declarados, com tipos compatíveis.

Um exemplo de arquivo .arff, correspondente aos dados mostrados na figura 1, é mostrado na Listagem 1.

Com dados no formato .arff, podemos facilmente escrever uma aplicação que cria a árvore de decisão correspondente. Os passos desta aplicação são:

1. ler o arquivo .arff para uma instância da classe Instances;
2. identificar qual atributo será o usado para classificação (no caso, o atributo Comida);
3. criar uma instância da classe J48 e criar o classificador para ela;
4. manipular a instância da classe J48 (para, por exemplo, obter representações da árvore e estatísticas de classificação).

Uma aplicação completa que cria a árvore a partir dos dados em um arquivo .arff, e que imprime a árvore de forma textual é mostrada na Listagem 2. O resultado da aplicação é mostrado na Listagem 3.

A Listagem 3 mostra que a árvore do tipo J48 não-podada criada pela classificação dos dados da Listagem 1. Uma árvore pode ser podada para obter uma representação mais simplificada. A poda (e outras opções usadas para a criação da árvore) podem ser passadas para a instância da classe J48 através do método setOptions daquela classe.

Listagem 1. Arquivo .arff correspondente aos dados mostrados na figura 1.

```
@RELATION mercado

@ATTRIBUTE Curso {Computação, Economia,
Direito}
@ATTRIBUTE Sexo {M,F}
@ATTRIBUTE Comida {Fast-food,Italiana,Japon
esa,Mineira}
@ATTRIBUTE Gasto numeric

@data
Computação,M,Fast-food,180
Computação,M,Fast-food,220
Computação,M,Fast-food,240
Computação,M,Fast-food,210
Computação,M,Fast-food,190
Computação,M,Mineira,340
Computação,F,Italiana,320
Computação,F,Italiana,340
Computação,F,Japonesa,280
Computação,F,Italiana,330
Economia,M,Fast-food,170
Economia,M,Fast-food,200
Economia,M,Mineira,330
Economia,M,Japonesa,270
Economia,M,Japonesa,290
Economia,F,Italiana,290
Economia,F,Italiana,280
Economia,F,Italiana,260
Economia,F,Fast-food,190
Economia,F,Fast-food,170
Direito,M,Fast-food,260
Direito,M,Italiana,350
Direito,M,Japonesa,310
Direito,M,Mineira,360
Direito,M,Mineira,370
Direito,F,Japonesa,320
Direito,F,Japonesa,350
Direito,F,Japonesa,360
Direito,F,Japonesa,380
Direito,F,Mineira,370
```

A Listagem 3 mostra a árvore criada com algumas estatísticas básicas. A árvore é mostrada como texto indentado, com os nós em níveis mais altos mostrados com menor indentação. Cada decisão é seguida ou de uma classe com valores de acerto ou por outras linhas com outras decisões a serem tomadas.

Os valores de acerto indicam quantos dos dados usados para treinamento podem ser corretamente classificados pela árvore. Por exemplo, a primeira linha da árvore é "Gasto <= 260: Fast-food (11.0/1.0)", indicando a primeira decisão a ser tomada na classificação e informando que esta decisão classificou 11 dados da base original, mas 1 foi classificado incorretamente (como exercício, localize na Listagem 1 quais dados seriam classificados com esta regra, e qual foi classificado incorretamente).

Além da árvore de decisão na forma textual, a aplicação da Listagem 2 mostra também o número de folhas e o tamanho da árvore (total de nós e folhas). Estas e outras informações sobre as dimensões árvore também podem ser obtidas com métodos da classe J48 (measureNumLeaves, measureNumRules, measureTreeSize).

Um método bastante interessante é o toSource, que recebe como argumento uma String e que retorna o código Java correspondente à árvore, codificado de forma estática (hardcoded). A String passada como argumento será o nome da classe usado para a geração do código. A classe terá vários métodos estáticos que serão executados a partir do método classify, que recebe como argumento um array de instâncias de Object que contém os valores dos atributos de um dado a ser classificado. Esta classe pode ser embutida em outras aplicações, de forma independente da API do Weka, para a criação de classificadores. As aplicações devem ser responsáveis, no entanto, pela leitura dos dados a serem classificados e pela interpretação dos resultados (os resultados da classificação são retornados como valores de ponto flutuante correspondentes às classes, no exemplo, o resultado 1.0 de uma classificação corresponde à classe Italiana, e o resultado 3.0 à classe Mineira).

A criação de árvores de decisão usando a API do Weka é simples, e podemos até criar uma aplicação com a árvore embutida usando o método toSource, com as decisões codificadas estaticamente na classe gerada. Esta abordagem pode não ser prática na maioria dos casos por causa da necessidade de compilar a classe gerada para aproveitá-la em aplicações. Para aplicações mais flexíveis, devemos considerar a capacidade de criar a árvore, armazenar a sua estrutura para uso posterior e fazer múltiplas classificações lendo os dados a serem classificados de uma fonte externa. Para isso, devemos escrever a aplicação em duas partes: uma para criar e armazenar (serializar) a árvore de decisão e outra para usar a árvore serializada para classificar novos dados, sem necessidade de compilar código gerado pela árvore de decisão.

A serialização de uma árvore de decisão (instância da classe J48) é simples: basta, após a criação da árvore com o método buildClassifier, usar o trecho de código mostrado na Listagem 4, que serializa a instância árvore da classe J48 através de um ObjectOutputStream.

Com a árvore criada e serializada, podemos escrever uma aplicação que usa a árvore pronta para classificar dados para os quais ainda não temos classes. Este passo é um pouco mais complicado – para classificar um dado com a árvore criada, precisaremos da estrutura dos dados usados para criar a árvore original (mas não dos dados, evidentemente eles já foram processados para a criação da representação pela árvore).

Listagem 2. Aplicação em Java que cria uma árvore de decisão.

```
import weka.classifiers.trees.J48;
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSource;

public class CriaArvoreSimples {
  public static void main(String[] args) throws Exception {
    // Lemos os dados.
    DataSource fonte = new DataSource("dados.arff ");
    Instances dados = fonte.getDataSet();
    // Indicamos o atributo para classificação.
    dados.setClassIndex(2);
    // Criamos a árvore.
    J48 árvore = new J48();
    árvore.buildClassifier(dados);
    // Mostramos a árvore como texto.
    System.out.println(árvore);
  }
}
```

Os passos para a criação de uma base de dados com a estrutura esperada pelo Weka sem usar leitura direta a partir de um arquivo .arff ou .xrff são os seguintes:

1. criamos os atributos usados na base de dados (instâncias da classe Attribute). Atributos numéricos são criados com uma chamada ao construtor da classe usando somente o nome do atributo como argumento. Atributos discretos ou nominais devem ser criados usando o construtor que recebe o nome do atributo e uma lista de valores (rótulos) que podem ser usados para aquele atributo. Esta lista deve ser uma instância de FastVector, uma implementação interna do Weka similar à classe Vector;

2. criamos uma lista (novamente FastVector) dos atributos;

3. usamos a lista de atributos para criar uma instância de Instances, passando para o construtor da classe um nome para a relação, a lista de atributos e uma capacidade inicial;

4. com a instância de Instances (correspondente a toda uma base de dados) podemos criar uma instância de Instance (no singular) que corresponde a somente uma entrada na base de dados (linha da tabela), e popular esta instância de Instance com os dados. Devemos também associar a instância de Instance à de Instances.

Estes passos parecem mais complicados do que são (ainda mais por causa dos nomes usados para as classes), portanto um pequeno programa que cria uma base de dados no formato .arff usando somente código pode esclarecer alguns pontos. O código na Listagem 5 mostra como podemos criar uma base de dados do Weka desta forma (sendo útil para quem precisar escrever programas que importem dados de uma fonte e exportem para o formato e padrão do Weka).

Listagem 3. Resultado da execução da aplicação mostrada na Listagem 2.

```
J48 pruned tree
------------------

Gasto <= 260: Fast-food (11.0/1.0)
Gasto > 260
|  Sexo = M
|  |  Gasto <= 320: Japonesa (3.0)
|  |  Gasto > 320: Mineira (5.0/1.0)
|  Sexo = F
|  |  Curso = Computação: Italiana (4.0/1.0)
|  |  Curso = Economia: Italiana (2.0)
|  |  Curso = Direito: Japonesa (5.0/1.0)

Number of Leaves  :     6

Size of the tree :      10
```

A execução do código da Listagem 5 é mostrada na Listagem 6.

O método toString da classe Instances cria uma saída bastante similar à mostrada na Listagem 1, exceto pelo nome da relação e por mostrar alguns atributos como sinais de interrogação, o que no Weka indica que desconhecemos o valor do atributo para aquele dado.

Alguns dos métodos estáticos da Listagem 5 podem ser reusados para o nosso classificador. Temos, neste ponto, a árvore de decisão serializada, precisamos criar instâncias de Instance para que a árvore as classifique. Estas instâncias devem ser associadas à estrutura da base de dados (classe Instances) através do método setDataSet da classe Instance.

A Listagem 7 mostra uma aplicação baseada

no código da Listagem 5 que faz a classificação de alguns dados (para os quais não temos classe).

Listagem 4. Aplicação em Java que cria uma árvore de decisão e que a serializa.

```
// Serializamos a árvore. Outros mecanismos de
// serialização poderiam ser usados.
ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("arvore.ser"));
oos.writeObject(árvore);
oos.close();
```

Listagem 5. Criando bases de dados no Weka através de código.

```
import weka.core.*;

public class CriaArff {
  // Alguns atributos estáticos.
  private static Attribute curso,sexo,comida,gasto;
  // Esta aplicação cria uma base de dados para o Weka usando somente
  // código (sem ler de arquivos).
  public static void main(String[] args) throws Exception {
    // Criamos uma instância de Instances com as estruturas de dados.
    Instances instâncias = criaInstances();
    // Criamos duas instâncias e associamos à base de dados.
    Instance i1 = new Instance(instâncias.numAttributes());
    i1.setValue(curso,"Computação");
    i1.setValue(sexo,"F");
    i1.setValue(gasto,200.0);
    instâncias.add(i1);
    Instance i2 = new Instance(instâncias.numAttributes());
    i2.setValue(curso,"Direito");
    i2.setValue(sexo,"M");
    i2.setValue(gasto,300.0);
    instâncias.add(i2);
    System.out.println(instâncias);
  }

  private static Instances criaInstances() {
    // Antes de criar a instância de Instances precisamos dos atributos!
    curso = criaAtributo("Curso",new String[]{"Computação","Economia","Direito"});
    sexo = criaAtributo("Sexo",new String[]{"M","F"});
    comida = criaAtributo("Comida",new String[]{"Fast-food","Italiana",
        "Japonesa","Mineira"});
    gasto = new Attribute("Gasto");
    // Agrupamos os atributos.
    FastVector atributos = new FastVector();
    atributos.addElement(curso);
    atributos.addElement(sexo);
    atributos.addElement(comida);
    atributos.addElement(gasto);
    // Criamos a instância de Instances
    Instances desconhecidos = new Instances("desconhecidos",atributos,0);
    return desconhecidos;
  }

  // Este método facilita a criação de atributos discretos (nominais).
  private static Attribute criaAtributo(String nome,String[] rótulos) {
    FastVector rot = new FastVector();
    for(String r:rótulos) rot.addElement(r);
    Attribute attr = new Attribute(nome,rot);
    return attr;
  }
}
```

Listagem 6. Resultado da execução do código da Listagem 5.

```
@relation desconhecidos

@attribute Curso {Computação,Economia,Direito}
@attribute Sexo {M,F}
@attribute Comida {Fast-food,Italiana,Japonesa,Mineira}
@attribute Gasto numeric

@data
Computação,F,?,200
Direito,M,?,300
```

Listagem 7. Classificador por árvore de decisão que usa uma instância serializada de J48.

```
import java.io.*;
import weka.classifiers.trees.J48;
import weka.core.*;

public class ClassificaComArvoreSerializada {
  // Alguns atributos estáticos.
  private static Attribute curso,sexo,comida,gasto;
  // Esta aplicação classifica instâncias usando uma árvore de decisão serializada.
  public static void main(String[] args) throws Exception {
   // Lemos a árvore serializada.
   ObjectInputStream ois = new ObjectInputStream(new FileInputStream("arvore.ser"));
   J48 árvore = (J48)ois.readObject();
   ois.close();
   // Criamos uma instância de Instances com as estruturas de dados.
   Instances instâncias = criaInstances();
   // O índice da classe é o 2 (Atributo Comida).
   instâncias.setClassIndex(2);
   // Classificamos algumas instâncias.
   System.out.println(classifica(árvore,instâncias,
      new Object[]{"Computação","F",200.}));
    System.out.println(classifica(árvore,instâncias,
      new Object[]{"Direito","M",300.}));
  }

  private static String classifica(J48 árvore,Instances instâncias,Object[] args)
    throws Exception {
    // Criamos a instância desconhecida.
    Instance instância = new Instance(4);
    instância.setDataset(instâncias);
    instância.setValue(0,(String)args[0]);
    instância.setValue(1,(String)args[1]);
    instância.setValue(3,(Double)args[2]);
    // Finalmente classificamos esta instância.
    int índice = (int)árvore.classifyInstance(instância);
    return comida.value(índice);
   }
   // Reusamos os métodos criaInstances e criaAtributo da Listagem 5.
}
```

O método main no código da Listagem 7 deserializa a instância de J48 através da classe ObjectInputStream, e cria a estrutura da base de dados através do método criaInstances (na Listagem 6). Também neste método indicamos, para a estrutura da base de dados, qual dos atributos deve ser considerado como classe (no caso, o atributo de índice 2, "Comida").

Após a criação destas estruturas, o método main executa o método classifica, que recebe como argumento a árvore, a estrutura da base de dados e um array de objetos para classificação. Internamente o método classifica cria uma instância de Instance com os valores passados (observem algumas aplicações de autoboxing no array de argumentos), e classifica a instância usando o método classifyInstance da classe J48. O resultado é retornado como uma String obtida do Attribute comida.

O resultado final da aplicação mostra que para a primeira chamada ao método classifica a classe é "Fast-food", e para a segunda chamada, "Japonesa", o que corresponde ao esperado de acordo com o que foi mostrado em seções anteriores.

Outro exemplo: encontrando regras de associação

Outra técnica bastante usada em mineração de dados é a identificação de co-ocorrências ou associações. Esta técnica usa como entradas conjuntos de dados que são agrupados de forma que em um registro de banco de dados (ou linha de planilha) temos listas de elementos que

são associados de alguma forma. Regras de associação indicam que elementos ocorrem freqüentemente com que outros elementos, geralmente na forma "se X e Y ocorrem em um registro então Y também ocorre".

Um exemplo simples e clássico de entrada para este tipo de algoritmo é o chamado carrinho de compras, em que cada registro em um banco de dados a ser minerado contém os itens que foram comprados em uma determinada transação. Outros exemplos de bases de dados que podem ser mineradas para identificação de co-ocorrências são séries temporais de eventos, em que um registro de banco de dados corresponde a um intervalo de tempo e contém um conjunto de eventos que ocorreram naquele intervalo (por exemplo, ocorrências em uma rede).

Regras de associação têm métricas associadas a elas, para que possamos inferir a importância e relevância das regras encontradas. Uma das métricas mais importantes é o suporte, que é um valor percentual usado para medir a relevância das regras encontradas considerando o número de registros da base de dados, e que filtra as regras encontradas de forma que somente regras com ocorrências maiores do que o suporte informado são listadas (eliminando assim regras que correspondem a eventos raros na base de dados). Outra métrica importante é a confiança (também medida em percentuais), que indica, para as regras encontradas, a confiança em que as associações ocorrem.

Para dar um exemplo simples, consideremos que menos que 2% das pessoas em uma base de dados indicaram preferência por comida vegetariana, mas que 80% dos que preferem comida vegetariana estudam filosofia. Se encontrarmos uma regra de associação que indica que pessoas que gostam de comida vegetariana estudam filosofia, esta regra terá suporte 2% e confiança 80%.

Na coluna Professor J do número 22 da Mundoj, mostrei como é possível encontrar regras de associação com um algoritmo bem simples, usando dados de um exemplo fictício de carrinho de compras. Embora o algoritmo demonstrado seja relativamente simples, podemos usar o algoritmo Apriori, parte da API do Weka, para minerar regras de associação mais efetivamente (e facilmente também!)

Uma característica do algoritmo Apriori é que ele só pode processar registros compostos de dados puramente discretos em outras palavras, atributos numéricos devem ser eliminados da mineração ou discretizados (convertidos para atributos discretos). Novamente a API do Weka vem ajudar: existem classes que implementam filtros que permitem a conversão de atributos de um tipo para outro ou remoção de atributos que podem ser efetuadas com poucas linhas de código.

Para exemplificar tanto o uso da classe que implementa o algoritmo Apriori quanto algumas que implementam filtros, consideremos outra maneira de analisar a base de dados de preferências de alimentos dos estudantes: procuraremos co-ocorrências de atributos que indicam quais acontecem com maior freqüência em conjunto. Não podemos usar o atributo Gasto diretamente, pois o mesmo é numérico. Para este primeiro exemplo, optamos por simplesmente ignorar o atributo (através de sua remoção por um filtro) na criação das regras de associação.

Para minerar as regras de associação, primeiro leremos a base de dados de um arquivo (como feito na Listagem 2), filtraremos a base de dados para remover o atributo Gasto e aplicaremos o algoritmo Apriori para encontrar as regras de associação. A Listagem 8 mostra uma aplicação simples que encontra e lista regras de associação.

A aplicação na Listagem 8 lê a base de dados a ser minerada, cria uma instância do filtro Remove, usando os argumentos -R e 4 para indicar que a quarta coluna (Gasto) deve ser removida, cria uma nova base a partir do filtro e cria uma instância da classe Apriori para encontrar as regras de associação (através da chamada ao método buildAssociations). O resultado da execução desta aplicação (impressão da instância de Apriori) é mostrado na Listagem 9.

A Listagem 9 mostra o resultado da mineração de regras de associação. Este resultado pode ser dividido em três partes: a primeira mostra alguns valores de parâmetros (que podem ser mudados com métodos da classe Apriori). No exemplo, os parâmetros default indicam que somente regras com suporte mínimo de 10% (correspondente a três instâncias da base de dados) serão consideradas, e também que somente regras com 90% de confiança serão listadas. O resultado mostra também algumas estatísticas sobre o processamento das regras, incluindo o número de subconjuntos criados durante as fases do processamento (cada fase tenta criar regras envolvendo maior número de atributos, mais informações sobre a criação de subconjuntos podem ser obtidas chamando-se o método setOutputItemSets(true) para a instância de Apriori). A terceira parte do resultado é a mais interessante: mostra que regras de associação foram encontradas e algumas estatísticas.

As regras são divididas em antecedentes e conseqüentes, e na representação os antecedentes aparecem à esquerda do símbolo "==>" e os conseqüentes, à direita. A primeira regra pode ser lida como "se o curso é Computação e a comida preferida é Fast-food então o Sexo é masculino". A regra mostra também qual é o número de ocorrências de antecedentes e conseqüentes: na primeira regra, nas cinco ocorrências em que o curso era Computação e a comida era Fast-food também o Sexo era M (confiança de 100% ou 1). Como o suporte mínimo era 10% (ou três instâncias) outras regras não foram encontradas. Outro parâmetro default indica que no máximo 10 regras devem ser listadas, mas somente três foram encontradas.

Para exemplificar melhor na prática a influência dos parâmetros de suporte e confiança, podemos modificar a Listagem 8 para incluir as três linhas mostradas na Listagem 10. As linhas devem ser inseridas logo após a chamada ao construtor da classe Apriori e antes da chamada ao método buildAssociations.

As regras obtidas com a modificação dos parâmetros de confiança e suporte são mostradas na Listagem 11.

Podemos observar que as regras mostradas na Listagem 11 são diferentes das mostradas na Listagem 9 não somente em relação ao número de regras encontrado: muitas outras regras com confiança menor do que 1 foram encontradas, e outras, cujo suporte era pequeno, não foram incluídas entre as 15 mais significativas (as regras 2 e 3 da Listagem 9 apareceriam na Listagem 11 caso solicitássemos a Listagem das 25 regras mais significativas).

Regras de associação podem ser usadas para identificar subgrupos significativos de forma livre, isso é, podendo conter qualquer número de atributos como antecedentes e conseqüentes. Como é possível que muitas regras sejam geradas até mesmo para bases de dados simples, uma tarefa adicional de mineração de dados envolvendo regras de associação é a mineração das próprias regras, o que deve ser feito considerando-se a tarefa em questão.

Por exemplo, usando como base os resultados mostrados na Listagem 11 e o problema "como fazer um marketing dirigido para pessoas que têm preferência por comida japonesa , um analista poderia considerar somente as regras 6 a 10 (algumas destas regras são variantes de outras) e concluir que marketing dirigido a estudantes de direito do sexo feminino pode ser mais eficiente.

Duas observações importantes devem ser feitas sobre mineração de regras de associação, uma sobre características da base de dados e outra sobre a formatação desta base de dados para o Weka. Dependendo da complexidade e dimensão da base de dados é possível que regras realmente significativas tenham suporte muito baixo. Para identificar estas regras é preciso experimentar com o parâmetro que indica o suporte mínimo para corte e ao mesmo tempo com o parâmetro que indica quantas regras devem ser listadas. Usar um suporte mínimo muito alto pode fazer com que o algoritmo não encontre nenhuma regra significativa, enquanto um suporte mínimo muito baixo pode fazer com que muitas regras sejam geradas, dificultando a sua interpretação.

Dependendo da tarefa, a formatação da base de dados para o Weka deve variar. Consideremos um estudo de carrinho de compras, em que queremos saber que item foi comprado em conjunto com outro. A estrutura de base de dados mostrada na Listagem 1 é bastante estruturada e regular, ou seja, cada linha da tabela contém os mesmos campos na mesma ordem em que foram declarados no cabeçalho. Uma relação de carrinho de compras tem uma estrutura menos formal, ou seja, cada linha corresponde a uma transação (compra) e deve conter a lista dos itens comprados naquela transação.

Para transformar uma base de dados menos estruturada para uma aceitável para processamento pelo Weka, devemos listar todos os elementos passíveis de aparecer na lista e considerar que cada dado na base de dados será uma lista de verificação. Em outras palavras, se no total tivermos 15 tipos de itens que podem ser associados devemos declarar 15 atributos na nossa base de dados, e cada dado da base teria 15 variáveis indicando a existência ou não daquele atributo naquela transação.

Para exemplificar, vamos aproveitar a tabela de carrinhos de compra usada na coluna Professor J do número 22 da Mundoj. A tabela é mostrada na figura 3.

Listagem 8. Aplicação que cria e lista regras de associação.

```
// imports necessários

public class CriaRegrasDeAssociacao {
   public static void main(String[] args) throws Exception {
      // Lemos os dados.
      DataSource fonte = new DataSource("dados.arff );
      Instances dados = fonte.getDataSet();
      // Não queremos usar o atributo numérico Gasto!
      Remove filtro = new Remove();
      filtro.setOptions(new String[]{"-R , 4 });
      filtro.setInputFormat(dados);
      // Criamos um novo conjunto de instâncias a partir do filtro.
      Instances instâncias = Filter.useFilter(dados,filtro);
      // Criamos as regras de associação.
      Apriori assoc = new Apriori();
      assoc.buildAssociations(instâncias);
      System.out.println(assoc);
   }
}
```

Listagem 9. Resultado da execução da aplicação mostrada na Listagem 8.

```
Apriori
=======

Minimum support: 0.1 (3 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 18

Generated sets of large itemsets:

Size of set of large itemsets L(1): 9

Size of set of large itemsets L(2): 17

Size of set of large itemsets L(3): 4

Best rules found:

 1. Curso=Computação Comida=Fast-food 5 ==> Sexo=M 5    conf:(1)
 2. Curso=Computação Comida=Italiana 3 ==> Sexo=F 3    conf:(1)
 3. Curso=Economia Comida=Italiana 3 ==> Sexo=F 3    conf:(1)
```

Listagem 10. Modificando o suporte e confiança para busca de mais regras de associação.

```
assoc.setLowerBoundMinSupport(0.05); // 5% de suporte mínimo
assoc.setMinMetric(0.5); // 50% de confiança mínima
assoc.setNumRules(15); // Consideraremos as 15 regras mais importantes.
```

Listagem 11. Regras de associação obtidas com a modificação do suporte e confiança.

```
 1. Curso=Computação Comida=Fast-food 5 ==> Sexo=M 5    conf:(1)
 2. Comida=Italiana 7 ==> Sexo=F 6    conf:(0.86)
 3. Curso=Computação Sexo=M 6 ==> Comida=Fast-food 5    conf:(0.83)
 4. Comida=Fast-food 10 ==> Sexo=M 8    conf:(0.8)
 5. Comida=Mineira 5 ==> Sexo=M 4    conf:(0.8)
 6. Sexo=F Comida=Japonesa 5 ==> Curso=Direito 4    conf:(0.8)
 7. Curso=Direito Comida=Japonesa 5 ==> Sexo=F 4    conf:(0.8)
 8. Curso=Direito Sexo=F 5 ==> Comida=Japonesa 4    conf:(0.8)
 9. Comida=Japonesa 8 ==> Curso=Direito 5    conf:(0.63)
10. Comida=Japonesa 8 ==> Sexo=F 5    conf:(0.63)
11. Sexo=M Comida=Fast-food 8 ==> Curso=Computação 5    conf:(0.63)
12. Curso=Computação 10 ==> Sexo=M 6    conf:(0.6)
13. Sexo=M 16 ==> Comida=Fast-food 8    conf:(0.5)
14. Comida=Fast-food 10 ==> Curso=Computação 5    conf:(0.5)
15. Curso=Computação 10 ==> Comida=Fast-food 5    conf:(0.5)
```

Carrinho	Conteúdo
1	leite, ovos, café, açúcar, pão e manteiga
2	café, leite, farinha e chá
3	café e açúcar
4	café, pão, açúcar, leite e margarina
5	manteiga, ovos, pão, café e leite
6	café, açúcar e leite
7	café e leite
8	leite e açúcar

Figura 3. Conteúdo (simulado) de alguns carrinhos de compras.

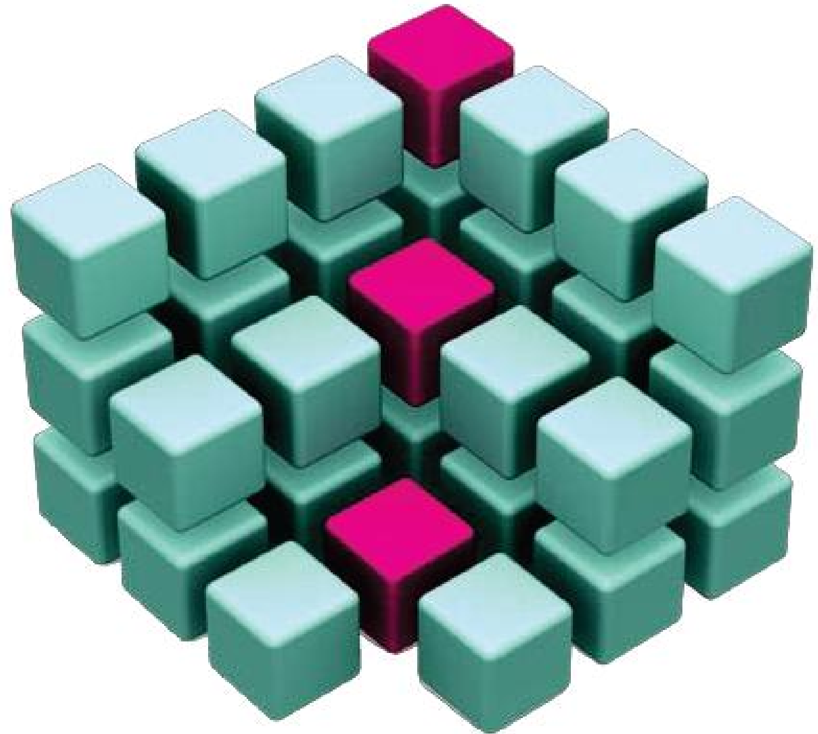
Como existem nove diferentes itens nos carrinhos nossa base deverá ter nove atributos. Cada uma das oito linhas de dados será uma lista de valores S ou N para indicar a existência ou não do item correspondente naquele carrinho. Um arquivo .arff gerado usando os dados da figura 3 e que segue esta regra é mostrado na Listagem 12.

Listagem 12. Arquivo .arff que representa um carrinho de compras de forma relacional.

```
@RELATION compras

@ATTRIBUTE Leite {S,N}
@ATTRIBUTE Ovos {S,N}
@ATTRIBUTE Café {S,N}
@ATTRIBUTE Açúcar {S,N}
@ATTRIBUTE Pão {S,N}
@ATTRIBUTE Manteiga {S,N}
@ATTRIBUTE Farinha {S,N}
@ATTRIBUTE Chá {S,N}
@ATTRIBUTE Margarina {S,N}

@data
S,S,S,S,S,S,N,N,N
S,N,S,N,N,N,S,S,N
N,N,S,S,N,N,N,N,N
S,N,S,S,S,N,N,N,S
S,S,S,N,S,S,N,N,N
S,N,S,S,N,N,N,N,N
S,N,S,N,N,N,N,N,N
S,N,N,S,N,N,N,N,N
```



Embora a representação relacional de um carrinho de compras seja simples, a sua criação pode ser consideravelmente complexa quando o número de itens for muito grande por exemplo, até mesmo para um mercado muito simples, com 50 itens únicos, teríamos que representar cada transação como uma lista de 50 valores S ou N. Seguramente será necessário usar métodos automáticos para conversão de dados na forma livre (listas de itens) para uma forma mais estruturada e passível de processamento pelo algoritmo Apriori.

Outro problema potencial relacionado com a representação de dados de carrinhos de compras é que os valores S e N usados para representar a presença ou ausência de um determinado item em uma transação são, para o algoritmo Apriori, símbolos sem significado semântico. Em outras palavras, o algoritmo Apriori não fará distinção entre dados com atributos S ou N, e procurará associações independentemente da presença ou ausência dos itens, possivelmente encontrando várias regras de associação negativas. Um exemplo deste tipo de regra, usando a base de dados mostrada na figura 12 como exemplo, seria "Onde Chá = N, Farinha = N com confiança 1, ocorrendo sete vezes na base de dados , o que é verdadeiro e pode ser confirmado com uma simples análise da tabela na figura 3 – sete linhas da base correspondem a transações nas quais nem chá nem farinha foram comprados.

Embora em alguns casos a busca de regras de associação negativas possa ser interessante, na maioria dos casos estas regras corresponderão a maior parte das regras de associação, podendo até mesmo ocultar regras positivas que tenham confiança menor. Se isso não for desejável, a solução é substituir os símbolos N por sinais de interrogação, que são, para o Weka, indicações de que os dados não são conhecidos. A substituição deve ser feita somente na seção @data do arquivo .arff, as declarações de atributos devem ser mantidas.

Considerações finais

Neste artigo, vimos conceitos básicos sobre mineração de dados e implementação de aplicações que usam a API do Weka para minerar dados. Os exemplos foram focados em conjuntos de dados e tarefas simples para facilitar a compreensão, mas podem facilmente ser estendidos e modificados para outras aplicações.

Nos exemplos, dois algoritmos clássicos e bastante usados: um de árvores de decisão e outro de busca de regras de associação. Muitos outros algoritmos, incluindo algoritmos de agrupamento, redes neurais, de regressão, etc. podem igualmente ser usados nos mais diversos tipos de aplicação. MJ

Referências

A coluna Professor J do número 22 da Mundoj demonstra como um algoritmo simples de busca de associações pode ser implementado usando coleções em Java.

O software Weka é associado ao livro "Data Mining: Practical machine learning tools and techniques , de Ian H. Witten e Eibe Frank, editora Morgan Kaufmann, San Francisco, segunda edição (2005). Este livro descreve bem conceitos teóricos que fundamentam os algoritmos, mas pode ser usado independentemente do software e vice-versa.

A página principal do Weka (com links para download) é http://www.cs.waikato.ac.nz/ml/weka/. Para melhor funcionalidade, sugiro o uso da versão 3.5 ou superior.

Outros exemplos de aplicação de mineração de dados podem ser vistos no site do autor em www.lac.inpe.br/~rafael.santos, que também contém o material usado em disciplinas relacionadas ministradas no Instituto Nacional de Pesquisas Espaciais (incluindo uma descrição mais completa e precisa do algoritmo C4.5/J4.8 que cria árvores de decisão). Dentre os documentos, um que pode ser de interesse para desenvolvedores é o http://www.lac.inpe.br/~rafael.santos/CAP/cap359/2005/weka.pdf ("Weka na Munheca"), que contém algumas informações complementares a este documento.