Técnicas de Mineração de Dados

Sandra de Amo

Universidade Federal de Uberlândia
Faculdade de Computação
deamo@ufu.br

Abstract

Data Mining is a multidisciplinary research area, including database tecnology, artificial intelligence, machine learning, neural networks, statistics, pattern recognition, knowledge-based systems, information retrieval, high-performance computing and data visualization. This course is based on a database perspective and is focused on techniques for a variety of data mining tasks. We present algorithms, optimization techniques, important application domains and also some theoretical aspects which may help to develop algorithms for new data mining tasks.

Resumo

Mineração de Dados é uma área de pesquisa multidisciplinar, incluindo tecnologia de bancos de dados, inteligência artificial, aprendizado de máquina, redes neurais, estatística, reconhecimento de padrões, sistemas baseados em conhecimento, recuperação da informação, computação de alto desempenho e visualização de dados. Este curso é baseado em um perspectiva de banco de dados e é focalizado sobretudo sobre as técnicas para realizar uma grande variedade de tarefas de mineração. Apresentamos algoritmos, técnicas de otimização, importantes domínios de aplicação e também alguns aspectos teóricos da área que podem ajudar no desenvolvimento de algoritmos para novas tarefas de mineração.

5.1 Introdução

Mineração de Dados é um ramo da computação que teve início nos anos 80, quando os profissionais das empresas e organizações começaram a se preocupar com os grandes volumes de dados informáticos estocados e inutilizados dentro da empresa. Nesta época, Data Mining consistia essencialmente em extrair informação de gigantescas bases de dados da maneira mais automatizada possível. Atualmente, Data Mining consiste sobretudo na análise dos dados após a extração, buscando-se por exemplo levantar as necessidades reais e hipotéticas de cada cliente para realizar campanhas de marketing. Assim, uma empresa de cartões de crédito, por exemplo, tem uma mina de ouro de informações: ela sabe os hábitos de compra de cada um dos seus seis milhões de clientes. O que costuma consumir, qual o seu padrão de gastos, grau de endividamento, etc. Para a empresa essas informações são extremamente úteis no estabelecimento do limite de crédito para cada cliente, e além disso, contém dados comportamentais de compra de altíssimo valor. Os seguintes pontos são algumas das razões por que o Data Mining vem se tornando necessário para uma boa gestão empresarial: (a) os volumes de dados são muito importantes para um tratamento utilizando somente técnicas clássicas de análise, (b) o usuário final não é necessariamente um estatístico, (c) a intensificação do tráfico de dados (navegação na Internet, catálogos online, etc) aumenta a possibilidade de acesso aos dados.

Este minicurso tem como objetivo fornecer um apanhado geral das principais tarefas e técnicas de mineração de dados. Discutiremos algumas técnicas de otimização e implementação de algoritmos de mineração de dados referentes a tarefas de regras de associação, descoberta de padrões sequenciais, classificação e análise de clusters. Além disto, discutiremos aspectos teóricos subjacentes a diferentes técnicas de mineração, que possibilitarão o desenvolvimento de algoritmos de mineração para novas tarefas.

5.1.1 O que é Mineração de Dados

Afinal, o que é Mineração de Dados ? Falando simplesmente, trata-se de *extrair* ou *minerar* conhecimento de grandes volumes de dados. Muitas pessoas consideram o termo *Mineração de Dados* como sinônimo de *Knowledge Discovery in Databases (KDD)* ou *Descoberta de Conhecimento em Banco de Dados*. Na verdade, KDD é um processo mais amplo consistindo das seguintes etapas, como ilustrado na Figura 5.1:

1. *Limpeza dos dados*: etapa onde são eliminados ruídos e dados inconsistentes.

2. *Integração dos dados*: etapa onde diferentes fontes de dados podem ser combinadas produzindo um único repositório de dados.

3. *Seleção*: etapa onde são selecionados os atributos que interessam ao usuário. Por exemplo, o usuário pode decidir que informações como endereço e telefone não são de relevantes para decidir se um cliente é um bom comprador ou não.

4. *Transformação dos dados*: etapa onde os dados são transformados num formato apropriado para aplicação de algoritmos de mineração (por exemplo, através de operações de agregação).

5. ***Mineração***: etapa essencial do processo consistindo na aplicação de técnicas inteligentes a fim de se extrair os padrões de interesse.

6. *Avaliação ou Pós-processamento*: etapa onde são identificados os padrões interessantes de acordo com algum critério do usuário.

7. *Visualização dos Resultados*: etapa onde são utilizadas técnicas de representação de conhecimento a fim de apresentar ao usuário o conhecimento minerado.

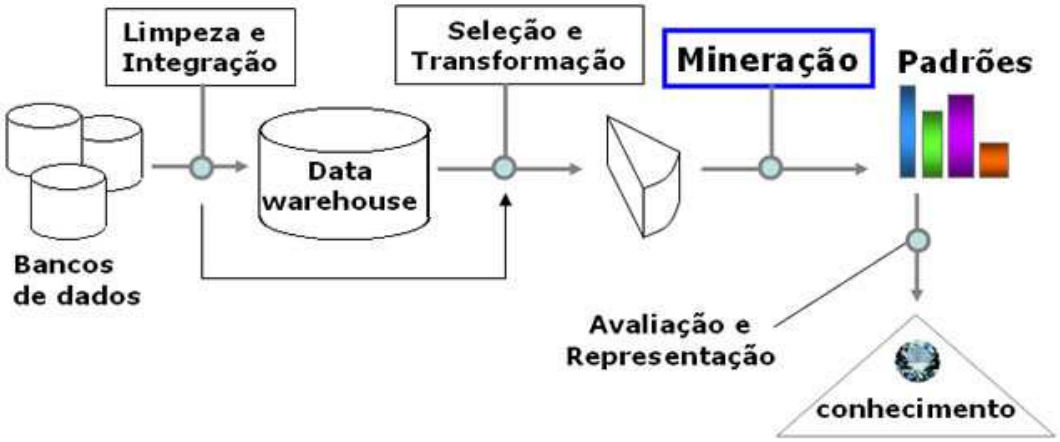


Figura 5.1: As etapas do processo de KDD

Este curso está focado sobretudo nas técnicas frequentemente utilizadas na etapa *Mineração* do processo de KDD. Supomos que os dados já foram devidamente selecionados e transformados, integrados num armazém de dados (*Data Warehouse*) e deles foram eliminados ruídos que possam afetar o processo de descoberta de conhecimento. A fase de visualização do conhecimento descoberto também não é tratada neste curso.

5.1.2 Tarefas e Técnicas de Mineração de Dados

É importante distinguir o que é uma *tarefa* e o que é uma *técnica* de mineração. A tarefa consiste na especificação *do que* estamos querendo buscar nos dados, que tipo de regularidades ou categoria de padrões temos interesse em encontrar, ou que tipo de padrões poderiam nos surpreender (por exemplo, um gasto exagerado de um cliente de cartão de crédito, fora dos padrões usuais de seus gastos). A *técnica* de mineração consiste na especificação de métodos que nos garantam *como* descobrir os padrões que nos interessam. Dentre as principais técnicas utilizadas em mineração de dados, temos técnicas estatísticas, técnicas de aprendizado de máquina e técnicas baseadas em *crescimento-poda-validação*. A seguir, descrevemos de forma sucinta as principais tarefas de mineração.

Análise de Regras de Associação. Uma *regra de associação* é um padrão da forma $X \rightarrow Y$, onde X e Y são conjuntos de valores (artigos comprados por um cliente, sintomas apresentados por um paciente, etc). Consideremos por exemplo um supermercado. O seguinte padrão "Clientes que compram pão também compram leite" representa uma regra de associação que reflete um padrão de comportamento dos clientes do supermercado. Descobrir regras de associação entre produtos comprados por clientes numa mesma compra pode ser útil para melhorar a organização das prateleiras, facilitar (ou dificultar) as compras do usuário ou induzi-lo a comprar mais.

Análise de Padrões Sequenciais. Um *padrão sequencial* é uma expressão da forma $< I_1, \ldots, I_n >$, onde cada I_i é um conjunto de items. A ordem em que estão alinhados

estes conjuntos reflete a ordem cronológica em que aconteceram os fatos representados por estes conjuntos. Assim, por exemplo, a sequência $<$ {carro}, {pneu, toca-fitas} $>$ representa o padrão "Clientes que compram carro, tempos depois compram pneu e toca-fitas de carro". Descobrir tais padrões sequenciais em dados temporais pode ser útil em campanhas de marketing, por exemplo.

Classificação e Predição. *Classificação* é o processo de encontrar um conjunto de modelos (funções) que descrevem e distinguem classes ou conceitos, com o propósito de utilizar o modelo para predizer a classe de objetos que ainda não foram classificados. O modelo construído baseia-se na análise prévia de um conjunto de dados de amostragem ou dados de treinamento, contendo objetos corretamente classificados. Por exemplo, suponha que o gerente do supermercado está interessado em descobrir que tipo de características de seus clientes os classificam em "bom comprador" ou "mau comprador". Um modelo de classificação poderia incluir a seguinte regra: "Clientes da faixa econômica B, com idade entre 50 e 60 são maus compradores". Em algumas aplicações, o usuário está mais interessado em predizer alguns valores ausentes em seus dados, em vez de descobrir classes de objetos. Isto ocorre sobretudo quando os valores que faltam são numéricos. Neste caso, a tarefa de mineração é denominada *Predição*. Veremos neste curso algumas técnicas usualmente empregadas em tarefas de classificação, tais como árvores de decisão e redes neurais. Boa parte dos métodos de classificação utilizam técnicas estatísticas e de aprendizado de máquina.

Análise de *Clusters* (Agrupamentos). Diferentemente da classificação e predição onde os dados de treinamento estão devidamente classificados e as etiquetas das classes são conhecidas, a análise de clusters trabalha sobre dados onde as etiquetas das classes não estão definidas. A tarefa consiste em identificar agrupamentos de objetos, agrupamentos estes que identificam uma classe. Por exemplo, poderíamos aplicar análise de clusters sobre o banco de dados de um supermercado a fim de identificar grupos homogêneos de clientes, por exemplo, clientes aglutinados em determinados pontos da cidade costumam vir ao supermercado aos domingos, enquanto clientes aglutinados em outros pontos da cidade costumam fazer suas compras às segundas-feira.

Análise de *Outliers*. Um banco de dados pode conter dados que não apresentam o comportamento geral da maioria. Estes dados são denominados *outliers*(exceções). Muitos métodos de mineração descartam estes *outliers* como sendo ruído indesejado. Entretanto, em algumas aplicações, tais como detecção de fraudes, estes eventos raros podem ser mais interessantes do que eventos que ocorrem regularmente. Por exemplo, podemos detectar o uso fraudulento de cartões de crédito ao descobrir que certos clientes efetuaram compras de valor extremamente alto, fora de seu padrão habitual de gastos.

5.1.3 Como Avaliar os Padrões Interessantes ?

Existem diversas medidas objetivas para avaliar o grau de interesse que um padrão pode apresentar ao usuário. Tais medidas são baseadas na estrutura do padrão descoberto e em estatísticas apropriadas. Por exemplo, uma medida objetiva para avaliar o interesse de uma regra de associação é o *suporte*, representando a porcentagem de transações de um banco de dados de transações onde a regra se verifica. Em termos estatísticos, o suporte de uma regra $X \rightarrow Y$ é a probabilidade $P(X \cup Y)$, onde $X \cup Y$ indica que a transação contém os dois conjuntos de items X e Y. Uma outra medida objetiva para regras de associação é a *confiança*, que mede o grau de certeza de uma associação. Em termos estatísticos, trata-se simplesmente da probabilidade condicional $P(Y \mid X)$, isto

é, a porcentagem de transações contendo os items de X que também contém os items de Y. Em geral, cada medida objetiva está associada a um limite mínimo de aceitação, que pode ser controlado pelo usuário. Por exemplo, o usuário pode decidir que regras cuja confiança é inferior a 0.5 devem ser descartadas como não-interessantes, pois podem simplesmente representar uma minoria ou exceção ou envolver ruídos.

Além das medidas objetivas, o usuário pode especificar medidas subjetivas para guiar o processo de descoberta, refletindo suas necessidades particulares. Por exemplo, padrões descrevendo as características dos clientes habituais de uma loja pode ser de interesse para o gerente de marketing da loja, mas com certeza é de pouco interesse para analistas que estão interessados em padrões de comportamento dos empregados da loja. Além disto, padrões que são interessantes segundo medidas objetivas podem representar conhecimento óbvio e portanto sem interesse. Pode-se por exemplo medir o grau de interesse de um padrão pelo fato de ele ser inesperado pelo usuário. Ou, ao contrário, pode-se dizer que um padrão é interessante se ele se adequa às expectativas do usuário, servindo para confirmar uma hipótese que o usuário deseja validar.

Medidas (objetivas ou subjetivas) de avaliação do grau interesse de padrões são essenciais para a eficiência do processo de descoberta de padrões. Tais medidas podem ser usadas durante o processo de mineração ou após o processo a fim de classificar os padrões encontrados de acordo com seu interesse para um dado usuário, filtrando e eliminando os não interessantes. Em termos de eficiência é importante incorporar medidas de interesse que restrinjam o espaço de busca dos padrões *durante* o processo de descoberta, ao invés de *após* o processo ter terminado.

5.1.4 Sistemas Comerciais de Mineração de Dados

O quadro abaixo contém dados sobre alguns sistemas mineradores importantes, juntamente com suas principais funcionalidades:

Nome	**Fabricante**	**Funções**	**Destaque**
Intelligent Miner	IBM	algoritmos para regras de associação, classificação, regressão, padrões sequenciais, clustering.	Integrado com o SGBD DB2 da IBM. Grande escalabilidade dos algoritmos.
MineSet	Silicon Graphics Inc.	algoritmos para regras de associação, classificação, análise estatística.	Um robusto conjunto de ferramentas avançadas de visualização.
Clementine	Integral Solutions Ltd.	algoritmos de regras de indução, redes neurais, classificação e ferramentas de visualização.	Interface orientada-objeto.
DBMiner	DBMiner Technology Inc.	algoritmos de regras de associação, classificação, clustering.	Data Mining utilizando OLAP
Genamics Expression	Genamics Developer	algoritmos de análise de sequências	Análise de proteínas e de sequências de DNA

Notas Bibliográficas. Existe uma grande variedade de livros texto sobre Mineração de Dados, desde os focados no *uso* da mineração para tomada de decisões (destinados a leitores interessados em usar softwares de mineração) até os focados principalmente nas *técnicas* de mineração (destinados a leitores interessados em construir softwares de

mineração). Dentro desta segunda linha, o leitor interessado poderá encontrar amplo material de estudo em [HK 2001, HMS 2001]. Em [WF 2000], o leitor interessado em técnicas de implementação de algoritmos de mineração encontra um bom material focado sobre a Linguagem Java. Em [FPPR 1996], o leitor de nível avançado tem um texto que reúne artigos científicos importantes na área de mineração de dados, abrangendo os principais tópicos de interesse na área. Em [Man 1997], o leitor mais apressado encontra um texto agradável e acessível que dá uma idéia geral dos métodos, técnicas e tarefas de mineração de dados. Em [Amo 2003], o leitor tem acesso a um curso de Data Mining ministrado pela autora do presente minicurso, com farto material didático e referências bibliográficas.

5.2 Técnicas para Regras de Associação e Sequências

5.2.1 Técnica *Apriori*

Suponha que você seja gerente de um supermercado e esteja interessado em conhecer os hábitos de compra de seus clientes, por exemplo, "quais os produtos que os clientes costumam comprar ao mesmo tempo, a cada vez que vêm ao supermercado". Conhecer a resposta a esta questão pode ser útil: você poderá planejar melhor os catálogos do supermercado, os folhetos de promoções de produtos, as campanhas de publicidade, além de organizar melhor a localização dos produtos nas prateleiras do supermercado colocando próximos os items frequentemente comprados juntos a fim de encorajar os clientes a comprar tais produtos conjuntamente. Para isto, você dispõe de uma *mina* de dados, que é o *banco de dados de transações* efetuadas pelos clientes (Figura 5.3). A cada compra de um cliente, são registrados neste banco todos os itens comprados. Para facilitar a representação dos artigos na tabela, vamos associar números a cada artigo do supermercado, como ilustrado na Figura 5.2.

Artigo (item)	número que o representa
Pão	1
Leite	2
Açúcar	3
Papel Higiênico	4
Manteiga	5
Fralda	6
Cerveja	7
Refrigerante	8
Iogurte	9
Suco	10

Figura 5.2: Representação numérica de cada artigo do supermercado

Cada conjunto de itens comprados pelo cliente numa única transação é chamado de *Itemset*. Um itemset com k elementos é chamado de k-itemset. Suponha que você, como gerente, decide que um itemset que aparece em *pelo menos* 50% de todas as compras registradas será considerado *frequente*. Por exemplo, se o banco de dados de que você dispõe é o ilustrado na Figura 5.3, então o itemset {1,3} é considerado frequente, pois aparece em mais de 60% das transações. Definimos *suporte* de um itemset como sendo a porcentagem de transações onde este itemset aparece. A tabela da Figura 5.4 contabiliza

TID	Itens comprados
101	{1,3,5}
102	{2,1,3,7,5}
103	{4,9,2,1}
104	{5,2,1,3,9}
105	{1,8,6,4,3,5}
106	{9,2,8}

Figura 5.3: Um banco de dados de transações de clientes

os suportes de diversos itemsets com relação ao banco de dados de transações da Figura 5.3.

Itemset	Suporte
{1,3}	0,6666
{2,3}	0,3333
{1,2,7}	0,16666
{2,9}	0,5

Figura 5.4: Suporte de alguns itemsets

Repare que o que identifica uma transação é o *identificador da transação* TID e não o *identificador do cliente*.

Caso a sua exigência mínima para um itemset ser considerado frequente seja 50%, então os seguintes itemsets da tabela da Figura 5.4 serão considerados frequentes: {1,3}, {2,9}.

Formalização do Problema

Seja $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ um conjunto de itens (o conjunto de todos os artigos do supermercado). Seja $\mathcal{D}$ um banco de dados de transações, isto é, uma tabela de duas colunas, a primeira correspondente ao atributo TID (identificador da transação) e o segundo correspondente à transação propriamente dita, isto é, a um conjunto de itens (itemset). Os elementos de $\mathcal{D}$ são chamados de *transações*. Um *itemset* é um subconjunto não vazio de $\mathcal{I}$. Dizemos que uma transação T *suporta* um itemset I se I $\subseteq$ T. Por exemplo, a primeira transação do banco de dados da Figura 5.3 suporta os itemsets {1}, {3}, {5}, {1,3}, {1,5}, {3,5}, {1,3,5}.

Repare que, embora uma transação e um itemset sejam a mesma coisa (conjunto de items), chamamos de *transação* somente aqueles itemsets que estão registrados no banco de dados como sendo a compra *total* feita por algum cliente.

Definição 5.2.1 Uma *regra de associação* é uma expressão da forma $A \rightarrow B$, onde A e B são itemsets. Por exemplo, {pão, leite} $\rightarrow$ {café} é uma regra de associação. A idéia por trás desta regra é que pessoas que compram pão e leite *têm a tendência de* também comprar café, isto é, **se** alguém compra pão e leite **então** também compra café. Repare que esta regra é diferente da regra {café} $\rightarrow$ {pão,leite}.

A toda regra de associação $A \rightarrow B$ associamos um *grau de confiança*, denotado por $conf(A \rightarrow B)$. Este grau de confiança é simplesmente a porcentagem das transações que suportam B dentre todas as transações que suportam A, isto é:

$$conf(A \rightarrow B) = \frac{\text{número de transações que suportam } (A \cup B)}{\text{número de transações que suportam } A}$$

Por exemplo, o grau de confiança da regra {cerveja} $\rightarrow$ {manteiga}, isto é, {7} $\rightarrow$ {5}, com relação ao banco de dados da Figura 5.3 é 1 (100%).

Será que o fato de uma certa regra de associação ter um grau de confiança relativamente alto é suficiente para a considerarmos uma "boa" regra ? Repare que no nosso banco de dados da Figura 5.3, os itens *cerveja, manteiga* aparecem juntos somente em uma transação entre 6, isto é, poucos clientes compram estes dois itens juntos. Entretanto, somente pelo fato de que em 100% das transações onde cerveja aparece também o item manteiga foi comprado, temos que o grau de confiança da regra {cerveja} $\rightarrow$ {manteiga} é de 100%. Nossa intuição diz que isto não é suficiente para que esta regra seja considerada "boa", já que esta confiança diz respeito somente às poucas transações que suportam {cerveja, manteiga}. A fim de garantirmos que uma regra $A \rightarrow B$ seja *boa* ou *interessante*, precisamos exigir que seu *suporte* também seja relativamente alto, além de seu grau de confiança.

A toda regra de associação $A \rightarrow B$ associamos um *suporte*, denotado por $sup(A \rightarrow B)$ definido como sendo o suporte do itemset $A \cup B$. Por exemplo, o suporte da regra {cerveja} $\rightarrow$ {manteiga} com relação ao banco de dados da Figura 5.3 é 0.1666%.

Uma regra de associação r é dita *interessante* se $conf(r) \geq \alpha$ e $sup(r) \geq \beta$, onde α e β são respectivamente um grau mínimo de confiança e um grau mínimo de suporte especificados pelo usuário. No nosso exemplo, caso $\alpha = 0.8$ e $\beta = 0.1$ então a regra {cerveja} $\rightarrow$ {manteiga} é interessante.

O problema da mineração de regras de associação é o seguinte: Dados um banco de dados de transações $\mathcal{D}$, um nível mínimo de confiança α e um um nível mínimo de suporte β, encontrar todas as regras de associação interessantes com relação a $\mathcal{D}$, α e β.

O algoritmo Apriori

Este algoritmo foi proposto em 1994 pela equipe de pesquisa do Projeto QUEST da IBM que originou o software *Intelligent Miner*. Trata-se de um algoritmo que resolve o *problema da mineração de itemsets frequentes*, isto é, dados um banco de dados de transações $\mathcal{D}$ e um um nível mínimo de suporte β, o algoritmo encontra todos os itemsets frequentes com relação a $\mathcal{D}$ e β.

Suponha que tenhamos obtido todos os itemsets frequentes com relação a $\mathcal{D}$ e β. A fim de obter as regras de associação interessantes, basta considerarmos, para cada itemset frequente L, todas as regras *candidatas* $A \rightarrow (L - A)$, onde $A \subset L$ e testarmos para cada uma destas regras candidatas se o seu grau de confiança excede o nível mínimo de confiança α. Para calcular a confiança de $A \rightarrow (L - A)$ não é preciso varrer novamente o banco de dados $\mathcal{D}$. De fato, durante a execução do algoritmo Apriori já calculamos o suporte de L e A. Note que:

$$conf(A \rightarrow (L-A)) = \frac{\text{total de trans. suportando L}}{\text{total de trans. suportando A}} = \frac{\frac{\text{total de trans. suportando L}}{\text{total de trans}}}{\frac{\text{total de trans. suportando A}}{\text{total de trans}}} = \frac{sup(L)}{sup(A)}$$

Assim, para calcular a confiança de $A \rightarrow (L - A)$ basta dividir o suporte de L pelo suporte de A.

As fases de Apriori: geração, poda, validação O algoritmo Apriori possui três fases principais: **(1)** a fase da geração dos candidatos, **(2)** a fase da poda dos candidatos e **(3)** a fase do cálculo do suporte. As **duas primeiras fases** são realizadas na memória principal e não necessitam que o banco de dados $\mathcal{D}$ seja varrido. A memória secundária

só é utilizada caso o conjunto de itemsets candidatos seja muito grande e não caiba na memória principal. Mas, mesmo neste caso é bom salientar que o banco de dados $\mathcal{D}$, que normalmente nas aplicações é extremamente grande, não é utilizado. Somente na terceira fase, a fase do cálculo do suporte dos itemsets candidatos, é que o banco de dados $\mathcal{D}$ é utilizado. Tanto na fase de geração de candidatos (Fase 1) quanto na fase da poda dos candidatos (Fase 2) a seguinte propriedade de **antimonotonia** é utilizada:
Propriedade Apriori - ou Antimonotonia da relação $\subseteq$: Sejam I e J dois itemsets tais que $I \subseteq J$. Se J é frequente então I também é frequente.

O algoritmo Apriori é executado de forma iterativa: os itemsets frequentes de tamanho k são calculados a partir dos itemsets frequentes de tamanho $k-1$ que já foram calculados no passo anterior (a partir dos itemsets frequentes de tamanho $k-2$, etc). No que se segue, suponhamos que estejamos no passo k e que portanto já tenhamos obtido no passo anterior o conjunto L_{k-1} dos **itemsets frequentes** de tamanho $k-1$.

A fase da geração dos candidatos de tamanho k**.** Nesta fase, vamos gerar os itemsets **candidatos** (não necessariamente frequentes) de tamanho k a partir do conjunto L_{k-1}. Como estamos interessados em gerar somente itemsets que tenham alguma chance de serem frequentes, devido à propriedade Apriori sabemos que todos os itemsetsde tamanho $k-1$ contidos nos nossos candidatos de tamanho k deverão ser frequentes, portanto, deverão pertencer ao conjunto L_{k-1}. Assim, o conjunto C'_k de itemsets candidatos de tamanho k é construído *juntando-se* pares de itemsets de tamanho $k-1$ que tenham $k-2$ elementos em comum. Desta maneira temos certeza de obter um itemset de tamanho k onde *pelo menos dois* de seus subconjuntos de tamanho $k-1$ são frequentes. A Figura 5.5 ilustra esta construção.

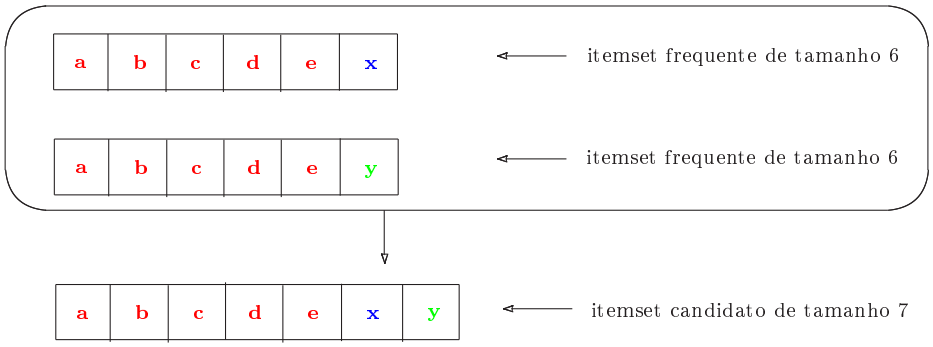


Figura 5.5: Construção de um k-itemset candidato a partir de dois frequentes de tamanho $k-1$

A função Apriori-Gen descrita abaixo é responsável pela construção do conjunto dos *pré-candidatos* C'_k:

insert into C'_k
select $p.\text{item}_1$, $p.\text{item}_2$, ..., $p.\text{item}_{k-2}$,$p.\text{item}_{k-1}$, $q.\text{item}_{k-1}$
from L_{k-1} p, L_{k-1} q
where $p.\text{item}_1 = q.\text{item}_1$, $p.\text{item}_2 = q.\text{item}_2$, ...,$p.\text{item}_{k-2} = q.\text{item}_{k-2}$,
$p.\text{item}_{k-1} < q.\text{item}_{k-1}$;

Exemplo 5.2.1 Consideremos o banco de dados de transações dado na Figura 5.3 e suponhamos que no passo 2 da iteração tenhamos obtido o seguinte conjunto de itemsets frequentes de tamanho 2:

$$L_2 = \{\{1,3\},\{1,5\},\{1,4\},\{2,3\},\{3,4\},\{2,4\}\}$$

Então o conjunto dos pré-candidatos C'_3 da iteração seguinte será:

$$C'_3 = \{\{1,3,5\},\{1,3,4\},\{1,4,5\},\{2,3,4\}\}$$

Fase da Poda dos Candidatos. Utilizando novamente a propriedade Apriori, sabemos que se um itemset de C'_k possuir um subconjunto de itens (um subitemset) de tamanho $k-1$ que não estiver em L_{k-1} ele poderá ser descartado, pois não terá a menor chance de ser frequente. Assim, nesta fase é calculado o conjunto $C_k = C'_k - \{I \mid$ existe $J \subseteq I$ tal que $\mid J \mid = k-1$ e $J \notin L_{k-1}\}$. A notação $\mid J \mid$ significa "o número de elementos" do itemset J".

Exemplo 5.2.2 Consideremos a situação apresentada no exemplo 5.2.1. Neste caso, $C_3 = C'_3 - \{\{1,4,5\},\{1,3,5\}\} = \{\{1,3,4\},\{2,3,4\}\}$. O itemset {1,4,5} foi podado pois não tem chance nenhuma de ser frequente: ele contém o 2-itemset {4,5} que **não é frequente**, pois não aparece em L_2. Por que o itemset {1,3,5} foi podado ?

Fase do Cálculo do Suporte. Finalmente, nesta fase é calculado o suporte de cada um dos itemsets do conjunto C_k. Isto pode ser feito varrendo-se uma única vez o banco de dados $\mathcal{D}$: Para cada transação t de $\mathcal{D}$ verifica-se quais são os candidatos suportados por t e para estes candidatos incrementa-se de uma unidade o contador do suporte.

Como são calculados os itemsets frequentes de tamanho 1. Os itemsets de tamanho 1 são computados considerando-se todos os conjuntos unitários possíveis, de um único item. Em seguida, varre-se uma vez o banco de dados para calcular o suporte de cada um destes conjuntos unitários, eliminando-se aqueles que não possuem suporte superior ou igual ao mínimo exigido pelo usuário.

5.2.2 Técnica Apriori para Sequências: Algoritmo GSP

O algoritmo GSP é projetado para a tarefa de mineração de padrões sequencias ou sequências. Uma *sequência* ou *padrão sequencial* de tamanho k (ou k-sequência) é uma coleção ordenada de itemsets $< I_1, I_2, ..., I_n >$. Por exemplo, $s =$ <{TV,aparelho-de-som},{Vídeo}, {DVDPlayer}> é um padrão sequencial. Uma k-sequência é uma sequência com k *itens*. Um item que aparece em itemsets distintos é contado uma vez para cada itemset onde ele aparece. Por exemplo, $< \{1,2\} >, < \{1\},\{2\} >, < \{1\},\{1\} >$ são 2-sequências.

Definição 5.2.2 Sejam s e t duas sequências, $s = < i_1 i_2 ... i_k >$ e $t = < j_1 j_2 ... j_m >$. Dizemos que s *está contida* em t se existe uma subsequência de itemsets em t, $l_1, ..., l_k$ tal que $i_1 \subseteq l_1$, ... , $i_k \subseteq l_k$. Por exemplo, sejam $t =< \{1,3,4\},\{2,4,5\},\{1,7,8\} >$ e $s =< \{3\},\{1,8\} >$. Então, é claro que s está contida em t, pois {3} está contido no primeiro itemset de t e {1,8} está contido no terceiro itemset de t. Por outro lado, a sequência $s' =< \{8\},\{7\} >$ não está contida em t.

De agora em diante, vamos utilizar a seguinte nomenclatura para diferenciar as sequências: sequências que fazem parte do banco de dados de sequências (correspondem a alguma sequência de itemsets comprados por algum cliente) são chamadas de *sequência do cliente*. Sequências que são possíveis padrões que podem aparecer nos dados são chamadas de *padrão sequencial*.

Definimos *suporte* de um padrão sequencial s com relação a um banco de dados de sequências de clientes $\mathcal{D}$ como sendo a porcentagem de sequências de clientes que suportam s, isto é:

$$sup(s) = \frac{\text{número de sequências de clientes que suportam } s}{\text{número total de sequências de clientes}}$$

Um padrão sequencial s é dito *frequente* com relação a um banco de dados de sequências de clientes $\mathcal{D}$ e um nível minimo de suporte α, se $\text{sup}(s) \geq \alpha$. Assim como acontecia com os itemsets, a propriedade de ser frequente também é *antimonotônica*, no que diz respeito a padrões sequenciais. Isto é, se s é frequente então todos os seus subpadrões (ou subsequências) são frequentes. O problema da mineração de sequências consiste em, dados um banco de dados de sequências D e um nível mínimo de suporte α, encontrar todos os padrões sequenciais com suporte maior ou igual a α com relação a D.

Seguindo a mesma idéia dos algoritmos da família Apriori, o algoritmo GSP gera as k-sequências frequentes (sequência com k itens) na iteração k. Cada iteração é composta pelas fases de geração, de poda e de validação (cálculo do suporte).

Fase da Geração dos Candidatos. Caso $k \geq 3$

Suponhamos que L_{k-1} já tenha sido gerado na etapa $k-1$. Duas sequências $s = < s_1, s_2, \ldots, s_n >$ e $t =< t_1, t_2, \ldots, t_m >$ de L_{k-1} são ditas *ligáveis* se, retirando-se o primeiro item de s_1 e o último item de t_m as sequências resultantes são iguais. Neste caso, s e t podem ser ligadas e produzir a sequência v, onde:

- se t_m não é unitário: $v = < s_1, s_2, \ldots, s_n \cup t' >$, onde t' é o último item de t_m.
- se t_m é unitário: $v = < s_1, s_2, \ldots, s_n, t_m >$

Repare que estamos supondo que cada itemset está ordenado segundo a ordem lexográfica de seus itens. Estamos supondo também que o conjunto dos itens foi ordenado (a cada item foi associado um número natural). A Figura 5.6 ilustra o processo de junção de duas sequências de $k-1$ itens a fim de produzir uma sequência de k itens.

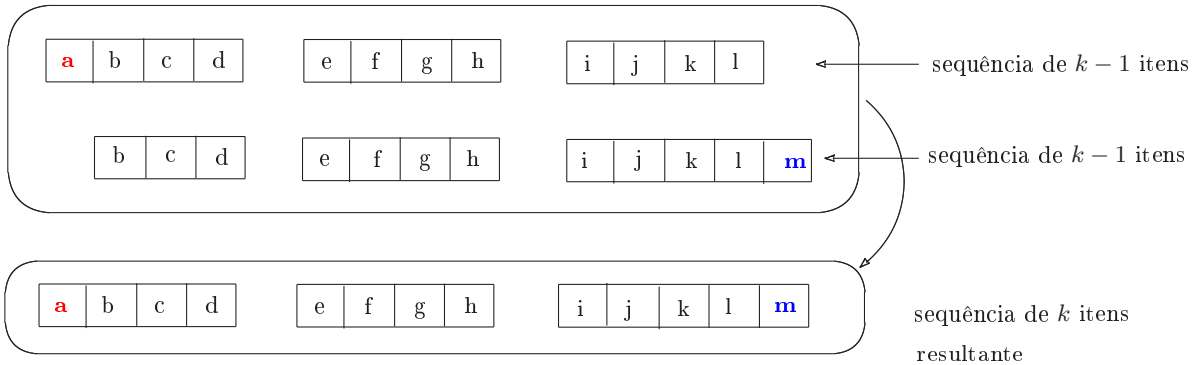


Figura 5.6: Como juntar sequências

Exemplo 5.2.3 Sejam $s =< \{1,2\}, \{3\}, \{5,7\} >, t =< \{2\}, \{3\}, \{5,7,10\} >$. Retirando-se o primeiro item de s_1 (o item 1) e o último item de t_3 (o item 10) obtemos a mesma sequência: $< \{2\}, \{3\}, \{5,7\} >$. Logo, s e t são ligáveis e sua junção produz a sequência $s =< \{1,2\}, \{3\}, \{5,7,10\} >$.

Definimos o conjunto dos pré-candidatos C'_k como sendo o conjunto obtido ligando-se todos os pares ligáveis de sequências de L_{k-1}.

Exemplo 5.2.4 As tabelas abaixo ilustram o conjunto L_3 (à esquerda) e o conjunto de candidatos C'_4 correspondente (à direita):

L_3
$< \{1,2\},\{3\} >$
$< \{1,2\},\{4\} >$
$< \{1\},\{3,4\} >$
$< \{1,3\},\{5\} >$
$< \{2\},\{3,4\} >$
$< \{2\},\{3\},\{5\} >$

C'_4
$< \{1,2\},\{3,4\} >$
$< \{1,2\},\{3\},\{5\} >$

Repare que uma propriedade da sequência resultante da junção de duas sequências s_1 e s_2 é que ao eliminarmos o primeiro item do primeiro itemset da junção obtemos s_2.

Caso $k = 2$: Para juntar duas sequências $s_1 =< \{x\} >$ e $s_2 =< \{y\} >$ de 1 item a fim de produzir uma de dois itens precisamos adicionar o item y de s_2 em s_1 tanto como parte do itemset $\{x\}$ quanto como um itemset isolado. Assim a junção de s_1 com s_2 produz duas sequências de 2 elementos: $< \{x,y\} >$ e $< \{x\},\{y\} >$.Repare que a propriedade acima mencionada se verifica para as duas sequências obtidas como resultado da junção de s_1 e s_2: nas duas sequências, ao eliminarmos o primeiro item do primeiro itemset obtemos a sequência $s_2 =< \{y\} >$.

Caso $k = 1$: O cálculo de C_1 é feito considerando-se todas as sequências de 1 item $< \{i\} >$ e testando-se o suporte para cada uma delas. As que são frequentes constituem o conjunto L_1.

Fase da Poda dos Candidatos. Seja s uma k-sequência. Se s for frequente, então, pela propriedade de antimonotonia, sabemos que toda subsequência de s deve ser frequente. Seja t uma subsequência qualquer obtida de s suprimindo-se um item de algum itemset. Se t não estiver em L_{k-1} então s não tem chance nenhuma de ser frequente e portanto pode ser podada.

Exemplo 5.2.5 Considere a mesma situação do exemplo 5.2.4. A sequência $< \{1,2\}, \{3\}, \{5\} >$ será podada, pois se retiramos o item 2 do primeiro itemset, a sequência resultante $< \{1\},\{3\},\{5\} >$ não está em L_3. Assim, após a fase da poda, o conjunto C_4 resultante é $\{ < \{1,2\},\{3,4\} > \}$.

Fase da Contagem do Suporte. A cada iteração, cada sequência de cliente d é lida *uma* vez e incrementa-se o contador dos candidatos de C_k que estão contidos em d. Assim, dado um conjunto C_k de sequências candidatas de uma sequência de cliente d, precisamos encontrar todas as sequências em C que estão contidas em d. Duas técnicas são utilizadas para resolver este problema: (1)usamos uma estrutura de árvore-hash para reduzir o número de candidatos de C que serão testados para d; (2) Transformamos a representação da sequência de cliente d de tal modo que possamos testar de forma eficiente se um determinado candidato de C é suportado (está contido) em d. Veremos detalhes da primeira técnica na Seção 5.4.

5.2.3 Mineração com Restrições nos Padrões

Imagine que você esteja interessado somente em minerar padrões sequenciais $< s_1, ..., s_n >$ que satisfazem uma determinada expressão regular, por exemplo, que começam por {TV} e terminam em {DVD Player}. Assim, somente serão gerados padrões satisfazendo a expressão regular: $\{TV\}a^*\{DVDPlayer\}$, onde a^* representa uma sequência qualquer de itemsets. Para ser mais exato: $a = (a_1 + a_2 + ... + a_n)$, onde $\{a_1, ..., a_n\}$ é o conjunto de todos os itemsets possíveis de serem formados com os itens dados. O problema de mineração que temos agora é o seguinte: Dados um banco de dados $\mathcal{D}$, um nível mínimo de suporte α e uma expressão regular $\mathcal{R}$, encontrar todas as sequências s com $\sup(s) \geq \alpha$ e que satisfazem $\mathcal{R}$.

Uma primeira idéia para resolver este problema de mineração seria a seguinte : Seja L^k = k-sequências frequentes satisfazendo $\mathcal{R}$.

Fase de Geração: usando L^k e $\mathcal{R}$, produzir um conjunto $\overline{C}^{k+1}$ de candidatos tais que (1) Os candidatos devem satisfazer $\mathcal{R}$; (2) Os candidatos são $k+1$-sequências potencialmente frequentes. Assim, os candidatos $\overline{C}^{k+1}$ devem conter L^{k+1}.

Fase da Podagem: Suprimir de $\overline{C}^{k+1}$ aquelas sequências σ que não têm nenhuma chance de serem frequentes. Repare que a dificuldade em utilizar esta idéia é que a fase de podagem deve ser efetuada utilizando somente o conjunto L^k calculado na fase anterior, e que é constituído de todas as sequências de tamanho k *que são frequentes e que satisfazem a expressão regular* R. Note que a restrição de ser frequente é *antimonotônica* mas a restrição de satisfazer uma expressão regular *não é*. Por exemplo, a sequência abb satisfaz a expressão regular ab^*, mas sua subsequência bb não satisfaz ab^*. Logo, na fase de podagem, não basta simplesmente eliminar as $k+1$-sequências que possuem uma k-sequência que não está em L^k.

Seja $L = L_1 \cup L_2 \cup ... \cup L_k$. Precisamos eliminar sequências σ que não sejam frequentes. Para isto, é *suficiente* que σ possua uma subsequência $\sigma' \subseteq \sigma$ que não seja frequente. Ora, se $\sigma' \notin L$ e σ' satisfaz a expressão regular R, teremos certeza de que σ' não é frequente. Assim:

$C^{k+1} = \overline{C}^{k+1} - \{\sigma \in \overline{C}^{k+1} \mid \exists \sigma' \subseteq \sigma, \quad \sigma' \notin L \text{ e } \sigma' \text{ satisfaz } \mathcal{R}\}$

O problema com esta idéia é o seguinte: Seja $A^{k+1} = \{\sigma \in \overline{C}^{k+1} \mid \exists \sigma' \subseteq \sigma, \quad \sigma' \notin L \text{ e } \sigma' \models \mathcal{R}\}$ o conjunto de sequências que são podadas. Repare que quanto *mais* restritiva for a expressão regular $\mathcal{R}$, *menor* será o conjunto A^{k+1}, isto é, menos sequências serão podadas. A tabela abaixo ilustra este fato:

"Poder de Restrição" de $\mathcal{R}$	$\overline{C}^{k+1}$	A^{k+1}	$\overline{C}^{k+1} - A^{k+1}$
$\uparrow$	$\downarrow$	$\downarrow$	$\uparrow$

Assim, a introdução da restrição $\mathcal{R}$, por um lado, na fase de geração restringe os candidatos gerados, mas por outro lado, na fase da podagem, também restringe as sequências podadas, o que não é interessante. Precisamos encontrar uma espécie de "meio-termo": como restringir suficientemente os candidatos na fase de geração sem diminuir muito o conjunto de sequências que serão podadas na fase de podagem ?

Idéia: Considerar um "relaxamento" apropriado da expressão regular $\mathcal{R}$. O que é um "relaxamento" de $\mathcal{R}$? Sabemos que a expressão regular $\mathcal{R}$ especifica uma *linguagem regular*, isto é, o conjunto de todas as palavras (sequências) que satisfazem

$\mathcal{R}$. Um "relaxamento" de $\mathcal{R}$ seria qualquer condição c (inclusive uma outra expressão regular $\mathcal{R}'$) mais fraca do que $\mathcal{R}$, isto é, tal que a linguagem satisfazendo c contivesse a linguagem satisfazendo $\mathcal{R}$. Assim, c é menos restritiva do que $\mathcal{R}$. Que tipo de relaxamento seria considerado "apropriado" ? Cada relaxamento $\mathcal{R}'$ de $\mathcal{R}$ corresponde a um Algoritmo SPIRIT($\mathcal{R}'$), cuja idéia geral de execução é a descrita acima, mas considerando, ao invés de $\mathcal{R}$, a condição $\mathcal{R}'$. Estuda-se as performances dos diversos algoritmos da família e chega-se à conclusão, de forma experimental, qual o relaxamento mais apropriado.

5.2.4 Os Quatro Algoritmos Principais da Família SPIRIT

Antes de discutirmos estes algoritmos, notamos que se $\mathcal{R}$ é *antimonotônica* então as fases de geração e podagem são exatamente como nos algoritmos da família Apriori. Neste caso, não é necessário procurar um relaxamento $\mathcal{R}'$ de $\mathcal{R}$, pois as fases de geração e podagem estarão em "sintonia": Se σ é frequente e satisfaz $\mathcal{R}$ e $\sigma' \subseteq \sigma$ então σ' deve ser frequente e satisfazer $\mathcal{R}$. Logo, a fase da podagem consiste simplesmente em eliminar as sequências candidatas $\sigma \in \overline{C}^{k+1}$ tais que $\exists \sigma' \subseteq \sigma,\ \sigma' \notin L^k$.

Os quatro principais algoritmos da família SPIRIT são SPIRIT(N), SPIRIT(L), SPIRIT(V) e SPIRIT($\mathcal{R}$), cada um deles correspondente a um relaxamento da restrição $\mathcal{R}$.

1. **SPIRIT(N)**: aqui consideramos o maior de todos os relaxamentos de $\mathcal{R}$, aquele que não impõe nenhuma restrição às sequências. Assim, neste caso, qualquer sequência satisfaz a "restrição" N.

2. **SPIRIT(L)**: neste relaxamento, somente são consideradas as sequências *legais* com respeito a algum estado do autômato correspondente à expressão regular $\mathcal{R}$, que denotamos por $A_{\mathcal{R}}$. Dizemos que uma sequência $a_1 a_2 ... a_n$ é *legal* com respeito ao estado q do autômato $\mathcal{R}$ se existe um caminho no autômato que começa no estado q e que percorre a palavra $a_1 a_2 ... a_n$.

3. **SPIRIT(V)**: neste relaxamento, somente são consideradas as sequências *válidas* com respeito a algum estado do autômato $A_{\mathcal{R}}$. Dizemos que uma sequência $a_1 a_2 ... a_n$ é *válida* com respeito ao estado q do autômato $\mathcal{R}$ se existe um caminho no autômato que começa no estado q e *termina num estado final* e que percorre a palavra $a_1 a_2 ... a_n$.

4. **SPIRIT($\mathcal{R}$)**: este, não é um relaxamento. Corresponde exatamente à expressão $\mathcal{R}$. Somente as sequências *válidas* (isto é, aquelas que começam no estado inicial e terminam num estado final do autômato) são aceitas.

A Figura 5.7 ilustra as noções de sequências *legais* com respeito a algum estado de $\mathcal{A}_{\mathcal{R}}$, de sequências *válidas* com respeito a algum estado de $\mathcal{A}_{\mathcal{R}}$ e de sequências válidas com respeito a $\mathcal{A}_{\mathcal{R}}$.

A sequência $< 1, 2 >$ é legal com respeito ao estado a do autômato, pois existe um caminho no autômato percorrendo a sequência $< 1, 2 >$.

A sequência $< 2 >$ é válida com respeito ao estado b do autômato, pois existe um caminho no autômato, saindo do estado b, percorrendo a sequência $< 2 >$ e chegando num estado final.

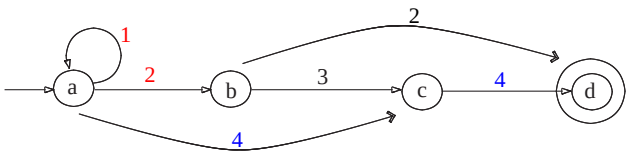


Figura 5.7: Sequências Legais e Válidas

sequência $< 4, 4 >$ é válida, pois existe um caminho no autômato, saindo do estado inicial, percorrendo a sequência $< 4, 4 >$ e chegando num estado final.

A tabela abaixo resume as restrições consideradas por cada um dos algoritmos SPIRIT.

Algoritmo	**Relaxamento $\mathcal{R}'$**
SPIRIT(N)	nenhuma restrição
SPIRIT(L)	somente sequências *legais* com respeito a algum estado de $\mathcal{A}_{\mathcal{R}}$
SPIRIT(V)	somente sequências *válidas* com respeito a algum estado de $\mathcal{A}_{\mathcal{R}}$
SPIRIT(R)	somente sequências válidas ($\mathcal{R}' = \mathcal{R}$)

O esquema geral dos algoritmos SPIRIT é o seguinte:

Etapa1 1: Etapa do relaxamento R'. Calcula-se o conjunto L' das sequências frequentes que satisfazem um relaxamento R' da expressão regular R original fornecida pelo usuário (vamos denotar A_R o autômato correspondente a R). R' pode ser: (1) o relaxamento total (algoritmo SPIRIT(N)), (2) o relaxamento correspondente às sequências legais com respeito a algum estado do autômato A_R (algoritmo SPIRIT(L)), (3) o relaxamento correspondente às sequências válidas com respeito a algum estado do autômato A_R (algoritmo SPIRIT(V)), (4) nenhum relaxamento, isto é, R' = R (algoritmo SPIRIT(R)).

Etapa 2: Etapa da Restrição R. Elimina-se de L' as sequências que não satisfazem R, obtendo-se assim o conjunto L das sequências frequentes e que satisfazem R. Isto se faz através de um procedimento que dado um autômato e um string, verifica se o string é ou não aceito pelo autômato.

Repare que o algoritmo SPIRIT(N) corresponde a aplicar o algoritmo GSP sem nenhuma restrição na fase de geração (Etapa 1). A etapa 2 corresponde a uma etapa de pós-processamento, onde são eliminadas as sequências que não interessam. Num outro extremo está o algoritmo SPIRIT(R), onde a etapa 2 não realiza nada, pois a etapa 1 já fornece o conjunto L das sequências frequentes e que satisfazem R.

Notas Bibliográficas. Detalhes do algoritmo Apriori e análise dos resultados experimentais podem ser encontrados no artigo [AS 1994]. Em [ASV 1997] são introduzidas restrições nos itemsets e diversos algoritmos de mineração são propostos. O algoritmo GSP foi introduzido em [AS 1996], onde foi adaptada para incorporar diversos tipos de restrições nos padrões, na fase de cálculo do suporte. Outros algoritmos de mineração de sequências, de desempenho inferior ao GSP tinham sido introduzidos anteriormente em [AS 1995]. Foi neste artigo onde foi tratado primeiramente o problema da mineração de sequências. O problema da mineração com restrições de expressão regular é tratado em [GRS 1999, GRS 2002].

5.3 Técnicas para Classificação e Análise de Clusters

Nesta seção, vamos estudar duas outras tarefas de mineração que, como mencionamos na Seção 5.1, estão de certa forma relacionadas. Trata-se das tarefas de classificação e

análise de *clusters*. Para cada uma destas tarefas, veremos algumas técnicas comumente utilizadas para realizá-las. Começaremos pela Classificação.

Suponha que você é gerente de uma grande loja e disponha de um banco de dados de clientes, contendo informações tais como *nome, idade, renda mensal, profissão* e se comprou ou não produtos eletrônicos na loja. Você está querendo enviar um material de propaganda pelo correio a seus clientes, descrevendo novos produtos eletrônicos e preços promocionais de alguns destes produtos. Para não fazer despesas inúteis você gostaria de enviar este material publicitário apenas a clientes que sejam potenciais compradores de material eletrônico. Outro ponto importante : você gostaria de, a partir do banco de dados de clientes de que dispõe no momento, desenvolver um método que lhe permita saber que tipo de atributos de um cliente o tornam um potencial comprador de produtos eletrônicos e aplicar este método no futuro, para os novos clientes que entrarão no banco de dados. Isto é, a partir do banco de dados que você tem hoje, você quer descobrir regras que classificam os clientes em duas classes : os que compram produtos eletrônicos e os que não compram. Que tipos de atributos de clientes (idade, renda mensal, profissão) influenciam na colocação de um cliente numa ou noutra classe ? Uma vez tendo estas regras de classificação de clientes, você gostaria de utilizá-las no futuro para classificar novos clientes de sua loja. Por exemplo, regras que você poderia descobrir seriam: *Se idade está entre 30 e 40 e a renda mensal é 'Alta' então ClasseProdEletr = 'Sim'. Se idade está entre 60 e 70 então ClasseProdEletr = 'Não'.* Quando um novo cliente João, com idade de 25 anos e renda mensal 'Alta' e que tenha comprado discos, é catalogado no banco de dados, o seu classificador lhe diz que este cliente é um potencial comprador de aparelhos eletrônicos. Este cliente é colocado na classe ClasseProdEletr = 'Sim', mesmo que ele ainda não tenha comprado nenhum produto eletrônico.

O que é um classificador ?. *Classificação* é um processo que é realizado em três etapas :

1. Etapa da criação do modelo de classificação. Este modelo é constituído de regras que permitem classificar as tuplas do banco de dados dentro de um número de classes pré-determinado. Este modelo é criado a partir de um banco de dados de *treinamento.* cujos elementos são chamados de *amostras ou exemplos.*

2. Etapa da verificação do modelo ou *Etapa de Classificação* : as regras são testadas sobre um outro banco de dados, completamente independente do banco de dados de treinamento, chamado de *banco de dados de testes.* A qualidade do modelo é medida em termos da porcentagem de tuplas do banco de dados de testes que as regras do modelo conseguem classificar de forma satisfatória. É claro que se as regras forem testadas no próprio banco de dados de treinamento, elas terão alta probabilidade de estarem corretas, uma vez que este banco foi usado para extraí-las. Por isso a necessidade de um banco de dados completamente novo.

Diversas técnicas são empregadas na construção de classificadores. Neste curso vamos ver duas técnicas: árvores de decisão e redes neurais.

5.3.1 Árvore de Decisão

Uma *árvore de decisão* é uma estrutura de árvore onde: (1) cada nó interno é um atributo do banco de dados de amostras, diferente do atributo-classe, (2) as folhas são valores do atributo-classe, (3) cada ramo ligando um nó-filho a um nó-pai é etiquetado com um

valor do atributo contido no nó-pai. Existem tantos ramos quantos valores possíveis para este atributo. (4) um atributo que aparece num nó não pode aparecer em seus nós descendentes.

Exemplo 5.3.1 Considere o banco de dados de treinamento:

Nome	Idade	Renda	Profissão	ClasseProdEletr
Daniel	=< 30	Média	Estudante	Sim
João	31..50	Média-Alta	Professor	Sim
Carlos	31..50	Média-Alta	Engenheiro	Sim
Maria	31..50	Baixa	Vendedora	Não
Paulo	=< 30	Baixa	Porteiro	Não
Otávio	> 60	Média-Alta	Aposentado	Não

A Figura 5.8 ilustra uma possível árvore de decisão sobre este banco de dados.

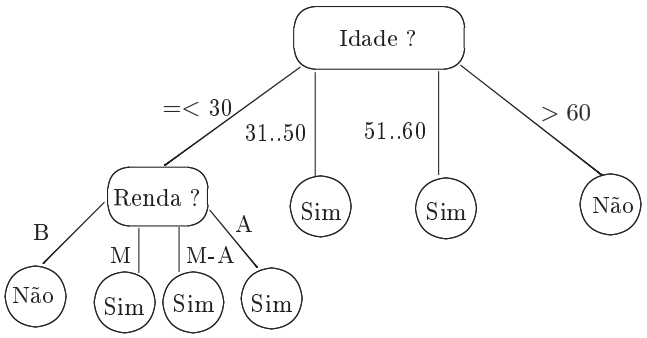


Figura 5.8: Uma árvore de decisão

Idéia geral de como criar uma árvore de decisão. A idéia geral é a que está por trás do algoritmo ID3, criado por Ross Quinlan, da Universidade de Sydney em 1986 e de seus sucessores (um deles, o algoritmo C4.5 também proposto por Quinlan em 1993). Trata-se do procedimento recursivo:
Gera-Arvore($\mathcal{A}$,Cand-List)
Entrada: Um banco de dados de amostras $\mathcal{A}$ onde os valores dos atributos foram categorizados, uma lista de atributos candidatos Cand-List.
Output: Uma árvore de decisão
Método
(1) Crie um nó N; Associe a este nó o banco de dados $\mathcal{A}$
(2) Se todas as tuplas de $\mathcal{A}$ pertencem à mesma classe C então transforme o nó N numa folha etiquetada por C. *Páre.*
(3) Caso contrário: Se Cand-List = $\emptyset$ então transforme N numa folha etiquetada com o valor do atributo-Classe que mais ocorre em $\mathcal{A}$. *Páre.*
(4) Caso contrário: calcule Ganho(Cand-List). Esta função retorna o atributo com o maior ganho de informação. Será detalhada no próximo parágrafo. Chamamos este atributo de Atributo-Teste.
(5) Etiquete N com o nome de Atributo-Teste
(6) Etapa da partição das amostras $\mathcal{A}$: para cada valor s_i do Atributo-Teste faça o

seguinte:
(7) Crie um nó-filho N_i, ligado a N por um ramo com etiqueta igual ao valor s_i e associe a este nó o conjunto $\mathcal{A}_i$ das amostras tais que o valor de Atributo-Teste = s_i.
(8) Se $\mathcal{A}_i = \emptyset$: transforme o nó N_i numa folha etiquetada pelo valor do atributo-Classe que mais ocorre em $\mathcal{A}$.
(9) Caso contrário: calcule Gera-Arvore($\mathcal{A}_i$, Cand-List - {Atributo-Teste}) e "grude" no nó N_i a árvore resultante deste cálculo.

Como decidir qual o melhor atributo para dividir o banco de amostras ?. Agora vamos detalhar a função Ganho(Cand-List) que decide qual atributo em Cand-List é o mais apropriado para ser utilizado no particionamento das amostras. Vamos utilizar como exemplo o banco de dados amostral da Figura 5.9 sobre condições meteorológicas. O objetivo é identificar quais as condições ideiais para se jogar um determinado jogo.

Aparência	Temperatura	Humidade	Vento	Jogo
Sol	Quente	Alta	Falso	Não
Sol	Quente	Alta	Verdade	Não
Encoberto	Quente	Alta	Falso	Sim
Chuvoso	Agradável	Alta	Falso	Sim
Chuvoso	Frio	Normal	Falso	Sim
Chuvoso	Frio	Normal	Verdade	Não
Encoberto	Frio	Normal	Verdade	Sim
Sol	Agradável	Alta	Falso	Não
Sol	Frio	Normal	Falso	Sim
Chuvoso	Agradável	Normal	Falso	Sim
Sol	Agradável	Normal	Verdade	Sim
Encoberto	Agradável	Alta	Verdade	Sim
Encoberto	Quente	Normal	Falso	Sim
Chuvoso	Agradável	Alta	Verdade	Não

Figura 5.9: Banco de dados amostral

Vamos considerar as quatro possibilidades para a escolha do atributo que será utilizado para dividir o banco de dados no primeiro nível da árvore. Estas possibilidades estão ilustradas na Figura 5.10.

Qual a melhor escolha ? Repare que se uma folha só tem 'Sim' ou só tem 'Não', ela não será mais dividida no futuro: o processo GeraArvore aplicado a esta folha pára logo no início. Gostaríamos que isto ocorresse o mais cedo possível, pois assim a árvore produzida será menor. Assim, um critério *intuitivo* para a escolha do atributo que dividirá um nó seria: "Escolha aquele que produz os nós mais puros". Por exemplo, no nosso caso, a escolha boa seria o atributo *Aparência*.

Grau de Pureza de um atributo num nó: Entropia. Vamos definir uma função Info que calcula o *grau de pureza* de um atributo num determinado nó. Este grau de pureza representa a *a quantidade de informação esperada que seria necessária para especificar se uma nova instância seria classificada em 'Sim' ou 'Não', uma vez chegado a este nó.* A idéia é a seguinte: se A_1, A_2, ..., A_n são as folhas (tabelas) saindo deste nó, n_i = tamanho de A_i e N = total dos tamanhos das tabelas, então Info(Nó) = $\Sigma_{i=1}^{n} \frac{n_i}{N}$ Entropia(A_i). Quanto maior a entropia, maior a informação. A entropia é uma medida estatística que mede o quão "confuso" é a distribuição das tuplas entre as classes. Por exemplo, se existem

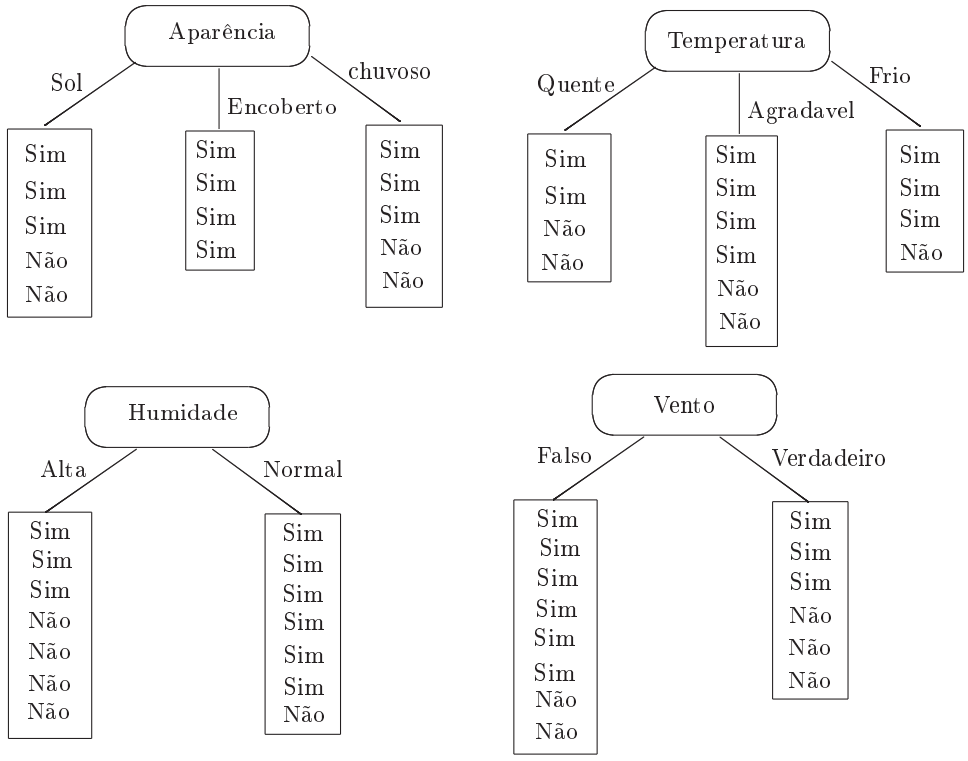


Figura 5.10: As quatro possibilidades para o atributo do nó raiz

2 classes, e exatamente metade das tuplas estão numa classe e a outra metade na outra classe, então a entropia seria maximal. Por outro lado, se todas as tuplas estão numa mesma classe, então a entropia é zero.

Seja A_i uma tabela com n_i tuplas, das quais S_i estão classificadas como 'Sim' e N_i estão classificadas como 'Não'. Então a entropia de A_i é definida como:

$$\text{Entropia}(A_i) = -(\frac{S_1}{n_i} log_2 \frac{S_1}{n_i} + \frac{N_i}{n_i} log_2 \frac{N_i}{n_i})$$

Esta fórmula para entropia é bem conhecida. Atente para o sinal negativo, necessário pois a entropia deve ser positiva e os logaritmos são negativos (já que são calculados sobre números entre 0 e 1). Esta fórmula é generalizada (da maneira óbvia) para um número de classes qualquer.

Exemplo 5.3.2 Consideremos as quatro possibilidades para o atributo do primeiro nó, conforme ilustrado na Figura 5.10.

- Se escolhemos o atributo Aparência:
  Info(Nó) = $\frac{5}{14}$ entropia(Folha 1) + $\frac{4}{14}$ entropia(Folha 2) + $\frac{5}{14}$ entropia(Folha3)

  entropia(Folha 1) = $\frac{2}{5}\ log_2\frac{2}{5} + \frac{3}{5}\ log_2\frac{3}{5} = 0.971$
  entropia(Folha 2) = $\frac{4}{4}\ log_2\frac{5}{5} + \frac{0}{4}\ log_2\frac{0}{4} = 0$
  entropia(Folha 3) = $\frac{3}{5}\ log_2\frac{3}{5} + \frac{2}{5}\ log_2\frac{2}{5} = 0.971$
  Logo, Info(Nó) = $\frac{5}{14}$ 0.971 + $\frac{4}{14}$ 0 + $\frac{5}{14}$ 0.971 = 0.693

- Se escolhemos o atributo Temperatura:
  Info(Nó) = $\frac{4}{14}$ entropia(Folha 1) + $\frac{6}{14}$ entropia(Folha 2) + $\frac{4}{14}$ entropia(Folha3) = 0.911

- Se escolhemos o atributo Humidade:
  Info(Nó) = $\frac{7}{14}$ entropia(Folha 1) + $\frac{7}{14}$ entropia(Folha 2) = 0.788

- Se escolhemos o atributo Vento:
  Info(Nó) = $\frac{8}{14}$ entropia(Folha 1) + $\frac{6}{14}$ entropia(Folha 2) = 0.892

Ganho de Informação ao escolher um Atributo. O ganho de informação ao escolher um atributo A num nó é a diferença entre a informação associada ao nó *antes*(Info-pré) da divisão e a informação associada ao nó após a divisão (Info-pós).

Info-pós = a informação do nó (Info(Nó)) que calculamos no passo anterior, ao escolhermos A como atributo divisor.
Info-pré = entropia do nó antes da divisão = $\frac{N_{Sim}}{N} log_2 \frac{N_{Sim}}{N} + \frac{N_{Nao}}{N} log_2 \frac{N_{Nao}}{N}$, onde N_{Sim} = total de tuplas classificadas como Sim; N_{Nao} = total de tuplas classificadas como Não; N = total de tuplas no nó.

Exemplo 5.3.3 Consideremos a situação do exemplo 5.3.2. Temos que Info-pré = $\frac{9}{14} log_2 \frac{9}{14} + \frac{5}{14} log_2 \frac{5}{14} = 0.940$. Logo, os ganhos de informação de cada uma das quatro escolhas são:
ganho(Aparência) = 0.940 - 0.693 = 0.247
ganho(Temperatura) = 0.940 - 0.911 = 0.029
ganho(Humidade) = 0.940 - 0.788 = 0.152
ganho(Vento) = 0.940 - 0.892 = 0.020

Logo, o atributo ideal para dividir as amostras é o atributo Aparência, como era de se supor deste o início. Veja que é o único atributo onde uma das folhas é "arrumadinha", todas as tuplas pertencendo a uma única classe.

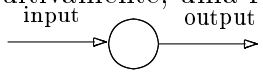
Como transformar uma árvore de decisão em regras de classificação. Uma árvore de decisão pode ser facilmente transformada num conjunto de regras de classificação. As regras são do tipo: *IF* L_1 *AND* L_2 ... *AND* L_n *THEN* Classe = Valor, onde L_i são expressões do tipo Atributo = Valor. Para cada caminho, da raiz até uma folha, tem-se uma regra de classificação. Cada par (atributo,valor) neste caminho dá origem a um L_i. Por exemplo, a árvore de decisão do exemplo 5.3.1 corresponde ao seguinte conjunto de regras de classificação:

- **IF** Idade =< 30 **AND** Renda = Baixa **THEN** Classe = Não
- **IF** Idade =< 30 **AND** Renda = Média **THEN** Classe = Sim
- **IF** Idade =< 30 **AND** Renda = Média-Alta **THEN** Classe = Sim
- **IF** Idade =< 30 **AND** Renda = Alta **THEN** Classe = Sim
- **IF** Idade 31..50 **THEN** Classe = Sim
- **IF** Idade 51..60 **THEN** Classe = Sim
- **IF** Idade > 60 **THEN** Classe = Não

5.3.2 Redes Neurais

Até o momento, temos dois tipos de conceitos que podem ser produzidos em resposta a uma tarefa de classificação: (1)Regras de Classificação; (2)Árvore de Decisão. Uma árvore de decisão pode ser facilmente transformada num conjunto de regras de classificação e vice-versa. Nesta subseção, vamos ver um terceiro conceito que pode ser produzido em resposta a uma tarefa de classificação: uma rede neural. O algoritmo de classificação terá como entrada um banco de dados de treinamento e retornará como output uma rede neural. Esta rede também poderá ser transformada num conjunto de regras de classificação, como foi feito com as árvores de decisão. A única diferença é que esta transformação não é tão evidente como no caso das árvores de decisão.

As redes neurais foram originalmente projetadas por psicólogos e neurobiologistas que procuravam desenvolver um conceito de *neurônio artificial* análogo ao neurônio natural. Intuitivamente, uma rede neural é um conjunto de unidades do tipo:



Tais unidades são conectadas umas às outras e cada conexão tem um *peso* associado. Cada unidade representa um *neurônio*. Os pesos associados a cada conexão entre os diversos neurônios é um número entre -1 e 1 e mede de certa forma qual a *intensidade* da conexão entre os dois neurônios. O processo de aprendizado de um certo conceito pela rede neural corresponde à associação de pesos adequados às diferentes conexões entre os neurônios. Por esta razão, o aprendizado utilizando Redes Neurais também é chamado de *Aprendizado Conexionista*.

Mais precisamente: uma *Rede Neural* é um diagrama como mostra a Figura 5.11.

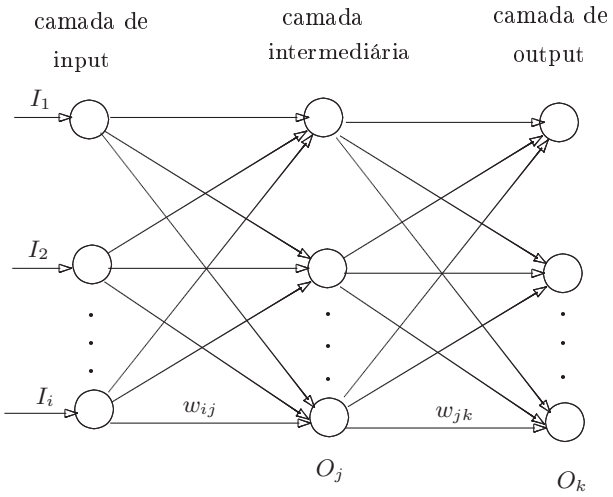


Figura 5.11: Uma rede neural

A rede é composta de diversas camadas (verticais): (1) *Camada de Input*: consiste dos nós da primeira coluna na Figura 5.11. Estes nós correspondem aos atributos (distintos do atributo classe) do banco de dados de treinamento. (2) *Camada de Output*: consiste dos nós da última coluna na Figura 5.11. Estes nós são em número igual ao número de classes. Eles correspondem, de fato, aos possíveis valores do Atributo-Classe. (3)*Camadas escondidos ou intermediários*: consiste dos nós das colunas intermediárias (indo da segunda até a penúltima). Numa rede neural existe pelo menos uma camada intermediária. Além disto, a seguinte propriedade se verifica: Cada nó de uma camada i deve estar conectado a todo nó da camada seguinte $i + 1$.

Uma rede neural é dita de n *camadas* se ela possui $n-1$ camadas intermediárias. Na verdade, conta-se apenas as camadas intermediárias e a de output. Assim, se a rede tem uma camada intermediária, ela será chamada de rede de *duas camadas*, pois possui uma camada intermediário e uma de output. A camada de input não é contado, embora sempre exista.

Como utilizar uma rede neural para classificação. Antes de entrar nos detalhes do algoritmo de classificação, vamos descrever de forma geral, como é o processo de classificação utilizando uma rede neural.

Dispõe-se de um banco de dados de treinamento composto de uma única tabela. Uma das colunas desta tabela corresponde ao Atributo-Classe. As amostras já estão classificadas. A rede será *treinada* para aprender (em cima destas amostras) como classificar corretamente novos dados. Para cada amostra $X = (x_1, ..., x_n)$, onde $x_1, ..., x_n$ são os valores correspondentes aos atributos não-classe, os elementos $x_1, ..., x_n$ são fornecidos a cada uma das unidades da camada de input. Cada unidade da camada de input fornece como output o mesmo x_i que recebeu como input. Cada unidade da próxima camada receberá como input uma combinação adequada (envolvendo os pesos das conexões) dos outputs de cada uma das unidades da camada precedente à qual está conectada. Esta unidade retornará como output o resultado da aplicação de uma certa *função de ativação* aplicada ao valor que recebeu como input. Este processo vai se repetindo camada por camada até chegar na última (camada de output). Seja C a classe à qual pertence a amostra X. Suponha que C corresponde à i-ésima unidade de output. Então o valor que deveria ter sido produzido pela rede se ela estivesse bem treinada seria (0,0,...,1,0,0,...0), onde o número 1 aparece na i-ésima coordenada. Calcula-se a diferença entre o vetor $(O_1, ..., O_n)$ (onde cada O_j é o valor de output produzido na j-ésima unidade da camada de output, para $j = 1, ..., n$) e o vetor ideal (0,0,...,1,0,0,...0) que deveria ter sido produzido se a rede estivesse treinada. Seja ϵ_j o valor da j-ésima coordenada do vetor (0,0,...,1,0,0,...0), para $j = 1, ..., n$. Caso a diferença $\Delta = \min \{ \mid O_j - \epsilon_j \mid, j = 1...n\}$ seja muito grande , é porque a rede ainda não está bem treinada, não aprendeu ainda a classificar uma amostra corretamente. Um processo de percurso inverso se inicia, com os pesos das conexões sendo reavaliados de acordo com uma função que depende da diferença Δ. Este processo é chamado de *Backpropagation*. Quando este processo de *Backpropagation* termina, estamos novamente na situação inicial, só que os pesos das conexões foram alterados. Agora, a segunda amostra do banco de dados será utilizada para o aprendizado da mesma forma como o foi a primeira. Quando todas as amostras do banco de dados foram escaneadas pela rede, tudo se repete a partir da primeira amostra. Cada iteração correspondendo a varrer o banco de dados de amostras uma vez é chamado de *época*.

O algoritmo pára quando uma das condições a seguir se verifica numa determinada época: (1) Na época precedente, para cada amostra testada, todas as diferenças entre os pesos Δw_{ij} são muito pequenas, isto é, menor que um certo nível mínimo fornecido. (2) Só uma pequena porcentagem de amostras (abaixo de um nível mínimo fornecido) foram mal classificadas na época precedente. (3) Um número de épocas máximo pré-especificado já se passou.

Na prática, centenas de milhares de épocas são necessárias para que os pesos convirjam. Quando isto acontece, dizemos que a rede neural está *treinada*. Teoricamente, a convergência não é garantida, mas na prática em geral, ela ocorre depois de um grande número de épocas.

Antes que o processo acima descrito comece a ser executado, obviamente é preciso

estabelecer como será a *topologia* da rede neural que será treinada para classificar corretamente as amostras. Esta *topologia* consiste na especificação do número de camadas intermediárias e do número de unidades em cada camada. Também é necessário inicializar os parâmetros de aprendizado: os pesos entre as diferentes conexões e os parâmetros envolvidos na função que calcula o input de cada unidade j a partir dos outputs recebidos das unidades da camada precedente; além disto, é preciso especificar qual a função (de ativação) que calcula o output de cada unidade j a partir do input recebido.

Como definir a melhor topologia de uma rede neural para uma certa tarefa de classificação.

1. *O número de unidades na camada de input*: Para cada atributo diferente do atributo classe, suponhamos que o número de valores deste atributo é n_A (é importante categorizar o domínio de cada atributo de modo que os valores assumidos sejam poucos). O número de unidades de input será igual a $n_{A_1}+...+n_{A_k}$ onde $A_1, ..., A_k$ são os atributos distintos do atributo classe. Por exemplo, suponhamos que Idade e RendaMensal sejam atributos não-classe e seus valores respectivos são: $\leq 30, 30..40, 40..50, \geq 50$ e {Baixa,Média,Média-Alta,Alta}. Então, teremos pelo menos 8 unidades de input, 4 para o atributo Idade (a primeira para o valor ≤ 30, a segunda para o valor 30..40, etc) e 4 para o atributo RendaMensal (a primeira para o valor 'Baixa', a segunda para o valor 'Média', etc). Uma amostra X tendo Idade = 30..40 e RendaMensal = 'Alta' vai entrar os seguintes inputs para as unidades: Unidade 1 = 1, Unidade 2 = 0, Unidade 3 = 0, Unidade 4 = 0, Unidade 5 = 0, Unidade 6 = 0, Unidade 7 = 0, Unidade 8 = 1.

   Caso não seja possível categorizar os atributos, então **normaliza-se** os valores dos atributos de modo a ficarem entre 0 e 1 ou entre -1 e 1. E constrói-se tantas unidades de input quanto for o número de atributos.

2. *O número de unidades na camada de output*: Se existirem somente duas classes, então teremos apenas uma unidade de output (o valor 1 de output nesta unidade representa a classe 1, o valor 0 de output nesta unidade representa a classe 0). Se existirem mais de duas classes, então teremos uma unidade de output para cada classe.

3. *O número de camadas intermediárias*: Normalmente é utilizada uma única camada intermediária. Não há regra clara para determinar este número.

4. *O número de unidades nas camadas intermediárias*: Não há regra clara para determinar este número. Determinar a topologia da rede é um processo de tentativa e erro. Este número de unidades nas camadas intermediárias pode afetar o processo de aprendizagem. Uma vez que uma rede neural foi treinada e o grau de acertos de sua atividade de classificação não é considerado bom na fase de testes, é comum repetir todo o processo de aprendizado com uma rede neural com topologia diferente.

Como inicializar os parâmetros da rede. Os pesos iniciais são inicializados por números pequenos (entre -1 e 1 ou entre -0.5 e 0.5). Outros parâmetros que determinam a função que calcula o input de cada unidade (numa camada intermediária ou de output) em função dos outputs das unidades da camada precedente são também inicializados por valores pequenos. Tais parâmetros serão detalhados a seguir. Assim como a definição

adequada da topologia da rede, a boa inicialização dos parâmetros da rede é um processo de tentativa e erro. Uma rede treinada que não produz resultados satisfatórios com uma base de testes pode ser re-treinada com outros valores iniciais para os pesos e demais parâmetros até que consiga bons resultados na etapa de testes.

O algoritmo de classificação por Backpropagation utilizando Redes Neurais. Vamos descrever abaixo cada passo do Algoritmo de classificação Backpropagation. O algoritmo vai varrer todas as amostras do banco de dados. Para cada amostra ele vai percorrer para frente a rede neural e depois para trás. Cada iteração do algoritmo corresponde a varrer todas as amostras do banco de dados. As etapas do algoritmo são descritas a seguir:

1. Inicializa os pesos de cada conexão e as tendências de cada unidade. Normalmente, os valores empregados são números muito pequenos entre -1 e 1 ou entre -0.5 e 0.5. A taxa de aprendizado é normalmente especificada pela função$\lambda(t) = \frac{1}{t}$, onde t é o número correspondente à iteração em que se está. Assim, na primeira iteração, a taxa é 1, na segunda é $\frac{1}{2}$, etc.

2. *Iteração 1:* Inicia-se o processo de varrer o banco de amostras. Para cada amostra X, entra-se os valores dos atributos de X na camada de input. Propaga-se os inputs para a frente na rede neural até chegar na camada de output final.

3. Chegando na camada final, faz-se um teste para ver se X foi ou não bem classificada de acordo com o limite mínimo de exigência estabelecido no input do algoritmo.

4. Independentemente do resultado do teste, inicia-se o processo de volta recalculando todos os pesos das conexões e tendências das unidades até chegar na camada de input.

5. Existem duas possibilidades para os updates dos pesos e tendências:(1) Estes são atualizados a cada passada por uma amostra. (2) Para cada amostra os novos pesos e tendências são estocados em variáveis, mas não são atualizados. Assim, a próxima amostra utiliza os mesmos pesos e tendências utilizados na amostra precedente. A atualização só ocorre no final da iteração, quando todas as amostras foram varridas. Normalmente, utiliza-se a primeira política de atualização, que produz melhores resultados.

   A iteração 1 termina quando todo o banco de dados de amostras foi varrido.

6. *Iteração k ($k > 1$)* : o processo se repete, isto é, o banco de dados é varrido novamente com os novos pesos e tendências atualizados para cada amostra.

7. *Condições de Parada*: como já dissemos antes, o algoritmo pára numa certa iteração k quando uma das condições se verifica (em ordem de prioridade):(1) Os Δ_{ij} calculados na iteração $k-1$ são todos muito pequenos, abaixo de um limite mínimo especificado no input do algoritmo. (2) A porcentagem de amostras malclassificadas na iteração precedente está abaixo de um limite mínimo especificado no input do algoritmo.(3) k é maior do que um limite máximo especificado no input do algoritmo.

Cada iteração do algoritmo corresponde a uma *época*.
Os processos de propagação para frente e para trás são descritos a seguir:

1. *Fase de ida: Como são calculados os inputs das unidades intermediárias ?* Para calcular o input I_j na unidade j de uma certa camada intermediária, utilizamos a seguinte fórmula $I_j = (\Sigma_i w_{ij} O_i) + \theta_j$, onde O_i é o output de cada unidade i da camada precedente, w_{ij} é o peso da conexão entre a unidade i da camada precedente e a camada j atual, e θ_j é um parâmetro próprio da unidade j, chamado *tendência*. Trata-se de um parâmetro de ajuste ligado à unidade. Ele serve para variar a atividade da unidade.

   Depois de calculado este I_j, aplicamos uma função f, chamada *função de ativação* que tem como tarefa normalizar os valores de I_j de modo a caírem num intervalo entre 0 e 1. O resultado deste cálculo é o output O_j da unidade j. Normalmente esta função deve ser não-linear e diferenciável. Uma função que é frequentemente utilizada é $f(x) = \frac{1}{1+e^{-x}}$. A Figura 5.12 esquematiza o cálculo do input I_j na unidade j em função dos outputs das unidades da camada precedente e o cálculo do output O_j na unidade j, aplicando a função de ativação do neurônio j em seu input I_j.

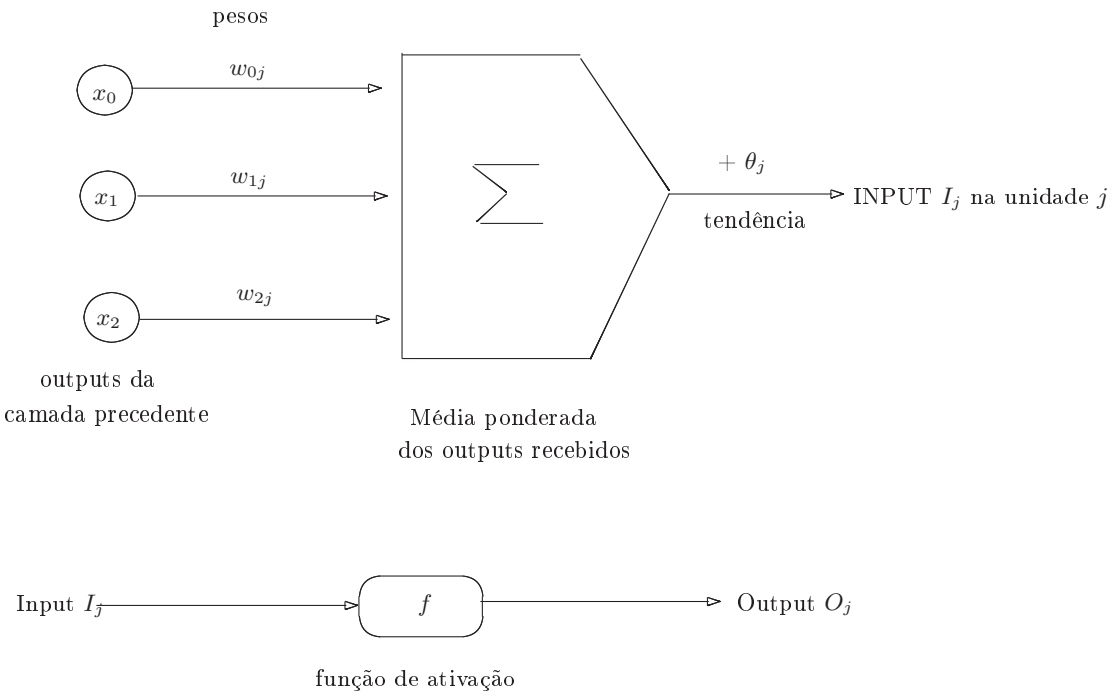


Figura 5.12: Cálculos do input e do output de uma unidade j

2. *Fase de volta: Como são recalculados os pesos das conexões e as tendências de cada unidade na volta para a camada de input.* Quando, no final da fase de ida, atinge-se a camada de output, é feito um teste para ver qual a diferença entre o output calculado em cada unidade e o valor correspondente à classe do input que está associado a esta unidade (1 ou 0). Se todas as diferenças calculadas em cada unidade de output são muito grandes (ultrapassam um certo limite fornecido), inicia-se um processo de volta (*Backpropagation*) que vai recalcular todos os pesos e tendências em função destas diferenças que foram encontradas nas camadas de output. Isto é, a rede vai tentar "ajustar" seus pesos e tendências de modo que estes novos parâmetros reflitam o seu aprendizado sobre a amostra que acabou de ser avaliada. Como é feito este ajuste ?

(a) Primeiramente são calculados os erros em cada unidade: Para a unidade j da camada de output é computado o erro $E_j = O_j(1 - O_j)(T_j - O_j)$, onde O_j = valor do output da unidade j, T_j = valor correspondente à classe da amostra X, associado à unidade j. Este valor é 1 ou 0. Repare que $O_j(1 - O_j)$ = derivada da função f em termos do *input* da unidade j.

Para as unidades das camadas intermediárias: para cada unidade j de uma camada intermediária, o erro E_j é calculado como $E_j = O_j(1-O_j)(\Sigma_k E_k w_{jk})$, onde w_{jk} = peso da conexão ligando a unidade j à unidade k da camada seguinte, e E_k é o erro que já foi calculado nesta unidade k.

(b) Em seguida, são calculados os novos pesos de cada conexão (unidade $i \rightarrow$ unidade j) Novo w_{ij} = Velho $w_{ij} + \Delta w_{ij}$, onde $\Delta w_{ij} = \lambda E_j O_i$. A constante λ é um parâmetro do algoritmo e é chamada de *taxa de aprendizado*. É um número entre 0 e 1. Este parâmetro ajuda a evitar que o processo fique "parado" num mínimo local, isto é, num ponto onde os pesos parecem convergir (a diferença entre os pesos é muito pequena) mas que na verdade se trata de um mínimo local e não global.

(c) Em seguida são calculados as novas tendências em cada unidade: Nova θ_j = Velha $\theta_j + \Delta\theta_j$, onde $\Delta\theta_j = \lambda E_j$.

A crítica maior que se faz com relação e esta técnica é a sua interpretabilidade, isto é, não é óbvio para um usuário interpretar o resultado do algoritmo, a saber, uma rede neural treinada como sendo um instrumento de classificação (as regras de classificação correspondentes a uma rede treinada não são óbvias como acontece com as árvores de decisão). Além disto, é difícil para os humanos interpretar o significado simbólico que está por trás dos pesos das conexões da rede treinada. As vantagens das redes neurais, incluem sua alta tolerância a dados contendo ruídos assim como a grande confiança dos resultados. Atualmente diversos algoritmos já foram desenvolvidos para se extrair regras de classificação de uma rede treinada, o que minimiza de certa forma a desvantagem do método mencionada acima.

5.3.3 Método K-Means para Análise de Clusters

Análise de Clusters é o processo de agrupar um conjunto de objetos físicos ou abstratos em classes de objetos similares. Um *cluster* é uma coleção de objetos que são similares uns aos outros (de acordo com algum critério de similaridade pré-fixado) e dissimilares a objetos pertencentes a outros clusters. Análise de Clusters é uma tarefa de *aprendizado não-supervisionado*, pelo fato de que os clusters representam classes que não estão definidas no início do processo de aprendizagem, como é o caso das tarefas de Classificação (*aprendizado Supervisionado*), onde o banco de dados de treinamento é composto de tuplas classificadas. Clusterização constitui uma tarefa de aprendizado *por observação* ao contrário da tarefa de Classificação que é um aprendizado *por exemplo*.

O método k-means para Análise de Clusters, recebe como input um banco de dados de objetos (tuplas) e um número k (representando o número de clusters que se deseja formar entre os objetos do banco de dados). O banco de dados é dado em forma de uma *matriz de dissimilaridade* entre os objetos. Nesta matriz o elemento da coluna j e linha i da matriz é o número $d(i, j)$ representando a distância entre os objetos i e j. Várias funções são utilizadas para medir a distância. Algumas delas são: distância Euclidiana,

distância de Manhattan, distância de Minkowski. O leitor encontrará no final desta seção referências sobre este assunto. O quadro abaixo ilustra uma matriz de dissimilaridade:

$$\begin{vmatrix} 0 & \dots & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d(n,1) & d(n,2) & d(n,3) & \dots & 0 \end{vmatrix}$$

Existem técnicas para construir tal matriz de dissimilaridade, dependendo do tipo de dados presentes no banco de dados.
Seja $\mathcal{C} = \{C_1, \dots, C_k\}$ uma partição do banco de dados em k clusters e sejam m_1, m_2, $\dots, m_k$ elementos escolhidos em cada um dos clusters, representando o centro dos mesmos (Estes elementos podem ser definidos de diversas maneiras). Definimos o *erro quadrático* da partição como sendo $\text{Erro}(\mathcal{C}) = \Sigma_{i=1}^{k} \Sigma_{p \in C_i} \mid p - m_i \mid^2$.

O método k-means que apresentamos a seguir procura construir uma partição $\mathcal{C}$ contendo k clusters, para a qual o erro quadrático é mínimo. Os elementos representativos de cada cluster são os seus centros de gravidade respectivos. A idéia geral do método é a seguinte: (1) Escolhe-se arbitrariamente k objetos $\{p_1, ..., p_k\}$ do banco de dados. Estes objetos serão os centros de k clusters, cada cluster C_i formado somente pelo objeto p_i.

(2) Os outros objetos do banco de dados são colocados nos clusters da seguinte maneira: para cada objeto O diferente de cada um dos p_i's, considera-se a distância entre O e cada um dos p_i's (isto é dado pela matriz de dissimilaridade). Considera-se aquele p_i para o qual esta distância é mínima. O objeto O passa a integrar o cluster representado por p_i.

(3) Após este processo, calcula-se a média dos elementos de cada cluster, isto é, o seu centro de gravidade. Este ponto será o novo representante do cluster.

(4)Em seguida, volta para o passo 2: varre-se o banco de dados inteiro e para cada objeto O calcula-se a distância entre este objeto O e os novos centros dos clusters. O objeto O será realocado para o cluster C tal que a distância entre O e o centro de C é a menor possível.

(5) Quando todos os objetos forem devidamente realocados entre os clusters, calcula-se os novos centros dos clusters.

(6) O processo se repete até que nenhuma mudança ocorra, isto é, os clusters se estabilizam (nenhum objeto é realocado para outro cluster distinto do cluster atual onde ele se encontra).

A Figura 5.13 ilustra o funcionamento do método k-means para $k = 3$. Na primeira iteração, os objetos circundados representam os que foram escolhidos aleatoriamente. Nas próximas iterações, os centros de gravidade são marcados com o sinal $+$.

É possível mostrar que o método k-means produz um conjunto de clusters que minimiza o erro quadrático com relação aos centros de gravidade de cada cluster. Este método só produz bons resultados quando os clusters são "núvens" compactas de dados, bem separadas umas das outras. As vantagens do método são sua eficiência em tratar grandes conjuntos de dados. Suas desvantagens são o fato do usuário ter que fornecer o número de clusters k, o fato de não descobrir clusters de formatos não-convexos e sobretudo o fato de ser sensível a ruídos, já que objetos com valores altos podem causar uma grande

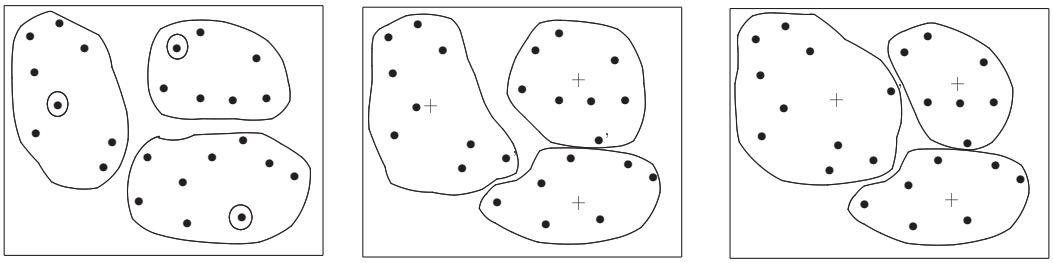


Figura 5.13: O método k-means

alteração no centro de gravidade dos clusters e assim, distorcer a distribuição dos dados nos mesmos.

5.3.4 Método CURE para Análise de Clusters

Os métodos de clusterização podem ser divididos, grosso modo, em métodos por particionamento e métodos hierárquicos. Um método de particionamento é o k-means que vimos na subseção precedente. Outros são indicados nas referências no final da seção. Os algoritmos de clusterização hierárquicos aglomerativos seguem a seguinte idéia geral: (1) Começam com um número de clusters igual ao tamanho do banco de dados: um cluster para cada objeto. (2) Pares de clusters são aglutinados a cada iteração até que um número k (dado) de clusters seja alcançado, ou que a distância mínima entre os clusters seja menor do que um limite fornecido (avaliar este parâmetro não é tarefa fácil). Geralmente, utiliza-se o critério de parada correspondente a atingir o número k de clusters desejado. (3) O critério para se aglutinar dois clusters é a distância entre eles: são aglutinados, a cada iteração, os dois clusters que estão mais próximos. Diferentes funções são utilizadas para medir a distância entre dois clusters C_i, C_j:

- $d_{mean}(C_i, C_j) =\mid m_i - m_j \mid$, onde m_i é o centro de gravidade do cluster C_i. Chamamos este enfoque de *enfoque baseado em centróide.*
- $d_{min}(C_i, C_j) = min \mid p - q \mid$, para todos $p \in C_i$ e $q \in C_j$. Chamamos este enfoque de *MST* (*Minimum Spanning Tree*).
- $d_{ave}(C_i, C_j) = \frac{1}{n_i * n_j} \Sigma_{p \in C_i} \Sigma_{q \in C_j} \mid p - q \mid$, onde $n_i =$ tamanho do cluster C_i.
- $d_{max}(C_i, C_j) = max \mid p - q \mid$, para todos $p \in C_i$ e $q \in C_j$.

O algoritmo CURE O algoritmo CURE recebe como **input** um banco de dados D e um número k que é o número de clusters que se deseja obter (assim como os métodos de particionamento, o k deve ser fornecido pelo usuário). CURE produz como resultado k clusters. Além disto, outros parâmetros de ajuste deverão ser fornecidos ao algoritmo para a sua execução. Tais parâmetros serão descritos durante a apresentação do algoritmo.

O algoritmo CURE é um algoritmo hierárquico aglomerativo que utiliza uma política mista para o cálculo da distância entre dois clusters, a cada iteração. Esta política é uma espécie de mistura entre a *política dos centróides* (onde a distância entre dois clusters é a distância entre seus centros de gravidade) e a chamada *políticca MST (Minimum Spanning Tree)* (onde a distância entre dois clusters é igual à distância mínima entre dois

pontos, um em cada cluster). CURE vai considerar um número adequado de representantes de cada cluster, elementos que estão adequadamente distribuídos no cluster e que representam zonas bem distintas dentro do mesmo). A distância entre dois clusters será min ($d(c_i, c'_j)$ onde todos os representantes do primeiro cluster ($\{c_1, ..., c_n\}$) e do segundo cluster ($\{c'_1, ..., c'_m\}$) são considerados. A Figura 5.14 ilustra as duas políticas clássicas nos métodos hierárquicos para cálculo de distância entre dois clusters (para efeito de aglutinamento dos mesmos) e a política adotada por CURE.

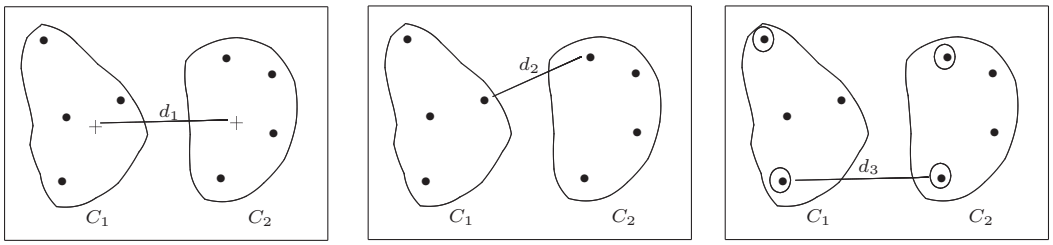


Figura 5.14: Políticas de cálculo de distância nos métodos hierárquicos

As características principais de CURE são:(1)Clusters de formas arbitrárias podem ser detectados por CURE (a exemplo dos métodos baseados em densidade). Isto normalmente não acontece com os métodos por particionamento. (2) CURE é robusto quanto à presença de ruídos. (3) Uma desvantagem de CURE é a sua complexidade O(n^2), onde n é o tamanho do banco de dados de input.

Idéia geral. Como todo algoritmo hierárquico aglomerativo, CURE inicia com cada objeto do banco de dados constituindo um cluster. Logo, temos n clusters no início, onde n = tamanho do banco de dados.

A fim de calcular a distância entre dois clusters, são armazenados c representantes de cada cluster. Este número c é um parâmetro de ajuste que deve ser fornecido como input ao algoritmo. Estes c pontos são escolhidos de forma a representar regiões bem distintas em cada cluster. Depois de escolhidos, é feita uma *retração* destes pontos na direção do centro do cluster, de um fator α, onde $0 \leq \alpha \leq 1$. Este número α é um dos parâmetros de ajuste que devem ser fornecidos como input. A distância entre dois clusters será a distância entre os pares de representantes mais próximos, um em cada cluster. Assim, somente os representantes são levados em consideração para o cálculo da distância entre dois clusters. Na Figura 5.15, os pontos dentro de círculos são os representantes antes da retração. Após uma retração de um fator $\alpha = 1/2$ na direção do centro +, os pontos obtidos são aqueles dentro de □.

Estes c representantes tentam capturar o formato do cluster. A retração em direção ao centro de gravidade do cluster tem o efeito de diminuir a influência dos ruídos. A razão para isto é que os outliers estarão longe do centro do cluster. Caso um outlier seja escolhido como representante do cluster, após a retração ele se aproximará do centro bem mais do que os outros representantes que não são outliers.

O parâmetro α também serve para capturar o formato do cluster. Valores de α pequenos favorecem clusters de formato menos compacto, não-convexo. Valores grandes para α (próximos de 1), aproximando os representantes do centro do cluster, favorecem a criação de clusters mais compactos, convexos, de forma esférica.

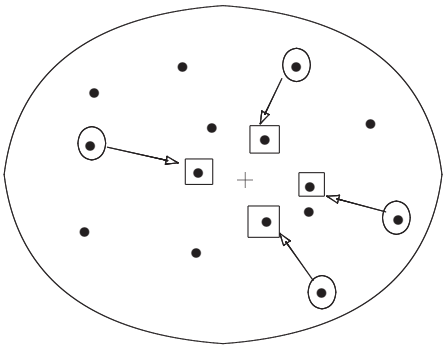


Figura 5.15: Como se formam os clusters no algoritmo CURE

Após calculada a distância entre os clusters, aglutina-se aqueles dois primeiros que estão mais próximos. Este processo de aglutinamento envolve a cálculo dos novos representantes para o cluster aglutinado.

Depois disso, volta-se para o passo anterior, calculando-se as distâncias entre os novos clusters e aglutinando-se aqueles que estão mais próximos. Este processo pára quando se atinge o número k de clusters desejado (o k é parâmetro do algoritmo).

Notas Bibliográficas. O algoritmo ID3 foi criado por Ross Quinlan, da Universidade de Sydney em 1986. O algoritmo C4.5 úm de seus sucessores, também proposto por Quinlan em 1993. Uma boa referência para estes algoritmos é [WF 2000]. O algoritmo de Backpropagation foi apresentado pela primeira vez em [RHW 1986]. Muitos livros sobre Aprendizado de Máquina fornecem uma boa explicação deste algoritmo. Por exemplo [WK 1991]. Um algoritmo para extração de regras de classificação de uma rede neural treinada pode ser encontrada em [LSL 1995]. O artigo [Roy 2000] traz uma interessante discussão sobre o aprendizado do cérebro humano e o aprendizado artificial de uma rede neural. Em [Set 1997], temos um algoritmo para poda de redes neurais. O algoritmo CURE para análise de clusters foi introduzido em [GRS 1998] em 1998. Outros algoritmos de clusterização podem ser encontrados em [ZRL 1996](BIRCH), [EKSX 1996](DBSCAN) e [NH 1994](k-medóides).

5.4 Técnicas de Otimização e Implementação

5.4.1 Algumas Técnicas de Otimização de Apriori

Diversas técnicas de otimização foram propostas e colocadas em prática para melhorar a performance do algoritmo Apriori. As técnicas atuam da seguinte maneira:
(1) *Categoria 1*: Diminuem o número de candidatos que são testados para cada transação (para ver quais são suportados por cada transação) na fase do cálculo do suporte. Também atuam na fase de poda, diminuindo o número de subconjuntos a serem testados para cada pré-candidato (lembre-se que um pré-candidato de tamanho k só é considerado bom se todos os seus subconjuntos de tamanho $k-1$ aparecem no L_{k-1}).

Categoria 2: Diminuem o tamanho do banco de dados a cada iteração.

Categoria 3: Diminuem o número de varridas do banco de dados. No algoritmo Apriori clássico, o banco de dados é varrido a cada iteração.

Categoria 4: Diminuem o número de candidatos gerados.

Utilizando uma árvore-hash para armazenar C_k e L_{k-1}. Uma *árvore-hash* é uma árvore onde as folhas armazenam conjuntos de itemsets, e os nós intermediários (inclusive

a raiz) armazenam tabelas-hash contendo pares do tipo (número, ponteiro), como mostra a Figura 5.16.

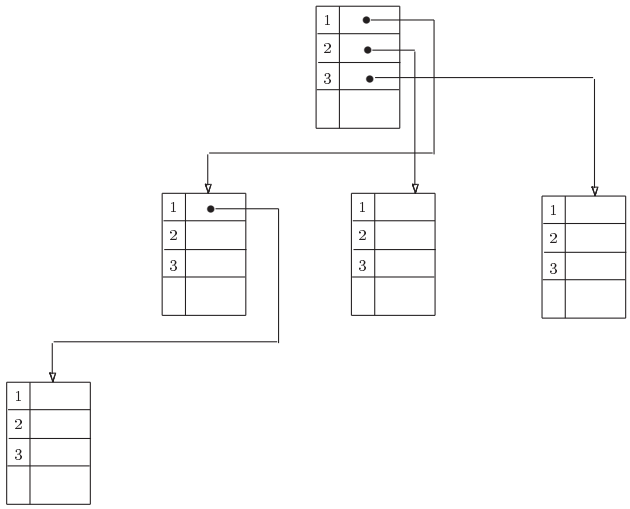


Figura 5.16: Árvore-hash

Uma árvore-hash é utilizada para estocar itemsets. Por exemplo, suponhamos que queiramos estocar o seguinte conjunto de 2-itemsets numa árvore-hash:

Pão	Leite
Pão	Açúcar
Pão	Manteiga
Leite	Açúcar
Leite	Manteiga
Açúcar	Manteiga

Vamos utilizar a enumeração dos itens descrita na Figura 5.2 e vamos ordenar os itemsets segundo a ordem lexográfica associada a esta enumeração. Suponhamos que o número máximo de itemsets numa folha é M = 3 e que cada nó tem no máximo N = 2 descendentes. Para a estocagem deste conjunto de itemsets numa árvore-hash respeitando estes parâmetros, utilizamos uma função hash h que servirá para distribuir os itemsets nas folhas da árvore. Esta função é definida como se segue: h(Pão) = 1, h(Leite) = 2, h(Açúcar) = 1, h(Manteiga) = 2. No início, todos os cinco itemsets estão na raiz, como mostra a Figura 5.17

Como o número máximo de itemsets numa folha é 3, é preciso transformar a raiz num nó intermediário e criar nós descendentes da raiz. A Figura 5.18 ilustra a criação dos dois filhos da raiz.

A primeira folha excede o número máximo de elementos permitido. Logo, deve se transformar num nó intermediário e nós-folha devem ser criados a partir deste primeiro nó. A Figura 5.19 ilustra este processo.

Repare que agora, o número de elementos das folhas não excede 3 e portanto o processo termina.

Como utilizar uma árvore-hash na fase de poda dos candidatos

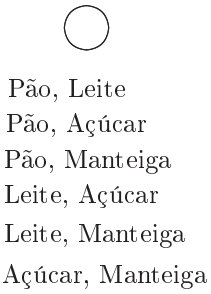


Figura 5.17: Raiz da árvore-hash

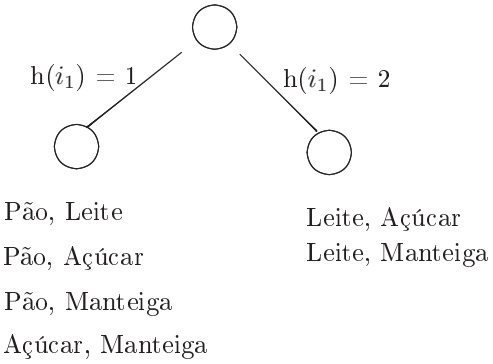


Figura 5.18: Filhos do nó-raiz

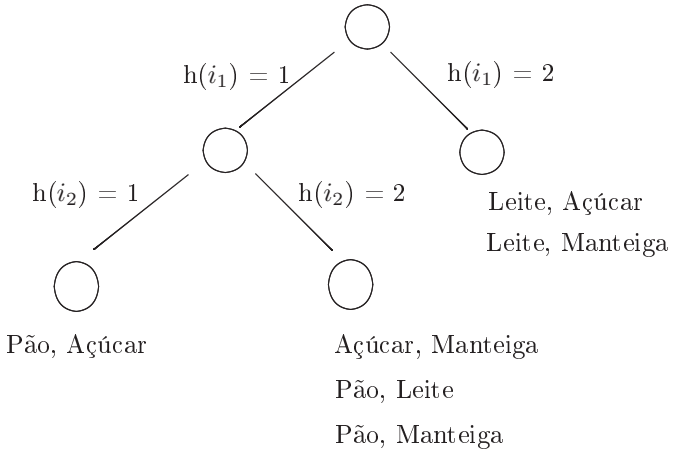


Figura 5.19: Itemsets armazenados numa árvore-hash

Suponhamos que tenhamos gerado os pré-candidatos C'_3 e vamos utilizar L_2 (itemsets frequentes de tamanho 2 calculados na fase anterior) para podar os pré-candidatos e obter C_3. Para isso, armazenamos o conjunto de itemsets L_2 numa árvore-hash, como descrevemos no parágrafo precedente. Para cada elemento de C'_3, temos que verificar se este contém um 2-itemset que não está em L_2. Por exemplo, suponha que I = {Pão, Açúcar, Manteiga} $\in C'_3$. Construimos todos os subconjuntos de 2 elementos de I: {Pão, Açúcar}, {Pão, Manteiga}, {Açúcar, Manteiga}. Suponha que nosso L_2 esteja armazenado na árvore-hash da Figura 5.19. Para cada um dos 3 subconjuntos de I de tamanho 2, vamos verificar se está presente numa das folhas da árvore. O itemset {Pão, Açúcar} é procurado somente na primeira folha, já que h(Pão) = 1 e h(Açúcar) = 1. Assim, evita-se de percorrer elementos de L_2 que decididamente nunca poderiam conter {Pão, Açúcar}. Os itemsets {Pão, Manteiga} e {Açúcar, Manteiga} são procurados somente na segunda

folha. A terceira folha não é varrida, economizando-se tempo com isso.

Como utilizar uma árvore-hash na fase de avaliação do suporte

Nesta fase, armazena-se os candidatos já podados C_k numa árvore-hash. Para cada transação t do banco de dados, vamos aumentar o contador de suporte de cada elemento de C_k que esteja contido em t (isto é, que seja suportado por t). Por exemplo, suponha que C_2 esteja armazenado na árvore-hash da Figura 5.19 e que $t = \{$Pão, Leite, Manteiga$\}$ seja uma transação do banco de dados. Quais candidatos de C_2 são suportados por t ? Para isso, fazemos uma busca recursiva dos subconjuntos de 2 elementos de t: {Pão, Leite}, {Pão, Manteiga}, {Leite, Manteiga}. O primeiro, {Pão, Leite} só pode se encontrar na folha 2 (já que h(Pão) = 1 e h(Leite) = 2). Logo, esta folha é varrida. Se o itemset é encontrado, aumentamos seu contador de 1. Analogamente, o segundo 2-itemset, {Pão, Manteiga} só pode se encontrar na folha 2 e o terceiro, {Leite, Manteiga} só pode se encontrar na folha 3. Veja que a folha 1 não é varrida, economizando-se tempo com isso. A Figura 5.20 ilustra os itemsets que são suportados pela transação t e que, portanto, têm seus contadores aumentados de 1.

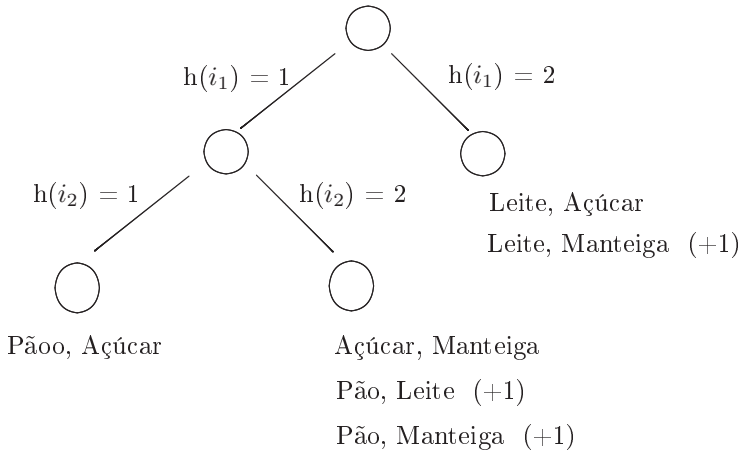


Figura 5.20: Utilização de uma árvore-hash na fase de cálculo do suporte

Notas Bibliográficas. A técnica da árvore-hash foi introduzida em [AS 1994] e adaptada para mineração de sequências [AS 1995, AS 1996]. Técnicas para diminuir o número de varridas no banco de dados podem ser encontradas em [SON 1996]. Técnicas de gerenciamento de buffer também são descritas em [AS 1994, AS 1995, AS 1996].

5.5 Aspectos Teóricos e Generalização

Apesar do interesse da área de Mineração de Dados estar concentrado nas potenciais aplicações das técnicas, os aspectos teóricos envolvidos devem ser investigados no sentido de encontrar os elementos essenciais das diversas técnicas de mineração e desenvolver métodos que avaliem sua complexidade e otimizá-las na medida do possível. Nesta seção, vamos dar algumas noções teóricas subjacentes à técnica de mineração *crescimento-poda-validação* que vimos na Seção 5.2.

5.5.1 Um Algoritmo Geral Baseado numa Hierarquização do Espaço de Busca

A tarefa de descobrir todas os padrões interessantes pode ser descrita de forma geral como segue: Suponha que temos uma linguagem $\mathcal{L}$ de especificação de padrões, e uma *condição de seleção* Q que é utilizada para avaliar se um padrão de $\mathcal{L}$ é ou não interessante. A tarefa em que estamos interessados é encontrar o conjunto S de todos os padrões de $\mathcal{L}$ que satisfazem a condição Q quando testada sobre um conjunto de dados D. O objetivo desta subseção é descrever um método geral para calcular o conjunto S. Veremos que os algoritmos de mineração de regras de associação e de sequências que estudamos na Seção 5.2 se encaixam neste esquema geral. Antes de mais nada, precisamos introduzir uma *relação de especialização* entre os padrões de $\mathcal{L}$. Uma tal relação é simplesmente uma *ordem parcial*[1] $\preceq$ sobre $\mathcal{L}$. Dizemos que φ é *mais geral* do que θ se $\varphi \preceq \theta$; neste caso, também dizemos que θ é *mais específico* do que θ. A relação $\preceq$ é *monotônica* com relação à condição de seleção Q se, sempre que $\varphi \preceq \theta$ e Q satisfaz θ, temos que Q também satisfaz φ. Em outras palavras, se um padrão é interessante então todos os padrões mais gerais do que ele também o são. Denotamos por $\theta \prec \varphi$ se $\theta \preceq \varphi$ e $\varphi \neq \theta$.

A idéia do algoritmo *Level Search* é começar gerando os padrões mais gerais possíveis, e em seguida ir gerando e testando padrões mais e mais específicos, mas *não* testar aqueles padrões que não têm chance de ser interessantes. A informação obtida em iterações anteriores dão indícios suficientes para garantir que certos padrões não podem ser interessantes, sem que se tenha de validá-los no banco de dados.

Algoritmo *Level Search*
Entrada: Um banco de dados D, uma linguagem de padrões $\mathcal{L}$ com uma relação de especialização $\preceq$, e uma condição de seleção Q sobre o banco de dados D.
Saída: O conjunto dos padrões S que satisfazem Q.

Método:

1. $C_1 := \{\varphi \in \mathcal{L} |$ não existe $\theta \in \mathcal{L}$ tal que $\theta \prec \varphi\}$;
2. $i := 1$;
3. *while* $C_i \neq \emptyset$ *begin*
4. // Validação: encontra os padrões de C_1 que satisfazem Q:
5. $F_i := \{\varphi \in C_i \mid \varphi$ satisfaz $Q\}$
6. // geração: calcula $C_{i+1} \subset \mathcal{L}$ usando $\bigcup_{j \leq i} F_j$:
7. $C_{i+1} := \{\varphi \in \mathcal{L} \mid$ para todo $\gamma \prec \varphi$ tem-se que $\gamma \in \bigcup_{j \leq i} F_j\} - \bigcup_{j \leq i} C_j$;
8. $i := i + 1$;
9. *end*
10. retorna $\bigcup_{j \leq i} F_j$;

O algoritmo executa iterativamente, alternando entre as fases de *geração de candidatos* e *validação dos candidatos*. Primeiro, na fase de geração da iteração i, é gerada a coleção C_i de novos padrões candidatos, usando para isso informação proveniente de padrões mais gerais. Em seguida, a condição de seleção é avaliada em cima destes novos padrões candidatos. A coleção F_i vai consistir daqueles padrões de C_i que satisfazem Q. Na próxima iteração $i+1$, padrões candidatos em C_{i+1} são gerados usando a informação sobre padrões

[1] Uma relação de ordem parcial sobre um conjunto V é uma um subconjunto R do produto cartesiano $V \times V$ satisfazendo as propriedades (1) *reflexiva*: para todo $a \in V$, tem-se $(a, a) \in R$; (2) *antissimétrica*: se $(a, b) \in R$ e $(b, a) \in R$ então $a = b$; (3) *transitiva*: se $(a, b) \in R$ e $(b, c) \in R$ então $(a, c) \in R$.

em $\bigcup_{j \leq i} F_j$. O algoritmo começa construindo o conjunto C_1 que contém os padrões mais gerais possíveis. A iteração pára quando nenhum padrão novo potencialmente interessante é encontrado. O algoritmo objetiva minimizar o número de varridas no banco de dados, i.e., o número de testes da condição Q (passo 5). Note que o cálculo para determinar o conjunto de candidatos (passo 7) não envolve o banco de dado.

5.5.2 Exemplos

Nesta subseção, vamos descrever como os algoritmos de mineração de Regras de Associação e Sequências que vimos na Seção 5.2 são na verdade casos especiais do algoritmo *Level Search* que descrevemos na subseção anterior (exemplos 5.5.1 e 5.5.2). No exemplo 5.5.3, mostramos como a técnica *Level Search* pode ser aplicada para desenvolver algoritmos para novas tarefas de mineração.

Exemplo 5.5.1 (Regras de Associação - o algoritmo Apriori) Consideremos a tarefa de *Regras de Associação* que discutimos na Sezção 5.2. Para esta tarefa, vamos identificar os elementos da técnica *Level Search*:

- A linguagem dos padrões $\mathcal{L}$: trata-se aqui do conjunto de todos os *itemsets* sobre o conjunto de items $\mathcal{I}$.
- Relação de Especialização: se I e J são itemsets então dizemos que I é mais específico do que J se $J \subseteq I$. O conjunto dos candidatos C_1 é constituído dos itemsets unitários. São os *itemsets* mais gerais possíveis, segundo esta relação de especialização.
- Propriedade Q: esta propriedade é a condição dada pelo suporte, isto é, I satisfaz Q se e somente se $\sup(I, D) \geq \alpha$.

Exemplo 5.5.2 (Sequências - O algoritmo GSP) Consideremos agora a tarefa de *Análise de Sequências* que vimos na Seção 5.2. Para esta tarefa, vamos identificar os elementos da técnica *Level Search*:

- A linguagem dos padrões $\mathcal{L}$: trata-se aqui do conjunto de todas as *sequências de itemsets* $s =< \{i_1, ..., i_n\}, ..., \{j_1, ..., j_m\} >$ sobre o conjunto de items $\mathcal{I}$.
- Relação de Especialização: se $s =< s_1, ..., s_k >$ e $s' =< s_1, ..., s_m >$ são sequências de itemsets então dizemos que s é mais específica do que s' se existe subsequência $s'' =< s_{i_1}, ..., s_{i_k} >$ tal que $s_{i_j} \subseteq s'_j$ para todo $j = 1, ..., k$. O conjunto dos candidatos C_1 é constituído das sequências do tipo $< \{a\} >$ compostas de um único *itemset* unitário. São as sequências mais gerais possíveis, segundo a relação de especialização acima (dada pela noção de *subsequência*).
- Propriedade Q: esta propriedade é a condição dada pelo suporte, isto é, s satisfaz Q se e somente se $\sup(s, D) \geq \alpha$.

Vamos agora aplicar a técnica *Level Search* a fim de desenvolver algoritmos para novas tarefas de mineração.

Exemplo 5.5.3 (Episódios) Vamos considerar a tarefa de descobrir *episódios* em sequências de eventos. Um *episódio* é uma coleção de eventos que ocorrem durante um intervalo de tempo de tamanho dado. Formalmente: Seja E um conjunto de eventos; um *episódio* é uma tripla $\varphi = (V, \leq, g)$ onde V é um conjunto de vértices, $\leq$ é uma relação de ordem parcial sobre V e $g : V \to E$ é uma aplicação associando a cada vértice de V um evento em E. Intuitivamente, um episódio é um padrão dado pelo grafo $g(V)$, isto é, um conjunto de eventos acontecendo na ordem dada pela relação de ordem parcial $\leq$. A Figura 5.21 (a) ilustra uma sequência de eventos. Esta sequência constitui os dados a serem varridos à procura de *episódios* que se repetem ao longo da sequência com uma certa frequência. A Figura 5.21(b) representa um episódio, contendo dois eventos A e B, mas não especifica nenhuma ordem entre estes eventos. Na Figura 5.21(c) temos um episódio contendo um evento adicional C e onde se exige que C deve ocorrer após A e B.

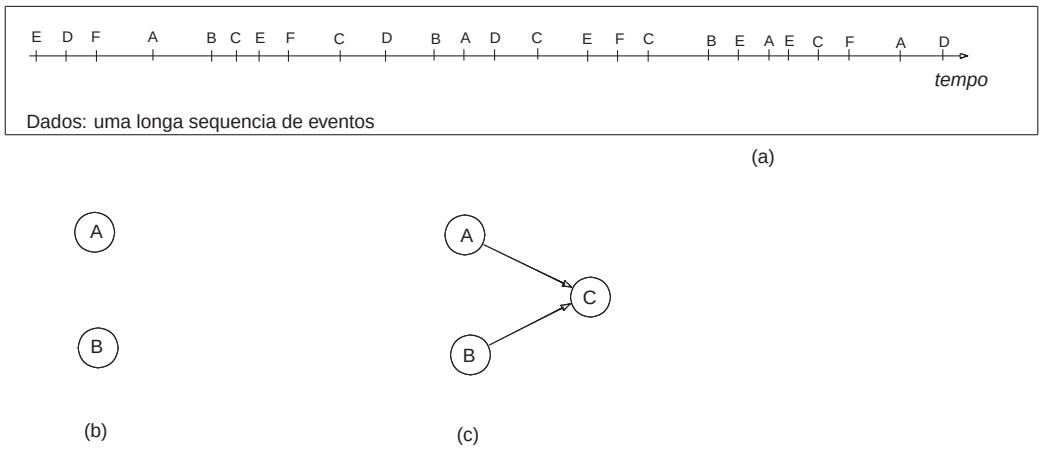


Figura 5.21: (a) Uma sequência de eventos; (b) e (c) dois episódios

A tarefa de mineração consiste em descobrir todos os episódios que aparecem na sequência de eventos dentro de um intervalo de largura w dada, com uma frequência superior a um dado limite mínimo α. A frequência de um episódio é a porcentagem de intervalos de largura w contendo o episódio. O algoritmo *Level Search* pode ser adaptado para minerar episódios. A linguagem dos padrões $\mathcal{L}$ é conjunto de todos os episódios definidos sobre um conjunto de eventos E. A relação de especialização é definida por: se $\varphi = (V, \leq, g)$ e $\theta = (V', \leq', g')$ são dois episódios, dizemos que $\varphi \preceq \theta$ se (1) $V \subseteq V'$, (2) para todo $v \in V$, $g(v) = g'(v)$, e (3) para quaisquer $v, w \in V$ com $v \leq w$ tem-se também $v \leq' w$. Por exemplo, o episódio da Figura 5.21(a) é mais geral (segundo a ordem $\preceq$) do que o episódio da Figura 5.21(b). A propriedade Q é a condição dada pela frequência de um episódio, isto é, um episódio φ satisfaz Q se e somente se sua frequência dentro de intervalos de largura w é superior a um α dados.

Notas Bibliográficas. Ao leitor interessado em ir mais a fundo no material tratado de forma sucinta nesta seção, sugerimos a leitura do artigo [MT 1997]. A técnica de crescimento de padrões descrita aqui, utilizando a relação de especialização e a passagem de um nível a outro do espaço de busca através do cálculo da fronteira negativa do nível precedente foi primeiramente introduzida em [Mit 1982]. Uma outra referência importante que trata desta técnica é [Lan 1996]. Em [MTV 1997] é tratado o problema de mineração de episódios.

5.6 Aplicações de Mineração de Sequências

5.6.1 Web Log Mining

Descobrir padrões de comportamento frequentes na navegação dos usuários de um site não somente pode ajudar a melhorar a arquitetura do site (isto é, fornecer acessos eficientes entre objetos altamente correlacionados) mas também ajudar a tomar decisões apropriadas no que diz respeito à distribuição de material publicitário no site. Os caminhos percorridos pelos usuários são armazenados num arquivo de logs, num servidor especial da organização que gerencia o site em questão. Os dados armazenados são registros da forma:

$$\begin{array}{ccc} u_1 & (s_1, d_1) & t_1 \\ u_2 & (s_2, d_2) & t_2 \\ \vdots & \vdots & \vdots \end{array}$$

onde u_i é um identificador do usuário (seu IP, por exemplo), o par (s_i, d_i) (chamado *refer log*) é constituído de dois endereços: (1) o endereço *fonte* s_i que é o endereço da página anterior visitada pelo usuário, e (2) o endereço *destino* que é o endereço da página atualmente visitada pelo usuário. t_i é o tempo correspondente à operação de passagem de uma página para outra.

Cada vez que um usuário inicia sua visita num site, um registro do tipo $(null, d)$ é inserido no arquivo de logs, onde d é o endereço da página principal (Home) do site. No momento em que o usuário entra num outro site, um novo registro do tipo $(null, d')$ é armazenado, indicando que a sessão de visita do site anterior se encerrou e uma nova sessão está se iniciando no site d'. Consideremos o seguinte exemplo ilustrado na Figura 5.22.

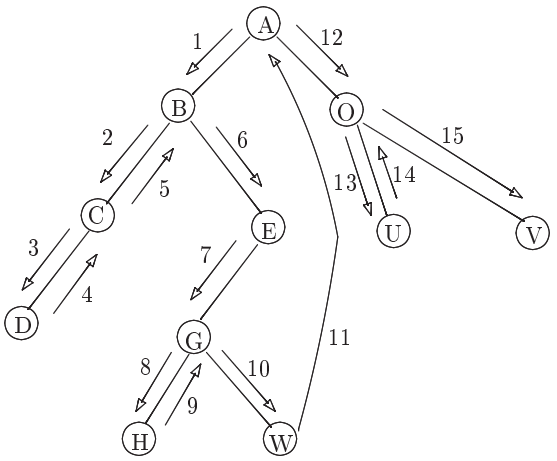


Figura 5.22: Uma sequência de páginas visitadas por um usuário

A Figura 5.22 corresponde à navegação de um usuário através do site cujo endereço principal (home) é A. No arquivo de log, teremos o seguinte caminho correspondente a cada uma das páginas visitadas pelo usuário dentro do site (deste sua entrada no site até sua saída):

$< A, B, C, D, C, B, E, G, H, G, W, A, O, U, O, V >$

É suposto que após a visita da página V, o usuário sai da Internet, ou entra num outro site, iniciando uma nova série de registros de log com o primeiro registro $(null, d')$ correspondendo ao endereço d' do novo site.

Podemos transformar esta sequência de páginas visitadas num conjunto de sequências correspondendo somente às visitas *para frente*, isto é, as voltas para páginas anteriormente visitadas são eliminadas. O resultado da transformação é o seguinte conjunto de sequências:
$\{< A, B, C, D >, < A, B, E, G, H >, < A, B, E, G, W >, < A, O, U >, < A, O, V >\}$

Tais sequências *para frente*, correspondentes a uma única sequência de acessos do usuário é chamada de *sequências maximais para frente*. Assim, o banco de dados de registros de log, pode ser visto como um banco de dados de sequências maximais para frente. Repare que, cada usuário possui diversas sequências maximais para frente (o identificador do usuário não é chave).

Uma *sequência de referências* é uma sequência $s =< s_1, s_2, ..., s_n >$ onde os s_i são símbolos que mapeiam endereços de páginas dentro de um site. Uma sequência de referências $s =< s_1, s_2, ..., s_n >$ é suportada por uma sequência maximal para frente $t =< t_1, t_2, ..., t_m >$ se existe $i = \{1, ..., m\}$ tal que $s_1 = t_i, s_2 = t_{i+1}, ..., s_m = t_{i+n-1} >$. O *suporte* de uma sequência de referências s é dado por:

$$sup(s) = \frac{\text{número de seq. max. para frente que suportam } s}{\text{total de seq. max para frente}}$$

Uma sequência de referências é dita *frequente* se seu suporte é maior ou igual a um nível mínimo de suporte dado.
Um problema de *Web Log Mining* é o seguinte: dado um banco de dados D de sequências maximais para frente e um nível de suporte mínimo α entre 0 e 1, descobrir todas as sequências de referências frequentes com relação a D e a α. Existem algumas diferenças entre este problema e o problema clássico de mineração de sequências:(1) um usuário pode ser contado diversas vezes no cálculo do suporte de uma sequência de referências. Isto não acontece no cálculo do suporte de sequências de itemsets, pois o identificador do usuário é chave do banco de dados de sequências. (2) para que uma sequência s esteja contida numa sequência t é necessário que os símbolos de s ocorram consecutivamente em t. Isto não é exigido na definição de inclusão de duas sequências de itemsets. (3) as sequências de referências não possuem elementos repetidos, o que não acontece em geral para sequências de itemsets: um mesmo itemset pode ser comprado em diferentes momentos pelo mesmo usuário.

5.6.2 Padrões Sequenciais em Bioinformática

Problemas de descoberta de padrões aparecem em diferentes áreas de biologia, por exemplo, padrões que regulam a função de genes em sequências de DNA, ou padrões que são comuns em membros de uma mesma família de proteínas.

O banco de dados de sequências onde os padrões serão descobertos consiste de sequências sobre o alfabeto $\Sigma = \{A, C, G, T\}$ (no caso de sequências de DNA, onde os símbolos A, C, G, T representam os 4 nucleotídeos que compõem uma sequência de DNA) ou sobre o alfabeto dos 20 amino-ácidos que compõem as proteínas (no caso de sequências especificando proteínas). Os seguintes tipos de problemas são objeto de investigação:

Problemas de descobrir padrões significantes. O problema da descoberta de padrões geralmente é formulado da seguinte maneira: define-se uma classe de padrões de interesse

que se quer encontrar e tenta-se descobrir quais os padrões desta classe que aparecem no banco de dados, com um suporte maior. O suporte de um padrão normalmente é o número de sequências nas quais o padrão ocorre. Muitas variações neste conceito ocorrem: pode-se exigir que o padrão ocorra em todas as sequências, ou num número mínimo de sequências especificado pelo usuário. Em alguns casos, o número de ocorrências não é especificado, mas é simplesmente um parâmetro de uma função que relaciona características da sequência (tamanho, por exemplo), com o número de ocorrências.

Descoberta de Padrões versus Reconhecimento de Padrões. O problema discutido acima se refere a Descoberta de Padrões. Um outro problema importante em biologia, relacionado a sequências, é o problema do *Reconhecimento de Padrões*: Em biologia, muitos padrões significantes são conhecidos *a priori* e é importante desenvolver ferramentas para descobrir ocorrências destes padrões conhecidos em novas sequências. Programas para reconhecimento de padrões podem ser bem gerais no sentido de que podem ser projetados para reconhecer padrões diversos (o padrão é um parâmetro de input do problema) ou podem ser bem específicos, no sentido de que são projetados para reconhecer um padrão particular. Do ponto de vista de ciência da computação, estes últimos não apresentam muito interesse, porém, são muito importantes para biólogos.

Problemas de Classificação. Tais problemas ocorrem, por exemplo, em famílias de proteínas. Um dos objetivos de se encontrar motivos comuns em famílias de proteínas é utilizar estes motivos como *classificadores*. Assim, dada uma proteína desconhecida, podemos classificá-la como membro ou não-membro de uma família, simplesmente baseando-se no fato de que ela contém os padrões que caracterizam esta família. Neste caso, temos um típico problema de aprendizado de máquina: dado um conjunto de sequências que pertencem à família (exemplos positivos) e um conjunto de sequências que não pertencem à família (exemplos negativos), queremos descobrir uma função F que associa a cada proteína um elemento de $\{0,1\}$, dizendo que a proteína pertence ($F(proteina) = 1$) ou não pertence ($F(proteina) = 0$) à família.

Notas Bibliográficas. Alguns artigos que constituem leitura interessante para o tópico de *Web Mining* são [KB 2000], [SF 1999], [CMS 1997], [Spi 1999], [SY 2001], [CPY 1998]. Em [Flo 1999], o leitor pode encontrar um excelente *survey* sobre mineração de biosequências bem como algoritmos importantes, como o algoritmo o TEIRESIAS. Em [BDV+ 2000], o leitor vai encontrar endereços de diversos bancos de dados de proteínas e instruções para carregá-los. Uma discussão detalhada dos tipos de problema em mineração de biosequências pode ser encontrada em [BJEG 1998].

Referências Bibliográficas

[Amo 2003] S. de Amo: Curso de Data Mining, Programa de Mestrado em Ciência da Computação, Universidade Federal de Uberlândia, 2003. http://www.deamo.prof.ufu.br/CursoDM.html

[AS 1994] R. Agrawal, R. Srikant : Fast Algorithms for Mining Association Rules. Proc. 20th Int. Conf. Very Large Data Bases, VLDB, 1994.

[AS 1995] R. Agrawal, R. Srikant : Mining Sequential Patterns. In Proc. of 1995 Int. Conf. on Data Engineering, Taipei, Taiwan, March 1995.

[AS 1996] R. Agrawal, R. Srikant : Mining Sequential Patterns : Generalizations and Performance Improvements. Proc. 5th EDBT, 3-17, 1996.

[ASV 1997] R. Srikant, Q. Vu, R. Agrawal: Mining Association Rules with Item Constraints Proc. of the 3rd Int'l Conference on Knowledge Discovery in Databases and Data Mining, Newport Beach, California, August 1997.

[BDV+ 2000] B. Brejova, C. DiMarco, T. Vinar, S.R. Hidalgo, G. Holguin, C. Patten : Finding Patterns in Biological Sequences. Project Report, Department of Biology, University of Waterloo, 2000.

[BJEG 1998] A. Brazma, I. Jonassen, I. Eidhammer, D. Gilbert: Approaches to the Automatic Discovery of Patterns in Biosequences. Journal of Computational Biology 5(2): 277-304 (1998).

[CMS 1997] R.Cooley, B. Mobasher, J. Srivastava : Web Mining : Information and Pattern Discovery on the World Wide Web. In Proceedings of the Nineth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97), pages 558–567, November 1997.

[CPY 1998] M.S. Chen, J. S. Park, P.S. Yu : Efficient Data Mining for Path Traversal Patterns. IEEE Transactions on Knowledge Discovery and Data Engineering 10(2), 209-221, Mars 1998.

[EKSX 1996] M. Ester, H-P Kriegel, J. Sander, X. Xu : A density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, KDD 1996.

[Flo 1999] A. Floratos : Pattern Discovery in Biology: Theory and Applications. Ph.D. Thesis, Department of Computer Science, New York University, Jan. 1999.

[FPPR 1996] Usam M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, Ramasamy Uthurusamy : Advances in Knowledge Discovery and Data Mining. AAAI Press, 1996.

[GRS 1998] S. Guha, R. Rastogi, K. Shim : CURE: An Efficient Clustering Algorithm for Large Databases. ACM/SIGMOD 1998.

[GRS 1999] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: sequential pattern mining with regular expression constraints. Proc. VLDB, 223-234, 1999.

[GRS 2002] M. Garofalakis, R. Rastogi, and K. Shim. Mining Sequential Patterns with Regular Expression Constraints. IEEE Transactions on Knowledge and Data Engineering Vol. 14, No. 3, 2002, pp. 530-552.

[HK 2001] J.Han, M. Kamber : Data Mining : Concepts and Techniques. Morgan Kaufmann, 2001.

[HMS 2001] D. Hand, H. Mannila, P. Smith : Principles of Data Mining. MIT Press, 2001.

[KB 2000] R. Kosala, H. Blockeel : Web Mining Research: a Survey. SIGKDD Explorations. ACM SIGKDD, July 2000, Vol 2, Issue 1.

[Lan 1996] P. Langley : Elements of Machine Learning. San Mateo, CA, Morgan Kaufmann, 1996.

[LSL 1995] H. Lu, R. Setiono, H. Liu : Neurorule: A connectionist approach to data mining. In Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95), pp. 478-489, Zurich, Switzerland, 1995.

[Man 1997] H. Mannila. Methods and problems in data mining. In Proc. 7th International Conference on Database Theory (ICDT 97), pages 41-55m Delphi, Greece, 1997.

[Mit 1982] T.M. Mitchell: Generalization as search. Artificial Intelligence, 18:203-226.

[MT 1997] H. Mannila, H. Toivonen : Levelwise Search and Borders of Theories in Knowledge Discovery. Data Mining and Knowledge Discovery 1, 241-258, 1997. Kluwer Academic Publishers.

[MTV 1997] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery, 1(3), 259-289, 1997.

[NH 1994] R.T.Ng, J. Han : Efficient and Effective Clustering Methods for Spatial Data Mining. Proc. of the International Conference on Very Large Data Bases, (VLDB Conference), 1994.

[RHW 1986] D.E.Rumelhart, G.E. Hinton, R.J.Williams: Learning internal representations by error propagation. In D.E. Rumelhart and J.L.McClelland, editors, *Parallel Distributed Processing*. Cambridge, MA,MIT Press, 1986.

[Roy 2000] Asin Roy: Artificial Neural Networks: A Science in Trouble. SIGKDD Explorations, Vol. 1, Issue 2, Jan. 2000, 33-38.

[Set 1997] R. Setiono : A penalty-function approach for pruning feedforward neural networks. Neural Computation, Vol. 9, n. 1, 185-204, 1997.

[SF 1999] M. Spiliopoulou, L.C. Faulstich : WUM : A Web Utilization Miner. In Proceedings of EDBT Workshop WebDB98, Valencia, LNCS 1590, Springer Verlag, 1999.

[SON 1996] A. Savasere, E. Omiecinski, S. B. Navathe: An Efficient Algorithm for Mining Association Rules in Large Databases. Proc. 1995 Int. Conf. on Very Large Databases (VLDB 95),432-444.

[Spi 1999] M. Spiliopoulou : The Labourious Way From Data Mining to Web Log Mining. International Journal of Computer Systems Science and Engineering: Special Issue on Semantics of the Web, 14:113–126, March 1999.

[SY 2001] R. Srikant, Y. Yang : Mining Web Logs to Improve Website Organization. Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001, pages 430-437.

[WF 2000] I. H. Witten, E. Frank : Data Mining : Practical Machine Learning - Tools and Techniques with Java Implementations. Morgan Kaufmann, 2000.

[WK 1991] S.M.Weiss, C.A. Kulikowski: Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning and Expert Systems. San Mateo, CA, Morgan Kaufmann, 1991.

[ZRL 1996] T. Zhang, R. Ramakrishnan, M. Livny. BIRCH: An efficient data clustering method for very large databases. In Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96), 103-114, Montreal, Canada, 1996.