UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMO NARFO PARA MINERAÇÃO DE REGRAS DE ASSOCIAÇÃO GENERALIZADAS NÃO REDUNDANTES BASEADA EM UMA ONTOLOGIA DIFUSA

Rafael Garcia Miani

SÃO CARLOS
Abril/2009

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMO NARFO PARA MINERAÇÃO DE REGRAS DE ASSOCIAÇÃO GENERALIZADAS NÃO REDUNDANTES BASEADA EM UMA ONTOLOGIA DIFUSA

Rafael Garcia Miani

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Mauro Biajiz

SÃO CARLOS
2009

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

M618an

Miani, Rafael Garcia.
Algoritmo narfo para mineração de regras de associação generalizadas não redundantes baseada em uma ontologia difusa / Rafael Garcia Miani. -- São Carlos : UFSCar, 2009.
226 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2009.

1. Banco de dados. 2. Data mining (Mineração de dados). 3. Regras de associação. 4. Lógica difusa. I. Título.

CDD: 005.74 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Algoritmo Narfo para Mineração de Regras de Associação Generalizadas não Redundantes Baseada em uma Ontologia Difusa"

RAFAEL GARCIA MIANI

Dissertação de **Mestrado** apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Profa. Dra. **Marilde** Teresinha Prado Santos
(DC/UFSCar)

Profa. Dra. **Marina** Teresa Pires **Vieira**
(UNIMEP)

Prof. Dra. **Sandra** Aparecida de Amo
(UFU)

São Carlos
Maio/2008

DEDICATÓRIA

Para minha mãe Lucineide, meu pai Ângelo, minha namorada Thais, todos meus amigos e familiares que sempre me apoiaram em todos os momentos durante esse trabalho de mestrado.

AGRADECIMENTOS

A Deus, que me concedeu saúde e paz necessária para a conclusão deste trabalho.

Aos professores Dr. Mauro Biajiz, Dra Marilde T. P. Santos e Dr. Ricardo Rodrigues Ciferri, pela ajuda, conselhos e dedicação que tiveram.

A todos os membros do Grupo de Banco de Dados (GBD - UFSCar), pela presença, auxílio e companhia em todos os momentos e reuniões do grupo.

À CAPES pelo auxílio fornecido para o desenvolvimento deste trabalho.

RESUMO

Abordagens tradicionais para mineração de regras de associação generalizadas são somente baseadas no conteúdo do banco de dados, tendo um maior foco em combinações exatas entre os itens. No entanto, em muitas aplicações, o uso de um conhecimento de apoio, como ontologias, pode aprimorar o processo de descoberta de conhecimento e gerar regras de associação semanticamente mais ricas. Desse modo, neste trabalho de mestrado foi desenvolvido o algoritmo NARFO, um novo algoritmo para a mineração de regras de associação generalizadas não redundantes baseada em uma ontologia difusa. A ontologia difusa é utilizada como um conhecimento prévio de apoio, para dar suporte ao processo de descoberta e geração das regras. Uma importante contribuição desse algoritmo é a implementação da generalização de *itemsets* não frequentes durante o processamento do algoritmo que auxilia na obtenção de regras significantes. O algoritmo NARFO também contribui na etapa de pós-processamento, com seu tratamento de generalização e redundância.

ABSTRACT

Traditional approaches for mining generalized association rules are based only on database contents, and focus on exact matches among items. However, in many applications, the use of some background knowledge, as ontologies, can enhance the discovery process and generate semantically richer rules. In this way, this paper proposes the NARFO algorithm, a new algorithm for mining non-redundant and generalized association rules based on fuzzy ontologies. Fuzzy ontology is used as background knowledge, to support the discovery process and the generation of rules. One contribution of this work is the generalization of non-frequent itemsets that helps to extract meaningful knowledge. NARFO algorithm also contributes at post-processing stage with its generalization and redundancy treatment.

LISTA DE ABREVIATURAS

NARFO – Non-redundant and generalized Association Rule based on Fuzzy Ontologies

MD – Mineração de Dados

KDD – Knowledge Discovery in Databases

DHP – Direct Hashing and Pruning

DLG – Direct Large itemset Generation

OWL – Web Ontology Language

FARM – Fuzzy Association Rule Mining

GAR – Generalized Association Rule

FGAR – Fuzzy Generalized Association Rule

SSDM – Semantically Similar Data Mining

XSSDM – eXtended Semantically Similar Data Mining

RDF – Resource Description Framework

ER – Entidade e Relacionamento

IBGE – Instituto Brasileiro de Geografia e Estatística

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

SUMÁRIO

1. INTRODUÇÃO

1.1. MOTIVAÇÃO

A mineração de dados é uma etapa importante no processo de descoberta de conhecimento em banco de dados (CHEN; WEI; KERRE, 2000). Um importante tópico nas pesquisas de mineração de dados está focado na descoberta de regras de associação interessantes (FARZANYAR; KANGAVARI; HASHEMI, 2006). Muitos trabalhos na mineração de regras de associação são motivados por encontrar novas maneiras de trabalhar com diferentes tipos de atributos ou para aumentar a performance computacional. Ao mesmo tempo, um crescente número de pesquisas foi desenvolvido considerando a semântica dos dados minerados, com o objetivo de melhorar a qualidade do conhecimento obtido. Nessa linha de pesquisa, ontologias estão sendo muito utilizadas para representar informação semântica definida por especialistas de domínio, e também pode ser aplicada para aprimorar o processo de mineração de regras de associação. (SRIKANT; AGRAWAL, 1995) e (HOU, *et al*, 2005) estenderam o processo de mineração de regras de associação para obter regras que representem relações entre itens de dados básicos, assim como entre itens em qualquer nível de uma taxonomia ou ontologia, resultando nas regras de associação generalizadas.

O uso de ontologias tradicionais baseadas na lógica proposicional, a qual considera a discriminação do raciocínio exato em *verdadeiro* ou *falso*, é um ponto comum em abordagens para regras de associação generalizadas. Tal restrição se torna inapropriada para representar alguns conceitos e relacionamentos do mundo real. Por exemplo, é difícil a representação em ontologias tradicionais de conceitos como *novo*, *velho*, *alto* ou *baixo*. Esses conceitos são melhores representados por relacionamentos difusos como o relacionamento de similaridade (ZADEH, 1987b), que possui um grau para representar o quanto os conceitos são similares entre si. Assim, a combinação de ontologias com lógica difusa, baseada na teoria de conjuntos difusos (ZADEH, 1987a), é apropriada para representar informações imprecisas em domínios específicos.

Portanto, alguns trabalhos (CHEN, *et al*, 2003) (ESCOVAR; YAGUINUMA; BIAJIZ, 2006) têm usado ontologias difusas com o objetivo de extrair regras de associação

semanticamente mais ricas. Entretanto, o processo de extração de regras de associação, adotando tanto ontologias tradicionais quanto ontologias difusas na mineração de regras de associação generalizadas, normalmente traz aos usuários uma grande quantidade de regras que representam a mesma informação. Por conseguinte, é importante evitar a mineração de regras desnecessárias que expressam conhecimento redundante e, ao mesmo tempo, manter o foco na riqueza semântica provida por ontologias difusas.

1.2. OBJETIVO

Considerando o contexto inserido na seção 1.1, o objetivo deste trabalho de mestrado é a realização do processo de mineração de regras de associação generalizadas baseada em uma ontologia difusa, de modo que as regras obtidas pelo algoritmo criado para realizar a mineração (algoritmo NARFO) não tragam nenhuma informação redundante. Para isso, alguns métodos foram utilizados, como a generalização de *itemsets* não frequentes durante o processo de mineração de regras de associação, o tratamento de generalização das regras a partir das regras obtidas e, por fim, a verificação e eliminação de possíveis regras redundantes geradas.

1.3. ESTRUTURA DA DISSERTAÇÃO

Esta dissertação de mestrado está organizada do seguinte modo:

- Capítulo 2: apresenta conceitos básicos de mineração de dados, além de um estudo sobre mineração de regras de associação e a descrição do algoritmo clássico Apriori para minerar regras de associação. Também são apresentados conceitos de regras de associação generalizadas e regras redundantes;
- Capítulo 3: relata conceitos da lógica difusa utilizados neste trabalho, tais como teorias de conjuntos difusos e relações difusas;
- Capítulo 4: descreve conceitos e definições básicas de ontologias, os quais serão úteis para o desenvolvimento deste trabalho;
- Capítulo 5: esse capítulo apresenta alguns trabalhos envolvendo lógica

difusa na mineração de dados, ontologias na mineração de dados e ontologias difusas na mineração de dados;

- Capítulo 6: descreve detalhadamente o algoritmo NARFO, criado neste trabalho para realizar a mineração de regras de associação generalizadas não redundantes baseada em uma ontologia difusa;
- Capítulo 7: traz os experimentos realizados para validar o algoritmo NARFO;
- Capítulo 8: apresenta a conclusão da dissertação e propõe alguns trabalhos futuros.

2. MINERAÇÃO DE DADOS

2.1. INTRODUÇÃO

Mineração de Dados (MD) se refere à mineração ou descoberta de novas informações em função de padrões ou regras em grandes quantidades de dados (ELMASRI; NAVATHE, 2005). MD surgiu no final da década de 1980 quando profissionais de grandes empresas e organizações começaram a se preocupar com a grande quantidade de dados que estavam armazenados nos seus bancos de dados. Tem como objetivo a extração de um novo conhecimento obtido a partir de grandes volumes de dados utilizando computador, como pode ser observado através da figura 2.1. Temos na figura uma grande quantidade de dados (representando o banco de dados), uma menina e um carrinho contendo ouro. O ouro em questão representa conhecimento útil obtido a partir do processo de Mineração de Dados sobre a grande massa de dados e a menina representa a ferramenta utilizada para obter esse conhecimento. Seu crescimento a partir dos anos 90 se deu em virtude da necessidade que as empresas tinham de transformar essa grande massa de dados em informação útil e em conhecimento.

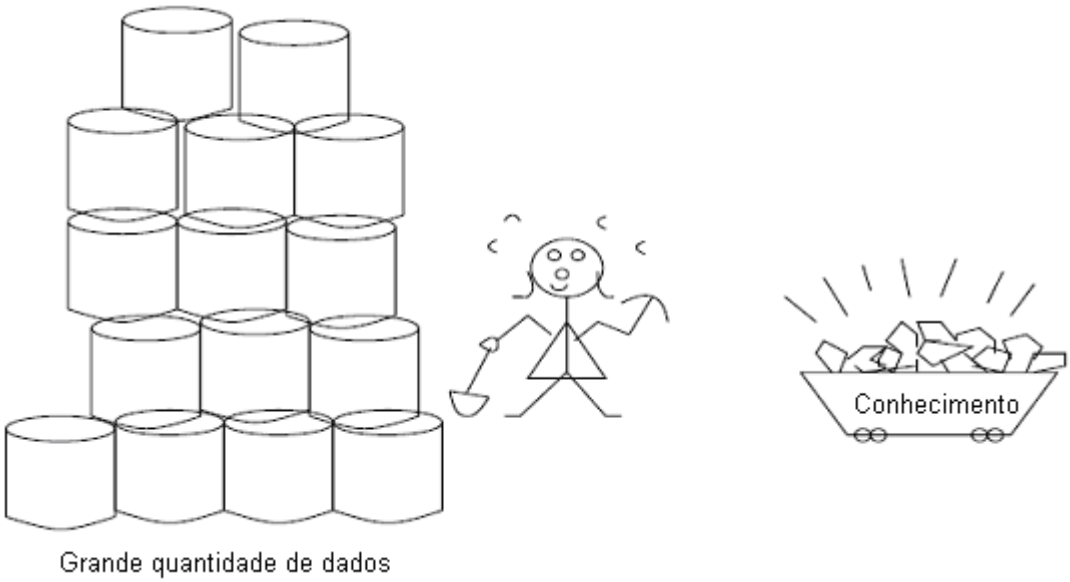


Figura 2.1 - Representação de mineração de dados.

Mineração de Dados está diretamente relacionada com uma área conhecida como Knowledge Discovery in Databases (KDD), ou Descoberta de Conhecimento em Banco de Dados. Para (FAYYAD; PIATETSKY-SHAPIRO; SMYITH, 1996), o KDD se refere a todo o processo de descoberta de conhecimento útil a partir dos dados. Além disso, eles consideram a MD como uma etapa do processo de descoberta de conhecimento em banco de dados. Em geral, o KDD consiste em fazer com que dados em baixo nível tornem-se conhecimento em alto nível. A figura 2.2 mostra a Mineração de Dados como uma etapa do KDD, segundo (HAN; KAMBER, 2006), consistindo de uma sequência iterativa de passos:

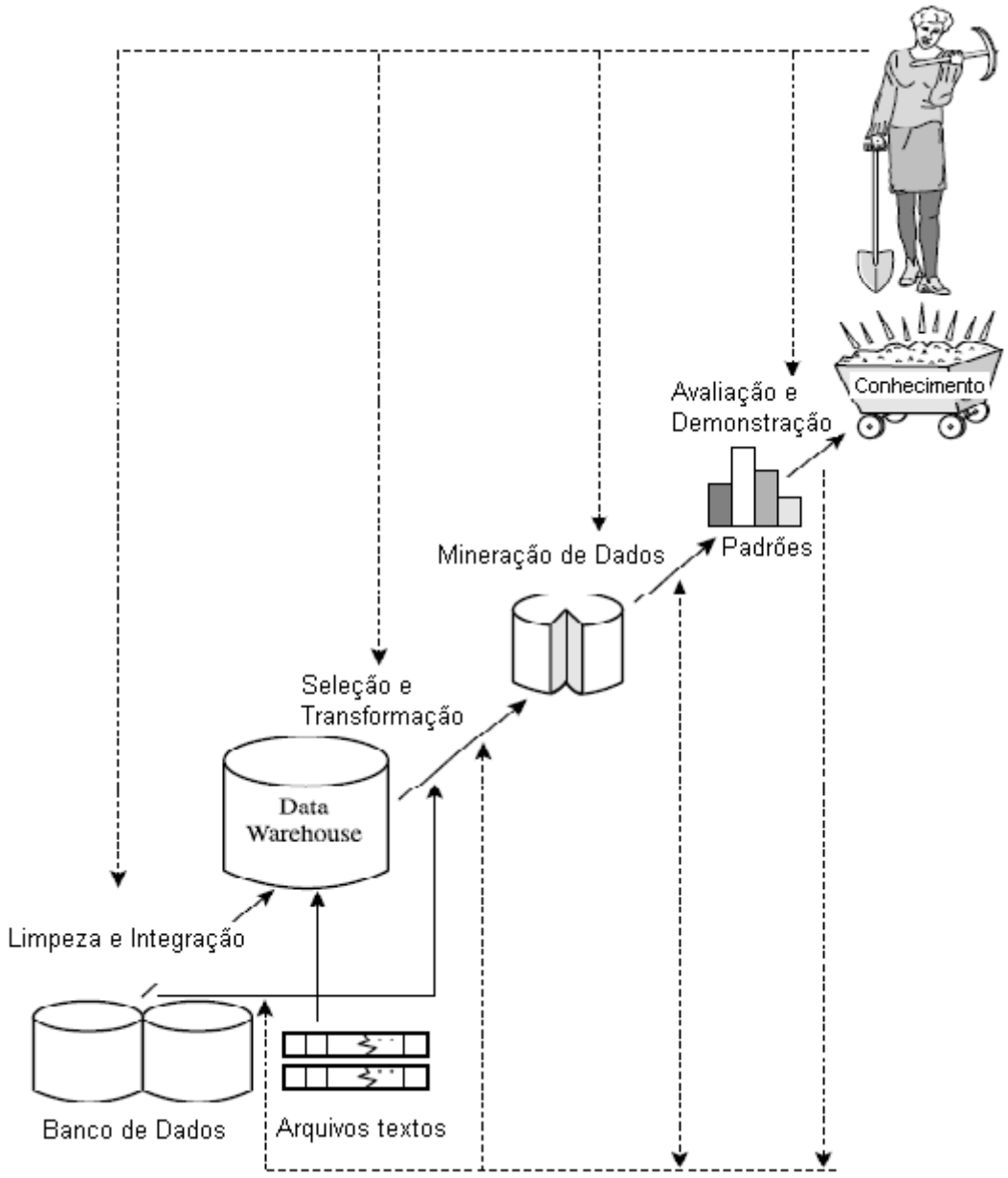


Figura 2.2 - Mineração de Dados como uma etapa do processo de KDD (adaptada de (HAN; KAMBER, 2006)).

1. **Limpeza dos dados** (para remover ruídos e dados inconsistentes).
2. **Integração dos dados** (onde múltiplas fontes de dados podem ser combinadas).
3. **Seleção dos dados** (onde dados relevantes para a tarefa de análise são recuperados do banco de dados).
4. **Transformação dos dados** (onde os dados são transformados ou consolidados em formas apropriadas para a mineração, fazendo um sumário ou agregando operações).

5. **Mineração de Dados** (processo essencial onde métodos inteligentes são aplicados com o objetivo de extrair padrões de dados).
6. **Avaliação dos Dados** (para identificar os verdadeiros padrões de interesse que representam conhecimento baseado em alguma medida de interesse).
7. **Apresentação do conhecimento** (onde visualização e técnicas de representação de conhecimento são utilizadas para apresentar o conhecimento minerado para o usuário).

Os passos de 1 a 4 são diferentes formas de pré-processamento de dados, no qual os dados são preparados para a mineração. A etapa de mineração deve interagir com o usuário ou a base de conhecimento. Os padrões de interesse são apresentados ao usuário e devem ser armazenados como um novo conhecimento na base de conhecimento.

O KDD tem como objetivo a descoberta de um novo padrão ou conhecimento útil. Esse conhecimento depende dos dados a serem analisados e do tipo de informação que se pretende obter. Para obter tal conhecimento, o usuário define o tipo de tarefa de MD a ser utilizado, de acordo com o tipo de conhecimento que se deseja obter, logo no início do problema. Uma tarefa representa a especificação *do que* estamos querendo buscar ou descobrir nos dados. As tarefas podem ser classificadas em duas categorias: *descritivas* e *preditivas* (HAN; KAMBER, 2006). Tarefas descritivas caracterizam as propriedades gerais dos dados no banco de dados, enquanto tarefas preditivas fazem uma inferência a partir dos dados presentes para fazer predições sobre dados futuros. Existem várias tarefas de Mineração de Dados. As tarefas mais conhecidas são: *associação, agrupamento, classificação e regressão*. As tarefas de classificação e regressão são consideradas preditivas, e as tarefas de associação e agrupamento são descritivas. Essa divisão é mostrada na figura 2.3. A tarefa de associação consiste na descoberta de regras de associação (seção 2.2). De acordo com (HIPP; GUNTZER; NAKHAEIZADEH, 2000), a tarefa de minerar regras de associação tem recebido muita atenção e minerar tais regras é um dos métodos mais populares para a descoberta de padrões.

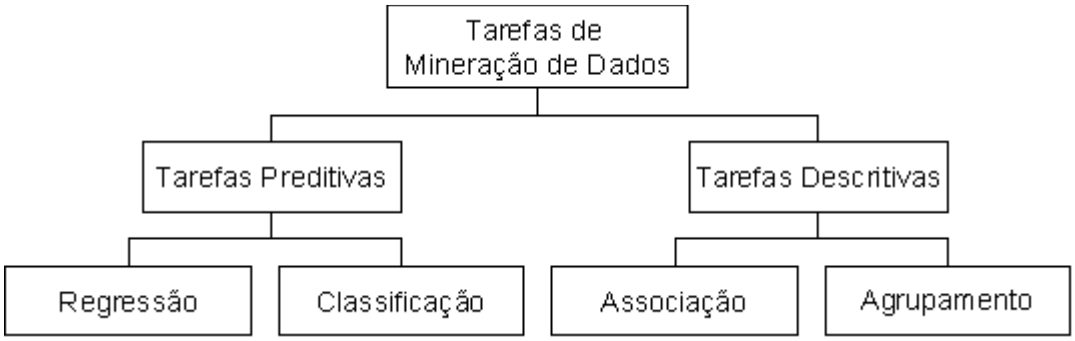


Figura 2.3 - Hierarquia das Tarefas de Mineração de Dados.

2.2. REGRAS DE ASSOCIAÇÃO

A tarefa de associação consiste em encontrar regras de associação. A Mineração de Dados envolvendo regras de associação foi inicialmente proposta em (AGRAWAL; IMIELINSKI; SWAMI, 1993). Considere I = (i1, i2,…, in) o conjunto de itens de um banco de dados D. Uma regra de associação é uma implicação $X \rightarrow Y$, onde $X \subseteq I$, $Y \subseteq I$, e $X \cap Y = \varnothing$. X representa o lado do antecedente da regra, enquanto Y representa o lado consequente. Cada regra possui uma medida de suporte e confiança. Suporte de uma regra reflete a porcentagem de transações em D que contém ambos X e Y, podendo ser calculado através da equação 2.1 (na equação, ocorrências($X \cup Y$) é o número de transações que X e Y ocorrem juntas na base de dados, e T é o total de transações). A medida confiança é a porcentagem de transações contendo X que também contém Y, sendo calculada de acordo com a equação 2.2. Uma regra de associação é considerada forte se seus respectivos graus de suporte e confiança forem maiores ou iguais às medidas *minsup* (suporte mínimo desejado) e *minconf* (confiança mínima desejada).

$$\text{suporte } (X \rightarrow Y) = \frac{\text{ocorrências}(X \cup Y)}{T}$$

Equação 2.1 - Cálculo do suporte de uma regra de associação.

$$\text{confiança } (X \rightarrow Y) = \frac{\text{suporte}(X \cup Y)}{\text{suporte}(X)}$$

Equação 2.2 – Cálculo da confiança de uma regra de associação.

Um exemplo de regra de associação seria a relação *pão⇒ presunto*, *queijo*, com 3% de grau de suporte e com um grau de confiança de 80%. Isso significa que em 3% de

todas as transações da base de dados, os itens *pão, presunto e queijo* apareceram na mesma transação. Já a confiança de 80% significa que em 80% das vezes que o item *pão* apareceu em uma transação, também ocorreu a presença de presunto e queijo. Uma regra é considerada forte se o seu grau de confiança e o seu grau de suporte são maiores ou iguais às medidas *minsup* e *minconf*, respectivamente. Na figura 2.4 temos uma ilustração de uma situação em que as regras de associação podem ser aplicadas. Desse modo, as regras de associação caracterizam o quanto um determinado conjunto de itens de uma base de dados pode implicar na presença de algum outro item ou conjunto de itens nos mesmos registros.

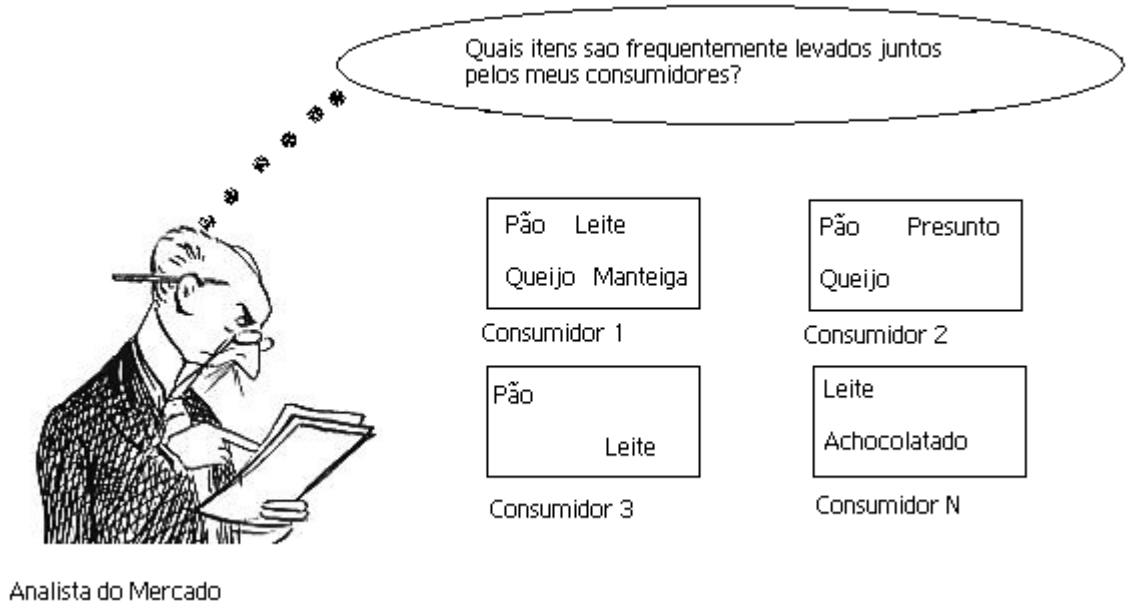


Figura 2.4 - Exemplo de uma aplicação de regra de associação.

Um *itemset* é um conjunto de itens. O suporte de um *itemset* é a porcentagem de transações que contém todos os itens do *itemset*.

Resumindo, a tarefa de regras de associação consiste de duas etapas:

1. Encontrar todos os *itemsets* frequentes.
2. Gerar regras de associação fortes a partir dos *itemsets* frequentes.

Existem diversos algoritmos para realizar a mineração de regras de associação. (AGRAWAL; SRIKANT, 1994) foram quem inicialmente implementaram um algoritmo para a mineração de regras de associação, denominado Apriori. A partir dele, vários algoritmos

foram desenvolvidos, como o DHP (PARK; CHEN; YU, 1995), o DLG (YEN; CHEN, 1996) e o Max-Miner (BAYARDO, 1998), que não serão detalhados neste trabalho. O algoritmo Apriori será descrito em seguida.

Apriori é o algoritmo mais aplicado na mineração de conjuntos de itens frequentes para Regras de Associação (HAN; KAMBER, 2006). Sua criação representou um grande diferencial em relação aos algoritmos anteriores a ele, principalmente no que se refere ao desempenho e à estratégia de solução do problema de minerar regras de associação (WOJCIECHOWSKI; ZAKRZEWICZ, 2002). Desse modo, o algoritmo Apriori é considerado um algoritmo clássico, e a partir dele outros algoritmos foram criados, resultando na denominada "família Apriori" (ORLANDO; PALMERINI; PEREGO, 2001).

O objetivo do algoritmo Apriori é identificar os conjuntos de *itemsets* frequentes e construir regras a partir desse conjunto. As regras que tiverem suporte e confiança maior ou igual que *minsup* e *minconf*, respectivamente, são consideradas relevantes. Esse algoritmo trabalha com a ideia de que se um conjunto não é frequente, ou seja, se um conjunto não possui seu grau de suporte maior ou igual ao suporte mínimo desejado, todos seus super conjuntos são descartados. Isso resulta em ganho de tempo.

O Apriori emprega um processo iterativo, onde k-itemsets, itemsets de tamanho k, são usados para encontrar (k+1)-itemsets. Primeiramente, o conjunto de frequência 1-itemset é encontrado. Então, esse conjunto é utilizado para encontrar os conjuntos 2-itemsets. O conjunto 1-itemset é denominado L_1 e 2-itemset, L_2. L_2 é usado para encontrar L_3. Esse processo é repetido até que k-itemsets frequentes não possam ser encontrados. A procura por cada L_k exige uma varredura completa pelo banco de dados. Deve-se encontrar um L_k que satisfaça o suporte mínimo desejado. Assim, tem-se um conjunto de itemsets freqüentes que serão utilizados para gerar regras de associação. As regras que possuírem os graus de suporte e confiança maior ou igual ao suporte e confiança mínima desejada, respectivamente, serão consideradas regras válidas.

O algoritmo Apriori é apresentado na figura 2.5.

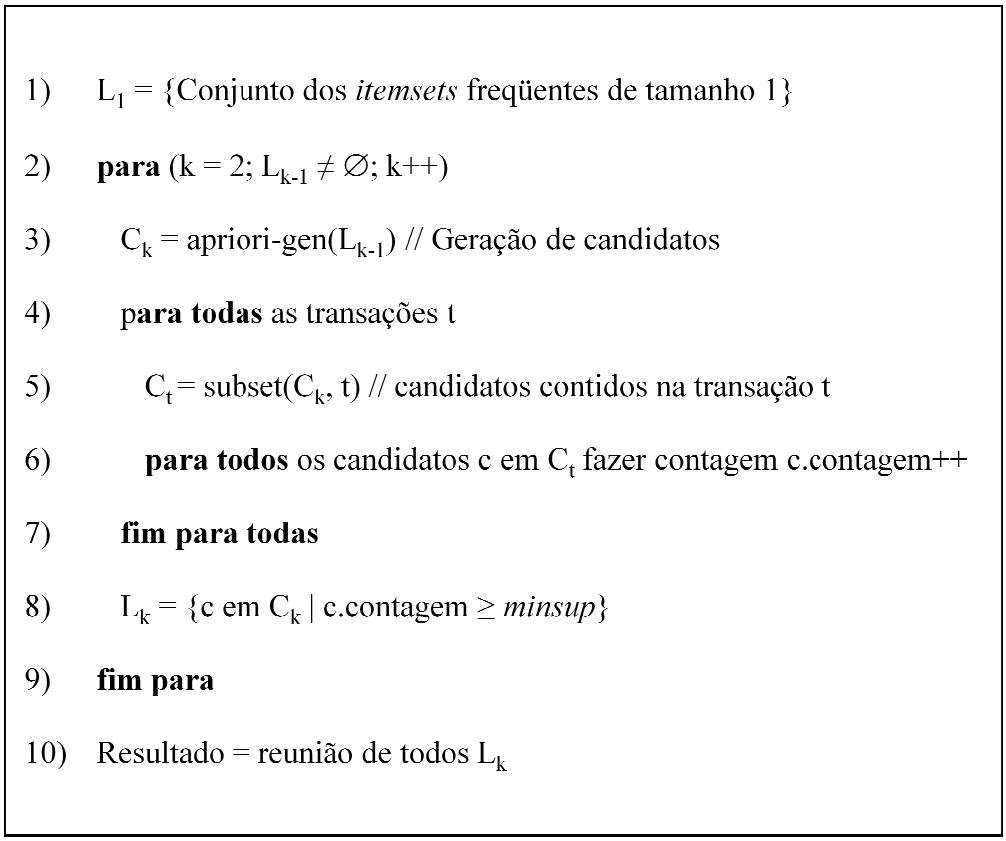


Figura 2.5 - Algoritmo Apriori.

Podemos dividir esse algoritmo em duas etapas:

1. Geração de *itemsets* candidatos: busca por itens frequentes e os identifica.
2. Geração das regras de associação: uso dos *itemsets* frequentes para gerar as regras desejadas.

Na geração dos candidatos, a função apriori-gen (linha 3 do algoritmo da figura 5) recebe como parâmetro L_{k-1} (conjunto de todos os *itemsets* frequentes de tamanho k - 1) e retorna todos os *itemsets* frequentes de tamanho k. Essa função pode ser dividida em dois passos: *Join* (Junção) e *Prune* (Poda).

Esses dois passos são mostrados nos algoritmos das figuras 2.6 e 2.7. No passo *Join* são realizadas combinações através da junção de L_{k-1} com L_{k-1} conforme é mostrado na figura 2.6.

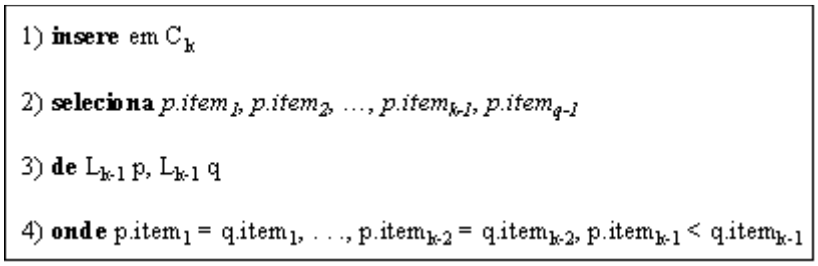


Figura 2.6 - Passo *Join* – adaptado de (AGRAWAL,SRIKANT, 1994).

No passo *Prune* (figura 2.7) são eliminados os *itemsets* candidatos que possuem subconjuntos não frequentes. Ou seja, é feito um ajuste no qual todos os itens $c \in C_k$ são removidos tal que qualquer subitem de c de tamanho k – 1 não esteja em L_{k-1}.

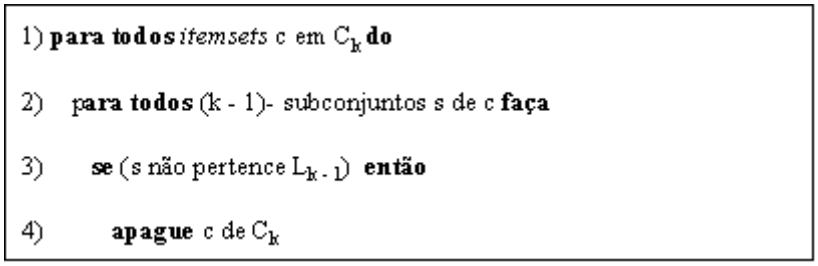


Figura 2.7 - Passo *Prune* – adaptado de (AGRAWAL,SRIKANT, 1994).

Depois do passo *Prune*, temos todos os itens frequentes (L_k). A partir deles, tem-se início à segunda etapa do algoritmo, que é a geração de regras de associação.

Considerando um *itemset I* frequente do conjunto de *itemsets* frequentes, uma regra de associação pode ser apresentada como $(I - a) \rightarrow a$, onde $a \subseteq I$. Para essa regra ser considerada válida, a medida de confiança da mesma tem que ser maior ou igual à confiança mínima desejada. A fórmula do cálculo da confiança é mostrada na figura 2.8.

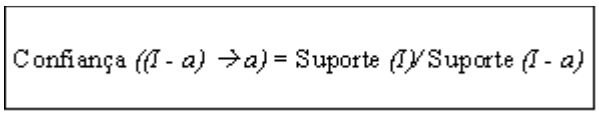


Figura 2.8 - Cálculo da confiança de uma regra.

Assim, todas as regras de associação obtidas a partir do conjunto de *itemsets* frequentes e que possuem grau de confiança maior ou igual a *minconf* são consideradas válidas e relevantes para o problema em questão.

Para uma melhor compreensão do algoritmo, vamos exemplificar. A figura a seguir representa um banco de dados de transações D.

TID	Lista de itens
T100	I1, I2, I3
T200	I2, I3, I4
T300	I3, I4
T400	I1, I2, I3, I4

Figura 2.9 - Exemplo de um banco de dados de transações D.

Na figura 2.10 é ilustrada a etapa de geração de *itemsets* candidatos do algoritmo Apriori, para procurar por itemsets frequentes a partir das transações do banco de dados da figura 2.9.

Na primeira iteração do algoritmo, cada item é membro de um conjunto de 1-itemsets candidatos, C_1. O algoritmo varre todas as transações para contar o número de ocorrência de cada item. Considere que o *minsup* (suporte mínimo desejado) requerido nas transações seja 2, ou seja, a soma de determinado item tem que ser no mínimo 2. Assim, L_1 pode ser obtida, resultando no conjunto de 1-itemsets frequente.

Para determinar L_2, o algoritmo faz uso da junção L_1 com ele mesmo para gerar o conjunto de candidatos 2-itemsets, C_2. Depois disso, L_2 é obtido, sendo composto pelos 2-itemsets candidatos de C_2 que possuem o *minsup*, ou seja, pelos *itemsets* candidatos que são frequentes.

O conjunto de 3-itemsets candidatos (C_3) é gerado pela junção de L_2 com ele mesmo. Na figura 2.11 esse processo é detalhado. L_3 é gerado a partir dos 3-itemsets candidatos de C_3 que satisfazem o *minsup*.

C_4 não possui nenhum *itemset* candidato. Assim, nenhum *itemset* frequente pôde ser obtido e o algoritmo termina seu processo.

A partir desse ponto, temos o conjunto de itemsets frequentes que serão utilizados na etapa de geração de regras de associação. Para cada itemset frequente, o algoritmo gera todas as combinações possíveis de *antecedente* → *consequente*. As combinações que tiverem grau de confiança maior ou igual a *minconf* serão consideradas regras fortes. Assim, todas as regras são geradas. Note que, como todos os *itemsets* utilizados para a geração de regras são frequentes, todas as regras satisfazem a medida *minsup* automaticamente. Regras que possuem suporte maior ou igual a *minsup* e confiança maior ou

igual a *minconf* são consideradas regras válidas e serão exibidas no final do algoritmo com o valor de suporte e confiança de cada uma dessas regras.

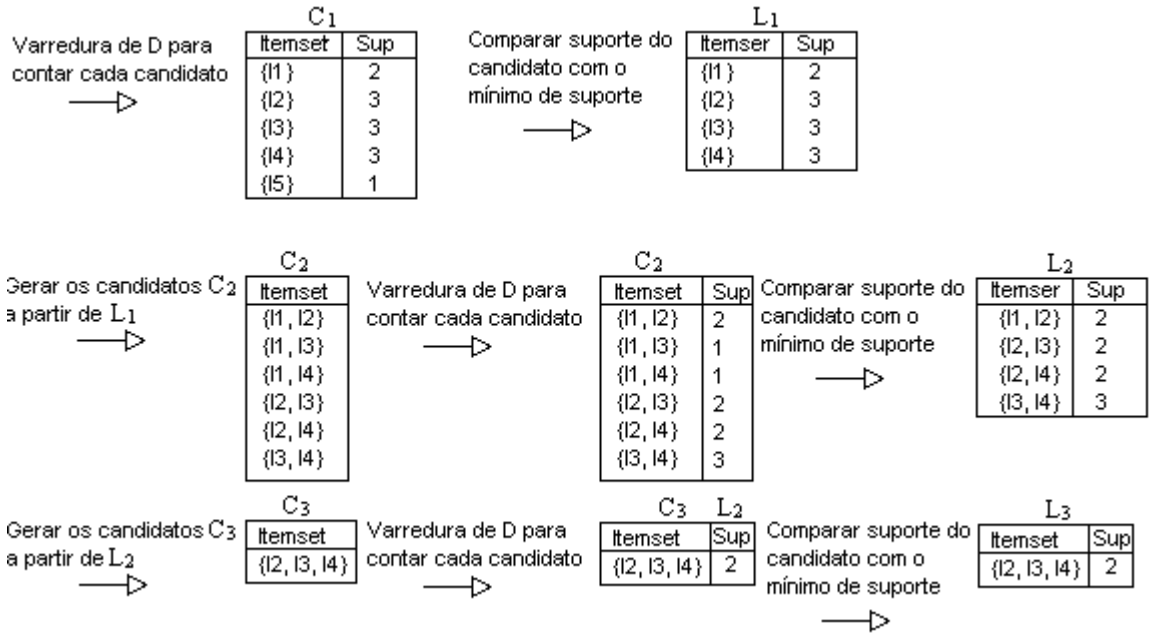


Figura 2.10 - Exemplo da etapa de Geração de Candidatos do algoritmo Apriori.

Considerando o *itemset* {I2, I3, I4} do conjunto de *itemsets* frequentes, poderemos obter regras conforme a figura 2.11.

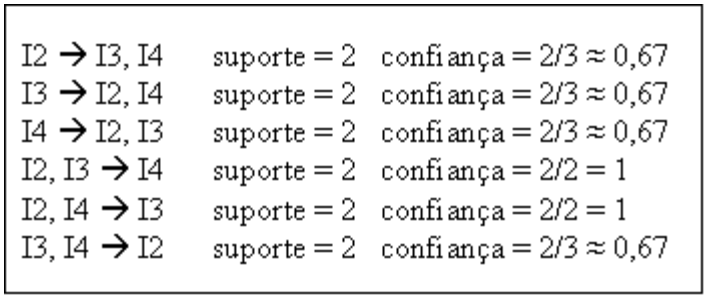


Figura 2.11 - Exemplo de geração de regras de associação.

O suporte de cada regra é obtido a partir do suporte do *itemset* em questão. Para o exemplo acima, todas as regras foram geradas a partir do mesmo *itemset* tendo, por isso, o mesmo suporte. Já a confiança foi obtida a partir da equação mostrada na figura 2.8.

Desse modo, o algoritmo é finalizado, mostrando ao usuário as regras resultantes com seus respectivos graus de suporte e confiança.

2.3. REGRAS DE ASSOCIAÇÃO GENERALIZADAS

Seja I = (i1, i2,…, in) o conjunto de itens de um banco de dados D e *T* uma taxonomia (ou hierarquia "é um"). Uma aresta em *T* representa um relacionamento "é um". Se existe uma aresta em *T* de *p* para *c*, dizemos que *p* é pai de *c* e *c* é filho de *p* (*p* representa a generalização de *c*). X' é um item *ancestral* de x (analogamente, x é *descendente* de x') se existe uma aresta de x' para x em *T*.

De acordo com (SRIKANT; AGRAWAL, 1995) uma regra de associação generalizada é uma implicação da forma $X \rightarrow Y$, onde $X \subset I$, $Y \subset I$, $X \cap Y = \varnothing$, e nenhum item em Y é *ancestral* de nenhum item em X. O motivo para nenhum item em Y ser *ancestral* de nenhum item em X é que a regra da forma "x → *ancestral*(x)" é trivialmente verdadeira com grau de confiança de 100%. O suporte de uma regra $X \rightarrow Y$ é a porcentagem de transações que possuem ambos X e Y. A confiança de $X \rightarrow Y$ é a porcentagem de transações que suportam X e também suportam Y. Uma regra é considerada válida se seu grau de suporte e confiança for maior ou igual ao suporte mínimo desejado (*minsup*) e à confiança mínima desejada, respectivamente. A regra é considerada generalizada porque ambos X e Y podem conter itens em qualquer nível da taxonomia.

Um exemplo de taxonomia é mostrado na figura 2.12. Para essa taxonomia, podemos obter regras de associação generalizadas como "*agasalhos* → *botas*" ou "*jaquetas* → *calçados*".

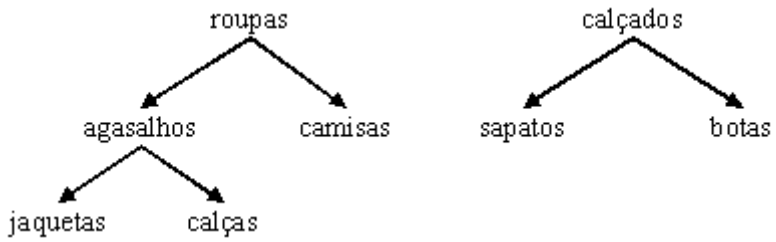


Figura 2.12 - Exemplo de uma taxonomia.

2.4. REGRAS DE ASSOCIAÇÃO REDUNDANTES

Um problema relevante a ser considerado na mineração de regras de associação

é a questão da quantidade de regras de associação que são geradas. Quanto maior o número de regras de associação produzidas, maior a probabilidade de ocorrerem a produção de regras de associação redundantes. Regras de associação redundantes são regras geradas que possuem informações com mesmo significado ou muito similar à outra regra produzida pelo algoritmo. Considerando a taxonomia da figura 2.12, as regras *jaquetas* → *sapatos* e *calças* → *sapatos* seriam redundantes caso a regra de associação generalizada *agasalhos* → *calçados* também fosse gerada.

Para resolver esse problema, vários algoritmos foram produzidos com a aplicação de diferentes métodos. Em (HAN; FU, 1999) foi proposto um método de mineração de regras de associação generalizadas em múltiplos níveis, com o objetivo de tentar reduzir o número de regras de associação generalizadas, assim, diminuindo a redundância. Esse método consiste em definir diferentes valores de suporte mínimo para cada nível da taxonomia. Níveis mais altos possuem suportes maiores.

(SRIPHAEW; THEERAMUNKONG, 2004) e (KUNKLE; ZHANG; COOPERMAN, 2008) desenvolveram seus trabalhos com o objetivo de reduzir o número de regras de associação generalizadas e regras redundantes durante a etapa de extração de padrões. (SRIPHAEW; THEERAMUNKONG, 2004) criaram o algoritmo cSET, o qual utiliza o conceito de *itemsets* fechados (PASQUIER, *et al.*, 1999). Segundo (HAN; KAMBER, 2006), um *itemset* X é fechado em um banco de dados D, se não existir nenhum super *itemset*[1] Y tal que Y tenha o mesmo suporte que X.

(KUNKLE; ZHANG; COOPERMAN, 2008) implementaram o algoritmo MFGI_class baseado na teoria de *itemsets frequentes máximos* (BAYARDO, 1998). Um *itemset* frequente X é dito máximo, se não existe nenhum super itemset Y tal que $X \subset Y$ e Y seja frequente. Considere a taxonomia da figura 2.13(a) e o banco de dados da figura 2.13(b). Os *itemsets* D e E tem suporte igual a 1/6. Supondo que um *minsup* seja 1/3, eles não são frequentes. Porém, o *itemset* generalizado (Z) possui suporte de 1/3, sendo assim frequente. Ele é um *itemset frequente máximo* uma vez que não tem nenhum super itemset frequente (D e E não são frequentes).

[1]Y é um super *itemset* de X se $X \subset Y$. Ou seja, todo item de X está em Y mas pelo menos um item de Y não está em X.

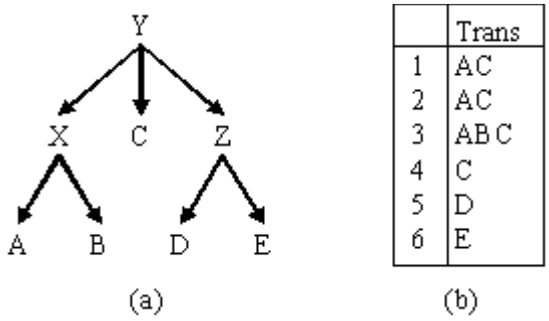


Figura 2.13 - Taxonomia e transações no banco de dados. (a) Taxonomia. (b) Banco de dados transacional (adaptada de (KUNKLE; ZHANG; COOPERMAN, 2008)).

(CHEN, et al., 2003) e (OLIVEIRA; REZENDE; CASTRO, 2007) são trabalhos que tratam o problema na etapa de pós-processamento. O algoritmo proposto por (CHEN, et al., 2003) faz o processo de generalização baseado em uma medida denominada *R-interest*. Essa medida elimina regras redundantes, considerando somente as regras que possuem grau de suporte e confiança R vezes maior que *minsup* e *minconf*. Desse modo, a generalização ocorre se o suporte e a confiança da regra a ser generalizada são R vezes maiores que *minsup* e *minconf*, respectivamente.

No algoritmo GARPA proposto em (OLIVEIRA; REZENDE; CASTRO, 2007), a generalização ocorre somente se os descendentes de um ancestral gerarem regras, e a regra desse ancestral (regra a ser generalizada) tiver grau de suporte *x*% maior que a regra de associação do descendente que possui o maior grau de suporte entre seus irmãos.

2.5. CONSIDERAÇÕES FINAIS

Como foi visto neste capítulo, a mineração de dados é utilizada em diversas aplicações. Dentre as suas tarefas, a tarefa de mineração de regras de associação é uma das mais utilizadas, tendo o algoritmo Apriori como um algoritmo clássico para a extração de tais regras. Além das regras de associação, foram explicados conceitos de regras de associação generalizadas e de regras redundantes, os quais são importantes para o entendimento deste trabalho.

Apesar dos diversos algoritmos que existem para mineração de regras de associação, com suas diversas metodologias, ainda existem problemas para fazer a mineração dessas regras, tais como uma melhor representação das regras para o usuário e a grande

quantidade de regras geradas. Em muitos trabalhos, utiliza-se lógica difusa para tentar resolver alguns desses problemas. Ontologias também são muito utilizadas para dar apoio ao processo de mineração de dados.

No capítulo 3 são descritos conceitos da lógica difusa, e conceitos sobre ontologia no capítulo 4. No capítulo 5 são apresentados alguns trabalhos que envolvem a lógica difusa na mineração de dados, assim como ontologia na mineração de dados e também, trabalhos que relacionam lógica difusa e ontologias na mineração de dados.

3. LÓGICA DIFUSA

3.1. INTRODUÇÃO

A Lógica difusa ou Lógica nebulosa (*Fuzzy Logic*) baseia-se na teoria de conjuntos difusos (*Fuzzy Sets*), e teve seus conceitos e princípios introduzidos por Lotfi A. Zadeh (ZADEH, 1987ª). Os conjuntos difusos são utilizados para modelar informações vagas ou imprecisas que estão presentes no mundo real, permitindo inferir uma resposta aproximada para uma questão baseada em um conhecimento inexato ou incompleto. Enquanto a lógica booleana clássica define apenas dois valores possíveis (verdadeiro (1) e falso (0)), a lógica difusa é multivalorada, ou seja, existe um conjunto de valores possíveis, que permite representar a informação imprecisa de maneira mais adequada.

3.2. CONJUNTOS DIFUSOS

A função característica de um conjunto clássico (*crisp*) pode assumir apenas os valores 0 ou 1, determinando os elementos que pertencem ou não ao conjunto. Ao definir um conjunto, sua função característica pode ser generalizada de forma que ela possa assumir valores em um determinado intervalo, e esses valores indicam o *grau de pertinência* do elemento no conjunto em questão. Essa função chama-se *função de pertinência* e o conjunto definido por ela chama-se *conjunto difuso* (KLIR; YUAN, 1995).

A função de pertinência associa elementos de um conjunto X a números reais no intervalo [0, 1]. Esta função é normalmente representada por: $\mu_A : X \rightarrow [0,1]$.

Assim, o valor de $\mu_A(x)$ representa o grau de pertinência do elemento *x* no conjunto difuso *A*. Quanto mais próximo o valor de $\mu_A(x)$ for de 1, maior o grau de pertinência de *x* no conjunto *A*.

Um exemplo comum de conjuntos difusos, extraído de (KLIR; YUAN, 1995), define três conjuntos difusos que representam os conceitos *jovem, adulto* e *idoso* em função da idade de um ser humano. A figura 3.1 define as funções de pertinência para cada um desses (respectivamente $\mu_J(x)$, $\mu_A(x)$ e $\mu_I(x)$), cujas representações gráficas são mostradas na figura 3.2.

$$\mu_J(x) = \begin{cases} 1 & \text{Quando } x \leq 20 \\ (35 - x)/15 & \text{Quando } 20 < x < 35 \\ 0 & \text{Quando } x \geq 35 \end{cases}$$

$$\mu_A(x) = \begin{cases} 0 & \text{Quando } x \leq 20 \text{ ou } x \geq 60 \\ (x - 20)/15 & \text{Quando } 20 < x < 35 \\ (60 - x)/15 & \text{Quando } 45 < x < 60 \\ 1 & \text{Quando } 35 \leq x \leq 45 \end{cases}$$

$$\mu_I(x) = \begin{cases} 0 & \text{Quando } x \leq 45 \\ (x - 45)/15 & \text{Quando } 45 < x < 60 \\ 1 & \text{Quando } x \geq 60 \end{cases}$$

Figura 3.1 - Funções de pertinência para os conjuntos *jovem, adulto* e *idoso* (KLIR; YUAN, 1995).

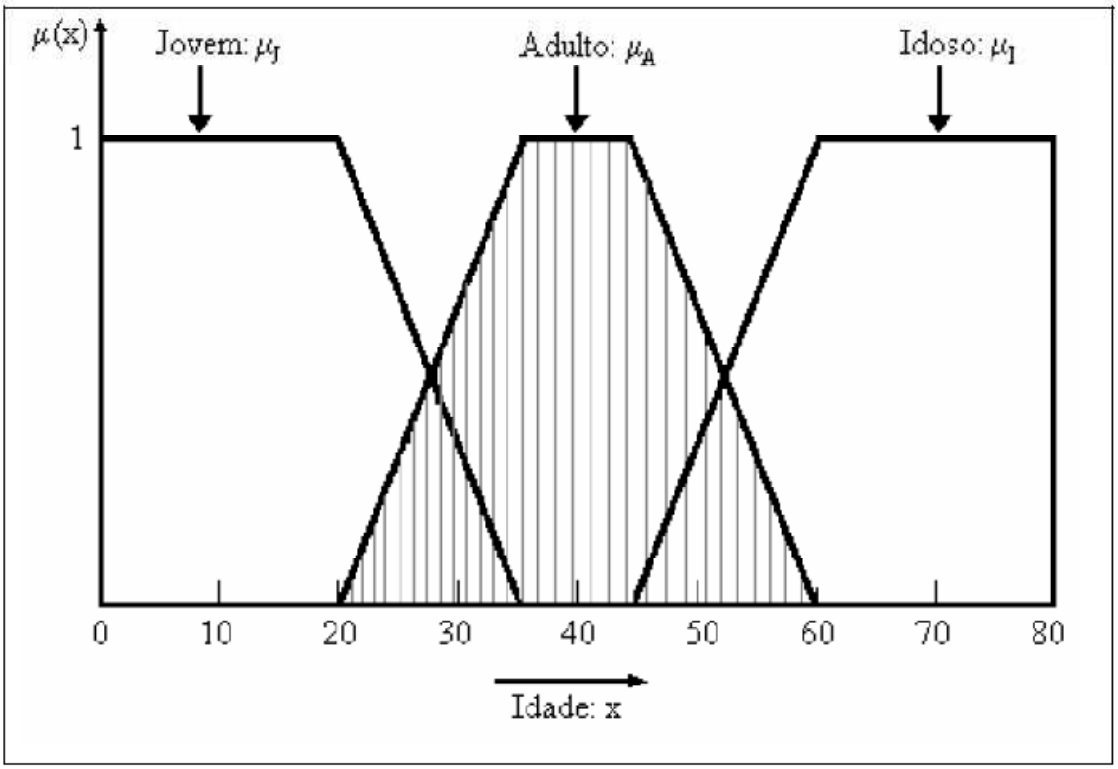


Figura 3.2 - Conjuntos representando os conceitos de *jovem, adulto* e *idoso* (KLIR; YUAN, 1995).

Assim como ocorre para conjuntos clássicos, também são definidas operações entre conjuntos nebulosos (ZADEH, 1987a) (DUBOIS; PRADE, 1980):

- *Conjunto vazio*: $\forall\, x \in X, \mu_{\varnothing}(x) = 0$;
- *Igualdade*: $\forall\, x \in X, A = B \Leftrightarrow \mu_A(x) = \mu_B(x)$;
- *Subconjunto difuso*: $\forall\, x \in X\,, A \subseteq B \Leftrightarrow \mu_A(x) \leq \mu_B(x)$;
- *Complemento*: $\forall\, x \in X,\ \mu_{\bar{A}}(x) = 1 - \mu_A(x)$;
- *União*: $\forall\, x \in X,\ \mu_{A \cup B}(x) = max(\mu_A(x), \mu_B(x))$;
- *Intersecção*: $\forall\, x \in X,\ \mu_{A \cap B}(x) = min(\mu_A(x), \mu_B(x))$,

onde *max* é o operador de máximo (que obtém o valor máximo da relação) e *min* é o operador de mínimo (que obtém o valor mínimo da relação).

Os elementos de dois ou mais conjuntos difusos podem ser relacionados para representar presença ou ausência de associação entre eles. Essas relações são chamadas de *relações difusas*, e são apresentadas na seção 3.3.

3.3. RELAÇÕES DIFUSAS

O conceito de relação difusa generaliza o conceito de relação clássica, uma vez que permite modelar a intensidade com a qual os elementos estão relacionados entre si. Desse modo, dada uma relação difusa *R*, atribui-se um valor $\mu_R(x,y)$ no intervalo [0, 1] representando o grau com que os elementos *x* e *y* estão relacionados por *R*. As relações difusas podem envolver diversos conjuntos nebulosos, e quando envolvem dois conjuntos nebulosos são chamadas de relações binárias.

Algumas propriedades de relações clássicas binárias também foram redefinidas para as relações difusas binárias (CHEN, 1998):

- *Propriedade reflexiva*: $\forall\, x \in X,\ \mu_R(x, x) = 1$;
- *Propriedade simétrica*: $\forall\, (x, y) \in X \times X,\ \mu_R(x, y) = \mu_R(y, x)$;
- *Propriedade transitiva* (*transitividade max-min*):

$\forall\, (x, z) \in X \times X,\ \mu_R(x, z) \geq max_{\,y \in X}\ min\ [\ \mu_R(x, y), \mu_R(y, z)\ ]$.

A expressão que define a transitividade difusa é o resultado da composição de

duas relações difusas, que nesse caso é a composição *max-min*. Alternativamente, outras definições de transitividade difusa são possíveis e úteis em algumas aplicações (KLIR; YUAN, 1995) (DUBOIS; PRADE, 1980).

Uma relação difusa binária que é reflexiva, simétrica e transitiva é conhecida como *relação difusa de equivalência* ou *relação de similaridade*. De acordo com (ZADEH, 1987b), “o conceito da relação de similaridade é essencialmente uma generalização do conceito da relação de equivalência”.

3.4. CONSIDERAÇÕES FINAIS

Os conceitos da lógica difusa são utilizados em diversas aplicações com a finalidade de melhor representar as incertezas geralmente presentes no mundo real. Assim, a lógica difusa fornece poderosas ferramentas para a computação de informações cujo significado não pode ser diretamente representado. Dentro da mineração de dados, muitas vezes a análise do conhecimento obtido pode ser influenciada pela qualidade de representação das informações, e na tentativa de melhorá-la, o uso da lógica difusa na mineração de dados tem sido bastante explorado. A solução apresentada no presente trabalho utiliza conceitos da lógica difusa para representar a similaridade semântica entre os itens.

Trabalhos e aplicações que usam a lógica nebulosa dentro da mineração de dados são apresentados no capítulo 5.

4. ONTOLOGIAS

4.1. DEFINIÇÃO

Do ponto de vista filosófico, ontologias referem-se a um sistema particular de categorias que descreve uma determinada visão de mundo (GUARINO, 1998). No meio computacional, a definição de ontologia mais utilizada é a de (GRUBER, 1993). Segundo ele, “Uma ontologia é uma especificação formal e explícita de uma conceitualização compartilhada”. Para uma melhor compreensão do significado da frase, (USCHOLD; GRUNINGER, 2004) analisaram o significado de cada um de seus termos:

- *Conceitualização*: refere-se a um modelo abstrato de como as pessoas pensam sobre as coisas do mundo, geralmente restrito a alguma área particular;
- *Especificação explícita*: significa que definições e nomes explícitos são atribuídos aos conceitos e relacionamentos do modelo abstrato. Neste caso, o nome é um termo e a definição é uma especificação do significado do conceito ou relacionamento;
- *Formal*: significa que a especificação da conceitualização é codificada em uma linguagem cujas propriedades formais são bem compreendidas. A formalização é um importante meio para remover a ambiguidade, que é prevalente na linguagem natural e em outras notações informais. Além disso, possibilita o uso de mecanismos de inferência automática para derivar novas informações a partir das especificações;
- *Compartilhada*: refere-se a um dos principais propósitos de uma ontologia, que é a utilização e reutilização em diferentes aplicações e comunidades.

De forma geral, ontologias proveem um vocabulário de termos que se referem aos conceitos de um domínio específico (conceitualização) e alguma especificação do significado desses termos (especificação explícita), geralmente tendo por base alguma teoria formal (formalismo). Segundo Uschold & Gruninger (USCHOLD; GRUNINGER, 2004), o que distingue as diferentes abordagens de ontologias é a forma de especificar o significado dos termos. A figura 4.1 apresenta um espectro evolutivo que ilustra os tipos de representações que geralmente são consideradas ontologias na literatura. À esquerda da

figura, existem representações simplificadas de ontologias, denominadas “ontologias leves”, que possuem apenas termos com pouca ou nenhuma especificação de significado. Seguindo para o lado direito da figura, temos que o grau de formalização das representações aumenta, assim como a especificação do significado se torna mais precisa. No extremo direito da figura, temos as “ontologias autênticas”, nas quais se encontram teorias rigorosamente formalizadas.

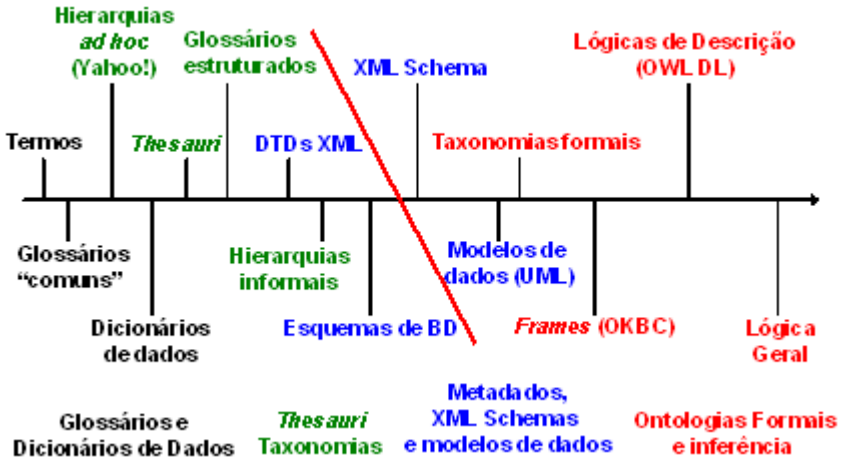
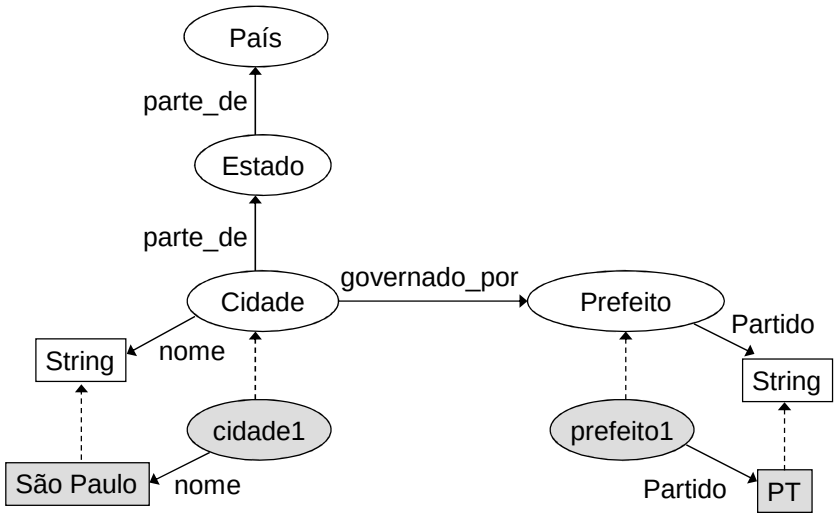


Figura 4.1 - Tipos de ontologias (USCHOLD; GRUNINGER, 2004).

Para modelar adequadamente uma conceitualização, a maioria das ontologias é constituída por conceitos, atributos, relacionamentos, axiomas formais e instâncias. Conceitos ou classes representam coisas que fazem parte de um domínio, tipicamente organizados em uma taxonomia de classes e subclasses. Atributos descrevem características dos conceitos e os relacionamentos expressam como os conceitos estão interligados. Atributos e relacionamentos também podem ser chamados de propriedades dos conceitos. Axiomas são restrições sobre conceitos e propriedades, cujo principal propósito é expressar significado de forma que máquinas sejam capazes de interpretá-lo por meio de mecanismos de raciocínio automático (USCHOLD; GRUNINGER, 2004). Por fim, as instâncias dos conceitos e das propriedades representam o conjunto de indivíduos ou objetos do mundo real que estão de acordo com a conceitualização modelada pela ontologia.

A figura 4.2 apresenta um exemplo de ontologia simples, contendo conceitos representados por elipses em branco, propriedades por setas contínuas e valores de atributos por retângulos. A instanciação é feita por setas tracejadas e as instâncias possuem tom acinzentado. No exemplo, o relacionamento *parte_de* possui a restrição do axioma de

transitividade, permitindo inferir que instâncias de *Cidade* também fazem parte de instâncias de *País*.



Axioma de transitividade para o relacionamento *parte_de*:

```
parte_de(X,Y).
parte_de (X, Z) :- parte_de (X,Y), parte_de (Y,Z).
```

Figura 4.2 - Exemplo de uma ontologia simples.

4.2. REPRESENTAÇÃO DE ONTOLOGIAS

Ao se desenvolver uma ontologia, é muito importante escolher uma forma de representação que seja expressiva o suficiente para capturar a conceitualização. De um modo geral, ontologias são fundamentadas em teorias formais a fim de que humanos e máquinas possam interpretar o conhecimento de domínio sem ambiguidade. Dentre os formalismos de representação existentes na literatura, as *Lógicas de Descrição* (*Description Logics* - DL) (NARDI; BRACHMAN, 2003) são consideradas uma das famílias mais importantes de representação formal de conhecimento, constituindo a base para linguagens de representação de ontologias como, por exemplo, a *Web Ontology Language* (OWL) (SMITH; WELTY; MCGUINESS, 2004).

As *Lógicas de Descrição* são um subconjunto da lógica de primeira ordem, a qual propõe a representação do conhecimento através de predicados. Uma base de conhecimento de *Lógica de Descrição* é composta por dois componentes: *TBox*

(*Terminological Box*) e *ABox* (*Assertional Box*). A *TBox* contém conhecimento intencional na forma de uma terminologia e é construída através de declarações que descrevem propriedades e relacionamentos gerais dos conceitos. A *ABox* contém conhecimento estensível, ou seja, conhecimento específico sobre os indivíduos do domínio considerado (NARDI; BRACHMAN, 2003).

Cada *Lógica de Descrição* é formada por um conjunto especifico de construções que permitem obter expressões complexas de conceitos a partir de conceitos mais simples. Exemplos de construções de conceitos são operações de complemento, intersecção, união, restrições de propriedade, quantificação universal, e restrições numéricas (CALVANESE; DE GIACOMO, 2005). As *Lógicas de Descrição* distinguem-se pelas construções que elas proveem. Desse modo, as diversas linguagens de *Lógicas de Descrição AL* (*Attributive Language*), diferem-se pela quantidade de construções que elas permitem modelar. Quanto maior a quantidade de construções, mais expressiva e complexa é a *Lógica de Descrição*.

Dentre as diversas linguagens baseadas no formalismo da *Lógica de Descrição*, a OWL é uma das linguagens mais difundidas por ser uma recomendação oficial do consórcio *World Wide Web Consortium* (W3C) para criação de ontologias na *Web Semântica* (BERNERS-LEE; HENDLER; LASSILA, 2001). De acordo com Grau, et al., (2008), a OWL é uma família de três linguagens diferentes (normalmente denominadas de espécies) de um poder expressivamente crescente:

- *OWL Lite*: provê os elementos básicos para a representação de conceitos, relacionamentos e restrições simples de propriedades, sendo o dialeto de OWL menos expressivo e complexo. Em função de suas construções mais simples, facilita o desenvolvimento de ferramentas e máquinas de inferência que manipulam as ontologias. Não permite os conceitos de operações como união e complemento;

- *OWL DL*: provê expressividade aliada à garantia de que todas as inferências sejam computáveis e processadas em um tempo finito. Corresponde à Lógica de Descrição *SHOIN* (FIKES; HAYESB; HORROCKS, 2004), sendo uma das linguagens mais utilizadas para a criação de ontologias na *Web* Semântica;

- *OWL Full*: é a variante da OWL mais expressiva, sem garantias computacionais. Desta forma, nenhuma implementação de OWL Full existe

atualmente, não sendo claro se é possível sua implementação na prática (GRAU, et al., 2008).

De forma geral, as linguagens baseadas em Lógicas de Descrição são adequadas para representar ontologias de forma expressiva, além de oferecerem suporte ao raciocínio lógico através de máquinas de inferência, possibilitando a verificação de consistência, a classificação correta das instâncias e a obtenção de novas informações a partir das especificações presentes nas ontologias.

4.3. APLICAÇÃO DE ONTOLOGIAS

No meio da Tecnologia da Informação, as ontologias podem ser aplicadas de diversas formas. Em geral, são utilizadas em contextos de compartilhamento de conhecimento, reuso e organização de informação semântica, servindo como um mecanismo de apoio para a compreensão e a comunicação de conhecimento de domínio entre usuários e sistemas computacionais. “Ontologias vem sendo desenvolvidas em diversas áreas como e-Science, medicina, biologia, geografia, astronomia, defesa, e nas indústrias automotivas e aeroespaciais” (GRAU, et al., 2008).

Uma das aplicações mais relevantes de ontologias é a estruturação semântica de informação (documentos, páginas *Web*, banco de dados, entre outros). Neste contexto, ontologias são utilizadas para descrever a semântica de repositórios de informação e tornar o seu conteúdo explícito (WACHE, *et al.*, 2001). Assim, é possível tratar conflitos semânticos presentes nos dados e propiciar respostas mais abrangentes às consultas. Uma das aplicações de ontologias que segue esta ideia é a *Web* Semântica (BERNERS-LEE; HENDLER; LASSILA, 2001), que propõe o uso de ontologias e metadados para estruturar e dar significado ao conteúdo das páginas *Web*. A partir da organização semântica do conteúdo *Web*, agentes inteligentes podem processar e interpretar o conhecimento embutido nos documentos para realizar consultas e tarefas complexas.

(HU; ZHAO, 2007) aplicam ontologias na *Biblioteca Digital* (*Digital Library*) - que é uma coleção de objetos digitais e de serviços associados ao armazenamento, descoberta, recuperação e preservação desses objetos -, para facilitar o entendimento de máquinas e processarem informações automaticamente, e assim, providenciar melhores serviços de conhecimento para usuários.

Seguindo na linha de aplicação de ontologias para a estruturação semântica da informação, este trabalho utiliza a semântica de domínio, representada por ontologias difusas para realizar a mineração de regras de associação generalizadas, levando em consideração a similaridade semântica entre os itens. Desse modo, a ontologia difusa serve como um conhecimento prévio sobre possíveis relações de similaridades que possam existir entre os itens.

4.4. CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados conceitos de ontologias, bem como as diversas formas de representá-las, além de alguns exemplos onde ontologias podem ser aplicadas. Neste trabalho, ontologias são utilizadas como um conhecimento prévio de apoio ao processo de mineração de dados.

No capítulo seguinte, ontologias na mineração de dados e ontologias difusas na mineração de dados são apresentadas.

5. LÓGICA DIFUSA E ONTOLOGIAS NA MINERAÇÃO DE DADOS

5.1. LÓGICA DIFUSA NA MINERAÇÃO DE DADOS

Como foi visto na seção 2, regras de associação correspondem a um importante tópico dentro da Mineração de Dados. Os primeiros trabalhos com regras de associação realizavam seus algoritmos em dados categóricos (como CEP e marca do carro). No entanto, na maioria das aplicações científicas e comerciais, os bancos de dados possuem tabelas relacionais contendo atributos categóricos e quantitativos (idade e rendimento). Para trabalhar com banco de dados que possuem atributos quantitativos e categóricos, (SRIKANT; AGRAWAL, 1996) desenvolveram um método para mineração de regras de associação quantitativa. Segundo (LEE, 2001), o método proposto por (SRIKANT; AGRAWAL, 1996) consiste em particionar o domínio de atributos quantitativos, combinando as partições adjacentes, e transformando o problema em um binário (isto é, mineração de regras de associação em atributos categóricos). Em mineração de regras de associação quantitativas, os registros de uma tabela do banco de dados são considerados como transações, onde o par atributo e seu respectivo valor desempenham a função de um item na mineração de regras de associação categóricas. Por conseguinte, as regras de associação quantitativas têm pares de um atributo e seu respectivo valor como *<Idade: 20...30>* e *<Sexo: Feminino>* → *<Número de Amigos: 2...5>*.

A mineração de regras de associação envolvendo dados quantitativos requer a especificação de intervalos apropriados para cada atributo. Porém, muitas vezes esses intervalos podem não ser suficientemente concisos e significativos. Desse modo, ao invés de utilizar intervalos, algumas abordagens fazem uso de *termos linguísticos*[2]. Em tese, a representação linguística faz com que as regras descobertas sejam muito mais naturais para o entendimento humano. (AU; CHAN, 1999) relatam que a definição de *termos linguísticos* é baseada na teoria de conjuntos difusos (seção 3), e por isso diz-se que as regras que possuem esses termos são *regras de associação difusas*.

Na subseção 5.1.1 são relatados alguns trabalhos que realizam a mineração de

[2] Termos linguísticos são expressões ou modificadores linguísticos usados para representar valores nebulosos. Exemplos de termos linguísticos: *muito*, *pouco*, *bastante*, *muito pouco*.

dados quantitativos com o auxílio da lógica difusa. Já na seção 5.1.2, serão descritas abordagens que realizam a mineração de dados empregando conceitos da lógica difusa em taxonomias.

5.1.1. Lógica Difusa empregada na Mineração de Dados Quantitativos

Na maioria dos algoritmos para mineração de dados quantitativos, a identificação de regras interessantes é feita através de parâmetros definidos pelo usuário. Segundo (AU; CHAN, 1999), essa abordagem é frágil uma vez que muitos usuários não têm ideia de como estabelecer esses parâmetros. Caso o valor seja muito alto, o usuário pode perder regras úteis. Por outro lado, se o parâmetro for muito baixo, os usuários se deparam com uma quantidade de regras muito grande.

Para resolver esse problema, (AU; CHAN, 1999) criaram o algoritmo *FARM* (Fuzzy Association Rule Mining), no qual os termos linguísticos são associados, e existe um cálculo denominado *diferença ajustada* cujo propósito é identificar quais dessas associações são interessantes, o que torna desnecessário o uso de um parâmetro definido pelo usuário. Assim que as associações entre os termos linguísticos são identificadas como sendo interessantes, a formação das regras de associação é realizada com base em uma métrica confiança denominada de *peso de evidência*, a qual é utilizada para representar a incerteza nas regras de associação difusas. O *peso de evidência* tem valor positivo se um valor de atributo (ou termo linguístico) determina a presença de outro valor de atributo, e tem valor negativo caso um valor de atributo determina a ausência de outro valor de atributo. Desse modo, as regras de associação difusas obtidas pelo algoritmo *FARM* podem ser denominadas como regras de associação difusas *positivas* ou *negativas*, conforme os seus *pesos de evidência*.

Em (LEE; LEE-KWANG, 1997), através de conjuntos difusos estabelecidos pelo usuário, as tuplas com dados quantitativos são estendidas e a seguir convertidas em tuplas com dados binários. Na última etapa do trabalho, um algoritmo convencional de mineração de regras de associação é aplicado nas tuplas com os dados binários, resultando nas *regras de associação estendidas*. Um exemplo dessa regra estendida é *(Hambúrguer, $5)* → *(Refrigerante, $2)*, que pode ser interpretada como “clientes que gastam $5 com hambúrguer são propícios a gastar $2 com refrigerante”. Como se pode observar pela regra de associação estendida do exemplo acima, as regras de associação descobertas são compostas por pares (*atributo, valor*). Com o uso de conjuntos difusos, pode-se diminuir o número desses pares

nas regras, diminuindo também a quantidade de regras geradas. Além disso, conjuntos difusos tornam a descrição das regras de associação concisas e generalizadas. Por exemplo, caso as seguintes regras de associação existirem,

(Hambúrguer, $5) → (Refrigerante, $2)
(Hambúrguer, $6) → (Refrigerante, $3)
(Hambúrguer, $4) → (Refrigerante, $1,5)

elas poderiam ser escritas da seguinte forma:

(Hambúrguer, médio) → (Refrigerante, pequeno).

Segundo os autores, os usuários podem compreender mais facilmente as relações entre os atributos desse modo, já que as regras de associação podem ser apresentadas em formas linguísticas.

(ZHANG; SUN; WU, 2007) implementaram um algoritmo que consiste em particionar os atributos quantitativos contínuos usando um método de agrupamento difuso para transformar os dados dos atributos quantitativos contínuos em uma matriz funcional composta por membros difusos. A partir desse ponto, as regras de associação podem ser obtidas com base no algoritmo Apriori (seção 2.2.1) e na teoria de conjuntos difusos (seção 3). Para uma melhor compreensão do algoritmo, considere a tabela 5.1.

Tabela 5.1 - Matriz funcional de membros difusos (adaptada de (ZHANG; SUN; WU, 2007)).

Número	A			B			C			D		
	baixo	médio	alto	baixo	médio	alto	baixo	médio	alto	baixo	médio	alto
1	1	0	0	1	0	0	0	1	0	0	1	0
2	0,8	0,2	0	0	0	1	0,5	0,5	0	0	0	1
3	0,9	0,1	0	0,8	0,2	0	0,8	0,2	0	0	1	0
4	1	0	0	1	0	0	0,1	0,8	0	1	0	0
5	0	1	0	0	0,9	0,1	1	0	0	0	1	0
6	0,9	0,1	0	1	0	0	0,9	0,1	0	0	1	0
7	0	1	0	0,6	0,4	0	1	0	0	0	1	0
8	0,9	0,1	0	0,9	0,1	0	0,1	0,9	0	0,1	0,9	0
9	1	0	0	1	0	0	0,9	0,1	0	1	0	0
10	0	0	1	0	1	0	0	0	1	0	0	1
suporte	0,7	0,2	0,1	0,7	0,2	0,1	0,5	0,3	0,1	0,2	0,6	0,2

A tabela possui dados de informações sobre o estoque farmacêutico de uma drogaria de Shangai, e possui quatro atributos quantitativos: A (*earnings per share*), B (*net*

assets return rate), C (*cash flow per share*) e D (*stock price*). O primeiro passo de mineração de regras de associação quantitativas é transformar dados de atributos quantitativos em conjuntos difusos correspondentes. Além disso, cada um dos dados de um atributo é transformado em membros difusos. Para esse trabalho, foi utilizado o algoritmo de agrupamento difuso C-means (BEZDEK, 1981) para transformar a base original de dados em uma matriz funcional de membros difusos. *C* representa o número de conjuntos difusos obtidos a partir da partição dos atributos quantitativos. Para a tabela 5.1, *C* = 3.

O algoritmo de mineração de regras de associação difusas desenvolvido por (ZHANG; SUN; WU, 2007) possui os seguintes passos:

- Identificação de todos os k-pares de conjuntos frequentes, que são colocados em L_k. K é o número de pares que o conjunto frequente possui;
- Encontrar as regras de associação fortes a partir de L_k.

Considerando o valor de suporte mínimo (*minsup*) igual a 0.4 e os valores de suporte expressos na tabela 5.1, obtemos os seguintes conjuntos frequentes:

L1 = {*A.baixo, B.baixo, C.baixo, D.médio*}

L2 = {(<*A.baixo B.baixo*>*0,6*), (<*A.baixo C.baixo*>0,4),
(<*A.baixo D.médio*>0,4), (<*B.baixo C.baixo*>0,4),
(<*B.baixo D.médio*>0,5), (<*C.baixo D.médio*>0,4)}

L3 = {<*A.baixo B.baixo D.médio*>0,4}.

A partir do conjunto de pares de *itemsets* frequentes obtidos, podem-se obter regras de associação difusas. Um exemplo de uma regra para esse exemplo seria <*B.baixo D.médio*> → <*A.baixo*>.

Até esse ponto foram apresentados trabalhos envolvendo a lógica difusa na mineração de dados quantitativos. Essa abordagem é muito utilizada, porém não é a única a utilizar conceitos da lógica difusa na mineração de dados. Na seção 5.1.2 será descrita outra forma de utilizar a lógica difusa na mineração de dados.

5.1.2. Lógica Difusa empregada na Mineração de Dados Taxonômicos

A mineração de regras de associação generalizadas (seção 2.3) em taxonomias difusas é outra forma considerada para a mineração de dados envolvendo conceitos da lógica difusa.

A principal ideia por trás dos trabalhos envolvendo a mineração de regras de associação é a seguinte: enquanto nos trabalhos que utilizam taxonomias convencionais (*crisp*) um filho pertence ao seu ancestral com grau 1, nas taxonomias difusas esse grau de pertinência corresponde a γ ($0 \leq \gamma \leq 1$). Esse grau de pertinência de um item em uma taxonomia é levado em consideração no momento dos cálculos de suporte e confiança.

O trabalho desenvolvido por (CHEN; WEI; KERRE, 2000) propõe a mineração de regras de associação em taxonomias difusas. De acordo com esses autores, em muitas aplicações do mundo real um item pode pertencer parcialmente a um ancestral em uma taxonomia. Por exemplo, um *tomate* pode ser considerado uma *fruta* ou um *vegetal*, embora em diferentes graus cada um. Um exemplo de tal taxonomia é mostrado na figura 5.1, na qual um subitem pertence ao seu superitem com um certo grau. Nesse contexto, os cálculos do grau de suporte e de confiança precisam ser estendidos para considerar as características difusas da taxonomia. O algoritmo para mineração de regras de associação generalizadas proposto em (SRIKANT; AGRAWAL, 1995) foi estendido, com o objetivo de incorporar os conceitos de suporte, confiança e interesse das regras considerando a taxonomia difusa. Esse algoritmo para realizar a mineração de regras de associação considerando os graus de pertinência de cada subitem na taxonomia difusa em relação ao seu *superitem* para calcular o suporte e confiança das regras obtidas é denominado de *Algoritmo Estendido* (*Extended Algorithm*), uma vez que é uma extensão do algoritmo desenvolvido em (SRIKANT; AGRAWAL, 1995).

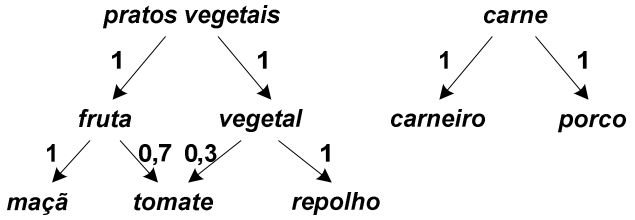


Figura 5.1 - Exemplo de taxonomia difusa (adaptada de (CHEN; WEI; KERRE, 2000)).

Posteriormente, (CHEN; WEI, 2002) desenvolveram outro trabalho envolvendo mineração de regras de associação generalizadas em taxonomias difusas. No entanto, o uso de *hedges linguisticos*[3] foi adicionado nas regras de associação difusas com o objetivo de expressar mais naturalmente o conceito obtido. "*Produtos muito caros → frutas tropicais*" é um tipo de regra de associação difusa com *hedges linguísticos*. Segundo os autores, o uso de *hedges linguísticos* torna o conhecimento descoberto mais compreensível e próximo da linguagem humana. Para quem toma decisões em empresas e instituições, especialmente os gerentes de níveis mais altos, esse tipo de conhecimento pode ser mais frequentemente usado e significativo. Além disso, o uso de *hedges linguísticos* pode enriquecer a semântica das regras de associação obtidas e torná-las mais granulares. Por exemplo, regras como "*maçã → jeans*", "*maçã cara → jeans legal*", "*maçã muito cara → jeans muito legal*" poderiam ser obtidas. A aplicação de *hedges linguísticos* modifica a estrutura da taxonomia difusa (pensando no exemplo da figura 5.1, além do item "maçã", ter-se-ia os itens "*maçã cara*" e "*maçã muito cara*", por exemplo), e então uma estratégia para a construção de uma nova estrutura taxonômica difusa é necessária. Após a construção da nova estrutura taxonômica, um algoritmo implementado para minerá-la é aplicado. Nesse trabalho, o algoritmo clássico para mineração de regras de associação generalizadas (SRIKANT; AGRAWAL, 1995) é chamado de GAR, o algoritmo por eles desenvolvido para lidar com taxonomias difusas (CHEN; WEI; KERRE, 2000) é denominado de FGAR, e por fim, o algoritmo que faz uso de *hedges linguísticos* criando uma nova estrutura taxonômica difusa antes de minerá-la recebe o nome de HFGAR.

As abordagens citadas anteriormente são muito utilizadas na mineração de dados. No entanto, (ESCOVAR; BIAJIZ; VIEIRA, 2005) introduziram um novo algoritmo, denominado SSDM, o qual considera a similaridade semântica entre os itens do banco de dados. Esse algoritmo utiliza conceitos da lógica difusa para representar o grau de similaridade entre os itens, e propõe um novo método para o cálculo de suporte e confiança, devido à similaridade semântica entre os itens. Caso o valor do grau de similaridade entre os itens for igual a 1 (um), significa que a comparação entre os itens tem similaridade máxima e, de acordo com a propriedade reflexiva de relações binárias difusas, isto só pode ocorrer se um

[3]*Hedges linguísticos* são termos linguísticos como "muito", "mais ou menos", "tipo de" que acabam modificando o significado do termo que o segue. Maiores detalhes podem ser encontrados em (CHEN; WEI, 2002).

item for comparado com ele mesmo. Consequentemente, quando dois itens não idênticos são comparados, o grau de similaridade entre eles deve ser um valor maior ou igual a zero e menor que 1 (0 ≤ grau de similaridade < 1). Durante o processo de mineração, caso o grau de similaridade entre os itens seja maior ou igual à similaridade mínima definida pelo usuário, uma associação de similaridade semântica é detectada, significando que itens nessa associação são similarmente suficientes, sendo interessante para o usuário. Um exemplo de taxonomia que utiliza similaridade semântica entre os itens é representada na figura 5.2. Caso dois itens sejam suficientemente similares, uma associação difusa é feita. As associações difusas são expressas no algoritmo através de itens difusos, que são representações onde o símbolo "~" indica a relação entre os itens. A partir de itens comuns e itens difusos, regras de associação como "*maçã~tomate → porco"* e "*repolho → carneiro*" podem ocorrer.

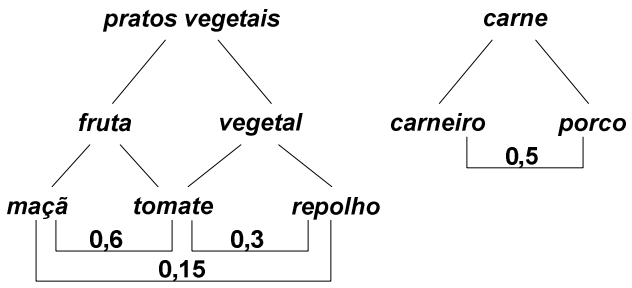


Figura 5.2 - Exemplo de uma taxonomia envolvendo similaridade semântica entre os itens.

5.2. ONTOLOGIAS NA MINERAÇÃO DE DADOS

Ontologias vem sendo aplicadas como um conhecimento de apoio na mineração de dados para auxiliar o processo de descoberta de conhecimento. Basicamente, elas podem ser aplicadas para um melhor entendimento da aplicação do problema, preparando e minerando os dados, gerando regras de associação em alto nível e para analisar interesses nas regras descobertas ou padrões.

O trabalho desenvolvido por (CHEN, et al, 2003) usa ontologia para aprimorar o suporte no processo de mineração. Nesse trabalho, os dados são elevados para conceitos mais generalizados de acordo com a ontologia. Depois disso, o processo de mineração de dados é realizado por um algoritmo convencional de mineração de regras de associação como o Apriori. Os autores afirmam que a prévia generalização dos dados torna possível considerar subcategorias no processo de cálculo de suporte, gerando regras com maior suporte. Mais

tarde, as regras obtidas podem ser mais fáceis de interpretar, uma vez que possuem conceitos em alto nível que representam informações mais ricas do que os termos específicos do banco de dados.

O algoritmo ExCIS, implementado por (BRISSON; COLLARD; PASQUIER, 2005), usa o conhecimento de domínio tanto na etapa de pré-processamento quanto na de pós-processamento. A etapa de pré-processamento utiliza ontologia para guiar a construção de conjunto de dados específicos para tarefas de mineração particulares. A próxima etapa é a aplicação de um algoritmo padrão para extrair padrões desses conjuntos de dados. Por último, na etapa de pós-processamento as regras mineradas podem ser interpretadas e/ou filtradas, uma vez que seus termos são generalizados de acordo com a ontologia. Por essa razão, a informação semântica utilizada no ExCIS suporta a preparação do conjunto de dados e permite reduzir o volume de padrões extraídos.

(BRISSON; COLLARD, 2008) sugerem modelar o conhecimento de domínio durante as etapas de compreensão das regras de negócio e dos dados com o objetivo de construir um sistema de informação baseado em uma ontologia. Após esse passo, a ontologia é utilizada em todo o processo de mineração de dados. A metodologia desenvolvida por esses autores, denominada KEOPS, permite utilizar o conhecimento de domínio para fazer uma seleção eficiente dos dados, bem como para realizar a preparação dos dados e a avaliação e interpretação dos resultados obtidos.

De acordo com (KOTSIFAKOS; MARKETOS; THEODORITIS, 2007), nenhuma medida para a avaliação dos padrões obtidos é suficientemente eficiente como os especialistas de domínio. Os especialistas de domínio podem melhor avaliar os padrões e decidirem se são ou não triviais. De modo a automatizar o processo de avaliação de padrões, esses autores incorporam o conhecimento de domínio neles. A ontologia é utilizada na etapa de pós-processamento para avaliar os padrões extraídos a partir dos algoritmos de mineração de dados. Os padrões que contradizem o conhecimento aceito de acordo com a ontologia são marcados como possivelmente inválidos. Caso contrário, os padrões aceitos serão posteriormente avaliados por um especialista de domínio e, se forem reconhecidos como conhecimento útil, a ontologia pode ser atualizada para incorporar esses novos padrões. Segundo os autores, esse método pode reduzir o custo em termos de tempo total para realizar o algoritmo de mineração de dados bem como reduzir o esforço dos especialistas de domínio para avaliar os padrões descobertos. Além disso, os padrões marcados como inválidos não são descartados, a menos que o usuário do sistema opte por invalidá-lo.

5.3. ONTOLOGIAS DIFUSAS NA MINERAÇÃO DE DADOS

Nesta seção são apresentadas abordagens de mineração de dados baseada em uma ontologia difusa e de mineração de regras de associação difusas baseada em uma ontologia. Antes de entrar em detalhes nos trabalhos envolvendo essas abordagens, será relatado o conceito de ontologia difusa e alguns trabalhos.

5.3.1. Ontologias Difusas

De acordo com o capítulo 4 e a seção 5.2, ontologias tradicionais (*crisp*) são capazes de representar conceitos, propriedades, relacionamentos, instâncias e axiomas de um domínio de aplicação. No entanto, essas ontologias seguem a teoria clássica de conjuntos (*booleana*), capturando apenas informações precisas ou completas. Contudo, existem alguns domínios em que a definição de conceitos, instâncias e relacionamentos é vaga ou imprecisa e, portanto, tais domínios não são representados adequadamente pelas ontologias tradicionais. Por exemplo, é difícil representar em ontologias *crisp* conceitos como “cremoso”, “escuro”, “quente”, “alto” ou “espesso”, para os quais não é possível obter uma definição clara e precisa (STRACCIA, 2006). Assim, ontologias tradicionais têm sido estendidas para incorporar conceitos da Lógica Difusa, resultando em ontologias difusas que possibilitam representar e inferir conhecimento sobre informações imprecisas.

Nessa seção são mostrados alguns trabalhos envolvendo ontologias difusas. A recuperação de informação semântica baseada em ontologia é um tópico muito discutido nas pesquisas atuais. Para propiciar a recuperação de informação semântica difusa, várias abordagens foram desenvolvidas. (ZHAI; SHEN; LIANG; JIANG, 2008) aplicam uma estrutura de ontologia difusa para sistemas de recuperação de informação em *e-Commerce* (Comércio eletrônico). Essa estrutura é composta por três partes: conceitos, propriedades dos conceitos e valores das propriedades, onde os valores de propriedades podem ser variáveis linguísticas de conceitos difusos. A expansão de consulta semântica é construída por relação de ordem, equivalência, inclusão, reversão e relação de complemento entre conceitos difusos definidos nas variáveis linguísticas da ontologia através de *RDF* (*Resource Description Framework*). A aplicação por eles construída para recuperar informações de consumidores, produtos e fornecedores pode superar os modelos de outras ontologias difusas e facilitar a

recuperação de informação semântica através de conceitos difusos na Web Semântica. (LEITE; RICARTE, 2008) descrevem uma estrutura para codificar uma base de conhecimento geográfica composta por múltiplas ontologias que estão relacionadas, cujos relacionamentos estão expressos como relações difusas. Cada ontologia representa uma área distinta do conhecimento de domínio relacionado com referências geográficas. Essa organização do conhecimento é utilizada em métodos nebulosos para expandir a consulta inicial do usuário.

(ZHANG, et al, 2008) estenderam o modelo de ontologia de domínio para ontologia difusa para extrair conhecimento de domínio a partir de modelos de banco de dados difusos, de tal modo que modelos de entidade e relacionamento difusos (ER difusos) possam suportar proveitosamente o desenvolvimento da ontologia difusa. Eles também estendem a *OWL DL*, criando a *OWL DL* difusa. Dados o modelo de ER difusa e a ontologia *OWL DL* difusa, e, uma vez estabelecidos os relacionamentos entre ambos, é aplicado um algoritmo de tradução e preservação semântica, o qual realiza a tradução dos termos do ER difuso para a ontologia difusa.

5.3.2. Ontologias Difusas na Mineração de Dados e Mineração de Regras de Associação Difusas baseada em Ontologias

Como visto anteriormente (seção 5.2), ontologias estão sendo utilizadas como um conhecimento prévio de apoio na mineração de dados para auxiliar o processo de descoberta de conhecimento. Na seção 5.2, foram citados trabalhos envolvendo mineração de regras de associação baseado em uma ontologia. Na subseção 5.3.1 foram apresentados alguns trabalhos envolvendo o uso de ontologias difusas. Nessa seção são descritos trabalhos envolvendo a mineração de dados baseada em uma ontologia difusa e a mineração de regras de associação difusas baseadas em uma ontologia.

Em muitas aplicações do mundo real, as taxonomias (hierarquia *é um*) que estão representadas nas ontologias nem sempre possuem itens com grau de pertinência igual a 1 (um) entre um ancestral e seu descendente imediato, ou seja, um descendente pertence cem por cento ao seu respectivo ancestral. Nessas ocasiões, o uso de ontologias difusas é aplicado.

Em (ESCOVAR; YAGUINUMA; BIAJIZ, 2006), as ontologias difusas são utilizadas como um conhecimento de apoio para proporcionar representação semântica sobre os dados minerados. Em outras palavras, o algoritmo desenvolvido por eles, denominado

XSSDM (eXtended Semantically Similar Data Mining) utiliza a ontologia difusa para representar as relações de similaridade semântica entre os dados. A ontologia difusa inclui grau de similaridade entre os conceitos, o qual é processado pelo algoritmo para gerar regras de associação mais compreensíveis que refletem a similaridade semântica entre os dados. Na figura 3, os números contidos nas ligações entre dois itens representam o grau de similaridade semântica que existe entre eles. Assim, os itens *maçã e caqui*, por exemplo, possuem grau de similaridade semântica entre eles igual a 0,75. Esse grau de similaridade é obtido a partir da ontologia. Essa abordagem também trata a questão de generalização dos itens. A generalização das regras de associação ocorre somente se todos os descendentes (filhos) do ancestral imediato (pai) são, dois a dois, suficientemente similares, ou seja, possuem similaridade mínima maior ou igual à similaridade mínima desejada (*minsim explicado da seção 5.1*). Um exemplo de regra de associação generalizada para a representação de ontologia da figura 5.3 seria *Fruta* → *Frango*, obtida a partir da regra de associação *Maçã~Caqui~Tomate* → *Frango*. Note que *Maçã~Caqui~Tomate* é um item difuso (seção 5.1) que foi gerado por seus itens serem dois a dois suficientemente similares. É importante levar em consideração que este trabalho não faz nenhum tratamento de redundância.

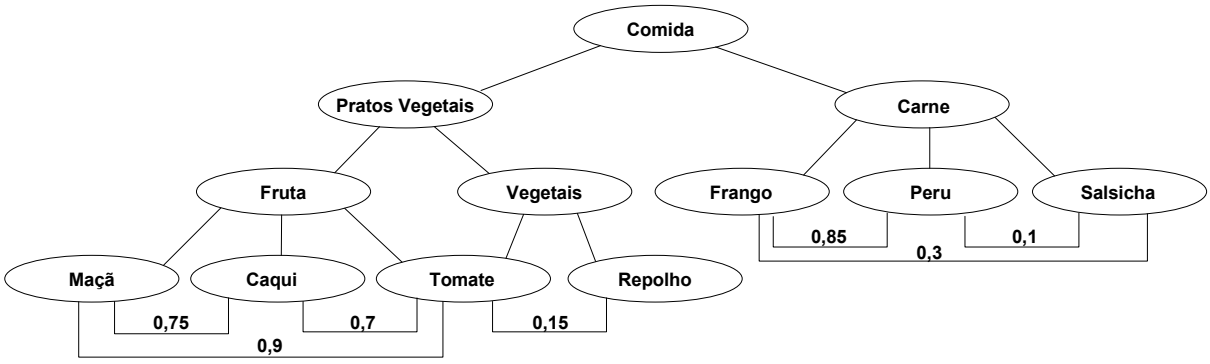


Figura 5.3 - Representação de uma Ontologia Difusa.

Em (FARZANYAR; KANGAVARI; HASHEMI, 2006), a mineração de regras de associação difusas é conduzida por uma ontologia de domínio. Ela faz uso de uma hierarquia de conceitos para melhorar os resultados da mineração de regras de associação difusas. Nesse trabalho, cada atributo é tratado como uma variável linguística, e essas variáveis são divididas em vários termos linguísticos. A ontologia é constituída por uma taxonomia de relacionamentos que estão relacionadas com todo conceito, e pela relação semântica entre conceitos. Cada atributo quantitativo do banco de dados é substituído por um

ou alguns conjuntos difusos. Para reduzir o tempo de execução, são considerados apenas os relacionamentos entre os itens relacionados a um conceito ou os itens relacionados a conceitos tendo relacionamento semântico na ontologia. Segundo os autores, o algoritmo de mineração de dados baseado em ontologia torna as regras mais visuais, mais interessantes e compreensíveis.

5.4. CONSIDERAÇÕES FINAIS

A lógica difusa vem sendo muito utilizada na mineração de dados para tentar solucionar as incertezas presentes no mundo real. Tendo isso em consideração, este capítulo descreveu diversos trabalhos que aplicam a lógica nebulosa dentro da mineração de dados. Ontologias também são muito utilizadas na mineração de dados como um mecanismo de apoio ao processo de mineração. Desse modo, ontologias difusas também são aplicadas na mineração de dados, cujas relações e conceitos difusos estão presentes na ontologia, e essa, é utilizada para apoiar o processo de mineração de dados.

Tendo em consideração os conceitos apresentados até este capítulo, o capítulo 6 propõe um novo algoritmo, denominado NARFO, para realizar a mineração de regras de associação generalizadas e não redundantes baseadas em uma ontologia difusa.

6. ALGORITMO NARFO

6.1. INTRODUÇÃO

Como foi visto no capítulo 2, o algoritmo Apriori é um algoritmo clássico de mineração de regras de associação e, a partir dele, diversos outros algoritmos foram desenvolvidos. No entanto, esse algoritmo realiza a MD apenas nos níveis folhas de uma determinada hierarquia. Para ampliar a capacidade de geração de regras de associação, (SRIKANT; AGRAWAL, 1995) desenvolveram um algoritmo para realizar a mineração de regras de associação em qualquer nível de uma hierarquia. As regras de associação obtidas foram chamadas de regras de associação generalizadas (capítulo 2.3). Um grande problema, no entanto, ocorrente na maioria dos algoritmos de regras de associação e regras de associação generalizadas é a grande quantidade de regras de associação geradas e de regras redundantes. Para resolver essa questão, vários algoritmos, utilizando diversos métodos, foram desenvolvidos (capítulo 2.4). O presente trabalho tem como essência realizar a mineração de regras de associação generalizadas não redundantes. Os procedimentos e métodos utilizados para tal serão descritos nas seções seguintes deste capítulo.

No capítulo 3 e no capítulo 4, conceitos de lógica difusa e de ontologias foram apresentados, respectivamente. Com base nos conceitos apresentados nos capítulos 3 e 4, no capítulo 5 foram apresentados alguns trabalhos envolvendo a lógica difusa na mineração de dados (capítulo 5.1), ontologias na mineração de dados (capítulo 5.2) e trabalhos que utilizam ambos os conceitos na mineração de dados (capítulo 5.3). Na abordagem deste trabalho, foi utilizada uma ontologia difusa servindo como um conhecimento prévio de apoio para o processo de mineração de regras de associação generalizadas não redundantes.

Na seção seguinte é apresentada a contextualização dos conceitos utilizados na mineração de dados semanticamente similares. Esses dados possuem as similaridades semânticas entre eles representadas pela ontologia difusa.

6.2. CONTEXTUALIZAÇÃO

Os algoritmos que trabalham com a mineração de dados categóricos

geralmente analisam se a base de dados é composta por atributos desse tipo, e então verificam a sua ocorrência ao longo da base, assim como a sua associação com outros itens. Esses algoritmos identificam cada item simplesmente como uma sequência de caracteres (*string*). Desse modo, itens como *pão* e *baguete*, por exemplo, seriam considerados *strings* totalmente diferentes. De fato, as *strings* são completamente distintas, mas os itens representados por elas constituem noções muito semelhantes. Afinal, *pão* e *baguete* são utilizados para o mesmo fim (alimentação), possuem formato semelhante, são constituídos basicamente pelos mesmos ingredientes e geralmente são colocados para a venda no mesmo local. Trata-se de uma similaridade semântica entre os itens, ignorada por algoritmos convencionais, que dessa forma podem desperdiçar informações importantes. Essa ideia foi proposta por (ESCOVAR; BIAJIZ; VIEIRA, 2005) com a criação do algoritmo SSDM (Semantically Similar Data Mining), e estendida em (ESCOVAR; YAGUINUMA; BIAJIZ, 2006) com o algoritmo XSSDM (eXtended Semantically Similar Data Mining) para o uso de ontologias. Esses trabalhos, porém, possuem apenas um pequeno tratamento de generalização das regras e nenhum tratamento de redundância. Minerar os dados sem considerar a similaridade semântica entre os itens *pão* e *baguete*, por exemplo, leva a regras de associação que envolvem apenas cada item individualmente. Analisar esses itens similares, levando em consideração as associações entre eles, pode levar a descoberta de regras de associação relevantes também.

Seguindo essa linha, a mineração de dados semanticamente similares estende a forma de mineração de regras de associação convencional, de maneira que os itens semanticamente similares sejam considerados como se a sua associação constituísse um único item. A similaridade semântica entre os dados é representada por um grau de similaridade entre os itens, que é obtido de acordo com a definição de relação de similaridade (seção 3.3). Assim, se o grau de similaridade entre os itens for igual a 1, significa que a similaridade entre os itens é máxima. Segundo a propriedade reflexiva das relações difusas, isso só pode ocorrer caso um item seja comparado a ele mesmo. Por conseguinte, sempre que dois itens não idênticos forem comparados, o valor do grau de similaridade entre eles terá um valor maior ou igual a 0 (zero) e menor que 1 (um). Por exemplo, suponha que o grau de similaridade entre os itens *pão* e *baguete* seja igual a 0,8. Em uma avaliação simples, pode-se concluir que não seria adequado simplesmente somar os suportes obtidos por cada um dos itens, uma vez que eles não são idênticos, mas similares. Para que o valor do suporte da associação entre eles reflita a influência de ambos, o grau de similaridade entre os itens precisa ser considerado no cálculo do suporte.

Neste trabalho, as similaridades semânticas entre os itens estão contidas na ontologia difusa. Desse modo, o algoritmo obtém o grau de similaridade entre os itens a partir da ontologia difusa. Na próxima seção será explicado o algoritmo NARFO, criado para realizar a mineração de regras de associação generalizadas não redundantes baseada em uma ontologia difusa.

6.3. ALGORITMO NARFO

O algoritmo NARFO (Non-redundant and generalized Association Rules based on Fuzzy Ontologies) é um algoritmo que realiza mineração de regras de associação. Por isso, necessita dos parâmetros de suporte mínimo e confiança mínima. Pela adição dos conceitos da lógica difusa, um parâmetro indicando o grau de similaridade semântica mínima desejado também se fez necessário e recebeu o nome de *minsim* (ESCOVAR; BIAJIZ; VIEIRA, 2005), assim como no SSDM e XSSDM. Além desses, outro parâmetro, denominado *mingen*, foi adicionado com a finalidade de indicar a porcentagem de generalização mínima. Esse parâmetro foi adicionado, pois em muitas ocasiões a generalização de uma certa porcentagem dos descendentes de um mesmo ancestral diminui o número de regras geradas e torna a regra generalizada mais compreensível aos olhos do usuário. O processo de generalização utilizando esse parâmetro será abordado adiante.

Assim, o algoritmo possui os seguintes parâmetros:

- *minsup*, que indica o suporte mínimo desejado;
- *minconf*, que representa a confiança mínima desejada;
- *minsim*, que é o grau de similaridade mínimo necessário entre os itens para que eles sejam considerados suficientemente similares e assim sejam associados durante a execução dos algoritmos;
- *mingen*, que corresponde à porcentagem mínima de descendentes de um mesmo ancestral que devem estar inclusos em regras geradas com itens idênticos (exceto pelo descendente em questão), de tal forma que se pode generalizar as regras contendo os descendentes para uma única regra contendo o ancestral e os demais itens comuns entre as regras que originaram a regra generalizada.

Esses parâmetros são fornecidos pelo usuário, sendo expressos por um valor real no intervalo [0, 1].

As etapas realizadas pelo algoritmo podem ser divididas conforme mostra a figura 6.1. As etapas destacadas em negrito usam o conteúdo contido na ontologia difusa. As etapas contendo fundo preenchido possuem tratamentos e técnicas inovadoras como a generalização de *itemsets* não frequentes (Avaliação dos candidatos) e o tratamento de generalização e redundância (Tratamento de Generalização e Redundância), que correspondem às contribuições e às inovações do presente trabalho em comparação com o XSSDM.

Para uma melhor compreensão do algoritmo NARFO, será utilizado um exemplo de transações de compra de um supermercado apresentadas na tabela 6.1 e de acordo com a ontologia de itens alimentícios apresentada na figura 6.2. Essa ontologia contém a especificação das relações e similaridades entre os itens. Serão considerados os seguintes valores dos parâmetros para o suporte mínimo desejado (*minsup*), a confiança mínima desejada (*minconf*), a similaridade mínima (*minsim*) e de generalização mínima (*mingen*):

- *minsup* = 0,4;
- *minconf* = 0,7;
- *minsim* = 0,7;
- *mingen* = 0,75.

Na figura 6.1, as etapas que possuem setas que chegam do raciocinador da ontologia difusa são as que utilizam relações, conceitos e inferências que estão contidos na ontologia difusa para auxiliar o processo de mineração de regras de associação. Ou seja, a ontologia difusa é utilizada como um conhecimento de apoio para essas etapas no processo de MD. Dentro da etapa de Avaliação dos candidatos está uma das contribuições do presente trabalho em relação ao algoritmo XSSDM, que é a Generalização de Itemsets não frequentes. E, por fim, após a aplicação do tratamento de generalização e redundância, temos o conjunto de regras de associação generalizadas não redundantes. Cada etapa do algoritmo será descrita a partir da seção 6.3.1 até a seção 6.3.8.

Ocódigo fonte do algoritmo NARFO está descrito no Apêndice deste trabalho.

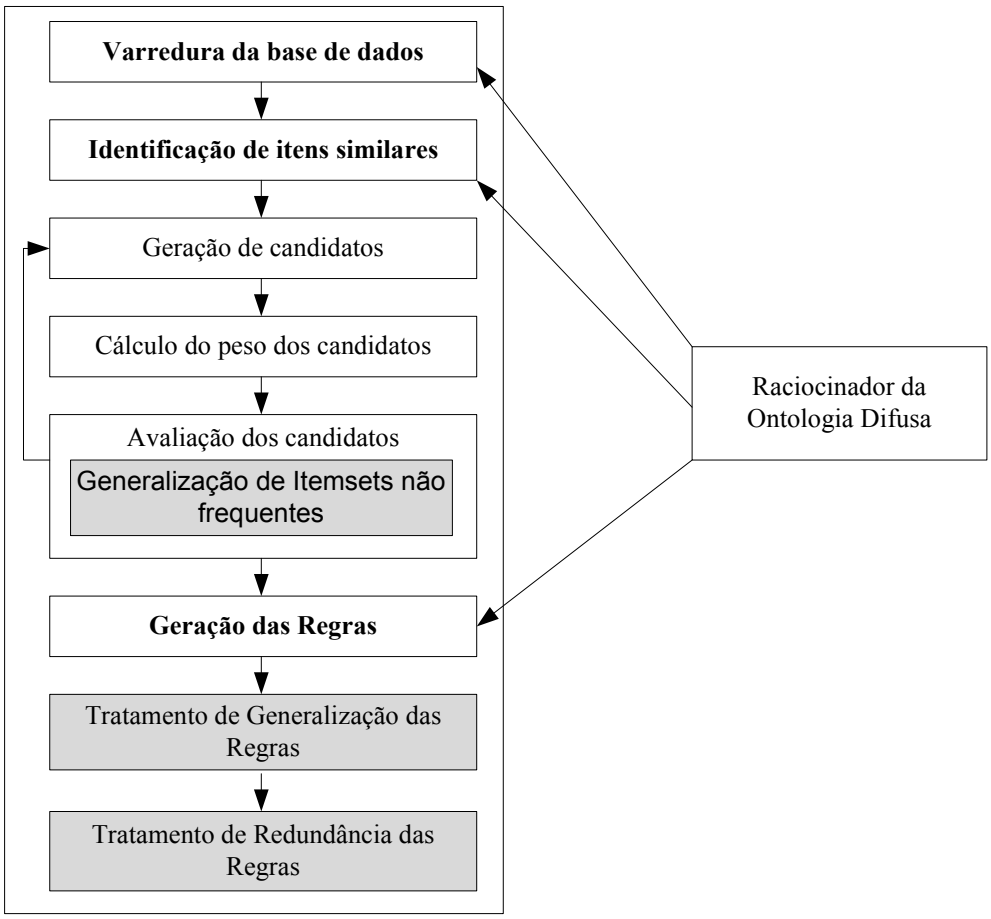


Figura 6.1 - Etapas do algoritmo NARFO.

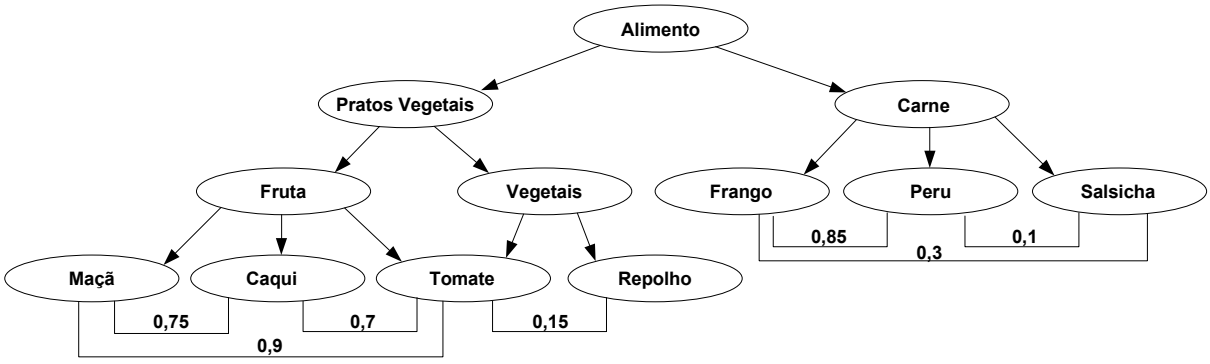


Figura 6.2 – Ontologia de itens alimentícios.

Tabela 6.1 - Tabela de Transações de compra de um supermercado.

Id	Pratos Vegetais	Carne
10	Maçã	Frango
20	Caqui	Peru
30	Tomate	Frango
40	Maçã	Peru
50	Repolho	Salsicha
60	Maçã	Frango
70	Tomate	Peru
80	Maçã	Frango
90	Caqui	Peru
100	Maçã	Peru

6.3.1. Varredura da base de dados

A etapa de varredura da base de dados é similar à etapa correspondente no algoritmo XSSDM. Nela, os itens no banco de dados são identificados, gerando itemsets de tamanho unitário (1-itemsets). Os itens do banco de dados têm total correspondência com os nós-folhas da ontologia difusa e, consequentemente, as relações de similaridades de um nó-folha com seus irmãos podem ser facilmente identificadas (com o auxílio do raciocinador da ontologia difusa). Então, considerando o exemplo de transações de compra de um supermercado da tabela 6.1, os seguintes itens são identificados após a varredura: *Maçã, Caqui, Tomate, Repolho, Frango, Peru e Salsicha*.

6.3.2. Identificação de itens similares

No algoritmo NARFO, os valores do grau de similaridade entre os itens irmãos são fornecidos pela ontologia difusa, a qual especifica a semântica do conteúdo da base de dados. Esses valores são fornecidos ao algoritmo pelo raciocinador da ontologia difusa. Tal ontologia pode ser criada por especialistas do domínio ou ser reutilizada de uma já existente, uma vez que há um número crescente de ontologias disponível na Web Semântica. Ademais, as ontologias pré-existentes podem ser estendidas por especialistas de domínio, os quais podem definir valores de grau de similaridade apropriados, fazendo com que a tarefa de mineração de dados possa utilizar informação semântica consistente.

Uma vez identificada a ontologia difusa, essa etapa navega por sua estrutura

com a finalidade de encontrar similaridade semântica entre os itens. Caso o grau de similaridade semântica entre os itens seja maior ou igual à similaridade mínima desejada (*minsim*), uma associação de similaridade semântica é detectada, significando que itens contidos nessa associação são suficientemente similares (portanto interessantes para o usuário). Esses pares de associação compõem associações difusas de tamanho 2 e são expressos por itens difusos, onde o símbolo ~ é usado para representar relação de similaridade entre itens ($item_1$~$item_2$).

Após a identificação de associações difusas de tamanho 2, esta etapa verifica a existência de ciclos de similaridades. Esses ciclos são associações difusas de tamanho maior que 2, e para que eles possam existir, todos os itens que o compõem devem ser, dois a dois, suficientemente similares. Ou seja, conforme a teoria de intersecção entre conjuntos difusos (seção 3.2), o valor mínimo dos graus de similaridade envolvidos no ciclo deve ser maior ou igual a *minsim*.

Assim como todas as relações de similaridade, os ciclos de similaridades envolvem apenas itens irmãos na ontologia difusa. Desse modo, se um item pertence a mais de um ancestral, ele pode ser envolvido em muitos ciclos de similaridades, estabelecendo que cada um dos ciclos contém somente irmãos. Por exemplo, de acordo com a ontologia da figura 6.2, *Tomate* pertence a *Fruta* e a *Vegetais*, e, portanto *Tomate* pode estar envolvido em ciclos de similaridade contendo descendentes de *Fruta* e também em ciclos de similaridade que contêm descendentes *Vegetais*. Ao passo que o tamanho mínimo de um ciclo de similaridade é igual a 3, seu tamanho máximo é definido pelo número de descendentes que o ancestral do ciclo em questão possui. Então, a existência de ciclos de similaridade é verificada considerando os ancestrais dos itens difusos identificados, de modo a obter associações difusas de tamanho k ($k \in N$, $3 \leq k \leq s$, onde s é o número de descendentes que um ancestral possui).

Considerando o exemplo de transações do supermercado, a ontologia difusa da figura 6.2 é analisada e, com *minsim* = 0.7, é possível identificar os itens difusos mostrados na tabela 6.2. De acordo com a tabela, note a identificação de um ciclo de similaridade (*Tomate~Caqui~Maçã*) uma vez que todos os seus itens são, em pares, suficientemente similares. A notação sim($item_1$, $item_2$) representa a relação de similaridade entre $item_1$ e $item_2$.

Tabela 6.2 - Relações de similaridade que satisfazem *minsim*.

Relação de Similaridade	Item difuso	Grau de similaridade
sim(Tomate, Caqui)	*Tomate~Caqui*	0,7
sim(Tomate, Maçã)	*Tomate~Maçã*	0,9
sim(Caqui, Maçã)	*Caqui~Maçã*	0,75
sim(Peru, Frango)	*Peru~Frango*	0,85
sim(Tomate, Caqui, Maçã)	*Tomate~Caqui~Maçã*	0,7

No final dessa etapa temos todas as associações difusas que satisfazem *minsim*, podendo conter ou não ciclos de similaridade. Doravante, esses itens difusos podem ser considerados para a geração das regras de associação.

6.3.3. Geração de Candidatos

A maneira como os itemsets candidatos são gerados no algoritmo NARFO é similar à do Apriori. No entanto, no NARFO, além dos itens verificados na etapa de varredura da base de dados, os itens difusos, identificados na etapa de identificação de itens similares (seção 6.3.2) também integram os *itemsets* candidatos gerados.

Ao final dessa etapa têm-se p conjuntos de *itemsets* candidatos de tamanho *k*, os quais são submetidos à etapa de cálculo do peso dos candidatos (seção 6.3.4).

6.3.4. Cálculo do peso dos candidatos

Nessa etapa é feito o cálculo do peso de cada *itemset* candidato. Esse peso corresponde a cada ocorrência de um item na base de dados. No entanto, diferentemente do que ocorre no Apriori, na realização do algoritmo NARFO, assim como no XSSDM e SSDM, pode haver a presença de itens difusos em um *itemset*, e nesse caso, ele é chamado de *itemset* difuso. Os itens compondo um *itemset* difuso são muito similares, podendo ser considerados praticamente iguais. Desse modo, para um *itemset* difuso $item_1 \sim item_2$, por exemplo, a cada ocorrência encontrada de $item_1$ ou $item_2$ na base de dados, elas serão associadas e, juntamente com o valor do grau de similaridade entre os itens, comporão uma *ocorrência difusa* de $item_1 \sim item_2$, em oposição às ocorrências exatas encontradas por algoritmos convencionais.

Por conseguinte, é necessário verificar se um *itemset* é difuso ou não antes de

realizar sua pesagem. Caso o *itemset* não seja difuso, a pesagem é feita de maneira convencional, contando suas ocorrências exatas na base de dados. Caso contrário, se um *itemset* for difuso, o cálculo do peso deve considerar suas ocorrências difusas. Para entender como essas ocorrências acontecem, suponha que o grau de similaridade entre $item_1$ e $item_2$ seja igual a 0,8. Nesse caso, cada ocorrência de $item_1$ na base de dados pode ser considerada igual a 80% da ocorrência de $item_2$ na base. Assim, a cada ocorrência de $item_2$ na base soma-se uma ocorrência de $item_2$, e a cada ocorrência de $item_1$ soma-se 0,8 ocorrências de $item_2$ (figura 6.3 – situação A).

O problema também pode ser visto de maneira inversa, somando-se uma ocorrência exata de $item_1$ quando $item_1$ é encontrado, e somando-se 0,8 ocorrências de $item_1$ a cada ocorrência de $item_2$ (figura 6.3 – situação B). Para a situação A, as ocorrências difusas totalizam o valor de 2,6 (0,8 + 0,8 +1,0), enquanto as ocorrências difusas da situação B totalizam o valor de 2,8 (1,0 + 1,0 + 0,8). Desse modo, dependendo da escolha da situação, os resultados obtidos para os mesmos itens podem ser diferentes. Para evitar isso, é necessário balancear essa contagem.

Tid	*Ancestral*$_1$	
10	*item*$_1$	0,8
20	*item*$_1$	0,8
30	*item*$_2$	1,0

situação A

Tid	*Ancestral*$_1$	
10	*item*$_1$	1,0
20	*item*$_1$	1,0
30	*item*$_2$	0,8

situação B

Figura 6.3 - Ocorrências difusas.

Esse balanceamento pode ser feito da seguinte forma: considere peso($item_1$) como o número de ocorrências do $item_1$; peso($item_2$) o número de ocorrências do $item_2$; e sim($item_1$, $item_2$) o grau de similaridade entre $item_1$ e $item_2$. Assim, para as situações A e B da figura 6.3 o número de ocorrências de cada situação é dado, respectivamente, através das seguintes expressões:

$$peso(item_2) + peso(item_1) \text{ x } sim(item_1, item_2);$$
$$peso(item_1) + peso(item_2) \text{ x } sim(item_1, item_2).$$

Para o balanceamento da contagem das ocorrências difusas, adota-se a média aritmética entre as situações A e B, de acordo com a equação 6.1.

$$\frac{[peso(item_1) + peso(item_2)\ x\ sim(item_1, item_2)] + [peso(item_2) + peso(item_1)\ x\ sim(item_1, item_2)]}{2}$$

Equação 6.1 – Equação para balanceamento da contagem das ocorrências difusas entre as situações A e B.

Analisando-se as possibilidades de contagem das ocorrências, verifica-se também que o valor mínimo possível seria obtido somando o valor de similaridade entre os itens (0,8) a cada ocorrência de $item_1$ ou $item_2$ no exemplo da figura 6.2, o que corresponderia ao valor 2,4 (0,8 + 0,8 + 0,8). De modo semelhante, pode-se calcular o valor máximo possível, somando-se 1,0 a cada ocorrência de $item_1$ ou $item_2$, resultando no valor 3,0 para o peso (1,0 + 1,0 + 1,0). Considerando os valores extremos, o valor do peso difuso corresponde à média entre eles, conforme a equação 6.2.

$$\frac{[peso(item_1)\ x\ sim(item_1, item_2) + peso(item_2)\ x\ sim(item_1, item_2)] + [peso(item_1) + peso(item_2)]}{2}$$

Equação 6.2 – Equação para balanceamento da contagem das ocorrências difusas com os valores extremos.

Note que o valor obtido para a expressão acima é 2,7, o mesmo valor obtido através do cálculo da média aritmética dos valores das situações A e B (2,6 e 2.8), o que reforça a ideia de adoção dessa média para balancear a contagem das ocorrências nebulosas.

As expressões apresentadas acima são equivalentes, o que pode ser comprovado por simples manipulação algébrica. Usando esse mesmo artifício, chega-se a equação 6.3 abaixo (ESCOVAR; BIAJIZ; VIEIRA, 2005), criada para calcular o peso difuso entre dois itens.

$$Peso\ Difuso = \frac{[peso(item_1) + peso(item_2)][1 + sim(item_1, item_2)]}{2}$$

Equação 6.3 – Cálculo do *peso difuso* entre dois itens (ESCOVAR; BIAJIZ; VIEIRA, 2005).

No entanto, a equação para o cálculo do *peso difuso* apresentada contém apenas 2 itens similares. Para associações difusas envolvendo mais itens, é necessária uma generalização dessa equação. Com isso, a fórmula algébrica para o cálculo do *peso difuso* de associações difusas envolvendo *n* itens, $n \in \mathrm{N}$ e $n > 2$, é apresentada através da equação 6.4 (desenvolvida por (ESCOVAR; BIAJIZ; VIEIRA, 2005) e utilizada neste trabalho). O fator difuso *f* que aparece na equação do cálculo do *peso difuso* generalizado é obtido através da equação da figura 6.4, e o seu valor corresponde ao mínimo entre as similaridades envolvidas na associação difusa, conforme a teoria de intersecção de conjuntos difusos (seção 3.2).

$$Peso\ Difuso = \left[\sum_{i=1}^{n} peso(\text{item}_1)\right]\left[\frac{1+f}{2}\right]$$

Equação 6.4 - Equação *genérica* do *peso difuso* (ESCOVAR; BIAJIZ; VIEIRA, 2005).

$$f = min(sim(item_1, item_2), \ldots, sim(item_{n-1}, item_n))$$

Figura 6.4 - Cálculo do fator difuso *f*.

Explicada a maneira como é feito o cálculo do peso de cada *itemset* candidato, será dada sequência nas tarefas realizadas durante essa etapa.

A cada passo do algoritmo, a base de dados é percorrida, e cada uma de suas linhas é confrontada com o conjunto de *itemsets* candidatos. Se o *itemset* candidato não for difuso, a cada ocorrência do mesmo na base de dados, o seu peso é incrementado de 1, uma vez que sua ocorrência é exata (ocorre ou não a presença do *itemset*). Caso contrário, se o *itemset* em questão for difuso, todas suas ocorrências também serão, e o valor do incremento do peso do *itemset* difuso é calculado pela equação 6.4. Após a varredura na base de dados ter terminado, ou seja, tenha percorrido todas as linhas da base de dados, a pesagem dos *itemsets* candidatos está completa, e o algoritmo passa para a próxima etapa (avaliação dos candidatos).

6.3.5. Avaliação dos candidatos

Nessa etapa se encontra uma das principais contribuições deste trabalho, que é a generalização de *itemsets* não frequentes baseado na teoria de *Itemsets Frequentes Máximos*, descrita na seção 2.3. Essa etapa é bem similar à etapa correspondente do algoritmo Apriori, na qual o suporte de cada *itemset* candidato é avaliado. O valor do suporte de cada *itemset* corresponde ao peso dividido pelo número de linhas da base de dados (total de transações). Caso o *itemset* seja difuso, seu peso também o será, e o cálculo do suporte é feito do mesmo modo. Portanto, o cálculo do suporte, difuso ou não, é calculado através da equação 6.5.

$$Suporte = \frac{peso(itemset)}{número\ de\ linhas\ na\ base\ de\ dados}$$

Equação 6.5 – Cálculo do suporte de um *itemset*.

Nos algoritmos convencionais como o Apriori, se o valor do suporte de um *itemset* é maior ou igual ao *minsup*, esse *itemset* é considerado frequente, e é inserido junto ao conjunto de *itemsets* frequentes. No entanto, se o valor do suporte for menor que o *minsup*, esse *itemset* não é considerado frequente, sendo descartado.

No algoritmo NARFO, caso o suporte de um *itemset* seja maior ou igual ao *minsup*, ele é considerado um *itemset* frequente e é adicionado ao conjunto de *itemsets* frequentes (assim como no Apriori). Entretanto, caso o valor do suporte de um *itemset* seja menor que *minsup*, ele não é simplesmente descartado. Inicialmente, esse *itemset* é inserido em um conjunto de *itemsets* não frequentes.

Após o algoritmo ter verificado se os *itemsets* candidatos são frequentes ou não, o algoritmo avalia o conjunto de *itemsets* não frequentes. Para cada ancestral contido na ontologia, o algoritmo verifica se todos seus descendentes imediatos (filhos) estão contidos no conjunto de *itemsets* não frequentes. A cada descendente encontrado do respectivo ancestral, o suporte do ancestral (que inicialmente está em zero) é incrementado com o valor do suporte do descendente. Se todos os descendentes do respectivo ancestral estiverem nesse conjunto e se o valor do suporte do ancestral resultar em um valor maior ou igual ao *minsup*, é realizada a generalização desses descendentes para o ancestral em questão e o *itemset* generalizado é adicionado ao conjunto de *itemsets* frequentes.

O pseudoalgoritmo para generalização de *itemsets* não frequentes a partir do conjunto de *itemsets* não frequentes é apresentado na tabela 6.3. No pseudoalgoritmo da tabela 6.3, supAnc é uma variável para o suporte do ancestral e contDesc é uma variável que conta o número de descendentes de um determinado ancestral.

Tabela 6.3 - Pseudoalgoritmo para generalização de *itemsets* não frequentes.

0	Para cada ancestral na ontologia difusa faça
1	real supAnc = 0
2	inteiro contDesc = 0
3	Para cada *itemset* não frequente de tamanho k
4	Se *itemset* pertence ao ancestral
5	incremente supAnc com o suporte do *itemset*
6	contDesc++
7	Fim_se
8	Fim_para
9	Se contDesc == número de filhos do ancestral && supAnc >= *minsup*
10	Insere ancestral no conjunto de *itemsets* frequentes
11	//generalização
12	Fim_se
13	Fim_para

Essa generalização representa um conhecimento significante e relevante que não é descoberto pelo Apriori, XSSDM e vários outros algoritmos de mineração de regras de associação, uma vez que eles não fazem a generalização de *itemsets* não frequentes. Essa é uma das principais contribuições desse trabalho.

No final dessa etapa, temos todos os *itemsets* candidatos de tamanho k.

6.3.6. Geração das Regras

As regras de associação são constituídas por antecedentes (itens à esquerda da seta) e consequentes (itens à direita da seta), conforme a figura 6.5.

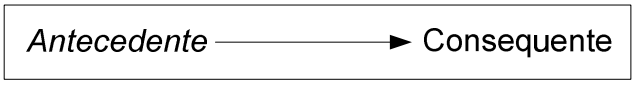


Figura 6.5 - Antecedente e Consequente da regra.

A etapa de geração de regras consiste em gerar todas as possibilidades de antecedentes e consequentes para cada *itemset* pertencente ao conjunto de *itemsets* frequentes. Se o valor da confiança da regra de associação gerada for maior ou igual à *minconf*, essa regra é considerada válida.

Após a geração de todas as regras, o algoritmo checa se as regras válidas contêm itens difusos para verificar a possibilidade de generalizá-los. Um item difuso pode ser generalizado se ele contém todos os descendentes de um mesmo ancestral, de acordo com a

ontologia difusa. Por exemplo, considere que a regra *Tomate~Caqui~Maçã* → *Peru* tenha sido gerada. O algoritmo generaliza esse item difuso para o ancestral *Fruta*, uma vez que todos os descendentes estão contidos no item difuso. Esse processo é idêntico ao desenvolvido em (ESCOVAR; YAGUINUMA; BIAJIZ, 2006). Note que todos os itens do ancestral devem estar presentes no item difuso. Caso tivéssemos uma regra como *Repolho* → *Peru~Frango*, por exemplo, o item difuso não seria generalizado para *Carne*, pois não estão contidos todos os descendentes de *Carne* no item difuso.

Ao final dessa etapa, temos todas as regras geradas. Essas são analisadas pelo algoritmo NARFO, na etapa da seção 6.3.7, para verificar outras generalizações de acordo com o parâmetro *mingen* e , na seção 6.3.8, para verificar a presença de redundâncias.

6.3.7. Tratamento de Generalização das Regras

Após a geração de todas as regras de associação pela etapa anterior, elas recebem um tratamento de generalização e redundância. Além da generalização citada na seção anterior, outros meios de generalização, que não ocorrem no XSSDM, são considerados. Nesse momento, o processo de generalização ocorre não somente se todos os descendentes de um ancestral estão contidos em um item difuso, mas também se todos os descendentes de um ancestral estão em regras diferentes que possuem os mesmos antecedentes e consequentes, exceto pelo descendente em questão.

Por exemplo, caso o algoritmo gere as seguintes regras:

Caqui → Salsicha
Maçã → Salsicha
Tomate → Salsicha.

Temos que todos os descendentes de *Fruta* apareceram em regras distintas que têm os mesmos antecedentes e consequentes, exceto pelo item descendente do ancestral *Fruta*. Desse modo, o tratamento de generalização aplicado pelo algoritmo NARFO realiza a generalização dos descendentes para o ancestral *Fruta*, resultando em uma única regra, como pode ser observado a seguir:

Fruta → *Salsicha* .

Além disso, o algoritmo NARFO também considera o parâmetro de generalização *mingen*. Esse parâmetro diz respeito à porcentagem mínima de descendentes de um ancestral que devem aparecer em regras diferentes contendo os mesmos antecedentes e consequentes, exceto pelo descendente, para que a generalização possa ser feita. Considere o valor do *mingen* sendo como 0,6 (60%) e as seguintes regras:

Caqui~Maçã → Salsicha
Tomate → Peru~Frango
Repolho → Peru~Frango.

Note que a primeira regra possui um item difuso no antecedente. Esse item contém dois descendentes do ancestral *Fruta* e, como os itens são suficientemente similares, podem ser considerados iguais. Desse modo, temos dois itens do ancestral *Fruta*, o que corresponde a, aproximadamente, 66,67% dos descendentes de *Fruta*. A regra pode, então, ser generalizada para a regra *Fruta → Salsicha*.

Considerando as duas últimas regras, duas generalizações podem ser realizadas (uma no antecedente, e outra no consequente da regra). *Tomate* e *Repolho* constituem 100% dos descendentes de *Vegetais*, estão em regras distintas com os mesmos antecedentes e consequentes, menos o descendente. Desse modo é feita a generalização dos descendentes para *Vegetais*.

Analogamente ao ocorrido com a primeira regra, também ocorre com as duas últimas: os itens *Peru* e *Frango* correspondem a, aproximadamente, 66,67% dos descendentes de *Carne*, podendo ser generalizados. Assim, considerando as duas generalizações possíveis, as duas regras de associação geradas resultam em uma única regra

Vegetais → Carne,

tornando o resultado mais compreensível para o usuário e auxiliando a diminuir o número de regras geradas.

Porém, caso a porcentagem mínima de generalização tenha sido atingida e a generalização tenha sido feita, o algoritmo NARFO verifica se todos os descendentes do ancestral generalizado fizeram parte da generalização, ou se algum dos descendentes não esteve em regras que originaram a generalização. Nesse último caso, o algoritmo coloca entre parênteses, na frente da regra no momento de sua apresentação ao usuário, qual (is)

descendente (s) não originou-(aram) a regra. Isso ocorre para o usuário não obter nenhuma informação equivocada. A última regra do exemplo apresentado seria mostrada para o usuário na seguinte forma:

Vegetais → *Carne* suporte = 0,7 e confiança = 0,85 (Ancestral *Carne* generalizado, exceto pelo descendente *Salsicha*).

O pseudoalgoritmo do tratamento de generalização é apresentado na tabela 6.4.

Tabela 6.4 - Pseudoalgoritmo do tratamento de generalização.

1	`Vetor v // vetor de regras de associação`
2	`Para cada regra gerada`
3	`   Se regra pode ser generalizada considerando` *`mingen`*
	`   // generalização é feita tanto na antecedente quanto no consequente`
4	`      generaliza a regra`
5	`   Fim_se`
6	`   Adiciona regra em v`
7	`Fim_para`

6.3.8. Tratamento de Redundância das Regras

Depois de realizada a generalização das regras, o algoritmo NARFO trata dois tipos de redundância a partir das regras geradas. O primeiro tipo de redundância tratado é quando uma regra de associação generalizada é uma sub-regra de outra regra de associação gerada. Uma regra r_1 é sub-regra de outra regra de associação r_2 se ambas as regras possuem os mesmos itens no lado do antecedente e do consequente, exceto por r_1 conter pelo menos um item que é descendente de algum item em r_2, no mesmo lado da regra (antecedente ou consequente). Desse modo, a regra r_1 é excluída do conjunto de regras de associação válidas que serão mostradas para o usuário final.

Como exemplo desse tipo de redundância, considere que fossem geradas as regras:

Caqui → *Salsicha*

Fruta → *Salsicha.*

A primeira regra é eliminada, uma vez que o item *Caqui* é descendente imediato do ancestral *Fruta*.

A outra redundância tratada seria a redundância difusa. Esse tipo de redundância ocorre quando uma regra de associação é sub-regra de outra regra de associação difusa. Uma sub-regra de associação difusa r_1 é uma regra que possui os mesmos itens de outra regra r_2(essa é obrigatoriamente difusa), no lado do antecedente e do consequente, exceto por r_1 possuir pelo menos um item que está contido em um item difuso, no mesmo lado da regra, em r_2. Por exemplo, considere que o algoritmo gere as regras de associação

Maçã → *Salsicha*

Maçã~Tomate → *Salsicha*.

Note que as regras acima são muito semelhantes, a não ser pelo item Maçã, contido no antecedente da primeira regra, estar contido no item difuso *Maçã~Tomate*, no antecedente da segunda regra. A primeira regra é, então, eliminada do conjunto final de regras. No entanto, a segunda regra somente existe pela presença da similaridade entre os itens e porque os itens que compõem o item difuso (*Maçã* e *Tomate*) são suficientemente similares. Caso não houvesse a presença da similaridade entre os itens, uma regra como

Tomate → *Salsicha*

não seria gerada, provavelmente, e ter-se-ia apenas a regra

Maçã → *Salsicha*.

Como temos a presença da similaridade entre os itens e a primeira regra foi eliminada, já que é uma sub-regra da segunda, o algoritmo NARFO, ao mostrar as regras para o usuário final, identifica qual (is) item (ns) da regra difusa é (são) mais relevante (s). Para o exemplo acima considerado, a impressão da regra indicaria que o item *Maçã* é mais relevante no item difuso, pois teve um sub-regra, contendo esse item, excluída:

Maçã~Tomate → *Salsicha* suporte = 0,56, confiança = 0,7, possuindo o item *Maçã* com maior relevância.

O pseudoalgoritmo do tratamento de redundância é apresentado na tabela 6.5.

Tabela 6.5 - Pseudoalgoritmo do tratamento de redundância.

1	`Vetor v1,v2 // vetores de regras de associação`
2	`Booleano verifica = falso // verifica sub-regras`
3	`Para cada regra em v // vetor da tabela 6.4`
4	`   verifica = sub-regra(regra,v)// verifica se é uma sub-regra em v`
5	`   Se verifica == falso`
6	`      Adiciona regra em v1`
7	`   Fim_se`
8	`Fim_para`
9	`Para cada regra em v1`
10	`   verifica = sub-regraDifusa(regra,v1)`
11	`   // verifica se é uma sub-regra de uma regra difusa em v1`
12	`   Se verifica == falso`
13	`      Adiciona regra em v2`
14	`   Fim_se`
15	`Fim_para`

Considerando a tabela 6.5, têm-se em v2 (vetor de regras de associação) as regras de associação generalizadas e difusas não redundantes que serão apresentadas ao usuário. No próximo capítulo são apresentados os experimentos realizados.

7. EXPERIMENTOS

7.1. CONSIDERAÇÕES INICIAIS

Conforme apresentado no capítulo 6, o algoritmo NARFO realiza a mineração de regras de associação generalizadas e não redundantes baseada em uma ontologia difusa, com o objetivo de obter regras de associação que sejam relevantes, mais compreensíveis e que não possuam o mesmo significado entre elas.

Para desenvolver o algoritmo, foi utilizada a tecnologia Java, e o framework *Jena* (CARROL, *et al*., 2004) para desenvolvimento de aplicações baseadas em ontologias. As inferências sobre a ontologia difusa são realizadas pelo raciocinador de *Jena*, que oferece suporte a ontologias em OWL DL. O framework *Jena* também serviu de base para a implementação dos mecanismos de inferência sobre conceitos e relacionamentos difusos contidos na ontologia difusa.

A seção a seguir contém os testes realizados e os resultados obtidos.

7.2. TESTES E RESULTADOS

Para a realização dos testes, foram considerados dados reais do Censo Demográfico Brasileiro de 2000, fornecidos pelo IBGE (Instituto Brasileiro de Geografia e Estatística). Quatro conjuntos de dados foram analisados:

- IBGE1, contendo informações sobre *Sexo*, *Anos de estudo* e *Raça ou etnia*;
- IBGE2, envolvendo relações entre *Anos de Estudo* e *Estado Civil*;
- IBGE3, tendo relações de *Raça ou etnia* e *grupos de idade*;
- IBGE4, relacionando *Local de Moradia* com *Raça ou etnia*.

Antes de realizar a mineração de regras de associação foi criada uma ontologia difusa para fornecer as relações de similaridade semântica entre os itens. Após a análise dos três conjuntos de dados, a ontologia difusa, modelada com *Protégé OWL* (KNUBLAUCH, et al., 2004) e representada na figura 7.1, foi criada. Note que a ontologia em questão não

contém todos os itens (filhos) dos conjuntos de dados, sendo uma representação genérica da ontologia difusa completa. As figuras 7.2, 7.3 e 7.4 representam partes da ontologia difusa da figura 7.1. Essa divisão foi feita para uma melhor compreensão de como cada conjunto de dados é relacionado com a ontologia. Assim, a figura 7.2 contém a representação de todos os itens do conjunto de dados IBGE1 e a hierarquia em que o mesmo está contido. As figuras 7.3, 7.4 e 7.5 são representações dos conjuntos de dados IBGE2, IBGE3 e IBGE4, respectivamente.

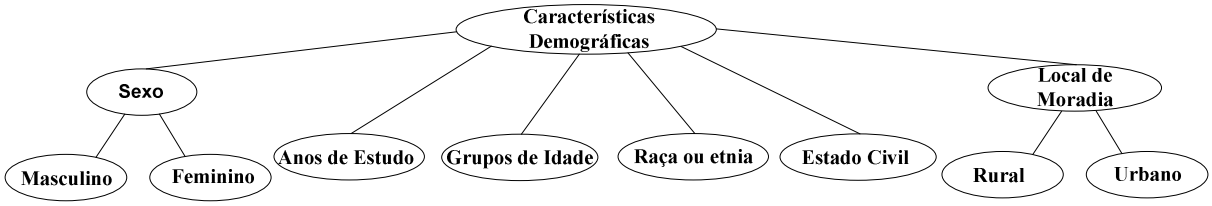


Figura 7.1 - Representação da ontologia difusa de características demográficas.

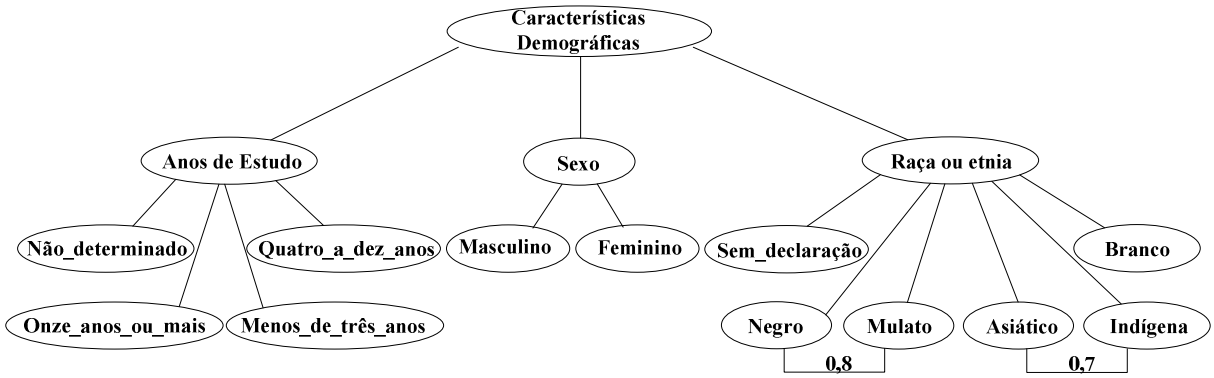


Figura 7.2 - Representação completa da ontologia difusa para o conjunto de dados IBGE1.

Na figura 7.2 tem-se a representação completa da ontologia difusa da figura 7.1 para *Anos de Estudo* (composta pelos itens *Não_determinado*, *Menos_de_três*_anos, *Quatro_a_dez_anos* e *Onze_anos_ou_mais*), *Sexo* (itens *Masculino* e *Feminino*) e *Raça ou etnia* (itens *Sem_declaração*, *Negro*, *Mulato*, *Asiático*, *Indígena* e *Branco*), incluindo os relacionamentos de similaridade semântica entre os itens. Note que *Anos de Estudo*, *Sexo* e *Raça ou etnia* correspondem aos ancestrais, e as folhas da figura 7.2 são os itens que compõem o conjunto de dados. De forma análoga, o mesmo procedimento é feito para a figura 7.3 (representação completa da ontologia difusa da figura 7.1 para *Anos de Estudo* e *Estado Civil*), para a figura 7.4 (representação completa da ontologia difusa da figura 7.1 para

Grupos de idade e *Raça ou etnia*) e para a figura 7.5 (representação completa da ontologia difusa da figura 7.1 para *Local de Moradia* e *Raça ou etnia*).

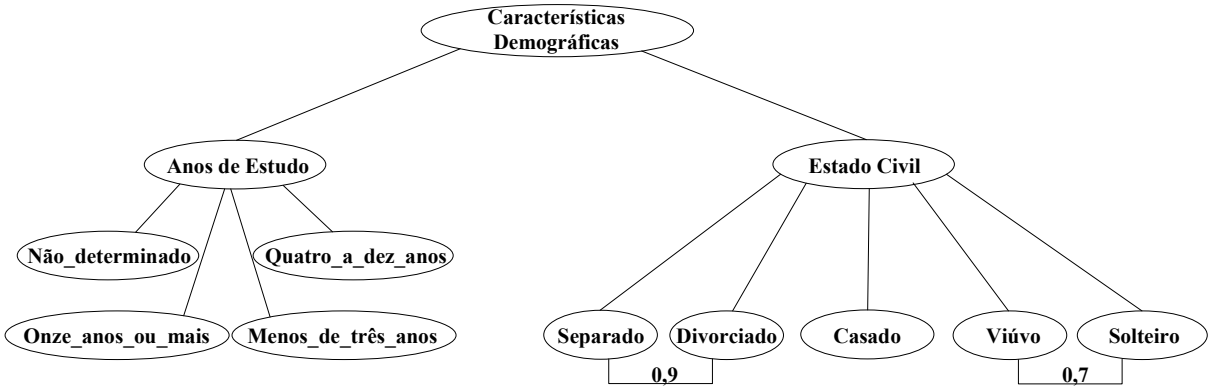


Figura 7.3 – Representação completa da ontologia difusa para o conjunto de dados IBGE2.

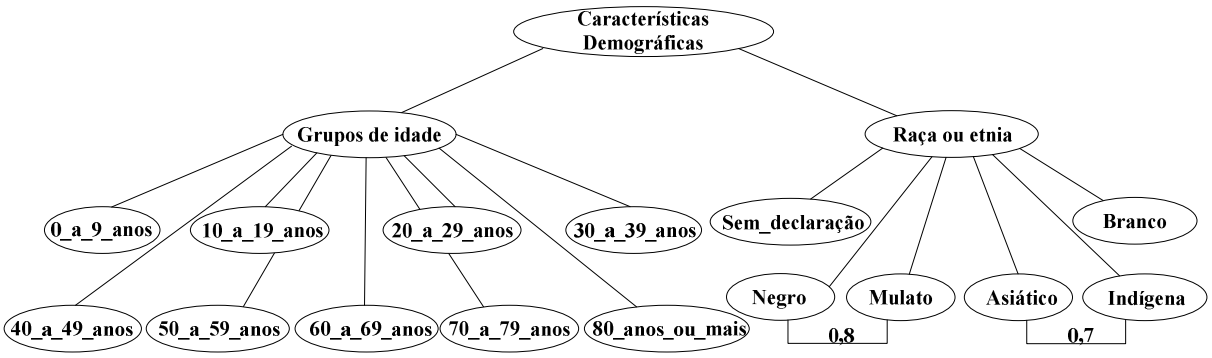


Figura 7.4 - Representação completa da ontologia difusa para o conjunto de dados IBGE3.

Todos os testes realizados com algoritmo NARFO também foram aplicados ao algoritmo XSSDM, que também realiza mineração de regras de associação generalizadas baseada em uma ontologia difusa, para comprovar a eficiência e melhora que o NARFO possui em comparação ao XSSDM. Para cada conjunto de dados especificado acima serão realizadas duas séries de testes. A primeira série de testes é realizada com os seguintes valores:

- *minsup* variando entre 0,5 e 0,05, reduzindo o valor do *minsup* em 0,05 a cada teste realizado;
- *minconf* fixo em 0,2;
- *minsim* com valor igual a 0,6;
- *mingen* igual a 1.

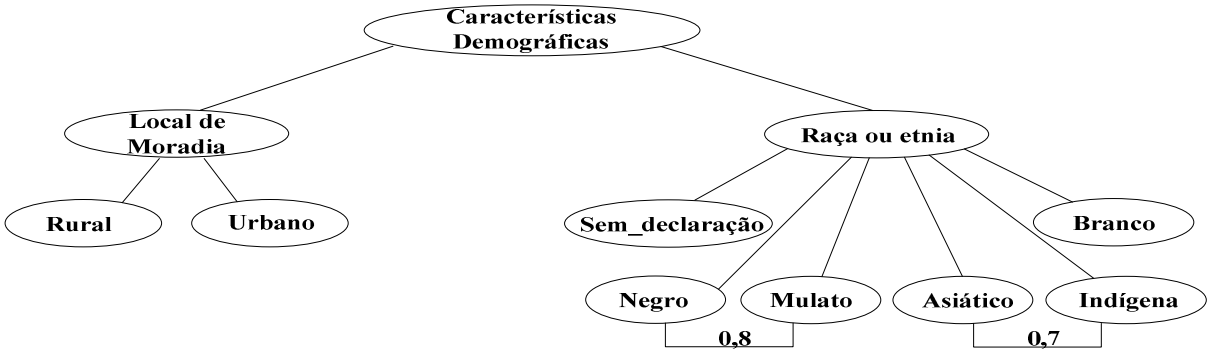


Figura 7.5 - Representação completa da ontologia difusa para o conjunto de dados IBGE4.

A segunda série de testes foi realizada com a aplicação dos mesmos valores de *minsup*, *minconf* e *minsim*, porém o valor de *mingen* é igual a 0,65. As figuras 7.5 e 7.6 mostram os resultados obtidos ao aplicar as duas séries de testes acima com o conjunto de dados IBGE1.

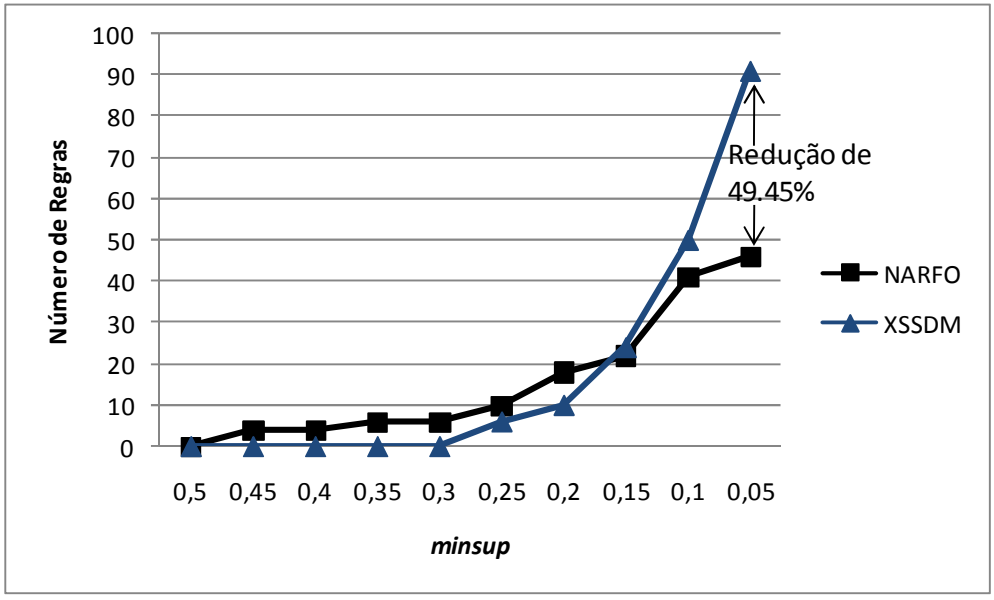


Figura 7.6 - Primeira série de testes para o conjunto de dados IBGE1.

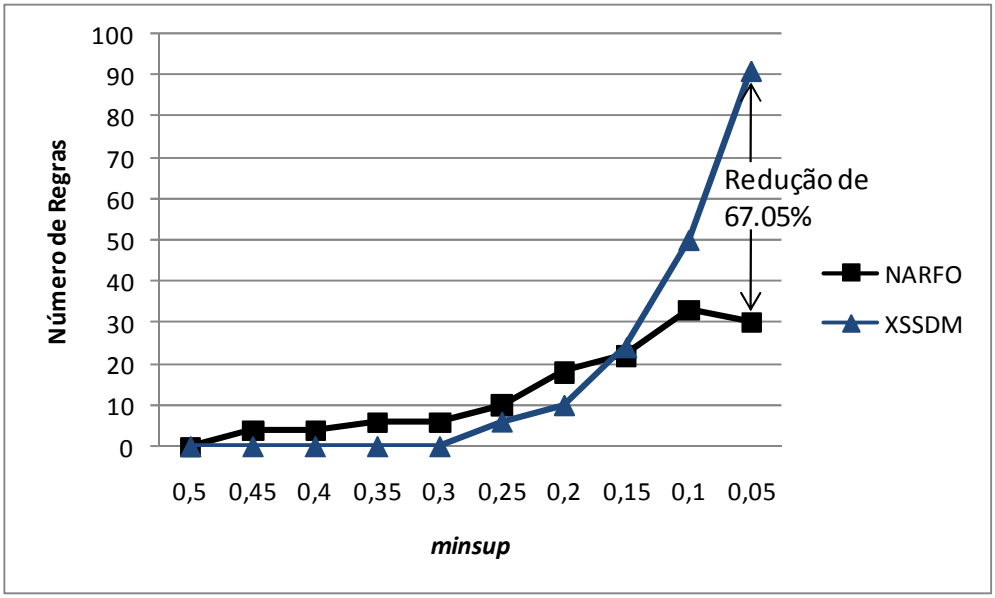


Figura 7.7 - Segunda série de testes para o conjunto de dados IBGE1.

Note que, em ambos os gráficos acima, o algoritmo NARFO produz um número maior de regras de associação, para valores de *minsup* entre 0,5 e 0,2, em comparação com o algoritmo XSSDM. Isso ocorre devido à implementação da generalização de *itemsets* não frequentes (seção 6.3.5) no algoritmo NARFO, resultando em um conhecimento significativo e interessante que não é obtido pelo XSSDM. Conforme o valor do suporte mínimo é decrementado em 0,05, a quantidade de regras de associação geradas pelo algoritmo NARFO é menor do que a quantidade gerada pelo XSSDM, situação ocorrida com valores de suporte mínimo entre 0,15 e 0,05. Isso é devido ao tratamento de generalização das regras e do tratamento de redundância (seções 6.3.7 e 6.3.8) aplicado pelo algoritmo NARFO. Comparando somente os resultados obtidos pelo algoritmo NARFO nas duas séries de testes para o conjunto de dados IBGE1, é possível observar que, para a segunda série de testes, NARFO gera um número menor de regras de associação quando o valor do *minsup* varia entre 0,15 e 0,05. Isso ocorre devido ao parâmetro de porcentagem mínima de generalização (*mingen*) utilizado na segunda série de testes ser menor do que o da primeira. Desse modo, um número menor de descendentes de um mesmo ancestral deve estar em regras idênticas, que se diferem pelos descendentes. As tabelas 7.1 e 7.2 mostram as regras obtidas para os algoritmos NARFO e XSSDM, respectivamente, ao aplicar a segunda bateria de testes para o conjunto de dados IBGE1, com o valor de *minsup* igual a 0,25. Note que o NARFO gera mais regras devido à produção de regras resultantes da generalização de *itemsets* não frequentes.

Tabela 7.1 – Regras geradas pelo algoritmo NARFO com o conjunto de dados IBGE1 e suporte 0,25 com a segunda bateria de testes.

	Regras geradas
1	Sexo->Branco suporte=0.54906476, confiança=0.54906476
2	Branco->Sexo suporte=0.54906476, confiança=1.0
3	Quatro_a_dez_anos->Branco suporte=0.2706812, confiança=0.554281
4	Branco->Quatro_a_dez_anos suporte=0.2706812, confiança=0.492986
5	Menos_de_três_anos->Sexo suporte=0.27778333, confiança=1.0
6	Sexo->Menos_de_três_anos suporte=0.27778333, confiança=0.27778333
7	Quatro_a_dez_anos->Sexo suporte=0.48834652, confiança=1.0
8	Sexo-> Quatro_a_dez_anos suporte=0.48834652, confiança=0.48834652
9	Mulato~Negro->Sexo suporte=0.39197758, confiança=1.0, possuindo item 'Mulato' com maior relevância!!!
10	Sexo->Mulato~Negro suporte=0.39197758, confiança=0.39197758, possuindo item 'Mulato' com maior relevância!!!

Tabela 7.2 - Regras geradas pelo algoritmo XSSDM com o conjunto de dados IBGE1 e suporte 0,25 com a segunda bateria de testes.

	Regras geradas
1	Feminino->Branco suporte=0.28978693, confiança=0.56350905
2	Branco->Feminino suporte=0.28978693, confiança=0.52778286
3	Masculino->Branco suporte=0.2592778, confiança=0.53377265
4	Branco->Masculino suporte=0.2592778, confiança=0.4722172
5	Quatro_a_dez_anos->Branco suporte=0.2706812, confiança=0.554281
6	Branco->Quatro_a_dez_anos suporte=0.2706812, confiança=0.492986

A mesma análise foi feita com os outros três conjuntos de dados. Aplicando as duas séries de testes ao conjunto de dados IBGE2, os resultados apresentados nos gráficos das figuras 7.8 e 7.9 são obtidos.

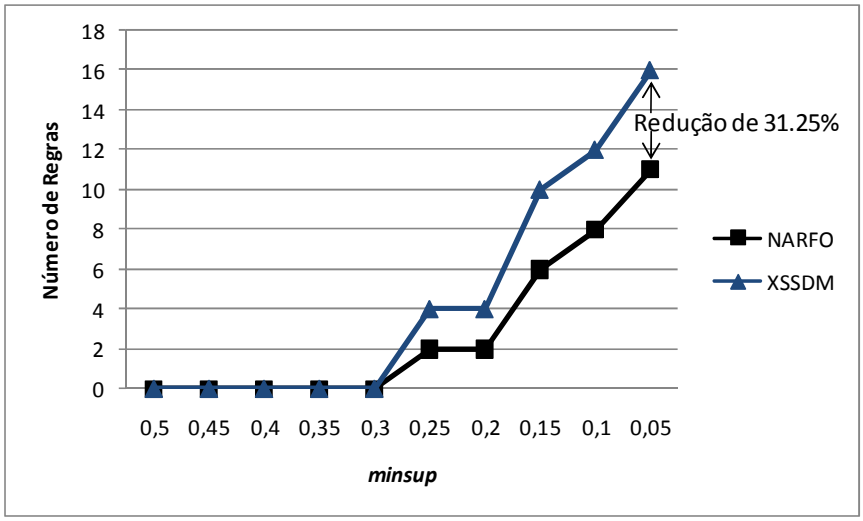


Figura 7.8 - Primeira série de testes para o conjunto de dados IBGE2.

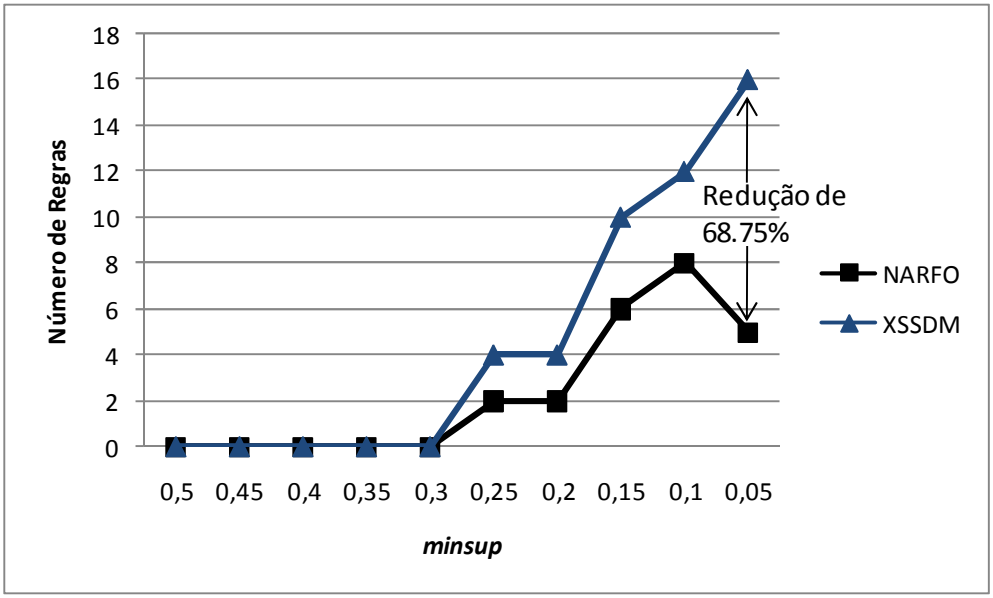


Figura 7.9 - Segunda série de testes para o conjunto de dados IBGE2.

Observando as figuras 7.8 e 7.9, é possível notar que para ambas as series de testes o algoritmo NARFO gera um menor número de regras de associação, em comparação com o algoritmo XSSDM, para todos os valores em que o suporte mínimo gera regras (*minsup* entre 0,3 e 0,05). Isso ocorre devido ao tratamento de redundância e de generalização que é aplicado no NARFO. Comparando somente os resultados obtidos pelo algoritmo NARFO para essas figuras, nota-se que a figura 7.9 possui um menor número de regras de associação que a figura 7.8 quando o valor do suporte mínimo está entre 0,1 e 0,05, uma vez que o valor mínimo para generalização é menor (0,65 para a figura 7.9 e 1 para a figura 7.8). Desse modo, algumas das regras geradas pelo NARFO ao se aplicar a primeira série de testes são generalizadas, o que reduz o número de regras de associação resultantes. As tabelas 7.3 e 7.4 apresentam as regras geradas pelos algoritmos aplicando a segunda bateria de testes com suporte mínimo de 0,05 no conjunto de dados IBGE2. NARFO gera um menor número de regras em função do tratamento de generalização e redundância, e em função do parâmetro *mingen* (valor de 0,65).

Tabela 7.3 - Regras geradas pelo algoritmo NARFO com o conjunto de dados IBGE2 e suporte 0,05 com a segunda bateria de testes.

	Regras geradas
1	Anos de Estudo->Casado suporte=0.36737347, confiança=0.37063572 (Ancestral *Anos de Estudo* generalizado, exceto pelo descendente *Não determinado*)
2	Casado->Anos de Estudo suporte=0.36737347, confiança=0.9916306 (Ancestral *Anos de Estudo* generalizado, exceto pelo descendente *Não determinado*)
3	Anos de Estudo->Viúvo~Solteiro suporte=0.49964494, confiança=0.5040817 (Ancestral *Anos de Estudo* generalizado, exceto pelo descendente *Não determinado*), possuindo item 'Solteiro' com maior relevância!!!
4	Viúvo~Solteiro->Menos_de_três_anos suporte=0.15609123, confiança=0.3095078, possuindo item 'Solteiro' com maior relevância!!!
5	Viúvo~Solteiro->Quatro_a_dez_anos suporte=0.26338267, confiança=0.5222522, possuindo item 'Solteiro' com maior relevância!!!

Tabela 7.4 - Regras geradas pelo algoritmo XSSDM com o conjunto de dados IBGE2 e suporte 0,05 com a segunda bateria de testes.

	Regras geradas
1	Onze_anos_ou_mais->Casado suporte=0.09011802, confiança=0.45922527
2	Casado->Onze_anos_ou_mais suporte=0.09011802, confiança=0.24325053
3	Menos_de_três_anos->Casado suporte=0.10702141, confiança=0.35869932
4	Casado->Menos_de_três_anos suporte=0.10702141, confiança=0.2888769
5	Quatro_a_dez_anos->Casado suporte=0.17023404, confiança=0.34279957
6	Casado->Quatro_a_dez_anos suporte=0.17023404, confiança=0.4595032
7	Onze_anos_ou_mais->Solteiro suporte=0.09011802, confiança=0.45922527
8	Menos_de_três_anos->Solteiro suporte=0.15853171, confiança=0.5313443
9	Solteiro->Menos_de_três_anos suporte=0.15853171, confiança=0.28939202
10	Quatro_a_dez_anos->Solteiro suporte=0.2940588, confiança=0.59214497
11	Solteiro->Quatro_a_dez_anos suporte=0.2940588, confiança=0.5367902
12	Onze_anos_ou_mais->Viúvo~Solteiro suporte=0.08017104, confiança=0.40853724
13	Menos_de_três_anos->Viúvo~Solteiro suporte=0.15609123, confiança=0.52316463
14	Viúvo~Solteiro->Menos_de_três_anos suporte=0.15609123, confiança=0.3095078
15	Quatro_a_dez_anos->Viúvo~Solteiro suporte=0.26338267, confiança=0.5303726
16	Viúvo~Solteiro->Quatro_a_dez_anos suporte=0.26338267, confiança=0.5222522

As duas séries de testes aplicadas com o conjunto de dados IBGE3 obtiveram os mesmos resultados. Embora na segunda bateria de testes o valor mínimo de generalização do algoritmo NARFO seja menor, não é feita nenhuma generalização a mais em virtude disso.

No entanto, quando comparamos os resultados alcançados com o algoritmo XSSDM, assim como nos testes anteriores, também ocorre diminuição do número de regras

de associação geradas sem a perda de informação.

Os resultados são idênticos, ao aplicar as duas séries de testes utilizando ambos os algoritmos, portanto são apresentados em uma única figura (figura 7.10). As tabelas 7.5 e 7.6 mostram as regras geradas por ambos os algoritmos quando o valor de *minsup* é igual a 0,05.

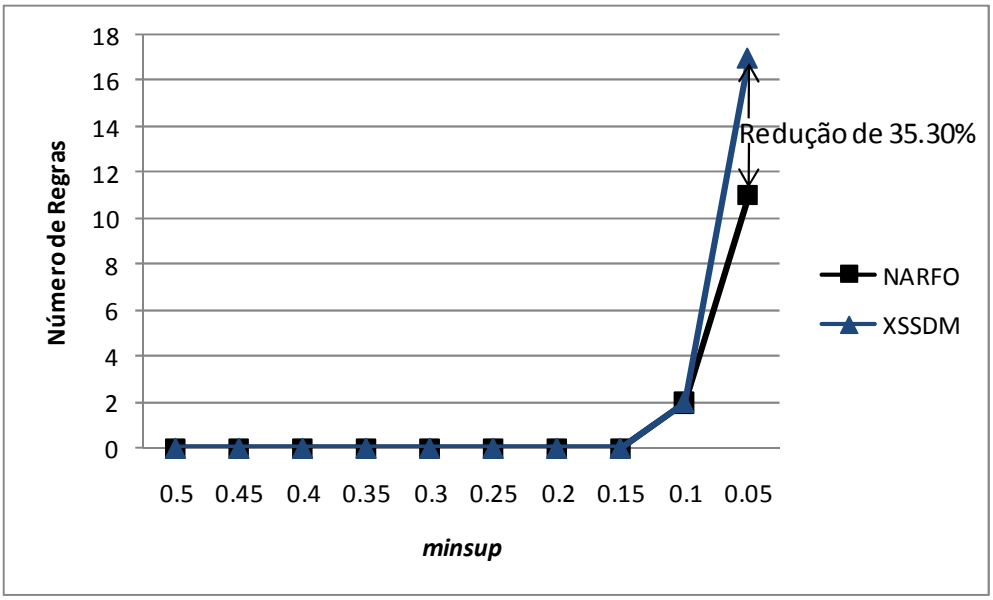


Figura 7.10 - Primeira e segunda série de testes para o conjunto de dados IBGE3.

Tabela 7.5 - Regras geradas pelo algoritmo NARFO com o conjunto de dados IBGE3 e suporte 0,05 com ambas as baterias de testes.

	Regras geradas
1	10_a_19_anos->Branco suporte=0.10317222, confiança=0.49615017
2	Mulato~Negro->10_a_19_anos suporte=0.09132393, confiança=0.22725236, possuindo item 'Mulato' com maior relevância!!!
3	10_a_19_anos->Mulato~Negro suporte=0.09132393, confiança=0.4391723, possuindo item 'Mulato' com maior relevância!!!
4	30_a_39_anos->Branco suporte=0.0818573, confiança=0.5501009
5	30_a_39_anos->Mulato~Negro suporte=0.058360852, confiança=0.39219907, possuindo item 'Mulato' com maior relevância!!!
6	0_a_9_anos->Branco suporte=0.10097068, confiança=0.52063984
7	Mulato~Negro->0_a_9_anos suporte=0.08078655, confiança=0.20103094 possuindo item 'Mulato' com maior relevância!!!
8	0_a_9_anos->Mulato~Negro suporte=0.08078655, confiança=0.41656345 possuindo item 'Mulato' com maior relevância!!!
9	40_a_49_anos->Branco suporte=0.06464525, confiança=0.5696649
10	20_a_29_anos->Branco suporte=0.09246472, confiança=0.5235127
11	20_a_29_anos->Mulato~Negro suporte=0.073401384, confiança=0.41558075, possuindo item 'Mulato' com maior relevância!!!

Tabela 7.6- Regras geradas pelo algoritmo XSSDM com o conjunto de dados IBGE3 e suporte 0,05 com ambas as baterias de testes.

	Regras geradas
1	Mulato->10_a_19_anos suporte=0.08906234, confiança=0.2317105
2	10_a_19_anos->Mulato suporte=0.08906234, confiança=0.42829645
3	10_a_19_anos->Branco suporte=0.10317222, confiança=0.49615017
4	Mulato~Negro->10_a_19_anos suporte=0.09132393, confiança=0.22725236
5	10_a_19_anos->Mulato~Negro suporte=0.09132393, confiança=0.4391723
6	30_a_39_anos->Mulato suporte=0.055038527, confiança=0.36987224
7	30_a_39_anos->Branco suporte=0.0818573, confiança=0.5501009
8	30_a_39_anos->Mulato~Negro suporte=0.058360852, confiança=0.39219907
9	Branco->0_a_9_anos suporte=0.08035625, confiança=0.20906015
10	0_a_9_anos->Mulato suporte=0.08035625, confiança=0.4143447
11	0_a_9_anos->Branco suporte=0.10097068, confiança=0.52063984
12	Mulato~Negro->0_a_9_anos suporte=0.08078655, confiança=0.20103094
13	0_a_9_anos->Mulato~Negro suporte=0.08078655, confiança=0.41656345
14	40_a_49_anos->Branco suporte=0.06464525, confiança=0.5696649
15	20_a_29_anos->Mulato suporte=0.069948964, confiança=0.39603397
16	20_a_29_anos->Branco suporte=0.09246472, confiança=0.5235127
17	20_a_29_anos->Mulato~Negro suporte=0.073401384, confiança=0.41558075

A figura 7.11 ilustra os resultados obtidos ao aplicar as duas séries de testes ao conjunto de dados IBGE4, e as tabelas 7.7 e 7.8 mostram as regras geradas pelos dois algoritmos com valor de suporte mínimo 0,2.

Semelhante ao ocorrido com os testes realizados com a base de dados IBGE3, ambas as séries obtêm o mesmo resultado. Desse modo, os resultados por elas encontrados são apresentados em uma única figura (figura 7.11).

Para essa base de dados, o algoritmo NARFO não gera nenhuma regra de associação resultante da generalização de *itemsets* não frequentes. Portanto, o algoritmo NARFO gera o mesmo número de regras de associação que o XSSDM quando o valor de *minsup* está entre 0,5 e 0,3. Isso ocorre, pois para esse intervalo, nenhuma redundância é encontrada assim como nenhuma generalização é feita. Os tratamentos de generalização e redundância contidos no algoritmo NARFO têm efeito quando *minsup* está entre 0,3 e 0,05, resultando em todo esse intervalo um menor número de regras de associação, obtendo até 63.64% de redução das regras quando comparado ao XSSDM.

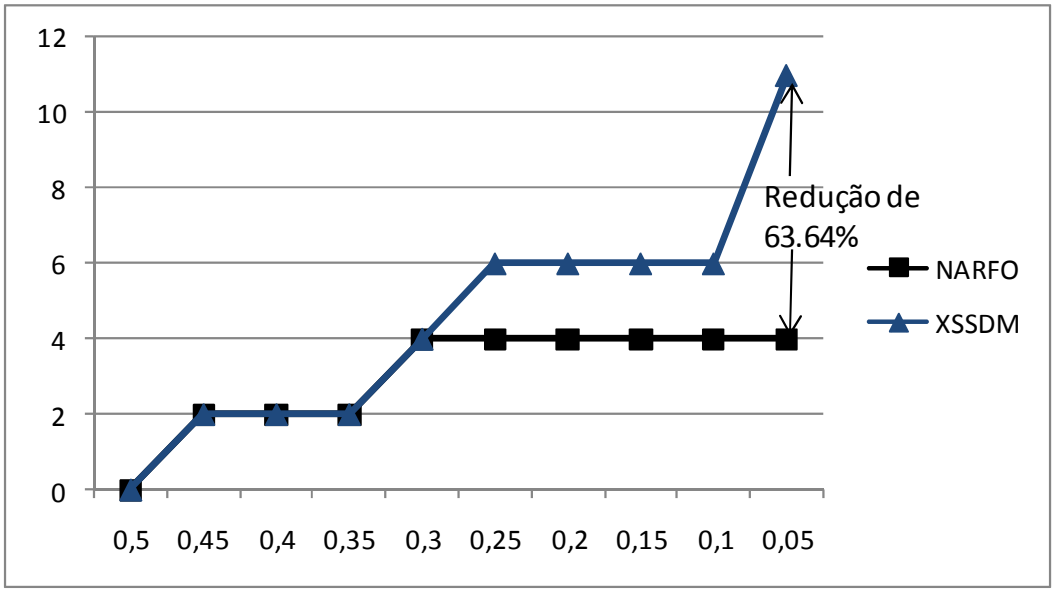


Figura 7.11 - Primeira e segunda série de testes para o conjunto de dados IBGE4.

Tabela 7.7 - Regras geradas pelo algoritmo NARFO com o conjunto de dados IBGE4 e suporte 0,2 com ambas as baterias de testes.

	Regras geradas
1	Branco->Urbano suporte=0.4559, confiança=0.848186
2	Urbano-> Branco suporte=0.4559, confiança=0.56152236
3	Mulato~Negro->Urbano suporte=0.30987, confiança=0.770936, possuindo item 'Mulato' com maior relevância!!!
4	Urbano->Mulato~Negro suporte=0.30987, confiança=0.3816603, possuindo item 'Mulato' com maior relevância!!!

Tabela 7.8 - Regras geradas pelo algoritmo XSSDM com o conjunto de dados IBGE4 e suporte 0,2 com ambas as baterias de testes.

	Regras geradas
1	Mulato->Urbano suporte=0.2952, confiança=0.7677503
2	Urbano->Mulato suporte=0.2952, confiança=0.36359155
3	Branco->Urbano suporte=0.4559, confiança=0.848186
4	Urbano->Branco suporte=0.4559, confiança=0.56152236
5	Mulato~Negro->Urbano suporte=0.30987, confiança=0.770936
6	Urbano-> Mulato~Negro suporte=0.30987, confiança=0.3816603

Pelos experimentos realizados, pode-se concluir que o algoritmo NARFO é eficiente quando o valor do suporte mínimo desejado decresce, resultando em uma quantia menor de regras de associação que são mais compreensíveis ao usuário. Além disso, com a implementação do método de generalização de *itemsets* não frequentes, podemos obter regras relevantes e interessantes e que não eram descobertas pelo XSSDM quando o valor de *minsup* é consideravelmente alto.

Demonstrada a eficiência do algoritmo NARFO através dos experimentos, o capítulo a seguir trás conclusões finais sobre este trabalho de mestrado e relata possíveis trabalhos futuros.

8. CONCLUSÕES

8.1. RESULTADOS OBTIDOS

Com o desenvolvimento e aplicação do algoritmo NARFO, tornou-se possível descobrir regras de associação generalizadas não redundantes que refletem a similaridade semântica entre os dados. Para representar essa similaridade semântica, uma ontologia difusa é criada e utilizada como um conhecimento de apoio ao processo de mineração de regras de associação. Assim, o algoritmo NARFO é capaz de gerar regras de associação generalizadas semanticamente mais ricas sem redundância, evitando a apresentação de informação incorreta ao usuário. Pelo mecanismo de generalização implementado, mesmo se algum descendente não estiver contido na generalização, o usuário não recebe informação equivocada, uma vez que o NARFO mostra o(s) descendente(s) que não fazem parte da regra generalizada.

Os resultados alcançados durante a aplicação dos testes mostram a eficácia do NARFO. Para valores de suporte mínimo relativamente alto, o algoritmo pode produzir regras relevantes e potencialmente úteis que não eram descobertas anteriormente. Isso é resultado da generalização de *itemsets* não frequentes. Além disso, conforme o valor do *minsup* vai decrescendo, NARFO gera uma quantia menor de regras devido ao tratamento de generalização e redundância realizado no pós-processamento, obtendo até 67.05% de redução do número de regras geradas em comparação ao XSSDM.

Pode-se concluir que o algoritmo desenvolvido neste trabalho de mestrado é um algoritmo muito adequado para realizar a mineração de regras de associação generalizadas não redundantes baseada em uma ontologia difusa, obtendo regras semanticamente mais ricas e compreensíveis para o usuário.

8.2. CONTRIBUIÇÕES

Esse trabalho possui as seguintes contribuições:

- A criação de um novo algoritmo (NARFO) para a mineração de regras de associação generalizadas sem redundância baseada em uma ontologia difusa;

- A definição de um novo parâmetro (*mingen*) para indicar a porcentagem mínima que descendentes de um mesmo ancestral devem aparecer em regras idênticas, exceto pelo descendente em questão, para realizar a generalização das regras;
- Método eficiente para tratamento de generalização;
- Tratamento de redundância;
- Implementação da técnica de generalização de *itemsets* não frequentes.

8.3. TRABALHOS FUTUROS

8.3.1. Análise de desempenho

Foram realizados testes comparativos com o algoritmo XSSDM. Porém, comparar seu desempenho com outros algoritmos existentes, como o Apriori ou o outro algoritmo convencional de mineração de regras de associação, também é importante.

8.3.2. Possibilitar a mineração em vários níveis de uma hierarquia

NARFO realiza a mineração de regras de associação com taxonomias de até três níveis. Desenvolver uma estrutura que possibilite realizar o algoritmo NARFO envolvendo hierarquias com mais níveis seria um trabalho interessante e importante a ser realizado.

8.3.3. Realizar mais testes

Os testes realizados comprovaram a eficiência do algoritmo NARFO, tanto em relação ao tratamento de *itemsets* não frequentes quanto em relação ao tratamento de generalização e redundância implementados. No entanto, realizar os testes do capítulo 7 com outro algoritmo e compará-lo com o NARFO, testar o algoritmo NARFO com outras bases e aplicar diferentes tipos de testes comprovariam a importância do algoritmo.

8.3.4. Automatizar as medidas de suporte e confiança

No algoritmo NARFO e na maioria dos algoritmos de mineração de regras de associação, o próprio usuário estabelece os valores de suporte mínimo e confiança mínima. Adicionar ao NARFO uma implementação que estabeleceria valores de *minsup* e *minconf* automaticamente, sem a presença do usuário, torna-se uma questão relevante.

Isso poderia ser feito do seguinte modo: o algoritmo geraria as regras de associação a partir de valores de *minsup* e *minconf* pré-estabelecidos. Caso o usuário considere que as regras geradas são relevantes, o algoritmo termina e têm-se as regras resultantes dessa geração. Caso contrário, o algoritmo modificaria os valores de *minsup* e *minconf* (aumentando-os ou decrescendo-os) até que o usuário do sistema considere que as regras geradas pelo algoritmo sejam relevantes.

REFERÊNCIAS

AGRAWAL, R; SRIKANT, R. Fast Algorithms for Mining Association Rules. In: CONFERENCE ON VERY LARGE DATA BASES (VLDB), n. 20, 1994, Santiago, Chile. **Anais**. San Francisco, United States of America: Morgan Kaufmann Publishers Inc., 1994. p. 487-499.

AGRAWAL, R; IMIELINSKI, T; SWAMI, A. M. Mining Association Rules between Sets of Items in Large Databases. In: ACM SIGMOD ANNUAL CONFERENCE ON MANAGEMENT OF DATA, n. 19, 1993, Washington, United States of America. **Proceedings**. New York, United States of America: ACM, 1993, p. 207-216.

AU, W.-H.; CHAN, K. C. C. FARM: A Data Mining System for Discovering Fuzzy Association Rules. In: IEEE INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS, n. 8, 1999, Seoul, Coréia do Sul. **Proceedings.** Washington, USA: IEEE Computer Society, 1999, p. 1217-1222.

BAYARDO, J. R. J. Efficiently mining long patterns from databases. **ACM SIGMOD RECORD**, New York , United States of América, v. 7, n. 2, p. 85-93, 1998.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. **The Semantic Web**. Scientific American, v.284, n. 5, p. 34-43, 2001.

BEZDEK, J. C. **Pattern recognition with fuzzy objective functions algorithms.** 1 ed. Norwell, United States of America: Kluwer Academic Publishers, 1981, p. 225.

BRISSON, L.; COLLARD, M. An Ontology Driven Data Mining Process. In: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, n. 10, 2008, Barcelona, Espanha. **Proceedings.** Berlin, Alemanha: Springer-Verlag, p.54-61.

BRISSON, L.; COLLARD, M.; PASQUIER, N. Improving the Knowledge Process Discovery Using Ontologies. In: INTERNATIONAL WORKSHOP ON MINING COMPLEX DATA, n. 1, 2005, Houston, United States of America. Washington, USA: IEEE Computer Society.

CALVANESE, D.; DE GIACOMO, G. Data Integration: A Logic-Based Perspective. **AI Magazine**, Melon Park, United States of América, v. 26, n. 1, p. 59-70, 2005.

CARROL, J. J.; DICKINSON, I.; DOLLIN, C.; REYNOLDS, D.; SEABORN, A.; WILKINSON, K. Jena: implementing the semantic web recommendations. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, n. 13, 2004, New York, United States of America. **Proceedings.** New York, United States of America: ACM, 2004, p. 74-83.

CHEN, G.; WEI, Q. Fuzzy Association Rules and the Extended Mining Algorithms. **Information Sciences – Informatics and Computer Science: An International Journal,** Nova York, United States of America, v. 147, n. 1, p. 201-228, 2002.

CHEN, G.; WEI, Q.; KERRE, E. E. Fuzzy Data Mining: Discovery of Fuzzy Generalized Association Rules. In: BORDOGNA, G.; PASI, G. **Recent Issues on Fuzzy Data Mining.** Wurzburg, Alemanha: Physica-Verlag, 2000. p. 45-66.

CHEN, X.; ZHOU, X.; SCHERL, R. B.; GELLER, J. Using an Interest Ontology for Improved Support in Rule Mining. In: INTERNATIONAL CONFERENCE ON DATA WAREHOUSING AND KNOWLEDGE DISCOVERY, n. 5, 2003, Praga, República Checa. **Proceedings.** Berlin, Alemanha: Springer, 2003, p. 320-329.

ELMASRI, R.; NAVATHE, S.B. **Sistemas de Banco de Dados**. 4. Ed. São Paulo: Pearson Addison Wesley, 2005. p. 625-625.

ESCOVAR, E. L. G.; BIAJIZ, M.; VIEIRA, M. T. P. SSDM: A Semantically Similar Data Mining Algorithm. In: BRAZILIAN SYMPOSIUM ON DATABASES, n. 20, 2005, Uberlândia, Brazil. **Anais.** p. 265-279.

ESCOVAR, E. L. G; YAGUINUMA, C. A.; BIAJIZ, M. Using Fuzzy Ontologies to Extend Semantically Similar Data Mining. In: BRAZILIAN SYMPOSIUM ON DATABASES, n. 21, 2006, Florianópolis, Brazil. **Anais.** p 16-30.

FARZANYAR, Z.; KANGAVARI, M.; HASHEMI, S. A New Algorithm for Mining Fuzzy

Association Rules in the Large Databases Based on Ontology. In: INTERNATIONAL CONFERENCE ON DATA MINING – WORKSHOPS, n. 6, 2006, Hong Kong, China. **Proceedings.**

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. The KDD process for extracting useful knowledge from volumes of data. **Communications of the ACM**, New York, v. 39, n. 11, p. 27-34, 1996. Washington, USA: IEEE Computer Society, 2006, p. 65-69.

FIKES, R.; HAYESB, P.; HORROCKS, I. OWL-QL—a language for deductive query answering on the Semantic Web. **Web Semantics: Science, Services and Agents on the World Wide Web**, v. 2, n. 1, p. 19-29, 2004.

GRAU, B. C.; HORROCKS, I.; MOTIK, B.; PARSIA, B.; PATEL-SCHNEIDER, P.; SATTLER, U. OWL 2: The next step for OWL. **Web Semantics: Science, Services and Agents on the World Wide Web,** v. 6, n. 2, p. 309-322, 2008.

GRUBER, T. R. A Translation Approach to Portable Ontology Specifications. **Knowledge Acquisition**, London, United Kingdom, v. 5, n. 2, p. 199-220, 1993.

GUARINO, N. Formal Ontology and Information Systems. In: INTERNATIONAL CONFERENCE ON FORMAL ONTOLOGIES IN INFORMATION SYSTEMS, n. 1, 1998, Trento, Italy. **Proceedings**. Amsterda: IOS Press, 1988. p. 3-15.

HAN, J.; FU, Y. Mining Multiple-Level Association Rules in Large Databases. **IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING**, Piscataway, United States of America, v. 11, n. 5, p.798-805, set 1999.

HAN, J.; KAMBER, M. **Data Mining:** Concepts and Techniques. 2. Ed. Morgan Kaufmann, 2006. 743 p.

HIPP, J.; GUNTZER, U.; NAKHAEIZADEH, G. Algorithms for Association Rule Mining – A General Survey and Comparison. **ACM SIGKDD Explorations Newsletter,** New York, United States of America, v. 2, n. 1, p. 58-64, 2000.

HOU, X.; GU, J.; SHEN, X.; YAN, W. Application of Data Mining in Fault Diagnosis Based on Ontology. In: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY AND APPLICATIONS, n. 3, 2005, Sydney, Australia. **Proceedings.** Washington, USA: IEEE Computer Society, 2005, p. 260-263.

HU, C.; ZHAO, Y. An Ontology-based Framework for Knowledge Service in Digital Library. In: INTERNATIONAL CONFERENCE ON WIRELESS COMMUNICATIONS, NETWORKING AND MOBILE COMPUTING, n. 3, 2007, Shangai, China. **Proceedings.** Washington, USA: IEEE Computer Society, 2007, p. 5345-5348.

KLIR, G. J.; YUAN, B. **Fuzzy Sets and Fuzzy Logic - Theory and Applications**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1995. 592 p.

KNUBLAUCH, H.; FERGERSON, R. W.; NOY, N. F.; MUSEN, M. A. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In: INTERNATIONAL SEMANTIC WEB CONFERENCE, n. 3, 2003, Hiroshima, Japan. **Proceedings.** Berlin, Alemanha: Springer-Verlag, 2004, p. 229-243.

KOTSIFAKOS, E.; MARKETOS, G.; THEODORIDIS, Y. A Framework for Integrating Ontologies and Pattern-bases. In: NIGRO, H. O.; CISARO, S. E. G.; XODO, D. H. **Data Mining with Ontologies: Implementations, Findings, and Frameworks.** Buenos Aires, Argentina: Information Science Reference, 2008. Capítulo 13.

KUNKLE, D.; ZHANG, D.; COOPERMAN, G. Mining Frequent Generalized Itemsets and Generalized Association Rules Without Redundancy. **JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY**, Boston, United States of America, v. 23, n. 1, p. 77-102, jan 2008.

LEE, J.-K.; LEE-KWANG, H. An Extension of Association Rules Using Fuzzy Sets. In: INTERNATIONAL FUZZY SYSTEMS ASSOCIATION WORLD CONGRESS, n. 7, 1997, Praga, República Tcheca. **Proceedings.** Praga, República Tcheca: 1997, p. 399-402.

LEE, K.-M. Mining Generalized Fuzzy Quantitative Association Rules with Fuzzy Generalization Hierarchies. In: NORTH AMERICAN FUZZY INFORMATION

PROCESSING SOCIETY, n. 20, 2001, Vancouver, Canadá. **Proceedings.** Washington, USA: IEEE Computer Society, 2001, p. 2977-2982.

LEITE, M. A. A.; RICARTE, I. L. M. Document Retrieval Using Fuzzy Related Geographic Ontologies. In: INTERNATIONAL WORKSHOP ON GEOGRAPHIC INFORMATION RETRIEVAL, n. 2, 2008, Napa Valley, United States of America. **Proceedings.** New York, United States of America: ACM, 2008, p. 47-54.

NARDI, D.; BRACHMAN, R. J. An Introduction to Description Logics. In: Baader, F., *et al.* **The Description Logic Handbook.** New York, USA: Cambridge University Press, 2003, p. 5-44.

ORLANDO, S.; PALMERINI, P.; PEREGO, F. Enhancing the Apriori algorithm for Frequent Set Counting. **Lecture Notes in Computer Science**, London, UK, v. 2114, p. 71-82, 2001.

PARK, J. S.; CHEN, M-S.; YU, P. S. An effective hash-based algorithm for mining association rules. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, n. 21, 1995, San Jose, United States of America. **Proceedings.** New York, United States of America: ACM, 1995, p. 175-186.

PASQUIER, N.; BASTIDE, Y.; TAOUIL, R.; LAKHAL, L. Discovering frequent closed itemsets for association rules. In: INTERNATIONAL CONFERENCE ON DATABASE THEORY, n. 7, 1999, Jerusalém, Israel. **Proceedings**. Aubière Cedex, France: Springer Berlin, 398-416.

SMITH, M. K.; WELTY, C.; MCGUINESS, D. L. **W3C Proposed Recomendation: OWL Web Ontology Language Guide**. Disponível em: <http://www.w3.org/TR/2004/REC-owl-guide-20040210>. Acesso em: 17 fev. 2009.

SRIKANT, R; AGRAWAL, R. Mining Generalized Association Rules. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, n. 21, 1995, Zurich, Switzerland. **Proceedings**. Morgan Kaufmann, 1995, 407-419.

SRIKANT, R.; AGRAWAL, R. Mining Quantitative Association Rules in large Relational

Tables. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, n. 22, 1996, Montreal, Canadá. **Proceedings.** New York, United States of America: ACM, 1996, p.1-12.

SRIPHAEW, K.; THEERAMUNKONG, T. Fast Algorithms for Mining Generalized Frequent Patterns of Generalized Association Rules. **IEICE TRANSACTIONS on Information and Systems**, Tokyo, Japan, v. E87-D, n. 3, p. 761-770, mar 2004.

STRACCIA, U. A Fuzzy Logic Description Logic for the Semantic web. In: SANCHEZ, E. **Fuzzy Logic and the Semantic Web.** Pisa, Itália: Elsevier, 2006, p. 73-90.

USCHOLD, M.; GRUNINGER, M. Ontologies and Semantics for Seamless Connectivity. **ACM SIGMOD Record**, New York, United States of America, v. 33, n. 4, p. 58-64, 2004.

WACHE, H.; VÖGELE, T.; VISSER, U.; STUCKENSCMIDT, G.; SCHUSTER, G.; NEUMANN, H.; HUBNER, S. Ontology-based integration of information - a survey of existing approaches. In: IJCAI-01 WORKSHOP: ONTOLOGIES AND INFORMATION SHARING, n. 17, 2001, Seattle, USA. **Proceedings.** Seattle, USA: 2001. p. 108-117.

WOJCIECHOWSKI, M.; ZAKRZEWICZ, M. On Efficiency of Dataset Filtering Implementations in Constraint-Based Discovery of Frequent Itemsets. In: JOINT CONFERENCE ON KNOWLEDGE-BASED SOFTWARE ENGINEERING, n. 5, 2002, Maribor, Slovenia. **Anais**. Maribor, Slovenia, 2002.

YEN, S-H.; CHEN, A. L. P. An Efficient Approach to Discovering Knowledge from Large Databases. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED INFORMATION SYSTEMS, n. 4, 1996, Miami Beach, United States of America. **Proceedings.** Washington, USA: IEEE Computer Society, 1996, p. 8-18.

ZADEH, L. A. **Fuzzy Sets**. In: Yager, R. R., *et al.* Fuzzy sets and applications: Selected Papers by L.A. Zadeh. New York, USA:Wiley-Interscience, 1987a, p. 29-44.

ZADEH, L. A. **Similarity Relations and Fuzzy Orderings**. In: Yager, R. R., *et al.* Fuzzy Sets and Applications: Select Papers by L. A. Zadeh. New York, USA:Wiley-Interscience,

1987b, p. 81-104.

ZHAI, J.; SHEN, L.; LIANG, Y.; JIANG, J. Application of Fuzzy Ontology to Information Retrieval for Electronic Commerce. In: INTERNATIONAL SYMPOSIUM ON ELECTRONIC COMMERCE AND SECURITY, n. 1, 2008, Guangzhou, China. **Proceedings.** Washington, USA: IEEE Computer Society, 2008, p. 221-225.

ZHANG, F.; MA, Z. M.; LV, Y.; WANG, X. Formal Semantics-Preserving Translation from Fuzzy ER Model to Fuzzy OWL DL Ontology. In: INTERNATIONAL CONFERENCE ON WEB INTELLIGENCE AND INTELLIGENT AGENT TECHNOLOGY, n. 8, 2008, Sydney, Austrália. **Proceedings.** Washington, USA: IEEE Computer Society, 2008, p. 503-509.

ZHANG, S.; SUN, J.; WU, P. Research on the Fuzzy Quantitative Association Rules Mining Algorithm and Its Simulation. In: INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS AND KNOWLEDGE DISCOVERY, n. 4, 2007, Haikou, China. **Proceedings.** Washington, USA: IEEE Computer Society, 2007, p. 401-405.

APÊNDICE

Este apêndice contém o código fonte de todas as classes utilizadas para o desenvolvimento do algoritmo NARFO.

Classe Apriori

```
package aprioriminer;

import java.util.*;
import java.io.IOException;

import ontologyReader.*;

/**

   Apriori.java<P>

   This class implements the Apriori algorithm
   for finding large itemsets.

   (see "Fast Algorithms for Mining Association Rules"
   by Rakesh Agrawal and Ramakrishnan Srikant
   from IBM Almaden Research Center 1994)

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

//public class Apriori implements LargeItemsetsFinder {
public class Apriori implements FuzzyItemsetsFinder {

      private static final int INITIAL_CAPACITY = 10000;

      // our collections of itemsets
      private Vector candidates;
      private Vector k_frequent;
      private Vector large;
      private Vector nonLarge;
      private Vector generilizedDomain;

      // the hashtrees associated with candidates and k_frequent
      private HashTree ht_candidates;
      private HashTree ht_k_frequent;
      private HashTree ht_non_frequent;

      // the domains associated with items
```

```
private Domain[] D;
private Hashtable domains;
private String[] DN; //Domain names
private String[] generalized;
// the fuzzy candidates created and their factors
private Vector fuzzy_candidates;
private Vector factors;

// this remembers the number of passes and also indicates the
// current cardinality of the candidates
private int pass_num;

// our interface to the outside world
private DBReader db_reader, sims;
private DBCacheWriter cache_writer;

// useful information
private long num_rows;
private long min_weight;
private float minsim;

// Strings to store ontology identifier and path to the owl file
private String ontURL;
private String ontPath;

/**
 * Find the frequent itemsets in a database
 *
 * @param dbReader   the object used to read from the database
 * @param cacheWriter   the object used to write to the cache
 * if this is null, then nothing will be saved, this is useful
 * for benchmarking
 * @param minSupport   the minimum support
 * @return   the number of passes executed over the database
 */
public int findLargeItemsets(
      DBReader dbReader,
      DBCacheWriter cacheWriter,
      float minSupport,
      float minSim,
      String ontologyURL,
      String ontologyPath) {
      // save the following into member fields
      db_reader = dbReader;
      cache_writer = cacheWriter;
      num_rows = dbReader.getNumRows();
      min_weight = (long) (num_rows * minSupport);
      minsim = minSim;
      //sims = similarities;
      ontURL = ontologyURL;
      ontPath = ontologyPath;

      // initialize the collections
      candidates = new Vector(INITIAL_CAPACITY);
      k_frequent = new Vector(INITIAL_CAPACITY);
      large = new Vector(INITIAL_CAPACITY);
      nonLarge = new Vector(INITIAL_CAPACITY);
      generilizedDomain = new Vector(INITIAL_CAPACITY);

      // initialize the hash trees
      ht_k_frequent = new HashTree(k_frequent);
```

```
            ht_candidates = new HashTree(candidates);
            ht_non_frequent = new HashTree(nonLarge);
            Hashtable ht_cand2 = new Hashtable();
//           Get domain names
            try{
            DN = db_reader.getDomainNames();
            }catch(Exception e){
                  e.printStackTrace();
            }
            generalized = new String[DN.length];
            for (int i = 0; i < generalized.length; i++) {
                  generalized[i] = "";
            }
            // The initial candidates are all 1-itemsets
            Itemset is, ist;
            try {
                  ist = db_reader.getFirstRow();
                  for (int i = 0; i < ist.size(); i++) {
                        try {
                              if (ht_cand2.get(ist.getItem(i)) == null) {
                                    is = new Itemset(1);
                                    is.addItem(ist.getItem(i));

                                    candidates.add(is);
                                    ht_candidates.add(candidates.size() -
1);
                                    ht_cand2.put(ist.getItem(i),
ist.getItem(i));
                              }
                        } catch (NullPointerException e) {
                        }
                  }

                  for (int i = 1; i < db_reader.getNumRows()-1; i++) {
                        ist = db_reader.getNextRow();
                        for (int j = 0; j < ist.size(); j++) {
                              try {
                                    if (ht_cand2.get(ist.getItem(j)) ==
null) {
                                          is = new Itemset(1);
                                          is.addItem(ist.getItem(j));

                                          candidates.add(is);

      ht_candidates.add(candidates.size() - 1);
                                          ht_cand2.put(ist.getItem(j),
ist.getItem(j));
                                    }
                              } catch (NullPointerException e) {

                              }
                        }
                  }
            } catch (Exception e) {
                  e.printStackTrace();
            }

            // set the items' domains
            //(this task now is done within OntologyAnalysis)
            // setDomains();
```

```
            // set similarities between items
            //(this task now is done within OntologyAnalysis)
            // setSimilarities();

            // Fuzzy Analysis
            //(this task now is done within OntologyAnalysis)
            // fuzzyAnalysis();

            // Ontology Analysis
            OntologyAnalysis();

            // we start with first pass
            for (pass_num = 1;; pass_num++) {

                  // compute the weight of each candidate
                  weighCandidates();

                  // Adjust weights of fuzzy sets by applying factors
                  adjustWeights();

                  // Output of candidates, step by step (uh baby)
                  // to show domains, the parameter must be set true
                  outputCandidates(true);

                  // verify which candidates are frequent (have weight
                  // greater than or equal to min_weight)
                  evaluateCandidates();

                  //Itemset.pruneNonMaximal(large);
                  // compute maximum cardinality of large itemsets found so
far
                  int card, maxcard = 0;
                  for (int i = 0; i < large.size(); i++)
                        if ((card = ((Itemset) large.get(i)).size()) >
maxcard)
                              maxcard = card;

                  // if last pass didn't produce any large itemsets
                  // then we can stop the algorithm
                  if (pass_num > maxcard)
                        break;

                  // if we just examined the top itemset (the one
containing
                  // all items) then we're done, nothing more to do.
                  if (pass_num >= db_reader.getNumColumns())
                        break;

                  // generate new candidates from frequent itemsets
                  generateCandidates();

                  // exit if no more candidates
                  if (candidates.size() == 0)
                        break;
            }

            return pass_num;
      }

      // this procedure scans the database and computes the weight of each
      // candidate
```

```
	private void weighCandidates() {
		ht_candidates.prepareForDescent();
		try {
			Itemset row = db_reader.getFirstRow();
			ht_candidates.update(row);

			while (db_reader.hasMoreRows()) {
				row = db_reader.getNextRow();
				ht_candidates.update(row);
			}
		} catch (Exception e) {
			System.err.println("Error scanning database!!!\n" + e);
		}
	}

	// this procedure checks to see which itemsets are frequent
	private void evaluateCandidates() {
		Itemset is;

		for (int i = 0; i < candidates.size(); i++){
			is = (Itemset) candidates.get(i);
			// if this is a frequent itemset
			if ((is).getWeight() >= min_weight) {

				// compute support of itemset
				is.setSupport((float) is.getWeight() / (float)
(num_rows-1));

				// write itemset to the cache
				try {
					if (cache_writer != null)
						cache_writer.writeItemset(is);
				} catch (IOException e) {
					System.err.println("Fishy error!!!\n" + e);
				}

				// then add it to the large and k_frequent
collections
				large.add(is);
				k_frequent.add(is);
				ht_k_frequent.add(k_frequent.size() - 1);
			}else{
				is.setSupport((float) is.getWeight() / (float)
(num_rows-1));
				nonLarge.add(is);
			}
		}
		// reinitialize candidates for next step
		domains = db_reader.getDomains();


		OntReader ontReader = new OntReader(ontURL, ontPath);

		//tenta verificar se todos os items de um dominio sao nao
frequentes e os adiciona se tds foram infrequentes
		/*
		for (int i = 0; i <DN.length; i++) {
			int p = ontReader.getClassSize(DN[i]);
			boolean dn = true;
			//Itemset []domainsItems = new Itemset[p];
```

```
                  for (int j = 0; j < nonLarge.size(); j++) {
                        Itemset itemset  = (Itemset) nonLarge.get(j);
                        if (itemset.size() == pass_num ){
                              boolean bool = false;
                              for (int t = 0; t < itemset.size() && !bool;
t++) {
                                    String s =
DN[((Integer)(domains.get(itemset.getItem(t)))).intValue()];
                                    if (s.compareTo(DN[i]) == 0)
                                          bool = true;
                              }
                              if (bool){
                                    float suport = itemset.getSupport();

                                    float weight = itemset.getWeight();
                                    int count = 1;
                                    for (int index = j+1; index <
nonLarge.size() && count!= p; index++) {
                                          Itemset itemsetAux = (Itemset)
nonLarge.get(index);
                                          int countItemset = 0;
                                          if ((itemset.size() ==
itemsetAux.size()) && (itemset.size() == pass_num) &&
(!itemsetAux.itemsSameDomain(itemsetAux,DN,domains))){
                                                for (int k = 0; k <
itemset.size(); k++) {
                                                      String classname =
DN[((Integer)(domains.get(itemset.getItem(k)))).intValue()];
                                                      for (int aux = 0; aux
< itemsetAux.size(); aux++) {
                                                            String
classnameAux =
DN[((Integer)(domains.get(itemsetAux.getItem(aux)))).intValue()];
                                                            if
((itemset.getItem(k).compareTo(itemsetAux.getItem(aux)) == 0 &&
classname.compareTo(DN[i]) != 0) |||((DN[i].compareTo(classnameAux) == 0) &&
(classname.compareTo(classnameAux) == 0) &&
(itemset.getItem(k).compareTo(itemsetAux.getItem(aux)) != 0)))

      countItemset++;
                                                      }
                                                }
                                          }
                                          if(countItemset ==
itemset.size()){
                                                count++;
                                                suport = suport +
itemsetAux.getSupport();
                                                weight = weight +
itemsetAux.getWeight();
                                          }
                                          if (count == p && weight >=
min_weight){
                                                Itemset isAux = new
Itemset();
                                                isAux.setWeight(weight);
                                                isAux.setSupport(suport);
                                                isAux.addItem(DN[i]);
                                                boolean verify = true;
                                                for (int k = 0; k <
large.size() && verify; k++) {
```

```
                                                      Itemset item =
(Itemset) large.get(k);
                                                      if
(isAux.isEqualTo(item))
                                                            verify = false;
                                                }
                                                if (verify){
                                                      for (int k = 0; k <
itemset.size(); k++) {
                                                            String s =
DN[((Integer)(domains.get(itemset.getItem(k)))).intValue()];
      if(s.compareTo(DN[i])!= 0)
      isAux.addItem(itemset.getItem(k));
                                                      }
                                                      //boolean boolAux =
false;
                                                      for (int k = 0; k <
large.size() ; k++) {
                                                            Itemset
generalizeItemset = new Itemset();
                                                            Itemset aux =
(Itemset) large.get(k);
                                                            float sup =
aux.getSupport();
                                                            float wei =
aux.getWeight();
                                                            int itemsDomain
= 1;
                                                            if
(aux.isSubItemset(aux,isAux,DN[i],domains,DN) && aux.size() == (pass_num -
1)){
                                                                  for (int x
= k +1; x < large.size() && itemsDomain < p; x++) {
      Itemset aux2 = (Itemset) large.get(x);
                                                                        int
itemsetSize = 0;//count if an itemset is equal
                                                                        if
(aux.size() == (itemset.size()-1) && (aux2.size() == aux.size())){
      for (int index2 = 0; index2 < aux.size() && itemsDomain < p;
index2++) {
            String getAuxDomain = "";
            boolean boolY = true;
            for (int y = 0; y < DN.length && boolY; y++) {
                  if (aux.getItem(index2).compareTo(DN[y]) == 0){
                        boolY = false;
                        getAuxDomain = DN[y];
                  }
            }
```

```
            if (boolY)
                  getAuxDomain =
DN[((Integer)(domains.get(aux.getItem(index2)))).intValue()];
            for (int index3 = 0; index3 < aux2.size(); index3++) {
                  String getAux2Domain = "";
                  boolY = true;
                  for (int y = 0; y < DN.length && boolY; y++) {
                        if (aux2.getItem(index3).compareTo(DN[y]) == 0){
                              boolY = false;
                              getAuxDomain = DN[y];
                        }
                  }
                  if (boolY)
                        getAux2Domain =
DN[((Integer)(domains.get(aux2.getItem(index3)))).intValue()];
                  if ((getAuxDomain.compareTo(DN[i]) == 0 &&
getAuxDomain.compareTo(getAux2Domain) == 0)
                              || (getAuxDomain.compareTo(DN[i]) != 0 &&
aux.getItem(index2).compareTo(aux2.getItem(index3)) == 0)){
                        itemsetSize++;
                  }
            }//verificar suportes
      }
                                                                  }
                                                                  if
(itemsetSize == aux.size()){
      itemsDomain++;
      sup = sup + aux2.getSupport();
      wei = wei + aux2.getWeight();
                                                                  }
                                                            }
                                                      }
                                                      if (itemsDomain
== p && wei >= min_weight){
      generalizeItemset.addItem(DN[i]);
                                                            float
support = 0;
```

```
                                                                  boolean b
= true;
                                                                  if
(generalizeItemset.getSupport() == 0){
                                                                        for
(int z = 0; z < large.size() && b; z++) {
      Itemset itemSet = (Itemset) large.get(z);
      if (itemSet.size() < 2){
            String help =
DN[((Integer)(domains.get(itemSet.getFirstItem()))).intValue()];
            if (help.compareTo(DN[i]) == 0)
                  support = support + itemSet.getSupport();
      }else
            b = false;
                                                                        }
      generalizeItemset.setSupport(support);
                                                                        try
{
      if (cache_writer != null){
            //put on cache the generilex item of size one
            cache_writer.writeItemset(generalizeItemset);


      }
                                                                        }
catch (IOException e) {
      System.err.println("Fishy error!!!\n" + e);
                                                                        }
                                                                  }
                                                                  Itemset
gen = new Itemset();
      gen.setSupport(sup);
      gen.setWeight(wei);
      gen.addItem(DN[i]);


                                                                  for (int x
= 0; x < aux.size(); x++) {
      String s = DN[((Integer)(domains.get(aux.getItem(x)))).intValue()];
                                                                        if
(!(DN[i].compareTo(s) == 0))
      gen.addItem(aux.getItem(x));
                                                                  }
```

```
                                                                  try {
                                                                        if
(cache_writer != null){


      cache_writer.writeItemset(gen);

                                                                        }
                                                                  } catch
(IOException e) {
      System.err.println("Fishy error!!!\n" + e);
                                                                  }
                                                            }


                                                      }
                                                      try {
                                                            if (cache_writer
!= null)
      cache_writer.writeItemset(isAux);
                                                      } catch (IOException
e) {
                                                            // TODO Auto-
generated catch block
      System.err.println("Fishy error!!!\n" + e);
                                                      }
                                                      large.add(isAux);
                                                      k_frequent.add(isAux);

      ht_k_frequent.add(k_frequent.size() - 1);
                                                }
                                          }

                                    }
                              }
                        }
                  }
            }
            */
            candidates.clear();
            ht_candidates = new HashTree(candidates);

            // sends to HashTree the fuzzy_candidates created
            ht_candidates.getFuzzyCandidates(fuzzy_candidates);

      }

      // this procedure generates new candidates out of itemsets
      // that are frequent according to the procedure described
      // by Agrawal a.o.
      private void generateCandidates() {
            ht_k_frequent.prepareForDescent();
            if (k_frequent.size() == 0)
                  return;

            for (int i = 0; i < k_frequent.size() - 1; i++)
                  for (int j = i + 1; j < k_frequent.size(); j++)
```

```
                        if (!getCandidate(i, j))
                              break;

                // reinitialize k_frequent for next step
                k_frequent.clear();
                ht_k_frequent = new HashTree(k_frequent);
        }

        // this procedure tries to combine itemsets i and j and returns
        // true if succesful, false if it can't combine them
        private boolean getCandidate(int i, int j) {
                Itemset is_i = (Itemset) k_frequent.get(i);
                Itemset is_j = (Itemset) k_frequent.get(j);

                // if we cannot combine element i with j then we shouldn't
                // waste time for bigger j's. This is because we keep the
                // collections ordered, an important detail in this
implementation
                if (!is_i.canCombineWith(is_j))
                        return false;
                else {
                        Itemset is = is_i.combineWith(is_j);

                        // a real k-frequent itemset has k (k-1)-frequent subsets
                        if (ht_k_frequent.countSubsets(is) == is.size()) {
                                candidates.add(is);
                                ht_candidates.add(candidates.size() - 1);
                        }

                        return true;
                }
        }

        private void setDomains(){

                // Gets domains from DBReader
                //(domains obtained at DBReader.getNextRow())
                domains = db_reader.getDomains();

                // Get domain names
                try{
                DN = db_reader.getDomainNames();
                }catch(Exception e){e.printStackTrace();}

                // Number of domains is equal to the number of columns
                Long number_of_domains = new Long(db_reader.getNumColumns());

                D = new Domain[number_of_domains.intValue()];

                // Counters to count the items in each domain
                int[] counter = new int[number_of_domains.intValue()];
                // Initialization of the counters
                for(int q = 0; q < number_of_domains.intValue(); q++)
                        counter[q] = 0;

                // Counts how many items each domain has
                for (Enumeration e = domains.elements(); e.hasMoreElements() ;)
{
                        counter[Integer.parseInt(e.nextElement().toString())]++;

                }
```

```
            // Creates the domains, which size was obtained by the counters
            for(int l = 0; l < number_of_domains.intValue(); l++)
                  D[l] = new Domain(counter[l]);

            // Add item to correspondent domain
            Enumeration k = domains.keys();
            Enumeration e = domains.elements();
            for (; e.hasMoreElements() ;) {

      D[Integer.parseInt(e.nextElement().toString())].addItem(k.nextElement
().toString());
            }
      }

//    Put similarities among items in each Domain
      private void setSimilarities(){

            String s[] = new String[4];
            int domain;
            String item1, item2;
            float simv;

            try {

                  // each row contains {domain, item1, item2,
similarity_value}
                  s = sims.getFirstSim();

                  // in the itemset, itens are ordered lexicographically
                  domain = Integer.parseInt(s[0]);
                  D[domain].setUsed(); // inform that domain is used
                  item1 = s[1];
                  item2 = s[2];
                  simv = Float.parseFloat(s[3]);

                  // If similarity value does not exist yet
                  if(D[domain].getSimilarity(item1,item2) == 0)
                        D[domain].addSimilarity(item1,item2,simv);

                  for (int i = 1; i < sims.getNumRows(); i++) {
                        s = sims.getNextSim();

                        domain = Integer.parseInt(s[0]);
                        D[domain].setUsed(); // inform that domain is used
                        item1 = s[1];
                        item2 = s[2];
                        simv = Float.parseFloat(s[3]);

                        // If similarity value does not exist yet
                        if(D[domain].getSimilarity(item1,item2) == 0)

                              D[domain].addSimilarity(item1,item2,simv);
                  }
                  } catch (Exception e) { e.printStackTrace(); }
      }

      // output of candidates, showing domains if parameter is true
      private void outputCandidates(boolean domains){

            System.out.println("\nPasso #" + pass_num + ": Candidatos");
```

```
            Itemset I;
            for(int z = 0; z < candidates.size(); z++){
                  I = (Itemset) candidates.get(z);
                  System.out.println(I);

                  if(domains){
                        for(int a = 0; a < I.size(); a++){
                              boolean bool = true;
                              for (int i = 0; i < DN.length && bool; i++) {
                                    if (I.getItem(a).compareTo(DN[i]) == 0)
                                          bool = false;
                              }
                              if (bool)
                                    System.out.println(I.getItem(a) + "<" +
DN[((Integer)(this.domains.get(I.getItem(a)))).intValue()] + ">");
                              else
                                    System.out.println(I.getItem(a) + "<" +
I.getItem(a) + ">");
                        }
                  }
            }
      }

       // Checks every domain's looking for values equal or greater than
minsim
       // if so, items are combined into fuzzy items
       private void fuzzyAnalysis(){

            // Number of domains is equal to the number of columns
            Long number_of_domains = new Long(db_reader.getNumColumns());

            // where fuzzy candidates will be stored
            fuzzy_candidates = new Vector();

            // where fuzzy associations will be stored
            Vector fuzzyAssoc = new Vector();

            // their factors
            factors = new Vector();
            float factor = 0;

            // index to keep the last 2-association position
            int A2 = 0, max_domain_size = 0;

            // Hashtable is used to avoid insertion repetition
            Hashtable ht_fis = new Hashtable();

            // Fuzzy itemsets will be created from these
            Itemset is, fis;

            // for each domain
            for(int d = 0; d < number_of_domains.intValue(); d++){

                  // Trick to get the maximum domain size
                  if(max_domain_size < D[d].size()) max_domain_size =
D[d].size();

                  if(D[d].isUsed()){
                        // tests similarities among domain's items
                        for(int i = 0; i < D[d].size(); i++){
                              for(int j = i + 1; j < D[d].size(); j++){
```

```
      if(D[d].getSimilarity(D[d].items[i],D[d].items[j]) >= minsim){
                                          // items are similar enough, so we
store them in a itemset
                                          fis = new Itemset(1);
                                          fis.addItem(D[d].items[i]);
                                          fis.addItem(D[d].items[j]); //
items are added in lexicographic order


                                          try{

                                                // checks if the fuzzified
form of itemset is not on ht_fis, before put it
                                                if
(ht_fis.get(fis.fuzzify()) == null) {


                                                // factor computation
                                                factor =
(D[d].getSimilarity(D[d].items[i],D[d].items[j])+1)/2;

                                                fuzzyAssoc.add(fis);// add
the fuzzy itemset to fuzzyAssoc
                                                A2++; // increment the last
2-association position


      domains.put(fis.fuzzify(),new Integer(d));
                                                is = new Itemset(1);
                                                is.addItem(fis.fuzzify());

                                                fuzzy_candidates.add(is);
                                                factors.add(new
Float(factor)); //add itemset's factor in the same index
                                                ht_fis.put(fis.fuzzify(),
fis.fuzzify());
                                                }
                                          }catch(NullPointerException e){}

                                    }
                               }
                          } // end fuzzy associations between 2 items
                     } // end if
           } // end domains loop


           // Identifying Similarity Cycles (fuzzy associations between 3
items or more)

           Itemset a, b;
           float cfactor = 0; // for cycles' factor computing
           float fu = 0, fv = 0, ft = 0;
           int d = 0; // for domain
           int Ai = 0, Af = A2; // indexes for associations positions
                                     // Ai = begin, Af = end

           // For each pass of the Similarity Cycles algorithm
           for(int k = 3; k <= max_domain_size; k++){
```

```
                  // compare associations 2 by 2
                  for(int u = Ai; u < Af; u++)
                        for(int v = u + 1; v < Af; v++){
                              is = (Itemset) fuzzyAssoc.elementAt(u);
                              fis = (Itemset) fuzzyAssoc.elementAt(v);

                              // a = is - intersection(is,fis)
                              a = is.subtract(fis);
                              // b = fis - intersection(is,fis)
                              b = fis.subtract(is);

                              // if the non-intersection size of both is 1
(suffix)
                              if(a.size() == 1 && b.size() == 1){

                                    // join non-intersections

                                    a = a.add(b);

                                    // if 2-associations contains 'a'
(suffixes union), then cycle exists
                                    for(int t = 0; t < A2; t++)


      if(((Itemset)fuzzyAssoc.get(t)).isEqualTo(a)){

                                                      // join itemsets
                                                      is = is.add(fis);


                                                      // store domain, factor and
fuzzy candidate
                                                      try{

                                                            // checks if the
fuzzified form of itemset is not on ht_fis, before put it
                                                            if
(ht_fis.get(is.fuzzify()) == null) {

                                                            // add resultant
itemset to fuzzyAssoc
                                                            fuzzyAssoc.add(is);


                                                            // calculate cycle
factor
                                                            fu = ((Float)
factors.elementAt(u)).floatValue();
                                                            fv = ((Float)
factors.elementAt(v)).floatValue();
                                                            ft = ((Float)
factors.elementAt(t)).floatValue();

                                                            // cfactor is the
minimum value among fu, fv and ft
                                                            if(fu > fv) cfactor =
fv;
                                                            else cfactor = fu;

                                                            if(cfactor > ft)
cfactor = ft;
```

```
                                                      // get new itemset's
domain (the same as the first item's domain)
                                                      d =
((Integer)domains.get(is.getItem(0))).intValue();



      domains.put(is.fuzzify(),new Integer(d));
                                                      fis = new Itemset(1);

      fis.addItem(is.fuzzify());


      fuzzy_candidates.add(fis);
                                                      factors.add(new
Float(cfactor)); //add itemset's factor in the same index

      ht_fis.put(is.fuzzify(), is.fuzzify());
                                                      }
                                                 }catch(NullPointerException
e){}
                                           }
                              }
                         }
                         // Update Ai and Af values
                         Ai = Af;
                         Af = fuzzyAssoc.size();

            }

            for(int u = 0; u < fuzzy_candidates.size(); u++){
                  try{
                        candidates.add(fuzzy_candidates.get(u));
                        ht_candidates.add(candidates.size() - 1);
                  }catch(NullPointerException e){}

            }

            // sends to HashTree the fuzzy_candidates
            ht_candidates.getFuzzyCandidates(fuzzy_candidates);
      }

      // checks if each candidate contains fuzzy items.
      // If does, compute the factor and adjust weights.
      private void adjustWeights(){
            Itemset is, fis;
            float factor = 0;

            for (int i = 0; i < candidates.size(); i++)
                  for(int k = 0; k < fuzzy_candidates.size(); k++){

                        // If candidate has at least one fuzzy item
                        if((is = (Itemset) candidates.get(i)).isFuzzy())

                              // if weight is nonzero (we don't need to
adjust zero weights)
                              if(is.getWeight() != 0)
                                    // if fuzzy_item is included in the
itemset
```

```
                                    if((fis = (Itemset)
fuzzy_candidates.get(k)).isIncludedIn(is)){
                                          factor = ((Float)
factors.elementAt(k)).floatValue();
                                          is.setWeight(is.getWeight() *
factor);
                                    }
                  }
      }

       // This method uses OntReader to get pair of items which similarity
       // is greater or equal to minsim, combining them into fuzzy
itemsets.
       private void OntologyAnalysis(){

            // Gets domains from DBReader
            //(domains obtained at DBReader.getNextRow())
            domains = db_reader.getDomains();

            // Get domain names
            try{
                  DN = db_reader.getDomainNames();
            }catch(Exception e){e.printStackTrace();}

            // Number of domains is equal to the number of columns
            Long number_of_domains = new Long(db_reader.getNumColumns());

            // where fuzzy candidates will be stored
            fuzzy_candidates = new Vector();

            // where fuzzy associations will be stored
            Vector fuzzyAssoc = new Vector();

            // where fuzzy associations from the domain k will be stored
            Vector []Ak = new Vector[number_of_domains.intValue()];

            for(int c = 0; c < number_of_domains.intValue(); c++){
                  Ak[c] = new Vector();
            }

            // their factors
            factors = new Vector();
            float factor = 0;

            // Hashtable is used to avoid insertion repetition
            Hashtable ht_fis = new Hashtable();

            // Fuzzy itemsets will be created from these
            Itemset is, fis;

            // Open ontology reader
            OntReader ontReader = new OntReader(ontURL, ontPath);

            // Vector to get results from ontology
            Vector result;

            // for each domain
            for(int d = 0; d < number_of_domains.intValue(); d++){

                  // Get from ontology similar itens (with similarity >=
minsim)
```

```
                  // getSimilarity(domain name, minsim)
                  result = ontReader.getSimilarity(DN[d], minsim);

                  if (result != null) { // If result is null, there is no
similarity >= minsim in this domain
                        Enumeration e = result.elements();
                        while (e.hasMoreElements()) {
                              Vector similarity = (Vector) e.nextElement();

                              // items are similar enough, so we store them
in a itemset
                              fis = new Itemset(1);
                              fis.addItem(similarity.get(0).toString());
                              fis.addItem(similarity.get(1).toString()); //
items are added in lexicographic order

                              try{
                                    // checks if the fuzzified form of
itemset is not on ht_fis, before put it
                                    if (ht_fis.get(fis.fuzzify()) == null) {


                                    // factor computation
                                    factor =
(((Float)similarity.get(2)).floatValue()+1)/2;

                                    Ak[d].add(fis);// add the fuzzy itemset
to the domain Ak

                                    domains.put(fis.fuzzify(),new
Integer(d));
                                    is = new Itemset(1);
                                    is.addItem(fis.fuzzify());

                                    fuzzy_candidates.add(is);
                                    factors.add(new Float(factor)); //add
itemset's factor in the same index
                                    ht_fis.put(fis.fuzzify(),
fis.fuzzify());
                                    }
                              }catch(NullPointerException exc){}
                        } // end while
                  } // end if

             } //end for (domains loop)

            // Identifying Similarity Cycles (fuzzy associations between 3
items or more)

            Itemset a, b;
            float cfactor = 0; // for cycles' factor computing
            float fu = 0, fv = 0, ft = 0;

            // for each domain
            for(int d = 0; d < number_of_domains.intValue(); d++){

                  // For each pass of the Similarity Cycles algorithm
                  for(int k = 3; k <= ontReader.getClassSize(DN[d]); k++){

                        // compare associations 2 by 2
                        for(int u = 0; u < Ak[d].size(); u++)
```

```
                              for(int v = u + 1; v < Ak[d].size(); v++){
                                    is = (Itemset) Ak[d].elementAt(u);
                                    fis = (Itemset) Ak[d].elementAt(v);

                                    // a = is - intersection(is,fis)
                                    a = is.subtract(fis);
                                    // b = fis - intersection(is,fis)
                                    b = fis.subtract(is);

                                    // if the non-intersection size of both
is 1 (suffix)
                                    if(a.size() == 1 && b.size() == 1){

                                          // join non-intersections

                                          a = a.add(b);

                                          // if 2-associations contains 'a'
(suffixes union), then cycle exists
                                          for(int t = 0; t < Ak[d].size();
t++)

      if(((Itemset)Ak[d].get(t)).isEqualTo(a)){

                                                      // join itemsets
                                                      is = is.add(fis);


                                                      // store domain,
factor and fuzzy candidate
                                                      try{

                                                            // checks if the
fuzzified form of itemset is not on ht_fis, before put it
                                                            if
(ht_fis.get(is.fuzzify()) == null) {

                                                            // add resultant
itemset to Ak[d]
                                                            Ak[d].add(is);


                                                            // calculate
cycle factor
                                                            fu = ((Float)
factors.elementAt(u)).floatValue();
                                                            fv = ((Float)
factors.elementAt(v)).floatValue();
                                                            ft = ((Float)
factors.elementAt(t)).floatValue();

                                                            // cfactor is
the minimum value among fu, fv and ft
                                                            if(fu > fv)
cfactor = fv;
                                                            else cfactor =
fu;

                                                            if(cfactor > ft)
cfactor = ft;
```

```
      domains.put(is.fuzzify(),new Integer(d));
                                                              fis = new
Itemset(1);
      fis.addItem(is.fuzzify());


      fuzzy_candidates.add(fis);
                                                              factors.add(new
Float(cfactor)); //add itemset's factor in the same index
      ht_fis.put(is.fuzzify(), is.fuzzify());
                                                              }

      }catch(NullPointerException e){}

                                                    }
                                          }
                                    }
                  } // end for k
            } //end for domain

            for(int u = 0; u < fuzzy_candidates.size(); u++){
                  try{
                        candidates.add(fuzzy_candidates.get(u));
                        ht_candidates.add(candidates.size() - 1);
                  }catch(NullPointerException e){}

            }

            // sends to HashTree the fuzzy_candidates
            ht_candidates.getFuzzyCandidates(fuzzy_candidates);
      }

      public Vector getLargeItemsets() {
            // TODO Auto-generated method stub
            return large;
      }

      public Vector getNonLargeItemsets() {
            // TODO Auto-generated method stub
            return nonLarge;
      }
}
```

Classe AproriRules

```
/*
 * Created on 16/07/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
```

```
package aprioriminer;

/**
 * @author Administrator
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
import java.util.*;
import java.io.IOException;
import java.io.EOFException;

/*

  Maintenance log started on November 30th, 2000

  Nov. 30th, 2000   - fixed rule generation procedure to use
                                        all frequent itemsets, not only the
                                        maximal ones

*/

/**

   AprioriRules.java<P>

   This class implements the Apriori algorithm
   for finding association rules.

   (see "Fast Algorithms for Mining Association Rules"
   by Rakesh Agrawal and Ramakrishnan Srikant
   from IBM Almaden Research Center 1994)

*/
```

```
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public class AprioriRules implements AssociationsFinder
{
  private SET supports;
  private Vector rules;

  private float min_support;
  private float min_confidence;

  private Itemset is_in_antecedent;
  private Itemset is_in_consequent;
  private Itemset is_ignored;

  private int max_antecedent;
  private int min_consequent;

  // this method stores all frequent itemsets that have support
  // greater than the minimum support in a SET for more efficient
  // access times.
  private void initializeSupports(DBCacheReader cacheReader)
  {
        // create new SET
        supports = new SET();
```

```
        try
          {
        Itemset is;
        while (true)
          {
                // get item from cache
                is = cacheReader.getNextItemset();
                // if item has support greater than the minimum support
                // required then we add it to the SET
                if (is.getSupport() >= min_support)
                  {
                supports.insert(is);
                  }
          }
          }
        catch (EOFException e)
          {
        // do nothing, we just reached the EOF
          }
        catch (IOException e)
          {
        System.err.println("Error scanning cache!!!\n" + e);
          }
        catch (ClassNotFoundException e)
          {
        System.err.println("Error scanning cache!!!\n" + e);
          }
}

/**
 * Find association rules in a database, given the set of
 * frequent itemsets.
 *
 * @param cacheReader   the object used to read from the cache
```

```
 * @param minSupport   the minimum support
 * @param minConfidence   the minimum confidence
 * @return   a Vector containing all association rules found
 */
public Vector findAssociations(DBCacheReader cacheReader,
                               float minSupport,
                               float minConfidence)
{
        min_support = minSupport;
        min_confidence = minConfidence;

        // create the vector where we'll put the rules
        rules = new Vector();

        // read from cache supports of frequent itemsets
        initializeSupports(cacheReader);

        // get the frequent itemsets
        Vector auxFrequent = supports.getItemsets();
        Vector frequent = new Vector();
        for (int i = 0; i < auxFrequent.size(); i++) {
                Itemset item = (Itemset) auxFrequent.get(i);
                if (item.getSupport() != 0 /*&& item.size() <= 3*/)
                        frequent.add(item);
        }

        // generate rules from each frequent itemset
        for (int i = 0; i < frequent.size(); i++)
          {
        // get a frequent itemset
        Itemset is_frequent = (Itemset)frequent.get(i);

        // skip it if it's too small
        if (is_frequent.size() <= 1)
```

```
  continue;

// get all possible 1 item consequents
Vector consequents = new Vector(is_frequent.size());
for (int k = 0; k < is_frequent.size(); k++)
  {
        //int item = is_frequent.getItem(k);
        String item = is_frequent.getItem(k);
        Itemset is_consequent = new Itemset(1);
        is_consequent.addItem(item);

        // is_consequent now contains a possible consequent
        // verify now that the rule having this consequent
        // satisfies our requirements

        Itemset is_antecedent = is_frequent.subtract(is_consequent);

        float antecedent_support = 0;
        try
          {
        antecedent_support = supports.getSupport(is_antecedent);
          }
        catch (SETException e)
          {
        System.err.println("Error geting support from SET!!!\n" + e);
          }
        float confidence = 0;
        confidence = (is_frequent.getSupport() / antecedent_support);
        if (confidence > 1){
                String t = Float.toString(confidence);
                String conf = t.substring(0,t.length() - 1);
                confidence = Float.parseFloat(conf);
        }
```

```
                if (confidence >= min_confidence)
                  {
                consequents.add(is_consequent);

                // we add the rule to our collection if it satisfies
                // our conditions
                rules.add(new AssociationRule(is_antecedent, is_consequent,
                                              is_frequent.getSupport(),
                                              confidence));
                  }
          }

        // call the ap_genrules procedure for generating all rules
        // out of this frequent itemset
        ap_genrules(is_frequent, consequents);
          }

        return rules;
}

// this is the ap-genrules procedure that generates rules out
// of a frequent itemset.
private void ap_genrules(Itemset is_frequent, Vector consequents)
{
        if (consequents.size() == 0)
          return;

        // the size of frequent must be bigger than the size of the itemsets
        // in consequents by at least 2, in order to be able to generate
        // a rule in this call
        if (is_frequent.size() > ((Itemset)(consequents.get(0))).size() + 1)
          {
        Vector new_consequents = apriori_gen(consequents);
        AssociationRule ar;
```

```
        for (int i = 0; i < new_consequents.size(); i++)
          {
                Itemset is_consequent = (Itemset)new_consequents.get(i);
                Itemset is_antecedent = is_frequent.subtract(is_consequent);
                float antecedent_support = (float)0.00001;
                try
                  {
                antecedent_support = supports.getSupport(is_antecedent);
                  }
                catch (SETException e)
                  {
                System.err.println("Error geting support from SET!!!\n" + e);
                  }
                float confidence = is_frequent.getSupport() / antecedent_support;

                // if the rule satisfies our requirements we add it
                // to our collection
                if (confidence >= min_confidence)
                  rules.add(new AssociationRule(is_antecedent, is_consequent,
                                                is_frequent.getSupport(),
                                                confidence));
                // otherwise we remove the consequent from the collection
                // and we update the index such that we don't skip a consequent
                else
                  new_consequents.remove(i--);
          }

        ap_genrules(is_frequent, new_consequents);
          }
}

// this is the apriori_gen procedure that generates starting from
// a k-itemset collection a new collection of (k+1)-itemsets.
```

```
private Vector apriori_gen(Vector itemsets)
{
        if (itemsets.size() == 0)
          return new Vector(0);

        // create a hashtree so that we can check more efficiently the
        // number of subsets
        // this may not really be necessary when generating rules since
        // itemsets will probably be a small collection, but just in case
        HashTree ht_itemsets = new HashTree(itemsets);
        for (int i = 0; i < itemsets.size(); i++)
          ht_itemsets.add(i);
        ht_itemsets.prepareForDescent();

        Vector result = new Vector();
        Itemset is_i, is_j;
        for (int i = 0; i < itemsets.size() - 1; i++)
          for (int j = i + 1; j < itemsets.size(); j++)
        {
          is_i = (Itemset)itemsets.get(i);
          is_j = (Itemset)itemsets.get(j);

          // if we cannot combine element i with j then we shouldn't
          // waste time for bigger j's. This is because we keep the
          // collections ordered, an important detail in this implementation
          if (!is_i.canCombineWith(is_j))
                break;
          else
                {
                  Itemset is = is_i.combineWith(is_j);

                  // a real k-itemset has k (k-1)-subsets
                  // so we test that this holds before adding to result
                  if (ht_itemsets.countSubsets(is)
```

```
                == is.size())
                result.add(is);
                }
        }

        return result;
}

/**
 * Find association rules in a database, given the set of
 * frequent itemsets and a set of restrictions.
 *
 * @param cacheReader   the object used to read from the cache
 * @param minSupport   the minimum support
 * @param minConfidence   the minimum confidence
 * @param inAntecedent   the items that must appear in the antecedent
 * of each rule, if null then this constraint is ignored
 * @param inConsequent   the items that must appear in the consequent
 * of each rule, if null then this constraint is ignored
 * @param ignored   the items that should be ignored,
 * if null then this constraint is ignored
 * @param maxAntecedent   the maximum number of items that can appear
 * in the antecedent of each rule, if 0 then this constraint is ignored
 * @param minConsequent   the minimum number of items that should appear
 * in the consequent of each rule, if 0 then this constraint is ignored
 * @return   a Vector containing all association rules found
 */
public Vector findAssociations(DBCacheReader cacheReader,
                               float minSupport,
                               float minConfidence,
                               Itemset inAntecedent,
                               Itemset inConsequent,
                               Itemset ignored,
                               int maxAntecedent,
```

```
                                int minConsequent)
{
        min_support = minSupport;
        min_confidence = minConfidence;

        is_in_antecedent = inAntecedent;
        is_in_consequent = inConsequent;
        is_ignored = ignored;
        max_antecedent = maxAntecedent;
        min_consequent = minConsequent;

        // create the vector where we'll put the rules
        rules = new Vector();

        // read from cache supports of frequent itemsets
        initializeSupports(cacheReader);

        // get the frequent itemsets
        Vector frequent = supports.getItemsets();

        if (frequent.size() == 0)
          return rules;

        // if we need to ignore some items
        if (ignored != null)
          {
        // remove all frequent itemsets that contain
        // items to be ignored; their subsets that do
        // not contain those items will remain
        for (int i = 0; i < frequent.size(); i++)
          {
                Itemset is = (Itemset)frequent.get(i);
                if (is.doesIntersect(ignored))
                  {
```

```
            // replace this element with last, delete last,
            // and don't advance index
            frequent.set(i, frequent.lastElement());
            frequent.remove(frequent.size() - 1);
            i--;
              }
  }

if (frequent.size() == 0)
  return rules;
  }

// if we need to have some items in the antecedent or consequent
if (inAntecedent != null || inConsequent != null)
  {
// remove frequent itemsets that don't have the
// required items
for (int i = 0; i < frequent.size(); i++)
  {
            Itemset is = (Itemset)frequent.get(i);
            if (inAntecedent != null && !inAntecedent.isIncludedIn(is))
              {
            // replace this element with last, delete last,
            // and don't advance index
            frequent.set(i, frequent.lastElement());
            frequent.remove(frequent.size() - 1);
            i--;
              }
            else if (inConsequent != null && !inConsequent.isIncludedIn(is))
              {
            // replace this element with last, delete last,
            // and don't advance index
            frequent.set(i, frequent.lastElement());
            frequent.remove(frequent.size() - 1);
```

```
            i--;
              }
  }

if (frequent.size() == 0)
  return rules;
  }

// generate rules from each frequent itemset
for (int i = 0; i < frequent.size(); i++)
  {
// get a frequent itemset
Itemset is_frequent = (Itemset)frequent.get(i);

// skip it if it's too small
if (is_frequent.size() <= 1 ||
            is_frequent.size() <= minConsequent)
  continue;

// get all possible 1 item consequents
Vector consequents = new Vector(is_frequent.size());
for (int k = 0; k < is_frequent.size(); k++)
  {
            //int item = is_frequent.getItem(k);
            String item = is_frequent.getItem(k);
            Itemset is_consequent = new Itemset(1);
            is_consequent.addItem(item);

            // is_consequent now contains a possible consequent
            // verify now that the rule having this consequent
            // satisfies our requirements

            Itemset is_antecedent = is_frequent.subtract(is_consequent);
            float antecedent_support = (float)0.00001;
```

```
try
  {
antecedent_support = supports.getSupport(is_antecedent);
  }
catch (SETException e)
  {
System.err.println("Error geting support from SET!!!\n" + e);
  }
float confidence = is_frequent.getSupport() / antecedent_support;

if (confidence >= min_confidence)
  {
consequents.add(is_consequent);

// check whether it also satisfies our constraints
boolean approved = true;

if (approved && is_in_antecedent != null
        && !is_in_antecedent.isIncludedIn(is_antecedent))
  approved = false;

if (approved && is_in_consequent != null
        && !is_in_consequent.isIncludedIn(is_consequent))
  approved = false;

if (approved && max_antecedent > 0
        && is_antecedent.size() > max_antecedent)
  approved = false;

if (approved && min_consequent > 0
        && is_consequent.size() < min_consequent)
  approved = false;

// if the rule satisifes all requirements then
```

```
                // we add it to the rules collection
                if (approved)
                  rules.add(new AssociationRule(is_antecedent,
                                                is_consequent,
                                                is_frequent.getSupport(),
                                                confidence));
                }
          }

        // call the ap-genrules procedure for generating all rules
        // out of this frequent itemset
        ap_genrules_constraint(is_frequent, consequents);
          }

        return rules;
}

// this is the ap-genrules procedure that generates rules out
// of a frequent itemset.
private void ap_genrules_constraint(Itemset is_frequent, Vector consequents)
{
        if (consequents.size() == 0)
          return;

        // the size of frequent must be bigger than the size of the itemsets
        // in consequents by at least 2, in order to be able to generate
        // a rule in this call
        if (is_frequent.size() > ((Itemset)(consequents.get(0))).size() + 1)
          {
        Vector new_consequents = apriori_gen(consequents);
        AssociationRule ar;

        for (int i = 0; i < new_consequents.size(); i++)
          {
```

```
Itemset is_consequent = (Itemset)new_consequents.get(i);
Itemset is_antecedent = is_frequent.subtract(is_consequent);
float antecedent_support = (float)0.00001;
try
  {
antecedent_support = supports.getSupport(is_antecedent);
  }
catch (SETException e)
  {
System.err.println("Error geting support from SET!!!\n" + e);
  }
float confidence = is_frequent.getSupport() / antecedent_support;

// if the rule satisfies our confidence requirements
if (confidence >= min_confidence)
  {
// check whether it also satisfies our constraints
boolean approved = true;

if (approved && is_in_antecedent != null
        && !is_in_antecedent.isIncludedIn(is_antecedent))
  approved = false;

if (approved && is_in_consequent != null
        && !is_in_consequent.isIncludedIn(is_consequent))
  approved = false;

if (approved && max_antecedent > 0
        && is_antecedent.size() > max_antecedent)
  approved = false;

if (approved && min_consequent > 0
        && is_consequent.size() < min_consequent)
  approved = false;
```

```
                // if the rule satisifes all requirements then
                // we add it to the rules collection
                if (approved)
                  rules.add(new AssociationRule(is_antecedent,
                                                is_consequent,
                                                is_frequent.getSupport(),
                                                confidence));
                  }
                // otherwise we remove the consequent from the collection
                // and we update the index such that we don't skip a consequent
                else
                  new_consequents.remove(i--);
          }

        ap_genrules_constraint(is_frequent, new_consequents);
          }
  }
}
```

Classe AssociationRule

```
/*
 * Created on 16/07/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package aprioriminer;

/**

   AssociationRule.java<P>

   An association rule has two parts: the antecedent of the rule
   and the consequent of the rule, both of which are sets of items.
   Associated with these are a support and a confidence. The support
   tells how many rows of a database support this rule, the
   confidence tells what percentage of the rows that contain the
   antecedent also contain the consequent.

*/
/*
```

```
   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public class AssociationRule implements java.io.Serializable
{
  public static final int ANTECEDENT_SIZE = 1;
  public static final int CONSEQUENT_SIZE = 2;
  public static final int SUPPORT         = 3;
  public static final int CONFIDENCE      = 4;

  /**
   * The antecedent.
   *
   * @serial
   */
  //private int[] antecedent;
   private String[] antecedent;
  /**
   * The consequent.
   *
   * @serial
   */
  //private int[] consequent;
  private String[] consequent;

  /**
   * The support of the association rule.
   *
   * @serial
   */
  private float support;

  /**
   * The confidence of the association rule.
   *
   * @serial
   */
  private float confidence;

  /**
   * Creates a new association rule.
   *
   * @param antecedent   the antecedent of the association rule
   * @param consequent   the consequent of the association rule
   * @param support   the support of the association rule
   * @param confidence   the confidence of the association rule
   * @exception IllegalArgumentException   <code>antecedent</code>
   * or <code>consequent</code> are null or <code>support</code>
   * or <code>confidence</code> are not between 0 and 1
   */
  public AssociationRule(Itemset antecedent, Itemset consequent,
                  float support, float confidence)
  {
    if (antecedent == null || consequent == null
```

```
      || support < 0 || support > 1
      || confidence < 0 || confidence > 1)
      throw new IllegalArgumentException("constructor requires itemsets as
arguments");

    //Atencao gerar items frequentes comenter
    this.antecedent = new String[antecedent.size()];
    for (int i = 0; i < antecedent.size(); i++)
      this.antecedent[i] = antecedent.getItem(i);

    this.consequent = new String[consequent.size()];
    for (int i = 0; i < consequent.size(); i++)
      this.consequent[i] = consequent.getItem(i);

    this.support = support;

    this.confidence = confidence;
  }

  public AssociationRule(String antecedent, String consequent, float
support, float confidence) {
      // TODO Auto-generated constructor stub
        String[] ant = antecedent.split(",");
        this.antecedent = new String[ant.length];
        for (int i = 0; i < antecedent.split(",").length ; i++) {
            this.antecedent[i] = ant[i];
        }

        String[] cons = consequent.split(",");
        this.consequent = new String[cons.length];
        for (int i = 0; i < cons.length; i++) {
            this.consequent[i] = cons[i];
        }
        this.support = support;
        this.confidence = confidence;
}

/**
   * Return size of antecedent.
   *
   * @return   size of antecedent
   */
  public int antecedentSize()
  {
    return antecedent.length;
  }

  /**
   * Return size of consequent.
   *
   * @return   size of consequent
   */
  public int consequentSize()
  {
    return consequent.length;
  }

  /**
   * Return support of association rule.
   */
  public float getSupport()
```

```
  {
    return support;
  }

  /**
   * Return confidence of association rule.
   */
  public float getConfidence()
  {
    return confidence;
  }

  /**
   * Return i-th item in antecedent.
   *
   * @param i   the index of the item to get
   * @exception IndexOutOfBoundsException   <code>i</code> is an invalid
index
   * @return   the <code>i</code>-th item in antecedent
   */
  public String getAntecedentItem(int i)
  {
    if (i < 0 || i >= antecedent.length)
      throw new IndexOutOfBoundsException("invalid index");

    return antecedent[i];
  }

  /**
   * Return i-th item in consequent.
   *
   * @param i   the index of the item to get
   * @exception IndexOutOfBoundsException   <code>i</code> is an invalid
index
   * @return   the <code>i</code>-th item in consequent
   */
  public String getConsequentItem(int i)
  {
    if (i < 0 || i >= consequent.length)
      throw new IndexOutOfBoundsException("invalid index");

    return consequent[i];
  }

  /**
   * Compare two AssociationRule objects on one of several criteria.
   *
   * @param ar   the AssociationRule object with which we want to
   * compare this object
   * @param criteria   the criteria on which we want to compare, can
   * be one of ANTECEDENT_SIZE, CONSEQUENT_SIZE, SUPPORT or CONFIDENCE.
   * @exception IllegalArgumentException   <code>ar</code> is null
   * or criteria is invalid
   * @return   a negative value if this object is smaller than
   * <code>ar</code>, 0 if they are equal, and a positive value if this
   * object is greater.
   */
  public int compareTo(AssociationRule ar, int criteria)
  {
    if (ar == null)
```

```
      throw new IllegalArgumentException("method requires association rule
as argument");

    float diff;

    if (criteria == ANTECEDENT_SIZE)
      return this.antecedent.length - ar.antecedent.length;
    else if (criteria == CONSEQUENT_SIZE)
      return this.consequent.length - ar.consequent.length;
    else if (criteria == SUPPORT)
      diff = this.support - ar.support;
    else if (criteria == CONFIDENCE)
      diff = this.confidence - ar.confidence;
    else
      throw new IllegalArgumentException("invalid criteria");

    if (diff < 0)
      return -1;
    else if (diff > 0)
      return 1;
    else
      return 0;
  }

  /**
   * Compare two AssociationRule objects on one of several criteria.
   *
   * @param ar   the AssociationRule object with which we want to
   * compare this object
   * @param criteria   the criteria on which we want to compare, can
   * be one of ANTECEDENT_SIZE, CONSEQUENT_SIZE, SUPPORT or CONFIDENCE.
   * @return   true if the objects are equal in terms of antecedent
   * and consequent items; false otherwise.
   */
  public boolean equals(Object obj)
  {
    if (!(obj instanceof AssociationRule) || obj == null)
      return false;

    AssociationRule other = (AssociationRule)obj;

    if (antecedent.length != other.antecedent.length)
      return false;

    if (consequent.length != other.consequent.length)
      return false;

    for (int i = 0; i < antecedent.length; i++)
      if (antecedent[i].compareTo(other.antecedent[i]) != 0)
      return false;

    for (int i = 0; i < consequent.length; i++)
      if (consequent[i].compareTo(other.consequent[i]) != 0)
      return false;

    return true;
  }

  /**
   * Return a String representation of the AssociationRule.
   *
```

```
   * @return   String representation of AssociationRule
   */
  public String toString()
  {
    String s = "{";

    for (int i = 0; i < antecedent.length; i++)
      s += antecedent[i] + " ";
    s += "}->{";

    for (int i = 0; i < consequent.length; i++)
      s += consequent[i] + " ";
    s += "} (" + support + ", " + confidence + ")";

    return s;
  }

  /**
   * for testing purposes only !!!
   */
  public static void main(String[] args)
  {
   /* Itemset is1 = new Itemset();
    Itemset is2 = new Itemset();

    is1.addItem(7);
    is1.addItem(3);
    is1.addItem(15);

    System.out.println("is1: " + is1);

    is2.addItem(12);
    is2.addItem(5);
    is2.addItem(8);

    System.out.println("is2: " + is2);

    AssociationRule ar = new AssociationRule(is1, is2,
                                  (float)0.5055,
                                  (float)0.3033);

    System.out.println("ar: " + ar);*/
  }
}
```

Classe AssociationsFinder

```
/*
 * Created on 16/07/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
```

```
package aprioriminer;

/**
 * @author Administrator
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
import java.util.*;

/**

   AssociationsFinder.java<P>

   This interface must be implemented by the algorithms that will look
   for associations.

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public interface AssociationsFinder
{
  /**
   * Find association rules in a database, given the set of
   * frequent itemsets.
```

```
 *
 * @param cacheReader   the object used to read from the cache
 * @param minSupport   the minimum support
 * @param minConfidence   the minimum confidence
 * @return   a Vector containing all association rules found
 */
Vector findAssociations(DBCacheReader cacheReader,
                        float minSupport,
                        float minConfidence);

/**
 * Find association rules in a database, given the set of
 * frequent itemsets and a set of restrictions.
 *
 * @param cacheReader   the object used to read from the cache
 * @param minSupport   the minimum support
 * @param minConfidence   the minimum confidence
 * @param inAntecedent   the items that must appear in the antecedent
 * of each rule
 * @param inConsequent   the items that must appear in the consequent
 * of each rule
 * @param ignored   the items that should be ignored
 * @param maxAntecedent   the maximum number of items that can appear
 * in the antecedent of each rule
 * @param minConsequent   the minimum number of items that should appear
 * in the consequent of each rule
 * @return   a Vector containing all association rules found
 */
Vector findAssociations(DBCacheReader cacheReader,
                        float minSupport,
                        float minConfidence,
                        Itemset inAntecedent,
                        Itemset inConsequent,
                        Itemset ignored,
```

int maxAntecedent,

int minConsequent);

}

Classe DBCacheReader

```
package aprioriminer;


import java.io.*;

/**

   DBCacheReader.java<P>

   A DBCacheReader deserializes itemsets from cache.<P>

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public class DBCacheReader
{
  private ObjectInputStream instream ;
  private String filename;

  /**
   * Initializes a DBCacheReader to read from the specified cache file.
   *
   * @param name   name of the cache file
   * @exception IllegalArgumentException   <code>name</code> is null
   * @exception IOException   from java.io package
   */
  public DBCacheReader(String name)
    throws IOException
  {
    if (name == null)
      throw new IllegalArgumentException("Constructor argument must be non
null");

    filename = name;
    instream = new ObjectInputStream(new FileInputStream(filename));
  }

  /**
   * Closes the cache file.
   *
   * @exception IOException   from java.io package
```

```
   */
  public void close()
    throws IOException
  {
    instream.close();
  }

  /**
   * Return the first itemset from cache.
   *
   * @exception IOException   from java.io package
   * @exception ClassNotFoundException   from java.io package
   * @return   first itemset in cache
   */
  public Itemset getFirstItemset()
    throws IOException, ClassNotFoundException
  {
    instream.close();
    instream = new ObjectInputStream(new FileInputStream(filename));
    return ((Itemset)instream.readObject());
  }

  /**
   * Return next itemset from cache.
   *
   * @exception IOException   from java.io package
   * @exception ClassNotFoundException   from java.io package
   * @return   next itemset in cache
   */
  public Itemset getNextItemset()
    throws IOException, ClassNotFoundException
  {
    return ((Itemset)instream.readObject());
  }

  /**
   * for testing purposes only !!!
   */
  public static void main(String[] args)
  {
    try
      {
      DBCacheReader dbcache = new DBCacheReader("test.cache");

      try
        {
          while (true)
            System.out.println(dbcache.getNextItemset());
        }
      catch (EOFException e)
        {
          dbcache.close();
        }
      }
    catch (Exception e)
      {
      System.out.println(e);
      }
  }
}
```

Classe DBCacheWriter

```
package aprioriminer;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author unascribed
 * @version 1.0
 */

import java.io.*;

/**

   DBCacheWriter.java<P>

   A DBCacheWriter serializes itemsets to a cache.<P>

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public class DBCacheWriter
{
  private ObjectOutputStream outstream;
  private String filename;

  /**
   * Initializes a DBCacheWriter to write to the specified cache file.
   *
   * @param name   name of the cache file
   * @exception IllegalArgumentException   <code>name</code> is null
   * @exception IOException   from java.io package
   */
  public DBCacheWriter(String name)
    throws IOException
  {
    if (name == null)
      throw new IllegalArgumentException("Constructor argument must be non
null");

    filename = name;
    outstream = new ObjectOutputStream(new FileOutputStream(filename));
  }

  /**
   * Closes the cache file.
```

```
   *
   * @exception IOException   from java.io package
   */
  public void close()
    throws IOException
  {
    outstream.close();
  }

  /**
   * Write an itemset to the cache.
   *
   * @exception IOException   from java.io package
   */
  public void writeItemset(Itemset is)
    throws IOException
  {
    outstream.writeObject(is);
  }

  /**
   * for testing purposes only !!!
   */
  public static void main(String args[])
  {
/*   try
     {
     DBCacheWriter dbcache = new DBCacheWriter("test.cache");

     Itemset is = new Itemset();

     is.addItem(1);
     is.addItem(13);
     is.setSupport((float)3.2);
     dbcache.writeItemset(is);

     is = new Itemset();
     is.addItem(2);
     is.addItem(7);
     is.setSupport((float)44.72);
     dbcache.writeItemset(is);

     is = new Itemset();
     is.addItem(10);
     is.addItem(5);
     is.addItem(5);
     is.addItem(7);
     is.setSupport((float)13.2);
     dbcache.writeItemset(is);

     is = new Itemset();
     is.addItem(51);
     is.addItem(13);
     is.setSupport((float)3.33);
     dbcache.writeItemset(is);

     dbcache.close();
     }
   catch (IOException e)
     {
     System.out.println(e);
```

```
      }*/
  }
}
```

Classe DBExcpetion

```
package aprioriminer;


public class DBException extends Exception
{
  /**
   * Creates a new DBException with description string s.
   * @param msg   the error message associated with the exception
   */
  public DBException(String msg)
  {
    super(msg);
  }
}
```

Classe DBReader

```
package aprioriminer;

import java.io.*;
import java.util.StringTokenizer;
import java.util.Hashtable;

/**

  DBReader.java<P>

  A DBReader is used to read data from a database.<P>

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/
/*
  Database header specification:

  1. Identifier - 3 characters - CKL
  2. Database format version number - 3 integers - 1 12 3 (meaning 1.12.3)
  3. Header size - 1 long - the size of the header in bytes, lets us know
  where the data starts
  4. Number of rows - 1 long
  5. Number of columns - 1 long
  6. Column names - fixed length fields (32 characters). There are as many
```

```
  such entries as we have columns in the database.
  7. Identifier - 3 characters - CKL : for verification only, it indicates
  the end of the column names enumeration.
  8. Database description - fixed length field (256 characters).
  9. Identifier - 3 characters - CKL : again for verifying the end of the
  description
  10. CRC field - 1 int - CRC of the data part of the file only

  Database contents: each row consists of ints: no_items, item 1, ... item
n
*/

public class DBReader {
      private static final int CHAR_SIZE = 2;
      private static final int INT_SIZE = 4;
      private static final int LONG_SIZE = 8;
      private static final int CRC_SIZE = INT_SIZE;

      private static final int COLUMN_LENGTH = 32;
      private static final int DESCRIPTION_LENGTH = 256;

      private static final int ID_SIZE = CHAR_SIZE * 3;
      private static final int VERSION_SIZE = INT_SIZE * 3;
      private static final int HEADER_SIZE = LONG_SIZE;
      private static final int NUMROWS_SIZE = LONG_SIZE;
      private static final int NUMCOLUMNS_SIZE = LONG_SIZE;
      private static final int COLUMN_SIZE = COLUMN_LENGTH * CHAR_SIZE;
      private static final int DESCRIPTION_SIZE = DESCRIPTION_LENGTH *
CHAR_SIZE;

      private static final int HEAD_SIZE_OFFSET = ID_SIZE + VERSION_SIZE;
      private static final int NUMROWS_OFFSET = HEAD_SIZE_OFFSET +
HEADER_SIZE;
      private static final int COLUMN_NAME_OFFSET =
            NUMROWS_OFFSET + NUMROWS_SIZE + NUMCOLUMNS_SIZE;



      // private RandomAccessFile inStream;
      private FileReader br;
      private BufferedReader inStream;

      private long numRows;
      private boolean needReposition;
      private long lastPosition;
      private long currRow;
      private long nbrCol;
      private String fileName;

      // The items' domains.
      private Hashtable domains;

      // Domain names
      private String [] DN;

      /**
       * Create a new DBReader according to the input file name.
       *
       * @param fileName   the name of the file
       * @exception  IOException   from library call
       */
```

```
    public DBReader(String fileName) throws IOException {
          this.fileName = fileName;
          br = new FileReader(fileName);
          numRows = countLines();
          BufferedReader inStream = new BufferedReader(br);
          currRow = 1;
          needReposition = false;
          domains = new Hashtable();

    }

    public long getNumColumns(){
          return this.nbrCol;
    }

public long getNbrColLine(String line){
     int nbrcol=1;
     for(int i=0; i<line.length(); i++){
          if(line.charAt(i)==';')
          nbrcol++;
     }
     return nbrcol;

}


    public long countLines() {
          int nbrlines = 0;
          try {

                BufferedReader inStream = new BufferedReader(br);
                try {
                      String line = inStream.readLine();
                      nbrCol = getNbrColLine(line);
                      nbrlines = 1;
                      while (inStream.readLine()!=null) {
                            nbrlines++;
                      }
                } catch (IOException e) {
                } //ok just find the end of the document
                inStream.close();
          } catch (Exception e) {
                e.printStackTrace();
          }

          return nbrlines++;
    }

    /**
     * Close the I/O stream.
     */
    public void close() throws IOException {
          inStream.close();
    }

    /**
     * Get the number of the rows in database.
     *
     * @return   number of rows in database
     */
    public long getNumRows() {
```

```
            return numRows;
      }

      /**
       * Get first row of the table from the data file.
       *
       * @exception  IOException   from library call
       * @exception  DBException   an invalid item has been met
       * @return   first Itemset from database
       */
      /**
       * @return
       * @throws IOException
       * @throws DBException
       */
      /**
       * @return
       * @throws IOException
       * @throws DBException
       */
      public Itemset getFirstRow() throws IOException, DBException {
            currRow = 1;

            br.close();
            br = new FileReader(fileName);
            inStream = new BufferedReader(br);
            //inStream.seek(0);
            //    inStream = new BufferedReader(new
InputStreamReader(file_in));
            //inStream.readLine(); //numero de linhas
            //inStream.readLine(); //numero de colunas
            //    inStream.readLine(); //numero de nome das colunas
            needReposition = false;

            // skip column names in the first line
            inStream.readLine();
            currRow++;

            return getNextRow();
      }

      /**
       * Get first row of the table from the data file.
       * (Modification for similarities)
       *
       * @exception  IOException   from library call
       * @exception  DBException   an invalid item has been met
       * @return   first similarity relation from file
       */
      public String[] getFirstSim() throws IOException, DBException {
            currRow = 1;

            br.close();
            br = new FileReader(fileName);
            inStream = new BufferedReader(br);
            needReposition = false;
            return getNextSim();
      }

      /**
       * Get first row of the table from the data file, which contains
```

```
 * column headers (domain names).
 *
 * @exception  IOException   from library call
 * @exception  DBException   an invalid item has been met
 * @return   domain names on DN
 */
public String[] getDomainNames() throws IOException, DBException {
      currRow = 1;

      br.close();
      br = new FileReader(fileName);
      inStream = new BufferedReader(br);
      needReposition = false;

      // Get first line, i.e., domain names
      StringTokenizer line =
            new StringTokenizer(inStream.readLine().trim(), ";");
      int numItems = line.countTokens();

      DN = new String[numItems];

      // adding each domain name into the Vector DN
      for(int k = 0; k < numItems; k++){
            DN[k] = line.nextToken();
            //System.out.println("Domain "+ k + ": " + DN[k]);
      }

      return DN;
}

/**
 * Get the next row of data since last reading.
 *
 * @exception  IOException   from library call
 * @exception  DBException   an invalid item has been met
 */
public Itemset getNextRow() throws IOException, DBException {
            StringTokenizer line =
            new StringTokenizer(inStream.readLine().trim(), ";");
      int numItems = line.countTokens();
      String item;
      Itemset is = new Itemset(numItems);

      for (int i = 0; i < numItems; i++) {
            //item = Integer.parseInt(line.nextToken());
            item = line.nextToken();
            is.addItem(item);
            domains.put(item, new Integer(i));
      }

      currRow++;

      return is;
}

/**
 * Get the next row of data since last reading.
 * (Modification for similarities)
 *
 * @exception  IOException   from library call
 * @exception  DBException   an invalid item has been met
```

```
	 */
	public String[] getNextSim() throws IOException, DBException {
			StringTokenizer line =
			new StringTokenizer(inStream.readLine().trim(), ";");
		int numItems = line.countTokens();
		String item;
		String []s = new String[numItems];

		for (int i = 0; i < numItems; i++) {
			//item = Integer.parseInt(line.nextToken());
			item = line.nextToken();
			s[i] = item;
		}

		currRow++;

		return s;
	}

	/**
	 * Tell whether there are more rows to be read from the database.
	 *
	 * @return   true if there are more rows, false otherwise
	 */
	public boolean hasMoreRows() {
		if (currRow <= numRows)
			return true;
		else
			return false;
	}

	/**
	 * Returns the hashtable 'domains'.
	 *
	 * @return   Hashtable
	 */
	public Hashtable getDomains() {
		return domains;
	}

	public static void main(String arg[]) {
		try {

			DBReader x = new DBReader("base2.txt");
			System.out.println(x.getFirstRow());
			for (int i = 1; i < x.numRows; i++)
				System.out.println(x.getNextRow());
		} catch (Exception e) {
			e.printStackTrace();
		}

	}
}
```

Classe DBWriter

```
package aprioriminer;
```

```
import java.io.*;
import java.util.Vector;

/**

  DBWriter.java<P>

  A DBWriter is used to write itemsets into a database.

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/
/*
  Database header specification:

  1. Identifier - 3 characters - CKL
  2. Database format version number - 3 integers - 1 12 3 (meaning 1.12.3)
  3. Header size - 1 long - the size of the header in bytes, lets us know
  where the data starts
  4. Number of rows - 1 long
  5. Number of columns - 1 long
  6. Column names - fixed length fields (32 characters). There are as many
  such entries as we have columns in the database.
  7. Identifier - 3 characters - CKL : for verification only, it indicates
  the end of the column names enumeration.
  8. Database description - fixed length field (256 characters).
  9. Identifier - 3 characters - CKL : again for verifying the end of the
  description
  10. CRC field - 1 int - CRC of the data part of the file only

  Database contents: each row consists of ints: no_items, item 1, ... item
n
*/

public class DBWriter
{
  private static final int CHAR_SIZE          = 2;
  private static final int INT_SIZE           = 4;
  private static final int LONG_SIZE          = 8;
  private static final int CRC_SIZE           = INT_SIZE;

  private static final int COLUMN_LENGTH      = 32;
  private static final int DESCRIPTION_LENGTH = 256;

  private static final int ID_SIZE            = CHAR_SIZE * 3;
  private static final int VERSION_SIZE       = INT_SIZE * 3;
  private static final int HEADER_SIZE        = LONG_SIZE;
  private static final int NUMROWS_SIZE       = LONG_SIZE;
  private static final int NUMCOLUMNS_SIZE    = LONG_SIZE;
  private static final int COLUMN_SIZE        = COLUMN_LENGTH * CHAR_SIZE;
  private static final int DESCRIPTION_SIZE   = DESCRIPTION_LENGTH *
CHAR_SIZE;
```

```
  private static final int HEAD_SIZE_OFFSET   = ID_SIZE + VERSION_SIZE;
  private static final int NUMROWS_OFFSET     = HEAD_SIZE_OFFSET +
HEADER_SIZE;
  private static final int COLUMN_NAME_OFFSET = NUMROWS_OFFSET +
NUMROWS_SIZE
    + NUMCOLUMNS_SIZE;

  private static final String ID              = "CKL";

  private RandomAccessFile outStream;
  private String description;
  private long numRows;
  private long numColumns;
  private int CRC;
  private long headerSize;
  private boolean wroteColumnNames;
  private boolean needReposition;
  private long lastPosition;

  // call this method to check the existence of the ID
  // the file pointer must be already positioned
  private void checkID()
    throws IOException
  {
    for (int i = 0; i < ID.length(); i++)
      if (outStream.readChar() != ID.charAt(i))
      throw new IOException("Attempting to load invalid database");
  }

  // call this method to write the column names to the file
  // the file pointer must be already positioned
  private void writeColumnNames(Vector names)
    throws IOException
  {
    String column;
    for (int i = 0, numCols = names.size(); i < numCols; i++)
      {
      column = (String)names.get(i);
      if (column.length() > COLUMN_LENGTH)
        column = column.substring(0, COLUMN_LENGTH);

      outStream.writeChars(column);
      for (int j = 0; j < (COLUMN_LENGTH - column.length()); j++)
        outStream.writeChar(' ');
      }
  }

  // call this method to write the description to the file
  // the file pointer must be already positioned
  // the description must have been set
  private void writeDescription()
    throws IOException
  {
    if (description.length() > DESCRIPTION_LENGTH)
      description = description.substring(0, DESCRIPTION_LENGTH);

    outStream.writeChars(description);
    for (int j = 0; j < (DESCRIPTION_LENGTH - description.length());
       j++)
      outStream.writeChar(' ');
  }
```

```
/**
 * Create a new DBWriter according to the input file name.
 *
 * @param fileName   the name of the file
 * @exception FileNotFoundException   from library call
 * @exception  IOException   from library call or if file is corrupted
 */
public DBWriter(String fileName)
  throws IOException
{
  outStream = new RandomAccessFile(fileName, "rw");

  lastPosition = outStream.length();

  // case this is a new database
  if (lastPosition == 0)
    {
    wroteColumnNames = false;

    headerSize     = 0;
    numRows        = 0;
    CRC            = 0;
    needReposition = false;
    }
  // case this might be an old database
  else if (lastPosition > COLUMN_NAME_OFFSET)
    {
    // check that we have a valid database here
    checkID();

    wroteColumnNames = true;

    // read header data
    outStream.seek(HEAD_SIZE_OFFSET);
    headerSize = outStream.readLong();
    numRows = outStream.readLong();
    numColumns = outStream.readLong();
    outStream.seek(headerSize - CRC_SIZE);
    CRC = outStream.readInt();
    needReposition = true;
    }
  // case this is not a database
  else
    throw new IOException("Attempting to load invalid database");
}

/**
 * Set the column names for the database.
 *
 * @param names   the column names
 * @exception  IOException   from library call
 * @exception  DBException   size of <code>names</code> does not match
 * number of columns
 */
public void setColumnNames(Vector names)
  throws IOException, DBException
{
  long namesSize = names.size();

  if (wroteColumnNames == false)
```

```
  {
  numColumns = namesSize;

  // ID
  outStream.writeChars(ID);

  // version number
  outStream.writeInt(1);
  outStream.writeInt(0);
  outStream.writeInt(0);

  headerSize = COLUMN_NAME_OFFSET + COLUMN_SIZE * numColumns
    + ID_SIZE + DESCRIPTION_SIZE + ID_SIZE + CRC_SIZE;
  outStream.writeLong(headerSize);

  // number of rows
  outStream.writeLong(0);

  // number of columns
  outStream.writeLong(numColumns);

  // columns
  writeColumnNames(names);

  // rest of the header
  outStream.writeChars(ID);

  // check if description has been set, otherwise fill in with blanks
  if (description != null)
    writeDescription();
  else
    {
      description = "";
      for (int i = 0; i < DESCRIPTION_LENGTH; i++)
        description += ' ';
      writeDescription();
    }

  outStream.writeChars(ID);

  outStream.writeInt(0);

  wroteColumnNames = true;
  }
else
  {
  if (namesSize != numColumns)
    throw new DBException("Cannot change the number of columns");
  else
    {
      if (needReposition == false)
        {
        lastPosition = outStream.getFilePointer();
        needReposition = true;
        }

      outStream.seek(COLUMN_NAME_OFFSET);

      writeColumnNames(names);
    }
  }
```

```
}

/**
 * Set the description of the database.
 *
 * @param description   the description of the database
 * @exception  IOException   from library call
 */
public void setDescription (String description)
  throws IOException
{
  this.description = description;

  if (wroteColumnNames)
    {
    if (needReposition == false)
      {
        lastPosition = outStream.getFilePointer();
        needReposition = true;
      }

    outStream.seek(headerSize - DESCRIPTION_SIZE - ID_SIZE - CRC_SIZE);

    writeDescription();
    }
}

/**
 * Add a new row to the database. If this is to be the first row
 * added to the database you must have called setColumnNames()
 * before.
 *
 * @param itemset   the new row to be added to the data file
 * @exception  IOException   from library call
 * @exception  DBException   column names have not been set
 * or an invalid item was contained in the itemset
 */
public void addRow(Itemset itemset)
  throws IOException, DBException
{
  if (wroteColumnNames == false)
    throw new DBException("Column names must be set first");

  Itemset is = new Itemset(itemset);
 /* while (is.hasMoreItems())
    if (is.getNextItem() > numColumns)
    throw new DBException("Attempt to write invalid item");*/

  if (needReposition == true)
    {
    outStream.seek(lastPosition);
    needReposition = false;
    }

  is = new Itemset(itemset);
  String item;

  int isSize = is.size();
  outStream.writeInt(isSize);
  CRC = updateCRC(CRC, isSize);
  while (is.hasMoreItems())
```

```
    {
    item = is.getNextItem();
    //outStream.writeInt(item);
    outStream.writeChars(item);
    //CRC = updateCRC(CRC, item);
    }

  numRows++;
}

/**
 * Close the I/O stream and save any unsaved data.
 *
 * @exception  IOException from library call
 */
public void close()
  throws IOException
{
  // write number of rows, CRC, and description
  outStream.seek(NUMROWS_OFFSET);
  outStream.writeLong(numRows);

  outStream.seek(headerSize - CRC_SIZE);
  outStream.writeInt(CRC);

  outStream.seek(headerSize - DESCRIPTION_SIZE - ID_SIZE - CRC_SIZE);
  writeDescription();

  outStream.close();
}

/**
 * Update a CRC-16 value.
 *
 * @param crc   the previous CRC value
 * @param value   the value for which we update the CRC
 * @return   the updated CRC value
 */
public static synchronized int updateCRC(int crc, int value)
{
  int mask;
  int i;

  mask = 0xA001;
  value <<= 8;

  for (i = 0; i < 8; i++)
    {
    if (((crc ^ value) & 0x8000) > 0)
      crc = (crc << 1) ^ mask;
    else
      crc <<= 1;

    value <<= 1;
    }

  return crc & 0xFFFF;
}

public static void main(String [] args)
{
```

```
    Itemset is1 = new Itemset();
    is1.addItem("a");
    is1.addItem("b");
    Itemset is2 = new Itemset();
    is2.addItem("c");
    is2.addItem("d");
    Itemset is3 = new Itemset();
    is3.addItem("c");
    is3.addItem("a");
    Itemset is4 = new Itemset();
    is4.addItem("cc");
    is4.addItem("c");

    Vector colNames = new Vector(3);
    colNames.add("cheese");
    colNames.add("pizza");
    colNames.add("beer");

    System.out.println("\n\nCreating invalid database:");
    try
      {
      RandomAccessFile invalid = new RandomAccessFile("invalid", "rw");
      invalid.writeChars(ID + " - a bogus file that looks like a valid
one");
      invalid.close();
      }
    catch (Exception e)
      {
      System.out.println("Shouldn't have happened: " + e);
      }

    System.out.println("\n\nCreating corrupted database:");
    try
      {
      DBWriter corrupted = new DBWriter("corrupted");

      try
        {
          corrupted.addRow(is1);
        }
      catch (DBException e)
        {
          System.out.println(e);
        }

      corrupted.setDescription("a corrupted database");
      corrupted.setColumnNames(colNames);

      corrupted.addRow(is1);
      corrupted.setDescription("a corrupted database - 2");
      corrupted.setColumnNames(colNames);
      corrupted.addRow(is2);
      corrupted.setDescription("a corrupted database - 3");
      corrupted.addRow(is3);

      try
        {
          corrupted.addRow(is4);
        }
      catch (DBException e)
        {
```

```
      System.out.println(e);
    }

  corrupted.close();

  System.out.println("corrupting file");

  RandomAccessFile raf = new RandomAccessFile("corrupted", "rw");
  raf.seek(770);
  // replace the 2 in the second itemset with a 3
  raf.writeInt(3);
  raf.close();
  }
catch (Exception e)
  {
  System.out.println("Shouldn't have happened: " + e);
  }

System.out.println("\n\nCreating empty database:");
try
  {
  DBWriter empty = new DBWriter("empty");

  empty.setDescription("an empty database");
  empty.setColumnNames(colNames);
  empty.close();
  }
catch (Exception e)
  {
  System.out.println("Shouldn't have happened: " + e);
  }

System.out.println("\n\nCreating correct database:");
try
  {
  DBWriter correct = new DBWriter("correct");

  correct.setDescription("a correct database");
  correct.setColumnNames(colNames);

  correct.addRow(is1);
  correct.setDescription("a correct database - 2");
  correct.setColumnNames(colNames);
  correct.addRow(is2);
  correct.setDescription("a correct database - 3");
  correct.addRow(is3);

  correct.close();

  correct = new DBWriter("correct");

  correct.setColumnNames(colNames);

  correct.addRow(is1);
  correct.setDescription("a correct database - 4");
  correct.setColumnNames(colNames);
  correct.addRow(is2);
  correct.setDescription("a correct database - 5");
  correct.addRow(is3);

  correct.close();
```

```
      }
    catch (Exception e)
      {
      System.out.println("Shouldn't have happened: " + e);
      }
  }
}
```

Classe Domain

```
/*
 * Created on 08/12/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package aprioriminer;

/**
 * @author Administrator
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
public class Domain {

      private int size = 20; //Default Number of Items in Domain

      String[] items;

      // 'pos' is the next position where a item can be added in the
'items' array
      private int pos = 0;   //initial position in the array

      boolean used; // Flag to know if Domain is used or not

      private SimilarityMatrix SM;

      public Domain() {
                  SM = new SimilarityMatrix(size);
                  items = new String[size];
                  used = false;
            }

      public Domain(int size) {
            SM = new SimilarityMatrix(size);
            items = new String[size];
            this.size = size;
            used = false;
      }

      /**
       * Return the Domain's size.
       *
       * @return size   (integer)
       */
      public int size(){
            return size;
```

```
}

/**
 * Add a item in the array items[].
 *
 * @param item1   the first item to be compared in the matrix
 * @param item2   the second item to be compared in the matrix
 * @return        If successful, return true. Else, return false.

 */
public boolean addItem(String item){

      if(pos < size && pos >= 0){
            items[pos] = item;
            pos++;
            return true;
      }else return false;
}

/**
 * Get the Item's position in the array items[].
 *
 * @param item   the item
 * @return       the position (integer)
 */
public int getItemPosition(String item){

      int i = 0;
      for(int j = 0; j < pos; j++){
            if(items[j].compareTo(item) == 0) i = j;
      }

      return i;
}

public String[] getItemsDomain(String domain){
      String[] s = null;
      for (int i = 0; i < items.length; i++){
            s[i] = items[i];
      }

      return s;
}

/**
 * Add the similarity value in the matrix.
 *
 * @param item1   the first item to be compared in the matrix
 * @param item2   the second item to be compared in the matrix
 * @param sim       the similarity value (float)
 */
public void addSimilarity(String item1, String item2, float sim){

      SM.add(getItemPosition(item1),getItemPosition(item2),sim);

}

/**
 * Get the similarity value from the matrix.
 *
 * @param item1   the first item to be compared in the matrix
```

```
         * @param item2   the second item to be compared in the matrix
         * @return        the similarity value (float)
         */
        public float getSimilarity(String item1, String item2){
              return SM.get(getItemPosition(item1),getItemPosition(item2));
        }

        /**
         * Tests if a item belongs to domain.
         *
         * @param item   the item to be tested (string)
         * @return       if belongs, true. Else, false.
         */
        public boolean containsItem(String item){

              boolean test = false;
              for(int j = 0; j < pos; j++){
                    if(items[j].compareTo(item) == 0) test = true;
              }
              return test;
        }

        public boolean isUsed(){
              return used;
        }

        public void setUsed(){
              used = true;
        }
}
```

Classe FuzzyItemsFinder

```
package aprioriminer;

/**

   FuzzyItemsetsFinder.java<P>

   This interface must be implemented by the algorithms that will look
   for fuzzy itemsets.

*/

public interface FuzzyItemsetsFinder
{
  /**
   * Find the frequent fuzzy itemsets in a database
   *
   * @param dbReader   the object used to read from the database
   * @param cacheWriter   the object used to write to the cache
   * if this is null, then nothing will be saved, this is useful
   * for benchmarking
   * @param minSupport   the minimum support
   * @param minSim            the minimum similarity
   * @param ontologyURL  the ontology identifier
   * @param ontologyPath the path to the owl file
   * @return   the number of passes executed over the database
```

```
   */
  int findLargeItemsets(DBReader dbReader,
                  DBCacheWriter cacheWriter,
                  float minSupport,
                  float minSim,
                  String ontologyURL,
                  String ontologyPath);
//                DBReader similarities);
}
```

Classe GeneralizedExceptFor

```
package aprioriminer;

import java.util.Vector;

public class GeneralizedExceptFor {

      private AssociationRule assoc;
      private Vector except;
      public GeneralizedExceptFor(AssociationRule assoc, Vector except) {
            super();
            // TODO Auto-generated constructor stub
            this.assoc = assoc;
            this.except = except;
      }
      public AssociationRule getAssoc() {
            return assoc;
      }
      public Vector getExcept() {
            return except;
      }
}
```

Classe HashTree

```
package aprioriminer;

import java.util.*;

/*

  Maintenance log started on April 3rd, 2001 by Laurentiu Cristofor

  Apr. 3rd, 2001    - modified the HashTree so that it can change
                      dynamically its HashNode size in order to
                      index large numbers of itemsets.

*/

/**

   HashTree.java<P>

   A HashTree is a special data structure that is used to index
```

```
   a Vector of Itemset objects for more efficient processing.

*/

public class HashTree
{
  private static final int LIST_NODE = 1;
  private static final int HASH_NODE = 2;

  private static final int DEFAULT_LIST_SIZE = 20;//original = 20
  private static final int DEFAULT_HASH_SIZE = 40;//original = 40

  private int LIST_SIZE;
  private int HASH_SIZE;

  private static class Node
  {
    public int type; // LIST_NODE or HASH_NODE
  }

  private class ListNode extends Node
  {
    public int[] indexes;   // index in Vector of Itemsets
    public int size;        // how many indexes we keep in above array
    public boolean visited; // have we seen this node?

    public ListNode()
    {
      type = LIST_NODE;
      indexes = new int[LIST_SIZE];
      size = 0;
      visited = false;
    }


  }

  //mudancas strings
  private class HashNode extends Node
  {
    //public MyHashtable children;
    public Hashtable children;
    public HashNode()
    {
      type = HASH_NODE;
    //children = new MyHashtable(HASH_SIZE);
    children = new Hashtable(HASH_SIZE);
    }
  }

  //classe tirada de cena p/ atender a um item string
  /*private static class MyHashtable
  {
    public Node[] contents;

    public MyHashtable(int size)
    {
      contents = new Node[size];
    }

    public void put(int key, Node n)
```

```
  {
    int index = key % contents.length;
    contents[index] = n;
  }

  public Node get(int key)
  {
    int index = key % contents.length;
    return contents[index];
  }

  public Enumeration elements()
  {
    return new Enumeration()
    {
      int i = 0;
      public boolean hasMoreElements()
      {
        while (i < contents.length
             && contents[i] == null)
          i++;

        if (i >= contents.length)
          return false;
        else
          return true;
      }
      public Object nextElement()
      {
        while (i < contents.length
             && contents[i] == null)
          i++;

        if (i >= contents.length)
          throw new NoSuchElementException();
        else
          return contents[i++];
      }
    };
  }
}*/

//the fuzzy candidates created in the fuzzy analysis
private Vector fuzzy_candidates;

private int counter;     // used for some computations

private Vector leaves;   // keeps all leaves of the HashTree
private Vector itemsets; // the Vector of Itemsets that we index

private Node theRoot;    // the root of the HashTree

private void unvisitLeaves()
{
  for (int i = 0; i < leaves.size(); i++)
    ((ListNode)leaves.get(i)).visited = false;
}

/**
 * Create a new HashTree. The <code>listSize</code> parameter determines
 * after how many inserts in a ListNode we have to change it to a
```

```
   * HashNode (i.e. perform a split). The <code>hashSize</code> parameter
   * can be specified to improve the efficiency of the structure.
   *
   * @param listSize   the size of the internal lists in the list nodes
   * @param hashSize   the size of the internal hashtables in the hash
nodes
   * @param itemsets   the Vector of Itemsets that we should index
   * @exception IllegalArgumentException   <code>itemsets</code> is null
   * or <code>listSize <= 0</code> or <code>hashSize <= 0</code>
   */
  public HashTree(int listSize, int hashSize, Vector itemsets)
  {
    if (itemsets == null || listSize <= 0 || hashSize <= 0)
      throw new IllegalArgumentException("invalid arguments to
constructor");

    LIST_SIZE = listSize;
    HASH_SIZE = hashSize;
    this.itemsets = itemsets;

    theRoot = new ListNode();
    leaves = new Vector();
  }

  /**
   * Create a new HashTree. This initializes the HashTree with
   * default parameters.
   *
   * @param itemsets   the Vector of Itemsets that we should index
   * @exception IllegalArgumentException   <code>itemsets</code> is null
   */
  public HashTree(Vector itemsets)
  {
    this(DEFAULT_LIST_SIZE, DEFAULT_HASH_SIZE, itemsets);
  }

  /**
   * This method should be called before calling update() to gather
   * all leaves of the HashTree for more efficient processing.
   */
  public void prepareForDescent()
  {
    leaves.clear();
    prepare(theRoot);
  }

  // private recursive method
  private void prepare(Node node)
  {
    if (node.type == HASH_NODE)
      {
      Enumeration e = ((HashNode)node).children.elements();
      while (e.hasMoreElements())
        prepare((Node)e.nextElement());
      }
    else // LIST_NODE
      leaves.add(node);
  }

  // reset HashTree and readd all itemsets added previously
  // this method may be called by add()
```

```
private void readdAll()
{
  // increase size of hash nodes
  HASH_SIZE = 2 * HASH_SIZE + 1;

  // first save leaf nodes
  prepareForDescent();

  // then reset HashTree
  theRoot = new ListNode();

  // readd everything that we added before
  for (int i = 0; i < leaves.size(); i++)
    {
    ListNode ln = (ListNode)leaves.get(i);
    for (int j = 0; j < ln.size; j++)
      add(ln.indexes[j]);
    }

  // clean up
  leaves.clear();
}

private static class HashTreeOverflowException extends RuntimeException
{
}

/**
 * This method indexes in the HashTree the Itemset at index
 * <code>index</code> from Vector <code>itemsets</code> which
 * was passed to the constructor of this HashTree.
 *
 * @param index   the index of the Itemset that we need to index in
 * this HashTree.
 */
public void add(int index)
{
  // repeat the operation if it results in a HashTree overflow
  while (true)
    try
    {
      theRoot = add(theRoot, 0, index);

      // if we get here then the add() was successful and we can
      // exit the loop
      break;
    }
    catch (HashTreeOverflowException e)
    {
      // call readdAll to increase the size of hash nodes
      // and readd all itemsets that were previously added
      readdAll();
    }
}

// private recursive method
private Node add(Node node, int level, int index)
{
  if (node.type == LIST_NODE)
    {
    ListNode ln = (ListNode)node;
```

```
    if (ln.size == LIST_SIZE) // list is full
      {
        // if the level is equal to the itemsets size and we
        // filled the list node, then we overflowed the HashTree.
        if (((Itemset)itemsets.get(index)).size() == level)
          throw new HashTreeOverflowException();

        // else, must split!
        HashNode hn = new HashNode();

        // hash the list elements
        for (int i = 0; i < LIST_SIZE; i++)
          add(hn, level, ln.indexes[i]);

        // add our node
        add(hn, level, index);

        // return this HashNode to replace old ListNode
        return hn;
      }
    else // append index at end of list
      {
        ln.indexes[ln.size++] = index;
        //System.out.println("a");
      }
    }
  else // HASH_NODE
    {
    HashNode hn = (HashNode)node;

    // compute hash key
    Itemset is = (Itemset)itemsets.get(index);
    String key = is.getItem(level);

    // try to get next node
    Node n = (Node) hn.children.get((String)key);
    if (n == null) // no node, must create a new ListNode
      {
        ListNode ln = new ListNode();
        ln.indexes[ln.size++] = index;
        hn.children.put(key, ln);
      }
    else // found a node, do a recursive call
      {
        n = add(n, level + 1, index);
        hn.children.put(key, n);
      }
    }

  return node;
}

/**
 * Update the weights of all indexed Itemsets that are included
 * in <code>row</code>
 *
 * @param row   the Itemset (normally a database row) against which
 * we test for inclusion
 */
public void update(Itemset row)
```

```
  {
    update(theRoot, row, 0);
    unvisitLeaves();
  }

  // private recursive method
  private void update(Node node, Itemset row, int index)
  {
    if (node.type == LIST_NODE)
      {
      ListNode ln = (ListNode)node;

      if (ln.visited)
        return;

      for (int i = 0; i < ln.size; i++)
        {
          Itemset is = (Itemset)itemsets.get(ln.indexes[i]);

          if(!is.isFuzzy()){ // if not Fuzzy, increment weight normally
          if (is.isIncludedIn(row))
                  is.incrementWeight();
          }else{ // if fuzzy, set weight accordingly
            Vector derived = new Vector();
            derived = is.getItemsetsFromFuzzy();
            Itemset aux;
            float new_weight = 0;

            // for each itemset in vector, tests if it is included in row
            for(int j = 0; j < derived.size(); j++){
                  aux = (Itemset)derived.elementAt(j);
                  if (aux.isIncludedIn(row)) new_weight++;
            }

            is.setWeight(is.getWeight() + new_weight); // set fuzzy item
weight
          }
        }

      ln.visited = true; // now we've seen this node
      }
    else // HASH_NODE
      {
      HashNode hn = (HashNode)node;

      // this is a tricky piece of algorithm that ensures we
      // look for all possible subsets of row
      for (int i = index; i < row.size(); i++)
        {
          String key = row.getItem(i);

          Node n = (Node)hn.children.get(key);
          if (n != null)
            update(n, row, i + 1);

            // fuzzy section
            // checks if key belongs to any fuzzy candidate, in any domain
            // if so, update the fuzzy candidate too.

            for(int j = 0; j < fuzzy_candidates.size(); j++){
                  Itemset is = (Itemset)fuzzy_candidates.elementAt(j);
```

```
                if(is.containsKey(key)){
                      Node nf = (Node)hn.children.get(is.getFirstItem());
                      if (nf != null)
                        update(nf, row, i + 1);
                }
            }

      }
    }
}

/**
 * Update the weights of all indexed Itemsets that are included
 * in <code>row</code> and also update the matrix <code>counts</code>
 *
 * @param row   the Itemset (normally a database row) against which
 * we test for inclusion
 * @param counts   a matrix used by some algorithms to speed up
 * computations; its rows correspond to Itemsets and its columns
 * correspond to items; each value in the matrix tells for how many
 * times had an item appeared together with an itemset in the rows
 * of the database.
 */
public void update(Itemset row, long[][] counts)
{
  update(theRoot, row, 0, counts);
  unvisitLeaves();
}

// private recursive method
private void update(Node node, Itemset row, int index, long[][] counts)
{
  if (node.type == LIST_NODE)
    {
    ListNode ln = (ListNode)node;

    if (ln.visited)
      return;

    for (int i = 0; i < ln.size; i++)
      {
        Itemset is = (Itemset)itemsets.get(ln.indexes[i]);
        if (is.isIncludedIn(row))
          {
          is.incrementWeight();

    //    for (int j = 0; j < row.size(); j++)
    //      counts[ln.indexes[i]][row.getItem(j) - 1]++;
          }
      }

    ln.visited = true; // now we've seen this node
    }
  else // HASH_NODE
    {
    HashNode hn = (HashNode)node;

    // this is a tricky piece of algorithm that ensures we
    // look for all possible subsets of row
    for (int i = index; i < row.size(); i++)
      {
```

```
        String key = row.getItem(i);
        Node n = (Node) hn.children.get(key);
        if (n != null)
          update(n, row, i + 1, counts);
      }
    }
}

/**
 * Count how many frequent Itemsets (frequent = having weight
 * greater than a specified minimum weight) are included in
 * <code>itemset</code>
 *
 * @param itemset   the Itemset for which we count the subsets
 * @param minWeight   the minimum weight
 */
public long countFrequentSubsets(Itemset itemset, long minWeight)
{
  counter = 0;
  countFrequentSubsets(theRoot, itemset, 0, minWeight);
  unvisitLeaves();
  return counter;
}

// private recursive method
private void countFrequentSubsets(Node node, Itemset itemset,
                          int index, long minWeight)
{
  if (node.type == LIST_NODE)
    {
    ListNode ln = (ListNode)node;

    if (ln.visited)
      return;

    for (int i = 0; i < ln.size; i++)
      {
        Itemset is = (Itemset)itemsets.get(ln.indexes[i]);
        if (is.isIncludedIn(itemset) && is.getWeight() >= minWeight)
          counter++;
      }

    ln.visited = true; // now we've seen this node
    }
  else // HASH_NODE
    {
    HashNode hn = (HashNode)node;

    // this is a tricky piece of algorithm that ensures we
    // look for all possible subsets of row
    for (int i = index; i < itemset.size(); i++)
      {
        String key = itemset.getItem(i);
        Node n = (Node) hn.children.get(key);
        if (n != null)
          countFrequentSubsets(n, itemset, i + 1, minWeight);
      }
    }
}

/**
```

```
 * Count how many Itemsets are included in <code>itemset</code>
 *
 * @param itemset   the Itemset for which we count the subsets
 */
public long countSubsets(Itemset itemset)
{
  counter = 0;
  countSubsets(theRoot, itemset, 0);
  unvisitLeaves();
  return counter;
}

// private recursive method
private void countSubsets(Node node, Itemset itemset, int index)
{
  if (node.type == LIST_NODE)
    {
    ListNode ln = (ListNode)node;

    if (ln.visited)
      return;

    for (int i = 0; i < ln.size; i++)
      {
        Itemset is = (Itemset)itemsets.get(ln.indexes[i]);
        if (is.isIncludedIn(itemset))
          counter++;
      }

    ln.visited = true; // now we've seen this node
    }
  else // HASH_NODE
    {
    HashNode hn = (HashNode)node;

    // this is a tricky piece of algorithm that ensures we
    // look for all possible subsets of row
    for (int i = index; i < itemset.size(); i++)
      {
        String key = itemset.getItem(i);
        Node n = (Node)hn.children.get(key);
        if (n != null)
          countSubsets(n, itemset, i + 1);
      }
    }
}

/**
 * Verifies if any of the indexed Itemsets is not large by checking
 * whether they're included in the frequent itemset <code>itemset</code>.
 * If an Itemset is not large then it will be marked.
 *
 * @param itemset   the Itemset we check
 */
public void checkLargeness(Itemset itemset)
{
  checkLargeness(theRoot, itemset, 0);
  unvisitLeaves();
}

// private recursive method
```

```
  private void checkLargeness(Node node, Itemset itemset, int index)
  {
    if (node.type == LIST_NODE)
      {
      ListNode ln = (ListNode)node;

      if (ln.visited)
        return;

      for (int i = 0; i < ln.size; i++)
        {
          Itemset is = (Itemset)itemsets.get(ln.indexes[i]);
          if (is.isIncludedIn(itemset))
            is.mark();
        }

      ln.visited = true; // now we've seen this node
      }
    else // HASH_NODE
      {
      HashNode hn = (HashNode)node;

      // this is a tricky piece of algorithm that ensures we
      // look for all possible subsets of row
      for (int i = index; i < itemset.size(); i++)
        {
          String key = itemset.getItem(i);
          Node n = (Node)hn.children.get(key);
          if (n != null)
            checkLargeness(n, itemset, i + 1);
        }
      }
  }

  public void getFuzzyCandidates(Vector f){
        this.fuzzy_candidates = f;
  }
}
```

Classe Item

```
/*
 * Created on 11/12/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package aprioriminer;

import java.io.Serializable;

/**
 * @author Administrator
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
public class Item implements Serializable{
```

```
	private String item;
	private int domain;

	public Item(String item, int domain){
		this.item = item;
		this.domain = domain;
	}

	public int getDomain(){
		return domain;
	}

	public String getItem(){
		return item;
	}
}
```

Classe ItemSet

```
package aprioriminer;


import java.util.*;

import ontologyReader.OntReader;

/**

   Itemset.java<P>

   An itemset is an ordered list of integers that identify items
   coupled with a float value representing the support of the itemset
   as a percentage.<P>

*/

public class Itemset implements java.io.Serializable
{
  private static final int SIZE_INCR = 7;

  //The character used to represent fuzzy items
  private static final char FUZZY_CHAR = '~';

  /**
   * The capacity of the itemset.
   *
   * @serial
   */
  private int capacity;

  /**
   * The number of items in the itemset.
   *
   * @serial
   */
  private int size;

  /**
   * The itemset.
```

```
   *
   * @serial
   */
  private String[] set;

  /**
   * The support of the itemset.
   *
   * @serial
   */
  private float support;

  /**
   * The weight of the itemset.
   *
   * @serial
   */
  private float weight;

  /**
   * The mark of the itemset.
   *
   * @serial
   */
  private boolean mark; // this can be used to mark the itemset for
  // various purposes

  /**
   * Internal index used for cycling through the itemset's items.
   *
   * @serial
   */
  private int index;

  /**
   * Creates a new empty itemset.
   */
  public Itemset()
  {
    capacity = SIZE_INCR;
    set = new String[capacity];
    size = 0;
    support = 0;
    weight = 0;
    mark = false;
    index = 0;
  }

  /**
   * Create a new empty itemset of specified capacity.
   *
   * @param c   the capacity of the itemset
   * @exception IllegalArgumentException   <code>c</code> is negative or
zero
   */
  public Itemset(int c)
  {
    if (c < 1)
      throw new IllegalArgumentException("constructor requires positive
argument value");
```

```
    capacity = c;
    set = new String[capacity];
    size = 0;
    support = 0;
    weight = 0;
    mark = false;
    index = 0;
  }

  /**
   * Create a new itemset by copying a given one.
   *
   * @param itemset   the itemset to be copied
   * @exception IllegalArgumentException   <code>itemset</code> is null
   */
  public Itemset(Itemset itemset)
  {
    if (itemset == null)
      throw new IllegalArgumentException("constructor requires an itemset
as argument");

    capacity = itemset.capacity;
    set = new String[capacity];
    size = itemset.size;
    support = itemset.support;
    weight = itemset.weight;
    mark = itemset.mark;
    for (int i = 0; i < size; i++)
      set[i] = itemset.set[i];

    index = 0;
  }

  /**
   * Return support of itemset.
   */
  public float getSupport()
  {
    return support;
  }

  /**
   * Return weight of itemset.
   */
  public float getWeight()
  {
    return weight;
  }

  /**
   * Return i-th item in set.
   *
   * @param i   the index of the item to get
   * @exception IndexOutOfBoundsException   <code>i</code> is an invalid
index
   * @return   the <code>i</code>-th item
   */
  public String getItem(int i)
  {
    if (i < 0 || i >= size)
      throw new IndexOutOfBoundsException("invalid index");
```

```
  return set[i];
}

/**
 * Return first item in set.
 *
 * @exception IndexOutOfBoundsException   there is no first item
 * @return   first item
 */
public String getFirstItem()
{
  index = 0;

  if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException("no first item");

  return set[index++];
}

/**
 * Return next item in set.
 *
 * @exception IndexOutOfBoundsException   there is no next item
 * @return   next item
 */
public String getNextItem()
{
  if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException("no next item");

  return set[index++];
}

/**
 * Return true if there are more items in the itemset. You can call
 * this method to find out whether you can call getNext without
 * raising an exception.
 *
 * @return   true if there are more items, false if not
 */
public boolean hasMoreItems()
{
  if (index < 0 || index >= size)
    return false;
  else
    return true;
}

/**
 * Return size of itemset.
 *
 * @return   size of itemset
 */
public int size()
{
  return size;
}

/**
 * Return true if this itemset has items in common
```

```
   * with <code>itemset</code>.
   *
   * @param itemset   the itemset with which we compare
   * @exception IllegalArgumentException   <code>itemset</code> is null
   * @return   true if <code>itemset</code> contains items of this
   * itemset, false otherwise.
   */
  public boolean doesIntersect(Itemset itemset)
  {
    if (itemset == null)
      throw new IllegalArgumentException("subtract() requires an itemset as
argument");

    Itemset result = new Itemset(capacity);
    int i = 0;
    int j = 0;
    for ( ; i < size && j < itemset.size; )
      {
      // if elements are equal, return true
      //--if (set[i] == itemset.set[j])
      if(set[i].compareTo(itemset.set[j])==0)
        return true;
      // if the element in this Itemset is bigger then
      // we need to move to the next item in itemset.
      //else if (set[i] > itemset.set[j])
      else if(set[i].compareTo(itemset.set[j])< 0)
        j++;
      // the element in this Itemset does not appear
      // in itemset so we need to add it to result
      else
        i++;
      }

    return false;
  }

//  /**
//   * Return a itemset containing the items not in common
//   * with <code>itemset</code>.
//   *
//   * @param itemset   the itemset with which we compare
//   * @exception IllegalArgumentException   <code>itemset</code> is null
//   * @return   a itemset
//   */
//  public Itemset nonIntersection(Itemset itemset)
//  {
//    if (itemset == null)
//      throw new IllegalArgumentException("nonIntersection() requires an
itemset as argument");
//
//    Itemset result = new Itemset(capacity);
//    int i = 0;
//    int j = 0;
//    for ( ; i < size && j < itemset.size; )
//      {
//    // if elements are equal, return true
//    //--if (set[i] == itemset.set[j])
//    if(set[i].compareTo(itemset.set[j])==0)
//      return true;
//    // if the element in this Itemset is bigger then
//    // we need to move to the next item in itemset.
```

```
//    //else if (set[i] > itemset.set[j])
//    else if(set[i].compareTo(itemset.set[j])< 0)
//      j++;
//    // the element in this Itemset does not appear
//    // in itemset so we need to add it to result
//    else
//      i++;
//      }
//
//    return false;
//  }

  /**
   * Return a new Itemset that contains only those items that do not
   * appear in <code>itemset</code>.
   *
   * @param itemset   the itemset whose items we want to subtract
   * @exception IllegalArgumentException   <code>itemset</code> is null
   * @return   an Itemset containing only those items of this Itemset that
   * do not appear in <code>itemset</code>.
   */
  public Itemset subtract(Itemset itemset)
  {
    if (itemset == null)
      throw new IllegalArgumentException("subtract() requires an itemset as
argument");

    Itemset result = new Itemset(capacity);
    int i = 0;
    int j = 0;
    for ( ; i < size && j < itemset.size; )
      {
      // if elements are equal, move to next ones
      //if (set[i] == itemset.set[j])
      if (set[i].equals( itemset.set[j]))
        {
          i++;
          j++;
        }
      // if the element in this Itemset is bigger then
      // we need to move to the next item in itemset.
      //else if (set[i] > itemset.set[j])
      else if (set[i].compareTo(itemset.set[j])<0)
        j++;
      // the element in this Itemset does not appear
      // in itemset so we need to add it to result
      else
        result.set[result.size++] = set[i++];
      }

    // copy any remaining items from this Itemset
    while (i < size)
      result.set[result.size++] = set[i++];

    // NOTE: the size of the resulting itemset
    // has been automatically updated
    return result;
  }

  /**
   * Return a new Itemset that contains all those items that appear
```

```
   * in this Itemset and in <code>itemset</code>.
   *
   * @param itemset   the itemset whose items we want to add
   * @exception IllegalArgumentException   <code>itemset</code> is null
   * @return   an Itemset containing all those items that appear
   * in this Itemset and in <code>itemset</code>.
   */
  public Itemset add(Itemset itemset)
  {
    if (itemset == null)
      throw new IllegalArgumentException("add() requires an itemset as
argument");

    Itemset result = new Itemset(capacity);
    int i = 0;
    int j = 0;
    for ( ; i < size && j < itemset.size; )
      {
      // if elements are equal, copy then move to next ones
      //if (set[i] == itemset.set[j])
      if (set[i].equals(itemset.set[j]))
        {
          result.set[result.size++] = set[i++];
          j++;
        }
      // if the element in this Itemset is bigger then
      // we need to copy from itemset then move to the next item.
      //else if (set[i] > itemset.set[j])
      else if (set[i].compareTo(itemset.set[j])<0)
        result.set[result.size++] = itemset.set[j++];

      // else we need to copy from this Itemset
      else
        result.set[result.size++] = set[i++];

      }

    // copy any remaining items from this Itemset
    while (i < size)
      result.set[result.size++] = set[i++];

    // copy any remaining items from itemset
    while (j < itemset.size)
      result.set[result.size++] = itemset.set[j++];


    // NOTE: the size of the resulting itemset
    // has been automatically updated
    return result;
  }

  /**
   * Add a new item to the itemset.
   *
   * @param item   the item to be added
   * @exception IllegalArgumentException   <code>item</code> is <= 0
   * @return   true if item was added, false if it wasn't added (was
   * already there!)
   */
  public boolean addItem(String item)
  {
```

```
    //if (item <= 0)
    if(item.equals(""))
      throw new IllegalArgumentException("negative or zero value for item
not allowed");

    if (size == 0)
            set[0] = item;
    else
      {
      // look for place to insert item
      int index;

    //      for (index = 0; index < size && item > set[index]; index++)
      //;

    for (index = 0; index < size && (item.compareTo(set[index])<0);
index++)
      ;


      // if item was already in itemset then return
      if (index < size && item == set[index])
        return false;

      // if set is full then allocate new array
      if (size == capacity)
        {
          capacity = size + SIZE_INCR;
          String[] a = new String[capacity];
          int[] d = new int[capacity];

          int i;
          for (i = 0; i < index; i++)
                  a[i] = set[i];


          a[i] = item;
          //d[i] = item.getDomain(i);

          for ( ; i < size; i++)
              a[i + 1] = set[i];

          set = a;
        }
      // otherwise make place and insert new item
      else
        {
          int i;
          for (i = size; i > index; i--)
            set[i] = set[i - 1];

          set[i] = item;
        }
      }
    // update size
    size++;

    return true;
  }
```

```
  /**
   * Removes a given item from the itemset.
   *
   * @param item   the item to remove
   * @exception IllegalArgumentException   <code>item</code> is <= 0
   * @return   true if item was removed, false if it wasn't removed (was
   * not found in itemset!)
   */
  //public boolean removeItem(int item)
  public boolean removeItem(String item)
  {
    //if (item <= 0)
    if(item.equals(""))
      throw new IllegalArgumentException("negative or zero value for item
not allowed");
      //encontrar a posicao do item e remover

    int index = findPosition(item);
    if (index ==-1)
       return false;

   // for (index = 0; index < size && item != set[index] ; index++)
   //;

    if (item == set[index])
      {
      for (++index; index < size; index++)
        set[index - 1] = set[index];
      size--;
      return true;
      }
    else
      return false;
  }

  /**
   * Finds the correct position of the item. Needs to be improved
   * @return
   */

  public int findPosition(String item)
  {
      for(int i = 0; i< size ; i++)
         if(set[i].equals(item))
          return i;
      return -1;

  }


  /**
   * Removes last item (which has the greatest value) from the itemset.
   *
   * @return   true if item was removed, false if it wasn't removed (the
   * itemset was empty)
   */
  public boolean removeLastItem()
  {
    if (size > 0)
      {
      size--;
```

```
      return true;
      }
    else
      return false;
  }

  /**
   * Set the support of the itemset.
   *
   * @param newSupport   the support of the itemset
   * @exception IllegalArgumentException   <code>newSupport</code> is < 0
   * or > 100
   */
  public void setSupport(float newSupport)
  {
    if (newSupport < 0 || newSupport > 1)
      throw new IllegalArgumentException("support must be between 0 and
1");

    support = newSupport;
  }

  /**
   * Set the weight of the itemset.
   *
   * @param newWeight   the weight of the itemset
   * @exception IllegalArgumentException   <code>newWeight</code> is < 0
   */
  public void setWeight(float newWeight)
  {
    if (newWeight < 0)
      throw new IllegalArgumentException("weight must be positive");

    weight = newWeight;
  }

  /**
   * Increment the weight of the itemset.
   */
  public void incrementWeight()
  {
    weight++;
  }

  /**
   * Checks equality with a given itemset.
   *
   * @param itemset   the itemset against which we test for equality
   * @exception IllegalArgumentException   <code>itemset</code> is null
   */
  public boolean isEqualTo(Itemset itemset)
  {
    if (itemset == null)
      throw new IllegalArgumentException("itemset required as argument");

    if (size != itemset.size())
      return false;

    int count = 0;

    for (int i = 0; i < size; i++)
```

```
      //if (set[i] != itemset.set[i])
         for (int j = 0; j < size; j++) {
               if (set[i].compareTo(itemset.getItem(j)) == 0)
                     count++;

      }
    if (count != size)
      return false;
    return true;
  }

  /**
   * Checks inclusion in a given itemset.
   *
   * @param itemset   the itemset against which we test for inclusion
   * @exception IllegalArgumentException   <code>itemset</code> is null
   */
  public boolean isIncludedIn(Itemset itemset)
  {
    if (itemset == null)
      throw new IllegalArgumentException("itemset required as argument");

    if (itemset.size() < size)
      return false;

    int i, j;

   /* for (i = 0, j = 0;
        i < size && j < itemset.size() && set[i] >= itemset.set[j];
        j++)
      if (set[i] == itemset.set[j])
      i++;*/

      for (i = 0, j = 0;
             i < size && j < itemset.size() &&
(set[i].compareTo(itemset.set[j])<=0);
             j++)
              if (set[i].equals(itemset.set[j]))
            i++;

    if (i == size)
      return true;
    else
      return false;
  }

  /**
   * Mark the itemset.
   *
   * @return   true if itemset was already marked, false otherwise
   */
  public boolean mark()
  {
    boolean old_mark = mark;
    mark = true;
    return old_mark;
  }

  /**
   * Unmark the itemset.
   *
```

```
 * @return   true if itemset was marked, false otherwise
 */
public boolean unmark()
{
  boolean old_mark = mark;
  mark = false;
  return old_mark;
}

/**
 * Return itemset mark.
 *
 * @return   true if itemset is marked, false otherwise
 */
public boolean isMarked()
{
  return mark;
}

/**
 * Return a String representation of the Itemset.
 *
 * @return   String representation of Itemset
 */
public String toString()
{
  String s = "{";

  for (int i = 0; i < size; i++)
    s += set[i]+ " ";
  s += "}/[" + support + "/" + weight + "] (" + size + ")";

  return s;
}

/**
 * Check whether two itemsets can be combined. Two itemsets can be
 * combined if they differ only in the last item.
 *
 * @param itemset   itemset with which to combine
 * @exception IllegalArgumentException   <code>itemset</code> is null
 * @return   true if the itemsets can be combined, false otherwise
 */
public boolean canCombineWith(Itemset itemset)
{
  if (itemset == null)
    throw new IllegalArgumentException("itemset required as argument");

  if (size != itemset.size)
    return false;

  if (size == 0)
    return false;

  for (int i = 0; i < size - 1; i++)
    //if (set[i] != itemset.set[i])
       if (!set[i].equals(itemset.set[i]))
           return false;

  return true;
}
```

```
/**
 * Combine two itemsets into a new one that will contain all the
 * items in the first itemset plus the last item in the second
 * itemset.
 *
 * @param itemset   itemset with which to combine
 * @exception IllegalArgumentException   <code>itemset</code> is null
 * @return   an itemset that combines the two itemsets as described
 * above
 */
public Itemset combineWith(Itemset itemset)
{
  if (itemset == null)
    throw new IllegalArgumentException("itemset required as argument");

  Itemset is = new Itemset(this);

  is.support = 0;
  is.weight = 0;

  is.addItem(itemset.set[itemset.size - 1]);

  return is;
}

/**
 * Remove all non-maximal itemsets from the vector v
 *
 * @param v   the collection of itemsets
 */
public static synchronized void pruneNonMaximal(Vector v)
{
  int i, j;
  int size = v.size();

  for (i = 0; i < size; i++)
    {
    // see if anything is included in itemset at index i
    for (j = i + 1; j < size; j++)
      if (((Itemset)v.get(j)).isIncludedIn((Itemset)v.get(i)))
          {
          // replace this element with last, delete last,
          // and don't advance index
          v.set(j, v.lastElement());
          v.remove(--size);
          j--;
          }

    // see if itemset at index i is included in another itemset
    for (j = i + 1; j < size; j++)
      if (((Itemset)v.get(i)).isIncludedIn((Itemset)v.get(j)))
          {
          // replace this element with last, delete last,
          // and don't advance index
          v.set(i, v.lastElement());
          v.remove(--size);
          i--;
          break;
          }
    }
```

```
}

/**
 * Remove all duplicate itemsets from the vector v
 *
 * @param v   the collection of itemsets
 */
public static synchronized void pruneDuplicates(Vector v)
{
  int i, j;
  int size = v.size();

  for (i = 0; i < size; i++)
    {
    // see if anything is equal to itemset at index i
    for (j = i + 1; j < size; j++)
      if (((Itemset)v.get(j)).isEqualTo((Itemset)v.get(i)))
          {
          // replace this element with last, delete last,
          // and don't advance index
          v.set(j, v.lastElement());
          v.remove(--size);
          j--;
          }
    }
}

/**
 * for testing purposes only !!!
 */
public static void main(String[] args)
{
  Itemset is1 = new Itemset();
  Itemset is2 = new Itemset();

  is1.addItem("id7");
  is1.addItem("id3");
  is1.addItem("id15");
  is1.addItem("id5");
  is1.addItem("id12");
  is1.addItem("id12");

  System.out.println("is1: " + is1);

  is2.addItem("id12");
  is2.addItem("id15");
  is2.addItem("id7");
  is2.addItem("id5");
  is2.addItem("id3");
  is2.addItem("id8");

  System.out.println("is2: " + is2);

  System.out.println("do is1 and is2 share items: "
                + is1.doesIntersect(is2));
  System.out.println("do is2 and is1 share items: "
                + is2.doesIntersect(is1));

  Itemset is3 = is1.subtract(is2);
  System.out.println("is3 <= subtracting is2 from is1:" + is3);
```

```
System.out.println("do is1 and is3 share items: "
               + is1.doesIntersect(is3));
System.out.println("do is3 and is1 share items: "
               + is3.doesIntersect(is1));

is3 = is2.subtract(is1);
System.out.println("is3 <= subtracting is1 from is2:" + is3);

System.out.println("do is1 and is3 share items: "
               + is1.doesIntersect(is3));
System.out.println("do is3 and is1 share items: "
               + is3.doesIntersect(is1));

System.out.println("do is3 and is2 share items: "
               + is3.doesIntersect(is2));
System.out.println("do is2 and is3 share items: "
               + is2.doesIntersect(is3));

System.out.println("adding is2 to is1:" + is1.add(is2));
System.out.println("adding is1 to is2:" + is2.add(is1));

System.out.println("is1 equal to is2: " + is1.isEqualTo(is2));
System.out.println("is1 included in is2: " + is1.isIncludedIn(is2));
System.out.println("is2 included in is1: " + is2.isIncludedIn(is1));

is1.addItem("id8");

System.out.println("is1: " + is1);

System.out.println("is1 equal to is2: " + is1.isEqualTo(is2));
System.out.println("is1 included in is2: " + is1.isIncludedIn(is2));
System.out.println("is2 included in is1: " + is2.isIncludedIn(is1));

is1.addItem("id1");

System.out.println("is1: " + is1);

System.out.println("is1 equal to is2: " + is1.isEqualTo(is2));
System.out.println("is1 included in is2: " + is1.isIncludedIn(is2));
System.out.println("is2 included in is1: " + is2.isIncludedIn(is1));

is1.addItem("id50");

System.out.println("is1: " + is1);

System.out.println("is1 equal to is2: " + is1.isEqualTo(is2));
System.out.println("is1 included in is2: " + is1.isIncludedIn(is2));
System.out.println("is2 included in is1: " + is2.isIncludedIn(is1));

is1.addItem("id100");

System.out.println("is1: " + is1);

System.out.println("is1 equal to is2: " + is1.isEqualTo(is2));
System.out.println("is1 included in is2: " + is1.isIncludedIn(is2));
System.out.println("is2 included in is1: " + is2.isIncludedIn(is1));

System.out.println("adding 70 to is2: " + is2.addItem("id70"));
System.out.println("adding 70 to is2: " + is2.addItem("id70"));

System.out.println("is2: " + is2);
```

```
  System.out.println("is1 equal to is2: " + is1.isEqualTo(is2));
  System.out.println("is1 included in is2: " + is1.isIncludedIn(is2));
  System.out.println("is2 included in is1: " + is2.isIncludedIn(is1));

  System.out.println("removing 1 from is1: " + is1.removeItem("id1"));
  System.out.println("removing 1 from is1: " + is1.removeItem("id1"));
  System.out.println("is1: " + is1);
  System.out.println("removing 50 from is1: " + is1.removeItem("id50"));
  System.out.println("is1: " + is1);
  System.out.println("removing 70 from is1: " + is2.removeItem("id70"));
  System.out.println("is2: " + is2);

  System.out.print("going through items of is1:");
  while (is1.hasMoreItems())
    System.out.print(" " + is1.getNextItem());
  System.out.println("");

  System.out.print("going through items of is2:");
  while (is2.hasMoreItems())
    System.out.print(" " + is2.getNextItem());
  System.out.println("");

  System.out.println("is1 first item: " + is1.getFirstItem());
  System.out.println("is1 next item: " + is1.getNextItem());
  System.out.println("is2 first item: " + is2.getFirstItem());
  System.out.println("is2 next item: " + is2.getNextItem());

  while (is2.removeLastItem())
    ;
  System.out.println("is2: " + is2);

  System.out.println("mark is1, previous state: " + is1.mark());
  System.out.println("mark is1, previous state: " + is1.mark());
  System.out.println("is1 mark state: " + is1.isMarked());
  System.out.println("unmark is1, previous state: " + is1.unmark());
  System.out.println("unmark is1, previous state: " + is1.unmark());

}


/**
 * If reaches at least one fuzzy item, the itemset is fuzzy.
 *
 * @return   true if fuzzy, false otherwise.
 */
public boolean isFuzzy()
{
    boolean result = false;
    for(int i = 0; i < this.size; i++){
          if(itemIsFuzzy(set[i])) result = true;
    }
    return result;
}

/**
 * If reaches at least one FUZZY_CHAR, the item is fuzzy.
 *
 * @return   true if fuzzy, false otherwise.
 */
private boolean itemIsFuzzy(String item)
```

```
  {
      boolean result = false;
      for(int j = 0; j < item.length(); j++){
            if(item.charAt(j) == FUZZY_CHAR) result = true;
      }
      return result;
  }


  /**
      * Returns a 2-array of strings.
      * The first one is the head of the fuzzy item.
      * The last one is the tail of the fuzzy item.
      *
      * @return   a 2-array of Strings
      */
      private String[] getHeadAndTail(String item){
            String [] is = new String[2];

            for(int j = 0; j < item.length(); j++){
                  if(item.charAt(j) == FUZZY_CHAR){
                        is[0] = item.substring(0,j); // the head
                        is[1] = item.substring(j+1,item.length()); // the
tail
                        break;
                  }
            }

            return is;
      }

   /**
      * Return, item by item, the number of items that can be generated
from this
      * fuzzy itemset.
      * (counter[i] is related to set[i])
      *
      * @return   counter (int[])
      */
      public int[] getNumberOfChildrenPerItem(){
            int counter[] = new int[this.size];

            for(int k = 0; k < this.size; k++) counter[k] = 0; //
initialization of counter[]

            for(int i = 0; i < this.size; i++){
                  if(!itemIsFuzzy(set[i])) counter[i]++;
                  else{
                        for(int j = 0; j < set[i].length(); j++){
                              if(set[i].charAt(j) == FUZZY_CHAR)
counter[i]++;
                        }
                        counter[i]++; // number of FUZZY_CHARs plus one
                  }
            }

            return counter;
      }


  /**
```

```
   * Gets a Vector of Itemsets taken from a fuzzy itemset.
   *
   * @return   a Vector of itemsets
   */
  public Vector getItemsetsFromFuzzy()
  {
      Vector fuzzy_itemsets = new Vector();
      Vector fuzzy_items[] = new Vector[this.size];
      String items[] = new String[2];

      int[] number_of_children = new int[this.size]; //number of children
each item can generate
      number_of_children = this.getNumberOfChildrenPerItem();
      int children = 1;

      for(int q = 0; q < this.size; q++){
            children = children * number_of_children[q]; // computes how
many combinations will be generated
      }

      // Stores in fuzzy_items[i] the items obtained from set[i]
      for(int i = 0; i < this.size; i++){ // for each set[i]

            String item = set[i];
            fuzzy_items[i] = new Vector();

            if(number_of_children[i] == 1){ // if item is not fuzzy

                  // add item to Vector fuzzy_items
                  fuzzy_items[i].add(item);

            }else{ // item is fuzzy

                  do{
                  items = getHeadAndTail(item); // get head and tail

                  fuzzy_items[i].add(items[0]); // add head to Vector
fuzzy_items
                  item = items[1];              // tail becomes item (loop)

                  }while(itemIsFuzzy(items[1])); // while tail remains
fuzzy

                  fuzzy_items[i].add(items[1]); // now tail is not fuzzy,
so we can add it to fuzzy_items
            }
      }

      // combination of items got from fuzzy_items
      Vector x = new Vector();
      Itemset is;

      // first, itemsets are created from fuzzy_items[0], and stored in
Vector x
      for(int k = 0; k < fuzzy_items[0].size(); k++){
            is = new Itemset(1);
            is.addItem((String)fuzzy_items[0].elementAt(k));
            x.add(is);
      }

      // now we see the other fuzzy_items[i], i > 0
```

```
      for(int i = 1; i < this.size; i++){

            if(fuzzy_items[i].size() == 1){ // if size == 1, item is added
to all former itemsets

                  // runs through the vector, for each former itemset
                  for(int n = 0; n < x.size(); n++){
                        Itemset aux = (Itemset)x.elementAt(n);
                        aux.addItem((String)fuzzy_items[i].firstElement());
                        x.remove(n);
                        x.add(n,aux);
                  }

            }else{ // is size > 1, multiply "size times" former itemsets

                  int xsize = x.size(); // to avoid infinite loop, because
the x.size() is dynamic
                  for(int n = 0; n < xsize; n++){
                        Itemset aux = (Itemset)x.elementAt(n);
                        Itemset base = new Itemset(aux);
                        Itemset temp;
                        aux.addItem((String)fuzzy_items[i].firstElement());

                        for(int q = 1; q < fuzzy_items[i].size(); q++){
                              temp = new Itemset(base);

      temp.addItem((String)fuzzy_items[i].elementAt(q));

                              x.add(temp);
                        }
                  }

//                for(int n = 0; n < x.size(); n++){
//                      Itemset aux = (Itemset)x.elementAt(n);
//                      Itemset temp = new Itemset(aux);
//                      aux.addItem((String)fuzzy_items[i].firstElement());

//                      for(int q = 1; q < fuzzy_items[i].size(); q++){
//
      temp.addItem((String)fuzzy_items[i].elementAt(q));

//                            x.add(temp);
//                      }
//                }
            }
      }

      fuzzy_itemsets = x;

      return fuzzy_itemsets;
  }

  // checks if a fuzzy itemset contains a key
  public boolean containsKey(String key){
      boolean result = false;
      String[] items = new String[2];

      String item = this.getFirstItem();
      items = getHeadAndTail(item); // get head and tail
```

```
      do{
            items = getHeadAndTail(item); // get head and tail
            if(items[0].compareTo(key) == 0) result = true; // compare head
and key

            item = items[1];              // tail becomes item (loop)

      }while(itemIsFuzzy(items[1])); // while tail remains fuzzy

      if(items[1].compareTo(key) == 0) result = true; // now tail is not
fuzzy, so we can compare it to key

      return result;
  }

  // Transforms a ordinary itemset into a fuzzy itemset (fuzzify)
  public String fuzzify(){

      String result = set[0];

      for(int i = 1; i < this.size; i++)
            result = result + FUZZY_CHAR + set [i];

      return result;
  }

public boolean isSubItemset(Itemset itemset,Itemset is, String string,
Hashtable domains,String[] DN) {
      // TODO Auto-generated method stub
      int count = 0;
      for (int i = 0; i < itemset.size; i++) {
            boolean bool = true;
            String s = "";
            for (int j = 0; j < DN.length && bool; j++) {
                  if (DN[j].compareTo(itemset.getItem(i)) == 0)
                        bool = false;
            }
            if (bool)
                  s =
DN[((Integer)(domains.get(itemset.getItem(i)))).intValue()];
            else
                  s = string;
            for (int j = 0; j < is.size; j++) {
                  if ((s.compareTo(is.getItem(j)) == 0 &&
s.compareTo(string) == 0) || itemset.getItem(i).compareTo(is.getItem(j)) ==
0){
                        count++;
                  }
            }
      }
      if (count == itemset.size)
            return true;
      return false;
}

public boolean itemsSameDomain(Itemset itemsetAux, String[] dn, Hashtable
domains) {
      // TODO Auto-generated method stub
      for (int i = 0; i < dn.length; i++) {
            int count = 0;
            for (int j = 0; j < itemsetAux.size; j++) {
```

```
                  String s =
dn[((Integer)(domains.get(itemsetAux.getItem(j)))).intValue()];
                  if (s.compareTo(dn[i]) == 0)
                        count++;
            }
            if (count == itemsetAux.size)
                  return true;
      }
      return false;
}
}
```

Classe LargeItemSetsFinder

```
package aprioriminer;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author unascribed
 * @version 1.0
 */

/*

ARMiner - Association Rules Miner
Copyright (C) 2000  UMass/Boston - Computer Science Department


This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
USA


The ARMiner Server was written by Dana Cristofor and Laurentiu
Cristofor.

The ARMiner Client was written by Abdelmajid Karatihy, Xiaoyong Kuang,
and Lung-Tsung Li.


The ARMiner package is currently maintained by Laurentiu Cristofor
(laur@cs.umb.edu).

*/
```

```
/**

   LargeItemsetsFinder.java<P>

   This interface must be implemented by the algorithms that will look
   for large itemsets.

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public interface LargeItemsetsFinder
{
  /**
   * Find the frequent itemsets in a database
   *
   * @param dbReader   the object used to read from the database
   * @param cacheWriter   the object used to write to the cache
   * if this is null, then nothing will be saved, this is useful
   * for benchmarking
   * @param minSupport   the minimum support
   * @return   the number of passes executed over the database
   */
  int findLargeItemsets(DBReader dbReader,
                  DBCacheWriter cacheWriter,
                  float minSupport);
}
```

Classe MainFile

```
package aprioriminer;

import ontologyReader.*;

/**
 * <p>Title: </p>
 * <p>It runs Apriori and finds the execution time of it</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author unascribed
 * @version 1.0
 */

import java.io.IOException;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;

import javax.swing.text.StyledEditorKit.BoldAction;
```

```
public class MainFile {
	public static long s1; //time counter used in program

	public MainFile(String args[]) {
		//variables
		float minsup = 0, minconf = 0, minsim = 0, mingen = 0;
		Date d;
		DBReader reader = null;//, similarities = null; // similarities
added
		DBCacheWriter cacheWriter = null;
		Apriori pri = null;
		String ontologyURL = null, ontologyPath = null; // ontology
information

		//start timer
		d = new Date();
		s1 = d.getTime();

		//initializing input paramenters
		try {
			reader = new DBReader(args[0]);
			cacheWriter = new DBCacheWriter("second.cache");
			minsup = Float.parseFloat(args[1]);
			minconf = Float.parseFloat(args[2]);
			minsim = Float.parseFloat(args[3]);
			mingen = Float.parseFloat(args[4]);
			//similarities = new DBReader(args[4]);
			ontologyURL = args[5];
			ontologyPath = args[6];
		} catch (Exception e) {
			System.out.println(
				"Wrong input parameters: filename floatminsup
floatminconf floatminsim ontologyURL ontologyPath");
		}

		try {
			pri = new Apriori();
			//pri.findLargeItemsets(reader, cacheWriter, minsup);
			//pri.findLargeItemsets(reader, cacheWriter, minsup,
minsim, similarities);
			pri.findLargeItemsets(reader, cacheWriter, minsup,
minsim, ontologyURL, ontologyPath);

			// Open ontology reader
			OntReader ontReader = new OntReader(ontologyURL,
ontologyPath);

			d = new Date();
			long s2 = d.getTime();
			System.out.println(
				"Time used in finding frequent itemsets: "
					+ (s2 - s1)
					+ " miliseconds");

			DBCacheReader cacheReader = new
DBCacheReader("second.cache");
			Vector large = pri.getLargeItemsets();//vector of large
itemsets
			Vector nonLarge = pri.getNonLargeItemsets();// vector of
non large Itemsets
			Vector items = new Vector();
```

```
                  Vector exceptItems = new Vector();
                  AprioriRules pirules = new AprioriRules();
                  Vector rules = pirules.findAssociations(cacheReader,
minsup, minconf);
                  System.out.println("\nRules generated");
                  String rule;
                  ///Vector rules = new Vector();
                  Vector redundantRules = new Vector(); // pegar regras
redundantes após a realização da generalização
                  Vector nonRedundantRules = new Vector(); // pegar regras
não redundantes
                  Vector nonFuzzyRedundantRules = new Vector();// rules
without fuzzy redundancy
                  Vector subItemFuzzy = new Vector(); // subitems fuzzy
                  //modified by Rafael

                  for (int i = 0; i < large.size(); i++) {
                        Itemset is = (Itemset) large.get(i);
                        if ( !is.isFuzzy())//is.size() == 1 &&
                              items.add(is);
                  }
                  for (int i = 0; i < nonLarge.size(); i++) {
                        Itemset is = (Itemset) nonLarge.get(i);
                        if (is.size() == 1 && !is.isFuzzy())
                              items.add(is);
                  }

                  String []DN;
                  Hashtable domains = reader.getDomains();

                  DN = reader.getDomainNames();
                  Vector auxRedundant = new Vector();

                  // see the generalized rules obteined by the infrequent
treatment
                  for (int i = 0; i < rules.size(); i++) {
                        for (int j = 0; j < DN.length; j++) {
                              AssociationRule assoc = (AssociationRule)
rules.get(i);
                              if (isGeneralized(assoc,DN[j]))
                                    auxRedundant.add(assoc);
                        }
                  }
                  // to eliminate fuzzy redundancy through non-frequents
generalized itemsets
                  for (int i = 0; i < auxRedundant.size(); i++) {
                        boolean subFuzzyRule = false;
                        boolean fuzzyRule = false;
                        AssociationRule assoc = (AssociationRule)
auxRedundant.get(i);
                        //if (isFuzzyRule(assoc)){
                              for (int j = 0; j < auxRedundant.size() &&
subFuzzyRule == false; j++) {
                                    AssociationRule assocJ =
(AssociationRule) auxRedundant.get(j);
                                    if (i != j)
                                          subFuzzyRule =
isSubFuzzyRule(assoc,assocJ);

                              }
                              if (!subFuzzyRule)
```

```
                                    nonRedundantRules.add(assoc);
                              else{
                                    //String s =
getSubItem(assoc,auxRedundant,i,DN,domains);
                                    subItemFuzzy.add(assoc);

                              }
                              rules.remove(assoc);
                        //}
                  }


                  //for each domain
                  // the generalization is done when possible
                  for (int i = 0; i < DN.length; i++){

                        int p = ontReader.getClassSize(DN[i]);

                        boolean bool = true;
                        for (int j = 0; j < rules.size(); j++) {
                              int count = 0;
                              String []domainsItems = new String[p];
                              for (int k = 0; k < domainsItems.length; k++)
{
                                    domainsItems[k] = "";
                              }
                              AssociationRule assoc = (AssociationRule)
rules.get(j);
                              //generalize antecedent
                              count++;
                              float sup = assoc.getSupport();
                              float conf = assoc.getConfidence();
                              float antecedentSupport =
getAntecedentSupport(assoc,items);
                              for (int k = 0; k < assoc.antecedentSize() ;
k++) {
                                    String classname =
getDomain(DN,assoc.getAntecedentItem(k),domains);


                                    if ((classname.compareTo(DN[i])) == 0){
                                          domainsItems[count - 1] =
assoc.getAntecedentItem(k);
                                          for (int index = j + 1; index <
rules.size() && count < p; index++) {
                                                AssociationRule assoc2 =
(AssociationRule) rules.get(index);

                                                if ((assoc.antecedentSize()
== assoc2.antecedentSize()) && (assoc.consequentSize() ==
assoc2.consequentSize())
                                                            &&
(verifyAntecedentConsequent(assoc,assoc2,DN[i],DN,domains))){
                                                      count++;
                                                      sup = sup +
assoc2.getSupport();
                                                      //conf = conf +
assoc2.getConfidence();
                                                      antecedentSupport =
antecedentSupport + getAntecedentSupport(assoc2,items);
```

```
                                                    for (int x = 0; x <
assoc2.antecedentSize(); x++) {//insert on array items of a domain
                                                          String
classname2 = getDomain(DN,assoc2.getAntecedentItem(x),domains);
                                                          if
(classname2.compareTo(classname) == 0){
                                                                String []s
= assoc2.getAntecedentItem(x).split("~");
                                                                boolean b
= true;
                                                                for (int y
= 0; y < s.length; y++) {
                                                                      for
(int z = 0; z < domainsItems.length && b; z++) {
      if (s[y].compareTo(domainsItems[z]) == 0)
            b = false;
      else if (domainsItems[z].compareTo("") == 0){
            domainsItems[z] = s[y];
            b = false;
      }

                                                                }
                                                          }
                                                    }
                                              }
                                        }
                                  }
                            }
                            if (count >= p*mingen){

                                  String ant = DN[i];
                                  String cons =
assoc.getConsequentItem(0);
                                  Itemset antItemset = new Itemset();
                                  antItemset.addItem(DN[i]);
                                  for (int k = 0; k <
assoc.antecedentSize(); k++) {
                                        if
(getDomain(DN,assoc.getAntecedentItem(k),domains).compareTo(DN[i]) != 0){
                                              ant = ant + "," +
assoc.getAntecedentItem(k);
      antItemset.addItem(assoc.getAntecedentItem(k));
                                        }
                                  }
                                  for (int k = 1; k <
assoc.consequentSize(); k++) {
                                        cons = cons + "," +
assoc.getConsequentItem(k);
                                  }


      antItemset.setSupport(antecedentSupport);
```

```
                                    conf = sup / antItemset.getSupport();
                                    items.add(antItemset);
                                    AssociationRule assocAux = new
AssociationRule(ant,cons,sup,conf);
                                    boolean verify = false;
                                    for (int k = 0; k < rules.size() &&
!verify; k++) {
                                          AssociationRule assoK =
(AssociationRule) rules.get(k);
                                          if (assocAux.equals(assoK)&& k!=j)
                                                verify = true;

                                    }
                                    if (!verify){
                                          rules.insertElementAt(new
AssociationRule(ant,cons,sup,conf),j);
                                          rules.remove(j+1);
                                          Vector exception = new Vector();
                                          for (int k = 0; k < items.size();
k++) {
                                                Itemset is = (Itemset)
items.get(k);
                                                String s =
is.getFirstItem();
                                                //String classname =
DN[((Integer)(domains.get(s))).intValue()];
                                                String classname =
getDomain(DN,s,domains);
                                                if
(classname.compareTo(DN[i]) == 0){
                                                      boolean b = true;
                                                      for (int index = 0;
index < domainsItems.length && b; index++) {
                                                            if
(s.compareTo(domainsItems[index]) == 0)
                                                                  b = false;
                                                            else if
(domainsItems[index].compareTo("") == 0 && s.compareTo(DN[i]) != 0)

      exception.add(s);
                                                      }
                                                }
                                          }
                                          exceptItems.add(new
GeneralizedExceptFor(new AssociationRule(ant,cons,sup,conf),exception));
                                    }
                              }
                              count = 0;//generalize consequent
                              domainsItems = new String[p];
                              for (int k = 0; k < domainsItems.length; k++)
{
                                    domainsItems[k] = "";
                              }
                              if (count == 0)
                                    count ++;
                              for (int k = 0; k < assoc.consequentSize() ;
k++) {
                                    String classname =
getDomain(DN,assoc.getConsequentItem(k),domains);

                                    if ((classname.compareTo(DN[i])) == 0){
```

```
                                        domainsItems[count - 1] =
assoc.getConsequentItem(k);
                                        for (int index = j + 1; index <
rules.size() && count < p; index++) {
                                              AssociationRule assoc2 =
(AssociationRule) rules.get(index);

                                        if ((assoc.antecedentSize()
== assoc2.antecedentSize()) && (assoc.consequentSize() ==
assoc2.consequentSize())
                                                                    &&
(verifyConsequentAntecedent(assoc,assoc2,DN[i],DN,domains))){
                                                          count++;
                                                          sup = sup +
assoc2.getSupport();
                                                          conf = conf +
assoc2.getConfidence();
                                                          for (int x = 0; x <
assoc2.consequentSize(); x++) {//insert on array items of a domain
                                                                    String
classname2 = getDomain(DN,assoc2.getConsequentItem(x),domains);
                                                          if
(classname2.compareTo(classname) == 0){
                                                                    String []s
= assoc2.getConsequentItem(x).split("~");
                                                                    boolean b
= true;
                                                                    for (int y
= 0; y < s.length; y++) {
                                                                          for
(int z = 0; z < domainsItems.length && b; z++) {
      if (s[y].compareTo(domainsItems[z]) == 0)
            b = false;
      else if (domainsItems[z].compareTo("") == 0){
            domainsItems[z] = s[y];
            b = false;
      }
                                                                    }
                                                          }
                                                    }
                                              }
                                        }
                                  }
                            }
                            if (count >= p*mingen){
                                  count = 0;
                                  Itemset antItemset = new Itemset();

      antItemset.addItem(assoc.getAntecedentItem(0));
                                  String cons = DN[i];
                                  String ant = assoc.getAntecedentItem(0);
                                  for (int k = 0; k <
assoc.consequentSize(); k++) {
```

```
                                        if
(getDomain(DN,assoc.getConsequentItem(k),domains).compareTo(DN[i]) != 0)
                                              cons = cons + "," +
assoc.getConsequentItem(k);
                                  }
                                  for (int k = 1; k <
assoc.antecedentSize(); k++) {
                                        ant = ant + "," +
assoc.getAntecedentItem(k);

      antItemset.addItem(assoc.getAntecedentItem(k));
                                  }
                                  //redundantRules.add(new
AssociationRule(ant,cons,sup,conf));
                                  antItemset.getSupport();
                                  //fazer comparacao desse itemset com
items.
                                  boolean control = false;

                                  for (int k = 0; k < items.size() &&
!control; k++) {
                                        Itemset is = (Itemset)
items.get(k);
                                        if (is.isEqualTo(antItemset)){
                                              control = true;

      antItemset.setSupport(is.getSupport());
                                        }

                                  }
                                  for (int k = 0; k < large.size() &&
!control; k++) {
                                        Itemset is = (Itemset)
large.get(k);
                                        if (is.isEqualTo(antItemset)){
                                              control = true;

      antItemset.setSupport(is.getSupport());
                                        }
                                  }
                                  for (int k = 0; k < nonLarge.size() &&
!control; k++) {
                                        Itemset is = (Itemset)
nonLarge.get(k);
                                        if (is.isEqualTo(antItemset)){
                                              control = true;

      antItemset.setSupport(is.getSupport());
                                        }
                                  }
                                  conf = sup / antItemset.getSupport();
                                  AssociationRule assocAux = new
AssociationRule(ant,cons,sup,conf);
                                  boolean verify = false;
                                  for (int k = 0; k < rules.size() &&
!verify; k++) {
                                        AssociationRule assoK =
(AssociationRule) rules.get(k);
                                        if (assocAux.equals(assoK)&& k!=j
)
                                              verify = true;
```

```
                                    }
                                    if (!verify){
                                          rules.insertElementAt(new
AssociationRule(ant,cons,sup,conf),j);
                                          rules.remove(j+1);

                                          Vector exception = new Vector();
                                          for (int k = 0; k < items.size();
k++) {
                                                Itemset is = (Itemset)
items.get(k);
                                                String s =
is.getFirstItem();
                                                //String classname =
DN[((Integer)(domains.get(s))).intValue()];
                                                String classname =
getDomain(DN, s, domains);
                                                if
(classname.compareTo(DN[i]) == 0){
                                                      boolean b = true;
                                                      for (int index = 0;
index < domainsItems.length && b; index++) {
                                                            if
(s.compareTo(domainsItems[index]) == 0)
                                                                  b = false;
                                                            else if
(domainsItems[index].compareTo("") == 0 && s.compareTo(DN[i]) != 0)

      exception.add(s);
                                                      }
                                                }
                                          }
                                          exceptItems.add(new
GeneralizedExceptFor(new AssociationRule(ant,cons,sup,conf),exception));
                                    }
                              }
                              //}

                        }
                  }

                  for (int i = 0; i < rules.size(); i++) {
                        AssociationRule assoc = (AssociationRule)
rules.get(i);
                        boolean bool = true;

                        for (int j = 0 ; j < rules.size() && bool; j++) {
                              AssociationRule assocRed = (AssociationRule)
rules.get(j);
                              if ((i != j) && isSubRule(assoc,assocRed,DN,
domains))
                                    bool = false;
                        }
                        if (bool)
                              redundantRules.add(assoc);
                  }


                  //tratamento de redundância
                  // verificar se tratamento de redundancia está correto
```

```
                  for (int i = 0; i < nonRedundantRules.size(); i++) {
                        boolean subRule = false;
                        AssociationRule assoc = (AssociationRule)
nonRedundantRules.get(i);
                        for (int j = 0; j < redundantRules.size() &&
!subRule; j++) {
                              AssociationRule assocJ = (AssociationRule)
redundantRules.get(j);
                              if (isSubRule(assoc,assocJ,DN,domains))
                                    subRule = true;
                              if (equalRules(assoc,assocJ) && !subRule){
                                    redundantRules.add(j,assoc);
                                    redundantRules.remove(j+1);
                                    subRule = true;
                              }
                        }
                        if (!subRule)
                              redundantRules.add(assoc);
                  }


                  //eliminar redundancias fuzzy

                  for (int i = 0; i < redundantRules.size(); i++) {
                        boolean subFuzzyRule = false;
                        boolean fuzzyRule = false;
                        AssociationRule assoc = (AssociationRule)
redundantRules.get(i);
                        for (int j = 0; j < redundantRules.size() &&
subFuzzyRule == false; j++) {
                              AssociationRule assocJ = (AssociationRule)
redundantRules.get(j);
                              if (i!=j){
                                    //fuzzyRule = isFuzzyRule(assocJ);

                                          subFuzzyRule =
isSubFuzzyRule(assoc,assocJ);
                              }
                        }
                        if (!subFuzzyRule){

      if(eliminateEqualRules(assoc,nonFuzzyRedundantRules))
                              nonFuzzyRedundantRules.add(assoc);
                        }
                        else{
                              //nonFuzzyRedundantRules.remove(assoc);
                              //String s =
getSubItem(assoc,redundantRules,i,DN,domains);
                              boolean control = true;
                              for (int j = 0; j < subItemFuzzy.size() &&
control; j++) {
                                    AssociationRule assocJ =
(AssociationRule) subItemFuzzy.get(j);
                                    if (equalRules(assoc,assocJ))
                                          control = false;
                              }
                              if (control)
                                    subItemFuzzy.add(assoc);
                        }
                  }
```

```
                  //eliminating subRules of exceptionItems

                  for (int i = 0; i < exceptItems.size(); i++) {
                        GeneralizedExceptFor gen = (GeneralizedExceptFor)
exceptItems.get(i);
                        AssociationRule assoc = gen.getAssoc();
                        for (int j = 0; j < exceptItems.size(); j++) {
                              if(i != j){
                                    GeneralizedExceptFor genJ =
(GeneralizedExceptFor) exceptItems.get(j);
                                    AssociationRule assocJ =
genJ.getAssoc();
                                    if (isSubRule(assoc,assocJ,DN,domains)){
                                          if    (genJ.getExcept().size() >
0){
                                                for (int k = 0; k <
gen.getExcept().size(); k++) {
                                                      String sK = (String)
gen.getExcept().get(k);
                                                      boolean bool = false;
                                                      for (int index = 0;
index < genJ.getExcept().size() && !bool; index++) {
                                                            String sIndex =
(String) gen.getExcept().get(index);
                                                            if
(sIndex.compareTo(sK) == 0)
                                                                  bool =
true;
                                                      }
                                                      if (!bool)

      genJ.getExcept().add(gen.getExcept().get(k));
                                                }
                                          }else
                                                for (int k = 0; k <
gen.getExcept().size(); k++) {

      genJ.getExcept().add(gen.getExcept().get(k));
                                                }
                                    }

                              }
                        }
                  }

                  int countVector = 0;
                  for (int i = 0; i < nonFuzzyRedundantRules.size(); i++) {

                        rule = "";
                        AssociationRule assoc = (AssociationRule)
nonFuzzyRedundantRules.get(i);
                        boolean generalized = false;
                        for (int j = 0; j < assoc.antecedentSize(); j++) {
                              rule = rule + assoc.getAntecedentItem(j) +
",";
                        }
                        rule = rule.substring(0,rule.length()-1); //delete
the last comma before the arrow
                        rule = rule + "->";
                        for (int j = 0; j < assoc.consequentSize(); j++){
```

```
                              rule = rule + assoc.getConsequentItem(j) +
","; 
                              //fuzzyVerifier(assoc.getConsequentItem(j),
reader, ontReader);
                        }
                        rule = rule.substring(0,rule.length()-1); //delete
the last comma
                        if (isFuzzyRule(assoc)){
                              String s = "";
                              for (int j = 0; j < subItemFuzzy.size(); j++)
{
                                    AssociationRule assocJ =
(AssociationRule) subItemFuzzy.get(j);
                                    if (isSubFuzzyRule(assocJ,assoc)){
                                          s = s  +
getRelevantItem(assoc,assocJ) + ",";
                                    }
                              }

                              //if (hasSubRule(assoc,redundantRules)){
                              if (s.compareTo("") != 0){
                                    s= s.substring(0,s.length()-1);
                                    rule =
                                          rule
                                                + " sup="
                                                + assoc.getSupport()
                                                + " conf="
                                                + assoc.getConfidence();

                                    for (int j = 0; j < DN.length &&
!generalized; j++) {
                                          if (isGeneralized(assoc,DN[j])){
                                                generalized = true;
                                                boolean bool = false;
                                                String getItem = "";
                                                for (int k = 0; k <
exceptItems.size() && !bool; k++) {
                                                      GeneralizedExceptFor
genK = (GeneralizedExceptFor) exceptItems.get(k);
                                                      AssociationRule assocK
= genK.getAssoc();
                                                      if
(equalRules(assoc,assocK) && genK.getExcept().size() > 0) {
                                                            bool = true;
                                                            for (int index =
0; index < genK.getExcept().size(); index++) {
                                                                  getItem =
getItem + genK.getExcept().get(index) + ",";
                                                            }
                                                            getItem =
getItem.substring(0,getItem.length() - 1);
                                                      }
                                                }
                                                if (bool)
                                                      rule = rule + " regra
generalizada, exceto por: " + getItem;

                                          }
                                    }
```

```
                                    System.out.println(rule + " (Possuindo
item '" + s + "' com maior relevância!!!)" + "\n");
                                    //countVector++;
                              }else{
                                    rule =
                                          rule
                                                + " sup="
                                                + assoc.getSupport()
                                                + " conf="
                                                + assoc.getConfidence();
                                    for (int j = 0; j < DN.length &&
!generalized; j++) {
                                          if (isGeneralized(assoc,DN[j])){
                                                generalized = true;
                                                boolean bool = false;
                                                String getItem = "";
                                                for (int k = 0; k <
exceptItems.size() && !bool; k++) {
                                                      GeneralizedExceptFor
genK = (GeneralizedExceptFor) exceptItems.get(k);
                                                      AssociationRule assocK
= genK.getAssoc();
                                                      if
(equalRules(assoc,assocK) && genK.getExcept().size() > 0) {
                                                            bool = true;
                                                            for (int index =
0; index < genK.getExcept().size(); index++) {
                                                                  getItem =
getItem + genK.getExcept().get(index) + ",";
                                                            }
                                                            getItem =
getItem.substring(0,getItem.length() - 1);
                                                      }
                                                }
                                                if (bool)
                                                      rule = rule + " regra
generalizada, exceto por: " + getItem;

                                          }
                                    }
                                    System.out.println(rule+"\n");
                              }
                        }else{
                              rule =
                                    rule
                                          + " sup="
                                          + assoc.getSupport()
                                          + " conf="
                                          + assoc.getConfidence();
                              for (int j = 0; j < DN.length && !generalized;
j++) {
                                    if (isGeneralized(assoc,DN[j])){
                                          generalized = true;
                                          boolean bool = false;
                                          String getItem = "";
                                          for (int k = 0; k <
exceptItems.size() && !bool; k++) {
                                                GeneralizedExceptFor genK =
(GeneralizedExceptFor) exceptItems.get(k);
                                                AssociationRule assocK =
genK.getAssoc();
```

```
                                                if (equalRules(assoc,assocK)
&& genK.getExcept().size() > 0) {
                                                      bool = true;
                                                      for (int index = 0;
index < genK.getExcept().size(); index++) {
                                                            getItem =
getItem + genK.getExcept().get(index) + ",";
                                                      }
                                                      getItem =
getItem.substring(0,getItem.length() - 1);
                                                }
                                          }
                                          if (bool)
                                                rule = rule + " regra
generalizada, exceto por: " + getItem;

                                    }
                              }
                              System.out.println(rule+"\n");
                        }

                  }

                  //TESTE TESTE TESTE
                  /*
                  for (int i = 0; i < rules.size(); i++) {
                        rule = "";
                        AssociationRule assoc = (AssociationRule)
rules.get(i);

                        for (int j = 0; j < assoc.antecedentSize(); j++){

                              rule = rule + assoc.getAntecedentItem(j) +
",";
                              fuzzyVerifier(assoc.getAntecedentItem(j),
reader, ontReader);
                        }
                        rule = rule.substring(0,rule.length()-1);//delete
the last comma before the arrow
                        rule = rule + "->";
                        for (int j = 0; j < assoc.consequentSize(); j++){
                              rule = rule + assoc.getConsequentItem(j) +
",";
                              fuzzyVerifier(assoc.getConsequentItem(j),
reader, ontReader);
                        }
                        rule = rule.substring(0,rule.length()-1);//delete
the last comma
                        rule =
                              rule
                                    + " sup="
                                    + assoc.getSupport()
                                    + " conf="
                                    + assoc.getConfidence();
                        System.out.println(rule+"\n");
                  }*/
                  d = new Date();
                  s2 = d.getTime();
                  System
                        .out
```

```
                        .println("Time used in mining association rules: "
+ (s2-s1) +" miliseconds ");
                  } catch (Exception e) {
                  e.printStackTrace();
                  }
      }

      private String getRelevantItem(AssociationRule assoc, AssociationRule
assocJ) {
            // TODO Auto-generated method stub
            String[] ant = new String[assoc.antecedentSize()];
            String[] antJ = new String[assocJ.antecedentSize()];
            String[] cons = new String[assoc.consequentSize()];
            String[] consJ = new String[assocJ.consequentSize()];
            String relevant = "";
            for (int i = 0; i < ant.length; i++) {
                  ant[i] = assoc.getAntecedentItem(i);
            }
            for (int i = 0; i < antJ.length; i++) {
                  antJ[i] = assocJ.getAntecedentItem(i);
            }
            for (int i = 0; i < cons.length; i++) {
                  cons[i] = assoc.getConsequentItem(i);
            }
            for (int i = 0; i < consJ.length; i++) {
                  consJ[i] = assocJ.getConsequentItem(i);
            }
            int count = 0;
            String aux = "";
            for (int i = 0; i < ant.length; i++) {
                  String[] s = ant[i].split("~");
                  if (s.length > 1){
                        for (int j = 0; j < s.length; j++) {
                              for (int k = 0; k < antJ.length; k++) {
                                    if (s[j].compareTo(antJ[k]) == 0){
                                          aux = aux + s[j];
                                          count++;
                                    }
                              }
                        }
                  }
                  if (count == s.length)
                        aux = "";

            }
            relevant = relevant + aux;
            count = 0;
            aux = "";
            for (int i = 0; i < cons.length; i++) {
                  String[] s = cons[i].split("~");
                  if (s.length > 1){
                        for (int j = 0; j < s.length; j++) {
                              for (int k = 0; k < consJ.length; k++) {
                                    if (s[j].compareTo(consJ[k]) == 0){
                                          aux = aux + s[j];
                                          count++;
                                    }
                              }
                        }
                  }
                  if (count == s.length)
```

```
                        aux = "";
                }
                relevant = relevant + aux;
                return relevant;
        }

        private float getAntecedentSupport(AssociationRule assoc, Vector
items) {
                // TODO Auto-generated method stub
                float antecedent = 0;
                Itemset is = new Itemset();
                boolean control = false;
                for (int i = 0; i < assoc.antecedentSize(); i++) {
                        is.addItem(assoc.getAntecedentItem(i));
                }
                for (int i = 0; i < items.size() && !control; i++) {
                        Itemset item = (Itemset) items.get(i);
                        if (is.isEqualTo((item))){
                                control = true;
                                return item.getSupport();
                        }
                }

                return 0;
        }

        private boolean equalRules(AssociationRule assoc, AssociationRule
assocJ) {
                // TODO Auto-generated method stub
                if (assoc.antecedentSize() != assocJ.antecedentSize())
                        return false;
                if (assoc.consequentSize() != assocJ.consequentSize())
                        return false;
                int countAnt = 0, countCons = 0;
                for (int i = 0; i < assoc.antecedentSize(); i++) {
                        for (int j = 0; j < assocJ.antecedentSize(); j++) {

        if(assoc.getAntecedentItem(i).compareTo(assocJ.getAntecedentItem(j))
== 0)
                                        countAnt++;
                        }
                }
                for (int i = 0; i < assoc.consequentSize(); i++) {
                        for (int j = 0; j < assocJ.consequentSize(); j++) {

        if(assoc.getConsequentItem(i).compareTo(assocJ.getConsequentItem(j))
== 0)
                                        countCons++;
                        }
                }
                if (countAnt == assoc.antecedentSize() && countCons ==
assoc.consequentSize())
                        return true;
                return false;
        }

        private boolean isSubRule(AssociationRule assoc, AssociationRule
assocRed, String[] DN, Hashtable domains) {
                // TODO Auto-generated method stub
                String[] ant = new String[assoc.antecedentSize()];
                String[] antRed = new String[assocRed.antecedentSize()];
```

```
            String[] cons = new String[assoc.consequentSize()];
            String[] consRed = new String[assocRed.consequentSize()];
            int countGenAnt = 0, countGenCons = 0, equal = 0;// count
generilized items
            boolean bool = false;
            //inicialização dos arrays de string
            for (int i = 0; i < ant.length; i++) {
                  ant[i] = assoc.getAntecedentItem(i);
            }
            for (int i = 0; i < antRed.length; i++) {
                  antRed[i] = assocRed.getAntecedentItem(i);
            }
            for (int i = 0; i < cons.length; i++) {
                  cons[i] = assoc.getConsequentItem(i);
            }
            for (int i = 0; i < consRed.length; i++) {
                  consRed[i] = assocRed.getConsequentItem(i);
            }
            if (ant.length == antRed.length && cons.length ==
consRed.length){
                  for (int i = 0; i < ant.length; i++) {
                        String getAnt = getDomain(DN,ant[i],domains);
                        for (int j = 0; j < antRed.length; j++) {
                              String getAntRed =
getDomain(DN,antRed[j],domains);
                              if (getAnt.compareTo(getAntRed) == 0){//same
domain
                                    if (ant[i].compareTo(antRed[j]) != 0 &&
getAntRed.compareTo(antRed[j]) == 0)// verify ancestor-descendent relation
                                          countGenAnt++;
                                    if (ant[i].compareTo(antRed[j]) == 0) //
if an item is equal to another one
                                          equal++;
                              }
                        }
                  }
                  if (!((countGenAnt + equal) == ant.length))
                        return false;
                  equal = 0;
                  for (int i = 0; i < cons.length; i++) {
                        String getCons = getDomain(DN,cons[i], domains);
                        for (int j = 0; j < consRed.length; j++) {
                              String getConsRed = getDomain(DN,consRed[j],
domains);
                              if (getCons.compareTo(getConsRed) == 0){
                                    if (cons[i].compareTo(consRed[j]) != 0
&& getConsRed.compareTo(consRed[j]) == 0)
                                          countGenCons++;
                                    if (cons[i].compareTo(consRed[j]) == 0)
                                          equal++;
                              }
                        }
                  }
                  if (!((countGenCons + equal) == cons.length))
                        return false;
                  if ((countGenAnt > 0 || countGenCons > 0))
                        return true;
            }


            return false;
```

```
	}

	private String getDomain(String[] DN, String antecedentItem,
Hashtable domains) {
		// TODO Auto-generated method stub

		for (int index = 0; index < DN.length ; index++) {
			if (antecedentItem.compareTo(DN[index]) == 0){
				return antecedentItem;
			}

		}
		return DN[((Integer)(domains.get(antecedentItem))).intValue()];


	}

	private boolean verifyConsequentAntecedent(AssociationRule assoc,
AssociationRule assoc2, String domain, String[] DN, Hashtable domains) {
		// TODO Auto-generated method stub
		int count = 0;
		boolean bool = false;
		for (int i = 0; i < assoc.consequentSize(); i++) {
			String s1 =
getDomain(DN,assoc.getConsequentItem(i),domains);
			for (int j = 0; j < assoc2.consequentSize(); j++) {

				String s2 =
getDomain(DN,assoc2.getConsequentItem(j),domains);
				if (((s1.compareTo(s2) == 0) &&
(s1.compareTo(domain) == 0) &&
(assoc.getConsequentItem(i).compareTo(assoc2.getConsequentItem(j)) != 0)))
					count++;

	if((assoc.getConsequentItem(i).compareTo(assoc2.getConsequentItem(j))
== 0) && (s1.compareTo(domain) != 0))
					count++;
			}
		}
		if (count == assoc.consequentSize())
			bool = true;
		else
			return false;
		count = 0;
		for (int i = 0; i < assoc.antecedentSize(); i++) {
			for (int j = 0; j < assoc2.antecedentSize(); j++) {

				if
(assoc.getAntecedentItem(i).compareTo(assoc2.getAntecedentItem(j)) == 0)
					count++;
			}
		}
		if (count == assoc.antecedentSize())
			bool = true;
		else
			return false;
		return bool;
	}

	private boolean verifyAntecedentConsequent(AssociationRule assoc,
AssociationRule assoc2, String domain,String[] DN, Hashtable domains) {
```

```
            // TODO Auto-generated method stub
            int count = 0;
            boolean bool = false;
            for (int i = 0; i < assoc.antecedentSize(); i++) {
                  String s1 =
getDomain(DN,assoc.getAntecedentItem(i),domains);
                  for (int j = 0; j < assoc2.antecedentSize(); j++) {

                        String s2 =
getDomain(DN,assoc2.getAntecedentItem(j),domains);
                        if (((s1.compareTo(s2) == 0) &&
(s1.compareTo(domain) == 0) &&
(assoc.getAntecedentItem(i).compareTo(assoc2.getAntecedentItem(j)) != 0)))
                              count++;

      if((assoc.getAntecedentItem(i).compareTo(assoc2.getAntecedentItem(j))
== 0) && (s1.compareTo(domain) != 0))
                              count++;
                  }
            }
            if (count == assoc.antecedentSize())
                  bool = true;
            else
                  return false;
            count = 0;
            for (int i = 0; i < assoc.consequentSize(); i++) {
                  for (int j = 0; j < assoc2.consequentSize(); j++) {

                        if
(assoc.getConsequentItem(i).compareTo(assoc2.getConsequentItem(j)) == 0)
                              count++;
                  }
            }
            if (count == assoc.consequentSize())
                  bool = true;
            else
                  return false;
            return bool;
      }

      private boolean hasSubRule(AssociationRule assoc, Vector
nonRedundantRules) {
            // TODO Auto-generated method stub
            for (int i = 0; i < nonRedundantRules.size(); i++) {
                  AssociationRule assocJ = (AssociationRule)
nonRedundantRules.get(i);
                  if((assoc.antecedentSize() == assocJ.antecedentSize() &&
(assoc.consequentSize() == assocJ.consequentSize())
                              && !isFuzzyRule(assocJ))){
                        if (isSubFuzzyRule(assocJ,assoc)){
                              return true;
                        }
                  }
            }
            return false;
      }

      private boolean eliminateEqualRules(AssociationRule assoc, Vector
nonFuzzyRedundantRules) {
            // TODO Auto-generated method stub
```

```
                  for (int index = 0; index < nonFuzzyRedundantRules.size()
; index++) {
                        AssociationRule assocJ = (AssociationRule)
nonFuzzyRedundantRules.get(index);
                        if (assoc.equals(assocJ))
                              return false;
            }
            return true;
      }

      private boolean isGeneralized(AssociationRule assoc, String DN) {
            // TODO Auto-generated method stub
            for (int i = 0; i < assoc.antecedentSize(); i++) {
                  if (assoc.getAntecedentItem(i).compareTo(DN) == 0)
                        return true;
            }
            for (int i = 0; i < assoc.consequentSize(); i++) {
                  if (assoc.getConsequentItem(i).compareTo(DN) == 0)
                        return true;
            }
            return false;
      }

      private String getSubItem(AssociationRule assoc, Vector
nonRedundantRules, int k,String[] DN,Hashtable domais) {
            // TODO Auto-generated method stub
            String[] ant = new String[assoc.antecedentSize()];

            String[] cons = new String[assoc.consequentSize()];

            int countAnt = 0;
            int countCons = 0;
            String item ="";
            //inicialização dos arrays de string
            for (int i = 0; i < ant.length; i++) {
                  ant[i] = assoc.getAntecedentItem(i);
            }

            for (int i = 0; i < cons.length; i++) {
                  cons[i] = assoc.getConsequentItem(i);
            }

            for (int j = 0; j < nonRedundantRules.size(); j++) {
                  AssociationRule assocJ = (AssociationRule)
nonRedundantRules.get(j);
                  if (isSubFuzzyRule(assoc,assocJ)){
                        countAnt = 0;
                        countCons = 0;
                        if (k!=j && (assocJ.antecedentSize() ==
assoc.antecedentSize()) && (assocJ.consequentSize() ==
assoc.consequentSize())){
                              String[] antJ = new
String[assocJ.antecedentSize()];
                              String[] consJ = new
String[assocJ.consequentSize()];
                              for (int i = 0; i < antJ.length; i++) {
                                    antJ[i] = assocJ.getAntecedentItem(i);
                              }
                              for (int i = 0; i < consJ.length; i++) {
                                    consJ[i] = assocJ.getConsequentItem(i);
                              }
```

```
                              for (int i = 0; i < antJ.length; i++) {
                                    String[] s = antJ[i].split("~");

                                    for (int index = 0; index <
assoc.antecedentSize(); index++) {
                                          for (int aux = 0; aux < s.length;
aux++) {
                                                if
(s[aux].compareTo(ant[index]) == 0){
                                                      countAnt ++;
                                                      String[] s2 =
ant[index].split("~");
                                                      if (s.length >
s2.length)
                                                            item = item +
assoc.getAntecedentItem(index);
                                                }
                                          }
                                    }
                              }
                              if (countAnt == antJ.length){
                                    for (int i = 0; i < consJ.length; i++) {
                                          String[] s = consJ[i].split("~");

                                          for (int index = 0; index <
assoc.consequentSize(); index++) {
                                                for (int aux = 0; aux <
s.length; aux++) {
                                                      if
(s[aux].compareTo(cons[index]) == 0){
                                                            countCons ++;
                                                            String[] s2 =
cons[index].split("~");
                                                            if (s.length >
s2.length)
                                                                  item =
item + assoc.getConsequentItem(index);
                                                      }
                                                }
                                          }
                                    }
                                    if (countCons == consJ.length)
                                          return item;
                              }
                        }
                  }
            }
            return item;
      }

      private boolean isFuzzyRule(AssociationRule assocJ) {
            // TODO Auto-generated method stub
            String[] antJ = new String[assocJ.antecedentSize()];
            String[] consJ = new String[assocJ.consequentSize()];
            for (int i = 0; i < antJ.length; i++) {
                  antJ[i] = assocJ.getAntecedentItem(i);
            }
            for (int i = 0; i < consJ.length; i++) {
                  consJ[i] = assocJ.getConsequentItem(i);
            }
```

```
            for (int i = 0; i < antJ.length; i++) {
                  String[] s = antJ[i].split("~");
                  if (s.length > 1)
                        return true;
            }
            for (int i = 0; i < consJ.length; i++) {
                  String[] s = consJ[i].split("~");
                  if (s.length > 1)
                        return true;
            }
            return false;
      }

      private boolean isSubFuzzyRule(AssociationRule assoc, AssociationRule
assocJ) {
            // TODO Auto-generated method stub
            String[] ant = new String[assoc.antecedentSize()];
            String[] antJ = new String[assocJ.antecedentSize()];
            String[] cons = new String[assoc.consequentSize()];
            String[] consJ = new String[assocJ.consequentSize()];
            int countAnt = 0;
            int countCons = 0;

            //inicialização dos arrays de string
            for (int i = 0; i < ant.length; i++) {
                  ant[i] = assoc.getAntecedentItem(i);
            }
            for (int i = 0; i < antJ.length; i++) {
                  antJ[i] = assocJ.getAntecedentItem(i);
            }
            for (int i = 0; i < cons.length; i++) {
                  cons[i] = assoc.getConsequentItem(i);
            }
            for (int i = 0; i < consJ.length; i++) {
                  consJ[i] = assocJ.getConsequentItem(i);
            }
            if (ant.length == antJ.length && cons.length == consJ.length){
                  for (int i = 0; i < antJ.length; i++) {
                        String[] s = antJ[i].split("~");
                        for (int j = 0; j < ant.length; j++) {
                              for (int k = 0; k < s.length; k++) {
                                    if (s[k].compareTo(ant[j]) == 0)
                                          countAnt ++;
                              }
                        }
                  }
                  if (countAnt == antJ.length){
                        for (int i = 0; i < consJ.length; i++) {
                              String[] s = consJ[i].split("~");
                              for (int j = 0; j < cons.length; j++) {
                                    for (int k = 0; k < s.length; k++) {
                                          if (s[k].compareTo(cons[j]) == 0)
                                                countCons ++;
                                    }
                              }
                        }
                        if (countCons == cons.length)
                              return true;
                  }
            }else
                  return false;
```

```
            return false;
      }

      private boolean subItemset(AssociationRule assoc, AssociationRule
assocJ, String[] DN, Hashtable domains) {
            // TODO Auto-generated method stub
            String[] ant = new String[assoc.antecedentSize()];
            String[] antJ = new String[assocJ.antecedentSize()];
            String[] cons = new String[assoc.consequentSize()];
            String[] consJ = new String[assocJ.consequentSize()];
            int countAnt = 0;
            int countCons = 0;

            //inicialização dos arrays de string
            for (int i = 0; i < ant.length; i++) {
                  ant[i] = assoc.getAntecedentItem(i);
            }
            for (int i = 0; i < antJ.length; i++) {
                  antJ[i] = assocJ.getAntecedentItem(i);
            }
            for (int i = 0; i < cons.length; i++) {
                  cons[i] = assoc.getConsequentItem(i);
            }
            for (int i = 0; i < consJ.length; i++) {
                  consJ[i] = assocJ.getConsequentItem(i);
            }

            if (ant.length == antJ.length && cons.length == consJ.length){

                  for (int i = 0; i < ant.length; i++) {
                        String getAnt = "";
                        getAnt = getDomain(DN,ant[i],domains);
                        /*for (int j = 0; j < DN.length; j++) {
                              if (ant[i].compareTo(DN[j]) == 0)
                                    getAnt = DN[j];
                        }
                        if (getAnt.compareTo("") == 0)
                              getAnt =
DN[((Integer)(domains.get(ant[i]))).intValue()];*/
                        for (int j = 0; j < antJ.length; j++) {//verifica
se os antecedentes são iguais
                              String getAntJ = "";
                              getAntJ = getDomain(DN,antJ[i],domains);
                              /*for (int k = 0; k < DN.length; k++) {
                                    if (antJ[j].compareTo(DN[k]) == 0)
                                          getAntJ = DN[k];
                              }
                              if (getAntJ.compareTo("") == 0)
                                    getAntJ =
DN[((Integer)(domains.get(antJ[j]))).intValue()];*/
                              if ((ant[i].compareTo(getAntJ) == 0) ||
(getAnt.compareTo(antJ[j]) == 0) || (ant[i].compareTo(antJ[j]) == 0))
                                    countAnt++;
                        }
                  }
                  if (countAnt == ant.length){//antecedentes iguais?
                        for (int i = 0; i < cons.length; i++) {
                              String getCons = "";
                              for (int j = 0; j < DN.length; j++) {
                                    if (cons[i].compareTo(DN[j]) == 0)
```

```
                                    getCons = DN[j];
                              }
                              if (getCons.compareTo("") == 0)
                                    getCons =
DN[((Integer)(domains.get(cons[i]))).intValue()];
                              for (int j = 0; j < consJ.length; j++) {
                                    String getConsJ = "";
                                    for (int k = 0; k < DN.length; k++) {
                                          if (consJ[j].compareTo(DN[k]) ==
0)
                                                getConsJ = DN[k];
                                    }
                                    if (getConsJ.compareTo("") == 0)
                                          getConsJ =
DN[((Integer)(domains.get(consJ[j]))).intValue()];
                                    if((cons[i].compareTo(getConsJ) == 0) ||
(getCons.compareTo(consJ[j]) == 0) || (cons[i].compareTo(consJ[j]) == 0)){
                                          countCons++;
                                    }
                              }
                        }
                        if (countCons == cons.length)
                              return true;
                  }
            }else
                  return false;
            return false;
      }

      private Vector generateRulesForEachAntecedent(Hashtable domains,
OntReader ontReader, int[] domainsNumber, String[] DN, String[] getAnt,
String[] getAllCons, float[] sup, float[] conf, int auxAnt) {
            // TODO Auto-generated method stub
            Vector rules = new Vector();
            int count = 0;
            for (int i = 0; i < DN.length; i++) {
                  count = count + ontReader.getClassSize(DN[i]);
            }
            String[] allItens = new String[count];
            float[] getSups = new float[sup.length];
            float[] getConfs = new float[conf.length];

            for (int i = 0; getAnt[i].compareTo("") != 0; i++) {
                  int aux = 0;
                  for (int j = 0; j < getConfs.length; j++) {
                        getConfs[j] = 0;
                        getSups[j] = 0;
                  }
                  String[] get = new String[auxAnt];
                  String getCons = getAllCons[i];
                  if (auxAnt >= 1)
                        get = getAnt[i].split(",");
                  if (get.length == auxAnt){
                        for (int j = i + 1; getAnt[j].compareTo("") != 0;
j++) {
                              String[] getJ = getAnt[j].split(",");

                              for (int k = 0; k < get.length; k++) {

                                    if (get[k].compareTo(getAnt[j]) != 0 &&
getCons.compareTo(getAllCons[j]) == 0 && get.length == getJ.length){
```

```
                                          String s =
DN[((Integer)(domains.get(get[k]))).intValue()];
                                          for (int index = 0; index <
domainsNumber.length; index++) {
                                                if (s.compareTo(DN[index])
== 0){
                                                      if
(domainsNumber[index] == 0)

      domainsNumber[index]++;


      domainsNumber[index]++;
                                                      getConfs[index] =
getConfs[index] + conf[j];
                                                      getSups[index] =
getSups[index] + sup[j];
                                                }
                                                if
(ontReader.getClassSize(DN[index]) == domainsNumber[index]){
                                                      for (int aux2 = 0;
aux2 < get.length; aux2++) {
                                                            if
(get[k].compareTo(get[aux2]) != 0)
                                                                  s = s +
"," + get[aux2];
                                                      }
                                                      rules.add(new
AssociationRule(s,getCons,(getSups[index] + getSups[i]),((getConfs[index] +
getConfs[i])/domainsNumber[index])));
                                                      //System.out.println(s
+ " --> " + getCons + " sup: " + (getSups[index] + getSups[i]) + " conf: "
+ ((getConfs[index] + getConfs[i])/domainsNumber[index]));
                                                      domainsNumber[index] =
0;
                                                }//if
                                          }//for
                                    }//if
                              }//for
                        }//for
                  }// if
            }//   for mais externo
            return rules;
      }//function

      private Vector generateRulesForEachConsequent(Hashtable domains,
OntReader ontReader, int[] domainsNumber, String[] DN, String[] getAllAnt,
String[] getCons, float[] sup, float[] conf, int auxCons) {
            // TODO Auto-generated method stub

            Vector rules = new Vector();
            int aux = 0;
            int count = 0;
            for (int i = 0; i < DN.length; i++) {
                  count = count + ontReader.getClassSize(DN[i]);
            }
            String[] allItens = new String[count];
            float[] getSups = new float[sup.length];
            float[] getConfs = new float[conf.length];
            for (int j = 0; j < getConfs.length; j++) {
                  getConfs[j] = 0;
                  getSups[j] = 0;
```

```
            }
            //          consequentes generalizados
            while (getCons[aux].compareTo("") != 0){
                  String[] get = new String[auxCons];
                  if (auxCons >= 1)
                        get = getCons[aux].split(",");

                  //    ajustar cd dominio do consequente com sua contagem
                  if (get.length == auxCons){
                        for (int i = 0; i < get.length; i++) {
                              String s =
DN[((Integer)(domains.get(get[i]))).intValue()];
                              for (int j = 0; j < domainsNumber.length; j++)
{
                                    if (s.compareTo(DN[j]) == 0){
                                          int k = 0;
                                          while(allItens[k] != null &&
get[i].compareTo(allItens[k]) != 0){
                                                k++;
                                          }
                                          if (allItens[k] == null){//insere
o item e faz a contagem do mesmo no determinado domínio
                                                allItens[k] = get[i];
                                                domainsNumber[j]++;
                                                getConfs[j] = getConfs[j] +
conf[aux];
                                                getSups[j] = getSups[j] +
sup[aux];
                                          }
                                    }
                              }
                        }


                  /*if (getCons[aux].length() <= 1)
                        get =
DN[((Integer)(domains.get(getCons[aux]))).intValue()];
                  else{

                  }*/
                        for (int i = 0; i < get.length; i++){
                              String s =
DN[((Integer)(domains.get(get[i]))).intValue()];
                              for (int j = 0; j < domainsNumber.length; j++)
{
                                    if (ontReader.getClassSize(DN[j]) ==
domainsNumber[j] && s.compareTo(DN[j]) == 0) {
                                          /*for (int k = 0; k <
rules.size(); k++) {
                                                AssociationRule assoc =
(AssociationRule) rules.get(k);

                                          }*/
                                          for (int k = 0; k < get.length;
k++) {
                                                if (get[i].compareTo(get[k])
!= 0)
                                                      s = s + "," +get[k];
                                          }
                                          //s.substring(s.length() - 1);
//last comma
```

```
                                          rules.add(new
AssociationRule(getAllAnt[aux],s,getSups[j],(getConfs[j]/domainsNumber[j]))
);

      //System.out.println(getAllAnt[aux] + " --> " + s + " sup: " +
getSups[j] + " conf: " + (getConfs[j]/domainsNumber[j]));
                                          domainsNumber[j] = 0;
                                    }
                              }
                        }
                  }
                  aux++;
            }// fim consequentes generalizados

            aux = 0;
            // consequentes nao generalizados
            while (getCons[aux].compareTo("") != 0){
                  String[] get = new String[auxCons];
                  //boolean control = false;
                  if (auxCons >= 1)
                        get = getCons[aux].split(",");

                  if (get.length == auxCons) {
                        for (int i = 0; i < get.length; i++) {
                              String s =
DN[((Integer)(domains.get(get[i]))).intValue()];
                              for (int j = 0; j < domainsNumber.length; j++)
{
                                    if (domainsNumber[j] > 0 &&
domainsNumber[j] < ontReader.getClassSize(DN[j]) && s.compareTo(DN[j]) ==
0) {
                                          rules.add(new
AssociationRule(getAllAnt[aux],getCons[aux],sup[aux],conf[aux]));

      //System.out.println(getAllAnt[aux] + " --> " + getCons[aux] + " sup:
" + sup[aux] + " conf: " + conf[aux]);
                                    }
                              }
                        }
                  }
                  aux++;
            }// fim consequentes nao generalizados
            return rules;
      }

      private String verifyAntCons(String string, DBReader reader,
OntReader ontReader) {
            // TODO Auto-generated method stub
            String []DN;
            Hashtable domains = reader.getDomains();

            try {
                  DN = reader.getDomainNames();


            }catch(Exception e){e.printStackTrace();}
            return null;
      }
```

```
	/**
	 * @param args : filename minsupport minconfidence
	 * @author Administrator
	 */
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		MainFile mainFile1 = new MainFile(args);
	}

	/**
	 * @param dbreader: DBReader
	 * @param ontReader: OntReader
	 * @param ontPath: String (path to ontology file (owl))
	 * @param instance: String
	 * @author Administrator
	 */
	public void pathBuilder(DBReader reader, OntReader ontReader, String
instance){

		// get items' domains
		Hashtable domains = reader.getDomains();

		String []DN;

		// Get domain names
		try{
			DN = reader.getDomainNames();
			String classname =
DN[((Integer)(domains.get(instance))).intValue()];
			String path="";
			Vector superClasses = new Vector();
			superClasses = ontReader.getSuperclasses(classname,
instance);

			Enumeration e = superClasses.elements();
			while(e.hasMoreElements()){
				path = path + (String)e.nextElement() + " > ";
			}
			path = path.substring(0,path.length()-3); // delete the
last " > "

			System.out.println(path) ;

		}catch(Exception e){e.printStackTrace();}
	}

	/**
	 * @param item: String (item from the rule)
	 * @author Administrator
	 */
	public void fuzzyVerifier(String item, DBReader reader, OntReader
ontReader){

		Itemset is = new Itemset(), fis; // Itemsets to check if item
is fuzzy
		Vector fuzzyItems = new Vector();

		//check if item is fuzzy
		is.addItem(item);
```

```
            if(is.isFuzzy()){
                  fuzzyItems = is.getItemsetsFromFuzzy();
                  Enumeration fi = fuzzyItems.elements();
                  while(fi.hasMoreElements()){
                        fis = (Itemset)fi.nextElement();
                        // build the path from ontology
                        pathBuilder(reader, ontReader, fis.getFirstItem());
                  }
            }else{
                  pathBuilder(reader, ontReader, is.getFirstItem());

            }
      }
}
```

Classe Set

```
/*
 * Created on 16/07/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package aprioriminer;

/**
 * @author Administrator
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
import java.util.*;

/*

  Maintenance log started on November 30th, 2000

  Nov. 30th, 2000   - added getItemsets method
                              - added and renamed some private methods
                              - improved toString method

*/

/**

   SET.java<P>

   Implements a Set Enumeration Tree, which is a prefix tree used
   for storing and retrieving itemset information.<P>

*/
/*

   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Laurentiu Cristofor
   Lung-Tsung Li
```

```
*/

public class SET
{
  // inner class
  private class HashNode
  {
      Hashtable children;
      float support;

      public HashNode()
      {
        support = 0;
        children = new Hashtable();
      }

      public String toString()
      {
        String s = new String();
        s += "<children: " + children.toString()
      + " support: " + support + ">\n";
        return s;
      }
  }

  private HashNode root;
  private int level = 0;

  /**
   * Create a new empty SET.
   */
  public SET()
  {
      root = new HashNode();
  }

  /**
   * Insert a new itemset in the SET.
   *
   * @param itemset   the itemset to be inserted
   * @exception IllegalArgumentException   <code>itemset</code> is null
   * or is empty
   */
  public void insert(Itemset itemset)
  {
      if (itemset == null || itemset.size() == 0)
        throw new IllegalArgumentException("argument to insert() must be
non null and non empty");

      Itemset is = new Itemset(itemset);
      Object obj;
      HashNode node;
      HashNode walker = root;
      //Integer key;
      String key;

      while (is.hasMoreItems())
        {
      //key = new Integer(is.getNextItem());
      key = is.getNextItem();
```

```
      if ((obj = walker.children.get(key)) != null)
        walker = (HashNode)obj;
      else
        {
            node = new HashNode();
            walker.children.put(key, node);
            walker = node;
        }
        }

      walker.support = is.getSupport();
  }

  /**
   * Return the support for a given itemset.
   *
   * @param itemset   the itemset for which we want to obtain the support
   * @exception IllegalArgumentException   <code>itemset</code> is null
   * or is empty
   * @exception SETException   <code>itemset</code> not found in SET
   * @return   support
   */
  public float getSupport(Itemset itemset)
      throws SETException
  {
      if (itemset == null || itemset.size() == 0)
        throw new IllegalArgumentException("argument to getSupport() must
be non null and non empty");

      Object obj;
      Itemset is = new Itemset(itemset);
      HashNode walker = root;
      //Integer key;
      String key;

      while (is.hasMoreItems())
        {
      key = (String)(is.getNextItem());

      if ((obj = walker.children.get(key)) == null)
        throw new SETException("itemset not found in SET!");

      walker = (HashNode)obj;
        }

      return walker.support;
  }

  /**
   * Return the maximal itemsets of the SET.
   *
   * @return   a vector containing the maximal itemsets from the SET
   */
  public Vector getLargeItemsets()
  {
      Vector v = new Vector();

      if (!root.children.isEmpty())
        traverseGatherLeaves(root, new Itemset(), v);
```

```
    Itemset.pruneNonMaximal(v);

    return v;
}

/**
 * Return the itemsets of the SET.
 *
 * @return   a vector containing the itemsets from the SET
 */
public Vector getItemsets()
{
    Vector v = new Vector();

    if (!root.children.isEmpty())
      traverseGatherAll(root, new Itemset(), v);

    return v;
}

/*
 * A private method which gets called recursively to retrieve itemsets
 * from the leaf nodes of the SET.
 *
 * @param   node node starts from the root node of the SET.
 * @param   itemset for storing hashtable keys as it traverses node by
 * node to the leaf.
 * @param   vector for storing itemsets .
 */
private void traverseGatherLeaves(HashNode node, Itemset itemset,
                          Vector vector)
{
    if (node.children.isEmpty())
      {
    Itemset is = new Itemset(itemset);
    is.setSupport(node.support);
    vector.addElement(is);
    return;
      }

    Enumeration e = node.children.keys();

    while (e.hasMoreElements())
      {
    //Integer key = (Integer)e.nextElement();
    String key = (String)e.nextElement();
    //itemset.addItem(key.intValue());
    itemset.addItem(key);
    traverseGatherLeaves((HashNode)node.children.get(key), itemset,
                         vector);
    itemset.removeLastItem();
      }
}

/*
 * A private method which gets called recursively to retrieve itemsets
 * from all the nodes of the SET.
 *
 * @param   node node starts from the root node of the SET.
 * @param   itemset for storing hashtable keys as it traverses node by
 * node to the leaf.
```

```
 * @param   vector for storing itemsets .
 */
private void traverseGatherAll(HashNode node, Itemset itemset,
                      Vector vector)
{
    if (node.children.isEmpty())
      return;

    Enumeration e = node.children.keys();

    while (e.hasMoreElements())
      {
    //Integer key = (Integer)e.nextElement();
    String key = (String)e.nextElement();
    HashNode child_node = (HashNode)node.children.get(key);
    //itemset.addItem(key.intValue());
    itemset.addItem(key);
    Itemset is = new Itemset(itemset);
    is.setSupport(child_node.support);
    vector.addElement(is);

    traverseGatherAll(child_node, itemset, vector);
    itemset.removeLastItem();
      }
}

/*
 * A private method which gets called recursively to retrieve itemsets
 * from each node of the SET and print them to a String.
 *
 * @param node   node to traverse
 * @param s   StringBuffer in which we store the representation
 * of the nodes we saw so far
 */
private void traversePrint(HashNode node, StringBuffer sb)
{
    level++;

    if (node.children.isEmpty())
      {
    level--;
    return;
      }

    Enumeration e = node.children.keys();
    while (e.hasMoreElements())
      {
    Integer key = (Integer)e.nextElement();
    for (int i = 1; i < level; i++)
      sb.append("   ");
    sb.append("<" + key.toString() + ">" + ":[" + level + "]\n");
    traversePrint((HashNode)node.children.get(key), sb);
      }

    level--;
}

/**
 * Return a string representation of the SET.
 *
 * @return   string representation of SET
```

```
     */
    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        traversePrint(root, sb);
        return sb.toString();
    }

    /**
     * for testing purposes only !!!
     */
    public static void main(String[] args)
    {
/*      SET set = new SET();

        Itemset is2 = new Itemset();
        Itemset is1 = new Itemset();
        Itemset is3 = new Itemset();
        Itemset is4 = new Itemset();
        Itemset is5 = new Itemset();
        Itemset is6 = new Itemset();
        Itemset is7 = new Itemset();
        Itemset is8 = new Itemset();
        Itemset is9 = new Itemset();

        is1.addItem(1);
        is1.addItem(2);
        is1.addItem(3);
        is1.addItem(4);
        is1.setSupport((float)0.4);
        is2.addItem(1);
        is2.addItem(2);
        is2.addItem(3);
        is2.addItem(7);
        is2.setSupport((float)0.3);
        is3.addItem(1);
        is3.addItem(2);
        is3.addItem(3);
        is3.addItem(5);
        is3.addItem(6);
        is3.setSupport((float)0.5);
        is4.addItem(1);
        is4.addItem(2);
        is4.addItem(4);
        is4.setSupport((float)0.65);
        is5.addItem(1);
        is5.addItem(2);
        is5.addItem(5);
        is5.setSupport((float)0.6);
        is6.addItem(2);
        is6.addItem(4);
        is6.addItem(5);
        is6.setSupport((float)0.55);
        is7.addItem(1);
        is7.addItem(2);
        is7.setSupport((float)0.2);
        is8.addItem(2);
        is8.addItem(4);
        is8.addItem(7);
        is8.addItem(8);
        is8.setSupport((float)0.52);
```

```
      is9.addItem(2);
      is9.addItem(4);
      is9.addItem(6);
      is9.setSupport((float)0.59);

      set.insert(is2);
      set.insert(is3);
      set.insert(is1);
      set.insert(is4);
      set.insert(is5);
      set.insert(is6);
      set.insert(is7);
      set.insert(is8);
      set.insert(is9);

      System.out.println(set);

      try
        {
      System.out.println("Support for is1: " + set.getSupport(is1));
      System.out.println("Support for is2: " + set.getSupport(is2));
      System.out.println("Support for is3: " + set.getSupport(is3));
      System.out.println("Support for is4: " + set.getSupport(is4));
      System.out.println("Support for is5: " + set.getSupport(is5));
      System.out.println("Support for is6: " + set.getSupport(is6));
      System.out.println("Support for is7: " + set.getSupport(is7));
      System.out.println("Support for is8: " + set.getSupport(is8));
      System.out.println("Support for is9: " + set.getSupport(is9));
        }
      catch (SETException e)
        {
      System.out.println(e);
        }

      Vector v = set.getLargeItemsets();
      System.out.println("\nLarge itemsets are: " + '\n');
      for(int i = 0; i < v.size(); i++)
        System.out.println(v.get(i));*/
  }
}
```

Classe SETException

```
/*
 * Created on 16/07/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package aprioriminer;

/**

   SETException.java<P>

   Exception thrown by SET.

*/
/*
```

```
   This file is a part of the ARMiner project.

   (P)1999-2000 by ARMiner Server Team:

   Dana Cristofor
   Laurentiu Cristofor

*/

public class SETException extends Exception
{
  /**
   * @serial
   */
  private String text;

  /**
   * Constructor.
   *
   * @param name   error message
   */
  public SETException(String name)
  {
    text = name;
  }

  /**
   * Converts the exception to a String object.
   */
  public String toString()
  {
    return text;
  }
}
```

Classe SimilarityMatrix

```
/*
 * Created on 08/12/2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package aprioriminer;

/**
 * @author Administrator
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
public class SimilarityMatrix {

      private int size = 20; //initial size of the square matrix
      private float[][] matrix; //the matrix itself

      public SimilarityMatrix() {
            matrix = new float[size][size];
            initializeSM();
```

```
	}

	public SimilarityMatrix(int size) {
		matrix = new float[size][size];
		this.size = size;
		initializeSM();
	}

	/**
	 * Add the similarity value in matrix.
	 *
	 * @param col	the column of the matrix to add
	 * @param row	the row of the matrix to add
	 * @exception IndexOutOfBoundsException	col or row is an invalid
index
	 */
	public void add(int row, int col, float sim) {
		if (!testIndexes(row, col))
			throw new IndexOutOfBoundsException("Matrix index out of
bounds");

		else {
			matrix[row][col] = sim;
			matrix[col][row] = sim;
		}
	}

	/**
	 * Return the similarity value in matrix.
	 *
	 * @param col	the column of the matrix to get
	 * @param row	the row of the matrix to get
	 * @exception IndexOutOfBoundsException	col or row is an invalid
index
	 * @return	the similarity value
	 */
	public float get(int row, int col) {
		if (!testIndexes(row, col))
			throw new IndexOutOfBoundsException("Matrix index out of
bounds");

		return matrix[row][col];
	}
	/**
	 * Return the similarity value in matrix.
	 *
	 * @param col	the column of the matrix to get
	 * @param row	the row of the matrix to get
	 * @return	the similarity value
	 */
	public boolean testIndexes(int row, int col) {
		return (row >= 0 && col >= 0 && row < size && col < size);
	}

	//Initialize the Similarity Matrix, putting "1.0" values in the
diagonal
	public void initializeSM(){
		for(int k = 0; k < size; k++)
			add(k,k,(float)1.0);
	}
}
```