UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
ISSN 0103-2569

# Projeto e Implementação do Algoritmo de Aprendizado de Máquina Semi-Supervisionado CO-TRAINING

**Edson Takashi Matsubara**
**Maria Carolina Monard**

**Nº 229**

RELATÓRIOS TÉCNICOS

**São Carlos – SP**
**Abr./2004**

# Projeto e Implementação do Algoritmo de Aprendizado de Máquina Semi-supervisionado

## CO-TRAINING

**Edson Takashi Matsubara**
**Maria Carolina Monard**

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação e Estatística
C.P. 668, 13560-970 - São Carlos, SP - Brasil
e-mail: {edsontm, mcmonard}@icmc.usp.br

**Resumo:** Em bases de dados do mundo real, o número de exemplos rotulados é muito pequeno, quando não inexistente, tornando inviável o uso de Aprendizado de Máquina supervisionado nessas bases. Uma nova área de pesquisa em Aprendizado de Máquina está relacionada com algoritmos de aprendizado semi-supervisionado. Nesse tipo de aprendizado, dado um pequeno conjunto de exemplos rotulados, o objetivo principal consiste em rotular mais exemplos para serem utilizados para qualquer algoritmo de Aprendizado de Máquina supervisionado na indução de melhores classificadores. Neste trabalho é descrito em detalhes o projeto e implementação do algoritmo de aprendizado semi-supervisionado CO-TRAINING juntamente com um exemplo de execução.

**Palavras Chaves:** Aprendizado Semi-supervisionado, Aprendizado de Máquina.

**abril 2004**

Trabalho realizado com apoio CAPES.

# Sumário

# Lista de Figuras

# Lista de Abreviaturas

**AM** Aprendizado de Máquina

**ARFF** *Attribute-Relation File Format*

**MD** Mineração de Dados

**PLN** Processamento de Língua Natural

**RI** Recuperação de Informação

**MT** Mineração de Textos

**Labic** Laboratório de Inteligência Computacional

**KDD** *Knowledge Discovery in Databases*

**DOL** *Discover Object Library*

**DSX** *Discover Standard Sintax*

**CBR** *Case-Based Reasoning*

**ILP** *Inductive Logic Programming*

# Lista de Tabelas

# 1 Introdução

Aprendizado de Máquina (AM) pesquisa métodos computacionais relacionados à aquisição automática de novos conhecimentos, novas habilidades e novas formas de organizar o conhecimento já existente (Mitchell, 1997). Em geral, Aprendizado de Máquina pode ser dividido em supervisionado e não supervisionado. No aprendizado supervisionado é fornecido ao algoritmo de aprendizado (indutor), um conjunto de exemplos de treinamento para os quais o rótulo da classe associada é conhecido. Normalmente, cada exemplo é descrito por um vetor de atributos e o rótulo da classe associada. O objetivo do algoritmo de indução é construir um classificador (ou hipótese) que possa determinar corretamente a classe de novos exemplos ainda não rotulados, ou seja, exemplos que não tenham o rótulo da classe. Já no aprendizado não supervisionado, o indutor analisa os exemplos fornecidos e tenta determinar se alguns deles podem ser agrupados de alguma maneira, utilizando alguma medida de similaridade, formando agrupamentos ou clusters.

No caso de AM supervisionado é necessário um número significativo de exemplos rotulados para a indução de um bom classificador. No entanto, no mundo real, o número de exemplos rotulados é muito pequeno, quando não inexistente, o que torna inviável o uso de AM supervisionado para muitos casos.

Uma nova área de pesquisa em AM está relacionada com algoritmos que aprendem utilizando uma combinação das facilidades oferecidas no aprendizado supervisionado e não supervisionado. Esses algoritmos são chamados de algoritmos de aprendizado semi-supervisionado. Um dos objetivos do aprendizado semi-supervisionado é, entre outros, rotular mais exemplos para fornecer a um algoritmo de AM supervisionado para que este possa induzir melhores hipóteses.

Neste trabalho é descrito um novo algoritmo de aprendizado semi-supervisionado, proposto por Blum & Mitchell (1998), denominado CO-TRAINING. O método de CO-TRAINING consiste da indução de dois classificadores, cada um deles induzido utilizando uma visão (descrição) diferente do conjunto de exemplos rotulados (exemplos de treinamento) que cooperam entre si. Cada um desses classificadores somente rotula algum exemplo de seu conjunto de exemplos se tiver um alto grau de certeza da classificação desse exemplo. Esses novos exemplos rotulados são então adicionados ao conjunto original de exemplos rotulados de ambos indutores e o processo é repetido até não ser possível rotular exemplos com alto grau de certeza. Assim, sempre que um desses dois classificadores rotular um exemplo, ele colabora na melhoria da precisão do outro classificador, por estar incrementando o número de exemplos do conjunto de treinamento. Assim, após a execução de CO-TRAINING, tem-se um conjunto de trei-

namento com maior número de exemplos rotulados, para ser utilizado por qualquer algoritmo de AM supervisionado.

O objetivo deste trabalho é descrever em detalhes o projeto e a implementação do algoritmo de CO-TRAINING, bem como os dois modos de execução implementados e os parâmetros que podem ser utilizados para a execução de diversos experimentos.

Este trabalho está organizado da seguinte forma: na Seção 2 é apresentada uma visão geral do algoritmo de CO-TRAINING; na Seção 3 são descritos alguns detalhes de implementação bem como os parâmetros de execução; na Seção 4 são mostrados algum exemplos de execução e as consideração finais são apresentadas na Seção 5.

# 2 O Algoritmo CO-TRAINING

Em bases de dados do mundo real, o número de exemplos rotulados é muito pequeno, quando não inexistente, tornando inviável nesses casos o uso de Aprendizado de Máquina supervisionado. Logo, uma alternativa seria rotular manualmente os exemplos. Porém, esse processo é extremamente caro e, às vezes, impossível, dado o grande volume de dados a ser rotulado. Caso existir um pequeno número de exemplos rotulados, que poderiam ter sido rotulados manualmente por um especialista, o algoritmo CO-TRAINING tenta, entre outros, solucionar o problema de rotular mais dados do conjunto de dados não-rotulados para, assim, tentar melhorar a precisão de algoritmos de AM supervisionado.

Para ilustrar a idéia de CO-TRAINING pode ser utilizada a seguinte analogia: o ser humano tem a capacidade de descrever objetos, ou situações observadas, de formas diferentes. Dadas, por exemplo, as duas descrições que descrevem a mesma situação, o ser humano é capaz de aprender o mesmo conceito, mas induzindo duas hipóteses diferentes, cada uma delas consistente com uma dessas descrições da mesma situação do mundo real. Após o aprendizado, as duas descrições de uma mesma, mas nova, situação para as respectivas hipóteses induzidas, elas serão capazes, com alta probabilidade, de classificar cada uma dessas duas descrições, ou exemplos, com o mesmo rótulo da classe associada. O método de CO-TRAINING baseia-se nessa idéia para tentar rotular mais exemplos com o objetivo de melhorar a performance de AM supervisionado, quando o número original de exemplos rotulados não é suficiente para um bom aprendizado. Em outras palavras, CO-TRAINING, utilizando duas descrições de mundo, usa dois algoritmos de AM supervisionado, ou o mesmo algoritmo, para induzir duas hipóteses (classificadores). Cada hipótese é induzida utilizando como conjunto de treinamento o conjunto de exemplos que representa uma dessas descrições de mundo. Dessa maneira, tem-se duas hipóteses sobre a mesma situação, mas cada uma delas induzida sobre os mesmos exemplos mas descritos segundo uma visão diferente.

No presente trabalho, a maneira utilizada para a representação de exemplos é uma tabela no formato atributo-valor. Na Tabela 1 na página seguinte é apresentado o formato geral de uma tabela atributo-valor com $N$ exemplos $E_i$ com $i = 1, ..., N$, na forma $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N)\}$ para alguma função desconhecida $y = f(\mathbf{x})$. Os $\mathbf{x}_i$ são vetores da forma $< x_{i1}, x_{i2}, ..., x_{iM} >$, com valores discretos ou contínuos. Assim, $x_{ij}$ refere-se ao valor do atributo (ou *feature*) $j$, denominado $\mathbf{X}_j$, do exemplo $E_i$, como mostra a Tabela 1. Os valores $y_i$ referem-se ao valor do atributo $\mathbf{Y}$, que é o atributo *classe*.

Os valores $y_i$ pertencem a um conjunto $C$ de classes $C_v$ com $v = 1, ..., N_{Cl}$, ou seja, $C =$

| | $X_1$ | $X_2$ | ... | $X_M$ | $Y$ |
|---|---|---|---|---|---|
| $E_1$ | $x_{11}$ | $x_{12}$ | ⋮ | $x_{1M}$ | $y_1$ |
| $E_2$ | $x_{21}$ | $x_{22}$ | ⋮ | $x_{2M}$ | $y_2$ |
| ⋮ | ⋮ | ⋮ | ⋱ | ⋮ | ⋮ |
| $E_N$ | $x_{N1}$ | $x_{N2}$ | ⋮ | $x_{NM}$ | $y_N$ |

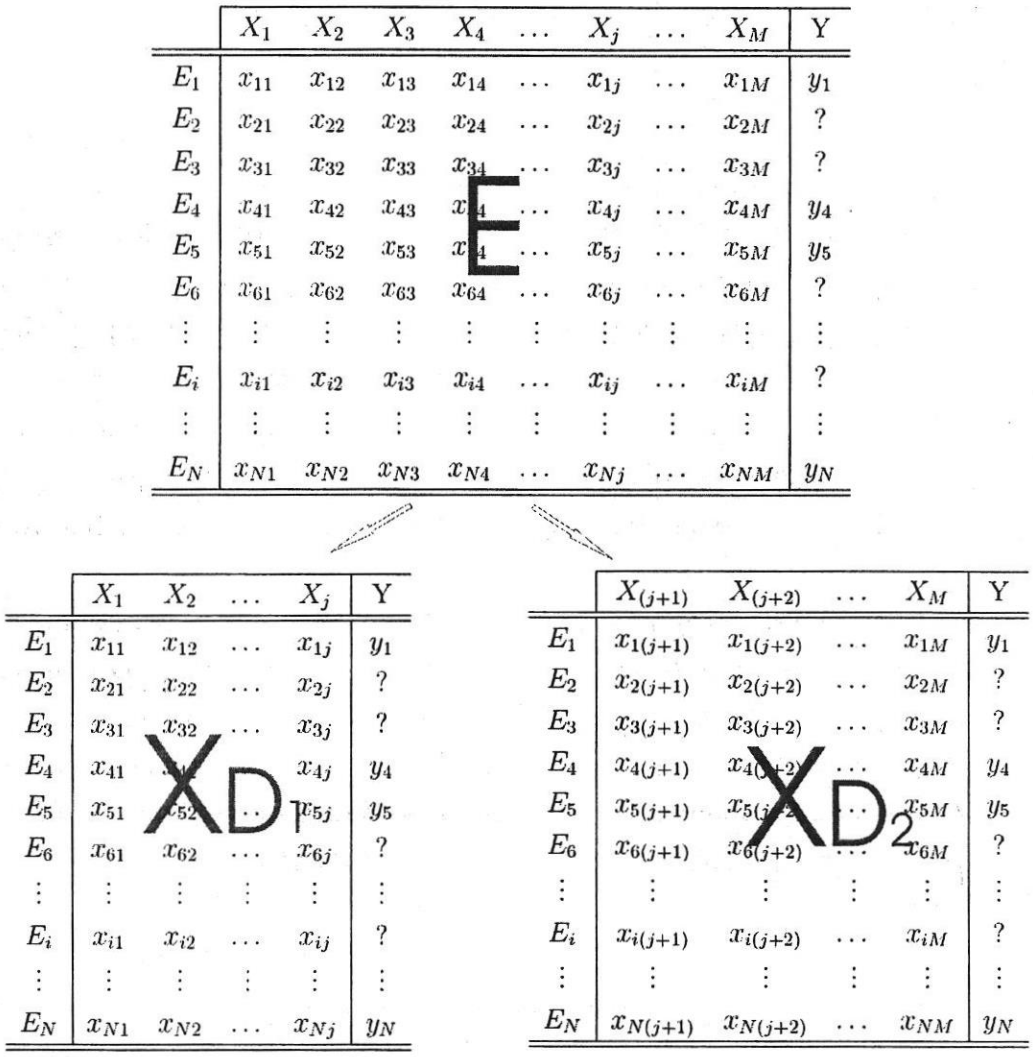
Tabela 1: Exemplos no formato atributo-valor

$\{C_1, ..., C_{N_{Cl}}\}$, quando se trata de *classificação* (valores discretos) ou ao conjunto de números reais (valores contínuos) no caso de *regressão*. CO-TRAINING é um algoritmo de classificação.

Para criar as duas descrições de CO-TRAINING é necessário particionar o conjunto de atributos $\mathbf{X}$, que descreve os exemplos, em dois subconjuntos, $\mathbf{X}_{D_1}$ e $\mathbf{X}_{D_2}$, que constituem as duas descrições da situação e descrevem os mesmos exemplos tal que $\mathbf{X} = \mathbf{X}_{D_1} \cup \mathbf{X}_{D_2}$ e $\mathbf{X}_{D_1} \cap \mathbf{X}_{D_2} = \emptyset$ como ilustra a Figura 1 na próxima página, na qual, por simplicidade, é considerado que $\mathbf{X}_{D_1} = \{X_1, X_2, ..., X_j\}$ e $\mathbf{X}_{D_2} = \{X_{j+1}, X_{j+2}, ..., X_M\}$. Exemplos com valor de Y igual a "?" representam exemplos não rotulados.

Além da separação em duas descrições, o conjunto de exemplos $E = \{E_1, ..., E_N\}$ deve ser separado em dois subconjuntos $L$ e $U$, também conhecidos como conjunto de exemplos rotulados (*Labeled*) e não-rotulados (*Unlabeled*) respectivamente. O subconjunto $L \subseteq E$ que contém exemplos que possuem o atributo classe conhecido é, por sua vez, dividido em duas descrições $L_{D_1}$ e $L_{D_2}$, tal que $L = L_{D_1} \cup L_{D_2}$ e $L_{D_1} \cap L_{D_2} = \emptyset$. Analogamente, o subconjunto $U \subseteq E$ que contém os exemplos nos quais o atributo classe é desconhecido é também dividido em duas descrições $U_{D_1}$ e $U_{D_2}$, tal que $U = U_{D_1} \cup U_{D_2}$ e $U_{D_1} \cap U_{D_2} = \emptyset$. Os dados de entrada do algoritmo de CO-TRAINING consistem desses quatro conjuntos de exemplos $L_{D_1}$, $L_{D_2}$, $U_{D_1}$ e $U_{D_2}$. Na Figura 2 encontram-se ilustrados esses quatro conjuntos de exemplos e no Algoritmo 1 na página 7 são descritos os principais passos de CO-TRAINING descritos a seguir.

No primeiro passo do algoritmo antes da iteração, são criados dois subconjuntos $U'_{D_1}$ e $U'_{D_2}$, tal que $U' = U'_{D_1} \cup U'_{D_2}$ e $U'_{D_1} \cap U'_{D_2} = \emptyset$, sendo $U'$ um subconjunto de, geralmente, poucos exemplos de $U$, ou seja, esses dois subconjuntos de exemplos não rotulados $U'_{D_1}$ e $U'_{D_2}$ são subconjuntos de $U_{D_1}$ e $U_{D_2}$ respectivamente. Os exemplos que compõem $U'_{D_1}$ e $U'_{D_2}$ são removidos de $U_{D_1}$ e $U_{D_2}$ a fim de verificar as condições $U'_{D_1} \cap U_{D_1} = \emptyset$ e $U'_{D_2} \cap U_{D_2} = \emptyset$ criando assim conjuntos disjuntos. Após cada iteração com os conjuntos de exemplos rotulados $L_{D_1}$ e $L_{D_2}$ são induzidos, respectivamente, dois classificadores $h_{D_1}$ e $h_{D_2}$ os quais são utilizados para rotular exemplos não rotulados de $U'_{D1}$ e $U'_{D2}$ respectivamente. No fim deste processo, $R'_{D_1}$ e $R'_{D_2}$ contém, respectivamente, o conjunto de exemplos rotulados nesse passo da iteração, os quais são argumentos da função




| | $X_1$ | $X_2$ | $X_3$ | $X_4$ | ... | $X_j$ | ... | $X_M$ | Y |
|---|---|---|---|---|---|---|---|---|---|
| $E_1$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | ... | $x_{1j}$ | ... | $x_{1M}$ | $y_1$ |
| $E_2$ | $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | ... | $x_{2j}$ | ... | $x_{2M}$ | ? |
| $E_3$ | $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | ... | $x_{3j}$ | ... | $x_{3M}$ | ? |
| $E_4$ | $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | ... | $x_{4j}$ | ... | $x_{4M}$ | $y_4$ |
| $E_5$ | $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | ... | $x_{5j}$ | ... | $x_{5M}$ | $y_5$ |
| $E_6$ | $x_{61}$ | $x_{62}$ | $x_{63}$ | $x_{64}$ | ... | $x_{6j}$ | ... | $x_{6M}$ | ? |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $E_i$ | $x_{i1}$ | $x_{i2}$ | $x_{i3}$ | $x_{i4}$ | ... | $x_{ij}$ | ... | $x_{iM}$ | ? |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $E_N$ | $x_{N1}$ | $x_{N2}$ | $x_{N3}$ | $x_{N4}$ | ... | $x_{Nj}$ | ... | $x_{NM}$ | $y_N$ |

| | $X_1$ | $X_2$ | ... | $X_j$ | Y |
|---|---|---|---|---|---|
| $E_1$ | $x_{11}$ | $x_{12}$ | ... | $x_{1j}$ | $y_1$ |
| $E_2$ | $x_{21}$ | $x_{22}$ | ... | $x_{2j}$ | ? |
| $E_3$ | $x_{31}$ | $x_{32}$ | ... | $x_{3j}$ | ? |
| $E_4$ | $x_{41}$ | $x_{42}$ | ... | $x_{4j}$ | $y_4$ |
| $E_5$ | $x_{51}$ | $x_{52}$ | ... | $x_{5j}$ | $y_5$ |
| $E_6$ | $x_{61}$ | $x_{62}$ | ... | $x_{6j}$ | ? |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $E_i$ | $x_{i1}$ | $x_{i2}$ | ... | $x_{ij}$ | ? |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $E_N$ | $x_{N1}$ | $x_{N2}$ | ... | $x_{Nj}$ | $y_N$ |

| | $X_{(j+1)}$ | $X_{(j+2)}$ | ... | $X_M$ | Y |
|---|---|---|---|---|---|
| $E_1$ | $x_{1(j+1)}$ | $x_{1(j+2)}$ | ... | $x_{1M}$ | $y_1$ |
| $E_2$ | $x_{2(j+1)}$ | $x_{2(j+2)}$ | ... | $x_{2M}$ | ? |
| $E_3$ | $x_{3(j+1)}$ | $x_{3(j+2)}$ | ... | $x_{3M}$ | ? |
| $E_4$ | $x_{4(j+1)}$ | $x_{4(j+2)}$ | ... | $x_{4M}$ | $y_4$ |
| $E_5$ | $x_{5(j+1)}$ | $x_{5(j+2)}$ | ... | $x_{5M}$ | $y_5$ |
| $E_6$ | $x_{6(j+1)}$ | $x_{6(j+2)}$ | ... | $x_{6M}$ | ? |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $E_i$ | $x_{i(j+1)}$ | $x_{i(j+2)}$ | ... | $x_{iM}$ | ? |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $E_N$ | $x_{N(j+1)}$ | $x_{N(j+2)}$ | ... | $x_{NM}$ | $y_N$ |

Figura 1: Duas descrições do conjunto de exemplos

*melhores/2* que é responsável pela seleção de exemplos que foram "melhor" rotulados, e com o mesmo rótulo, pelos respectivos classificadores $h_{D_1}$ e $h_{D_2}$. A função *melhores/2* pode ser considerada como a mais importante na implementação do algoritmo CO-TRAINING, e é descrita em maiores detalhes na Seção 3.1.4 na página 15. Após a execução da função *melhores/2*, os exemplos que foram "melhor" rotulados são adicionados, respectivamente, aos conjuntos de exemplos de treinamento $L_{D_1}$ e $L_{D_2}$. No caso de ainda existirem exemplos não rotulados, são utilizados os mesmos exemplos ainda não rotulados em $U_{D1}$ e $U_{D2}$ os quais são adicionados, respectivamente, aos conjuntos $U'_{D1}$ e $U'_{D2}$ e o processo é repetido.

Ainda que a saída do algoritmo consiste do conjunto de exemplos rotulados por CO-TRAINING, para serem utilizados por qualquer algoritmo de AM. no artigo original do método Blum & Mitchell (1998) são também considerados, em cada iteração dois classificadores $h_{D_1}$ e $h_{D_2}$ criado um terceiro classicador baseado nos dois primeiros, denominado classificador combinado. Esse classificador utiliza uma particularidade

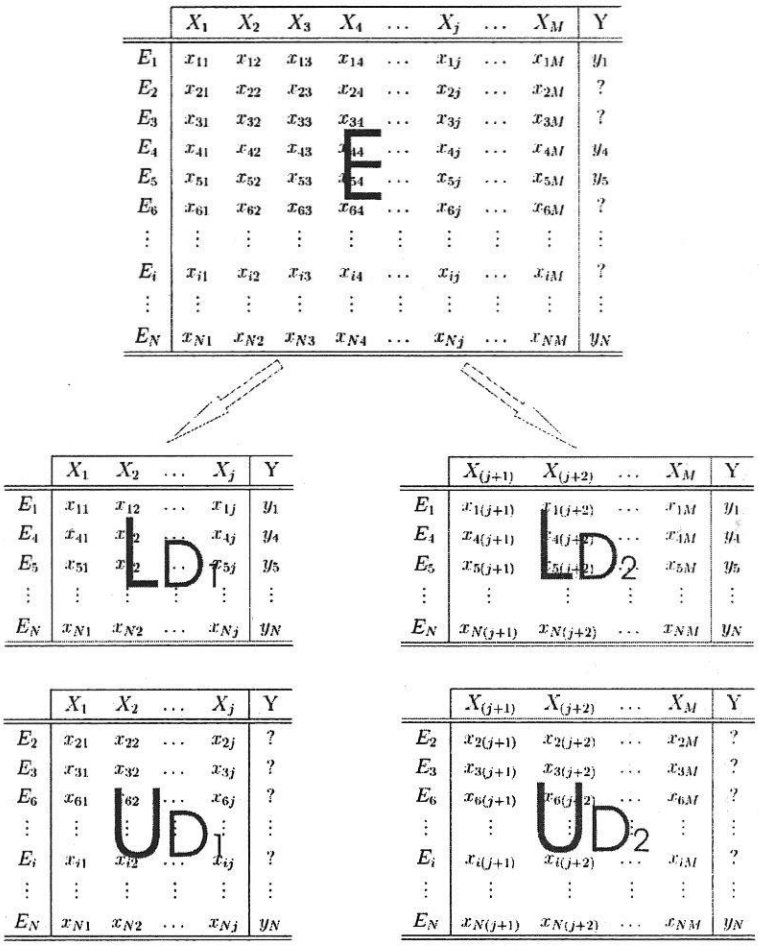



Figura 2: Conjunto de exemplos $L_{D_1}$, $L_{D_2}$ $U_{D_1}$ e $U_{D_2}$

do algoritmo *NaiveBayes* que fornece, além da classe, a probabilidade do rótulo do exemplo pertencer às classes.

O classificador combinado calcula a probabilidade $P(C_v|E_i)$ da classe $C_v$, onde $v = 1..N_{Cl}$ e $N_{Cl}$ é o número de classes possíveis e $E_i$ é o exemplo a ser rotulado. No entanto, os dois classificadores $h_{D_1}$ e $h_{D_2}$ fornecem apenas as probabilidades respectivas as suas descrições. Para calcular a probabilidade $P(C_v|E_i)$ considerando ambos os classificadores como se fosse um único classificador, cada exemplo $E_i$ pode ser dado como $(\mathbf{D1}_i \cup \mathbf{D2}_i)$ no qual $\mathbf{D1}_i$ e $\mathbf{D2}_i$ correspondem aos valores dos atributos da primeira e segunda descrição, respectivamente, do exemplo $E_i$. Portanto os dois classificadores, $h_{D_1}$ e $h_{D_2}$, fornecem $P(C_v|\mathbf{D1}_i)$ e $P(C_v|\mathbf{D2}_i)$ ou seja, a probabilidade do exemplo $E_i$ ser da classe $C_v$ dado pelo classificador induzido a partir de $L_{D_1}$ e a probabilidade de ser da classe $C_v$ dado pelo classificador induzido a partir de $L_{D_2}$. Como $v$ varia de 1 a $N_{Cl}$, os classificadores fornecem as probabilidades do exemplo pertecer a todas as $N_{Cl}$ classes. Para obter $P(C_v|E_i)$ são multiplicadas as probabilidades $P(C_v|\mathbf{D1}_i)$ e $P(C_v|\mathbf{D2}_i)$ calculadas por $h_{D_1}$ e $h_{D_2}$ para cada classe como mostrado no Algoritmo 2.

**Algoritmo 1:** cotraining

**Input**: $L_{D_1}, L_{D_2}, U_{D_1}, U_{D_2}, k$

**Output**: $L_{D_1}, L_{D_2}$

Construir $U'_{D_1}$ e $U'_{D_2}$ como descrito anteriormente;
$U_{D_1} = U_{D_1} - U'_{D_1}$;
$U_{D_2} = U_{D_2} - U'_{D_2}$;
**for** $i = 0$ *to* $k$ **do**
    induzir o classificador $h_{D_1}$ a partir dos exemplos de treinamento $L_{D_1}$;
    induzir o classificador $h_{D_2}$ a partir dos exemplos de treinamento $L_{D_2}$;
    $R'_{D_1}$ = todos os exemplos de $U'_{D_1}$ rotulados com o classificador $h_{D_1}$;
    $R'_{D_2}$ = todos os exemplos de $U'_{D_2}$ rotulados com o classificador $h_{D_2}$;
    $(R_{D_1}, R_{D_2}) = melhores(R'_{D_1}, R'_{D_2})$;
    **if** $R_{D_1} = \emptyset$ **then** return $L_{D_1}, L_{D_2}$ ;
    $L_{D_1} = L_{D_1} \cup R_{D_1}$;
    $L_{D_2} = L_{D_2} \cup R_{D_2}$;
    **if** $U_{D_1} = \emptyset$ **then** return $L_{D_1}, L_{D_2}$ **else**
        retirar aleatoriamente alguns exemplos de $U_{D_1}$ e os exemplos respectivos de $U_{D_2}$ e adicioná-los a $U'_{D_1}$ e $U'_{D_2}$ respectivamente;
    **end**
**end**
return $L_{D_1}, L_{D_2}$;

**Algoritmo 2:** Classificador Combinado

**for** $i = 1$ *to* $N$ **do**
    **for** $k = 1$ *to* $N_{Cl}$ **do**
        $P(C_v|E_i) \leftarrow P(C_v|\mathbf{D1}_i)P(C_v|\mathbf{D2}_i)$;
    **end**
**end**

Esse terceiro classificador permite representar CO-TRAINING como um único classificador, possibilitando assim comparações com outros algoritmos.

# 3 Implementação e Parâmetros de execução

O algoritmo CO-TRAINING foi desenvolvido em Perl (Wall et al., 1996) utilizando a biblioteca de classes *Discover Object Library* (DOL) (Batista & Monard, 2003). A DOL faz parte de um projeto maior, denominado DISCOVER, em desenvolvimento no Laboratório de Inteligência Computacional (Labic) [1]. O DISCOVER tem como objetivo fornecer um ambiente integrado para apoiar todas as etapas do processo de descoberta de conhecimento [2], oferecendo funcionalidades voltadas para o aprendizado de máquina, mineração de dados e mineração de textos. De uma maneira geral, o sistema DISCOVER pode ser entendido como um conjunto de métodos que são aplicados sobre os dados ou sobre o conhecimento extraído a partir dos dados. Assim, é muito importante que o ambiente DISCOVER ofereça uma base sólida para manipular dados e conhecimento. Essa base é composta por sintaxes padrões para a representação de dados e de conhecimento, e por bibliotecas que oferecem um conjunto de funcionalidades básicas de manipulação de dados e conhecimento. Atualmente, existem definidas sintaxes padrões para a representação de dados e para a representação do conhecimento induzido e diversos algoritmos de aprendizado de máquina, bem como bibliotecas que oferecem diversas funcionalidades sobre essas sintaxes padrão. Entre as diversas funcionalidades da biblioteca de classes DOL, pode-se citar: a manipulação de atributos e dados utilizados por sistemas de aprendizado bem como a integração de vários sistemas de aprendizado implementados pela comunidade e pelo nosso grupo de pesquisa; integração com sistemas gerenciadores de bases de dados; filtros para ocultar temporariamente subconjuntos de exemplos e atributos de um conjunto de dados; estatísticas e correlações e métodos de re-amostragem de dados, entre outros.

A integração do CO-TRAINING no projeto DISCOVER, por meio de uso da DOL, representa um ganho importante tanto na implementação do CO-TRAINING quanto na implementação de qualquer outro sistema que faça uso dos formatos padrões e das facilidade já implementadas no DISCOVER. Isso deve-se ao fato dessas bibliotecas estarem bem implementadas, além de já terem sido testadas e validadas pelos integrantes do nosso grupo de pesquisa e usuários externos.

Para utilizar a DOL é necessário que o conjunto de exemplos, no formato atributo-valor, obedeça a sintaxe padrão do sistema DISCOVER, a qual é denominada *Discover Standard Sintax* (DSX). Durante a execução, a DOL converte o conjunto desses exemplos, os quais encontram-se armazenados em arquivos, para a sintaxe padrão do algoritmo de aprendizado indicado pelo usuário. Esse algoritmo de aprendizado deve ser um dos conhecidos atualmente pelo DISCOVER, caso contrário ele deve ser previa-

[1] `http://labic.icmc.usp.br`

[2] KDD– Knowledge Discovery in Databases.

mente identificado e incluído na lista de algoritmos conhecidos pelo sistema. A DOL é responsável pela execução do algoritmo de aprendizado e pelo retorno dos resultados obtidos, além de calcular várias estatísticas. Portanto, todos os conjuntos de exemplos utilizados pelo CO-TRAINING devem estar na DSX, assim como os conjuntos de exemplos rotulados pelo algoritmo. A descrição da sintaxe padrão DSX encontra-se no Apendice A.

Além da implementação do algoritmo CO-TRAINING como parte integrante do DISCOVER, é importante pesquisar o comportamento do algoritmo levando em conta os possíveis ajustes de diversos parâmetros implementados para a sua execução. Assim, na implementação realizada, o CO-TRAINING pode ser executado em dois modos:

1. modo simulação
2. modo real

A principal diferença entre esses dois modos é o conjunto de exemplos não-rotulados $U$. No modo de simulação, esse conjunto é construído artificialmente a partir de um conjunto de exemplos rotulados. Dessa maneira, é possível verificar a cada iteração do algoritmo o rótulo atribuído por CO-TRAINING ao exemplos "melhor" rotulados coincide com o rótulo verdadeiro desses exemplos. No modo real, como o nome indica, não é conhecido o rótulo do conjunto de exemplos $U$.

A implementação realizada permite reconhecer o modo de execução automaticamente verificando, entre outras, se existem exemplos com o atributo classe igual a "?", que indica exemplos não rotulados. Quando são encontrados exemplos não rotulados, o programa ativa automaticamente o modo de execução real. Se não for encontrado nenhum exemplo com o atributo classe igual a "?" é ativado o modo de simulação.

## 3.1 Modo Simulação

Como mencionado, o modo simulação foi implementado para facilitar a avaliação do algoritmo de CO-TRAINING, permitindo verificar, passo a passo, o seu comportamento. Neste modo, o conjunto de exemplos não rotulados $U$ é criado a partir de um conjunto de exemplos rotulados ignorando o atributo classe. Dessa maneira, é possível verificar quais e quantos exemplos o algoritmo está rotulando "errado", em que iteração isso está ocorrendo e várias outras informações.

Na implementação realizada, antes da execução do CO-TRAINING, existe uma etapa prévia que realiza algumas verificações nos conjuntos de exemplos fornecidos pelo

usuário. Essa etapa consiste na verificação dos arquivos estarem nos seus devidos diretórios, se o nomes dos diretórios e arquivos estão corretos, se os exemplos e rótulos estão em suas devidas posições em todas as bases e, no fim essa etapa as bases de exemplos rotulados $L$, não rotulados $U$ e a base de teste são criadas em memória. Após essa etapa, a execução do algoritmo CO-TRAINING começa.

Nas próximas seções é explicada essa fase prévia de preparação dos dados e, em seguida, são descritos os parâmetros de execução e os arquivos de saída gerados pela implementação realizada.

### 3.1.1 Construção da Estrutura de Diretórios

A entrada para a execução do algoritmo é o nome de um diretório que contém as bases de dados necessárias. A implementação percorre esse diretório procurando pelas bases de dados. Assim, esse diretório deve ser definido seguindo alguns padrões para que o algoritmo possa executar corretamente.

No modo simulação, o usuário deve fornecer como entrada o nome do diretório que contém, pelo menos, dois subdiretórios cujos nomes especificam o algoritmo de aprendizado a ser utilizado no conjunto de exemplos (visão) nele definidos. Cada um desses subdiretórios deve conter uma das visões do conjunto de exemplos, na sintaxe DSX. A sintaxe da linguagem que deve ser utilizada para nomear esses diretórios e arquivos é mostrada na Figura 3, onde `<identificador>` é qualquer seqüência de letras, números ou *underscores* (_) e `<algoritmos>` é o nome do algoritmo de Aprendizado de Máquina a ser utilizado por CO-TRAINING para rotular exemplos da visão `d<número>` correspondente. A atual implementação utiliza o algoritmo *Naive Bayes* (`nb`), e futuramente serão considerados os seguintes algoritmos:

$C4.5$ : indução de árvores de decisão (Quinlan, 1988);

$C4.5$rules : indução de regras (Quinlan, 1988);

***svm*** : Support Vector Machine (Boser et al., 1992).

Por exemplo, suponha que se deseja executar uma simulação com uma descrição denominada `oneGram` e a outra `twoGram`. Então, ambos conjuntos de exemplos devem estar definidos nos respectivos arquivos `.names` e `.data` na sintaxe DSX. Sejam esses arquivos `oneGram.names` e `oneGram.data` para a primeira visão, `twoGram.names` e `twoGram.data` para a segunda visão. Considere que na primeira visão é desejado executar o algoritmo *Naive Bayes* o qual é representado pelo identificador `nb` e na

```
S ::= <diretorio>/d<numero>.<algoritmo>/<arquivos-dsx>

<diretorio> ::= <identificador>

<algoritmo> ::= nb|c4.5|c4.5rules|svm

<arquivo-dsx> ::= <identificador>.data
                 |<identificador>.names
                 |<identificador>.test
```

Figura 3: Sintaxe da linguagem para definir os nomes dos diretórios e arquivos

```
baseLNAI/
        d1.nb/
              oneGram.data
              oneGram.names
        d2.c4.5/
              twoGram.data
              twoGram.names
```

Figura 4: Exemplo de estrutura de diretórios de entrada

segunda visão é desejado executar o algoritmo $\mathcal{C}4.5$ representado pelo identificador `c4.5`. Assim, para criar os diretórios para executar CO-TRAINING com esses conjuntos de dados e algoritmos, deve ser criada pelo usuário a estrutura de diretórios e arquivos descrita na Figura 4.

Nessa figura, o diretório `baseLNAI` contém dois subdiretórios `d1.nb` e `d2.c4.5`. Esses dois subdiretórios indicam que os arquivos neles contidos representam, respectivamente, as duas descrições (visões) do conjunto de exemplos a ser utilizado por CO-TRAINING e também indicam que o algoritmo *Naive Bayes* deve ser executado na primeira visão e $\mathcal{C}4.5$ na segunda visão do conjunto de dados. Ou seja, nesses dois subdiretórios `d1.nb` e `d2.c4.5`, devem-se encontrar os respectivos arquivos `.data` e `.names` que definem, na sintaxe DSX, o conjunto de exemplos para cada descrição. No exemplo na Figura 4, esses arquivos são: `oneGram.data` e `oneGram.names` para o conjunto de exemplos da descrição 1 e `twoGram.data` e `twoGram.names` para o conjunto de exemplos da descrição 2.

Deve ser observado que o algoritmo necessita de, no mínimo, duas descrições do conjunto de exemplos, mas poderia utilizar mais de duas descrições. A implementação realizada suporta mais de duas descrições. Nesse caso, o usuário deve definir os diretórios que contém cada uma dessas descrições. Supondo que o usuário quer executar CO-TRAINING utilizando 4 (quatro) descrições, então deve definir os diretórios correspondentes, por exemplo `d3.nb`, `d4.c4.5rules`, e os respectivos arquivos em cada diretório.

### 3.1.2 Sincronização das Bases

Após verificar que a estrutura de diretório e os arquivos de dados fornecidos pelo usuário estão corretamente definidos, o programa aloca em memória as bases, encontradas na fase anterior, e depois verifica o que denominamos de "sincronização" dos arquivos de dados. Nesses arquivos, que possuem extensão `.names`, encontram-se armazenados os valores dos atributos dos conjuntos de exemplos de cada descrição— Apêndice A na página 29. Para isso, é mandatório que os valores do primeiro atributo definam uma identificação (*id*) única para cada exemplo nas diferentes visões. Utilizando esse identificador são verificados os seguintes três tipos de sincronia:

1. sincronia de existência: para cada identificador *id* de exemplo, é verificada a existência desse *id* em todas as outras descrições. Caso o *id* não seja encontrado em todas as descrições, o(s) exemplo(s) é(são) descartado(s).

2. sincronia de posição: para cada identificador *id* do exemplo na posição $i$ da tabela atributo-valor, é verificado se a posição desse exemplo é a mesma em todas as tabelas das outras descrições. Caso isso não aconteça, as tabelas de todas as descrições (arquivos `.data`) são reconstruídas tal que, no fim da execução de sincronia de posição, os exemplos com o mesmo *id*, encontram-se na mesma posição em todas as tabelas.

3. sincronia de classe: para cada identificador *id* de um exemplo com classe $y$, é verificado que o exemplo correspondente, ou seja, com o mesmo *id*, em todas as outras descrições, possui a mesma classe associada $y$. Se isso não acontecer, todos os exemplos com esse *id* são descartados.

Caso os dados de entrada não verifiquem as condições requeridas, o usuário é informado sobre a natureza do erro e a execução para. Além disso, no caso do problema detectado ser de sincronia, os arquivos originais com extensão `.data` são renomeados com extensão `.data.old` e são criados os respectivos novos arquivos com extensão `.data` contendo os exemplos devidamente sincronizados. Assim, o usuário tem condições de verificar as correções realizadas pelo programa nos conjuntos de dados originais.

### 3.1.3 Alocação em Memória dos Conjuntos de Exemplos

Com os arquivos que compõem as bases devidamente verificados, essas bases podem ser manipuladas para a construção dos conjuntos de exemplos requeridos por CO-TRAINING — Algoritmo 1 na página 7. Inicialmente é criada ou carregada em memória

a base de teste. O programa procura por essa base de teste, ou seja, verifica se existe um arquivo com extensão `.test`, definido pelo usuário, no subdiretório correspondente. Caso essa base de teste não seja encontrada, ela é criada a partir de uma porcentagem do conjunto de exemplos original. Essa porcentagem pode ser especificada pelo usuário. Após a base de teste devidamente alocada em memória, o programa cria as bases de exemplos rotulados $L$ e não rotulados $U$. Nessa fase, são construídas as bases $L$ e $U$ somente em memória, ou seja, não são criados os arquivos `.test`, `.data` e `.names`. Somente no final da execução o programa grava todas as bases em memória. A seguir é apresentado uma descrição de cada uma dessas bases.

**base de teste:** a base que contém os exemplos de teste pode ser criada de duas maneiras: por meio de um arquivo `.test` ou por uma porcentagem do conjunto de exemplos original, como explicado a seguir.

1. *arquivo de teste*: o usuário pode fornecer uma base de teste que deverá ter a extensão `.test`. Os exemplos contidos nesse arquivo devem estar no mesmo formato dos exemplos do arquivo `.data`.
2. *uma porcentagem do conjunto de exemplos original*: caso não seja fornecida uma base de teste, no modo simulação, o programa detecta a falta dessa base e cria uma nova base de teste a partir de uma porcentagem do conjunto de exemplos original. Essa porcentagem pode ser definida pelo parâmetro `-t porcentagem`. Para exemplificar, suponha que o CO-TRAINING seja executado com uma base de 1000 exemplos invocando a seguinte linha de execução:

```
cotraining.pl -t 0.3 baseLNAI
```

   o parâmetro `-t` definido como `0.3` indica 30% da base original. Portanto, a base de exemplos de teste será composta de 30% dos 1000 exemplos, ou seja 300 exemplos. Os 700 exemplos restantes irão compor o que denominamos base parcial. Valor padrão : 0.25;

**base de exemplos rotulados $L$ e não rotulados $U$:** no modo simulação, essas bases podem ser criadas de duas maneiras: indicando uma porcentagem da base parcial [3], ou fornecendo uma lista de exemplos. Essas duas maneiras são definidas pelos seguintes parâmetros:

`-l <porcentagem>`: define uma porcentagem sobre a base parcial, construída na criação da base de teste, que irá compor a base de exemplos rotulados

[3] A base parcial e a base de dados original coincidem caso o usuário fornecer a base de teste. Caso contrário, a base de dados obtidas após retirar o subconjunto de exemplos de teste da base de dados original

$L$. Por exemplo, se o CO-TRAINING for executado em uma base com 100 exemplos invocando a seguinte linha de execução:

```
cotraining.pl -l 0.3 baseLNAI
```

na qual o parâmetro `-l` é definido como `0.3` então $L$ consite de 30% dos exemplos da base parcial.

A base de teste sempre é alocada antes das bases de exemplos rotulados e não rotulados. Assim, no caso do usuário utilizar o valor padrão para `-t` e uma base original com 1000 exemplos, inicialmente é construída uma base de teste com 25% (parâmetro padrão) de 1000 exemplos, ou seja, uma base de teste com 250 exemplos e a base parcial com 750 exemplos. Dessa base parcial são retirados 30% dos 750 exemplos, ou seja, 225 exemplos para comporem a base de exemplos rotulados $L$ e os restantes, 525 (750-225 = 525) exemplos, constituem a base de exemplos não rotulados $U$. Resumidamente, utilizando uma base original com 1000 exemplos, como o valor padrão para o parametro `-t` e `-l` 0.3, os exemplos da base original serão distribuídos da seguinte maneira:

- 250 exemplos para a base de teste (25% de 1000);
- 225 exemplos para a base de exemplos rotulados (30% de 750);
- 525 exemplos para a base de exemplos não rotulados (70% de 750).

Valor padrão: 0.05;

`--seeds <lista de exemplos>`: O parâmetro anterior, `-l`, retira exemplos aleatoriamente da base parcial para criar a base de exemplos rotulados $L$. No entanto, em algumas situações o usuário pode desejar escolher os exemplos que deseja inserir em $L$ especificando-os um a um. Com o parâmetro `--seeds` o usuário pode especificar esses exemplos. Assim, caso o usuário escolha o primeiro, o terceiro e o quinto exemplo da base parcial para comporem o conjunto de exemplos rotulados $L$, o CO-TRAINING deve ser invocado com a seguinte linha de execução:

```
cotraining.pl --seeds "0,2,4" baseLNAI
```

Cada elemento dessa lista de exemplos indica a posição do exemplo no arquivo `.data`. Portanto, a lista `"0,2,4"` representa a primeira, terceira e a quinta linha do arquivo `.data`. Vale ressaltar que essa linha está relacionada a posição do exemplo no arquivo e não com o valor do *id* explicado na Seção 3.1.2 para descrever os passos da fase de sincronização dos arquivos. Note que os índices começam em zero, assim, para o milésimo exemplo deve

ser utilizado o número 999. Valor padrão: utiliza o parâmetro `-l` com valor 0.02;

Com os conjuntos de exemplos construídos a etapa preparatória para a execução do algoritmo termina, e o programa está pronto para executar o algoritmo de CO-TRAINING propriamente dito. Vários outros parâmetros podem ser utilizados na execução de CO-TRAINING, os quais são descritos a seguir.

### 3.1.4 Descrição dos Parâmetros

Além dos parâmetros para a construção dos conjuntos de exemplos, `-l`, `-t` e `--seeds`, existem vários outros parâmetros que podem ser utilizados para realizar experimentos e analisar o comportamento de CO-TRAINING. A combinação desses parâmetros pode mudar consideravelmente o comportamento do algoritmo, o que fornece ao usuário um número expressivo de ajustes possíveis. Esses parâmetros são descritos a seguir:

`-i <número>`: define o número de iterações que o algoritmo deve executar.Valor padrão : 100;

`--at-least <porcentagem>`: para CO-TRAINING funcionar, os classificadores, além de rotular os exemplos, devem fornecer um valor de confiança para cada exemplo rotulado. O parâmetro `--at-least` permite definir um valor mínimo aceitável (*lower bound*) para esses valores de confiança. Dessa maneira, todos os exemplos rotulados que possuírem o valor de confiança menor que o definido por esse parâmetro são descartados. Valor Padrão : 0.6;

`--lsm <tipo>`: *labeled selection method*, seleciona a função *melhores*/2 a ser utilizado. Como mencionado na descrição de CO-TRAINING — Algoritmo 1 na página 7 — a função *melhores*/2, responsável pela escolha dos "melhores" exemplos rotulados em cada iteração, é muito importante, e pode ser descrita de várias maneiras. Até agora, foram definidas duas funções *melhores*/2, `nbest` e `original`, mas outras funções podem ser facilmente implementadas e adicionadas ao programa. Em ambos os casos, a função *melhores*/2 ignora exemplos que foram rotulados diferentemente pelos classificadores $h_{D_1}$ e $h_{D_2}$ [4]. Após, cada uma das funções implementadas, `nbest` e `original`, utilizam critérios diferentes, como explicado a seguir:

[4]Por simplicidade, são considerados somente duas descrições no algoritmo apresentado, mas a implementação realizada permite utilizar mais de duas

1. `nbest <n>`: utilizando o valor de confiança dado por cada classificador $h_{D_1}$ e $h_{D_2}$ para cada par de exemplos correspondentes (mesmo *id* em ambas descrições) classificados com o mesmo rótulo, é calculado o que denominamos *ranking* dos exemplos, como o produto dos respectivos valores de confiança. Utilizando o valor desse *ranking* os exemplos são ordenados em forma decrescente e *melhores*/2 pega os `<n>` primeiros pares de exemplos desse *ranking* para construir $(R_{D_1}, R_{D_2})$.

2. `original`: neste caso, após calculardo o *ranking* de maneira semelhante ao caso anterior, os exemplos são seperados e ordenados por classe. Após a função seleciona os melhores exemplos de cada classe. Para definir o número de exemplos a serem inseridos por esse método um dos seguintes parâmetros devem ser inicializados:

   `-a <número>`: este parâmetro indica que devem ser escolhida a mesma quantidade de exemplo de cada classe. Por exemplo, considerando uma base de exemplos com duas classes POSITIVO e NEGATIVO, caso ¡número¿ for definido como 10, então, a função escolhe, sempre que possível, os 10 melhores exemplos de cada classe (máximo 2x10 = 20) para compor $R_{D_1}$ e $R_{D_2}$. Valor padrão: 2;

   `-c <lista de classes>`: esse parâmetro indica que dever ser escolhido uma quantidade de exemplos diferente de cada classe. Considerando o exemplo anterior, para inserir 2 exemplos da classe POSITIVO e 6 da classe NEGATIVO, basta invocar a seguinte linha de comando

   ```
   cotraining.pl -c "positivo=2,negativo=6" baseLNAI
   ```

   Neste caso, em cada iteração do algoritmo poderão ser rotulado no máximo 8 exemplos.

   Esses parâmetros, `-a` e `-c`, apenas definem quantos exemplos de cada classe o usuário deseja que o algoritmo rotule em cada iteração. Desse maneira, o algoritmo tenta conseguir esse número de exemplos mas isso nem sempre é possível e portanto é necessário utilizar o parâmetro `--force` explicado mais adiante. Valor padrão: 2 para cada classe;

Esse dois parâmetros, `-a` e `-c`, não devem ser confundidos com os parâmetros de construção dos conjuntos $L$ e $U$. Assim, vale ressaltar que ao definir esses dois parâmetros, esses valores não definem o número de exemplos no conjunto de exemplos $L$ inicial, mas definem o número máximo de exemplos a ser rotulados e inseridos em $L$ em cada iteração do algoritmo.

Valor Padrão : `original`;

`--force`: Este parâmetro força o algoritmo a completar os $n$ exemplos antes de iniciar a próxima iteração. A função *melhores*/2 nem sempre consegue obter `<n>` "melhores" exemplos do conjunto de exemplos rotulados (ver parâmetro `--lsm`). Se isso acontecer e o parâmetro `--force` estiver ativo, nenhum exemplo é rotulado, `<n>` exemplos não rotulados são adicionados a $U'_{D_1}$ e $U'_{D_2}$ respectivamente e o processo é repetido. Em outras palavras, são incrementados os conjuntos $U'_{D_1}$ e $U'_{D_2}$ a serem rotulados, até *melhores*/2 conseguir obter `<n>` exemplos rotulados. Se a situação persistir, o algoritmo tenta indefinidamente até conseguir ou o algoritmo parar. Valor padrão: desabilitado;

`-v`: *verbose*, com esse parâmetro ativado, o algoritmo irá mostrar o valor de confiança da classificação de todos os exemplos de $U'$. Valor padrão: desabilitado;

### 3.1.5 Arquivos de Saída

Como explicado anteriormente, caso os dados de entrada verifiquem as condições requeridas, o algoritmo CO-TRAINING é executado. Utilizando os diversos parâmetros que podem ser especificados pelo usuário, descritos na Seção 3.1.4 na página 15. Na forma mais simples, executando CO-TRAINING com os parâmetros padrões, basta invocar a seguinte linha de comando, com a base `baseLNAI` da Figura 4 na página 11.

```
cotraining.pl baseLNAI
```

No final da execução, os resultados são gravados em vários arquivos ilustrados na Figura 5. Como pode-se notar, a saída de CO-TRAINING consiste de 8 (oito) arquivos para cada visão e 3 (três) arquivos que contém informações detalhadas das execuções. Esses arquivos são descritos em maiores detalhes a seguir.

`cotraining.log`: o arquivo de log contém todas as informações que são mostradas na tela. Essas informações mostram ao usuário os resultados da execução de cada iteração do algoritmo bem como os erros dos classificadores gerados, entre várias outras informações.

`gnuplot.dat`: após cada iteração, os classificadores gerados são testados sobre o conjunto de teste. No arquivo `gnuplot.dat` são mostrados os erros de cada classificador nesse conjunto de teste.

`fullgnuplot.dat`: fornece os erros sobre o conjunto de teste, os erros sobre o conjunto de exemplos temporários $R'_{D_1}$ e $R'_{D_2}$ e o número de exemplos que foram rotulados

```
baseLNAI/
        d1.nb/
              oneGram.data
              oneGram.names
        d2.c4.5/
              twoGram.data
              twoGram.names

        cotraining.log
        gnuplot.dat
        fullgnuplot.dat
        rdata/

            d1Labeled.data
            d1Labeled.names
            d1Unlabeled.data
            d1Unlabeled.names
            d1Test.data
            d1Test.names
            d1Uaux.data
            d1Uaux.names

            d2Labeled.data
            d2Labeled.names
            d2Unlabeled.data
            d2Unlabeled.names
            d2Test.data
            d2Test.names
            d2Uaux.data
            d2Uaux.names
```

Figura 5: Estrutura de diretório após a execução do CO-TRAINING

errados e foram inseridos em $L_{D_1}$ e $L_{D_2}$. Esta informação é fornecida somente no caso de execução no modo simulação, no qual é conhecido o rótulo de todos os exemplos.

**diretório `rdata`:** o algoritmo pode ter a sua execução interrompida por várias razões: por um erro; por ter acabado o número de exemplos não rotulados; por ter atingido o número de iterações definidas pelo usuário; entre outros. Após parar, o CO-TRAINING salva nesse diretório todas as bases em memória. Assim, para cada visão ele cria 8 arquivos, que correspondem a 4 bases, a primeira base para os exemplos rotulados, a segunda para os exemplos não rotulados, a terceira para os exemplos de teste e finalmente a quarta base que correspondem a $U'_{D_1}$ e $U'_{D_2}$ ilustrados no Algoritmo 1 na página 7.

## 3.2 Modo Real

O modo real foi construído para executar bases do mundo real nas quais tem-se uma grande quantidade de exemplos não-rotulados e uma pequena quantidade de exemplos rotulados. Como já dito anteriormente, o modo real é ativado automaticamente pelo programa, caso seja encontrado o algum valor do atributo classe dos exemplos igual a "?".

Assim como no modo de simulação, no modo real existem as mesmas etapas de preparação para a execução do algoritmo, porém com algumas pequenas diferenças. Ainda que a construção de diretórios e a sincronização das bases é exatamente a mesma do modo simulação, bem como a alocação em memória dos conjuntos de exemplos de teste, a construção da base de exemplos rotulados $L$ e a base de exemplos não-rotulados $U$ é diferente, como explicado a seguir.

No modo real os arquivos `.data`, podem conter o símbolo "?" com valor para o atributo classe, o qual indica exemplos não rotulados. Ao encontrar exemplos com "?" o valor do atributo classe igual a "?", esses exemplos são colocados no conjunto de exemplos não rotulados $U$ e todos os exemplos diferentes de "?" são colocado no conjunto de exemplos rotulados $L$. É dessa maneira que são construídos esses dois conjuntos de exemplos.

Na Tabela 2 é ilustrada uma base com essas características. Nessa tabela os exemplos $E_1$, $E_5$, $E_7$ e $E_9$ irão compor o conjunto de exemplos rotulados $L$ e os exemplos $E_2$, $E_3$, $E_4$, $E_6$ e $E_8$ irão compor o conjunto de exemplos não rotulados $U$. Como as bases de exemplos rotulados e não rotulados são construídas, respectivamente, baseadas nos exemplos que tem ou não a classe "?" os parâmetros para a construção da base de

exemplos definidos no modo simulação não tem significado neste modo de execução, *i.e.*, o parâmetro `-l` que define a porcentagem de exemplos rotulados, o parâmetro `--seeds` que define quais são os exemplos que irão compor inicialmente a base de exemplos rotulados $L$. Todos os outros parâmetros podem ser também utilizados no modo real.

| | $X_1$ | $X_2$ | ... | $X_M$ | $Y$ |
|---|---|---|---|---|---|
| $E_1$ | $x_{11}$ | $x_{12}$ | ... | $x_{1M}$ | $y_1$ |
| $E_2$ | $x_{21}$ | $x_{22}$ | ... | $x_{2M}$ | ? |
| $E_3$ | $x_{31}$ | $x_{32}$ | ... | $x_{3M}$ | ? |
| $E_4$ | $x_{41}$ | $x_{42}$ | ... | $x_{4M}$ | ? |
| $E_5$ | $x_{51}$ | $x_{52}$ | ... | $x_{5M}$ | $y_5$ |
| $E_6$ | $x_{61}$ | $x_{62}$ | ... | $x_{6M}$ | ? |
| $E_7$ | $x_{71}$ | $x_{72}$ | ... | $x_{7M}$ | $y_7$ |
| $E_8$ | $x_{81}$ | $x_{82}$ | ... | $x_{8M}$ | ? |
| $E_9$ | $x_{91}$ | $x_{92}$ | ... | $x_{9M}$ | $y_9$ |

Tabela 2: Exemplos rotulados e não-rotulados na mesma base

```
1 ======================================
2 PARAMETERS
3 Number of Iterations        : 100
4 Percent Test                : 0.25000
5 Percent Labeled             : 0.05000
6 Force Completed             : disabled
7 Selected must have at least : 0.6
8 Labeled Selection Method    : original
9 ======================================
```

Figura 6: Parâmetros de execução - arquivo `cotraining.log`

# 4 Exemplo de Execução

Para ilustrar a execução da implementação realizada de CO-TRAINING, nas próximas seções é descrita uma execução no modo simulação, e os gráficos que podem ser gerados a partir dos arquivos criados durante a execução. A base de dados utilizada consiste de exemplos rotulados com duas classes diferentes, `ilp` (*Case Base Reasoning*) e `cbr` (*Inductive Logic Programming*). O programa foi executado com todos os parâmetros padrões invocando a seguinte linha de execução:

```
cotraining.pl baseLNAI
```

## 4.1 Um Exemplo de Execução Passo a Passo

Inicialmente são verificados os parâmetros de execução (linhas 1 a 9 da Figura 6). Como CO-TRAINING é executado com todos os parâmetros padrões, então, são executados no máximo 100 iterações do algoritmo, a base de teste deve ser construída com 25% dos exemplos da base original e 5% dos exemplos da base parcial constituem a base de conjunto de exemplos rotulados $L$. Vale ressaltar que caso sejam encontrados os arquivos `.test` no diretório `baseLNAI`, a base de teste seria o conjunto de exemplos nesses arquivos. Também, caso sejam encontrados exemplos rotulados com valor da classe igual a "?", o modo de execução trocaria para o modo real e a base de exemplos rotulados seria criada com os exemplos que possuem o valor do atributo classe diferente de "?".

Após a leitura e instanciação dos parâmetros passados na linha de comando, são executadas as fases preparatórias descritas na Seção 3.1.2 na página 12 para a execução do algoritmo. O usuário é informado do resultado da execução dessas fases, e essa informação é também armazenda no arquivo `cotraining.log`. As ações realizadas em cada fase são descritas a seguir:

- verificação da estrutura de diretórios (linhas 10 a 12 da Figura 7): o programa percorre todos os diretórios dentro de `baseLNAI` a procura das descrições e nas respectivas bases de exemplos.

- leitura e sincronização das bases (linhas 13 a 23 da Figura 7): conhecendo a localização das bases, o programa aloca as mesmas em memória e mostra o número de exemplos e o número de atributos dessas bases. Após isso, são feitas as verificações de sincronização — Figura 7 linha 22.

- construção dos conjuntos de exemplos (linhas 24 a 36 da Figura 7): com as bases devidamente alocadas em memória é possível saber se as bases correspondem ao modo simulação ou ao modo real. No exemplo, é considerado o modo de simulação. As linhas 24 a 27 correspondem ao número de exemplos de cada classe que será inserido em $L$ durante a execução do algoritmo (ver os parâmetros `-c` e `-a` descritos na Seção 3.1.4). Entre as linhas 28 a 31 são mostradas as mensagens relacionadas a construção das bases de teste. Note que o programa procura primeiramente pela base de teste e, como ela não foi encontrada, ela é criada a partir do conjunto de dados original. Retirados os exemplos do conjunto de teste do conjunto de dados original, são criadas as bases de exemplos rotulados $L$ e não rotulados $U$ (linhas 33 a 35). O conjunto de exemplos rotulados inicial é essencial para a execução do algoritmo, portanto, diferentemente dos outros conjuntos ele sempre é apresentado em detalhes. Entre as linhas 33 a 35 são ilustradas os $IDs$ dos exemplos que compõem o conjunto $L$ inicial, separados por classes; na linha 34 são mostrados os exemplos que compõem o conjunto $L$ e que possuem a classe `cbr`. A mensagem `(R 10/W 0)` na mesma linha informa a existência de 10 exemplos rotulados corretamente (`R` = *Right*) e 0 exemplo errado (`W` = *Wrong*) da classe `cbr`. Do mesmo modo, na linha 35 são mostrados os exemplos que compõem o conjunto $L$ com a classe `ilp`. Nas linhas seguintes, 36 e 37, é informada o número de exemplos que compõem os conjuntos $L$ e $U$ respectivamente.

Após a etapa preparatória, os dados estão prontos para serem utilizados pelo algoritmo de CO-TRAINING. A Figura 8 na página 24 ilustra a primeira iteração da execução.

O conjunto $L$ é uma das principais saídas do algoritmo, portanto, a cada iteração são apresentados os *ids* dos exemplos que o compõem, bem como informação sobre o número de exemplos correta e incorretamente rotulados que são inseridos nesse conjunto. Logo em seguida são induzidos os classificadores. Com eles, é possível rotular os exemplos contidos em `Uaux` ($U' = U'_{D_1} \cup U'_{D_2}$). Com o `Uaux` rotulado, ele é submetido a função *melhores*/2 escolhida pelo usuário, a qual ignora os exemplos que não

```
Checking directory structure
2 visions found
Directory structure OK!
Reading datasets
Datasets read
Filename              : base/d1.nb/oneGramMin2
Number of instances   : 395
Number of features    : 2916
Filename              : base/d2.nb/twoGramMin2
Number of instances   : 395
Number of features    : 3247
Datasets synchronized
Classes loaded
============== SIMULATION DATASET MODULE ACTIVATED =============
In each iteration will be inserted at most...
   cbr 2
   ilp 2
Number of instances to be inserted by iteration : 4
============================ Test set ============================
No test datasets found
Test set created!
Test set size                              : 98
================== Labeled and Unlabeled Sets ==================
Instances in Labeled set
(R    10 / W     0)  cbr   :99,131,146,153,169,170,192,202,231,273
(R     4 / W     0)  ilp   :322,325,343,379
Labeled set size                           : 14
Unlabeled set size                         : 283
```

Figura 7: Preparação dos dados - arquivo `cotraining.log`

```
38 ========================= Iteration 1 ========================
39 Instances in Labeled set
40 (R    10 / W     0)  cbr   :99,131,146,153,169,170,192,202,231,273
41 (R     4 / W     0)  ilp   :322,325,343,379
42 Labeled set size                          : 14
43 Unlabeled set size                        : 255
44 Test set size                             : 98
45 Uaux Set size                             : 28
46 Save Labeled
47 Induce Classifiers
48 Induce Classifiers
49 Combined Classifier enabled
50 Join Visions
51 Number of instances with same class       : 24
52 Number of instances with different class : 4
53 class cbr : 21
54 class ilp : 3
55 Selected Instances
56    instance  137       1.00 ( cbr  cbr )
57    instance  124       1.00 ( cbr  cbr )
58    instance  336       1.00 ( ilp  ilp )
59    instance  381       0.97 ( ilp  ilp )
60 (1) Uaux test set error                   : 14.29
61 (2) Uaux test set error                   : 21.43
62 (1) Test Dataset error                    : 13.27
63       cbr : 8.696
64       ilp : 24.138
65 (2) Test Dataset error                    : 12.24
66       cbr : 1.449
67       ilp : 37.931
68 (combined) Test Dataset error             : 12.24
69       cbr : 7.246
70       ilp : 24.138
71 Wrong selected until now                  : 0
```

Figura 8: Exemplo de iteração - arquivo `cotraining.log`

possuem a mesma classe entre as descrições; nesta primeira iteração são 4 exemplos (linha 52). Dos 24 exemplos restantes (linha 51), 21 são da classe `ilp` e 3 são da classe `cbr`. Como a função *melhores*/2 utilizado na execução é a definida pelo parâmetro `original`, então são escolhidos os dois melhores exemplos de cada classe. Os exemplos escolhidos são mostrados entre as linhas 55 e 59.

Note que foram escolhidas os exemplos 137, 124, 336 e 381 os dois primeiros da classe `ilp` e os dois últimos da classe `cbr`. Esses números indicam as posições desses exemplos nos respectivos arquivos `.data`. Ou seja, o exemplo 137 está na 138º linha do arquivo `.data`, o exemplo 124 está na 125º linha do arquivo `.data`, e assim por diante. Os valores da terceira coluna (1, 1, 1 e 0.97) representam o *ranking* dos melhores exemplos (ver Seção 3.1.4 na página 15).

Entre as linhas 60 e 70 são informadas os erros dos classificadores sobre os dois conjun-

tos: `Uaux` ($U'$) e Teste. Os números entre parênteses indicam o classificador utilizado para estimar os erros. Por exemplo, na linha 60 a mensagem "`(1) Uaux test set error` " informa o erro do classificador 1 ($h_{D_1}$) sobre o conjunto `Uaux`. De modo semelhante na linha 61 é apresentado o erro de $h_{D_2}$ tanto $h_{D_1}$ quanto $h_{D_2}$ referem-se, ao classificador induzido pelo *Naive Bayes*. O erro do conjunto de teste, no modo de simulação, é uma das informações mais importantes para avaliar o algoritmo de CO-TRAINING. Assim, o erro do classificador sobre o conjunto de teste é apresentado, além do o erro do classificador para cada classe. Por exemplo, na linha 62 é apresentado o erro do classificador 1 ($h_{D1}$) que teve 13,27% de erro. Nas duas linhas seguintes são apresentados os erros de cada classe. O classificador 1 para a classe `cbr` obteve 8,689% de erro e a para a classe `ilp` obteve 24.931% de erro. Os erros entre as linhas 68 a 70 representam os erros do classificador combinado, explicado anteriormente na Seção 2 na página 3.

Vale ressaltar que no modo simulação é conhecido o verdadeiro rótulo dos exemplos. Assim, é possível computar o número de exemplos rotulados errados. A última linha da Figura 8 mostra quantos exemplos selecionados foram rotulados errados e inseridos em $L$. Nessa iteração foram selecionados 4 exemplos (137, 124, 336 e 381) dos quais nenhum deles foi rotulado errado.

As saídas geradas quando o programa é executado no modo real são semelhantes ao do modo simulação, diferindo apenas na linha 23 e entre as linhas 60 a 71 da Figura 8 as quais não existem pois no modo real não é conhecida a classe correta dos exemplos.

## 4.2 Gráficos que Podem ser Gerados Pelo Sistema

Com as informações contidas nos arquivos `gnuplot.dat` e `fullgnuplot.dat`, é possível gerar gráficos que mostram os erros dos classificadores. Na Figura 9 é mostrado um exemplo do arquivo `gnuplot.dat`. A primeira linha mostra respectivamente, a iteração, o erro do classificador 1 ($h_{D_1}$), o erro do classificador 2 ($h_{D_2}$) e o erro do classificador combinado. Esses erros são calculados utilizando a base de teste. A implementação realizada constrói um arquivo `gnuplot.script` que gera o gráfico ilustrado na Figura 10. Vale ressaltar que, por ser um arquivo texto, esse arquivo pode ser utilizado com qualquer outra ferramenta para gerar os gráficos.

O arquivo `fullgnuplot.dat` mostra, além dos erros dos classificadores sobre a base de teste, como no arquivo `gnuplot.dat`, os erros sobre o conjunto `Uaux`. Na Figura 11 é mostrado um exemplo no arquivo `fullgnuplot.dat`. As linhas mostram, respectivamente, o número da iteração, o erro do classificador 1 sobre a base `Uaux`, o erro do classificador 2 sobre a base `Uaux`, o erro do classificador 1 sobre a base de teste, o erro

```
1  1 13.265 12.245 12.245
2  2 13.265 10.204 9.184
3  3 11.224 9.184 9.184
4  4 9.184 9.184 8.163
5  5 9.184 9.184 7.143
6  6 8.163 12.245 8.163
7  7 9.184 10.204 8.163
8  8 9.184 9.184 8.163
9  9 8.163 9.184 7.143
10 10 8.163 8.163 7.143
```

Figura 9: Exemplo de arquivo `gnuplot.dat`

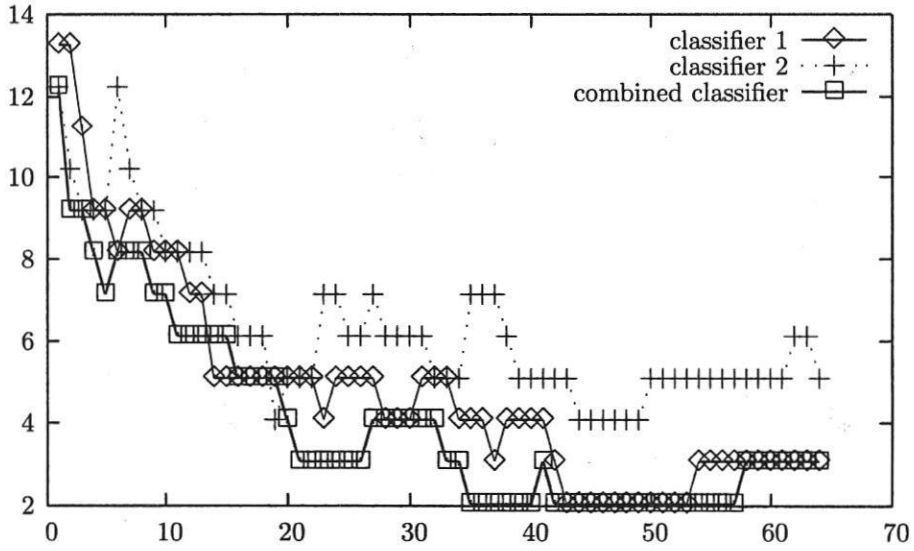



Figura 10: Exemplo de gráfico dos erros dos classificadores calculados utilizando o conjunto de teste

```
1  1 14.286 21.429 13.265 12.245 12.245 0
2  2 21.429 17.857 13.265 10.204 9.184 0
3  3 10.714 17.857 11.224 9.184 9.184 0
4  4 10.714 17.857 9.184 9.184 8.163 0
5  5 7.143 17.857 9.184 9.184 7.143 0
6  6 10.714 10.714 8.163 12.245 8.163 0
7  7 10.714 14.286 9.184 10.204 8.163 0
8  8 7.143 10.714 9.184 9.184 8.163 0
9  9 7.143 10.714 8.163 9.184 7.143 0
10 10 6.667 13.333 8.163 8.163 7.143 0
```

Figura 11: Exemplo de arquivo `fullgnuplot.dat`

do classificador 2 sobre a base de teste, o erro do classificador combinado e, finalmente, o número de exemplos rotulados até aquela iteração.

# 5 Consideração Finais

Encontrar bases de dados reais com um número significativo de exemplos, que pertence a família de algoritmos de aprendizado semi-supervisionado para a indução de bons classificadores é cada vez mais difícil, quando não impossível. O algoritmo CO-TRAINING, tenta solucionar o problema rotulando exemplos a partir de poucos exemplos rotulados. O objetivo principal dessa família de algoritmo é tentar rotular, com uma certa certeza, um número maior de exemplos para, após, serem utilizados por qualquer algoritmo de aprendizado supervisionado. Para realizar essa rotulação prévia, o CO-TRAINING necessita de duas ou mais descrições dos mesmos exemplos.

Neste relatório foram descritos em detalhes: o projeto e a implementação realizada do algoritmo de CO-TRAINING, integrado DISCOVER, os dois modos de execução implementados e os diversos parâmetros que podem ser utilizados na sua execução. Foi também mostrado um exemplo de execução utilizando uma base de exemplos que estamos investigando. Os resultados obtidos com essa base de exemplos no modo simulação são muito bons (Matsubara & Monard, 2004) , e mostram que a implementação de CO-TRAINING realizada conseguiu atingir os objetivos esperados, rotulado corretamente a maioria dos exemplos.

# A A Sintaxe *Discover Dataset Sintax — DSX*

**Este Apêndice que descreve a DSX foi retirada Tese de G. E. A. P. A. Batista (Batista, 2003), mentor e um dos principais desenvolvedores do sistema Discover.**

Na etapa de Mineração de Dados do processo de *Knowledge Discovery in Databases* (KDD), muito freqüentemente são utilizados sistemas de aprendizado, tanto acadêmicos quanto comerciais, para a extração de padrões. Infelizmente, não houve uma padronização no formato do arquivo de dados utilizado como entrada para esses sistemas. Como resultado, diferentes sistemas de aprendizado utilizam diferentes sintaxes de arquivos de dados. Os sistemas de aprendizado acadêmicos normalmente aceitam como entrada somente arquivos texto em um formato proprietário. Os sistemas de aprendizado comerciais geralmente aceitam, além de arquivos texto, outras formas de entrada de dados, como, conexões nativas a bancos de dados SQL e interface ODBC™.

Realizar uma investigação que envolve extrair conhecimento de vários conjuntos de dados utilizando diversos sistemas de aprendizado é normalmente muito trabalhoso pois necessita, entre outros, converter os arquivos de dados para a sintaxe utilizada por cada sistema de aprendizado. Para simplificar esse trabalho dentro do projeto *Discover*, foi decidido adotar uma sintaxe padrão para conjuntos de dados. A partir dessa sintaxe padrão, é possível utilizar a biblioteca de classes *DOL* para converter um arquivo de dados para a sintaxe utilizada em diversos sistemas de aprendizado.

Dessa maneira, decidiu-se criar uma nova sintaxe para o ambiente *Discover*, a qual foi dado o nome de *DSX* (*Discover Dataset Sintax*), ao invés de utilizar alguma sintaxe já definida por outros pesquisadores, como as sintaxes utilizadas nos projetos *MLC++* (Kohavi, Sommerfield, & Dougherty, 1997) e *Weka* (Witten & Frank, 2000). Essa decisão é justificada uma vez que as seguintes características são desejáveis na nova sintaxe:

**Suporte a diversos tipos da dados** Apesar de que os sistemas de aprendizado mais tradicionais são limitados a utilizar dados de tipo numérico (inteiros e reais) e nominal, os sistemas de aprendizado mais recentes, sobretudo os sistemas de aprendizado comerciais, são capazes de utilizar outros tipos de dado tais como data e hora. Assim, a nova sintaxe deve dar suporte aos tipos de dado mais atuais. Para a definição desses tipos, foram levantados os tipos de dado mais utilizados em sistemas gerenciadores de banco de dados, sendo que os tipos de dado mais utilizados nesses sistemas foram incorporados à sintaxe *DSX*;

**Suporte a indução construtiva apoiada pelo usuário** É bastante comum que o usuário deseje construir novos atributos a partir de atributos já presentes nos dados (Lee, 2000). Isso ocorre, pois um atributo que compõe informações sobre dois ou mais atributos pode ser muito mais relevante para a resolução do problema do que os atributos individuais. Por exemplo, em concessão de crédito pessoal, dois atributos altamente relevantes são a renda do cliente e o valor da prestação a ser paga. Entretanto, o percentual da renda do cliente comprometido com o pagamento da prestação pode fornecer uma medida mais direta relacionado ao cliente pode ou não ser capaz de honrar esse compromisso. A sintaxe *DSX* provê uma forma muito simples de realizar indução construtiva apoiada pelo usuário, por meio da definição de atributos virtuais, os quais podem ser definidos utilizando expressões aritméticas ou lógicas envolvendo um ou mais atributos existentes nos dados;

**Suporte a diversas tarefas de aprendizado** Conjuntos de dados declarados na sintaxe *DSX* podem ser utilizados tanto em aprendizado supervisionado quanto não supervisionado. Ainda, em aprendizado supervisionado, os dados podem ser utilizados em problemas de classificação ou regressão. Portanto, a sintaxe padrão deve dar suporte a conjuntos de dados que tenham classe nominal ou numérica, ou ainda não possuem uma classe definida explicitamente.

O projeto $\mathcal{MLC}$++ utiliza uma sintaxe para conjuntos de dados muito similar a sintaxe do sistema $\mathcal{C}$4.5. O projeto *Weka* propõe uma sintaxe chamada *Attribute-Relation File Format* (ARFF). Ambas sintaxes possuem limitações quanto aos objetivos propostos anteriormente. Tanto a sintaxe ARFF quanto a sintaxe utilizada pelo sistema $\mathcal{C}$4.5 dão suporte somente aos tipos de dado numérico e nominal. Também, essas sintaxes não oferecem suporte para realizar indução construtiva apoiada pelo usuário.

Nas próximas seções é feita uma apresentação detalhada da sintaxe *DSX* e das suas principais características.

## A.1 Uma Visão Geral da Sintaxe *DSX*

A sintaxe padrão utiliza arquivos texto para declarar os atributos e seus respectivos tipos, bem como os valores que esses atributos assumem em um conjunto de dados. Os atributos são declarados em um arquivo com a extensão `.names`. Os valores que esses atributos assumem em um conjunto de dados são declarados em um outro arquivo com a extensão `.data`. Os dois arquivos devem ter o mesmo nome, se diferenciando apenas pela extensão. Opcionalmente, pode haver também arquivos de dados com as extensões `.test` com casos rotulados de teste para medir o erro de classificação,

`.validation` com casos rotulados para validação de modelos e `.cases` com casos não rotulados para serem rotulados por um classificador.

A seguir é mostrado um conjunto de exemplos na sintaxe *DSX*. O conjunto de exemplos utilizado é o conjunto de dados artificial **voyage** (Quinlan, 1988). O arquivo de declaração de atributos, declarado no arquivo `voyage.names`, é descrito na Tabela 3.

```
1  class.              | Class Attribute
2
3  | Attributes
4  outlook:           nominal (sunny, overcast, rain).
5  temperature:       integer.
6  humity:            integer.
7  windy:             nominal (yes, no).
8  class:             nominal (go, dont_go).
```

Tabela 3: Exemplo de arquivo de declaração de atributos: `voyage.names`.

A primeira declaração em um arquivo de declaração de atributos define qual deve ser o atributo classe, se houver atributo classe definido. No caso de aprendizado supervisionado, o atributo classe é mandatório, e esse atributo pode ser qualquer atributo presente no conjunto de dados. Nesse exemplo, o atributo classe é o atributo `class`, o qual é declarado posteriormente como sendo um atributo nominal que pode assumir os valores `go` e `dont_go`. No caso de conjuntos de dados para aprendizado não supervisionado, o nome do atributo classe deve ser substituído pela palavra `null`, a qual indica que o conjunto de dados não possui classe definida. Após a declaração do atributo classe, os demais atributos são declarados. Cada atributo possui um identificador e um tipo de dado associado. São considerados identificadores válidos aqueles que são combinações de números, letras e "_" (*underscore*), em qualquer seqüência. Para identificadores mais complexos, que envolvem outros caracteres que não sejam os especificados anteriormente (como espaços, letras acentuadas, etc), é necessário colocar o identificador entre aspas. Dessa maneira, são identificadores válidos: `abc`, `1`, `1a`, `_1a`, `"_12a"` e `"válido"`.

Além do arquivo de declaração de atributos, existe o arquivo de declaração de dados. Nesse arquivo são declarados os valores que os atributos presentes no arquivo de declaração de atributos assumem para um determinado conjunto de exemplos. A Tabela 4 na próxima página mostra um trecho desse arquivo na sintaxe *DSX* para o conjunto de dados **voyage**.

Cada linha de um arquivo de declaração de dados representa um exemplo. Sendo assim, o caracter responsável por identificar o final de um exemplo (separador de registros)

```
1   sunny, 25, 72, yes, go
2   sunny, 28, 91, yes, dont_go
3   overcast, 23, 90, yes, go
4   overcast, 29, 78, no, go
5   rain, 22, 95, no, go
6   rain, 19, 70, yes, dont_go
```

Tabela 4: Exemplo de arquivo de declaração de dados: `voyage.data`.

é o caracter de nova linha (representado em muitas linguagem de programação por "\n"). Cada exemplo possui uma seqüência de valores separados por vírgula, ou seja, a vírgula é o caracter responsável por separar os valores de um exemplo (separador de campos). Os valores declarados em cada exemplo, em um arquivo de declaração de dados, devem estar na mesma ordem em que foram declarados no arquivo de declaração de atributos.

## A.2 Os Tipos de Dado da Sintaxe *DSX*

Como mencionado anteriormente, a sintaxe *DSX* deve suportar, além dos tipos de dado mais comuns (como os tipos numérico e nominal), tipos de dado mais sofisticados. Com a popularização do uso de Mineração de Dados para extrair conhecimento de Bases de Dados, os sistemas de aprendizado têm passado a suportar novos tipos de dado, como por exemplo os tipos data e hora. É importante que a sintaxe *DSX* ofereça suporte a diversos tipos de dado para que a sintaxe não limite o uso dos sistemas de aprendizado mais recentes. Por exemplo, se a sintaxe não suporta o tipo de dados data, então dados com esse tipo não podem ser declarados nessa sintaxe. Logo, se o conjunto de dados for convertido da sintaxe padrão para a sintaxe do sistema de aprendizado, o sistema de aprendizado não poderá utilizar a sua capacidade de processar dados do tipo data. Por outro lado, se a sintaxe *DSX* oferecer suporte aos tipos de dado mais sofisticados, é possível converter um conjunto de dados que utiliza esses tipos de dado mesmo para os sistemas de aprendizado que não suportam esses tipos. Isso pode ser feito pela definição de conversões padrão entre os tipos de dado mais complexos e os tipos de dado mais utilizados (numérico e nominal) pelos sistemas de aprendizado mais tradicionais.

Para definir quais tipos de dado devem ser suportados pela sintaxe *DSX*, foram analisados os principais tipos de dado suportados pelos sistemas gerenciadores de bancos de dados. Após essa análise foi decidido dar suporte, na sintaxe *DSX*, aos tipos de dado listados na Tabela 5.

| Nominal | Enumerated |
|---|---|
| Integer | Real |
| Date | Time |
| String | |

Tabela 5: Tipos de dado suportados pela sintaxe *DSX*.

Nas próximas seções são descritos em mais detalhes cada um dos tipos de dado aceitos pela sintaxe *DSX*.

### A.2.1 O Tipo de Dado `Nominal`

O tipo **`nominal`** é utilizado para declarar um atributo que pode assumir um grupo restrito de valores. Existem duas formas de declarar um atributo do tipo **`nominal`**:

1. A primeira consiste na palavra **`nominal`** seguida de uma lista de valores. Essa lista de valores dita quais são os possíveis valores que o atributo pode assumir. Esta opção é amplamente recomendada, uma vez que com essa informação é possível realizar verificações de tipo de dado;

2. A segunda consiste apenas na palavra **`nominal`**, sem a declaração de uma lista de valores. Nesse caso, o atributo em questão pode assumir qualquer valor, e verificações de tipo de dado não são possíveis. Entretanto, essa forma de declaração é útil quando o atributo do tipo **`nominal`** pode assumir um grupo mais numeroso de valores, e o usuário não deseja digitar todos esses valores no arquivo de declaração de atributos.

São exemplos de declarações de atributos do tipo **`nominal`**:

```
AtribEx1: nominal (azul, amarelo, vermelho).
AtribEx2: nominal.
```

### A.2.2 O Tipo de Dado `Enumerated`

O tipo de dado **`enumerated`** é semelhante ao tipo de dado **`nominal`**. A principal diferença é que com o tipo **`enumerated`** é possível definir uma ordem entre os valores que o atributo pode assumir. Entretanto, não existe uma definição explícita de distância entre esses valores. Um exemplo de tipo **`enumerated`** é um atributo que pode assumir os valores **`pequeno`**, **`médio`** e **`grande`**.

Existe somente uma forma de declarar um atributo do tipo `enumerated`. Consiste na palavra `enumerated` seguida de uma lista de valores que o atributo pode assumir. A lista de valores é obrigatória, pois sem ela não é possível identificar a ordem dos atributos, que é a principal informação desse tipo de dado. A lista de valores é ordenada de forma crescente. Um exemplo de declaração de atributos de tipo `enumerated` é:

```
AtribEx3: enumerated(pequeno, "médio", grande).
```

### A.2.3 O Tipo de Dado `Integer`

O tipo de dado `integer` é utilizado para declarar um atributo que pode assumir valores inteiros. Alguns sistemas de aprendizado, como o $C4.5$, não possuem um tipo inteiro, e os atributos numéricos (inteiros e reais) são declarados como um único tipo de dado. No caso específico do $C4.5$ é utilizado o tipo de dado `continuous`. Na sintaxe *DSX*, inteiros e reais são declarados separadamente, o tipo `real` é apresentado na próxima seção. A declaração de um atributo inteiro é feita como no exemplo a seguir:

```
AtribEx4: integer.
```

### A.2.4 O Tipo de Dado `Real`

O tipo `real` é semelhante ao tipo de dado `integer`, com a diferença que um atributo real pode armazenar números com ou sem parte fracionária. Um exemplo de declaração de atributo do tipo `real` é:

```
AtribEx5: real.
```

### A.2.5 O Tipo de Dado `String`

O tipo de dado `string` não está presente na maioria dos sistemas de aprendizado. Esse tipo foi incluído para dar algum suporte à *Mineração de Textos* (*Text Mining*) (Dörre, Gerstl, & Seiffert, 1999). Um atributo do tipo `string` pode assumir como valor uma seqüência de caracteres de tamanho indefinido, essa seqüência pode conter quaisquer caracteres incluindo quebra de linha (`\n`). Para identificar os limites de um dado do tipo `string` é necessário inserir aspas no início e no fim da seqüência de caracteres. Um exemplo de declaração de atributo do tipo `string` é:

```
AtribEx6: string.
```

### A.2.6 O Tipo de Dado `Date`

O tipo de dado `date` permite declarar atributos que podem armazenar datas (dia, mês e ano). A princípio, os valores das datas devem estar no formato `aaaa/mm/dd`, o qual é o formato adotado pela maioria dos sistemas gerenciados de bancos de dados. Entretanto, é possível utilizar declarações estendidas (Seção A.4 na página seguinte) para informar datas em outros formatos. Um exemplo de declaração de um atributo do tipo de dado `date` é:

```
AtribEx7: date.
```

### A.2.7 O Tipo de Dado `Time`

O tipo de dado `time` permite declarar um atributo que pode conter um horário (hora, minuto e segundo). A princípio, os valores dos horários devem estar no formato `hh:mm:ss`. Entretanto, é possível utilizar declarações estendidas (Seção A.4) para informar horários em outros formatos. Um exemplo de declaração de um atributo do tipo de dado `time` é:

```
AtribEx8: time.
```

## A.3 Atributos Virtuais

Como já mencionado, a sintaxe *DSX* permite realizar indução construtiva apoiada pelo usuário. Para isso, *atributos virtuais* podem ser definidos no arquivo de definição de atributos. Atributos virtuais são atributos que são definidos por meio de expressões aritméticas ou lógicas que envolvem um ou mais atributos já definidos no arquivo de declaração de atributos. Os valores que os atributos virtuais assumem no conjunto de dados dependem somente da expressão aritmética que os definem. Esses atributos não possuem dados declarados no arquivo de declaração de dados. Assim, quaisquer modificações nos valores dos atributos envolvidos na expressão aritmética acarreta em uma modificação automática no valor do atributo virtual.

A expressão aritmética que define um atributo virtual pode ser qualquer expressão aritmética válida em Perl (Wall, Christiansen, & Schwartz, 1996). Resumidamente, os identificadores dos atributos assumem o papel de variáveis. O símbolo "$" deve ser adicionado ao identificador da variável. Alguns operadores válidos são + (adição), - (subtração), / (divisão), * (multiplicação), % (resto de divisão), ** (exponenciação), além de funções de conversão de tipos e trigonométricas.

```
1       class.              | Class Attribute
2
3       | Attributes
4       outlook:            nominal (sunny, overcast, rain).
5       temperature:        integer.
6       humity:             integer.
7       windy:              nominal (yes, no).
8       temp_humity_rate:   real    := $temperature/$humity.
9       class:              nominal (go, dont_go).
```

Tabela 6: Exemplo de arquivo de declaração de atributos, `voyage.names`, com declaração de atributo virtual.

A Tabela 6 mostra o arquivo de declaração de atributos `voyage.names` (Tabela 3 na página 31), adicionado do atributo virtual `temp_humity_rate`. A declaração de um atributo virtual inicia com seu identificador e tipo e segue com o símbolo `:=`. Após esse símbolo é utilizada uma expressão aritmética ou lógica válida em Perl. Na expressão somente podem ser utilizados como variáveis os atributos declarados antes do atributo virtual. No exemplo anterior, o atributo `class` não poderia ser utilizado na expressão que define o atributo virtual `temp_humity_rate`, pois `class` está declarado após o atributo virtual.

## A.4 Declarações Estendidas

Declarações estendidas permitem declarar propriedades adicionais aos atributos. Essas propriedades podem ser definidas conforme as necessidades do usuário. Uma declaração estendida deve ser especificada no arquivo de declaração de atributos. Cada declaração estendida é associada a um atributo especifico. Uma declaração estendida permite, por exemplo, declarar qual é o valor mínimo e o máximo que um atributo numérico pode assumir. Caso essa informação seja conhecida, especificar esses valores é mais confiável do que estimar os valores mínimo e o máximo com base no conjunto de exemplos. Essa informação pode ser utilizada para, por exemplo, normalizar o atributo.

Uma declaração estendida inicia com o símbolo de pois pontos ( `:`) seguido por um identificador da declaração estendida. Se a declaração estendida requerer a especificação de um ou mais parâmetros, esses valores podem ser declarados entre parênteses logo após o identificador da declaração estendida. A sintaxe utilizada nas declarações estendidas é baseada na sintaxe dos *funtores* da linguagem de programação lógica Prolog (Bratko, 1990). Um exemplo de declaração estendida é:

```
AtribEx4: integer: min(0): max(100).
```

Nesse exemplo, foram utilizadas duas declarações estendidas para informar o valor máximo e mínimo para o atributo. A declaração `min(0)` informa que o valor mínimo que o atributo pode assumir é 0 e `max(100)` informa que o valor máximo é 100. Atualmente, diversas declarações estendidas são utilizadas pela biblioteca *DOL*. Entre as principais estão:

`min`, `max` e `std_dev` Utilizada para atributos dos tipos `real` e `integer`. Declara qual é o valor mínimo e máximo que o atributo pode assumir nos dados, e o desvio padrão do atributo, respectivamente. Essa informação e utilizada pelo módulo que normaliza os dados;

`date_language` e `date_order` Utilizada para atributos do tipo `date`. Em `date_language` é especificada a língua em que os valores de um atributo do tipo `date` estão escritos. Por exemplo, o valor "Segunda, 7 de janeiro de 2003" corresponde a uma data em português. Atualmente, a biblioteca *DOL* aceita datas em sete línguas diferentes, listadas na Tabela 7. Em `date_order` é especificada a ordem na qual os valores do dia, mês e ano estão especificados. Essa declaração pode assumir três valores `dmy` para dia-mês-ano, `mdy` para mês-dia-ano e `ymd` para ano-mês-dia.

| Identificador da língua | Língua |
|---|---|
| English | Inglês |
| Français | Francês |
| Deutsch | Alemão |
| Español | Espanhol |
| Português | Português |
| Nederlands | Holandês |
| Italiano | Italiano |
| Norsk | Norueguês |
| Svenska | Sueco |
| Dansk | Dinamarquês |
| Suomi | Finlandês |

Tabela 7: As línguas aceitas pela definição estendida `date_language`.

## A.5 Gramática da Sintaxe *DSX*

A sintaxe *DSX* para arquivos de declaração de atributos `.names` pode ser mais formalmente definida pela seguinte gramática:

```
1  S ::=
2      < class-defs >
3    | < feature-defs >
4
5  < class-defs > ::=
6      < feature-name > .
7    | null .
8
9  < feature-name > ::=
10     < identifier >
11
12 < feature-defs > ::=
13     < feature-name > : < feature-type > .
14   | < feature-name > : < feature-type > : < ext-defs > .
15   | < feature-name > : < feature-type > := < expr > : < ext-defs > .
16
17 < feature-type > ::=
18     real
19   | integer
20   | boolean
21   | nominal
22   | nominal ( < list > )
23   | enumerated ( < list > )
24   | date
25   | time
26   | string
27
28 < ext-defs > ::=
29     < ext-def >
30   | < ext-def > : < ext-defs >
31
32 < ext-def > ::=
33     < identifier >
34   | < identifier > ( < list > )
35
36 < list > ::=
37     < identifier >
38   | < identifier > , < list >
```

Um identificador válido (`< identifier >`) pode ser qualquer seqüência de letras, números e *underscores* (_). Ainda, são considerados identificadores válidos quaisquer seqüências de caracteres desde que colocados entre aspas (").

Uma expressão aritmética (`< expr >`) pode ser qualquer expressão aritmética válida em Perl.

# Referências


Batista, G. E. A. P. A. (2003, Março). Pré-processamento de Dados em Aprendizado de Máquina Supervisionado. Tese de Doutorado, ICMC-USP, http://www.icmc.usp.br/~gbatista. 29

Batista, G. E. A. P. A. & M. C. Monard (2003). Descrição da arquitetura e do projeto do ambiente computacional discover learning environment - dle. Technical Report 187, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_187.PDF. 8

Blum, A. & T. Mitchell (1998). Combining labeled and unlabeled data with co-training. In *Proc. 11th Annu. Conf. on Comput. Learning Theory*, pp. 92–100. ACM Press, New York, NY. 1, 5

Boser, B. E., I. Guyon, & V. Vapnik (1992). A training algorithm for optimal margin classifiers. In *Proceedings Fifth ACM Workshop on Computational Learning Theory*, pp. 144–152. 10

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence.* Addison-Wesley. 36

Dörre, J., P. Gerstl, & R. Seiffert (1999). Text Mining: Finding Nuggets in Mountains of Textual Data. In *Fifth International Conference on Knowledge Discovery and Data Mining (KDD-99)*, pp. 398–401. 34

Kohavi, R., D. Sommerfield, & J. Dougherty (1997). Data Mining Using $\mathcal{MLC}$++: A Machine Learning Library in C++. *International Journal on Artificial Intelligence Tools 6*(4), 537–566. 29

Lee, H. D. (2000). Seleção e Construção de Features Relavantes para o Aprendizado de Máquina. Dissertação de Mestrado, ICMC-USP, http://www.teses.usp.br/teses/disponiveis/55/55134/tde-15032002-113112. 30

Matsubara, E. T. & M. C. Monard (2004). Análise experimental do algoritmo de aprendizado de máquina semi-supervisionado co- training. Technical Report XXX, ICMC-USP. to be published. 28

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill. 1

Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning.* CA: Morgan Kaufmann. 10, 31

Wall, L., T. Christiansen, & R. L. Schwartz (1996). *Programming Perl* (2 ed.). O'Reilly & Associates. 8, 35

Witten, I. H. & E. Frank (2000). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann. 29