UNIVERSIDADE FEDERAL FLUMINENSE

# FERRAMENTA MIDAS-UFF
## Módulo de Classificação

Renata Milagres Pereira
Luis Filipe de Mello Santos

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

**Banca examinadora:**

Alexandre Plastino (orientador)
Ana Cristina Bicharra Garcia
Luiz Henrique de Campos Merschmann

**Departamento de Ciência da Computação**
Niterói – Rio de Janeiro – Brasil
Agosto de 2004

# Resumo

Neste trabalho propõe-se uma ferramenta de Mineração de Dados, chamada MIDAS-UFF, capaz de extrair automaticamente informações úteis, não óbvias e possivelmente desconhecidas, implícitas em bancos de dados. Um dos módulos desta ferramenta, o módulo de Classificação, é definido, implementado e documentado. É apresentada uma introdução sobre Mineração de Dados descrevendo suas origens, conceitos e principais aplicações, em especial a tarefa de Classificação, que é o principal tema abordado neste trabalho.

# Agradecimentos

*" Gostaria de agradecer aos meus queridos pais, Valdereis e José Mauro pelo apoio e amor incondicionais. À minha irmã Paulinha pela compreensão e paciência. À minha querida vovó Dilma pelo carinho e afeto. À toda família do Luis que me receberam com tanto carinho em sua casa. Ao professor Plastino pela chance que nos proporcionou. À Deus por tudo. E, principalmente, gostaria de agradecer ao meu Luis, pelo imenso amor, carinho, companhia, bom humor e paciência de sempre! "*

**Renata**

*" Agradeço aos meus pais Nelson e Mírian pelo amor e suporte que sempre me deram, à minha irmã Fernanda pela compreensão e amizade, à minha avó Maria José pelo carinho, à minha Jadinha pela amizade, à família da Renata pelo carinho com que me receberam, ao professor Plastino pela oportunidade, à Renata pelo amor, companhia, amizade e por me fazer uma pessoa mais feliz, e, principalmente, à Deus, pela força. "*

**Luis Filipe**

# Conteúdo

# Figuras

# Tabelas

Capítulo 1

# Introdução

A tecnologia de computação é uma das que se desenvolvem mais rapidamente nos dias atuais. Diz-se que em 1945, no final da Segunda Guerra Mundial, se iniciou a era moderna da computação. Naquela época, os computadores eram enormes e extremamente caros. O acesso a esses computadores era limitado a poucas empresas e a algumas instituições de ensino. Hoje em dia, o custo dos computadores é substancialmente menor e o poder computacional, como capacidade de armazenamento de informações e velocidade de processamento de dados, altamente superior. Este fato tornou possível o desenvolvimento das tecnologias de bancos de dados, como os SGBDs (Sistemas Gerenciadores de Bancos de Dados) e os recentes *Data Warehouses*. Através delas, empresas acumularam imensas quantidades de dados. Com o avanço das redes de computadores e da Internet, essas quantidades de dados se tornaram ainda maiores, chegando-se a *terabytes* de dados armazenados nos sistemas computacionais.

Este crescimento explosivo dos dados armazenados em bancos de dados viabilizou a criação de técnicas inteligentes e automáticas para transformar essas vastas quantidades de dados em conhecimento útil, como padrões, tendências e regras, implícitos nesses dados e potencialmente desconhecidos. Esta é a idéia básica que deu origem a esta nova linha de pesquisa, que une teorias de diversos campos de conhecimento, como estatística, *machine learning*, otimização, inteligência artificial e tecnologia de bancos de dados, denominada **Mineração de Dados**.

A Mineração de Dados é considerada por alguns autores como a parte chave integrante de um conceito mais abrangente, que é o processo de KDD – *Knowledge Discovery in Databases* [1, 18]. Este processo define que a extração de conhecimento de bancos de dados é realizada por uma seqüência bem definida de passos, descritos na Figura 1.1.

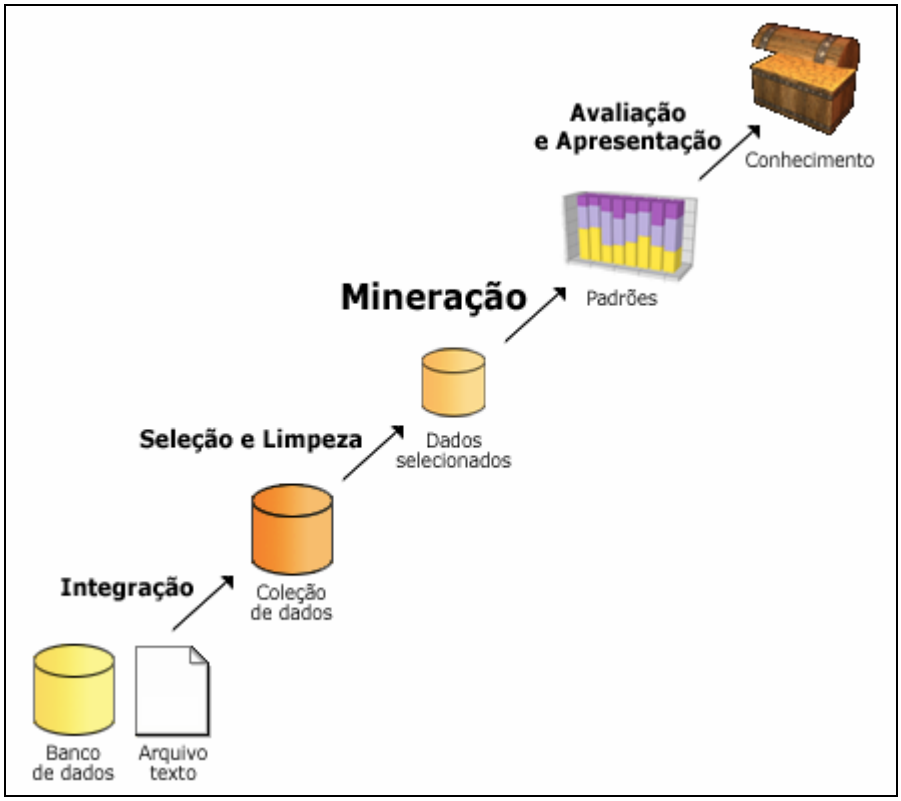


Figura 1.1 – Os passos do processo de KDD

Estes passos são:

1. **Integração:** várias fontes de dados, como bancos de dados armazenados em SGBDs, arquivos texto, etc., podem ser interligadas para formar o conjunto de dados a ser analisado;
2. **Seleção:** dados considerados relevantes para a extração de conhecimento são selecionados;
3. **Limpeza:** dados inconsistentes e com ruído, que podem distorcer o significado dos dados do banco, são tratados;
4. **Mineração:** a parte principal de todo o processo, onde os dados são analisados por técnicas automáticas e inteligentes para a extração de conhecimento;
5. **Avaliação:** a representação do conhecimento extraído é avaliada para identificar quais as informações são úteis ou não, baseada em alguma medida de interesse;

6. **Apresentação:** técnicas visuais são usadas para apresentar o conhecimento extraído pelo processo.

O principal objetivo do processo de KDD é a extração de informações úteis sobre os dados. A comunidade científica concluiu que a Mineração de Dados pode ser dividida em um grupo de "tarefas" principais [4] e diferentes tipos de informação são extraídos por cada uma destas tarefas.

Uma delas, principal tema explorado neste trabalho, é a tarefa de **Classificação**. Trata-se de um problema já bem estudado. Dado um conjunto de itens com múltiplos atributos (características) e pertencendo a uma dada classe, o objetivo da classificação é gerar um modelo que descreva precisamente cada classe com base nos atributos dos itens deste conjunto. A partir deste modelo é possível prever a classe de outros itens.

A Classificação tem uma grande variedade de aplicações, tais como detecção de fraudes, aprovação de crédito, diagnóstico médico, tratamento apropriado de doenças, identificação de mercado consumidor alvo, localização estratégica de pontos comerciais e muitas outras.

Um dos primeiros algoritmos propostos para realizar esta tarefa é o CLS [29]. Seus princípios influenciaram algoritmos subseqüentes, como o clássico ID3 [48] e o C4.5 (sucessor do ID3) [50]. Outros exemplos de algoritmos são o CHAID [32] e CART [11]. Na época em que estes algoritmos foram propostos, o volume de dados a ser considerado não era uma questão importante. Devido ao sensível aumento do tamanho dos bancos de dados, esta questão se tornou essencial nos projetos de algoritmos mais recentes. Exemplos destes algoritmos são: SLIQ [39], SPRINT [54], CLOUDS [8], BOAT [20], PUBLIC [51], RainForest [19], entre outros.

Outra tarefa é a de **Associação**, que consiste em identificar relações entre itens de transações. Estas relações são geralmente representadas através de regras de associação que tem a forma A→B, onde A e B são conjuntos de itens de um mesmo domínio de aplicação. Esta regra é interpretada da seguinte forma: se A ocorre em uma transação, por conseqüência, B também ocorre, com um certo grau de certeza. Usualmente, este grau de certeza é calculado usando a medida de suporte da regra, que pode ser definido como sendo o número de transações que contêm A e B sobre o

total de transações, e a medida de confiança, que pode ser definida como sendo a fração de transações que contêm A e B dentre todas as transações que contêm A.

O exemplo clássico de aplicação da tarefa de Associação é encontrar relações entre itens vendidos em um supermercado. Por exemplo, se a regra {fralda,leite}→{cerveja} fosse encontrada no banco de dados do supermercado, os itens fralda, leite e cerveja poderiam ser estrategicamente posicionados nas prateleiras para aumentar a venda de cerveja.

Muitos algoritmos foram propostos para realizar a tarefa de extração de regras de associação. O mais referenciado na literatura é o algoritmo Apriori [5]. Além deste, algoritmos importantes são FP-Growth [26], DCP [45], DHP [46], DIC [12], DCI [44], Tree Projection [2], entre outros.

A tarefa de **Clusterização** consiste em organizar dados, ou objetos, em grupos, ou *clusters*, de modo que os objetos de um mesmo *cluster* tenham grande similaridade, e objetos de *clusters* diferentes sejam distintos entre si. A similaridade entre os objetos é medida com base nos valores de atributos que os descrevem.

Geralmente, técnicas para realizar esta tarefa se baseiam em medidas de distância entre objetos. São aplicadas de modo que objetos próximos integrem um mesmo *cluster*.

Clusterização tem sido amplamente utilizada em áreas como processamento de imagens, reconhecimento de padrões, pesquisas de *marketing* e *e-commerce*. Na astronomia, por exemplo, utilizou-se a clusterização para dividir as estrelas em grupos, o que possibilitou um maior entendimento sobre os princípios de evolução das estrelas. Outro exemplo de aplicação é, em um banco de dados de um supermercado, encontrar grupos de consumidores que têm comportamento similar, o que poderia ajudar na elaboração de estratégias de *marketing.*

O algoritmo *K*-Means [36] é o mais referenciado na literatura para realizar esta tarefa. Além deste, são importantes os algoritmos Clarans [42], WaveCluster [55], CURE [23], Birch [35], Rock [24], Chameleon [31], AutoClass [14], entre outros.

A tarefa de encontrar **padrões de seqüência** equivale à extração de relações entre eventos que ocorrem ao longo do tempo (ou qualquer outro critério de ordenação entre eventos).

Alguns autores definem padrões de seqüência como uma especialização de regras de associação, onde a relação entre os itens não ocorre necessariamente na mesma transação, e sim em transações seqüenciais.

Algumas aplicações da extração de padrões de seqüência são: previsão do tempo, transações de negócios, mercado de ações, entre outras. Por exemplo, no mercado de ações, um padrão de seqüência hipotético pode ser: se as ações da AT&T se valorizam por dois dias consecutivos e as ações da DEC se mantêm estáveis, então as ações da IBM se valorizam no dia seguinte, com 75% de certeza.

Muitos algoritmos propostos para extração de padrões de seqüência são adaptações de algoritmos de extração de regras de associação. Por exemplo, o AprioriAll [6] e o GSP [7], dois dos algoritmos mais referenciados na literatura, são adaptações do algoritmo Apriori [5], assim como os algoritmos FreeSpan [27] e PrefixSpan [47] são adaptações do algoritmo FP-Growth [26].

Uma enorme quantidade de ferramentas para realizar estas tarefas existe hoje no mercado [21,16,33]. Muitas delas foram construídas por companhias fornecedoras de SGBDs (Sistemas Gerenciadores de Bancos de Dados) e são integradas a esses sistemas. Por exemplo, a Intelligent Miner [30], integrada ao DB2, da IBM Corporation e a Oracle Data Mining [43], integrada ao Oracle 10g, da Oracle Corporation. Existem também ferramentas construídas por terceiros, como a Clementine [56], da SPSS Inc.; a Enterprise Miner [53], do SAS Institute; a PolyAnalyst [37], da Megaputer Intelligence Inc. e muitas outras. Elas se diferenciam pelas técnicas de Mineração de Dados fornecidas, pela conectividade a bancos de dados, pela complexidade de uso, entre outros [16,21].

A Mineração de Dados é hoje uma linha de pesquisa em várias universidades de todo o mundo. Estas universidades também têm desenvolvido ferramentas para Mineração de Dados como produto da pesquisa que realizam. Por exemplo, a Simon Fraser University, do Canadá, desenvolveu a DBMiner [28,25,15], e a Waikato University, da Nova Zelândia, desenvolveu a WEKA (Waikato Environment for Knowledge Analysis) [59,57]. Esta última implementa algumas técnicas de Mineração de Dados utilizando a linguagem de programação Java (multi-plataforma) e é distribuída livremente.

A Universidade Federal Fluminense também atua na área de Mineração de Dados e, com inspiração dada por projetos como os das ferramentas DBMiner e WEKA, descritos acima, inicia com este trabalho o desenvolvimento de uma ferramenta de Mineração de Dados. Esta ferramenta se chama MIDAS-UFF (Mineração de Dados na UFF) e neste trabalho propõe-se sua primeira versão, trazendo a implementação de um dos módulos da ferramenta: o módulo de Classificação.

O restante do conteúdo deste trabalho está organizado da seguinte forma:

**Capítulo 2** – A tarefa de Classificação será explorada. Sua definição, aplicações e métodos existentes para sua realização serão apresentados. Os algoritmos implementados no módulo de Classificação da ferramenta MIDAS-UFF serão descritos detalhadamente.

**Capítulo 3** – O módulo de Classificação da ferramenta MIDAS-UFF, principal contribuição deste trabalho, será apresentado. A interface com o usuário e as formas de entrada e saída de dados serão descritas.

**Capítulo 4** – Serão abordados detalhes de implementação. A linguagem e o ambiente de programação, e as principais estruturas de dados utilizadas serão apresentadas. Também é feita uma análise de desempenho da implementação dos algoritmos que integram o módulo de Classificação.

**Capítulo 5** – Serão apresentadas conclusões e sugestões de trabalhos futuros.

Capítulo 2

# A Tarefa de Classificação

Este capítulo apresenta a tarefa de Classificação, que é o principal tema deste trabalho.

É fornecida uma descrição teórica sobre os principais conceitos que a tarefa envolve e como estes conceitos são tratados pelos métodos propostos para a sua realização.

Na Seção 2.1 é apresentada a definição da tarefa.

Uma descrição dos principais métodos existentes para realizar a tarefa é apresentada na Seção 2.2, com ênfase na técnica de geração de árvores de decisão.

A Seção 2.3 contém detalhes sobre dois dos algoritmos geradores de árvores de decisão mais referenciados na literatura. Estes algoritmos, SLIQ [39] e SPRINT [54], foram implementados para integrar o módulo de Classificação da ferramenta MIDAS-UFF.

## 2.1 – Definição da tarefa e aplicações

A tarefa de Classificação pode ser definida de maneira simples. Considere um banco de dados contendo um conjunto de registros onde cada registro possui um conjunto de atributos (ou *features*). Este banco de dados é conhecido como **banco de treinamento** (ou *training set*). Os atributos deste banco de dados são divididos em dois grupos: **atributos categóricos** e **atributos numéricos**. Os atributos categóricos se caracterizam por possuírem o domínio de valores discreto, e os atributos numéricos, por outro lado, por possuírem o domínio contínuo. Um dos atributos dos registros é o **atributo de classe** (ou *target attribute*), atributo categórico que indica a que classe (ou categoria) o registro pertence[1]. Os outros atributos são conhecidos como **atributos preditores**. O objetivo da tarefa de Classificação é extrair do banco de dados de treinamento um modelo que descreva cada classe em função de valores dos atributos preditores. Este modelo, então, pode ser usado para prever a classe de futuros registros ou para que se obtenha um maior entendimento do domínio do problema que está sendo tratado.

Para ilustrar uma aplicação da tarefa de Classificação, considere um banco de dados de uma empresa de crédito, representado na Tabela 2.1.

Tabela 2.1 – Banco de dados de uma empresa de crédito

| salário anual | Educação | CLASSE |
|---|---|---|
| 10.000 | Ensino médio | REJEITADA |
| 40.000 | Graduação | ACEITA |
| 15.000 | Graduação | REJEITADA |
| 75.000 | Pós-graduação | ACEITA |
| 18.000 | Pós-graduação | ACEITA |

Neste banco de dados, cada registro corresponde a uma requisição de empréstimo, classificada como ACEITA, quando o empréstimo é aprovado, ou REJEITADA, quando o empréstimo é negado. Cada registro tem dois atributos, salário

[1] A tarefa de Classificação é considerada pela comunidade de *Machine Learning* como uma tarefa de **aprendizado supervisionado** (*Supervised Learning*), dado que a informação da classe de cada registro é conhecida *a priori*.

anual e educação, relativos ao cliente que solicita o empréstimo. O atributo salário anual é um atributo numérico, pois possui o domínio de valores contínuo, e o atributo educação é um atributo categórico, pois seu domínio de valores, {Ensino médio, Graduação, Pós-graduação}, é discreto.

A execução de um método que realiza a tarefa de Classificação sobre este banco de dados poderia extrair o modelo ilustrado na Figura 2.1.

(**salário anual** < 20.000) $\wedge$ (**educação** $\in$ {Pós-graduação}) → ACEITA
(**salário anual** < 20.000) $\wedge$ (**educação** $\notin$ {Pós-graduação}) → REJEITADA
(**salário anual** ≥ 20.000) → ACEITA

Figura 2.1 – Exemplo de modelo de Classificação

Este modelo define regras que classificam uma requisição de empréstimo como ACEITA ou REJEITADA em função de valores dos atributos salário anual e educação. Esta empresa de crédito poderia, então, usar este modelo para auxiliar na decisão de aceitar ou rejeitar requisições de empréstimos a futuros clientes.

O simples exemplo descrito acima é apenas uma demonstração hipotética de como a tarefa de Classificação pode ser aplicada. Soluções de problemas do mundo real com um nível de complexidade extremamente maior têm sido obtidas através de aplicações bem sucedidas desta tarefa. Algumas áreas onde a tarefa de Classificação tem sido aplicada são:

- **Detecção de fraudes**

Um modelo de classificação de clientes de uma empresa ajuda a identificar as características que clientes fraudadores possuem. Com este conhecimento disponível, novas fraudes podem ser evitadas, reduzindo os prejuízos da empresa.

- **Diagnóstico médico**

Um modelo de classificação de pacientes de um hospital ajuda a identificar doenças que pacientes com determinados sintomas podem ter. Este conhecimento

pode ajudar na preparação de diagnósticos. Além disso, as causas de doenças podem ser melhor entendidas.

- **Previsão do tempo**

Um modelo de classificação de características meteorológicas ajuda a identificar fenômenos da natureza como, por exemplo, tornados ou terremotos. Este conhecimento pode ajudar na tomada de ações preventivas com o objetivo de diminuir os prejuízos causados por estes fenômenos.

- **Comércio eletrônico**

Um modelo de classificação de clientes ajuda a identificar os interesses que cada tipo de comprador pode ter. Isto permite a elaboração de melhores estratégias de *marketing*. Por exemplo, propagandas eletrônicas de determinados produtos podem ser enviadas a determinados grupos de clientes alvo, o que pode aumentar suas vendas.

## 2.2 – Técnicas existentes para Classificação

A tarefa de Classificação é um dos problemas mais importantes na área de Mineração de Dados e, provavelmente, é a tarefa mais utilizada. Este problema vem sendo extremamente estudado pela comunidade de *Machine Learning* ao longo dos anos. Por isto, muitos algoritmos, baseados em diferentes técnicas, já foram propostos para a realização desta tarefa [58].

Dentre as técnicas propostas mais conhecidas estão: métodos Bayesianos [13], algoritmos genéticos [22], redes neurais [10,34,52] e árvores de decisão [11,50]. Dentre elas, as árvores de decisão são consideradas pela comunidade científica como uma importante técnica para implementar a tarefa de Classificação [4]. Os motivos para isto são:

- Sua representação é simples e intuitiva, fácil de ser compreendida [11];
- O tempo de geração do modelo é substancialmente menor em comparação ao de outras técnicas, o que viabiliza seu uso para grandes bancos de dados;

- A qualidade do modelo gerado é igual ou maior a de outras técnicas [40];
- Não requer informações adicionais. As informações contidas no banco de treinamento são suficientes [17];
- Podem ser facilmente convertidas em consultas SQL[2] para acessar bancos de dados armazenados em SGBDs [3];

Com base nestes fatos, as árvores de decisão e os algoritmos existentes para gerá-las foram explorados com mais ênfase neste trabalho, e são o tema da próxima seção.

## 2.2.1 – Árvores de decisão

Para ilustrar o que é uma árvore de decisão, considere o banco de dados representado na Tabela 2.1, reproduzida abaixo:

Tabela 2.1 – Banco de dados de uma empresa de crédito

| salário anual | educação | CLASSE |
|---|---|---|
| 10.000 | Ensino médio | REJEITADA |
| 40.000 | Graduação | ACEITA |
| 15.000 | Graduação | REJEITADA |
| 75.000 | Pós-graduação | ACEITA |
| 18.000 | Pós-graduação | ACEITA |

Uma árvore de decisão que caracteriza um modelo de Classificação para este banco de dados está representada na Figura 2.2.

Uma árvore de decisão é composta de nós internos e folhas. Cada nó interno representa um teste (ou predicado) envolvendo um dos atributos preditores do banco de dados de treinamento. Para cada possível saída deste teste existe uma aresta que liga o nó interno a um de seus filhos. A cada folha é associada uma das possíveis classes dos registros.

[2] SQL (Structured Query Language) é uma linguagem padrão de acesso a bancos de dados, usada pela maior parte dos SGBDs atuais.

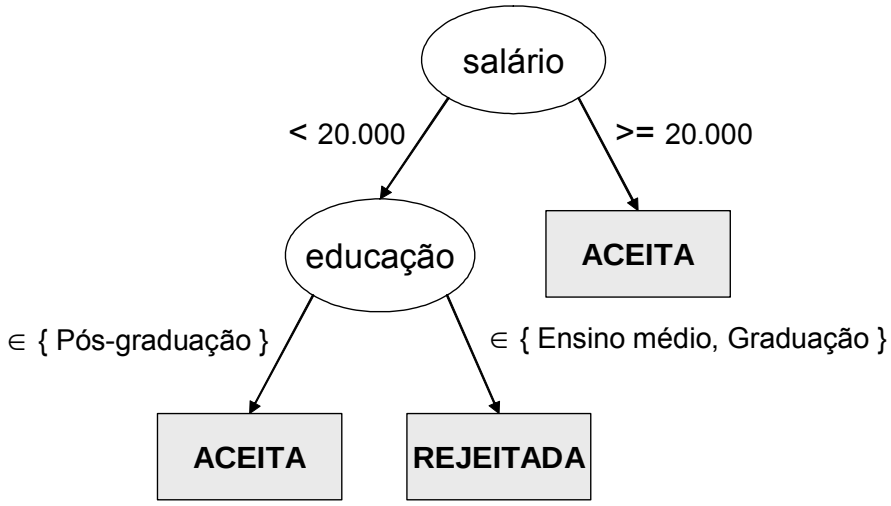


Figura 2.2 – Exemplo de árvore de decisão

Então, para classificar novos registros, a árvore de decisão é percorrida a partir do nó raiz até que uma folha seja alcançada. Em cada nó interno aplica-se o teste ao registro sendo analisado para que se descubra a qual nó filho o registro pertence. A folha alcançada ao final do processo representa a classe do registro. Por exemplo, a árvore de decisão da Figura 2.2 indica que uma requisição de empréstimo feita por um cliente com salário anual ≥ 20.000 ou educação ∈ {Pós-graduação} deve ser ACEITA; caso contrário, deve ser REJEITADA.

Muitos algoritmos para gerar árvores de decisão a partir de bancos de dados já foram propostos. A maioria deles gera a árvore de decisão em duas fases distintas: a **fase de construção** e a **fase de poda**.

Basicamente, a fase de construção de todos tem a mesma estrutura: inicialmente, o banco de dados é associado ao nó raiz da árvore de decisão. A cada nó aplica-se um critério de particionamento, conhecido como ***split***. Este *split* geralmente envolve um atributo ***A*** (conhecido como atributo de teste) e um número ***n*** de subconjuntos disjuntos que, unidos, formam o domínio de valores do atributo ***A***. Os dados deste nó são divididos em ***n*** partições onde cada uma corresponde a um destes ***n*** subconjuntos, ou seja, cada partição contém todos os registros cujos valores do atributo ***A*** pertencem ao seu subconjunto correspondente. Cada uma destas partições resultantes é associada a um novo nó, filho do nó sendo tratado. Este processo é realizado recursivamente até que cada nova partição gerada contenha apenas registros

pertencentes a uma mesma classe, correspondendo aos nós folha da árvore. A decisão de como realizar um particionamento segue uma estratégia gulosa, de maneira que o ***split*** escolhido é sempre o que gera partições descendentes contendo registros com a menor diversidade de classes possível. Esta estrutura é ilustrada no algoritmo da Figura 2.3.

```
(1) Particiona(Nó N) {
(2)     Se(todos os registros associados a N  pertencem a mesma classe)
(3)         retorna
(4)     Avalie splits para todos os atributos dos registros associados a N
(5)     Use o melhor split encontrado em (4) para criar nós N1, N2, ..., Nn filhos de N
(6)     Para i = 1 até n faça
(7)         Particiona(Ni)
(8) }
```

Figura 2.3 – Estrutura básica da fase de construção de árvores de decisão

O algoritmo tem como entrada um nó *N*. A linha (2) verifica se os registros associados a este nó pertencem a uma mesma classe. Em caso positivo, a linha (3) retorna para o ponto de chamada, sem particionar o nó *N*, que se torna uma folha da árvore de decisão. Caso contrário, a linha (4), provavelmente a mais importante do algoritmo, encontra o melhor *split* a ser usado para particionar o nó *N*. A linha (5) realiza o particionamento de *N* usando o *split* encontrado na linha (4). Os novos nós $N_1$, $N_2$, ..., $N_n$ representam, na árvore de decisão, nós filhos do nó *N*. Finalmente, as linhas (6) e (7) são responsáveis por particionar recursivamente os novos nós criados na linha (5). A geração da árvore de decisão é iniciada invocando-se o algoritmo com o nó raiz, que é inicialmente criado contendo todos os registros do banco de dados.

O ponto principal deste algoritmo é a avaliação de *splits* (linha (4)). Ela é feita com base no **índice de diversidade** (ou medida de entropia) entre os registros contidos nos nós que são gerados por um determinado *split*. Quanto menor este índice de diversidade, melhor é o *split*.

Existem várias técnicas para calcular este índice [58]. Entre as mais referenciadas na literatura estão o *information gain* e o *gini index* [11]. O *information gain* é definido pela equação 2.1:

$$\textbf{\textit{Gain}(S)} = -\sum_j \textbf{\textit{p}}_j * \textbf{log}_2\ \textbf{\textit{p}}_j \qquad (2.1)$$

E o gini index é definido pela equação 2.2:

$$\textbf{\textit{Gini}(S)} = 1 - \sum_j \textbf{\textit{p}}_j^2 \qquad (2.2)$$

onde S é um conjunto de registros; 1 ≤ j ≤ total de classes; e $p_j$ é a freqüência relativa da classe j em S.

Quanto menor o valor deste índice, menor a diversidade de classes entre os registros do conjunto S.

Para calcular o índice de diversidade de um *split*, faz-se uma média ponderada entre os índices de diversidade de cada partição gerada por este. Ou seja, se um *split L* divide um conjunto de registros S em *P* partições, $S_1$, ..., $S_p$, o seu índice de diversidade é encontrado pela seguinte equação:

$$\textbf{índice}_{\textbf{split}}\textbf{(}\textit{L}\textbf{)} = \frac{n_1}{n}\,\textbf{índice(S}_1\textbf{)} + \frac{n_2}{n}\,\textbf{índice(S}_2\textbf{)} + \ldots + \frac{n_P}{n}\,\textbf{índice(S}_P\textbf{)} \qquad (2.3)$$

onde índice($S_j$) representa uma técnica de cálculo de índice de diversidade (*information gain*, *gini index*, ou qualquer outra) e $n_j$ representa o número de registros na partição j, para 1 ≤ j ≤ *P*; e n representa o número de registros de S.

Então, o *split* com menor índice de diversidade, dado pela equação (2.3), é usado para particionar o nó em questão.

A árvore de decisão obtida na fase de construção é, em geral, excessivamente sensível às características peculiares e irregularidades estatísticas dos registros do banco de treinamento. Por isso, a maioria dos algoritmos propostos para gerar árvores de decisão realiza uma **fase de poda** após a fase de construção. O objetivo da fase de poda é retirar da árvore de decisão os nós que foram gerados como conseqüência de irregularidades estatísticas contidas no banco de treinamento, obtendo-se uma árvore menor, mais intuitiva e com menor estimativa de erro para futuros registros.

Existem vários algoritmos para podar árvores de decisão. Alguns usam o mesmo banco de dados utilizado para treinamento, como por exemplo, *Pessimistic-Pruning* [49], que são criticados por gerarem árvores muito grandes e, algumas vezes, com alta

taxa de erro. Outros usam um banco de dados específico, como por exemplo, *Cost-Complexity* [11], que são criticados por gerarem múltiplas árvores candidatas até encontrar a solução final, o que pode ser computacionalmente caro.

Outro importante grupo de algoritmos de poda são os que se baseiam no **Princípio MDL** (*Minimum Description Length principle*) [38]. Eles não necessitam de um banco de dados para realizar a poda, pois se baseiam apenas na estrutura da árvore. São considerados eficientes por gerarem árvores de pequena estimativa de erro e significativamente menores que outros algoritmos e por requererem pouco esforço computacional. Os algoritmos geradores de árvores de decisão mais recentes, como SLIQ [39], SPRINT [54] e PUBLIC [51], utilizam o princípio MDL na fase de poda. Na Seção 2.3, onde serão descritos os algoritmos SLIQ e SPRINT, o princípio MDL e a maneira como este é aplicado aos algoritmos serão abordados com maiores detalhes.

## 2.3 – Algoritmos implementados

Como visto anteriormente, as árvores de decisão representam uma técnica atrativa para realizar a tarefa de Classificação em um ambiente de Mineração de Dados. Muitos algoritmos geradores de árvores de decisão foram propostos nas últimas décadas. O CLS (*Computer Learning System*) [29], proposto em 1966, pode ser considerado o primeiro sistema a realizar esta tarefa. Seus autores eram pesquisadores da área de psicologia cognitiva. Sua maior contribuição foi de aspecto teórico, introduzindo um modelo de formação de conceitos, que se baseava na divisão recursiva de um conjunto de registros em subconjuntos com registros tendo o mesmo valor para uma determinada característica. Este processo continuava até que todos os registros de um conjunto representassem um mesmo conceito. Esta idéia de **divisão e conquista** influenciou a maioria dos algoritmos geradores de árvores de decisão.

O CLS não era eficiente. Ele analisava todo o espaço de possíveis árvores de decisão, para alguns valores de profundidade, para extrair a melhor. O clássico algoritmo ID3 (*Itemized Dichotomizer 3*) [48], e seu sucessor C4.5 [50], otimizaram o CLS utilizando a medida de entropia conhecida como *information gain* (definida pela equação 2.1) para decidir como particionar conjuntos de registros, o que fez com que

não fosse necessário gerar mais de uma árvore de decisão, como fazia o CLS. O algoritmo CART [11], similarmente, faz uso da medida de entropia conhecida como *gini index*.

Existe também uma outra família de algoritmos descendentes do sistema AID (*Automatic Interaction Detection*) [41]. Este sistema foi originalmente proposto para encontrar dependências estatísticas entre variáveis através da construção de árvores de decisão, mas também tem sido usado no contexto geral de Mineração de Dados. O mais famoso desta família é o CHAID (*Chi-squared Automatic Interaction Detector*) [32], implementado por famosas ferramentas de Mineração de Dados, como a Clementine [56], da SPSS Inc. e a Enterprise Miner [53], do SAS Institute. A grande diferença do CHAID para o ID3 e CART é que o CHAID não tem uma fase de construção e outra de poda. A geração da árvore é realizada em um único passo, que é interrompido antes que a árvore apresente sensibilidade a características exclusivas e flutuações estatísticas do banco de dados de treinamento. Outra diferença é que o CHAID só trabalha com atributos categóricos.

Estes algoritmos não foram projetados para trabalhar com bancos de dados residentes em disco. Com o aumento sensível dos bancos de dados atuais, a escalabilidade[3] tornou-se uma questão essencial. Algoritmos como SLIQ [39], SPRINT [54], PUBLIC [51], entre outros, foram projetados para solucionar esta questão.

Diante deste cenário e pelo fato de o SPRINT ser atualmente um dos algoritmos de extração de árvores de decisão mais referenciados na literatura, ele e seu predecessor, SLIQ, foram os algoritmos escolhidos para integrar o módulo de classificação da ferramenta MIDAS-UFF, e são apresentados a seguir.

### 2.3.1 – SLIQ

O SLIQ (*Supervised Learning In Quest*) [39] é um algoritmo gerador de árvores de decisão projetado para ser escalável. É capaz de tratar grandes bancos de dados de treinamento, independentemente da quantidade de atributos, que podem ser numéricos ou categóricos, e do número de classes existentes.

[3] Um algoritmo é dito escalável quando seu tempo de execução varia na mesma proporção que o tamanho do problema que ele soluciona, mantendo fixa a quantidade de memória principal.

Para atributos numéricos, consideram-se *splits* da forma $A < v$, onde $v$ é um valor real e *A* é um dos atributos numéricos. Este valor $v$ é a média entre dois valores consecutivos do atributo. Portanto, sendo $v_1, v_2, \ldots, v_n$ os valores do atributo, existem *n*–1 possíveis *splits*. Durante a avaliação de *splits* destes atributos, é preciso primeiramente ordenar os registros com base nos valores do atributo que está sendo considerado. Isto faz com que o custo da ordenação seja dominante na avaliação destes *splits*.

Para atributos categóricos, consideram-se *splits* da forma $A \in S$, onde *S* é um subconjunto dos possíveis valores do domínio do atributo *A*. Para um domínio de *n* possíveis valores, existem $2^n - 2$ possíveis *splits*[4], pois para cada subconjunto do domínio existe um possível *split*, com exceção do subconjunto vazio e do próprio conjunto S. Então, a avaliação de *splits* destes atributos é dominada pela geração destes subconjuntos.

Devido às formas de *splits* utilizadas, as árvores geradas pelo SLIQ são sempre binárias, ou seja, cada nó interno possui exatamente dois filhos. Os registros que satisfazem ao *split* são associados ao filho da esquerda, e os que não satisfazem, ao filho da direita.

As características do SLIQ que o fazem ser capaz de classificar grandes bancos de dados de treinamento são:

- **pré-ordenação**, que diminui o custo de avaliação de *splits* de atributos numéricos. Outros algoritmos, como CART [11] e C4.5 [50], reordenam o banco de dados sempre que um atributo numérico é analisado. O SLIQ faz a ordenação apenas uma vez;
- **construção da árvore em largura** (*breadth-first growth*), o que permite a avaliação de *splits* para vários nós da árvore de decisão simultaneamente em um único passo sobre o banco de dados;
- **algoritmo de geração de subconjuntos rápido e eficaz**, que possibilita uma análise mais eficiente de *splits* para atributos categóricos;

[4] Dado um conjunto **S** de ***n*** elementos, o total de subconjuntos de **S** é igual a $2^n$.

- **algoritmo de poda baseado no princípio MDL**, que requer pequeno esforço computacional e resulta em árvores de decisão compactas e de qualidade.

O SLIQ funciona da seguinte forma: inicialmente, o banco de dados de treinamento é dividido em **listas de atributos**. Para cada atributo preditor deste banco é criada uma lista de atributo correspondente. É criada também uma outra lista, denominada **lista de classe** (*class list*), que contém as classes de cada registro do banco de dados de treinamento e a informação de qual folha da árvore o contém. Então, um passo sobre o banco de dados é necessário para preencher estas listas. A estrutura de cada registro de uma lista de atributo tem a forma **<valor, índice da lista de classe>** e cada registro da lista de classe tem forma **<classe, ponteiro para uma folha da árvore de decisão>**. Cada registro ***i*** do banco de dados de treinamento é associado à entrada de índice ***i*** da lista de classe. Todas as entradas da lista de classe são associadas ao nó raiz, o que significa que todos os registros do banco de dados estão inicialmente contidos no nó raiz.

Após a divisão do banco em listas de atributos, as listas de atributos numéricos são pré-ordenadas.

Para ilustrar o funcionamento do SLIQ, considere o banco de treinamento da Figura 2.4. Cada registro deste banco corresponde aos dados de um cliente de uma empresa de seguro de carros.

| *rid* | IDADE | TIPO DE CARRO | RISCO |
|---|---|---|---|
| 0 | 23 | família | ALTO |
| 1 | 17 | esporte | ALTO |
| 2 | 43 | esporte | ALTO |
| 3 | 68 | família | BAIXO |
| 4 | 32 | carga | BAIXO |
| 5 | 20 | família | ALTO |

Figura 2.4 – Banco de dados de treinamento

Este banco contém três atributos, IDADE, TIPO DE CARRO e RISCO. O atributo IDADE é um atributo numérico, e os atributos TIPO DE CARRO e RISCO são categóricos. O que se deseja é um modelo que descreva os clientes como sendo de

“ALTO” ou “BAIXO” risco para a empresa. Então, IDADE e TIPO DE CARRO são os atributos preditores e RISCO é o atributo de classe.

O SLIQ, inicialmente, faz um passo sobre o banco de dados para criar as listas dos atributos IDADE e TIPO DE CARRO, além da lista de classe. As listas de atributos resultantes e a lista de classe estão ilustradas na Figura 2.5.

| VALOR | Índice da lista de classe |
|---|---|
| 23 | 0 |
| 17 | 1 |
| 43 | 2 |
| 68 | 3 |
| 32 | 4 |
| 20 | 5 |

Lista do atributo IDADE

| VALOR | Índice da lista de classe |
|---|---|
| família | 0 |
| esporte | 1 |
| esporte | 2 |
| família | 3 |
| carga | 4 |
| família | 5 |

Lista do atributo TIPO DE CARRO

| | CLASSE | FOLHA |
|---|---|---|
| 0 | ALTO | RAIZ |
| 1 | ALTO | RAIZ |
| 2 | ALTO | RAIZ |
| 3 | BAIXO | RAIZ |
| 4 | BAIXO | RAIZ |
| 5 | ALTO | RAIZ |

Lista de CLASSE

Figura 2.5 – Primeiro passo do SLIQ

A lista do atributo IDADE, que é um atributo numérico, é pré-ordenada, como ilustra a Figura 2.6.

| VALOR | Índice da lista de classe |
|---|---|
| 17 | 1 |
| 20 | 5 |
| 23 | 0 |
| 32 | 4 |
| 43 | 2 |
| 68 | 3 |

Lista do atributo IDADE

| VALOR | Índice da lista de classe |
|---|---|
| família | 0 |
| esporte | 1 |
| esporte | 2 |
| família | 3 |
| carga | 4 |
| família | 5 |

Lista do atributo TIPO DE CARRO

| | CLASSE | FOLHA |
|---|---|---|
| 0 | ALTO | RAIZ |
| 1 | ALTO | RAIZ |
| 2 | ALTO | RAIZ |
| 3 | BAIXO | RAIZ |
| 4 | BAIXO | RAIZ |
| 5 | ALTO | RAIZ |

Lista de CLASSE

Figura 2.6 – Pré-ordenação

A partir deste ponto, inicia-se a avaliação de *splits* para a criação de novos nós da árvore de decisão. A construção da árvore é feita em largura, o que faz com que *splits* para todas as folhas correntes da árvore sejam analisados simultaneamente. O SLIQ usa o *gini index* (definido na equação 2.2) para medir o índice de diversidade de

*splits*. Por isso, a freqüência de classes em partições do banco de dados contidas em cada folha precisa ser conhecida. Então, o SLIQ faz uso de estruturas de dados chamadas de **histogramas** para armazenar a distribuição de classes de cada partição. Estes histogramas estão associados a cada folha da árvore, e têm formas diferentes para atributos numéricos e categóricos.

A avaliação de *splits* usando um atributo numérico *A* requer um par de histogramas, chamados de $C_{above}$ e $C_{below}$, que são listas de itens com a estrutura <classe, contador>. Cada item destas listas corresponde a um contador para uma classe do banco de dados de treinamento. A avaliação é feita percorrendo-se toda a lista do atributo numérico. O $C_{below}$ armazena a distribuição das classes da parte da lista já processada, e o $C_{above}$, a distribuição das classes do restante da lista. Antes do processamento da lista, os contadores de cada classe do $C_{below}$ são iniciados com zero, enquanto os contadores do $C_{above}$ são iniciados com o número total de registros, existentes na partição, que pertencem as suas classes correspondentes (esta informação é descoberta no momento da criação do nó). Para cada valor $\boldsymbol{v}$ lido, encontra-se a entrada da lista de classe correspondente, que contém a classe $\boldsymbol{c}$ do registro associado ao valor $\boldsymbol{v}$, e a folha da árvore que o contém. Então, os histogramas desta folha são atualizados para refletir o processamento de $\boldsymbol{v}$. O contador da classe $\boldsymbol{c}$ do $C_{below}$ é incrementado, enquanto o contador da classe $\boldsymbol{c}$ do $C_{above}$ é decrementado. Com as informações dos histogramas, o *gini index* para o *split* $A < (\boldsymbol{v} + \boldsymbol{v_2}) / 2$, onde $\boldsymbol{v_2}$ é o valor seguinte de $\boldsymbol{v}$ na lista, é calculado. Cada folha armazena o *split* que será usado para particioná-la. Caso o *gini index* deste *split* seja menor que do *split* corrente armazenado pela folha, a folha o salva, substituindo o antigo. Este processo se repete até o penúltimo valor da lista (o último valor não é processado, pois seu *split* correspondente levaria todos os registros para a partição da esquerda e nenhum para a direita).

Para ilustrar como é realizado o processo de análise de *splits* de atributos numéricos, considere a Figura 2.7, que contém o processo de avaliação de *splits* do atributo IDADE.

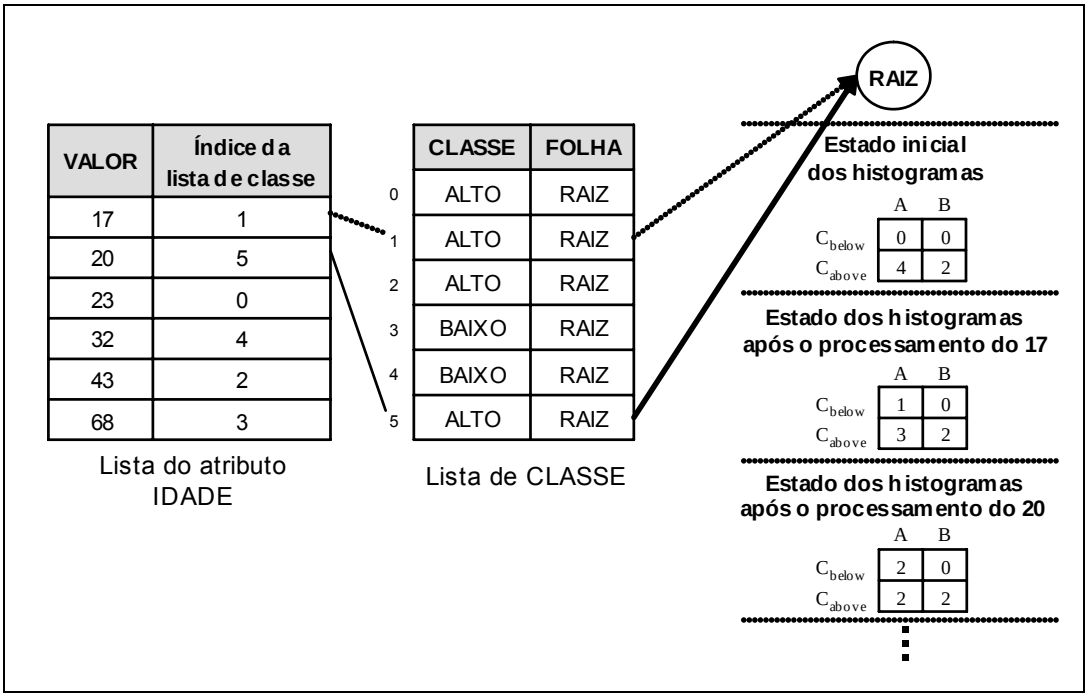


Figura 2.7 – Avaliação de *splits* de um atributo numérico

Antes do processo, $C_{below}$ tem seus contadores inicializados com zero e $C_{above}$ tem seus contadores inicializados com a distribuição de classes dos registros associados ao nó raiz (todos os registros do banco de treinamento). O primeiro valor lido é o 17. Então, o *split* que será avaliado é IDADE < (17 + 20) / 2. O valor 17 está associado à entrada de índice 1 da lista de classe. Esta entrada informa que a classe do registro associado ao valor 17 é "ALTO", e que este registro está associado ao nó raiz. Os histogramas são, então, atualizados para refletir o processamento do valor 17. O contador da classe "ALTO" de $C_{below}$ é incrementado e o de $C_{above}$ é decrementado. O *gini index* para este *split* é calculado antes do processamento do próximo valor. O histograma $C_{below}$ contém a distribuição de classes da partição da lista que obedece ao *split* e o $C_{above}$, a distribuição da que não obedece. Então, para saber a freqüência de uma classe em uma partição, basta dividir o contador desta classe pelo total de registros da partição. Por exemplo, a freqüência da classe "ALTO" na partição da esquerda é 1 / 1. O *gini index* da partição da esquerda é calculado da seguinte forma:

$$\mathbf{Gini(S) = 1 - \sum_j p_j^2} = 1 - ( (1/1)^2 + (0/1)^2 ) = 0$$

E o *gini index* da partição da direita:

$$\textbf{\textit{Gini}(S)} = 1 - \sum_j \boldsymbol{p_j^2} = 1 - ( (3/5)^2 + (2/5)^2 ) = 0{,}48$$

O índice do *split* (dado pela equação 2.3) é, então:

$$\textit{índice}_{split}(\text{IDADE} < 18.5) = (1/6) * (0) + (5/6) * (0{,}48) = 0{,}4$$

Caso este valor seja menor do que o índice do *split* corrente armazenado no nó raiz, ele é salvo em lugar do anterior.

Analogamente, os outros valores da lista são processados até que se alcance o penúltimo valor da lista.

A avaliação de *splits* usando um atributo categórico *B* é similar à de um atributo numérico, com pequenas diferenças. Apenas um histograma é necessário, conhecido como *count matrix*, que é uma lista de itens <valor, classe, contador>. Cada item desta lista corresponde a um contador para uma combinação de um valor do domínio do atributo *B* e uma classe do banco de dados. Como para atributos numéricos, a lista do atributo *B* é percorrida e para cada valor $\boldsymbol{v}$ lido, encontra-se a entrada da lista de classe correspondente, que informa a classe $\boldsymbol{c}$ do registro associado ao valor $\boldsymbol{v}$ e a folha da árvore que o contém. A *count matrix* é atualizada incrementando-se o contador associado ao valor $\boldsymbol{v}$ e a classe $\boldsymbol{c}$. Outra diferença em relação à análise de *splits* de atributos numéricos é que o cálculo de *gini index* para todos os possíveis *splits* só é feito após o processamento de toda a lista, quando a *count matrix* está completamente preenchida.

Para atributos categóricos existe um *split* para cada subconjunto dos valores do atributo, com exceção do subconjunto vazio e do próprio conjunto. A geração de todos os subconjuntos dos valores do atributo pode ser uma operação extremamente cara, dependendo da tamanho deste conjunto. Por exemplo, se este tamanho for igual a 20, $2^{20}$ – 2 (1048576) subconjuntos precisam ser gerados param terem seus *splits* correspondentes analisados. Para reduzir a quantidade de conjuntos gerados, o SLIQ usa um algoritmo guloso, que tenta gerar apenas os subconjuntos com mais chances de corresponderem a bons *splits*. Este algoritmo gera todos os subconjuntos exaustivamente caso o tamanho do conjunto seja menor que uma constante fornecida pelo usuário. Caso contrário, o algoritmo monta o subconjunto elemento a elemento. O

elemento adicionado em cada passo é aquele que leva ao melhor valor de *split*. Elementos são adicionados até que não se obtenha mais nenhum ganho neste valor de *split*.

Para exemplificar a avaliação de *splits* de atributos categóricos, assuma que o nó raiz foi particionado usando o *split* IDADE < 27.5. A avaliação de *splits* do atributo categórico TIPO DE CARRO está ilustrada na Figura 2.8.

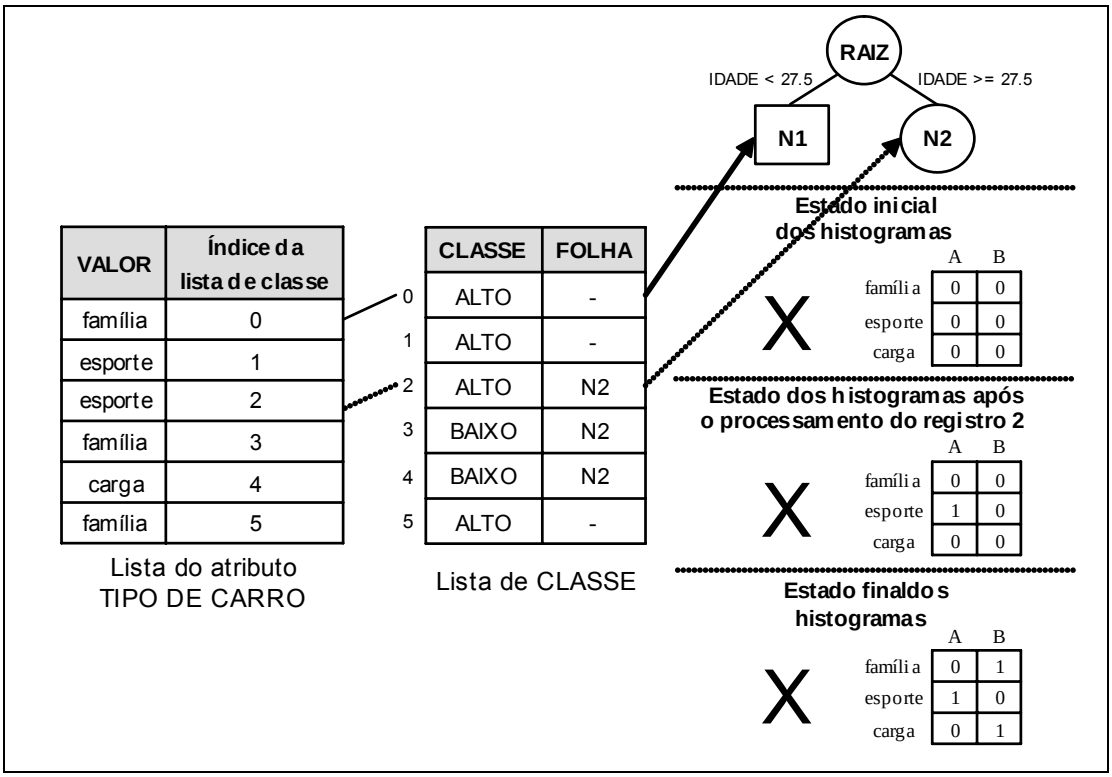


Figura 2.8 – Avaliação de *splits* de um atributo categórico

Inicialmente, os contadores da *count matrix* do nó N2 são inicializados com zero. O nó N1 não tem uma *count matrix* associada, pois todos os seus registros pertencem à classe “ALTO”, ou seja, se transformou em uma **folha pura** da árvore de decisão. O processo se inicia com a leitura do valor “família”, que está associado à entrada 0 da lista de classe. Esta entrada informa que o registro associado a este valor “família” não está contido em nenhuma folha. Isto quer dizer, na realidade, que a folha que o contém

já é uma folha pura e não precisa mais ter *splits* avaliados[5]. O processo, então, segue a partir do próximo valor, “esporte”. O mesmo caso se repete. Para o valor “esporte” seguinte, tem-se a entrada 2 da lista de classe, que indica que a classe do registro associado a ele é “ALTO” e que pertence ao nó N2. A *count matrix* deste nó é então atualizada. O contador associado ao valor “esporte” e a classe “ALTO” é incrementado. Este processo continua até o fim da lista ser alcançado. Após isto, os índices de todos os possíveis *splits* são calculados. Para isto, é utilizado o algoritmo gerador de subconjuntos, e, para cada subconjunto gerado, é calculado o índice do *split* correspondente.

Por exemplo, um possível *split* para este atributo é TIPO DE CARRO $\in$ {família, esporte}. Pelo estado final da *count matrix*, o *gini index* para a partição da esquerda seria:

$$\boldsymbol{Gini(S) = 1 - \sum_j p_j^2} = 1 - ( (1/2)^2 + (1/2)^2 ) = 0,5$$

E o *gini index* da partição da direita:

$$\boldsymbol{Gini(S) = 1 - \sum_j p_j^2} = 1 - ( (1/1)^2 + (0/1)^2 ) = 0$$

O índice do *split* (dado pela equação 2.3) é, então:

$$\textit{índice}_{split}(\text{TIPO DE CARRO} \in \{\text{família, esporte}\}) = (1/3) * (0) + (2/3) * (0,5) = 0,33$$

Este *split* é salvo na folha caso este índice seja menor que o do *split* correntemente armazenado.

As listas de atributos, então, são analisadas uma de cada vez. Isto permite que, enquanto uma lista está sendo processada, as outras fiquem em disco. Um fato importante é que a lista de classe, por conter dados referenciados durante a maior parte do tempo, deve permanecer em memória principal para que o algoritmo não perca eficiência.

Após a análise de *splits* de todos atributos, cada folha não pura da árvore tem armazenado o *split* a ser usado para particioná-la. O processo de criação de novos nós

[5] Para evitar este processamento desnecessário de valores, uma otimização pode ser implementada. Quando uma folha se transforma em uma folha pura, pode-se “limpar” as listas de atributos, excluindo delas todos os valores associados a registros contidos nesta folha. Este processo pode ser caro devido às operações de E/S envolvidas, logo, só deve ser usado quando a economia de esforço computacional que ela irá trazer valer a pena.

é bastante simples. São criados dois nós filhos para cada folha não pura. Para todos os atributos usados em algum *split*, percorre-se sua lista. Para cada valor lido, recupera-se sua entrada correspondente na lista de classe, e a partir dela, a folha *F* que contém seu registro associado. Se o *split* desta folha *F* é baseado no atributo em questão, este *split* é aplicado ao valor sendo processado. Se este valor satisfizer ao *split*, o campo FOLHA da entrada da lista de classe associada a este valor é atualizado com a referência para o filho da esquerda da folha *F*, se este filho da esquerda for uma folha não pura, ou NULO, se for uma folha pura. Caso contrário, ele é atualizado com a referência para o filho da direita de *F*, se este filho for uma folha não pura, ou NULO, se for uma folha pura.

A Figura 2.9 contém um exemplo de atualização da lista de classe.

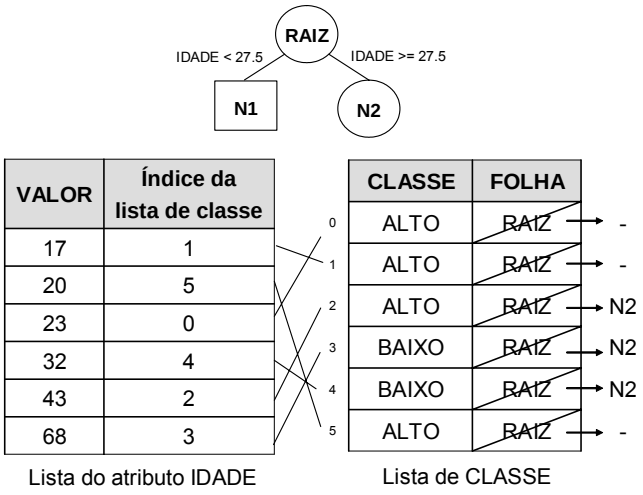


Figura 2.9 – Atualização da lista de classe

Se o nó raiz for particionado com o *split* IDADE < 27.5, seu nó filho da esquerda é uma folha pura (todos os registros da classe “ALTO”) e o da direita, uma folha não pura. A lista de atributo IDADE é percorrida, já que IDADE foi usado pelo *split* da raiz. Inicialmente, o valor 17 é lido, e sua entrada na lista de classe é acessada. O campo FOLHA indica que o registro associado ao valor 17 pertence ao no raiz. Verifica-se que o atributo IDADE foi usado no *split* do nó raiz. Então, este *split*, IDADE < 27.5, é aplicado ao valor 17, que indica que o registro passará a estar associado ao filho da esquerda da raiz, que é uma folha pura. Por isso, o campo folha da lista de classe

associado ao valor 17 é atualizado como NULO. O caso se repete para os valores 20 e 23. Já para os valores 32, 43 e 68, a aplicação do *split* indica que os registros correspondentes a eles devem passar a ser associados ao nó filho da direita da raiz, que é uma folha não pura. Então, o campo FOLHA das entradas da lista de classe associadas a estes valores é atualizado para N2. Neste ponto termina o particionamento do nó raiz, o que dá início a um novo passo de análise de *splits* para os novos nós folhas não puros (neste caso, apenas o nó N2).

O processo de análise de *splits* e particionamento de nós é repetido até que a árvore de decisão tenha todas as suas folhas puras, ou seja, contendo todos os registros de uma mesma classe.

Após a fase de construção da árvore, é realizada a fase de poda. O algoritmo de poda do SLIQ é rápido, simples e leva a árvores de decisão de boa precisão. Ele se baseia no chamado princípio MDL (*Minimum Description Length principle*).

O princípio MDL é um princípio usado para modelagem estatística, onde se busca descobrir características regulares em um conjunto de dados. A idéia básica deste princípio é que toda característica regular encontrada em dado conjunto de dados pode ser usada para comprimi-lo, ou seja, para descrevê-lo usando menos informações do que o necessário para descrevê-lo literalmente. Então, é considerado como o melhor modelo aquele que consegue descrever o conjunto de dados usando a menor quantidade de informações. Esta informação é normalmente medida em número de *bits*.

Mais formalmente, o princípio MDL pode ser definido como:

$$\textbf{custo}(\boldsymbol{M}, \boldsymbol{D}) = \textbf{custo}(\boldsymbol{D} \mid \boldsymbol{M}) + \textbf{custo}(\boldsymbol{M}) \qquad (2.4)$$

Onde custo($D$ | $M$) é o custo em *bits* de descrever os dados $D$ usando um modelo $M$ e custo($M$) é o custo em *bits* de descrever este modelo $M$.

Este princípio é aplicado às árvores de decisão considerando as classes do banco de dados de treinamento como os dados que se deseja descrever, e a árvore de decisão como o modelo que as descrevem. A árvore de decisão que descreve as classes do banco de dados usando a menor quantidade de informações, ou seja, a que tem o menor custo em *bits*, é considerada a melhor. Se a poda de uma subárvore da

árvore de decisão gerada na fase de construção diminui seu custo, ou pelo menos não o aumenta, esta subárvore é podada. Então, este custo reflete a qualidade da árvore. O fato da poda de uma de suas subárvores não aumentar seu custo indica que não haverá perda em sua qualidade.

Para que isto seja implementado, é necessário que se defina uma maneira de medir a quantidade de informações usadas por uma árvore de decisão para descrever as classes do banco de dados. Isto é feito através de um esquema de custos. O esquema de custos usado pelo algoritmo de poda do SLIQ define o custo para descrever as classes do banco de dados e o custo para descrever o modelo.

O custo para descrever as classes do banco de dados é considerado a soma de erros de classificação da árvore de decisão. Um erro de classificação acontece quando a árvore de decisão considera o registro como sendo de uma classe diferente da sua classe original.

O custo para descrever o modelo é igual à soma do custo para descrever a árvore de decisão com o custo para descrever os *splits* de cada nó interno da árvore. O custo de descrever a árvore de decisão é a soma dos custos para descrever seus nós.

O custo $L(t)$ para descrever um nó $t$ depende da estrutura da árvore. Se um nó só puder ter dois ou nenhum filho, o custo de cada nó é igual a 1 bit (2 possibilidades); se puder ter dois, um à direita, um à esquerda ou nenhum, 2 bits são necessários (4 possibilidades); Outra possibilidade é considerar apenas nós internos, que tem um filho à esquerda, um filho à direita ou os dois, o que necessita de $\log_2 3$ bits (3 possibilidades).

O custo $L_{split}$ para descrever *splits* depende do tipo do seu atributo. Se for numérico (*split* da forma $A < \boldsymbol{v}$), o custo é definido como 1 bit. Se for categórico (*split* da forma $A \in \boldsymbol{S}$), o custo é calculado em dois passos: primeiro, conta-se o número de *splits* usando este atributo na árvore, $n$, e depois, o custo é calculado como $\log_2 n$.

Esta estratégia de poda avalia o custo de cada subárvore da árvore de decisão para determinar se esta deve ser convertida em uma folha, se apenas sua subárvore da esquerda deve ser podada, se apenas a sua subárvore da direita deve ser podada, ou se deve permanecer intacta. O custo $C(n)$ de uma subárvore de raiz $t$ para cada uma destas possibilidades é:

$$C_{folha}(t) = L(t) + Erros_t \quad \text{se } t \text{ é uma folha} \quad (2.5)$$

$$C_{ambas}(t) = L(t) + L_{split} + C(t_1) + C(t_2) \quad \text{se } t \text{ tem os dois filhos} \quad (2.6)$$

$$C_{esquerda}(t) = L(t) + L_{split} + C(t_1) + C'(t_2) \quad \text{se } t_1 \text{ é o único filho de } t \quad (2.7)$$

$$C_{direita}(t) = L(t) + L_{split} + C'(t_1) + C(t_2) \quad \text{se } t_2 \text{ é o único filho de } t \quad (2.8)$$

Onde $Erros_t$ é a quantidade de registros associados ao nó $t$ que não pertencem à sua classe majoritária e $C'(t)$ é o custo associado aos registros da subárvore podada quando a poda de apenas uma das subárvores ocorre. Este custo é calculado de acordo com as estatísticas presentes em $t$.

Com base neste esquema, o SLIQ propõe três versões para o algoritmo de poda:

- ***FULL***: Deixa a subárvore com raiz $t$ intacta ou poda suas duas subárvores filhas. Então, apenas as opções (2.5) e (2.6) são consideradas. Se $C_{folha}(t)$ é menor que $C_{ambas}(t)$, o nó $t$ é convertido em uma folha. (nesta versão, o custo de cada nó é de 1 bit);
- ***PARTIAL***: Possibilita a poda de apenas uma das subárvores, portanto, considera as quatro opções. Cada nó é convertido na opção de menor custo (nesta versão, o custo de cada nó é de 2 bits);
- ***HYBRID***: Faz a poda em duas fases. Primeiro, a versão ***FULL*** é acionada para reduzir o tamanho da árvore, e depois, considera as opções (2.6), (2.7) e (2.8);

### 2.3.2 – SPRINT

O SLIQ, apesar de conseguir tratar bancos de dados muito maiores que os algoritmos clássicos de geração de árvores de decisão, não conseguiu atingir o objetivo da escalabilidade. Isto porque a principal estrutura de dados que ele usa, a lista de classe, precisa estar em memória principal durante toda a fase de construção da árvore, pois toda operação realizada depende de informações que ela contém. Como existe uma entrada na lista de classe para cada registro do banco de dados de treinamento, o seu tamanho é proporcional ao tamanho do banco de dados. Isto faz com que para um determinado tamanho de banco, a lista de classe se torne maior que a memória principal, o que deteriora a performance do algoritmo.

O SPRINT (Scalable PaRallelizable INduction of decision Trees) [54] é o successor direto do SLIQ. Sua principal contribuição foi eliminar a dependência a uma estrutura de dados central, o que acontecia com o SLIQ. Isto é feito replicando a informação da classe do registro em cada lista de atributo e associando a cada folha da árvore o conjunto de listas de atributos que corresponde à partição do banco de dados contida no folha.

Uma lista de atributo no SPRINT tem a forma **<valor, classe, rid>**, onde **classe** é a classe do registro associado ao **valor**, e **rid** é o identificador do registro. Após a criação destas listas, elas são associadas ao nó raiz. As que correspondem a atributos numéricos são pré-ordenadas, assim como no SLIQ. A Figura 2.10 ilustra a criação de listas de atributos para o banco de dados da Figura 2.4.

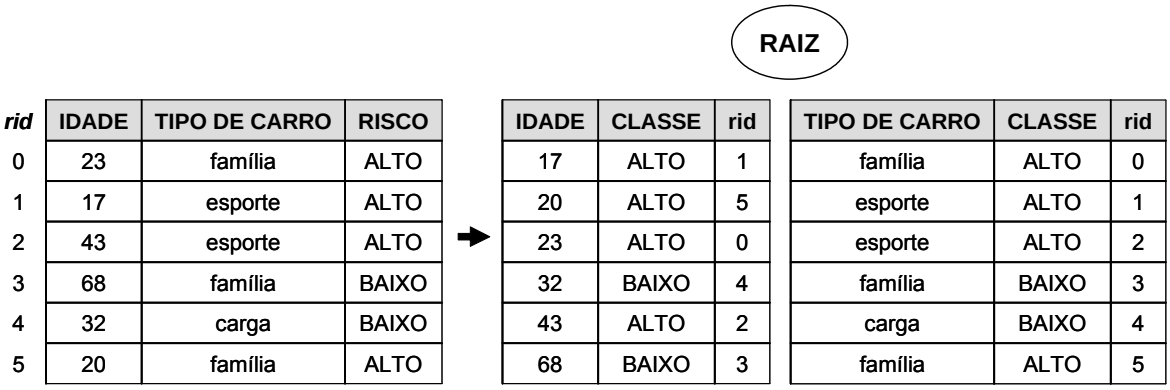


Figura 2.10 – Criação de listas de atributos no SPRINT

A análise de *splits* é feita da mesma forma que no SLIQ, com as exceções de que cada folha (não pura) é analisada por vez[6] e que a classe do registro não precisa ser recuperada da lista de classe. A estrutura dos histogramas e a maneira como eles são atualizados e utilizados para o calculo de *gini index* é exatamente a mesma. A diferença está no particionamento das folhas após a análise de *splits*. Quando uma folha é particionada, dois novos nós são criados como seus filhos. As suas listas de atributos também são particionadas de forma que as partes das listas que correspondem aos registros que obedecem ao *split* se transformam nas listas de atributos do filho da esquerda, enquanto que as partes que não obedecem se transformam nas listas de atributos do filho da direita.

[6] Isto faz com que seja indiferente a construção da árvore em largura ou em profundidade.

A lista do atributo usado neste *split* é a primeira a ser particionada, aplicando-se o *split* a cada um de seus valores. Este particionamento ainda gera informações a serem usadas para particionar as outras listas de atributos. Estas informações são da forma **<rid, destino>**, e indicam o destino, esquerda ou direita, do registro identificado por **rid**. Uma estrutura de dados auxiliar é necessária para armazenar estas informações, como por exemplo, uma tabela hash. O particionamento de cada uma das outras listas é feito recuperando-se desta estrutura de dados o destino de cada registro a partir de seu **rid**.

A Figura 2.11 ilustra o particionamento do nó raiz.

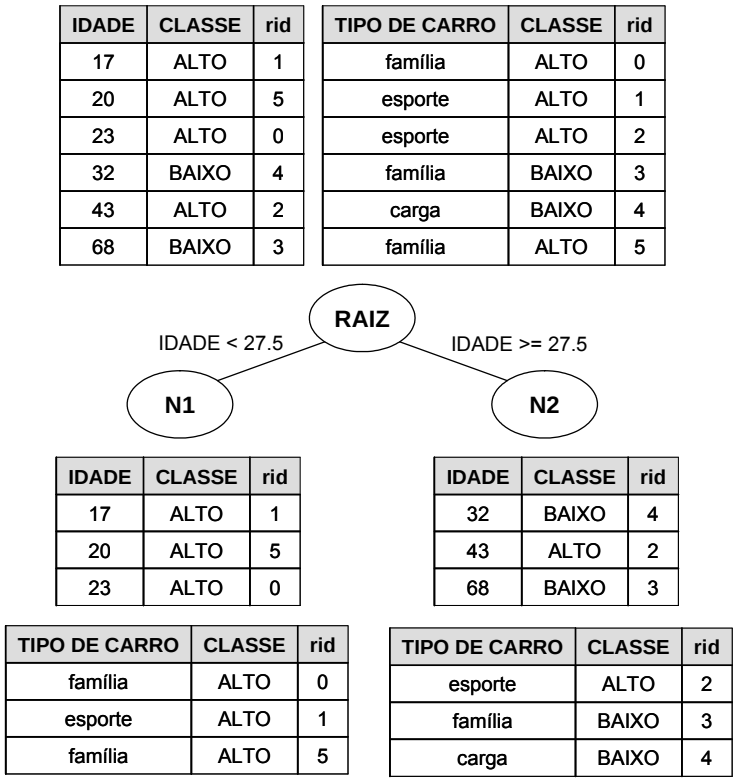


Figura 2.11 – Particionamento de uma folha no SPRINT

A lista do atributo IDADE é particionada primeiro, pois IDADE é o atributo do *split*. Após este particionamento, tem-se a informação de que os registros de rid 0, 1 e 5 foram associados ao filho da esquerda e os outros, ao filho da direita. Esta informação é utilizada para o particionamento da lista do atributo TIPO DE CARRO.

O processo de análise de *splits* para o particionamento de folhas não puras ocorre até que todas as folhas se tornem puras.

Os algoritmos para geração de subconjuntos e para poda são os mesmos usados pelo SLIQ.

Desta forma, não existe uma estrutura de dados que precise permanecer em memória durante a execução do algoritmo, devido ao fato de que cada lista de atributo contém todas as informações necessárias para sua análise. Isto tem como conseqüência um maior custo de operações de E/S que no SLIQ, pois o registro das listas de atributos é maior, mas em compensação, permite que o SPRINT consiga gerar árvores de decisão para qualquer tamanho de banco de dados de treinamento eficientemente.

Capítulo 3

# O Módulo de Classificação da Ferramenta MIDAS-UFF

Este capítulo apresenta o módulo de Classificação proposto para integrar a ferramenta MIDAS-UFF. Sua funcionalidade é fornecida na Seção 3.1. Detalhes sobre a interatividade com o usuário, como escolha de fonte de dados de entrada, escolha de algoritmo a ser utilizado e sobre os dados de saída, estão descritos na Seção 3.2.

## 3.1 – Funcionalidade

O módulo de Classificação é apenas um dos módulos planejados para integrar a ferramenta MIDAS-UFF. Outras tarefas de Mineração de Dados deverão estar disponíveis em versões futuras.

Para facilitar a realização destas tarefas pelo usuário, é proposta uma interface gráfica intuitiva. Então, na tela inicial da ferramenta, ilustrada na Figura 3.1, as tarefas de Mineração de Dados são oferecidas para que o usuário simplesmente escolha a que deseja realizar sobre seus dados.

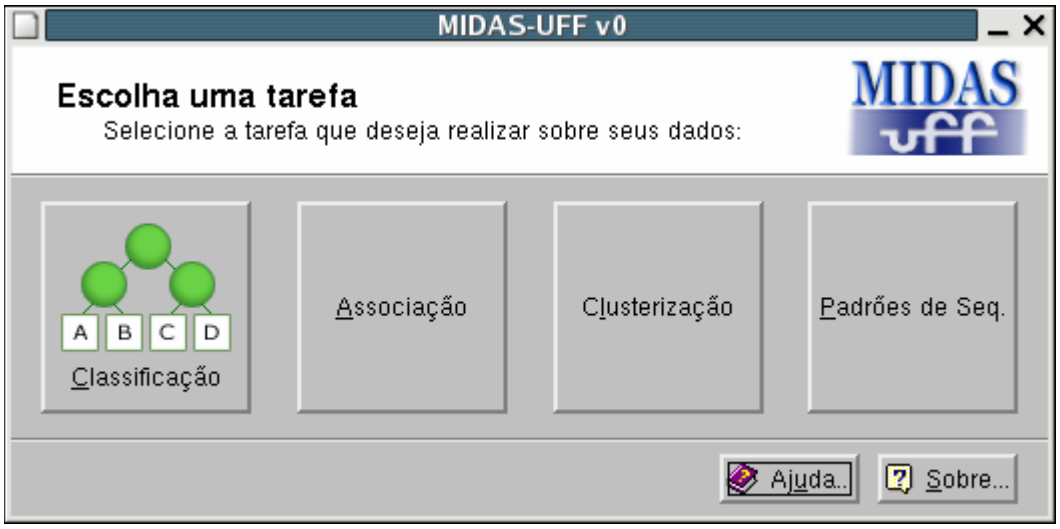


Figura 3.1 – Janela principal da ferramenta MIDAS-UFF

A escolha da tarefa de Classificação leva o usuário à tela correspondente à interface desta tarefa, que está ilustrada na Figura 3.2.

Nesta tela, o usuário deve escolher o banco de dados que deve ser considerado para geração do modelo, o algoritmo a ser utilizado e seus parâmetros de execução, a maneira como o modelo de Classificação gerado deve ser avaliado e como devem ser tratados valores nulos presentes no banco de dados escolhido. Realizados estes passos, o processo de geração do modelo é iniciado, como ilustra a Figura 3.3. Este processo pode ser cancelado a qualquer momento pelo usuário, e, quando chega ao fim, a tela de apresentação de resultados, ilustrada na Figura 3.4, é mostrada.


Figura 3.2 – Interface do módulo de Classificação

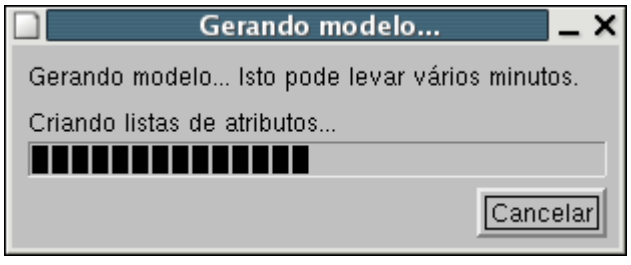


Figura 3.3 – Estado do processo de geração do modelo de Classificação

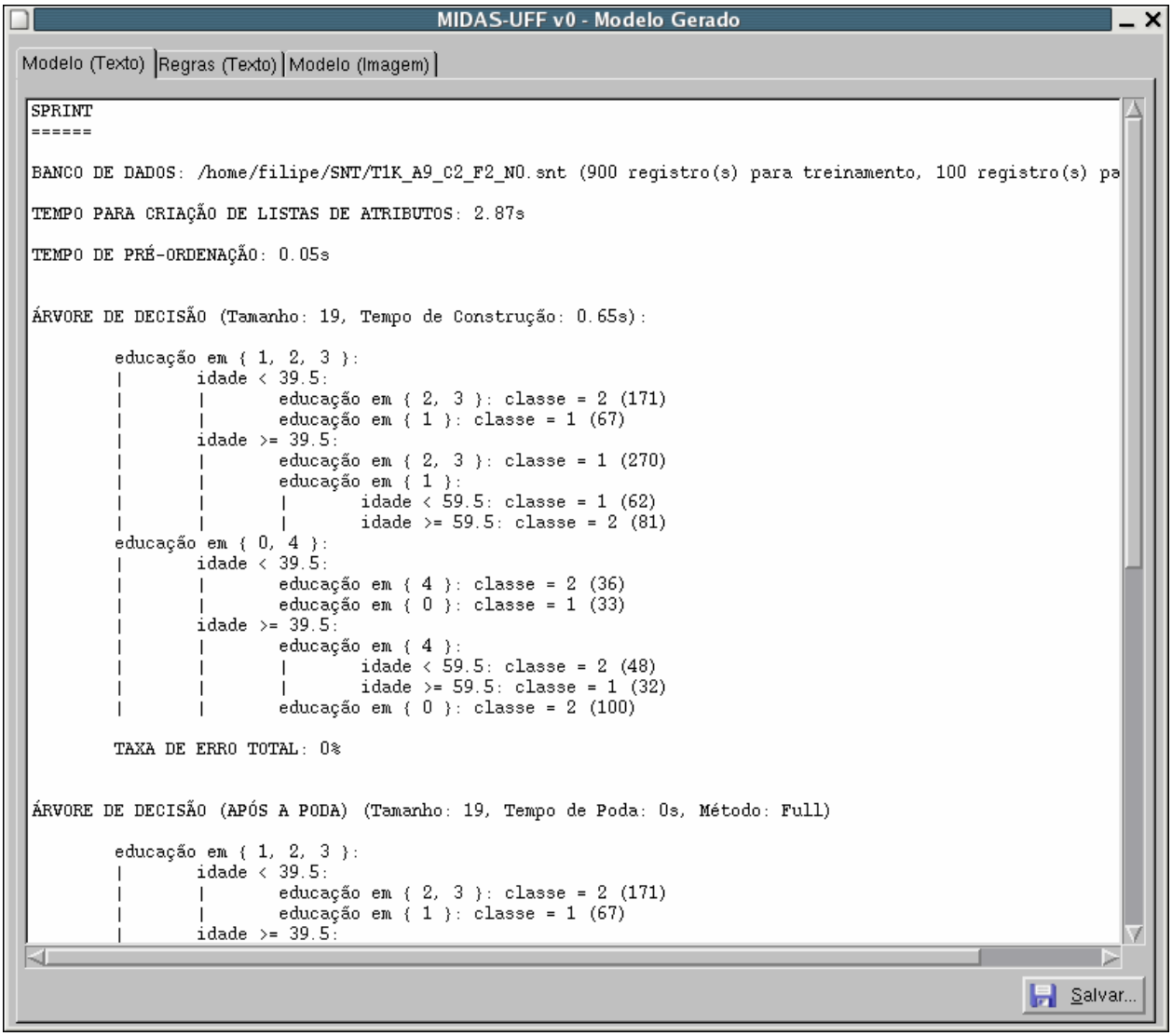


Figura 3.4 – Tela de apresentação de resultados

Nesta tela, o modelo de Classificação extraído pode ser visualizado em diferentes formatos, que podem ser salvos para posterior utilização.

## 3.2 – Detalhes de interatividade com o usuário

As telas apresentadas na seção anterior possuem itens que merecem uma descrição mais detalhada. A tela do módulo de Classificação (Figura 3.2), por exemplo, é composta de quatro grupos de opções, que são: **Banco de dados para treinamento**,

**Algoritmo**, **Avaliação do modelo gerado** e tratamento de **Valores não informados**. Os detalhes de cada um destes grupos são abordados a seguir.

### 3.2.1 – Banco de dados para treinamento

O usuário precisa inicialmente especificar qual banco de dados será considerado para geração do modelo (treinamento). Para isso, ao clicar no botão “Abrir”, uma tela contendo os tipos de fontes de dados disponíveis é mostrada. Esta tela está ilustrada na Figura 3.5.


Figura 3.5 – Tela de escolha de tipo de fonte de dados

Nesta primeira versão da ferramenta, apenas os bancos sintéticos para classificação foram considerados.

#### 3.2.1.1 – Bancos sintéticos

O desempenho de algoritmos que realizam a tarefa de Classificação é sensível a características do banco de dados de treinamento. Para que esta sensibilidade pudesse ser avaliada de uma maneira controlada, decidiu-se integrar o tratamento de bancos sintéticos à ferramenta. A geração de bancos sintéticos é realizada utilizando-se o gerador proposto pelo projeto QUEST[7] da IBM. Registros destes bancos correspondem

[7] http://www.almaden.ibm.com/software/quest/Resources/index.shtml

a dados de pessoas e são compostos por, no mínimo, nove atributos, sendo três categóricos e seis numéricos, como ilustra a Tabela 3.1.

Tabela 3.1 – Descrição de atributos de bancos de dados sintéticos

| Atributo | Descrição | Domínio |
|---|---|---|
| salário | salário anual | entre 20k e 150k |
| comissão | comissão adicionada ao salário | 0, se salário ≥ 75k<br>entre 10k e 75k, caso contrário |
| idade | idade | entre 20 e 80 |
| educação | nível de educação | {0, 1, ..., 4} |
| carro | tipo de carro que a pessoa possui | {0, 1, ..., 20} |
| cep | código postal | {0, 1, ..., 9} |
| valor_casa | valor da casa que a pessoa possui | entre 0,5*φ*100k e 1,5*φ*100k, onde φ ∈ {0, 1, ..., 9} está relacionado ao cep |
| yo | tempo, em anos, que a pessoa possui a casa (*years owned*) | entre 1 e 30 |
| empréstimo | total de empréstimos realizados | entre 0 e 500k |

Se o usuário optar por utilizar um banco de dados sintético como banco de treinamento, a tela ilustrada na Figura 3.6 é mostrada para que o banco sintético a ser considerado seja especificado.

O usuário pode optar por criar um novo banco ou utilizar um já existente. Caso deseje criar um novo banco, parâmetros para a geração do banco precisam ser informados. Eles são: o número de tuplas, número de atributos, número de classes, *random seed* (base para geração de valores aleatórios), função de classificação, porcentagens de ruído e perturbação e a distribuição das classes.

A função de classificação determina a classe do registro em função dos valores dos seus atributos. Elas variam em termos de quais, quantos e como os atributos são considerados. Quanto maior a complexidade da função, maior a dificuldade de geração de um modelo de Classificação para o banco de dados correspondente. A descrição completa destas funções pode ser encontrada em [4].

A porcentagem de ruído indica a quantidade de registros que terão a classe diferente da indicada pela função de classificação. A porcentagem de perturbação é usada para modificar os valores de atributos numéricos de um registro após a sua geração. Sua função é simular a disjunção não perfeita entre grupos de registros de mesma classe.

A distribuição de classes também pode ser definida. O parâmetro g0 indica a fração de registros do banco de dados que deve ser da classe 1. Similarmente, g1 e g2 indicam a fração de registros do banco de dados que devem ser da classe 2 e 3, respectivamente. Para bancos de dados com mais de 3 possíveis classes, não é possível especificar a distribuição de classes.

O estado de processamento de geração de um banco de dados sintético é informado em uma tela especial (Figura 3.7a). Esta geração pode ser interrompida a qualquer momento, cancelando a criação do banco.

Caso se deseje utilizar um banco de dados sintético já existente, o usuário precisa informar o caminho do arquivo que contém os dados. Para garantir que o arquivo informado corresponde a um banco de dados sintético, um procedimento de validação é executado (Figura 3.7b). Este procedimento, similarmente ao processo de geração de um banco de dados sintético, pode ser interrompido a qualquer momento, o que cancela a abertura do banco.

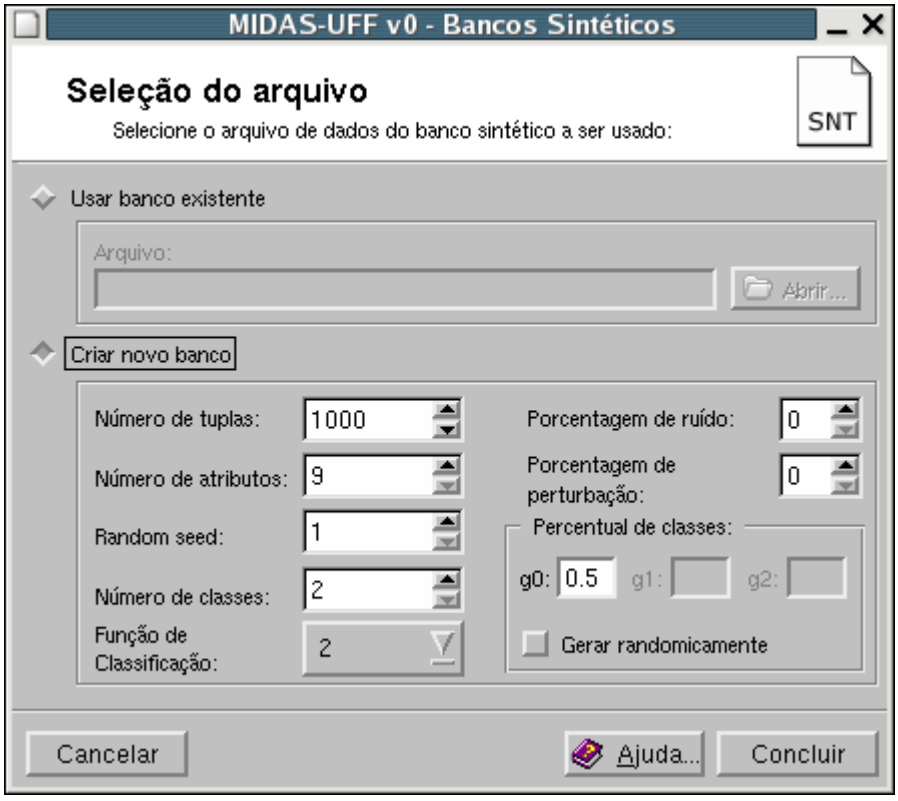


Figura 3.6 – Abertura de bancos sintéticos


(a) (b)

Figura 3.7 – (a) Tela de indicação de estado do processamento para geração de um novo banco de dados sintético (b) Tela de indicação de estado do processamento para validação de um banco já existente

### 3.2.2 – Algoritmos

O usuário pode escolher qual algoritmo gerador de modelos de Classificação será utilizado e, para o algoritmo escolhido, informar seus parâmetros. Dois algoritmos estão disponíveis nesta versão da ferramenta, o SLIQ e o SPRINT.

Tanto o SLIQ quanto o SPRINT esperam receber dois parâmetros, que são: o método de poda da árvore, que tem as opções *full*, *partial* e *hybrid*, e o método de geração de subconjuntos, que tem as opções exaustiva, gulosa e GRASP (não disponível nesta primeira versão da ferramenta). A análise exaustiva de subconjuntos é uma operação computacionalmente cara quando o tamanho do conjunto a ser analisado é grande. As opções gulosa e GRASP são melhores para bancos de dados contendo atributos categóricos cujos domínios possuem muitos valores. Quando uma destas duas opções é escolhida, a geração de subconjuntos ainda é feita exaustivamente para conjuntos de tamanho menor que um valor máximo, devido à análise exaustiva ser sempre mais eficaz e ser rápida para conjuntos pequenos. Este valor máximo deve ser especificado pelo usuário na tela principal do módulo de Classificação, como visto na Figura 3.2.

O uso do SLIQ é indicado quando o banco de dados é menor que a memória principal do sistema. Caso contrário, o SPRINT é uma melhor opção (como visto no Capítulo 2).

### 3.2.3 – Avaliação do modelo

Para que se tenha uma medida da qualidade do modelo de Classificação gerado, é possível que o usuário informe como este modelo deve ser avaliado. Esta avaliação pode estar desabilitada, pode considerar uma porcentagem dos registros do banco de dados de treinamento, pode considerar um número específico de registros do banco de dados de treinamento, ou pode ainda utilizar um banco de dados específico para avaliação. Esta avaliação fornece informações como o total de erros de classificação do modelo para os registros fornecidos para avaliação e uma matriz de confusão que mostra como os erros de classificação foram distribuídos. A estrutura desta matriz para um banco de dados com duas classes, *A* e *B*, está ilustrada na Figura 3.8. Cada elemento $a_{ij}$ desta matriz corresponde ao número de registros da classe *i* que foram classificados pelo modelo como *j*. Portanto, o modelo é considerado bom quando os elementos da diagonal principal da matriz são altos, enquanto os outros são próximos ou, preferencialmente, iguais a zero.

| | | **Classe prevista** | |
|---|---|---|---|
| | | **A** | **B** |
| **Classe real** | **A** | Quantidade de registros da classe A classificados como A | Quantidade de registros da classe A classificados como B |
| | **B** | Quantidade de registros da classe B classificados como A | Quantidade de registros da classe B classificados como B |

Figura 3.8 – Estrutura de uma matriz de confusão para um banco de duas classes

Uma outra medida de avaliação é exibida para modelos correspondentes a árvores de decisão. Esta medida é a taxa de erro total relacionada ao banco de dados de treinamento. Cada registro do banco de dados de treinamento é associado a uma folha da árvore de decisão. Cada folha possui uma classe associada. A quantidade de registros associados a esta folha que não pertencem a esta classe sobre a quantidade total de registros associados a esta folha corresponde à taxa de erro desta folha. A taxa de erro total da árvore de decisão, então, é definida como a soma ponderada das taxas

de erro de cada folha, considerando-se a probabilidade, associada a cada folha, de um registro ser representado por ela [9]. Esta probabilidade corresponde ao total de registros associados à folha sobre o total de registros da base de treinamento.

### 3.2.4 – Valores não informados

Se o banco de dados contém registros com valores de atributos não informados (nulos), o gerador do modelo de Classificação precisa saber como deve proceder. Em um gerador de árvores de decisão, por exemplo, em algum momento da geração do modelo um *split* da forma $A < v$ pode ser considerado para particionar o banco de dados. Caso um registro não tenha o valor do atributo $A$ informado, é preciso tomar a decisão de como tratar este registro. As opções de tratamento são **descartar o registro**, o que não é considerada uma boa estratégia, pois quanto mais informações forem consideradas pelo gerador maior a probabilidade de se obter um modelo de boa qualidade, considerar a **média dos valores deste atributo entre todos os registros**, ou a **média dos valores deste atributo entre os registros que pertencem à mesma classe que este registro**. Para cada caso, uma pode apresentar resultados melhores que as outras.

### 3.2.5 – Formatos de saída

Após a conclusão do processo de geração do modelo, as informações extraídas são exibidas ao usuário. Os algoritmos disponíveis nesta versão, SLIQ e SPRINT, fornecem dois formatos de saída de informações, um contendo a árvore de decisão em si (em modo texto) e o outro contendo as regras de classificação extraídas desta árvore.

#### 3.2.5.1 – Árvore de decisão

Para ilustrar o formatos de saída de informações foi gerado um modelo para um banco de dados sintético com 200 tuplas, 9 atributos, 2 classes e com a função de classificação 2 e com 1% de ruído. A saída contendo a árvore de decisão para este banco está ilustrada na Figura 3.9.

```
SPRINT
======

BANCO DE DADOS: /home/filipe/SNT/T200_A9_C2_F2_N1.snt (190 registro(s) para treinamento, 10
registro(s) para avaliação)

TEMPO PARA CRIAÇÃO DE LISTAS DE ATRIBUTOS: 0.12s

TEMPO DE PRÉ-ORDENAÇÃO: 0.02s


ÁRVORE DE DECISÃO (Tamanho: 23, Tempo de Construção: 0.69s):

   educação em { 1, 2, 3 }:
   |    idade < 39.5:
   |    |       educação em { 2, 3 }: classe = 2 (42)
   |    |       educação em { 1 }: classe = 1 (17)
   |    idade >= 39.5:
   |    |       educação em { 2, 3 }: classe = 1 (55)
   |    |       educação em { 1 }:
   |    |       |       idade < 57.5: classe = 1 (10)
   |    |       |       idade >= 57.5: classe = 2 (15)
   educação em { 0, 4 }:
   |    carro em { 0, 10, 14 }:
   |    |       cep em { 4 }: classe = 2 (2)
   |    |       cep em { 0, 1, 2, 3, 5, 6, 7, 8, 9 }: classe = 1 (6)
   |    carro em { 1, 11, 12, 13, 15, 16, 17, 18, 19, 2, 3, 4, 5, 6, 7, 8, 9 }:
   |    |       idade < 67.5:
   |    |       |       carro em { 4 }:
   |    |       |       |       salario < 10703.5: classe = 2 (3)
   |    |       |       |       salario >= 10703.5: classe = 1 (1)
   |    |       |       carro em {1,11,12,13,15,16,17,18,19,2,3,5,6,7,8,9}: classe = 2 (29)
   |    |       idade >= 67.5:
   |    |       |       educação em { 4 }: classe = 1 (4)
   |    |       |       educação em { 0 }: classe = 2 (6)

   TAXA DE ERRO TOTAL: 0%

ÁRVORE DE DECISÃO (APÓS A PODA) (Tamanho: 11, Tempo de Poda: 0s, Método: Full)

   educação em { 1, 2, 3 }:
   |    idade < 39.5:
   |    |       educação em { 2, 3 }: classe = 2 (42)
   |    |       educação em { 1 }: classe = 1 (17)
   |    idade >= 39.5:
   |    |       educação em { 2, 3 }: classe = 1 (55)
   |    |       educação em { 1 }:
   |    |       |       idade < 57.5: classe = 1 (10)
   |    |       |       idade >= 57.5: classe = 2 (15)
   educação em { 0, 4 }: classe = 2 (51/11)

   TAXA DE ERRO TOTAL: 5.789%

AVALIAÇÃO DO MODELO (Tempo para avaliação: 0s):
-------------------

        Registros para avaliação: 10
        Erros: 0 (0%)

        Matriz de Confusão:
        ------------------

                a b
                7 0 a : 1
                0 3 b : 2

TEMPO TOTAL DE GERAÇÃO DO MODELO: 0.83s.
```

Figura 3.9 – Formato de exibição do modelo

A parte inicial deste documento informa o algoritmo usado para a geração do modelo. Neste exemplo o SPRINT foi utilizado. Após isto, o banco de dados de treinamento utilizado é informado, além das quantidades de registros usados para treinamento e avaliação do modelo.

Algumas informações sobre a execução do algoritmo vêm a seguir. Para o SPRINT estas informações são o tempo para criação de listas de atributos e o tempo de pré-ordenação das listas de atributos numéricos.

Em seguida, a árvore de decisão original e a resultante da fase de poda são exibidas, além de dados como seu tamanho, tempo de construção/poda e taxa de erro em relação ao banco de treinamento.

A parte final contém informações sobre a avaliação do modelo. O total de registros para avaliação e o total de erros de classificação (total destes registros que eram de classe diferente da prevista pelo modelo) são exibidos. A seguir, é mostrada a matriz de confusão.

### 3.2.5.2 – Regras de classificação

O outro formato de saída de informações contém regras de classificação extraídas da árvore de decisão gerada. Estas regras têm a forma $X_1 \wedge X_2 \wedge ... \wedge X_n \rightarrow Y$, onde $X_i$, para $1 \leq i \leq n$, corresponde ao critério de *split* do nó *i* entre os *n* nós internos pertencentes ao caminho do nó raiz até uma das folhas da árvore de decisão, e Y corresponde a umas das classes possíveis do banco de dados e que está associada a esta folha.

A Figura 3.9 contém parte deste documento gerado para o caso do exemplo anterior.

A parte inicial contém o algoritmo utilizado e informações sobre o banco de dados de treinamento. As regras vêm a seguir, com a informação de quantos registros do banco de treinamento foram alcançados por ela. Caso tenham sido fornecidos registros para avaliação, o resultado desta é exibido para cada regra. Este resultado indica a quantidade destes registros que satisfizeram as condições da regra, e o erro desta, que corresponde à quantidade dentre estes registros que eram de classe diferente da prevista.

```
SPRINT
======

BANCO DE DADOS: /home/filipe/SNT/T200_A9_C2_F2_N1.snt (190 registro(s) para
treinamento, 10 registro(s) para avaliação)

REGRAS DE CLASSIFICAÇÃO:
-----------------------

       === Regra 1 ===

       SE
       (educação em { 1, 2, 3 }) E
       (idade < 39.5) E
       (educação em { 2, 3 })
       ENTÃO classe = 2 (42)

       Avaliação:
       ---------
              Quantidade de registros alcançados: 0
              Erros: 0 (0%)

       === Regra 2 ===

       SE
       (educação em { 1, 2, 3 }) E
       (idade < 39.5) E
       (educação em { 1 })
       ENTÃO classe = 1 (17)

       Avaliação:
       ---------
              Quantidade de registros alcançados: 1
              Erros: 0 (0%)

       === Regra 3 ===

       SE
       (educação em { 1, 2, 3 }) E
       (idade >= 39.5) E
       (educação em { 2, 3 })
       ENTÃO classe = 1 (55)

       Avaliação:
       ---------
              Quantidade de registros alcançados: 4
              Erros: 0 (0%)

                                          .
                                          .
                                          .
```

Figura 3.10 – Formato de exibição de regras de classificação

Capítulo 4

# Detalhes de Implementação

Este capítulo descreve a estrutura interna da primeira versão da ferramenta MIDAS-UFF, com ênfase na estrutura do módulo de Classificação e nos algoritmos que o integram. A Seção 4.1 fornece uma visão geral da arquitetura da ferramenta. A estrutura do módulo de Classificação é especificamente descrita na Seção 4.2, onde também é detalhada a implementação dos algoritmos disponíveis neste módulo. A Seção 4.3 contém resultados experimentais.

## 4.1 – Arquitetura básica da ferramenta MIDAS-UFF

A ferramenta MIDAS-UFF foi planejada para ser útil a comunidade científica relacionada à linha de pesquisa de Mineração de Dados, além de poder ser utilizada em projetos reais de descoberta de conhecimento. Optou-se por implementá-la na plataforma Linux, largamente encontradas em instituições acadêmicas.

A ferramenta foi desenvolvida com a linguagem de programação C++, que suporta o paradigma da programação orientada a objetos, além de ser uma das linguagens mais utilizadas para o desenvolvimento de sistemas Linux. Para facilitar o desenvolvimento da interface gráfica, o ambiente de desenvolvimento Kylix 3 Open Edition, da Borland Software Corp., foi utilizado.

A arquitetura básica da ferramenta está ilustrada na Figura 4.1.

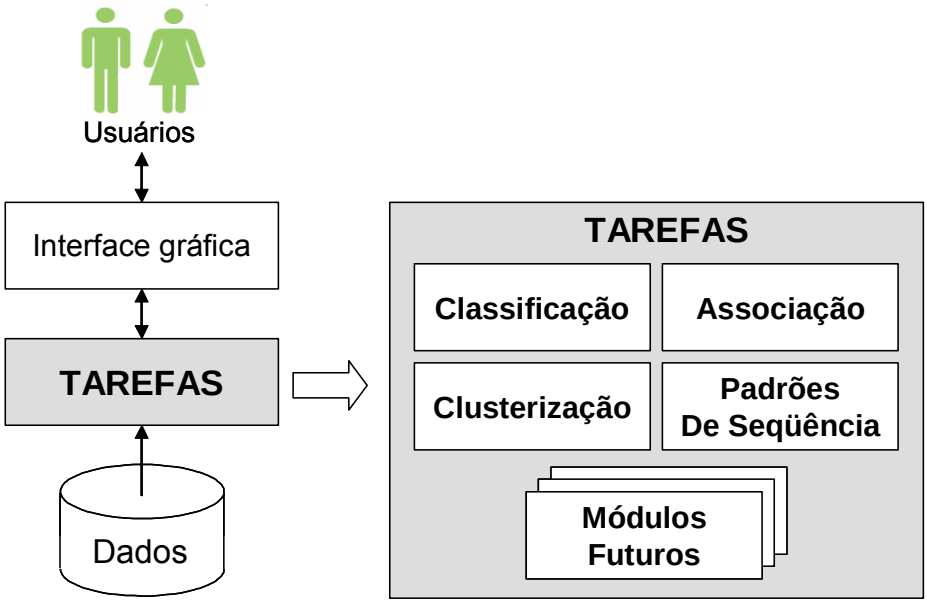


Figura 4.1 – Arquitetura básica da ferramenta MIDAS-UFF

Através de uma interface gráfica simples e intuitiva, usuários escolhem a tarefa que desejam realizar sobre seus dados e visualizam as informações extraídas por esta. Cada uma das principais tarefas de Mineração de Dados deverá corresponder a um dos módulos da ferramenta.

Esta arquitetura básica conceitual é implementada através do diagrama de classes ilustrado na Figura 4.2[8].

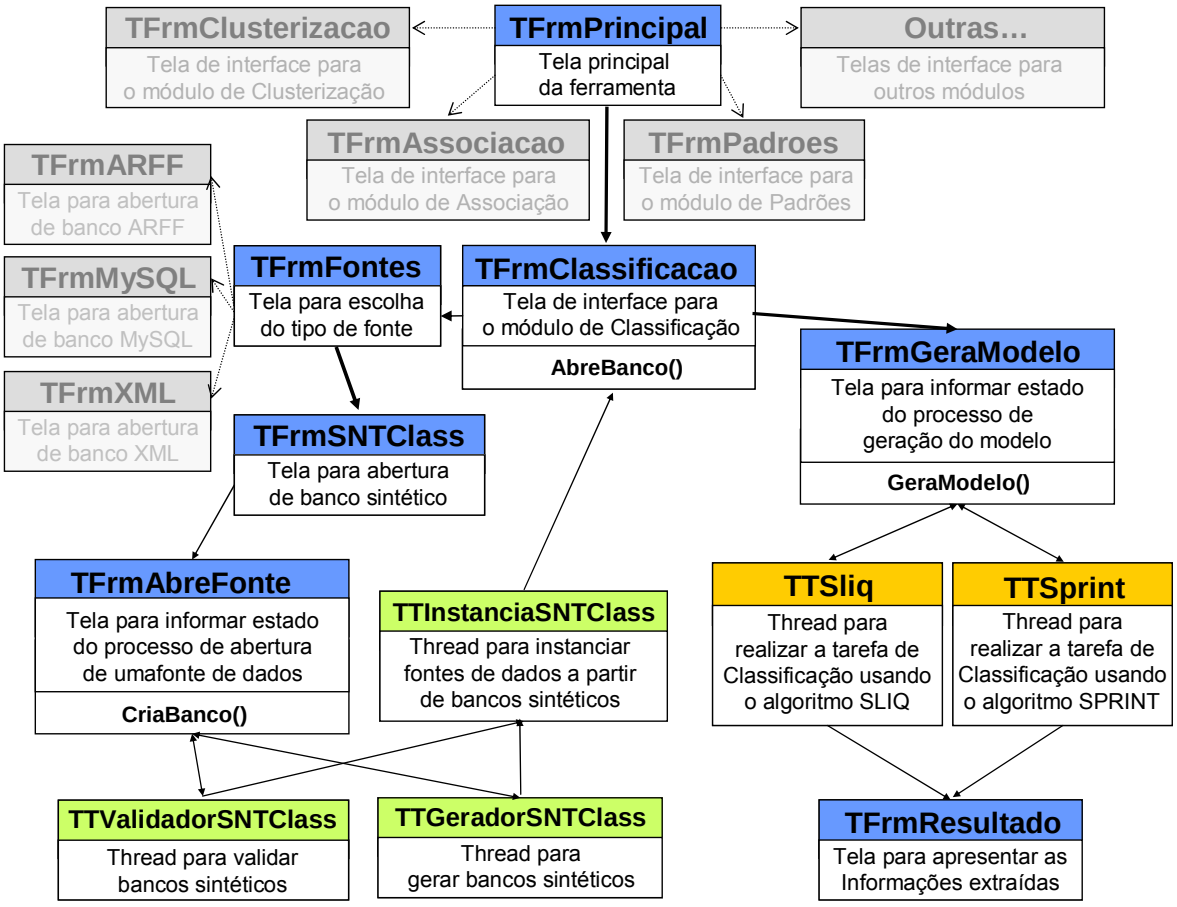


Figura 4.2 – Diagrama de classes da primeira versão da ferramenta MIDAS-UFF

Cada caixa deste diagrama corresponde a uma das classes da ferramenta. Elas contêm seu nome no topo, uma simples descrição de sua função e, para algumas, os métodos principais. As setas indicam algum tipo de interação entre elas. As classes **TFrmClusterizacao**, **TFrmAssociacao**, **TFrmPadroes**, **TFrmARFF**, **TFrmMySQL** e **TFrmXML** não estão ainda implementadas, as **TFrmPrincipal**, **TFrmFontes**, **TFrmClassificacao**, **TFrmSNTClass**, **TFrmGeraModelo**, **TFrmAbreFonte** e **TFrmResultado** representam telas da ferramenta, as **TInstanciaSNTClass**, **TTValidadosSNTClass** e **TTGeradorSNTClass** representam classes envolvidas com o

[8] Os diagramas de classes apresentados nesta seção seguem uma versão adaptada da notação UML (Unified Modeling Language - http://www.rational.com/uml/)

acesso a bancos de dados, enquanto que as classes **TTSliq** e **TTSprint** correspondem a classes envolvidas com realização da tarefa de Classificação. A classe TFrmPrincipal corresponde à tela principal da ferramenta (Figura 3.1). Sua função é apresentar ao usuário as tarefas que estão disponíveis para utilização e levar o usuário à tela de interface do módulo da tarefa escolhida.

Todos os módulos da ferramenta realizam tarefas a partir de dados fornecidos como entrada. Diferentes tipos de fontes de dados, como bancos de dados MySQL, arquivos XML, arquivos ARFF, bancos sintéticos, entre outras, podem ser consideradas como possíveis fontes de dados a serem tratadas em versões futuras da ferramenta. Para facilitar esta possível integração destas fontes de dados, decidiu-se por implementar uma classe (**TFonteDeDados**) para representar uma fonte de dados em geral, fornecendo métodos em que os módulos que realizam tarefas podem se basear para extrair todos os dados necessários para sua realização. A integração de uma nova fonte de dados à ferramenta pode ser feita simplesmente através de uma nova classe descendente desta que implemente estes métodos básicos[9]. Detalhes específicos sobre esta fonte de dados devem ser tratados na implementação destes métodos.

A estrutura de fontes de dados considerada pela ferramenta está ilustrada na Figura 4.3.

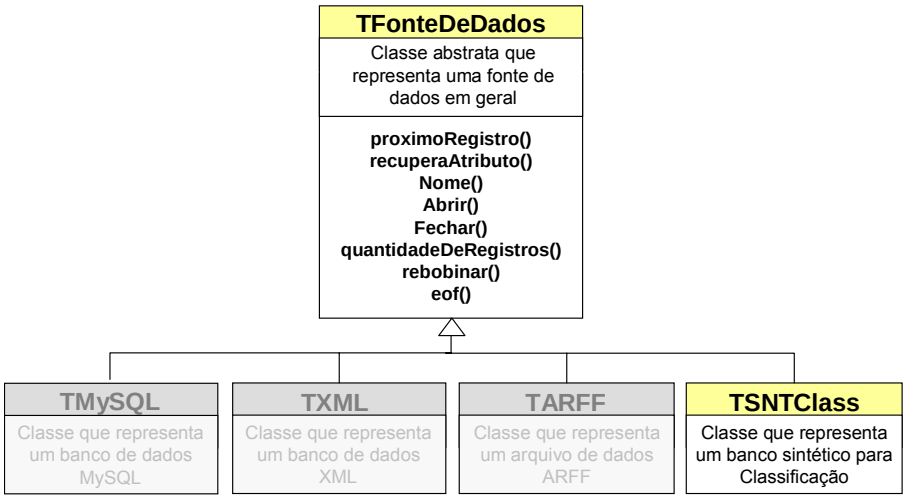


Figura 4.3 – Estrutura de tratamento de fontes de dados

[9] No paradigma da programação orientada a objetos, este processo de implementação de métodos de uma classe (conhecidos como métodos virtuais ou abstratos) por classes descendentes é chamado de ***overriding***. Portanto, classes de novas fontes de dados necessitam apenas efetuar o *override* de métodos virtuais da classe TFonteDeDados para serem integradas à ferramenta.

## 4.2 – Arquitetura do módulo de Classificação

Nesta versão da ferramenta, o único módulo disponível é o de Classificação. A classe **TFrmClassificacao** corresponde à sua tela de interface (Figura 3.2). Esta classe é responsável por colher informações do usuário, como a fonte de dados, algoritmo a ser usado, entre outras, que são necessárias para a geração do modelo.

Para permitir que o usuário informe o banco de dados a ser usado, a tela **TFrmFontes** (Figura 3.5) é exibida. Sua função é apenas intermediar o acesso a telas de interface para abertura de cada tipo de fonte de dados.

O único tipo de fonte de dados disponível nesta versão da ferramenta são os bancos de dados sintéticos para classificação. A tela **TFrmSNTClass** (Figura 3.6) é responsável por permitir ao usuário a abertura de um banco sintético, que pode ser um já existente ou um novo banco. A opção do usuário é passada para a tela correspondente à classe **TFrmAbreFonte**, através do seu método "CriaBanco()", que inicia o processo de abertura do banco. Este processo, dependendo do tamanho do banco de dados a ser aberto, pode ser extremamente lento. Para permitir que este seja abortado pelo usuário, decidiu-se realizar a operação através de *threads*[10] concorrentes à *thread* principal da aplicação. Para que o processo seja abortado a qualquer momento, é suficiente que a nova *thread* seja terminada. Então, a classe **TTValidadorSNTClass** representa uma *thread* para efetuar a validação de um banco sintético já existente. Durante sua execução, atualiza a tela TFrmAbreFonte indicando o estado do processamento.

A classe **TTGeradorSNTClass** representa a *thread* para gerar um novo banco sintético. Similarmente a classe TTValidadorSNTClass, atualiza a tela TFrmAbreFonte para indicar o estado do processamento. Uma outra classe é utilizada para o tratamento da abertura de bancos sintéticos. Esta classe é a **TTInstanciaSNTClass**, que também representa uma *thread* e tem a função de instanciar um objeto da classe **TSNTClass** (descendente de TFonteDeDados) a partir de um banco sintético e associá-lo à tela TFrmClassificacao. Esta *thread* espera pelo término do processamento realizado pelas classes TTGeradorSNTClass ou TTValidadorSNTClass, instancia a fonte de dados a

[10] *Threads* são linhas de execução **concorrentes** e independentes que pertencem a um mesmo processo e compartilham espaço de endereçamento.

partir do banco de dados resultante (validado ou gerado) e a associa à tela TFrmClassificacao através da chamada ao método “AbreBanco()” desta tela, passando esta fonte instanciada como parâmetro.

Com as informações requeridas na tela TFrmClassificacao preenchidas, o usuário pode iniciar a geração do modelo de classificação. Quando isto é feito, esta tela aciona a tela correspondente à classe **TFrmGeraModelo** através de seu método “GeraModelo()”, passando as informações necessárias. Esta tela, baseada nestas informações, dá inicio ao processo de geração do modelo, instanciando a classe **TTSliq**, utilizada quando o algoritmo SLIQ é escolhido, ou a classe **TTSprint**, para o algoritmo SPRINT. Estas duas classes, analogamente às classes de tratamento de bancos sintéticos, também são implementadas como *threads* para possibilitar o cancelamento da operação a qualquer momento e atualizam a tela TFrmGeraModelo para indicar o estado do processamento. Quando este processamento termina, a classe responsável pela realização da tarefa envia as informações extraídas para a tela correspondente à classe **TFrmResultado** para serem visualizadas pelo usuário.

### 4.2.2 – SLIQ e SPRINT

A Figura 4.2 ilustra como as classes que realizam a tarefa de Classificação, TTSliq e TTSprint, interagem com as outras classes da ferramenta. Nesta seção serão abordados detalhes específicos sobre as classes TTSliq e TTSprint e sobre as outras classes envolvidas na realização da tarefa de Classificação, além de como elas interagem para a realizar esta tarefa. Estas classes, com seus principais métodos e interligações, estão ilustradas na Figura 4.4.

Tanto a classe TTSliq quanto TTSprint recebem uma fonte de dados quando são instanciadas, para a qual o modelo de Classificação deve ser gerado. Esta instanciação aciona o método “Execute()” destes objetos, que é responsável por realizar o processo de geração da árvore de decisão a partir desta fonte de dados.

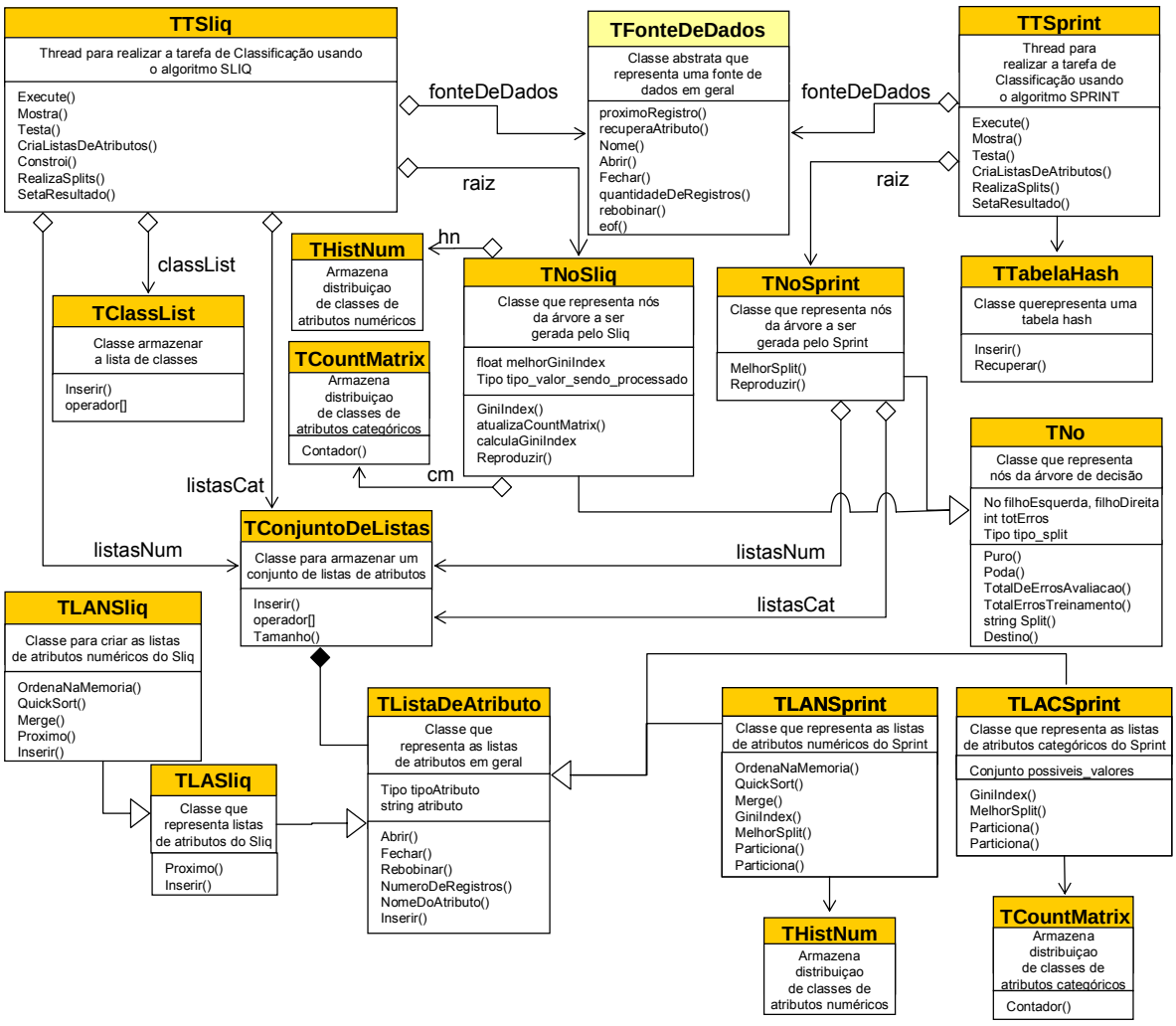


Figura 4.4 – Estrutura de classes dos algoritmos SLIQ e SPRINT

Como descrito no Capítulo 2, a geração do modelo é iniciada criando-se listas de atributos a partir desta fonte, o que é realizado pelo método “CriaListasDeAtributos()”, presente nas duas classes. As listas de atributos dos dois algoritmos contêm algumas similaridades e algumas diferenças. Isto causou a criação de uma hierarquia de classes para representá-las. A classe **TListaDeAtributo** contém características comuns entre as listas de atributos e é a classe base para as classes **TLANSprint**, que representa as listas de atributos numéricos do SPRINT, **TLACSprint**, que representa as listas de atributos categóricos do SPRINT, e **TLASliq**, que representa as listas de atributos do SLIQ. Esta última ainda possui uma especialização, **TLANSliq**, que representa as listas de atributos numéricos do SLIQ. Cada lista de atributo foi implementada como um

arquivo de dados binários. Listas de atributos numéricos precisam ser pré-ordenadas. Isto é realizado através da chamada ao método “Ordena()”, presente em cada classe que representa estas listas. Este método foi implementado com base no algoritmo *QuickSort*, considerado um importante e eficiente método de ordenação. O SLIQ ainda instancia um objeto da classe **TClassList**, necessário para sua execução.

As listas de atributos numéricos são armazenadas em um objeto da classe **TConjuntoDeListas**, denominado listasNum, e, analogamente, as listas de atributos categóricos são armazenadas em outro objeto desta classe denominado listasCat. Na implementação do SLIQ, estes objetos são propriedades da classe TTSliq, pois permanecem as mesmas até o final da geração do modelo. Já na implementação do SPRINT, estes objetos são propriedades da classe TNoSprint, pois cada nó da árvore sendo construída possui seu próprio conjunto de listas. Inicialmente, todos os registros destas listas são associados ao nó raiz, que no SLIQ corresponde a um objeto da classe TNoSliq, e no SPRINT, a um objeto da classe TNoSprint.

A partir deste ponto, inicia-se o processo de criação de novos nós a partir do nó raiz. No SLIQ, isto é feito pelo método “Constroi()”, que invoca o método “RealizaSplits()” até que todas as folhas da árvore contenham apenas registros da mesma classe. O método “RealizaSplits()” percorre todas as listas de atributos para calcular os *gini index* de cada possível *split* para que o melhor seja encontrado para todas as folhas não puras da árvore de decisão corrente (como descrito no Capítulo 2). Como cada lista de atributo corresponde a um arquivo, enquanto uma lista é processada, todas as outras podem permanecer em disco. Cada nó contém um histograma que é utilizado para cada lista de atributo processada. Para listas de atributos numéricos, este histograma é um objeto da classe **THistNum**. Caso uma lista de atributo categórico esteja sendo processada, este histograma é um objeto da classe **TCountMatrix**. Com este melhor *split* encontrado para cada uma destas folhas, estas são particionadas através de seus métodos “Reproduzir()” e, então, a *class list* é atualizada.

Já no SPRINT, o processo de criação de novos nós também utiliza um método chamado “RealizaSplits()”, porém, suas estruturas são diferentes. Este é um método recursivo que recebe uma folha como parâmetro e invoca o método “Reproduzir()”

desta. Caso a execução deste método tenha tido como conseqüência a criação de novos nós, o “RealizaSplits()” é executado recursivamente para cada um destes. Inicialmente, a raiz da árvore de decisão é utilizada como parâmetro. O processo termina quando não existem mais folhas a serem reproduzidas. A escolha do critério de particionamento, então, é de responsabilidade do método “Reproduzir()”, da folha. Este critério é encontrado através da chamada ao método “MelhorSplit()”. Este método invoca, para todas as listas de atributos contidas no nó, o método, de mesmo nome, “MelhorSplit()”. Este método encontra o melhor *split* possível para particionar sua lista de atributo correspondente. O nó, então, usa o melhor dentre todos os *splits* retornados pelas listas para particioná-las. Estas novas partições dão origem a novos nós, filhos deste. A primeira lista de atributo a ser particionada é a lista do atributo utilizado como critério de *split*, através de seu método “Particiona()”, passando-se este critério de *split* como parâmetro para que possa ser aplicado aos registros da lista, descobrindo-se a qual nova partição este registro corresponde. Este método é ainda responsável por criar um objeto da classe **TTabelaHash**, que deve conter informações para o particionamento das outras listas de atributos, o que é feito passando-se este objeto como parâmetro para o método “Particiona()” de todas as outras listas.

Após o processo de construção da árvore, a fase de poda é iniciada. Tanto na implementação do SLIQ quanto na do SPRINT, a poda corresponde à execução do método “Poda()” da classe **TNo**. Este é executado passando-se a raiz da árvore de decisão como parâmetro. Baseado no algoritmo de poda escolhido, este método encontra a opção de menor custo através de chamadas recursivas a ele mesmo, passando seus nós filhos como parâmetros. Estas execuções realizam, ou não, a poda de seus filhos, dependendo da opção de menor custo encontrada, e retornam o custo do nó filho correspondente após o processo. Com as informações do custo de manter seus filhos, disponíveis por estas chamadas recursivas, e com o custo de sua própria estrutura, calculada na fase de construção da árvore, este método efetua a ação que corresponda ao menor custo obtido.

Com a conclusão da fase de poda, inicia-se a fase de avaliação do modelo através do método “Testa()”, caso tenham sido fornecidos registros para avaliação. Esta fase é idêntica nas implementações do SLIQ e SPRINT. A quantidade de erros total é

calculada e uma matriz de confusão é montada. Para isso, a árvore é percorrida para cada registro, onde a cada nó aplica-se ao registro o critério de *split* correspondente para que se descubra qual o caminho a seguir. Chega-se, então, a uma das folhas da árvore de decisão, que tem uma classe associada. Caso esta classe seja diferente da classe do registro corrente, incrementa-se a quantidade de erros de classificação do modelo. A entrada da matriz de confusão correspondente é atualizada.

Neste ponto, a geração do modelo é finalizada. Os documentos contendo as informações extraídas são criados através do método “Mostra()” e passados como parâmetros para o método “SetaResultado()”, que os associa à tela de saída de informações, sendo disponibilizadas para o usuário.

## 4.3 – Resultados experimentais

Para avaliar a implementação dos algoritmos, foram efetuados testes para algumas bases dados. Estas bases são as que foram utilizadas na avaliação de desempenho do SLIQ e do SPRINT, como descrito em [39] e [54]. Elas são distribuídas pela UCI Machine Learning[11], um dos maiores repositórios de dados para estudo da comunidade científica. A Tabela 4.1 contém a descrição dos bancos de dados utilizados.

Tabela 4.1 – Bancos de dados de teste

| Banco | Domínio | Atributos | Classes | Registros |
|---|---|---|---|---|
| Australian | Análise de crédito | 14 | 2 | 690 |
| Diabetes | Diagnóstico da doença | 8 | 2 | 768 |
| DNA | Seqüenciamento de DNA | 180 | 3 | 3186 |
| Letter | Reconhecimento de textos escritos à mão | 16 | 26 | 20000 |
| Satimage | Imagens de satélite | 36 | 7 | 6435 |
| Segment | Segmentação de imagens | 19 | 7 | 2310 |
| Shuttle | Radiação em bases espaciais | 9 | 7 | 57000 |
| Vehicle | Identificação de veículos | 18 | 4 | 846 |

Comparamos os resultados com os apresentados no artigo do SLIQ [39] para validar a implementação. Eles estão apresentados na Tabela 4.2 e 4.3. A Tabela 4.2

[11] Os bancos de dados estão disponíveis em: http://www.ics.uci.edu/~mlearn/MLRepository.html

contém as de taxas de acerto dos modelos gerados para registros de avaliação fornecidos aleatoriamente. A Tabela 4.3 contém o tamanho das árvores de decisão geradas. A primeira coluna destas tabelas corresponde ao nome do banco de dados. Da segunda a quarta estão replicados os resultados apresentados no artigo do SLIQ. A segunda corresponde a uma implementação do algoritmo CART [11]. A terceira corresponde a uma versão anterior do algoritmo C4.5 [50], a C4. A terceira coluna corresponde aos resultados da implementação do SLIQ feita por seus autores. A última coluna contém o resultado da implementação do SLIQ que integra o módulo de Classificação da ferramenta MIDAS-UFF. Devido às árvores de decisão geradas pelos algoritmos serem idênticas, apenas o SLIQ foi avaliado.

Tabela 4.2 – Teste comparativo de taxas de acerto

| Banco | IND - Cart | IND – C4 | SLIQ (IBM) | SLIQ (MIDAS-UFF) |
|---|---|---|---|---|
| Australian | 85.3 | 84.4 | 84.9 | 87.7 |
| Diabetes | 74.6 | 70.1 | 75.4 | 69.8 |
| DNA | 92.2 | 92.5 | 92.1 | 92.1 |
| Letter | 84.7 | 86.8 | 84.6 | 83.2 |
| Satimage | 85.3 | 85.2 | 86.3 | 86.3 |
| Segment | 94.9 | 95.9 | 94.6 | 93.9 |
| Shuttle | 99.9 | 99.9 | 99.9 | 99.9 |
| Vehicle | 68.8 | 71.1 | 70.3 | 73.7 |

Tabela 4.3 – Teste comparativo de tamanhos de árvores de decisão

| Banco | IND - Cart | IND – C4 | SLIQ (IBM) | SLIQ (MIDAS-UFF) |
|---|---|---|---|---|
| Australian | 5.2 | 85 | 10.6 | 11.7 |
| Diabetes | 11.5 | 179.7 | 21.2 | 19.7 |
| DNA | 35 | 171 | 45 | 55 |
| Letter | 1199.5 | 3241.3 | 879 | 879 |
| Satimage | 90 | 563 | 133 | 127 |
| Segment | 52 | 102 | 16.2 | 39 |
| Shuttle | 27 | 57 | 27 | 27 |
| Vehicle | 50.1 | 249 | 49.4 | 47 |

Quando comparado com os resultados obtidos pela implementação dos autores do SLIQ, o desempenho da implementação realizada neste trabalho é idêntica para alguns bancos de dados e levemente diferente para outros, tanto para taxas de acerto

quanto para tamanho das árvores de decisão. Atribui-se a isto o fato dos registros fornecidos para avaliação terem sido escolhidos aleatoriamente. Evidencia-se, então, que as implementações correspondem às especificações dos algoritmos conforme suas propostas originais.

Capítulo 5

# Conclusões e Trabalhos Futuros

Neste trabalho foi proposta a primeira versão de uma ferramenta de Mineração de Dados, chamada MIDAS-UFF.

Esta versão da ferramenta disponibiliza o módulo de Classificação, trazendo a implementação de dois importantes algoritmos geradores de árvores de decisão encontrados na literatura, SLIQ [39] e SPRINT [54].

Existe ainda muito trabalho a ser explorado. O desenvolvimento dos outros módulos correspondentes às outras tarefas de Mineração de Dados é certamente o mais evidente. A integração de novas fontes de dados e o desenvolvimento de módulos de pré-processamento traria não só a possibilidade da análise das técnicas implementadas para um maior domínio de problemas como também tornaria a ferramenta útil a aplicações reais. O desenvolvimento de módulos de pós-processamento também seria importante para que apenas as informações relevantes sejam exibidas aos usuários.

Outras sugestões de trabalhos futuros estão relacionadas ao módulo de Classificação: O PUBLIC [51] é um algoritmo eficiente e sua implementação seria útil; o desenvolvimento de um formato de documento de saída que contivesse uma imagem do modelo gerado seria interessante para o entendimento mais intuitivo das informações extraídas, talvez usando a OpenGL (uma biblioteca disponível para desenvolvimento de gráficos e imagens); o documento que contém regras de classificação poderia ser adaptado para conter instruções SQL para acesso a sistemas de bancos de dados, caso algum deles seja considerado em versões futuras.

Existem também questões de otimização em relação à tarefa de Classificação que podem ser investigadas. A geração de subconjuntos durante a análise de *splits* dos algoritmos geradores de árvores de decisão pode ser uma operação computacionalmente cara quando o banco de dados de treinamento possui atributos categóricos com domínio de valores com muitos elementos. O SLIQ e o SPRINT, para estes casos, utilizam um método guloso. A utilização de meta-heurísticas, como GRASP, algoritmos genéticos, etc., poderia ser uma melhor solução para o problema.

Outra questão é a intensidade de operações de E/S realizadas por estes algoritmos. Os algoritmos mais recentes propostos na literatura ainda precisam realizar muitos passos sobre o banco de dados para gerar o modelo. Alguma estratégia que reduza este número de passos melhoraria sensivelmente o tempo de execução destes algoritmos.

# Bibliografia


[1] P. Adriaans, D. Zantinge, “Data Mining”, Addison-Wesley, 1996.

[2] R. Agarwal, C. C. Aggarwal, V. V. V. Prasad, “A Tree Projection Algorithm for Generation of Frequent Itemsets”, em *Parallel and Distributed Computing*, 61:350-371, 2000.

[3] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, A. Swami, “An Interval Classifier for Data Mining Applications”, em *Proceedings of the International Conference of Very Large DataBases*, 560-573, 1992.

[4] R. Agrawal, T. Imielinski, A. Swami, “Data Mining: A Performance Perspective”, em *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914-925, 1993.

[5] R. Agrawal, R. Srikant, “Fast Algorithms for Mining Association Rules”, em *Proceedings of the 20$^{th}$ International Conference of Very Large DataBases*, 487-499, 1994.

[6] R. Agrawal, R. Srikant, “Mining Sequential Patterns”, em *Proceedings of the 11$^{th}$ International Conference on Data Engineering*, 3-14, 1995.

[7] R. Agrawal, R. Srikant, “Mining Sequential Patterns: Generalizations and Performance Improvements”, em *Proceedings of the 5$^{th}$ International Conference on Extending DataBase Technology – EDBT’96*, 3-17, 1996.

[8] K. Alsabti, S. Ranka, V. Singh, “CLOUDS: A Decision Tree Classifier for Large Datasets”, em *Proceedings of the 1998 International Conference on Data Mining and Knowledge Discovery - KDD’98*, 2-8, 1998.

[9] M. J. A. Berry, G. Linoff, “Data Mining Techniques”, Wiley Computer Publishing, 1997.

[10] C. M. Bishop, “Neural Networks for Pattern Recognition”, Oxford University Press, 1995.

[11] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, “Classification and Regression Trees”, Wadsworth, 1984.

[12] S. Brin, R. Motwani, J. D. Ullman, S. Tsur, “Dynamic Itemset Counting and Implication Rules for Market Basket Data”, em *Proceedings of the ACM SIGMOD International Conference on Management of Data – SIGMOD’97*, 255-264, 1997.

[13] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, D. Freeman, “AutoClass: A Bayesian Classification System”, em *Proceedings of the 5th International Conference on Machine Learning*, Morgan-Kaufmann, 1988.

[14] P. Cheeseman, J. Stutz, “Bayesian Classification (AutoClass): Theory and Results”, em *Advances in Knowledge Discovery and Data Mining,* AAAI Press, 153-180, 1995.

[15] DBMiner Technology Inc., “DBMiner”, http://www.dbminer.com/, 2002.

[16] J. F. Elder IV, B. W. Abbott, “A Comparison of Leading Data Mining Tools”, em *Proceedings of the 1998 International Conference on Knowledge Discovery and Data Mining – KDD’98*, 1998.

[17] U. Fayyad, “On the Induction of Decision Trees for Multiple Concept Learning”, Tese (*Ph.D.*), The University of Michigan, 1991.

[18] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smith, “From Data Mining to Knowledge Discovery: An Overview”, em *Advances in Knowledge Discovery and Data Mining*, AAAI/MIT Press, 1-34, 1996.

[19] J. Gehrke, R. Ramakrishnan, V. Ganti, “RainForest: A Framework for Fast Decision Tree Construction of Large Datasets”, em *Proceedings of the International Conference of Very Large DataBases*, 416-427, 1998.

[20] J. Gehrke, V. Ganti, R. Ramakrishnan, W. Y. Loh, “BOAT: Optimistic Decision Tree Construction”, em *Proceedings of the ACM SIGMOD International Conference on Management of Data – SIGMOD’99*, 169-180, 1999.

[21] M. Goebel, L. Gruenwald, “A Survey of Data Mining and Knowledge Discovery Software Tools”, em *SIGKDD Explorations*, 1:20-33, 1999.

[22] D. E. Goldberg, “Genetic Algorithms and Search, Optimization and Machine Learning”, Morgan Kaufmann, 1989.

[23] S. Guha, R. Rastoji, K. Shim, “CURE: An Efficient Clustering Algorithms for Large Databases”, em *Proceedings of the ACM SIGMOD International Conference on Management of Data – SIGMOD’98*, ACM Press, 73-84, 1998.

[24] S. Guha, R. Rastoji, K. Shim, “ROCK: A Robust Clustering Algorithm for Categorical Attributes”, em *Proceedings of the 15$^{th}$ International Conference on Data Engineering,* 1999.

[25] J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, O. R. Zaiane, “DBMiner: A System for Mining Knowledge in

Large Relational Databases", em *Proceedings of the 1996 International Conference on Data Mining and Knowledge Discovery - KDD'96*, 250–255, 1996.

[26] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns Without Candidate Generation", em *Proceedings of the ACM SIGMOD International Conference on Management of Data – SIGMOD'00,* 1-12*,* 2000.

[27] J. Han, J. Pei, D. Mortazavi-Asl, Q. Chen, U. Dayal, M.-C. Hsu, "FreeSpan: Frequent Pattern–Projected Sequential Pattern Mining", em *Proceedings of the 2000 International Conference of Knowledge Discovery and Data Mining – KDD'00*, 355-359, 2000.

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

[29] E. B. Hunt, J. Marin, P. J. Stone, "Experiments in Induction", Academic Press, 1966.

[30] IBM Corporation, "DB2 Intelligent Miner", http://www.software.ibm.com/data/iminer/, 2004.

[31] G. Karypis, E.-H. Han, V. Kumar, "CHAMELEON: A Hierarchical Clustering Algorithm Using Dynamic Modeling", em *Computer*, 32:68-75, 1999.

[32] G. V. Kass, "An Exploratory Technique for Investigating Large Quantities of Categorical Data", em *Applied Statistics*, 29:119-127, 1980.

[33] Kdnuggets, "Software Suites for Data Mining and Knowledge Discovery", http://www.kdnuggets.com/software/suites.html, 2004.

[34] R. Lippmann, “An Introduction to Computing with Neural Nets”, em *IEEE ASSP Magazine*, 1987.

[35] M. Livny, R. Ramakrishnan, T. Zhang, “Birch: An Efficient Data Clustering Method for Large Databases”, em *Proceedings of the ACM SIGMOD International Conference on Management of Data – SIGMOD’96*, ACM Press, 103-114, 1996.

[36] J. B. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations”, em *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, 1:281-297, 1967.

[37] Megaputer Intelligence Inc., “PolyAnalyst 4.6”, http://www.megaputer.com/products/pa/index.php3, 2004.

[38] M. Mehta, J. Rissanen, R. Agrawal, “MDL-based Decision Tree Pruning”, em *Proceedings of the 1995 International Conference on Knowledge Discovery and Data Mining – KDD’95*, 1995.

[39] M. Mehta, R. Agrawal, J. Rissanen, “SLIQ: A Fast Scalable Classifier for Data Mining”, em *Proceedings of the 5th International Conference on Extending DataBase Technology – EDBT’96*, 1996.

[40] D. Michie, D. J. Spiegelhalter, C. C. Taylor, “Machine Learning, Neural and Statistical Classification”, Ellis Horwood, 1994.

[41] J. A. Morgan, J. N. Sonquist, “Problems in the Analysis of Survey Data and a Proposal”, em *Journal of the American Statistical Association*, 58, 415-434, 1963.

[42] R. T. Ng, J. Han, “Efficient and Effective Clustering Methods for Spatial Data Mining”, em *Proceedings of the 20th International Conference of Very Large DataBases,* Morgan Kaufmann, 144-155, 1994.

[43] Oracle Corporation, “Oracle Data Mining”, http://otn.oracle.com/products/bi/odm/index.html, 2004.

[44] S. Orlando, P. Palmerini, R. Perego, “DCI: A Hybrid Algorithm For Frequent Set Counting”, Relatório Técnico, TR-CS-01-9, Dipartimento di Informatica Università Ca' Foscari di Venezia, 2001.

[45] S. Orlando, P. Palmerini, R. Perego, "The DCP Algorithm for Frequent Set Counting", Relatório Técnico, TR-CS-01-7, Dipartimento di Informatica, Università Ca' Foscari di Venezia, 2001.

[46] J. S. Park, M.-S. Chen, P. S. Yu, “An Effective Hash-Based Algorithm for Mining Association Rules”, em *Proceedings of the ACM SIGMOD International Conference on Management of Data – SIGMOD’95*, 175-186, 1995.

[47] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, ”PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth", em *Proceedings of the 2001 International Conference on Data Engineering - ICDE'01*, 2001.

[48] J. R. Quinlan, “Induction of Decision Trees”, em *Machine Learning*, 1:81-106, 1986.

[49] J. R. Quinlan, “Simplifying Decision Trees”, em *International Journal of Man-Machine Studies*, 27: 221-234, 1987.

[50] J. R. Quinlan, “C4.5: Programs for Machine Learning”, Morgan Kaufmann, 1993.

[51] R. Rastogi, K. Shim, “PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning”, em *Proceedings of the 24$^{th}$ International Conference of Very Large DataBases*, 404-415, 1998.

[52] B. D. Ripley, “Pattern Recognition and Neural Networks”, Cambridge University Press, 1996.

[53] SAS Institute, “Enterprise Miner”, http://www.sas.com/technologies/analytics/datamining/miner/, 2004.

[54] J. Shafer, R. Agrawal, M. Mehta, “SPRINT: A Scalable Parallel Classifier for Data Mining”, em *Proceedings of the 22nd International Conference of Very Large DataBases,* 1996.

[55] G. Sheikholeslami, S. Chatterjee, A. Zhang, “WaveCluster: A Multi-Resolution Clustering Approach of Very Large Spatial Databases”, em *Proceedings of the 24th International Conference of Very Large DataBases,* Morgan Kaufmann, 428-439, 1998.

[56] SPSS Inc., “Clementine”, http://www.spss.com/clementine/, 2004.

[57] The University of Waikato, “Weka 3: Data Mining Software in Java”, http://www.cs.waikato.ac.nz/ml/weka/, 2004.

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

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