

Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

**Arquitetura, Projeto e Implementação da Biblioteca
Evolutionary Computing Learning Environment (ECLE)
para Construir Regras de Conhecimento com Propriedades Específicas**

Rafael Giusti

Adriano Donizete Pila

Maria Carolina Monard

Nº 289

RELATÓRIOS TÉCNICOS DO ICMC

**USP – São Carlos
Dezembro de 2006**

SYSNO	1563457
DATA	/ /
ICMC - SBAB	

**Arquitetura, Projeto e Implementação da Biblioteca
Evolutionary Computing Learning Environment (ECLE)
para Construir Regras de Conhecimento com Propriedades Específicas**

Rafael Giusti

Adriano Donizete Pila

Maria Carolina Monard

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação e Estatística
Laboratório de Inteligência Computacional
Caixa Postal 668, 13560-970 - São Carlos, SP, Brasil
e-mail: rg@grad.icmc.usp.br, {pila, mcmonard}@icmc.usp.br

Resumo

Os algoritmos simbólicos de aprendizado de máquina supervisionado são capazes de gerar um conjunto de regras capaz de explicar o conhecimento implícito em um conjunto de dados. Entretanto, desse conjunto de regras nem sempre é possível obter regras de conhecimento que causem surpresa ou representem conhecimento novo para o especialista do domínio. Neste trabalho propomos um ambiente computacional que consiste de uma biblioteca de classes baseada em algoritmos evolutivos, a qual foi desenvolvida para a construção de regras individuais de conhecimento com propriedades específicas. Também propomos neste trabalho uma representação para regras de conhecimento que codifica as condições da regras de forma a permitir a atuação do algoritmo evolutivo e dos métodos por nós propostos para a avaliação, seleção e recombinação dessas regras. Apresentamos também a arquitetura da biblioteca de classes com as definições e discussões sobre os métodos, bem como a convenção adotada na implementação de todo o trabalho. Finalmente, são apresentadas algumas conclusões sobre o desenvolvimento do projeto.

Palavras-Chave: aprendizado de máquina, regra de conhecimento, algoritmos evolutivos, multi-objetivo, *ranking*.

Este documento foi preparado com o formatador de textos $\text{\LaTeX}$. O sistema de citações de referências bibliográficas utiliza o padrão *Apalike* do sistema $\text{\BibTeX}$.

Sumário

Sumário	i
Lista de Figuras	iv
Lista de Tabelas	v
Lista de Algoritmos	vi
1 Introdução	1
* 2 Aprendizado de Máquina	2
2.1 Descrição de Exemplos	2
2.2 Aprendizado Supervisionado	4
2.3 Terminologia e Linguagem de Descrição	4
2.4 Matriz de Confusão	5
2.5 Matriz de Contingência	5
2.6 Medidas de Avaliação de Regras	6
3 Otimização de Problemas Multi-Objetivo e <i>Ranking</i>	10
3.1 Formulação	10
3.2 Problema de Otimização Multi-objetivo Convexo	11
3.3 Soluções Pareto-ótimas	12
3.4 Objetivos em Otimização Multi-Objetivo	13
3.5 Diferenças entre Otimização Simples-Objetivo e Multi-Objetivo	13
3.6 Operador de Dominância de Pareto	14
3.6.1 Propriedades da Relação de Dominância	15
3.6.2 Otimalidade de Pareto	15
3.7 Calculo do Pareto-ótimo utilizando Somatório de Pesos	17
3.8 Rankings em Problemas de Otimização Multi-objetivo	18
3.8.1 Notação	19
3.8.2 Indivíduos Empatados	19
3.8.3 Combinação de <i>Ranks</i>	20

3.8.4	Aplicação em Problemas de Otimização Multi-Objetivo	21
4	Computação Evolutiva	22
4.1	Indivíduos	23
4.1.1	Representação Pittsburgh	23
4.1.2	Representação Michigan	24
4.2	Seleção	24
4.2.1	Roda da Roleta	25
4.2.2	Torneio	26
4.2.3	<i>Ranking</i>	26
4.3	Operadores	26
4.3.1	<i>Crossover</i>	27
4.3.2	Mutação	27
4.4	Função de Avaliação	28
5	O Projeto DISCOVER	28
5.1	A Biblioteca de Classes Discover Object Library — DOL	29
5.2	O Ambiente Computacional SNIFFER	30
5.3	A Biblioteca de Classes Evolutionary Computing Learning Environment — <i>ECLE</i>	31
6	Definição e Projeto da Biblioteca Evolutionary Computing Learning Environment — <i>ECLE</i>	31
6.1	Indivíduos e População	32
6.1.1	A classe <i>ECRule</i>	34
6.1.2	A classe <i>ECMain</i>	35
6.2	Métodos de Seleção	37
6.2.1	A classe <i>SelectionAbstract</i>	37
6.2.2	A classe <i>OrderedSelectionAbstract</i>	38
6.2.3	A classe <i>ElitismSelection</i>	38
6.2.4	A classe <i>TournamentSelection</i>	38
6.2.5	A classe <i>RouletteWheelSelection</i>	38
6.2.6	A classe <i>LinearRankingSelection</i>	39
6.2.7	A classe <i>ExponentialRankingSelection</i>	39

6.3	Operadores de <i>Crossover</i>	39
6.3.1	A classe <i>CrossoverAbstract</i>	40
6.3.2	A classe <i>StructuralLevelCrossover</i>	40
6.3.3	A classe <i>FeatureLevelCrossover</i>	40
6.3.4	A classe <i>LocalLevelCrossover</i>	41
6.3.5	A classe <i>CompositeCrossover</i>	42
6.4	Operadores de Mutação	43
6.4.1	A classe <i>MutationAbstract</i>	43
6.4.2	A classe <i>StructuralLevelMutation</i>	44
6.4.3	A classe <i>LocalLevelMutation</i>	44
6.4.4	A classe <i>CompositeMutation</i>	44
6.5	Funções de Avaliação	45
6.5.1	A classe <i>MeasureAbstract</i>	45
6.5.2	As classes <i>Measure</i>	47
6.5.3	A classe <i>CompositeMeasure</i>	47
6.5.4	A classe <i>CompositeMORankingMeasure</i>	48
6.6	Estatísticas — A Classe <i>FeatureStats</i>	48
6.7	Iteradores	49
6.7.1	A classe <i>IteratorAbstract</i>	49
6.7.2	A classe <i>RegularIterator</i>	51
6.7.3	A classe <i>ClassifierIterator</i>	51
7	Implementação da Biblioteca <i>ECLE</i>	52
7.1	Convenções Utilizadas na Implementação	52
7.1.1	Nomes dos Identificadores	52
7.1.2	Documentação Interna	53
7.2	Indivíduos e População	53
7.2.1	A Classe <i>ECRule</i>	53
7.2.2	A Classe <i>ECMain</i>	55
7.3	Métodos de Seleção	56
7.4	Função de Avaliação	56

8 O Ambiente Computacional SnifferECLE	57
8.1 A Classe SnifferECLE	57
8.2 A Classe Configure	59
8.3 O Arquivo de Configuração de Experimentos	59
8.3.1 O comando <code>verbose</code>	60
8.3.2 O comando <code>seed</code>	60
8.3.3 O comando <code>select</code>	60
8.3.4 O comando <code>use</code>	61
8.3.5 O comando <code>add</code>	62
8.3.6 O comando <code>stop</code>	63
8.3.7 Especificação de parâmetros	63
8.3.8 Inserção de Comentários	64
8.3.9 Exemplo de Arquivo de Configuração	64
9 Considerações Finais	66
10 Conclusão	66
Agradecimentos	66
Referências Bibliográficas	66

Lista de Figuras

1 Mapeamento do espaço de decisão	11
2 Exemplo da Fronteira de Pareto	12
3 Fronteira de Pareto Diversa	14
4 Fronteira de Pareto pouco diversa	14
5 Conjuntos Pareto-ótimos	16
6 Média aritmética vs. média harmônica	21
7 Exemplo de sobreposição de regras	24
8 Método de seleção roda da roleta	25
9 Crossover de um ponto	27
10 Crossover de dois ponto	27

11	Operador de mutação	28
12	Interação entre filtros, sintaxes e bibliotecas	30
13	Arquitetura de funcionamento da <i>ECLE</i>	32
14	Diagrama de classes da <i>ECLE</i>	33
15	Formato <i>PBM</i> de regra de conhecimento	33
16	Codificação da regra de conhecimento	34
17	Exemplo de codificação da regra de conhecimento	34
18	Diagrama de classes dos métodos de seleção	37
19	Diagrama de classes dos métodos <i>crossover</i>	39
20	Operador de cruzamento em nível estrutural	41
21	Operador de cruzamento em nível de atributo	41
22	Operador de cruzamento em nível local	42
23	Diagrama de classes dos métodos de mutação	43
24	Operador de mutação em nível estrutural	44
25	Operador de mutação em nível local	45
26	Diagrama de classes das medidas de avaliação	46
27	Diagrama de classes de iteração	50
28	Esquema de funcionamento do SnifferECLE	58

Lista de Tabelas

1	Exemplos no formato atributo valor	3
2	Matriz de confusão – classificação binária	5
3	Matriz de contingência para uma regra <i>R</i>	6
4	Matriz de contingência com frequências relativas para uma regra <i>R</i>	6
5	Conjunto de indivíduos — <i>I</i>	18
6	Conjunto de indivíduos <i>rankeados</i>	19
7	Conjunto de indivíduos com empate	19
8	Conjunto de indivíduos com 2 <i>ranks</i> combinados	20

Lista de Algoritmos

4.1	Algoritmo evolutivo	22
-----	-------------------------------	----

6.1	Algoritmo evolutivo para gerar um classificador	51
-----	---	----

1 Introdução

Com o irrefreável crescimento da quantidade de informações armazenadas por organizações de todo o mundo, tem-se tornado cada vez mais importante o uso de técnicas computacionais para a compreensão dos dados. Na área de Inteligência Artificial existem algoritmos de aprendizado, notavelmente os pertencentes ao Aprendizado de Máquina, capazes de automatizar o processo de extração do conhecimento implícito de um conjunto de dados (Quinlan, 1988; Clark e Boswell, 1991; Monard e Baranauskas, 2003a,b). Esse conhecimento, armazenado na forma de exemplos, pode ser transformado em regras valiosas para o processo de tomada de decisão ou, mais ainda, causar surpresa e representar conhecimento novo para o especialista do domínio.

A maioria dos algoritmos de aprendizado recebem como entrada um conjunto de dados no formato atributo-valor. Nesse formato, cada exemplo observado (também chamado de caso ou experiência) é descrito em uma linha dessa tabela, a qual contém características que descrevem aquele caso. Esses exemplos podem estar rotulados, significando que aquele exemplo se enquadra em uma determinada categoria. No caso dos exemplos estarem rotulados, podem ser utilizados algoritmos simbólicos de aprendizado supervisionado. A tarefa desses algoritmos é induzir um conjunto de regras de conhecimento, *i.e.* classificador, consistentes com os exemplos de entrada e que sejam capazes de explicar o conhecimento implicitamente representado por aqueles exemplos. Ainda, esse conjunto de regras que constitui o classificador deve ser capaz de rotular um novo exemplo com boa precisão de acerto futuro. Em outras palavras, as regras devem ter um bom poder de generalização.

Entretanto, o interesse no conhecimento implícito nos dados nem sempre é por razões de predição futura de novos casos. O especialista do domínio procura, também, obter dos dados algum conhecimento que cause surpresa, seja novo ou raro. Em outras palavras, o especialista do domínio gostaria de obter conhecimento que contenha algumas propriedades específicas. Os algoritmos tradicionais de aprendizado não procuram induzir regras com tais propriedades, já que o seu objetivo é induzir um bom classificador, sendo necessário o uso de alguma outra metodologia para esse fim.

Neste trabalho, propomos o uso de algoritmos evolutivos para a construção de regras de conhecimento com propriedades específicas (Freitas, 2002; Bäck et al., 2000; Michalewicz, 1997). Os algoritmos evolutivos são conhecidos pela exploração global que promovem no espaço de busca ao qual são aplicados, implementando um método de busca artificial baseado na seleção natural e na Teoria da Evolução das Espécies (Darwin, 1859). Como a construção de regras de conhecimento é um espaço multimodal e a definição de uma heurística que explore essas características não é trivial, propomos neste trabalho o uso de algoritmos evolutivos para esse fim. Como esse objetivo, implementamos uma biblioteca de classes que implementa um algoritmo evolutivo e os métodos de seleção, recombinação e avaliação consistentes com o paradigma evolutivo, para construir regras de conhecimento com propriedades específicas. Essa biblioteca foi desenvolvida em PERL (Wall et al., 1996) e implementa uma estrutura de dados própria para a representação de regras de conhecimento como indivíduos. A estrutura proposta é apropriada para a manipulação dos indivíduos com os diversos operadores evolutivos, métodos de seleção e avaliação por nós implementados.

Para facilitar a execução de experimentos personalizados utilizando essa biblioteca de classes, também é fornecido um ambiente de configuração e execução capaz de conduzir diversos experimentos em paralelo e coletar os resultados ao final de sua execução. Os experimentos podem ser configurados utilizando uma

linguagem definida por uma gramática por nós proposta. Apresentamos também alguns resultados experimentais preliminares da aplicação dessa biblioteca sobre um conjunto de dados naturais proveniente do repositório da UCI (Newman et al., 1998).

O objetivo deste relatório é descrever o projeto, implementação e funcionamento dessa biblioteca de classes e seu relacionamento com outras bibliotecas de classes implementadas e em implementação no nosso laboratório de pesquisa, LABIC, bem como o funcionamento do ambiente de experimentos.

Este trabalho está organizado na seguinte forma: na Seção 2 são introduzidos conceitos básicos importantes para a compreensão de algoritmos de aprendizado supervisionado e a linguagem de descrição de regras de conhecimento. Na Seção 3 são apresentados conceitos de otimização de problemas multi-objetivo e o uso de *rankings* para esse fim. Na Seção 4 é explicado o funcionamento geral de um algoritmo evolutivo. Na Seção 5 introduz-se o Projeto Discover, sobre o qual se apóia este trabalho. Na Seção 6 são apresentadas definições da arquitetura e o projeto da biblioteca de classes da *ECCE*. Na Seção 7 discorre-se sobre alguns detalhes da implementação dessa biblioteca. Na Seção 8 é apresentado o ambiente computacional para a execução de experimentos. Finalmente, na Seção 10 são apresentadas as conclusões e trabalhos futuros.

2 Aprendizado de Máquina

Devido aos avanços tecnológicos, atualmente é possível armazenar, acessar e processar grandes volumes de dados. Ainda que o processo que gera esses dados não seja completamente conhecido, é possível encontrar um processo que explica os dados observados, já que eles não são gerados aleatoriamente. Pode ser que não seja possível identificar exatamente o processo que explica os dados observados, porém, pode-se tentar construir uma boa aproximação do mesmo tal que seja capaz de explicar parte desses dados por meio da identificação de certos padrões ou regularidades. Esses padrões podem ser úteis para entender o processo ou, baixo certas condições, fazer previsões. Esse é o um dos objetivos de Aprendizado de Máquina — AM.

Aprendizado de Máquina é uma sub área de Inteligência Artificial — IA — que procura otimizar os parâmetros de um modelo que aproxima o processo que explica os dados (exemplos) utilizando exemplos ou experiências passadas observadas. Esse modelo pode ser *preditivo*, *i.e* com capacidade de realizar previsões futuras, *descritivo*, para descrever ou ganhar conhecimento dos dados, ou ambos. Devido ao fato de AM realizar inferências utilizando um subconjunto (amostra) dos dados, ela apóia-se na estatística para construir esses modelos (Alpaydin, 2004). Algoritmos de aprendizado podem ser agrupados de maneiras diferentes segundo diversos critérios. Neste trabalho usamos o critério do grau de supervisão presente nos dados de treinamento (amostra) disponíveis para o aprendizado.

2.1 Descrição de Exemplos

Diversas linguagens podem ser utilizadas para representar os exemplos usados pelos algoritmos de aprendizado, as quais apresentam diferente complexidade e poder de descrição. Uma dessas linguagens, a qual apresenta poder de descrição equivalente ao da lógica proposicional, é amplamente utilizada pela maioria dos algoritmos de aprendizado. Nessa linguagem, utilizada neste trabalho, os exemplos são descritos utilizando um conjunto de atributos apropriados, os quais podem assumir valores diferentes, e cada ex-

emplo é representado em uma linha de uma tabela chamada tabela atributo-valor. Essa representação é praticamente onipresente, seja nas transações bancárias, nos registros escolares, nas compras de supermercado, entre muitos outros. Ela tem sido usada para armazenar milhares de terabytes de dados gerados diariamente, os quais povoam inúmeros bancos de dados.

Na Tabela 1 é apresentado o formato geral de uma tabela atributo-valor com N exemplos E_i com $i = 1, \dots, N$, na forma $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$. Os $\mathbf{x}_i$ são vetores da forma $(x_{i1}, x_{i2}, \dots, x_{iM})$, com valores discretos ou contínuos. Assim, x_{ij} refere-se ao valor do atributo j , denominado $\mathbf{X}_j$, do exemplo E_i , como mostrado na Tabela 1. Os valores y_i , os quais podem ou não existir, referem-se ao valor do atributo $\mathbf{Y}$, denominado atributo *classe*. No caso de classificação, tratado neste trabalho, os valores do atributo classe $\mathbf{Y}$ pertencem a um conjunto C de classes discretas C_v com $v = 1, \dots, N_{Cl}$, ou seja, $C = \{C_1, \dots, C_{N_{Cl}}\}$.

	X_1	X_2	$\dots$	X_M	Y
E_1	x_{11}	x_{12}	$\vdots$	x_{1M}	y_1
E_2	x_{21}	x_{22}	$\vdots$	x_{2M}	y_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
E_N	x_{N1}	x_{N2}	$\vdots$	x_{NM}	y_N

Tabela 1: Exemplos no formato atributo valor

Um conjunto de exemplos para o qual é fornecido o atributo classe é denominado conjunto de exemplos *rotulados*, caso contrario de exemplos *não rotulados*. A classe é o conceito-meta que descreve o fenômeno de interesse. Por exemplo, em uma base de dados médica, o conceito classe pode ser *doente* ou *não-doente*, enquanto que em uma base de mineração de textos o conceito classe pode ser o assunto ao qual o texto está relacionado, tal como *economia*, *esporte*, *política*, *informática* ou *ciência*.

Dado um conjunto de exemplos, esse conjunto de exemplos disponível deve ser utilizado em diferentes fases do aprendizado. Dependendo da fase, o subconjunto do conjunto de exemplos disponível é identificado como descrito a seguir:

- *conjunto de treinamento*: esse conjunto é a principal entrada dos algoritmos de aprendizado. É a partir dele que são construídos os modelos e, portanto, ele deve ser representativo da distribuição da população para realizar inferência sobre esses dados. Na literatura, esse conjunto também é conhecido como *seen cases*, pois refere-se aos exemplos que foram “vistos” pelo algoritmo de aprendizado durante a construção do modelo.
- *conjunto de teste*: esse conjunto é utilizado para avaliar o modelo construído. Esse conjunto, também conhecido como *unseen cases*, não deve ser apresentado ao algoritmo de aprendizado durante a construção do modelo. Idealmente, esse conjunto não deveria ter exemplos em comum com o conjunto de treinamento.
- *conjunto de validação*: em alguns casos, pode ser necessário utilizar exemplos para realizar ajustes no modelo construído pelo algoritmo de aprendizado. Esses exemplos não são utilizados diretamente na construção do modelo, mas são utilizados para o seu ajuste. Dessa maneira, esses exemplos são indiretamente “vistos” durante o processo de aprendizado, o que obriga que os exemplos de validação sejam distintos dos de teste.

Muitos algoritmos de aprendizado têm sido propostos na literatura, os quais podem ser agrupados utilizando diversos critérios. Como mencionado, neste trabalho usamos o critério do grau de supervisão presente nos dados para agrupar esses algoritmos.

2.2 Aprendizado Supervisionado

No aprendizado supervisionado é fornecido ao algoritmo, denominado de *indutor*, um conjunto de exemplos de treinamento $E = \{E_1, E_2, \dots, E_N\}$ no qual cada exemplo E_i tem o valor y_i do atributo classe Y associado, ou seja, $E_i = (\mathbf{x}_i, y_i)$ — Seção 2.1 — tal que $y_i = f(\mathbf{x}_i)$ para uma função f desconhecida, denominada de *função conceito*. O objetivo do aprendizado supervisionado é encontrar ou aproximar essa função conceito f . Entretanto, os exemplos de treinamento utilizados no processo são geralmente insuficientes para caracterizar f , assim, o objetivo dos algoritmos consiste em induzir uma boa aproximação h de f tal que $h(\mathbf{x}) \approx f(\mathbf{x})$. Nesse caso h é chamada de hipótese sobre f . Em outras palavras, do conjunto de possíveis hipóteses H que aproximam f , o algoritmo deve encontrar uma boa hipótese $h \in H$ a qual deve generalizar bem, ou seja, deve prever corretamente o valor da classe de exemplos não vistos durante a sua construção. Esse é o problema fundamental de indução (Russell e Norvig, 2003).

Se o atributo classe Y é quantitativo, *i.e.*, pode assumir valores reais, a hipótese h é denominada *regressor* e a tarefa de aprendizado é chamada *regressão*. Se o atributo classe é nominal, *i.e.*, pode assumir um conjunto finito de valores $\{C_1, \dots, C_{N_{cl}}\}$, a hipótese h é denominada *classificador* e a tarefa de aprendizado é chamada *classificação*, a qual é tratada neste trabalho. Mais especificamente, neste trabalho estamos interessados em conhecimento descrito em uma linguagem simbólica, descrita a seguir.

2.3 Terminologia e Linguagem de Descrição

Os conceitos induzidos por algoritmos de aprendizado simbólicos são geralmente representados por árvores de decisão ou conjuntos de regras. Como sempre é possível escrever uma árvore de decisão como um conjunto de regras disjuntas, deste ponto em diante o termo *regra* refere-se a uma regra extraída de uma árvore de decisão (regras disjuntas) ou uma regra diretamente induzida por um algoritmo de aprendizado simbólico supervisionado (Monard e Baranauskas, 2003a). Uma regra é geralmente representada na forma $R: \text{if } \langle \text{complexo} \rangle \text{ then } \langle \text{class} = C_i \rangle$, na qual C_i é um dos possíveis valores da classe, *i.e.* $C_i \in \{C_1, C_2, \dots, C_{N_{cl}}\}$ e $\langle \text{complexo} \rangle$ é uma disjunção de conjunções de condições para os atributos da forma $X_i \text{ op valor}$, com $X_i \in \mathbf{X}$ e $\text{op} \in \{\leq, \geq, =\}$. Qualquer regra

$$R : \underbrace{\text{if } \langle \text{complexo} \rangle}_{\text{Body ou } B} \text{ then } \underbrace{\langle \text{class} = C_i \rangle}_{\text{Head ou } H}$$

pode ser genericamente representada por $\text{Body} \rightarrow \text{Head}$ ou, resumidamente, $B \rightarrow H$. Assim, diz-se que um exemplo E_i é coberto por uma regra R , se e somente se o exemplo satisfaz todas as condições da regra. Ou seja, um exemplo E_i é coberto por uma regra R , se e somente se B é verdade. Com base nas possíveis coberturas em relação às condições e ao conseqüente da regra, pode-se definir a matriz de contingência de uma regra — Seção 2.5 — enquanto que a matriz de confusão, definida a seguir, refere-se a todo o classificador.

2.4 Matriz de Confusão

Após um classificador ser induzido ele pode ser avaliado de várias formas. Uma delas, frequentemente utilizada, é a taxa de erro ou, de forma complementar, a precisão do classificador. Essa taxa informa qual a estimativa do percentual de acertos na predição da classe de novos exemplos. Essa taxa é obtida a partir da matriz de confusão, que informa quantos erros e acertos ocorreram na predição da classe dos exemplos.

Considerando uma classificação binária, com dois valores para o atributo classe, (+) e (-), na predição da classe de cada exemplo o classificador pode prever corretamente ou incorretamente a classe do exemplo. Existem 4 possibilidades:

1. *Falso negativo (Fn)*, quando o exemplo pertence à classe (+) e é predito pelo classificador como pertencente à classe (-).
2. *Falso positivo (Fp)*, quando o exemplo pertence à classe (-) e é predito pelo classificador como pertencente à classe (+).
3. *Verdadeiro negativo (Tn)*, quando o exemplo pertence à classe (-) e é predito pelo classificador como pertencente à classe (-).
4. *Verdadeiro positivo (Tp)*, quando o exemplo pertence à classe (+) e é predito pelo classificador como pertencente à classe (+).

Considerando um valor da classe como (+) e os outros valores como (-), pode-se generalizar essas definições da matriz de confusão para um classificador com quaisquer número de classes. Na Tabela 2.4 é mostrada a matriz de confusão para o exemplo da classificação binária, onde $N = Tp + Tn + Fp + Fn$.

		Predita		Precisão da classe	Precisão Total
		(+)	(-)		
Verdadeira	(+)	Tp	Fn	$\frac{Tp}{Tp+Fn}$	$\frac{Tp+Tn}{N}$
	(-)	Fp	Tn	$\frac{Tn}{Fp+Tn}$	

Tabela 2: Matriz de confusão – classificação binária

A precisão de um classificador é uma medida importante na predição da classe de novos exemplos. No entanto, existem casos para os quais a precisão do classificador não é boa, mas o conhecimento induzido, *i.e.* conjunto de regras, pode conter regras que individualmente possuam uma boa precisão, ou mesmo possuam algum outro tipo de propriedade interessante da regra. Cada regra pode ser avaliada individualmente com o uso da matriz de contingência, apresentada a seguir.

2.5 Matriz de Contingência

Com base nas possíveis coberturas em relação às condições e ao conseqüente da regra, pode-se definir a matriz de contingência de uma regra, a qual é uma generalização da matriz de confusão do classificador. Em outras palavras, a matriz de confusão refere-se ao classificador como um todo, ou seja, o classificador é tratado como uma caixa preta, enquanto que a matriz de contingência é calculada para cada regra.

Dados uma regra R , e um exemplo $E_i = (\mathbf{x}_i, y_i)$ com a sua respectiva classe y_i , pode-se aplicar a regra ao exemplo e comparar o resultado previsto pela cabeça H da regra com a verdadeira classe y_i do exemplo. Aplicando a regra a um conjunto de teste que contenha N exemplos, podemos derivar, para cada regra, a sua matriz de contingência — Tabela 3. A matriz de contingência também pode ser representada usando frequências relativas, como mostrado na Tabela 4, na qual os valores foram divididos por n , onde n é o número de exemplos considerado para avaliar a regra, ou seja $f_\epsilon = \frac{\epsilon}{n}$. Neste trabalho, consideramos a frequência relativa $\frac{\epsilon}{n}$, associada ao evento ϵ , como uma estimativa de probabilidade para o evento ϵ , denotada $P(\epsilon)$.

	H	$\overline{H}$	
B	bh	$b\overline{h}$	b
$\overline{B}$	$\overline{b}h$	$\overline{b}\overline{h}$	$\overline{b}$
	h	$\overline{h}$	n

bh é o número de exemplos para os quais B é verdade e H é verdade.
 $b\overline{h}$ é o número de exemplos para os quais B é verdade e H é falso.
 $\overline{b}h$ é o número de exemplos para os quais B é falso, mas H é verdade.
 $\overline{b}\overline{h}$ é o número de exemplos para os quais B é falso e H é falso.
 b é o número total de exemplos para os quais B é verdade.
 $\overline{b}$ é o número total de exemplos para os quais B é falso.
 h é o número total de exemplos para os quais H é verdade.
 $\overline{h}$ é o número total de exemplos para os quais H é falso.
 n é o número total de exemplos.

Tabela 3: Matriz de contingência para uma regra R

	H	$\overline{H}$	
B	$\frac{bh}{n}$	$\frac{b\overline{h}}{n}$	$\frac{b}{n}$
$\overline{B}$	$\frac{\overline{b}h}{n}$	$\frac{\overline{b}\overline{h}}{n}$	$\frac{\overline{b}}{n}$
	$\frac{h}{n}$	$\frac{\overline{h}}{n}$	1

Tabela 4: Matriz de contingência com frequências relativas para uma regra R

Usando como base a matriz de contingência, é possível definir a maioria das medidas propostas na literatura para avaliação de regras.

2.6 Medidas de Avaliação de Regras

Várias medidas tem sido propostas na literatura com o objetivo de avaliar regras de conhecimento. Dentre esses trabalhos, destaca-se o trabalho de Lavrač et al. (1999), no qual é proposto tratar essas medidas dentro de um mesmo *framework*, utilizando como base a matriz de contingência com frequências relativas. Segue uma descrição dessas medidas dentro do *framework* proposto por Lavrač et al. (1999), utilizando as frequências relativas — Tabela 4 — bem como as frequências absolutas — Tabela 3.

Precisão (1): A precisão (consistência ou confiança) é uma medida do quanto uma regra é específica para o problema. A precisão pode ser definida como a probabilidade condicional de H ser verdade

dado que B é verdade. Quanto maior, mais precisamente a regra cobre a classe em questão.

$$Acc(R) = P(H|B) = \frac{P(HB)}{P(B)} = \frac{f_{hb}}{f_b} = \frac{hb}{b} \quad (1)$$

Laplace (2): A medida de precisão não considera o número de exemplos cobertos corretamente, o que privilegia regras que cubram corretamente poucos exemplos. A correção de Laplace tenta contornar esse problema penalizando regras que cubram poucos exemplos. Nessa definição N_{Cl} é o número de classes.

$$LAcc(R) = \frac{f_{(hb+1)}}{f_{(b+N_{Cl})}} = \frac{hb+1}{b+N_{Cl}} \quad (2)$$

Erro (3): O erro de uma regra é definido como $1 - Acc(R)$. Quanto maior o erro, menos precisamente a regra cobre a classe em questão.

$$Err(R) = 1 - Acc(R) = P(\bar{H}|B) = \frac{f_{\bar{h}b}}{f_b} = \frac{\bar{h}b}{b} \quad (3)$$

Confiança Negativa (4): é o correspondente à precisão, mas para os exemplos que não são cobertos pela regra. É definida como a probabilidade condicional de H ser falso dado que B também é falso.

$$NegRel(R) = P(\bar{H}|\bar{B}) = \frac{P(\bar{H}\bar{B})}{P(\bar{B})} = \frac{f_{\bar{h}\bar{b}}}{f_{\bar{b}}} = \frac{\bar{h}\bar{b}}{\bar{b}} \quad (4)$$

Sensitividade (5): Sensitividade (completeza ou recall) é uma medida do número (relativo) de exemplos da classe prevista em H cobertos pela regra. É definida como a probabilidade condicional de B ser verdade dado que H é verdade. Quanto maior a sensibilidade, mais exemplos são cobertos pela regra.

$$Sens(R) = P(B|H) = \frac{P(HB)}{P(H)} = \frac{f_{hb}}{f_h} = \frac{hb}{h} \quad (5)$$

Especificidade (6): é o correspondente à completeza, mas para os exemplos que não são cobertos pela regra R . É definida como a probabilidade condicional de B ser falso dado que H é falso.

$$Spec(R) = P(\bar{B}|\bar{H}) = \frac{P(\bar{H}\bar{B})}{P(\bar{H})} = \frac{f_{\bar{h}\bar{b}}}{f_{\bar{h}}} = \frac{\bar{h}\bar{b}}{\bar{h}} \quad (6)$$

Cobertura (7): Cobertura é uma medida do número (relativo) de exemplos cobertos pela regra R . É definida como a probabilidade de B ser verdade. Quanto maior a cobertura, maior o número de exemplos cobertos pela regra R .

$$Cov(R) = P(B) = f_b = \frac{b}{n} \quad (7)$$

Suporte (8): Suporte (frequência) é uma medida do número (relativo) de exemplos cobertos corretamente pela regra R . É definido como a probabilidade de H e B serem verdade. Quanto maior o

suporte, maior o número de exemplos da classe em questão que são cobertos corretamente pela regra R .

$$Sup(R) = P(HB) = f_{hb} = \frac{hb}{n} \quad (8)$$

Novidade (9): Novidade pode ser definida como a probabilidade de B e H ocorrerem juntos não poder ser inferida pelas probabilidades de B e H isoladamente, isto é, B e H não são estatisticamente independentes. A medida de novidade é obtida comparando o valor esperado $P(HB)$ com os valores de $P(H)$ e $P(B)$. Quanto mais o valor esperado diferir do observado, maior é a probabilidade que exista uma correlação verdadeira e inesperada entre B e H . Pode ser demonstrado que $-0,25 \leq Nov(R) \leq 0,25$, e quanto maior um valor positivo (mais próximo de 0,25) mais forte é a associação entre B e H enquanto que, quanto maior um valor negativo (mais próximo de -0,25), mais forte é a associação entre B e $\bar{H}$.

$$Nov(R) = P(HB) - P(H)P(B) = f_{hb} - f_h \times f_b = \frac{1}{n} \times \left(hb - \frac{h \times b}{n} \right) \quad (9)$$

Satisfação (10): Satisfação é o aumento relativo na precisão entre a regra $B \rightarrow \text{verdade}$ e a regra $B \rightarrow H$. É uma medida mais indicada para tarefas voltadas à descoberta de conhecimento, sendo capaz de promover um equilíbrio entre regras com diferentes condições e conclusões.

$$Sat(R) = \frac{P(\bar{H}) - P(\bar{H}|B)}{P\bar{H}} = \frac{f_{\bar{h}} - \frac{f_{\bar{h}b}}{f_b}}{f_{\bar{h}}} = 1 - \frac{\frac{\bar{h}b}{b}}{\frac{\bar{h}}{n}} \quad (10)$$

Precisão Relativa (11): A precisão relativa de uma regra mede o ganho de precisão obtido em relação à precisão de uma regra padrão $\text{verdade} \rightarrow H$, ou seja, que avalia B como verdade para todos os exemplos. Nesse caso, uma regra só interessa se melhorar a precisão da regra padrão.

$$RAcc(R) = P(H|B) - P(H) = \frac{f_{hb}}{f_b} - f_h = \frac{hb}{b} - \frac{h}{n} \quad (11)$$

Confiança Negativa Relativa (12): É o análogo a precisão relativa para os exemplos que não são cobertos pela regra. Nesse caso, a regra padrão é $\text{falso} \rightarrow \bar{H}$.

$$RNegRel(R) = P(\bar{H}|\bar{B}) - P(\bar{H}) = \frac{f_{\bar{h}\bar{b}}}{f_{\bar{b}}} - f_{\bar{h}} = \frac{\bar{h}\bar{b}}{\bar{b}} - \frac{\bar{h}}{n} \quad (12)$$

Sensitividade Relativa (13): A sensibilidade relativa mede o ganho de sensibilidade obtido em relação à sensibilidade de uma regra padrão $B \rightarrow \text{verdade}$, ou seja, uma regra que avalia H como verdade para todos os exemplos.

$$RAcc(R) = P(B|H) - P(B) = \frac{f_{hb}}{f_h} - f_b = \frac{hb}{h} - \frac{b}{n}$$

Especificidade Relativa (14): É o análogo a sensibilidade relativa para os exemplos que não

tos pela regra. Nesse caso, a regra padrão é $\overline{B} \rightarrow falso$.

$$RSpec(R) = P(\overline{B}|\overline{H}) - P(\overline{B}) = \frac{f_{\overline{h}\overline{b}}}{f_{\overline{h}}} - f_{\overline{b}} = \frac{\overline{h}\overline{b}}{\overline{h}} - \frac{\overline{b}}{n} \quad (14)$$

Um ponto importante referente as medidas relativas (medidas 11, 12, 13 e 14) é que elas dão mais informação sobre a utilidade de uma regra que as informações fornecidas pelas suas respectivas medidas absolutas (medidas 1, 4, 5 e 6). Por exemplo, se a precisão de uma regra é menor que a frequência relativa da classe que a regra prediz, então a regra tem um desempenho ruim, independentemente de sua precisão absoluta.

Existe, no entanto, um problema com a precisão relativa, pois é fácil obter uma alta precisão relativa para regras muito específicas, ou seja, regras com uma baixa generalidade de $P(B)$. Para contornar esse problema, é proposto em Lavrač et al. (1999) uma variante das medidas relativas, na qual é atribuído um peso para cada uma dessas medidas. Esse peso promove um balanceamento entre a generalidade e a relatividade dessas medidas.

Dessa forma, a medida $RAcc(R)$ é multiplicada pelo seu coeficiente de peso, $P(B)$, obtendo a nova medida **Precisão Relativa com Peso** (15), também conhecida na literatura como ganho. É possível mostrar que essa medida é equivalente a $Nov(R)$, ou seja, regras com alta precisão relativa também tem alta novidade e vice-versa.

$$WRAcc(R) = P(B)(P(H|B) - P(H)) = f_b \left(\frac{f_{hb}}{f_b} - f_h \right) \equiv Nov(R) \quad (15)$$

Analogamente, as medidas **Confiança Negativa** (16), **Sensitividade** (17) e **Especificidade** (18) **Relativas com Peso** são obtidas multiplicando as correspondentes medidas relativas, $RNegRel(R)$, $RSens(R)$ e $RSpec(R)$, pelos coeficientes de pesos $P(\overline{B})$, $P(H)$ e $P(\overline{H})$, respectivamente. No trabalho de Lavrač et al. (1999) é demonstrado que essas medidas são equivalentes entre si (Equação 19), ressaltando a importância da precisão relativa com peso como medida fundamental para a avaliação de regras, promovendo um balanceamento entre precisão e as outras medidas.

$$WRNegRel(R) = P(\overline{B})P(\overline{H}|\overline{B}) - P(\overline{H}) = f_{\overline{b}} \left(\frac{f_{\overline{h}\overline{b}}}{f_{\overline{b}}} - f_{\overline{h}} \right) = \frac{\overline{b}}{n} \left(\frac{\overline{h}\overline{b}}{\overline{b}} - \frac{\overline{h}}{n} \right) \quad (16)$$

$$WRsens(R) = P(H)(P(B|H) - P(B)) = f_h \left(\frac{f_{hb}}{f_h} - f_b \right) = \frac{h}{n} \left(\frac{hb}{h} - \frac{b}{n} \right) \quad (17)$$

$$WRSpec(R) = P(\overline{H})P(\overline{B}|\overline{H}) - P(\overline{B}) = f_{\overline{h}} \left(\frac{f_{\overline{h}\overline{b}}}{f_{\overline{h}}} - f_{\overline{b}} \right) = \frac{\overline{h}}{n} \left(\frac{\overline{h}\overline{b}}{\overline{h}} - \frac{\overline{b}}{n} \right) \quad (18)$$

$$WRAcc(R) \equiv WRNegRel(R) \equiv WRsens(R) \equiv WRSpec(R) \quad (19)$$

Muitas outras medidas para avaliação de regras podem ser encontradas na literatura. Por exemplo, em Horst (1999) são descritas outras medidas de qualidade e interessabilidade de regras. Freitas (1998a,b) faz o uso de medidas objetivas para avaliar qualidade e surpresa de regras. Também, o número de nós

(atributos) presentes em uma árvore ou o número de condições presentes no complexo de uma regra são usados, normalmente, como medida de compreensibilidade sintática de regras — medida 20. Na equação 20, $| < complexo > |$ é o número de condições que compõem o complexo e $|M|$ é o número de atributos presentes no conjunto de dados descrito na tabela atributo-valor.

$$Intel(R) = 1 - \frac{| < complexo > |}{M} \quad (20)$$

É importante ressaltar que existe a possibilidade de combinar as medidas aqui apresentadas para auxiliar o usuário no entendimento e utilização do conhecimento adquirido por sistemas de aprendizado. Também vale lembrar que, mesmo que a precisão global do classificador não seja considerada boa, podem existir algumas regras boas em termos de qualidade, novidade, interessabilidade e outras. Esse aspecto é muito importante na área de descoberta de conhecimento de bases de dados (KDD) (Fayyad et al., 1996a,b; Rezende et al., 2003). Outros trabalhos relacionados à análise do conhecimento de classificadores simbólicos e à combinação de regras de conhecimento utilizando as medidas aqui descritas, foram desenvolvidos por Gomes (2002) e Bernardini (2006), respectivamente.

3 Otimização de Problemas Multi-Objetivo e *Ranking*

Otimização se refere a encontrar a melhor solução possível para um problema dado um conjunto de restrições (Deb, 2001). Quando o problema envolve a otimização de um único objetivo, deseja-se encontrar a melhor solução disponível, chamada de ótimo global, ou ao menos uma boa aproximação dessa solução. Contudo, são frequentes os casos em que não existe somente um objetivo a ser otimizado, mas vários objetivos que na maior parte dos casos são conflitantes (Coello et al., 2002). Esses problemas com duas ou mais funções objetivo são chamados multi-objetivo e requerem a utilização de algoritmos computacionais e métodos matemáticos diferentes daqueles utilizados para solucionar problemas com um único objetivo. Na verdade, a noção de ótimo é diferente quando consideram-se problemas multi-objetivo porque é desejável um equilíbrio entre todos os objetivos ao invés de somente um objetivo atingir o ponto ótimo (Coello, 2006).

3.1 Formulação

Um problema de otimização multi-objetivo possui um conjunto de funções a serem otimizadas. A otimização pode estar relacionada à minimização ou maximização de cada uma dessas funções. Além dessas funções a serem otimizadas, existe uma série de restrições que devem ser satisfeitas por qualquer uma das soluções encontradas. Conforme Deb (2001) o enunciado geral de um problema de otimização multi-objetivo é o seguinte:

$$\left. \begin{array}{ll} \text{minimizar/maximizar} & f_m(\mathbf{x}), \quad m = 1, 2, \dots, \mathcal{M}; \\ \text{sujeito à} & g_j(\mathbf{x}) \geq 0, \quad j = 1, 2, \dots, \mathcal{J}; \\ & h_k(\mathbf{x}) = 0, \quad k = 1, 2, \dots, \mathcal{K}; \\ & x_i^{(\mathcal{L})} \leq x_i \leq x_i^{(\mathcal{U})}, \quad i = 1, 2, \dots, n. \end{array} \right\} \quad (21)$$

Uma solução $\mathbf{x}$ é um vetor de n variáveis de decisão: $\mathbf{x} = (x_1, x_2, \dots, x_n)^T$. O último conjunto de restrições é chamado de variáveis de restrição, restringindo cada variável de decisão x_i a assumir um valor entre o limite inferior $x_i^{(\mathcal{L})}$ e o limite superior $x_i^{(\mathcal{U})}$. Essas restrições constituem um *espaço da variável de decisão* $\mathcal{D}$, ou simplesmente espaço de decisão. O vetor $\mathbf{x}$ é utilizado como o vetor de *soluções*.

Associado ao problema existem $\mathcal{J}$ restrições de desigualdade e $\mathcal{K}$ restrições de igualdade. Os termos $g_j(\mathbf{x})$ e $h_k(\mathbf{x})$ são chamados funções de restrição. A solução $\mathbf{x}$ que não satisfaz todas as $(\mathcal{J} + \mathcal{K})$ restrições e todas as $2n$ variáveis de limite, definidas anteriormente, é chamada de solução inviável. Por outro lado, se uma solução $\mathbf{x}$ satisfaz todas as variáveis de restrição e de limite, então essa solução é chamada de viável.

Ainda, existem $\mathcal{M}$ funções objetivo $f(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_{\mathcal{M}}(\mathbf{x}))^T$ consideradas na formulação anterior. Cada uma das funções pode ser minimizada ou maximizada. Entretanto, o princípio da dualidade sugere que uma maximização pode ser convertida em uma minimização, e vice-versa, multiplicando a função por -1 , ou seja, minimizar $f(\mathbf{x}) = \text{maximizar } -f(\mathbf{x})$ (Deb, 2001; Coello et al., 2002).

Uma diferença notável entre otimização multi-objetivo e simples-objetivo é que em otimização multi-objetivo a função constitui um espaço multi-dimensional, o qual recebe o nome de espaço objetivo $\mathbf{z}$. Para cada solução $\mathbf{x}$ no espaço das variáveis de decisão, existe um ponto no espaço objetivo, denotado por $f(\mathbf{x}) = \mathbf{z} = (z_1, z_2, \dots, z_{\mathcal{M}})^T$. O mapeamento é feito entre um espaço n -dimensional do vetor solução e um espaço $\mathcal{M}$ -dimensional do vetor objetivo. Na Figura 1 são ilustrados esses dois espaços e o mapeamento existente, considerando no exemplo o espaço de decisão pertencente ao $\mathbb{R}^3$ e o espaço de objetivos pertencentes ao $\mathbb{R}^2$, sendo esse último formado pelas funções objetivo f_1 e f_2 .

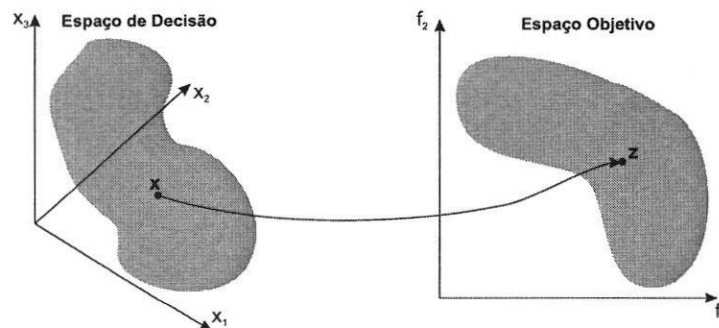


Figura 1: Mapeamento do espaço de decisão no espaço objetivo

3.2 Problema de Otimização Multi-objetivo Convexo

Para entender as propriedades de um problema de otimização multi-objetivo convexo é necessário definir o que é uma função convexa.

Definição 3.1 (Função convexa) Uma função $f : \mathbb{R}^n \rightarrow \mathbb{R}$ é uma função convexa se para quaisquer dois pares de soluções $x^{(1)}, x^{(2)} \in \mathbb{R}^n$, a seguinte condição é verdadeira:

$$f(\lambda x^{(1)} + (1 - \lambda)x^{(2)}) \leq \lambda f(x^{(1)}) + (1 - \lambda)f(x^{(2)}), \text{ para todo } 0 \leq \lambda \leq 1 \quad (22)$$

A definição possui as seguintes duas propriedades de uma função convexa:

1. A aproximação linear da função $f(x)$ em qualquer ponto no intervalo $[x^{(1)}, x^{(2)}]$ sempre subestima o valor real da função.
2. Para uma função convexa, um mínimo local é sempre um mínimo global.

É importante notar que se uma função $g(x)$ é não-convexa, o conjunto de soluções que satisfazem $g(x) > 0$ representa um conjunto convexo. Portanto, um espaço de busca factível formado por funções não convexas formarão uma região factível convexa, o que vai ao encontro da seguinte definição.

Definição 3.2 (Problema de Otimização Multi-objetivo Convexo) *Um problema de otimização multi-objetivo é convexo se todas as funções objetivo são convexas e o espaço factível é convexo.*

3.3 Soluções Pareto-ótimas

Otimizar uma função única resulta em uma única solução ótima viável. Entretanto, quando a otimização envolve múltiplos objetivos várias soluções viáveis podem ser encontradas, tal que nenhuma das soluções é quantitativamente melhor que a outra. Por exemplo, considere o problema ilustrado na Figura 2. O problema do exemplo envolve a compra de um carro cujo custo seja mínimo e o conforto seja máximo. Existem 7 possíveis opções de compra, considerando esses dois objetivos. Analisando a Figura 2 pode-se descartar a opção F, pois a opção B oferece o mesmo conforto a um custo menor. Analogamente, pode-se descartar a opção G, pois a opção E oferece um conforto maior ao mesmo custo. Tem-se então 5 opções de compra: A, B, C, D e E. Em termos quantitativos essas 5 soluções concorrem entre si, dado que nenhuma delas é melhor que a outra. Comparando aos pares, quando uma é melhor no custo, o conforto não é bom e, quando o conforto é alto, o custo não é adequado. Existe então um compromisso entre os objetivos, de forma que um dos objetivos não pode ser melhorado sem a deterioração do outro.

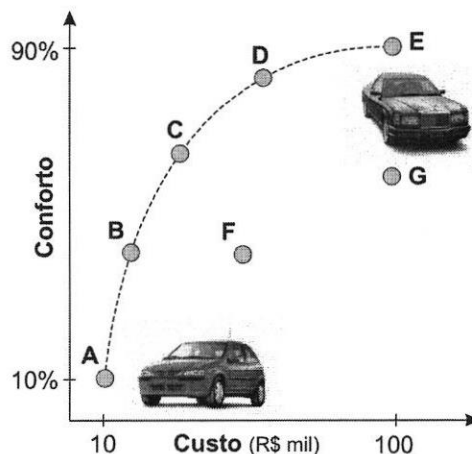


Figura 2: Exemplo da Fronteira de Pareto

Com base nesse compromisso existente entre os objetivos, pode-se dizer que uma solução domina a outra se os valores das funções objetivo que a primeira assume são todos melhores que o da segunda solução. Por exemplo, a solução G é dominada pelas soluções C, D e E. Também, a solução F é dominada pela solução B. As outras soluções que quando comparadas aos pares não se pode dizer qual é a melhor,

dado os valores das funções objetivo, são as chamadas solução não dominadas. No exemplo, as solução dominadas compõem o conjunto $\{F, G\}$ e as solução não dominadas compõem o conjunto $\{A, B, C, D, E\}$, chamadas de soluções Pareto-ótimas. As seguintes duas características descrevem esses dois conjuntos:

1. Quaisquer duas soluções pertencentes ao conjunto não dominado devem ser não dominadas uma em relação à outra.
2. Qualquer solução pertencente ao conjunto dominado deve ser dominada por pelo menos uma solução pertencente ao conjunto não dominado.

Os pontos contidos no conjunto de soluções não dominadas descrevem uma curva das soluções ótimas chamada de Fronteira de Pareto. Dessa formada, dado um problema multi-objetivo, ao obter a fronteira de Pareto, qualquer solução pertencente à fronteira satisfaz o problema inicial e a formulação descrita na Equação 21 na página 10.

3.4 Objetivos em Otimização Multi-Objetivo

Na maioria dos problemas multi-objetivo não se tem informação adicional a respeito da preferência de determinado objetivo sobre os demais. Dessa forma, todos os objetivos possuem o mesmo grau de importância e, portanto, todas as soluções Pareto-ótimas são importantes (Deb, 2001). Essa necessidade faz com que uma otimização multi-objetivo deva alcançar as seguintes duas metas:

1. Encontrar um conjunto de soluções o mais próximo possível da fronteira de Pareto.
2. Encontrar um conjunto de soluções o mais diversa possível.

A primeira meta é obrigatória em qualquer tarefa de otimização. Convergir para um conjunto de soluções que são próximas ao conjunto ótimo real é desejável. Só quando as soluções convergem para a vizinhança da solução ótima real é que se pode certificar das propriedades dessas soluções. Essa meta de otimização multi-objetivo é comum à otimização simples-objetivo.

Por outro lado, a segunda meta é específica da otimização multi-objetivo. Além das soluções encontradas convergirem para a fronteira de Pareto, essas soluções devem estar bem distribuídas ao longo da fronteira. Somente com um conjunto diverso de soluções é possível garantir que as soluções apresentem bons compromissos entre os objetivos considerados. Na Figura 3 está ilustrada uma boa distribuição das soluções Pareto-ótimas na fronteira de Pareto. Já na Figura 4 está ilustrada uma má distribuição das soluções Pareto-ótimas ao longo da fronteira, o que não garante um bom conjunto de soluções. Segundo Deb (2001), como os algoritmos evolutivos multi-objetivos tratam com dois espaços — espaço de variáveis e espaço de objetivos, então, em geral, diversidade em um dos espaços usualmente significa diversidade no outro espaço.

3.5 Diferenças entre Otimização Simples-Objetivo e Multi-Objetivo

Além de ter múltiplos objetivos, existem algumas diferenças fundamentais entre os métodos de otimização multi-objetivo e a simples-objetivo. As três diferenças principais são descritas a seguir:

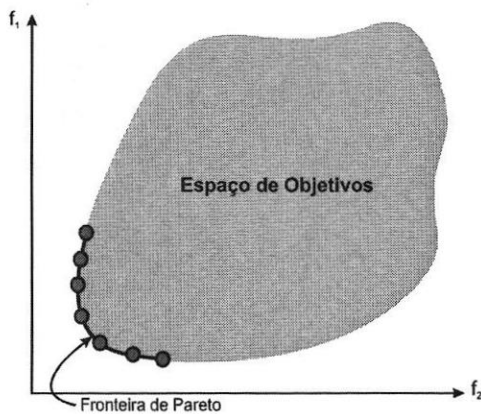


Figura 3: Fronteira de Pareto Diversa

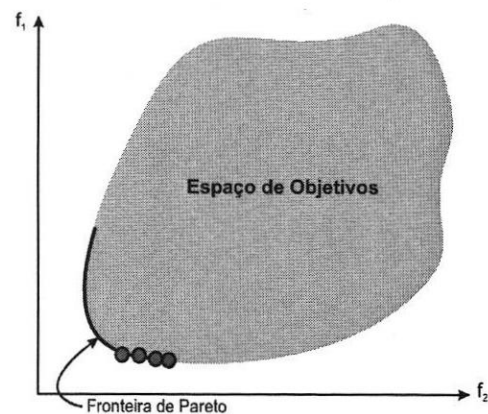


Figura 4: Fronteira de Pareto pouco diversa

1. Duas metas ao invés de uma: Na otimização simples-objetivo existe somente uma meta — encontrar a solução ótima. Embora no espaço de soluções existam soluções ótimas locais, a meta é sempre encontrar a solução ótima global. Mesmo no caso no qual existam várias soluções ótimas, é possível descartar a maioria e manter somente a que possua o melhor valor da função objetivo, que é única. Entretanto, na otimização multi-objetivo existem duas metas claramente definidas — encontrar um conjunto de soluções Pareto-ótimas que possuam bons compromissos entre os objetivos e manter a diversidade da fronteira de Pareto de forma que as soluções não dominadas estejam bem distribuídas ao longo da curva favorecendo o aparecimento de soluções variadas em termos de compromissos entre os valores das funções objetivo. Como ambas metas são importantes, um algoritmo de otimização multi-objetivo eficiente deve trabalhar no sentido de satisfazê-las.

2. Tratar dois espaços de busca: Um problema de otimização multi-objetivo trabalha com dois espaços de busca — variáveis e soluções. Na otimização simples-objetivo existe somente um espaço de busca, cujas soluções são aceitas ou rejeitadas com base nos valores da função objetivo. Embora o espaço de variáveis e de soluções estejam relacionados, o mapeamento entre eles nem sempre é linear e as propriedades desses espaços nem sempre são similares. Existe a necessidade de mapear o espaço das variáveis no espaço dos objetivos de forma que as soluções convirjam para a fronteira de Pareto, sem no entanto perder a diversidade.

3. Sem adaptações artificiais: A maioria dos problemas reais são naturalmente problemas de otimização multi-objetivo. Como existe uma dificuldade no tratamento dos problemas multi-objetivo como verdadeiros problemas com múltiplos objetivos, existem algumas adaptações possíveis para transformá-los em problemas simples-objetivo. Entretanto, tratar um problema multi-objetivo como sendo um problema simples-objetivo depende do método utilizado nessa transformação (soma dos pesos, método de restrições ou programação de metas) o que nem sempre garante soluções Pareto-ótimas.

3.6 Operador de Dominância de Pareto

A maioria dos algoritmos de otimização multi-objetivo utilizam o conceito de dominância. Nesses algoritmos, duas soluções são comparadas para verificar quando uma delas domina a outra.

Assume-se que existam $\mathcal{M}$ funções objetivo. Para definir adequadamente problemas de minimização e

maximização, utiliza-se o operador $\triangleleft$ entre duas soluções i e j , tal que $i \triangleleft j$ se a solução i é melhor que a solução j em um objetivo em particular. Similarmente, $i \triangleright j$ para um objetivo em particular significa que a solução j é melhor que a solução i .

Definição 3.3 (Dominância) *Uma solução $x^{(1)}$ domina a outra solução $x^{(2)}$, se as duas condições abaixo forem satisfeitas:*

1. *A solução $x^{(1)}$ não é pior que a solução $x^{(2)}$ em todos os objetivos, ou $f_j(x^{(1)}) \not\geq f_j(x^{(2)})$ para todo $j = 1, 2, \dots, M$.*
2. *A solução $x^{(1)}$ é estritamente melhor que a solução $x^{(2)}$ em ao menos um objetivo, ou $f_i(x^{(1)}) \triangleleft f_i(x^{(2)})$ para ao menos um $i \in \{1, \dots, M\}$.*

Se qualquer uma das duas condições anteriores for violada, a solução $x^{(1)}$ não domina a solução $x^{(2)}$. Se $x^{(1)}$ domina a solução $x^{(2)}$ (ou matematicamente $x^{(1)} \preceq x^{(2)}$), é usual escrever essa relação nas seguintes formas:

- $x^{(2)}$ é dominada por $x^{(1)}$;
- $x^{(1)}$ é não-dominada por $x^{(2)}$, ou;
- $x^{(1)}$ é não-inferior à $x^{(2)}$.

Utilizando esse conceito e analisando a Figura 2 na página 12, pode-se dizer que a solução E domina a solução G e que a solução B domina a solução F ou, simplesmente, utilizando a notação, $E \triangleleft G$ e $G \triangleleft B$. Intuitivamente, se a solução $x^{(1)}$ domina outra solução $x^{(2)}$, a solução $x^{(1)}$ é melhor que a solução $x^{(2)}$ no que se refere à otimização multi-objetivo.

3.6.1 Propriedades da Relação de Dominância

De acordo com as definições presentes no livro de Deb (2001) a relação de dominância apresenta as seguintes relações:

- Reflexiva: A relação de dominância não é reflexiva, pois de acordo com a Definição 3.3 uma solução p não pode dominar a si mesma.
- Simétrica: A relação de dominância também não é simétrica, porque $p \triangleleft q$ não implica que $q \triangleleft p$.
- Transitiva: A relação de dominância é transitiva, pois se $p \triangleleft q$ e $q \triangleleft r$, então $p \triangleleft r$.

3.6.2 Otimalidade de Pareto

O conjunto de possíveis soluções de um problema multi-objetivo pode ser dividido em dois sub-conjuntos, utilizando a relação de dominância e comparando as soluções duas a duas. Um dos conjuntos possui todas as soluções dominadas enquanto que o outro conjunto possui todas as soluções não dominadas.

Definição 3.4 (Conjunto Não-dominado) Dado um conjunto de soluções $\mathcal{P}$, o conjunto não-dominado de soluções $\mathcal{P}'$ é formado por aquelas soluções não dominadas por qualquer solução pertence ao conjunto $\mathcal{P}$.

Se um conjunto $\mathcal{P}$ é o espaço completo se busca ($\mathcal{P} = \mathcal{S}$), o conjunto $\mathcal{P}'$ é chamado de conjunto Pareto-ótimo. Na Figura 5 são mostradas possíveis conjuntos Pareto-óticos formadas pela combinação de duas funções f_1 e f_2 .

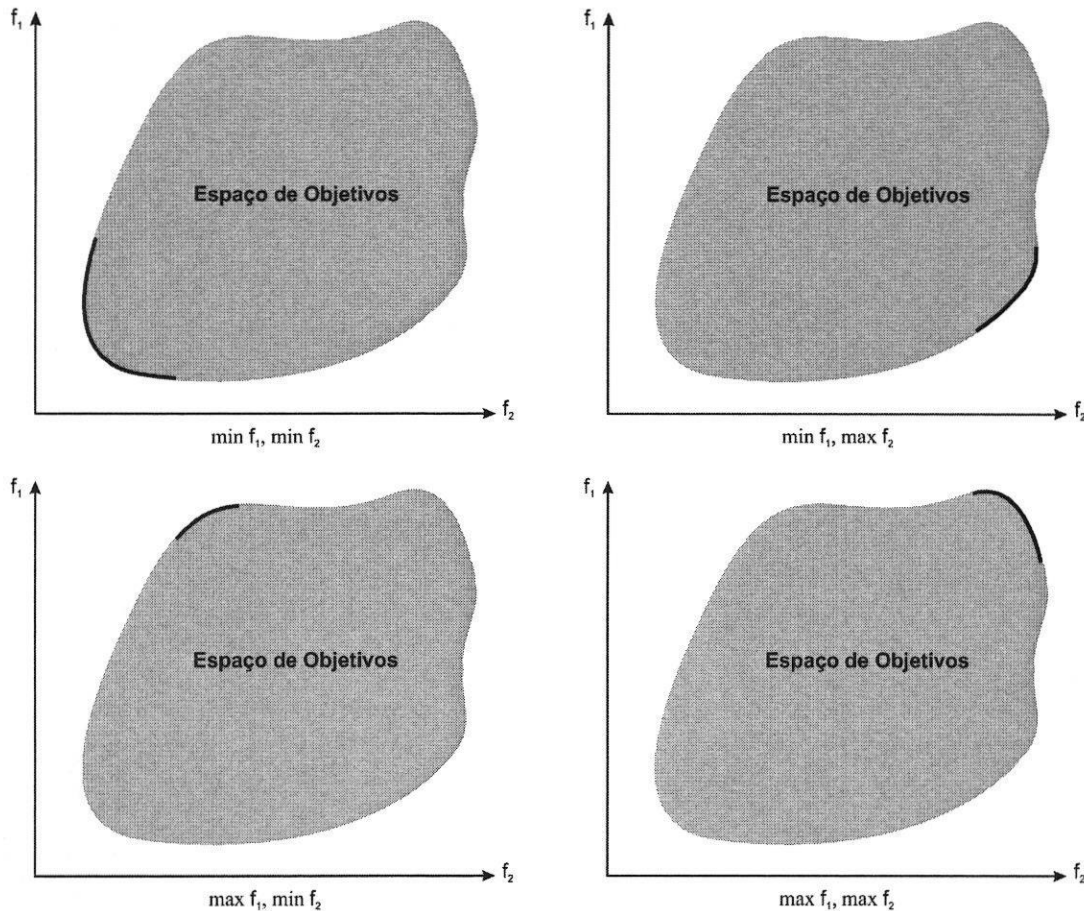


Figura 5: Conjuntos Pareto-óticos

Definição 3.5 (Conjunto Pareto-ótimo) O conjunto não dominado $\mathcal{P}'$ para a totalidade do espaço de busca $\mathcal{S}$ é chamado de conjunto Pareto-ótimo.

Na maioria das ocasiões o Pareto-ótimo global é simplesmente referido como o conjunto Pareto-ótimo. Já que soluções desse conjunto não são dominadas por qualquer outra solução factível do espaço de busca, elas são soluções ótimas para o problema de otimização multi-objetivo. Define-se um conjunto Pareto-ótimo local como:

Definição 3.6 (Conjunto Pareto-ótimo local) Se para cada membro x em um conjunto $\mathcal{P}$ não existe solução y (na vizinhança de x tal que $\|y - x\|_\infty \leq \epsilon$, onde ϵ é um pequeno número positivo) dominando qualquer membro de $\mathcal{P}$, então soluções pertencentes ao conjunto $\mathcal{P}$ constituem um conjunto Pareto-ótimo local.

Das definições anteriores conclui-se que como nem sempre é possível conhecer todo o espaço de busca $\mathcal{S}$, o conjunto Pareto-ótimo encontrado por qualquer método nem sempre representará o verdadeiro conjunto Pareto-ótimo. Dessa forma, o maior desafio está em como aproximar a fronteira de Pareto da verdadeira região Pareto-ótima $\mathcal{P}'$.

3.7 Cálculo do Pareto-ótimo utilizando Somatório de Pesos

A soma ponderada de pesos, como sugere o nome, reduz um conjunto de objetivos para um objetivo simples utilizando um somatório dos objetivos, cada qual multiplicado por um peso. Essa abordagem é a mais simples e talvez a mais utilizada de todas (Deb, 2001). Quando um problema de otimização multi-objetivo aparece essa é a abordagem mais natural a ser considerada. Por exemplo, se o problema multi-objetivo envolve a minimização dos custos de produção e a minimização do desperdício de material utilizado no processo de fabricação, é natural pensar em minimizar a soma ponderada desses dois objetivos.

Embora a idéia seja simples, ela introduz uma questão não muito simples. Quais são os valores de peso a serem considerados? Não existe uma única resposta para esse questionamento, pois os pesos escolhidos dependem da importância que se quer atribuir a cada um dos objetivos considerando o contexto do problema e ao fator de escala. O peso do objetivo é usualmente escolhido em proporção à importância relativa daquele objetivo para o problema. No exemplo anterior, o custo de produção pode ser mais importante que o desperdício de material, então, pode-se atribuir um peso maior àquele objetivo.

Contudo, associar um vetor de pesos aos objetivos depende muito da escala de cada um dos objetivos, pois cada objetivo pode possuir diferente ordem de magnitude. Novamente, no exemplo anterior, o custo de um produto pode variar entre R\$ 100,00 e R\$ 1.000,00 enquanto que o desperdício de material pode variar entre 0,01m³ e 0,1m³. Quando os objetivos são ponderados para formar uma função objetivo composta, é melhor que os objetivos sejam escalados de forma que todos possuam aproximadamente a mesma ordem de magnitude. Por exemplo, o custo do produto poderia ser multiplicado por $1(10^{-3})$ e o montante de material desperdiçado por $1(10^1)$. Esse processo é chamado de *normalização dos objetivos*.

Depois que os objetivos foram normalizados, uma função objetivo composta $\mathcal{F}(\mathbf{x})$ pode ser construída pela soma dos objetivos normalizados e o problema de otimização multi-objetivo definido pela Equação 21 é convertido em um problema de otimização de objetivo simples, como segue:

$$\left. \begin{array}{ll} \text{minimizar} & \mathcal{F}(\mathbf{x}) = \sum_{m=1}^{\mathcal{M}} w_m f_m(\mathbf{x}), \\ \text{sujeito à} & g_j(\mathbf{x}) \geq 0, \quad j = 1, 2, \dots, \mathcal{J}; \\ & h_k(\mathbf{x}) = 0, \quad k = 1, 2, \dots, \mathcal{K}; \\ & x_i^{(\mathcal{L})} \leq x_i \leq x_i^{(\mathcal{U})}, \quad i = 1, 2, \dots, n. \end{array} \right\} \quad (23)$$

Sendo que $w_m \in [0, 1]$ é o peso da m -ésima função objetivo. Como o mínimo do problema anterior não muda se todos os pesos forem multiplicados por uma constante, é usual escolher pesos de forma que a soma seja um, ou $\sum_{m=1}^{\mathcal{M}} w_m = 1$. Para o problema definido na Equação 21, existem dois teoremas interessantes que são apresentados a seguir.

Teorema 3.1 (Pesos positivos) *A solução para o problema representado pela Equação 21 é Pareto-ótimo se os pesos são positivos para todos os objetivos.*

A prova desse teorema é obtida assumindo-se a premissa que a solução ótima obtida com todos os pesos positivos não é uma solução Pareto-ótima e então mostra-se uma contradição da premissa. Este teorema é verdadeiro para qualquer problema de otimização multi-objetivo (Coello et al., 2002). Contudo, isso não implica que qualquer solução Pareto-ótima possa ser obtida utilizando um vetor de pesos positivos. O próximo teorema confirma isso para problemas convexos.

Teorema 3.2 (Problema Multi-objetivos Convexos) *Se $\mathbf{x}^*$ é uma solução Pareto-ótima para um problema de otimização multi-objetivo convexo, então existe um vetor não nulo de pesos positivos $\mathbf{W}$ tal que $\mathbf{x}^*$ é uma solução para o problema dado pela Equação 21.*

Este teorema sugere que para um problema de otimização multi-objetivo convexo, qualquer solução Pareto-ótima pode ser encontrada utilizando o método de somatório dos pesos. Demonstrações e exemplos confirmando esse resultado podem ser encontrados em Deb (2001) e Coello et al. (2002).

Em trabalhos recentes publicados por Flach (2003) e Fuernkranz e Flach (2005) é demonstrado utilizando análises do espaço ROC (Provost e Fawcett, 2001), que as funções que definem as medidas de avaliação de regras são convexas e, portanto, o resultado anterior se verifica. Em suma, é possível encontrar as soluções Pareto-ótimas utilizando o método de Somatório de Pesos a partir de uma função ponderada composta pelas medidas de avaliação de regras.

3.8 Rankings em Problemas de Otimização Multi-objetivo

Quando um número de indivíduos está ordenado segundo alguma medida, pode-se dizer que eles estão classificados (*rankeados*). O arranjo como um todo constitui uma classificação (*ranking*), sendo que cada indivíduo possui uma posição (*rank*)¹. É usual, mas não essencial, denotar os *ranks* por números ordinários 1, 2, ..., n , sendo que n é o número total de indivíduos no *ranking* (Kendall, 1990). Por exemplo, na Tabela 5 estão dispostos 10 indivíduos e seus *ranks* com relação à medida A . Se um indivíduo j possui *rank* i em relação à medida A , isso significa que para a medida A o indivíduo j está na i -ésima posição dentre os indivíduos do *ranking* e, portanto, existem outros $i - 1$ indivíduos que são melhores que o indivíduo j . Entretanto, não se sabe o quanto melhor o indivíduo j é em relação aos demais indivíduos do *ranking* porque não é considerada a magnitude da medida dos indivíduos, apenas sua posição no *ranking*.

Indivíduos	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
Medida (A)	30	80	40	10	60	20	90	50	1000	70
Rank (A)	8	3	7	10	5	9	2	6	1	4

Tabela 5: Conjunto de indivíduos — I

Para melhor ilustrar, a Tabela 5 foi reordenada para compor o *ranking* segundo a medida A , originando a Tabela 6. Para a medida A , quanto maior seu valor melhor posicionado no *ranking* o indivíduo estará.

A aplicação de *rankings* de indivíduos pode ser feita em diversas situações. Pode-se produzir um *ranking* para simplesmente ter um arranjo de indivíduos cuja posição sirva de referencial em relação aos demais indivíduos. Outra possibilidade é construir um *ranking* de acordo com alguma medida que não possa

¹Neste documento utilizamos os termos em inglês por serem os mais conhecidos da comunidade acadêmica.

Indivíduos	I_9	I_7	I_2	I_{10}	I_5	I_8	I_3	I_1	I_6	I_4
Medida (A)	1000	90	80	70	60	50	40	30	20	10
Rank (A)	1	2	3	4	5	6	7	8	9	10

Tabela 6: Conjunto de indivíduos *rankeados*

ser medida em nenhuma escala. Por exemplo, ordenar alguns minérios de acordo com a dureza, de forma que um minério é mais duro que outro se o primeiro for capaz de riscar o segundo. Note que não é preciso nenhuma medida de força, densidade do material ou tamanho do minério para produzir um *ranking* considerando esse critério. Também, pode-se criar um *ranking* de acordo com alguma medida mensurável mas sem a necessidade de aferição. Por exemplo, dividir os alunos de uma sala de aula de acordo com a estatura. Mesmo feito a olho nú, é possível produzir um *ranking* da altura dos alunos. Há ainda a possibilidade de criar um *ranking* de acordo com alguma medida qualitativa que não seja mensurável. Por exemplo, classificar pessoas de acordo com o grau de inteligência. É possível fazer isso, mas não existe nenhum subsídio teórico que comprove que determinada forma ou medida seja definitiva.

3.8.1 Notação

Seja I um conjunto de indivíduos e seja W um conjunto de medidas relacionadas ao primeiro conjunto. Considerando que w_{ij} é a i -ésima medida relacionada ao j -ésimo indivíduo, constitui-se uma *ranking* se para cada indivíduo $I_i \in I$ for atribuído um *rank* que é um número ordinário, de forma que dados quaisquer dois indivíduos $I_m, I_n \in I$, se $w_{im} > w_{in}$, então $r_{im} > r_{in}$, sendo $r_{ij} \in R$ o *rank* de classificação e R o conjunto formado por todos os *ranks*.

3.8.2 Indivíduos Empatados

Uma propriedade do *ranking* de indivíduos que deve ser mantida é que o somatório dos *ranks* deve ser uma constante para não influenciar na distribuição dos *ranks* e nem em comparações dos mesmos (Kendall, 1990). Dessa forma, a dimensão das diferentes medidas continua a não influenciar na ordem de preferência dos indivíduos. Em caso de empate, ou seja, indivíduos com o mesmo valor de medida, os *ranks* iniciais são substituídas pela média dos valores iniciais dos *ranks* empatados. Por exemplo, na Tabela 7 os indivíduos I_5 , I_8 e I_3 possuem os mesmos valores da medida A (50) e estão empatados. Dessa forma, o *rank* inicial desses indivíduos, que eram, respectivamente, 5, 6 e 7, passa a ser $\frac{1}{3}(5 + 6 + 7) = 6$.

Indivíduos	I_9	I_7	I_2	I_{10}	I_5	I_8	I_3	I_1	I_6	I_4	
Medida (A)	1000	90	80	70	50	50	50	30	20	10	
Rank (A) – com empate	1	2	3	4	5	6	7	8	9	10	$\sum = 55$
Rank (A) – sem empate	1	2	3	4	6	6	6	8	9	10	$\sum = 55$

Tabela 7: Conjunto de indivíduos com empate

Em outras palavras, seja w_{ij} a i -ésima medida relacionada ao j -ésimo indivíduo, em caso de empate, *i.e.*

$\exists j, k \in \mathbb{N} \mid w_{ij} = w_{ij+1} = \dots = w_{ik-1} = w_{ik}$, então

$$r_{ij} = r_{ij+1} = \dots = r_{ik-1} = r_{ik} = \frac{\sum r_{ix}}{|\bigcup r_{ix}|} = \frac{(r_{ik} + r_{ij})}{2} \quad (24)$$

sendo que $j \leq x \leq k$ e $|\bigcup r_{ix}|$ é número de elementos pertencentes ao conjunto formado pelos *ranks* de todos os indivíduos empatados.

3.8.3 Combinação de *Ranks*

Outra interpretação possível está na média dos *ranks*, que remete à prioridade média do indivíduo quando considerada mais que uma medida. Novamente, a magnitude das medidas, a escala ou a ordem de importância das medidas não influenciam na atribuição dos *ranks*. Na Tabela 8, os indivíduos estão *rankeados* segundo as medidas *A* e *B*, sendo que para a medida *A* o melhor indivíduo é o que possui o maior valor e para a medida *B* o raciocínio é o inverso. Por exemplo, o indivíduo I_2 possui como medida *A* o valor 80 e para a medida *B* o valor 67. Os respectivos *ranks* são 3 e 5, que na média aritmética resulta num *rank* 4. Os casos de empate são resolvidos isoladamente quando da criação do *ranking* de cada medida. Após a composição do *rankings*, eventuais *ranks* iguais permanecem empatados, pois o critério de desempate reside no caso de valores de medidas iguais.

Indivíduos	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
Medida (A)	30	80	40	10	60	20	90	50	1000	70
Rank (A)	8	3	7	10	5	9	2	6	1	4
Medida (B)	54	67	48	92	70	77	82	62	59	85
Rank (B)	2	5	1	10	6	7	8	4	3	9
Média Aritmética	5	4	4	10	5,5	8	5	5	2	6,5
Média Harmônica	3,6	3,8	2,2	10,0	5,5	7,9	3,6	4,9	1,6	5,8

Tabela 8: Conjunto de indivíduos com 2 *ranks* combinados

A combinação dos *ranks* utilizada neste trabalho, podem ser feitas basicamente de duas formas:

Média aritmética ponderada: o *rank* de cada indivíduo é a média aritmética de todos os *ranks* relativos à medidas daquele indivíduo.

$$r_j = \frac{\sum \alpha_{ij} r_{ij}}{|W|} \quad (25)$$

sendo r_j o *rank* final do j -ésimo indivíduo, r_{ij} o *rank* de cada uma das i medidas do j -ésimo indivíduo, α_{ij} o peso daquela medida e $|W|$ o número de elementos pertencentes ao conjunto W .

Média harmônica ponderada: o *rank* de cada indivíduo é a média harmônica de todos os *ranks* relativos à medida daquele indivíduo.

$$r_j = \frac{|W|}{\sum \frac{\alpha_{ij}}{r_{ij}}} \quad (26)$$

sendo r_j o *rank* final do j -ésimo indivíduo, r_{ij} o *rank* de cada uma das i medidas do j -ésimo indivíduo, α_{ij} o peso daquela medida e $|W|$ o número de elementos pertencentes ao conjunto W .

A média harmônica é interessante porque fornece valores maiores quando os indivíduos envolvidos no cálculo apresentam um melhor equilíbrio. Na Figura 6 são mostradas as curvas resultantes do cálculo da média aritmética e da média harmônica sobre os pares de valores $(x, y) | x \in \{1..10\}, y \in \{7\}$. O número 7 foi arbitrariamente escolhido como parâmetro de controle na comparação.

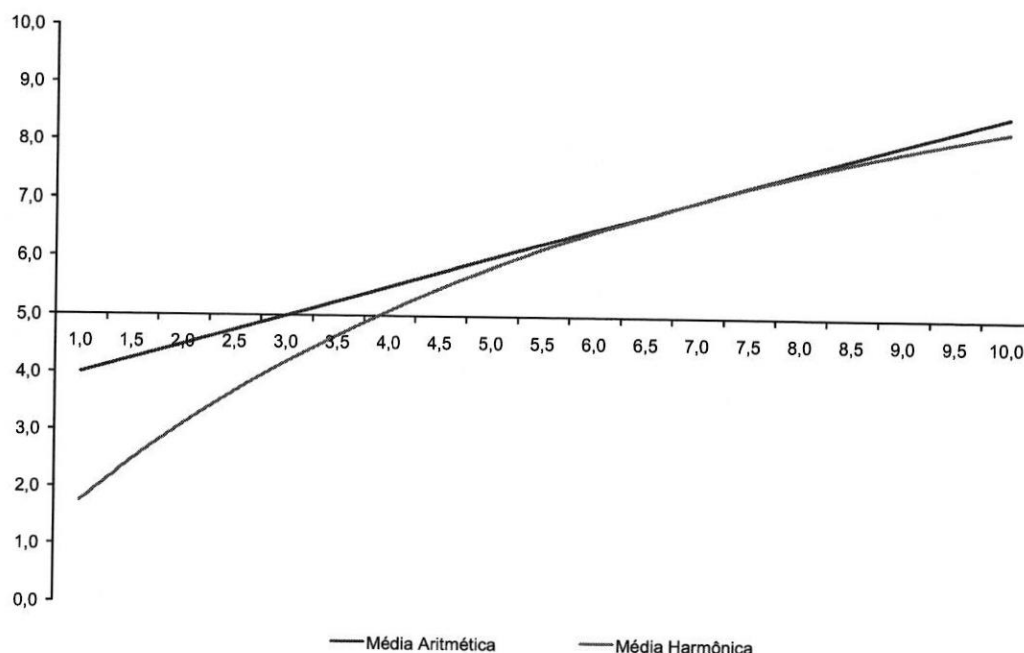


Figura 6: Média aritmética vs. média harmônica

Note que a média harmônica assume valores próximos ao da média aritmética quando os valores considerados, para esse caso, estão próximos ao valor 7. Valores muito discrepantes resultam numa média harmônica inferior à média aritmética. Por exemplo, o par $(1, 7)$ produz valor 4 para a média aritmética, enquanto que para a média harmônica produz um valor em torno de 1,75. Esse resultado é interessante porque o comportamento apresentado pela média harmônica vai ao encontro da definição de problemas multi-objetivo apresentada na Seção 3.

3.8.4 Aplicação em Problemas de Otimização Multi-Objetivo

O uso do método de somatório dos pesos para problemas de otimização multi-objetivo necessita da definição dos pesos para cada uma das funções objetivo. Entretanto, a definição desses pesos é outro problema multi-objetivo a ser solucionado, salvo o caso em que o usuário queira utilizar os pesos para atribuir preferências a cada uma das funções objetivo. Além disso, nessa abordagem existe o problema de uma solução assumir um valor muito grande para uma função objetivo e um valor muito pequeno para outra função objetivo. Por exemplo, se $f_1(\mathbf{x}) = 0,95$ e $f_2(\mathbf{x}) = 0,05$, e se forem utilizados pesos iguais à 0,5, então tem-se que o valor resultante $\mathcal{F}(\mathbf{x}) = 0,5 * 0,95 + 0,5 * 0,05 = 0,5$, ou seja, pode-se assumir que o resultado é bom, embora individualmente exista uma função que assumiu um valor muito pequeno.

Para contornar esses problemas, nesse trabalho utiliza-se *ranking* como forma de compor a função $\mathcal{F}$ sem que os diferentes valores das funções multi-objetivo interfiram demasiadamente no valor resultante da função composta. Para efeitos de comparação e possibilidade de diversidade de experimentos, também foi implementada a composição das funções objetivo utilizando a soma ponderada dos pesos.

4 Computação Evolutiva

Os algoritmos baseados em computação evolutiva utilizam as noções da seleção natural para simular um método de busca que implementa uma seleção artificial, na qual os indivíduos de melhor aptidão têm maior tendência à sobreviver e propagar suas características. Para tanto, é necessário um formalismo de representação e funcionamento que simule computacionalmente o que ocorre na natureza (Michalewicz, 1997).

Diferentes problemas requerem diferentes representações para o uso da computação evolutiva. Entretanto, na maior parte dos casos a representação é semelhante. Assume-se a existência de uma população inicial de indivíduos $P(t)$ ($t = 0$) que, geralmente, são inicializados de forma aleatória — neste trabalho a população inicial é composta por regras quaisquer consistentes com um mesmo conjunto de dados. Cada indivíduo é uma possível solução para o problema tratado, sendo que a população inicial será formada por possíveis soluções para esse problema. Dada a população inicial de indivíduos, cada um desses indivíduos será avaliado pela função de avaliação² com relação à real possibilidade de solucionar adequadamente o problema. Então, aos indivíduos são atribuídos valores que refletem de forma probabilística suas chances de serem escolhidos para a reprodução e criação de novos indivíduos. Quanto maior for o valor recebido na avaliação, maior será a probabilidade de seleção para a reprodução. Assim, um número de indivíduos é selecionado aleatoriamente, respeitando suas probabilidades de seleção. Os indivíduos são então, aos pares, geneticamente modificados (por meio dos operadores evolutivos) e novos indivíduos são gerados (indivíduos filhos), dando origem a uma nova população $P(t)$. Esse ciclo de seleção, recombinação e avaliação continua ($t = 1, 2, \dots, n$) até que o melhor indivíduo seja encontrado ou algum outro critério de parada seja satisfeito (Mitchell, 1998). Um Algoritmo Evolutivo —AE— típico é representado pelo Algoritmo 4.1.

Algoritmo 4.1 Algoritmo evolutivo

```
1:  $t \leftarrow 0$ 
2: inicializar  $P(0)$ 
3: avaliar  $P(0)$ 
4: enquanto (critério de parada não satisfeito) faça
5:    $t \leftarrow t + 1$ 
6:   selecionar  $P(t)$  de  $P(t - 1)$ 
7:   modificar  $P(t)$ 
8:   avaliar  $P(t)$ 
9: fim-enquanto
```

Cada ciclo de um algoritmo evolutivo corresponde à criação de novos indivíduos a partir dos indivíduos já existentes, utilizando para tanto os operadores evolutivos. Então, os novos indivíduos mais aptos tendem a substituir os indivíduos anteriores e menos aptos. Esse ciclo descrito corresponde a uma geração. Tal como descrito por Darwin, o que ocorre é que os indivíduos mais aptos sobrevivem, enquanto que os indivíduos menos aptos ao meio tendem a desaparecer.

²Os termos *função de avaliação* e *função de aptidão* se referem à função que avalia a aptidão do indivíduo quanto à solução de um determinado problema e, portanto, são termos que possuem significado idênticos. Na maior parte dos casos, a *função objetivo* que é a função a ser otimizada é utilizada como função de avaliação. Nesses casos, os três termos possuem o mesmo significado.

4.1 Indivíduos

Como dito anteriormente, num algoritmo evolutivo a população é formada por um conjunto de indivíduos que são possíveis soluções para o problema. Existem muitas formas de representar os indivíduos, o que dependerá muito do problema sendo solucionado. Frequentemente, os indivíduos são representados por uma sequência de *bits* que codificam uma solução. Assim, a população é composta por várias sequências de *bits* representadas separadamente. O tamanho do indivíduo depende diretamente do número de parâmetros levado em consideração e da variabilidade que esses parâmetros assumem. Quanto maior o número de parâmetros e da variabilidade dos valores dos parâmetros, maior será o tamanho da sequência de *bits*. Fazendo analogia com a área biológica, a sequência de *bits* é chamada de cromossomo, por conter as características de cada indivíduo. Na verdade, os indivíduos são bastante simplificados, de forma que cada cromossomo corresponde a um indivíduo (Goldberg, 1989). Entretanto, uma representação binária não supre as reais necessidades do trabalho aqui proposto. Como o objetivo é utilizar os algoritmos evolutivos como ferramenta na construção de regras de conhecimento, existem duas formas bem conhecidas de definir os indivíduos, que são as abordagens *Pittsburgh* e *Michigan*.

4.1.1 Representação Pittsburgh

Nesta abordagem, cada indivíduo da população representa um conjunto de regras, diferente do que ocorre na abordagem Michigan, na qual cada indivíduo representa uma única regra. Normalmente, a abordagem Pittsburgh é utilizada quando se deseja construir classificadores por meio de algoritmos evolutivos. Em outras palavras, o objetivo do refinamento dos indivíduos consiste em encontrar um bom classificador.

Existe uma grande vantagem nessa representação, conforme explicado por Freitas (2002). Em um bom classificador, é desejável que as regras que o compõem cubram de forma satisfatória uma porção significativa do conjunto de dados. Ao mesmo tempo, deseja-se que essas regras não contenham muita informação redundante. A sobreposição de regras é frequentemente indesejável, pois produz classificadores menos compreensíveis — no sentido de que o classificador é constituído por um conjunto maior de regras que cobrem exemplos comuns. Na Figura 7 é ilustrada a sobreposição de regras. As regras de conhecimento R1, R2, R3 e R4 cobrem uma grande quantidade de exemplos comuns.

A construção de classificadores com esse tipo de regras pode ser contornada com a representação Pittsburgh. Como cada indivíduo representa um conjunto de regras de conhecimento, é possível moldar uma função de avaliação que considere a relação entre essas regras. Dessa forma, indivíduos que representem classificadores com menor sobreposição podem ser considerados superiores aos indivíduos que possuem sobreposição muito elevada. Na representação Michigan, entretanto, a função de avaliação analisa cada indivíduo desconsiderando o relacionamento do mesmo com o resto da população, o que exige a aplicação de outros mecanismos para gerar classificadores constituídos de regras que cobrem exemplos diferentes.

Deve ser observado que, no contexto deste trabalho, a representação Pittsburgh possui um contraponto significativo. Uma vez que cada indivíduo representa um conjunto de regras, é difícil projetar operadores evolutivos capazes de considerar regras individuais. Isso exigiria a implementação de operadores específicos e computacionalmente caros, que observasse a cobertura de cada indivíduo (classificador), considerando a interação entre as regras que o constituem, antes de atuar no indivíduo. Por esse motivo, é proposto neste trabalho o uso da representação Michigan.

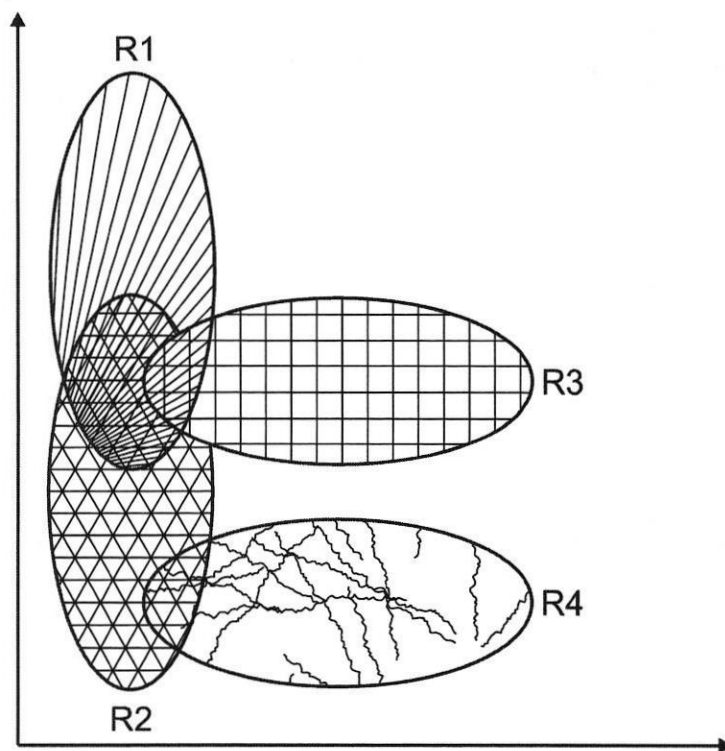


Figura 7: Exemplo de sobreposição de regras

4.1.2 Representação Michigan

Na representação Michigan, cada indivíduo da população é uma regra de conhecimento. Pode ser observado que a representação Michigan ignora completamente a interação entre as regras. Na realidade, uma das características dos algoritmos evolutivos é que eles normalmente não levam em consideração as relações entre os indivíduos.

Já no contexto deste trabalho, a representação Michigan possui a grande vantagem de permitir a atuação direta do algoritmo evolutivo em cada regra de conhecimento. Com isso, cada regra da população pode vir a desenvolver características específicas e desejadas.

4.2 Seleção

Um dos passos mais importantes em um algoritmo evolutivo é o processo de *seleção*. Nesse processo os indivíduos são selecionados para compor a próxima população e a forma como esse processo ocorre pode ser determinante para o sucesso ou fracasso do algoritmo evolutivo. No processo de seleção existe um coeficiente que mede a *pressão da seleção* perante os indivíduos da população (Haupt e Haupt, 1998). Quanto maior a pressão da seleção, mais rapidamente o algoritmo converge para uma solução, a qual não é necessariamente a melhor. Isso ocorre porque a velocidade da convergência é afetada pela pressão da seleção, fazendo com que nem todo o espaço de busca seja adequadamente vasculhado. Por outro lado, uma baixa pressão de seleção faz com que a convergência seja lenta e leve muito tempo para encontrar uma solução adequada.

Existem muitos métodos de seleção. A seguir são descritos os métodos frequentemente utilizados pela comunidade.

4.2.1 Roda da Roleta

No método de seleção roda da roleta, cada indivíduo recebe uma probabilidade de ser sorteado, a qual é calculada pela proporção da aptidão do indivíduo com o total da aptidão acumulada. Dessa forma, a indivíduos de maior aptidão será atribuída uma probabilidade maior, como ilustrado na Figura 8, tal que indivíduos com maior probabilidade possuam maiores chances de serem sorteados. Nessa figura tem-se cinco indivíduos (I_1 , I_2 , I_3 , I_4 e I_5) dispostos de forma a representar suas probabilidades de sorteio quando a roleta girar e o indicador apontar qual é o indivíduo sorteado. Esse método de seleção pressupõe a reposição dos elementos, ou seja, quando um indivíduo é sorteado ele volta a figurar na lista dos possíveis indivíduos a serem sorteados.

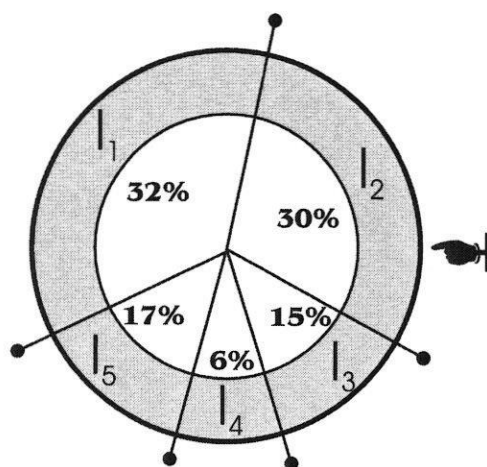


Figura 8: Método de seleção roda da roleta

Nos métodos de seleção para a composição dos indivíduos da próxima população é comum a manutenção do melhor indivíduo da população atual para a nova população. Esse artifício é chamado de *elitismo* e tem por objetivo privilegiar a melhor solução atual de forma a aumentar as chances desse indivíduo propagar as suas características entre os demais indivíduos da população.

Contudo, esse método de seleção apresenta alguns problemas nos casos em que a distribuição das probabilidades de sorteio tendem para extremos: quando um indivíduo possui aptidão destacada em relação aos demais ou quando todos os indivíduos da população possuem aptidões muito próximas (Deb, 2000). Quando um indivíduo possui uma alta aptidão em relação aos demais indivíduos da população, a proporção da aptidão desse indivíduo em relação à aptidão acumulada tende a ser grande, assim como a probabilidade de ser o escolhido. Como os demais indivíduos da população possuem uma aptidão bem menor, se comparada ao *super indivíduo*, a tendência é que esse indivíduo seja selecionado muitas vezes, fazendo com que suas características se propaguem rapidamente pela população e causando uma convergência prematura do algoritmo evolutivo sem que, necessariamente, a solução ótima seja encontrada. Por outro lado, se todos os indivíduos da população apresentam aptidões muito próximas, a probabilidade de cada indivíduo ser sorteado também é aproximadamente a mesma. Logo, a população seguinte será formada, praticamente, pelos mesmo indivíduos da população anterior e a busca, geração após geração, tende a ter um comportamento aleatório (Grefenstette, 2000a).

4.2.2 Torneio

A seleção por torneio é um método muito simples que inicia pela escolha de um número fixo de indivíduos selecionados aleatoriamente da população atual. Então, dentre esses indivíduos, apenas o indivíduo que possui a maior aptidão é copiado para a população seguinte. O processo se repete o número de vezes que for necessário até que a nova população esteja completa (Haupt e Haupt, 1998).

Esse método de seleção pode ser com ou sem reposição dos indivíduos já sorteados. Embora esse método de seleção pareça demandar bastante tempo computacional, sua ordem de complexidade é linearmente proporcional ao tamanho da população, pois, diferentemente do método de seleção roda da roleta, este método independe de uma ordenação prévia dos elementos e do cálculo das probabilidades de seleção (Blickle, 2000).

4.2.3 Ranking

O método de seleção por *ranking* não faz uso de probabilidades proporcionais à aptidão tal como no método de seleção roda da roleta, mas, dependendo do tipo de problema, os indivíduos são ordenados de forma decrescente ou crescente de acordo com os valores obtidos da função de avaliação. Então, para cada indivíduo é atribuído um número inteiro que representa o *ranking* em relação aos demais indivíduos da população. Quanto melhor o *ranking* do indivíduo, melhor sua aptidão em relação ao problema e, portanto, maiores suas chances de sorteio (Freitas, 2002; Grefenstette, 2000b). Neste trabalho são utilizados dois tipos de *ranking* para a seleção dos indivíduos: linear e exponencial.

O *ranking* linear de um indivíduo I_i qualquer é dado pela Equação 27. Nessa equação, *min* e *max* são parâmetros de entrada para o cálculo do *rank*, enquanto que *size* é dado pelo número total de indivíduos da população. Quanto maior a diferença entre *min* e *max* maior será a pressão de seleção e, portanto, a chance de convergência prematura.

$$LinearRank(I_i) = min + (max - min) \times \left(\frac{size - i}{size - 1} \right) \quad (27)$$

Já o *ranking* exponencial de um indivíduo I_i qualquer é dado pela Equação 28. Nessa equação, *base* $\in [0, 1]$. Quanto menor o valor de *base* maior será a pressão de seleção e, portanto, a chance de convergência prematura.

$$ExponentialRank(I_i) = base^{i-1} \quad (28)$$

4.3 Operadores

Como os algoritmos evolutivos são um método de busca baseados na evolução das melhores soluções, os operadores que guiam a busca adotam a mesma filosofia. Existem dois operadores que influenciam na busca por novas soluções: *crossover* e *mutação*.

4.3.1 Crossover

O operador de *crossover* tem o objetivo de gerar dois novos indivíduos a partir de outros dois indivíduos previamente selecionados. Os algoritmos evolutivos partem do pressuposto que a soma das melhores partes de determinadas soluções devem originar a melhor solução. Partindo dessa premissa, o operador de *crossover* combina os melhores indivíduos originando dois novos indivíduos que, a princípio, espera-se que sejam melhores que os indivíduos pais. O que o operador de *crossover* faz é guiar a busca no sentido de encontrar a melhor solução, de forma a recombinar as melhores soluções parciais e encontrar a solução ótima (Carvalho et al., 2003).

Existem vários tipos de operadores de *crossover*. O mais simples é o *crossover* de um ponto. Dados dois indivíduos a serem recombinados, um ponto é aleatoriamente escolhido. Assim, determina-se onde os cromossomos são divididos e compartilham suas partes. Na Figura 9 é mostrado o funcionamento do *crossover* de um ponto, para o caso em que a apresentação é por um conjunto de *bits*.

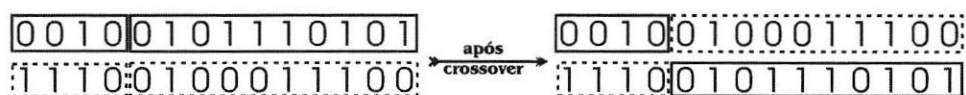


Figura 9: Representação do operador de crossover de um ponto. Dois novos indivíduos são gerados trocando-se as partes destacadas pelos retângulos de traços contínuo e pontilhados

Outra variação desse mesmo operador é o *crossover* de dois pontos. A sistemática é a mesma, entretanto, dois pontos são aleatoriamente selecionados. Nessa variação, obviamente, a recombinação é maior. Na Figura 10 é mostrado o funcionamento do *crossover* de dois pontos.

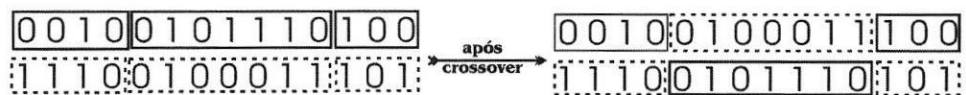


Figura 10: Representação do operador de crossover de dois pontos. Dois novos indivíduos são gerados trocando-se as partes destacadas pelos retângulos de traços contínuo e pontilhados

Partindo desses dois tipos de *crossover*, pode-se generalizar a idéia para o *crossover* de *c*-pontos. Quanto maior o *c* utilizado na definição do *crossover*, maior será o efeito da recombinação na geração dos novos filhos. Entretanto, isso afeta diretamente os esquemas sendo construídos e, portanto, o algoritmo pode demorar, ou talvez nunca convergir para a solução esperada.

A possibilidade dos indivíduos sofrerem *crossover* é determinada pela *taxa de crossover*. Essa taxa informa a probabilidade de ocorrer *crossover* nos pares de indivíduos a cada novo emparelhamento. Normalmente, a taxa de *crossover* é igual para todos os indivíduos durante todas as gerações.

4.3.2 Mutação

Diferentemente do operador de *crossover*, o operador de mutação atua *in loco*, não gerando portanto um novo indivíduo. Se por um lado o *crossover* leva em consideração a hipótese dos blocos de construção e guia a busca na direção do melhor indivíduo, a mutação faz exatamente o oposto. Com base na *taxa de mutação*, que normalmente é um valor de baixa magnitude (0,1% a 0,5%), a cada nova geração a probabilidade de cada novo indivíduo sofrer mutação é considerada segundo essa taxa. Se o indivíduo

for escolhido para sofrer mutação, então um *bit* é aleatoriamente escolhido, o qual é então invertido. Na Figura 11 é representado o operador de mutação.

0 0 1 0 0 1 0 0 0 1 1 1 0 0 $\xrightarrow{\text{após mutação}}$ 0 0 1 0 0 1 0 1 0 1 1 1 0 0

Figura 11: Representação do operador de mutação. Alteração genética *in loco* que inverte o *bit* selecionado

O papel desse operador é inserir diversidade na população, fazendo com que os indivíduos explorem novas áreas do espaço de busca (Michalewicz, 1997). O fato é que o operador de mutação é muito importante para evitar mínimos e máximos locais. Entretanto, não se deve considerar altas taxas de mutação, pois a inserção de muita diversidade na população pode fazer com que ela não convirja para uma solução ótima, fazendo com que a convergência da população oscile indefinidamente.

4.4 Função de Avaliação

A cada nova geração, novos indivíduos são gerados utilizando os operadores de *crossover* e mutação, os quais passam a figurar entre os indivíduos já existentes. Entretanto, é necessário uma forma eficaz de qualificar cada indivíduo como sendo apto ou não à solução do problema. Para isso, os algoritmos evolutivos utilizam a *função de avaliação* para medir a aptidão de cada indivíduo. Como a aptidão para a solução do problema é fortemente ligada à função sendo otimizada, normalmente se utiliza a função de avaliação como sendo a própria função que define o problema (função objetivo). Logo, na maior parte dos problemas, $avaliar(P(t)) = F(P(t))$ (Freitas, 2002).

Deve-se esclarecer que na maior parte dos casos isso ocorre. No entanto, não existe nenhum impedimento quanto à definição de funções de avaliação diferentes das funções que definem o problema propriamente dito. Por exemplo, se a função f fosse dada pela Equação 29.

$$f(x) = x^2 \times \sin(x), -2 \leq x \leq 2 \quad (29)$$

Nesse caso, como a função $f(x)$ assume o valor máximo quando $x = 2$, basta utilizar a função de avaliação como sendo o próprio x para encontrar a solução ótima. A função de avaliação (função de aptidão) é a verdadeira responsável pela obtenção de sucesso de um algoritmo evolutivo em particular, pois serve de guia no processo de busca e exploração do espaço de soluções.

5 O Projeto DISCOVER

Nos últimos anos, diversos trabalhos na área de aquisição automática de conhecimento, a maioria dos quais são realizados de forma cooperativa, vêm sendo desenvolvidos no nosso laboratório de pesquisa LABIC — Laboratório de Inteligência Computacional³. Com o decorrer dos anos, foram observadas algumas sobreposições de implementações. Esses, e outros fatores, motivaram alguns pesquisadores a propor a implementação de um ambiente computacional denominado DISCOVER (Baranauskas e Batista, 2000).

³<http://labic.icmc.usp.br>

O objetivo principal do projeto DISCOVER é reduzir problemas de implementação e integração de ferramentas e algoritmos para apoiar todas as etapas do processo de descoberta de conhecimento, oferecendo funcionalidades voltadas para aprendizado de máquina (Mitchell, 1997), mineração de dados (Weiss e Indurkha, 1998) e mineração de textos (Dörre et al., 1999). O projeto DISCOVER pode ser entendido como um conjunto de métodos para manipular dados e conhecimento por meio de sintaxes padrão (Batista e Monard, 2003b; Batista, 2001; Prati et al., 2001) para a representação de dados e do conhecimento induzido por diferentes algoritmos de aprendizado, bem como por bibliotecas que oferecem um conjunto de funcionalidades básicas para tais manipulações. Assim, foi criado um ambiente integrado, no qual bibliotecas de classes desenvolvidas em PERL (Wall et al., 1996), podem ser integradas pelos usuários do ambiente. Um *framework* para a integração dos componentes do ambiente DISCOVER foi proposta por (Prati, 2003) utilizando *software patterns*, no qual os componentes são integrados por meio de uma linguagem baseada em XML, a qual foi denominada de xDML.

O DISCOVER conta com vários ambientes, tais como o ambiente Discover Learning Environment — DLE, composto pela biblioteca de classes Discover Object Library — DOL, e o ambiente para gerenciamento de experimentos SNIFFER, que foram desenvolvidos para as diferentes tarefas de pré-processamento de dados no formato atributo-valor e análise dos classificadores induzidos por algoritmos de aprendizado utilizados no processo de Mineração (Batista, 2003b). Novas funcionalidades estão sendo especificadas para representação de regras de regressão e regras de associação.

Na Figura 12 é mostrado, de uma forma simplificada, como os filtros, sintaxes e bibliotecas interagem uns com os outros para o caso de algoritmos simbólicos proposicionais de aprendizado supervisionado. Os dados, no formato atributo-valor da sintaxe padrão do DISCOVER, podem ser submetidos a diversos processos de pré-processamento já implementados (Batista, 2003b; Batista e Monard, 2003a; Batista et al., 2004; Prati et al., 2004). Feito isso, esses dados são convertidos para a sintaxe padrão do algoritmo de aprendizado que o usuário deseja executar. O conhecimento (classificador) simbólico induzido por esse algoritmo de aprendizado, é posteriormente convertido para a sintaxe padrão da linguagem de representação de conceitos proposicionais (regras) do DISCOVER (Prati et al., 2001). Esse conhecimento pode ser posteriormente submetido a diversos processos de pós-processamento de conhecimento implementados e que estão sendo implementados. As linhas tracejadas na Figura 12 mostram os processos que utilizam a biblioteca de classes DOL e rotinas implementadas no ambiente DLE do DISCOVER. O RuleSet contém métodos para transformar regras de conhecimento para a sintaxe padrão do DISCOVER — *PBM* — bem como para avaliar essas regras utilizando um conjunto de exemplos.

5.1 A Biblioteca de Classes Discover Object Library — DOL

A biblioteca DOL é uma biblioteca orientada a objeto baseada em padrões de projeto (Gamma et al., 1995). As classes dessa biblioteca implementam as tarefas de manipulação e gerenciamento de dados mais comuns em pré-processamento, tais como gerenciamento de diferentes sintaxes de arquivos de dados e atributos, amostragens, métodos de *resampling*, estatísticas descritivas, normalizações de dados, entre outros (Batista e Monard, 2003b). O objetivo da DOL é ser uma biblioteca de métodos de pré-processamento e dar suporte à criação de novos métodos de pré-processamento de dados.

O conjunto de métodos da DOL permite ao desenvolvedor uma forma simples de ter acesso aos dados, os quais podem estar disponíveis em arquivos texto ou em tabelas de banco de dados relacionais. Os dados são carregados em uma estrutura que pode ser acessada por uma variedade de métodos imple-

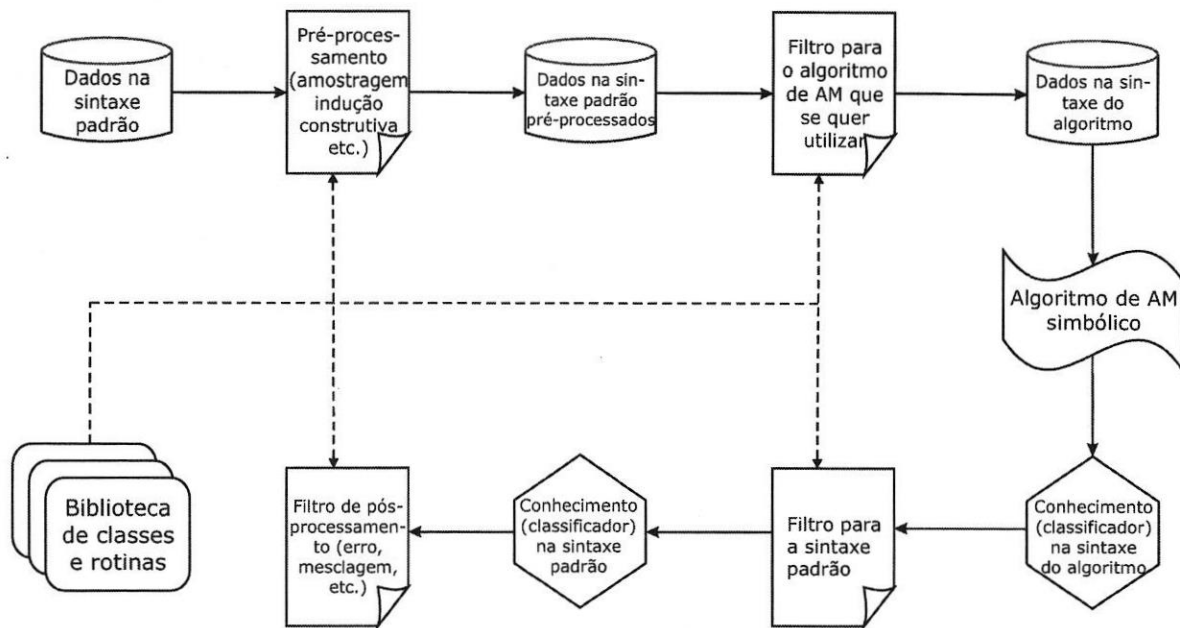


Figura 12: Interação entre filtros, sintaxes e bibliotecas, (Chiara e Monard, 2003)

mentados para manipular essa estrutura. Após, ela pode ser novamente armazenada em arquivos texto, em diferentes sintaxes, ou carregados em uma tabela de banco de dados relacional. Entre as principais funcionalidades da biblioteca DOL pode-se citar: manipulação de atributos e dados, integração com diversos sistemas de aprendizado de máquina, integração com sistemas gerenciadores de banco de dados, filtros de exemplos, estatísticas descritivas e correlações, além de diversos métodos de *resampling*. A biblioteca DOL foi desenvolvida em uma arquitetura modular na qual cada módulo é constituído de uma ou mais classes que realizam um conjunto bem definido de tarefas. A biblioteca possui um módulo central chamado **Core**, o qual carrega um conjunto de dados em uma estrutura e disponibiliza, atualmente, mais de 60 métodos capazes de consultar e manipular essa estrutura. O módulo **Core** é o único que precisa obrigatoriamente ser carregado por uma aplicação que utiliza a biblioteca, os demais módulos são carregados apenas quando for necessário.

5.2 O Ambiente Computacional SNIFFER

O ambiente computacional SNIFFER é um ambiente de gerenciamento de avaliações e comparações experimentais de algoritmos de aprendizado de máquina. O ambiente computacional SNIFFER automatiza a avaliação experimental, e está atualmente integrado com diversos sistemas de AM simbólico, entre eles o *C4.5* (Quinlan, 1988), *C4.5 Rules* (Quinlan, 1987), *ID3* (Quinlan, 1986), *CN2* (Clark e Boswell, 1991), *NewId* (Boswell, 1990), *APRIORI* (Agrawal et al., 1996) e *SVM* (Tsang et al., 2005).

O ambiente SNIFFER complementa a biblioteca DOL pois permite que diferentes indutores proposicionais, bem como diferentes métodos de pré-processamento de dados, sejam avaliados e comparados experimentalmente de uma forma rápida e segura (Batista e Monard, 2003b). As comparações são realizadas utilizando testes estatísticos de significância, permitindo identificar quando um método supera outro com 95% ou 99% de confiança. Deve ser observado que o SNIFFER provê uma *API* (*Application Programming Interface*) que permite estender as funcionalidades do ambiente. Por exemplo, essa *API* pode ser utilizada para avaliar sistemas de aprendizado utilizando outras medidas de desempenho ou

implementar novos testes estatísticos de significância.

5.3 A Biblioteca de Classes Evolutionary Computing Learning Environment — *ECLE*

A biblioteca *ECLE*, tal qual a *DOL*, é uma biblioteca de classes orientadas a objeto. As classes dessa biblioteca implementam os diversos métodos que compõem um algoritmo evolutivo para o fim de aprendizado de máquina, tais como métodos de seleção, *crossover*, mutação e avaliação. O objetivo da *ECLE* é fornecer suporte à execução de experimentos com diversos tipos de parâmetros e métodos existentes na área de computação evolutiva aplicada ao aprendizado de regras de conhecimento com propriedades específicas.

Por ser um conjunto de classes orientadas a objeto, a *ECLE* permite ao desenvolvedor sua fácil extensão e implementação de novos métodos que são automaticamente reconhecidos pelas demais classes que compõem essa biblioteca. Maiores detalhes sobre o projeto e a implementação dessa biblioteca são discutidos a seguir.

6 Definição e Projeto da Biblioteca Evolutionary Computing Learning Environment — *ECLE*

Nas seções anteriores, foram descritos a representação de regras de conhecimento considerada e os conceitos de computação evolutiva necessários ao entendimento deste trabalho. Nesta Seção, é apresentada uma estrutura capaz de codificar regras de conhecimento de forma a facilitar sua manipulação pelo algoritmo evolutivo. Também são apresentadas as definições, os detalhes de implementação e a hierarquia das classes que compõem a *ECLE*, sendo a hierarquia de classes mostrada através de diagramas de classes em *Unified Modeling Language* — *UML* (Booch et al., 1998). Além disso, os principais métodos de execução e acesso a dados são explanados.

A biblioteca de classes *ECLE* foi projetada para trabalhar com regras de conhecimento quaisquer, desde que estas sejam consistentes com a base de dados às quais essas regras se referem. A arquitetura geral para aplicação da *ECLE* está ilustrada na Figura 13. No funcionamento geral, à *ECLE* é fornecido inicialmente um conjunto de regras de conhecimento relacionadas a um conjunto de exemplos rotulados. Esse conjunto de regras pode ser um subconjunto de regras induzidas por diferentes algoritmos de aprendizado supervisionado ou regras propostas pelo usuário da *ECLE*. As regras devem ser fornecidas dentro do formato padrão de regras *PBM* proposto em Prati et al. (2001). Como mencionado, esse formato permite, entre outras, unificar as várias linguagens de descrição de regras de conhecimento adotadas por diferentes algoritmos de aprendizado simbólico. Essas regras são usadas pelo algoritmo evolutivo para a construção de uma população inicial de indivíduos, a qual é iterativamente trabalhada na tentativa de construir regras de conhecimento que atendam melhor a propriedades específicas definidas pelo usuário.

Todo esse ambiente é projetado e implementado segundo o paradigma de orientação a objetos. Cada funcionalidade da *ECLE* é empacotada numa classe bem-definida, conforme ilustrado na Figura 14. Todas as classes da *ECLE* podem ser agrupadas em oito pacotes distintos, de acordo com as funcionalidades implementadas em cada uma delas. São pacotes do *ECLE*: indivíduos e população, métodos de seleção, operadores de *crossover*, operadores de mutação, funções de avaliação, estatísticas e iteradores. A seguir, explica-se cada um desses pacotes, suas classes e métodos mais importantes.

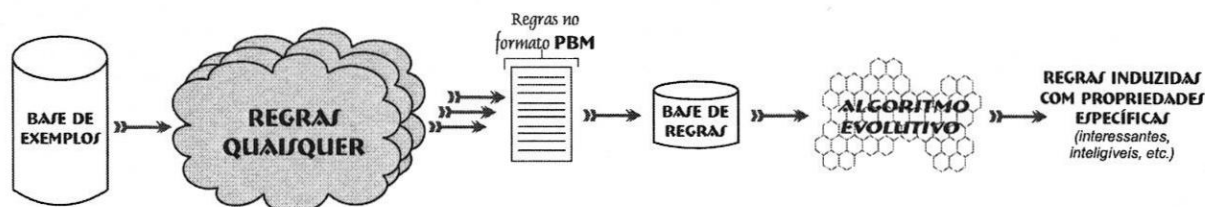


Figura 13: Arquitetura de funcionamento da *ECLE*

6.1 Indivíduos e População

Como dito na Seção 4.1 na página 23, a representação de um indivíduo depende muito do problema a ser resolvido pelo algoritmo evolutivo. A mais comum é a representação binária, na qual cada indivíduo é uma sequência de bits que codifica a presença ou ausência de uma série de fenótipos. Essa não é, entretanto, uma representação adequada para o problema aqui proposto. As regras de conhecimento manipuladas pela *ECLE* são complexas e uma solução mais elaborada se faz necessária. Essa representação deve ser capaz de codificar adequadamente as condições que compõem as regras, as quais contêm símbolos de operadores e valores reais.

A representação de indivíduos proposta neste trabalho se baseia no formato padrão *PBM* para regras de conhecimento, tal qual visto na Seção 2.3 na página 4. Nesse formato, o corpo de cada regra é composto por um conjunto de condições conectadas por conjunções — também chamadas **complexos** (Prati et al., 2001). A cabeça da regra, que é a classe para a qual os exemplos são classificados, é identificada após a palavra *THEN*. Um exemplo desse formato pode ser observado na Figura 15. A regra mostrada nessa figura classifica como favorecido de *empréstimo* todos os exemplos que atendem às condições *casado* e *salário entre R\$ 2.000,00 e R\$ 4.500,00*.

A solução de algoritmo evolutivo proposta codifica o corpo de uma regra como um conjunto de literais e intervalos. Se a regra *PBM* equivalente ao indivíduo impõe restrição a algum atributo qualitativo, o conjunto do indivíduo contém uma literal que restringe o atributo qualitativo ao mesmo valor. Se a regra *PBM* possui uma ou mais condições que impõem restrições a algum atributo quantitativo, o conjunto de literais e intervalos do indivíduo contém um intervalo que restringe o atributo aos mesmos valores.

Na Figura 16 é ilustrada a representação geral de uma regra de conhecimento no formato *PBM*. Nessa representação, todos os atributos presentes no conjunto de dados são mapeados para pares de triplas. Cada par representa o limite inferior e superior que um possível atributo quantitativo possa atingir. No caso de atributos qualitativos, apenas a primeira tripla é utilizado. Dessa forma, todos os indivíduos são codificados com tamanho idêntico, mesmo no caso de regras de conhecimento de tamanhos diferentes. Essa representação de tamanho fixo favorece o *crossover* porque não é necessário fazer o alinhamento das condições dos indivíduos.

As triplas são compostas por um *bit* indicador da presença daquela condição na regra, pelo operador daquela condição e pelo valor limitante daquela condição. Além disso, também fazem parte dessa codificação as matrizes de contingência da regra, as quais contemplam os exemplos com valores conhecidos e desconhecidos, respectivamente.

Existem quatro possibilidades de restrições a atributos quantitativos que podem estar codificados nessa representação. São eles os seguintes:

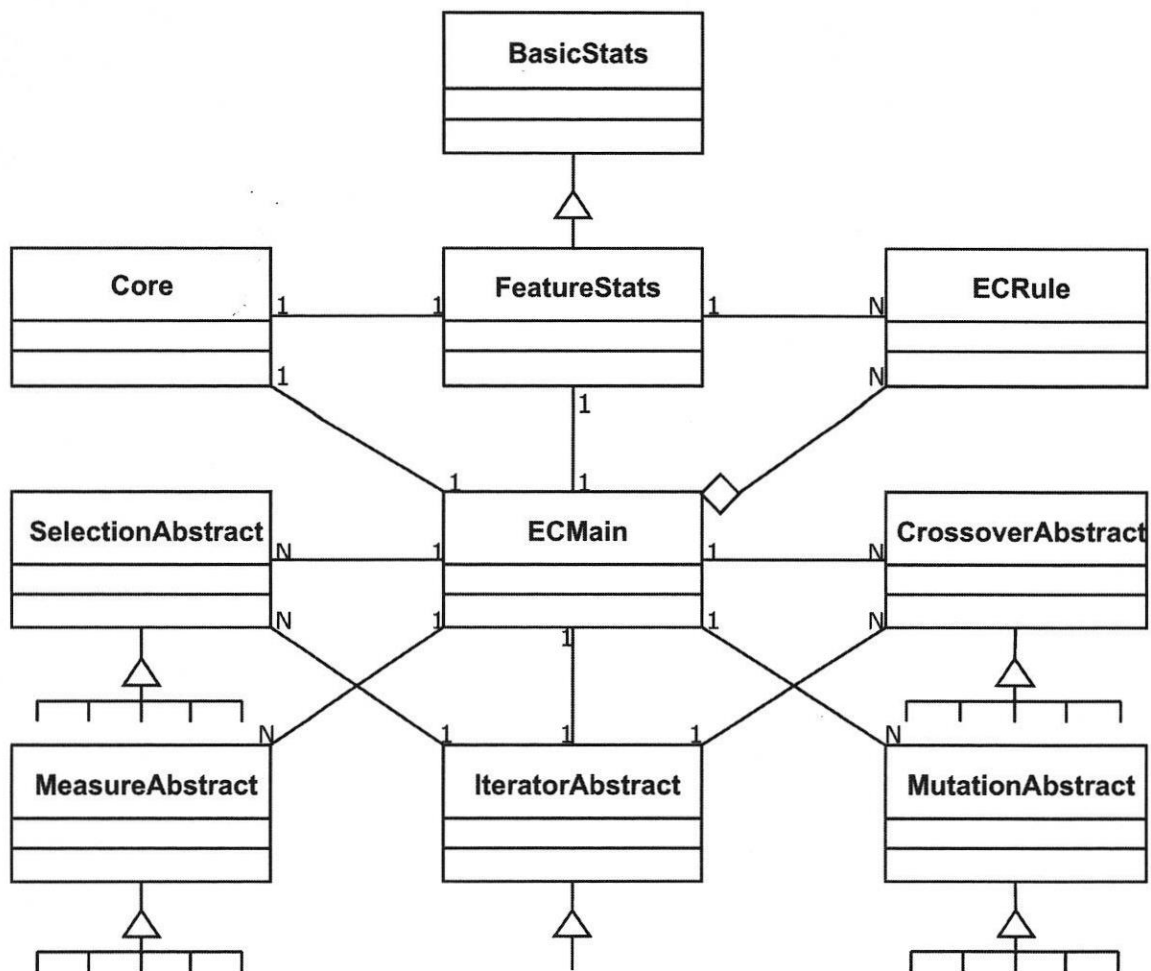


Figura 14: Diagrama de classes da *ECLE*. As classes abstratas possuem classes descendentes que foram omitidas

1. A regra não impõe nenhuma restrição. Nesse caso, o indivíduo equivalente armazena o intervalo $(-\infty, +\infty)$. Qualquer valor presente no exemplo será aceito por esta regra.
2. A regra impõe um valor mínimo para os exemplos, através de uma condição do tipo `atr >= X`. O indivíduo equivalente armazena um intervalo do tipo $[X, +\infty)$.
3. A regra impõe um valor máximo para os exemplos, através de uma condição do tipo `atr <= Y`. O indivíduo equivalente armazena um intervalo do tipo $(-\infty, Y]$.
4. A regra impõe um valor mínimo e um valor máximo para os exemplos, através de duas condições semelhantes aos dos dois casos anteriores. O indivíduo equivalente armazena um intervalo do tipo $[X, Y]$.

```

if  salário >= 2000
   and salário <= 4500
   and casado = sim
then empréstimo = sim

```

Figura 15: Representação de regras de conhecimento no padrão *PBM*

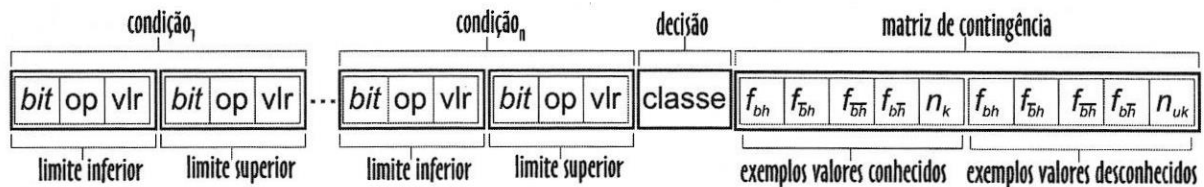


Figura 16: Codificação da regra de conhecimento

No caso de atributos qualitativos, existe somente um tipo de restrição que é feito com o operador de igualdade. Nesse caso, a regra possui uma condição do tipo $atr = X$. O indivíduo equivalente armazena um valor unitário $[X]$.

Para ilustrar melhor como essa codificação é feita, a regra de conhecimento ilustrada na Figura 15 é representada, no algoritmo evolutivo, como mostrado na Figura 17. Vale ressaltar que se o conjunto de dados possuísse outros atributos, no caso dessa regra especificamente, todos os demais atributos não utilizados na composição das condições teriam seus *bits* de presença desabilitados (iguais a 0). Na regra representada nesse exemplo o conjunto de dados utilizado para testar a regra possui 20 exemplos, dos quais 80% são cobertos corretamente pela regra (f_{bh}) e 20% são cobertos incorretamente pela regra ($f_{b\bar{h}}$).

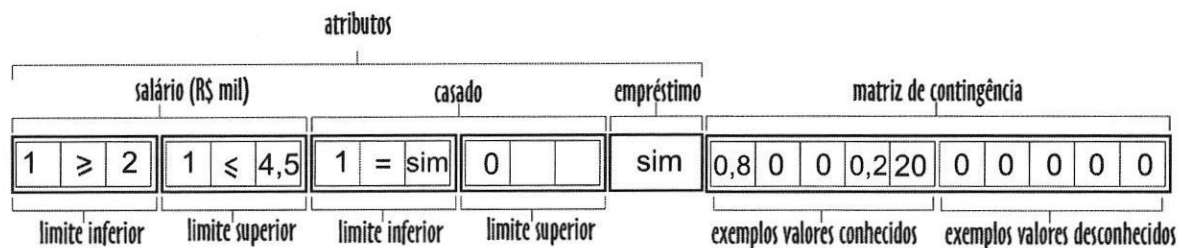


Figura 17: Exemplo de codificação da regra de conhecimento

São duas as classes utilizadas para definir os indivíduos e as populações. As definições dos indivíduos com seus complexos e matriz de contingência são implementados pela classe **ECRule**. Já as populações formadas de indivíduos e os métodos de manipulação das populações são implementados pela classe **ECMain**. Mais detalhes a respeito dessas classes são explicados a seguir.

6.1.1 A classe **ECRule**

A classe **ECRule** armazenam e manipula uma quantidade bastante variada de informações de cada indivíduo. Além do conteúdo genético (complexo), esta classe armazena também a matriz de contingência, a cabeça da regra (classe predita) e o identificador único, ID (número inteiro).

Na seção anterior foi abordada a necessidade de se marcar as condições como não-presentes no indivíduo, mas sem descartar suas informações. A **ECLE** utiliza esses bits de atividade para tentar iniciar os indivíduos com valores coerentes. Por isso, todo indivíduo mantém alguma restrição para cada atributo do conjunto de exemplos, mesmo que a regra por ele codificada não imponha restrição a esse atributo. Isso torna as operações de mutação e *crossover* mais simples e diminui a chance de que elas insiram na população informações inúteis ou mesmo prejudiciais, que podem fazer com que o indivíduo desapareça em poucas gerações. É importante destacar também que essa tarefa não é realizada pela classe **ECRule**.

Todo objeto recém-criado dessa classe possui o complexo completamente vazio. Na verdade, a tarefa de criar esses valores cabe à classe **ECMain**, explicada na próxima seção. Os principais métodos da classe **ECRule** são descritos a seguir.

import_rule(<rule>): importa o corpo de uma regra armazenada como um objeto **Rule** (do **RuleSet**). Os bits de atividade são automaticamente ativados. O complexo importado pode sobrescrever o complexo anteriormente armazenado pela regra **ECRule**.

export_rule_to_evaluation: exporta as condições ativas da regra como um objeto **Rule**. O objeto criado contém a mesma classe e ID que o objeto **ECRule** exportado.

update_contingency_table(<rule>): sobrescreve a matriz de contingência da regra pela matriz de contingência de um objeto de regra **Rule**.

set_condition(<feature>, <value>): define uma condição para algum atributo. O parâmetro **feature** corresponde ao nome do atributo. O parâmetro **value** é dependente de implementação, requerendo somente que ele represente um intervalo numérico para atributos quantitativos ou um valor literal para atributos qualitativos.

equals(<rule>): verifica se a regra é igual à regra **ECRule** passada como argumento. Duas regras são consideradas iguais se cada restrição ativa na primeira é idêntica à restrição do mesmo atributo na segunda e se, para cada restrição inativa na primeira, a restrição equivalente da segunda também está inativa.

print_rule: imprime no dispositivo padrão a regra no formato *PBM*.

calc_active_conditions: calcula o número de condições cujos *bits* de presença estejam ativos. Esse método é útil para decidir quando uma operação evolutiva deve ser desfeita por ter ocasionado uma regra com complexo vazio.

A classe **ECRule** recebe em seu construtor um objeto de **FeatureStats**, o qual é utilizado na construção e alteração do conjunto de condições de forma consistente.

6.1.2 A classe **ECMain**

A tarefa básica da classe **ECMain** é armazenar e permitir acesso fácil às populações de indivíduos utilizadas pelo algoritmo evolutivo. A classe armazena três populações, a saber: população inicial, população intermediária e população final.

O **ECMain** recebe três importantes parâmetros em seu construtor. O primeiro é um objeto da classe **Core**, o qual é instanciado pelo DLE e contém o conjunto de dados. Os outros dois argumentos são uma referência para um objeto de **FeatureStats** e uma referência para um objeto de **RuleSet**.

O **ECMain** é também responsável por criar a chamada “regra padrão”. A regra padrão é um indivíduo que contém um conjunto de condições que atendem, individualmente, à maioria ou à totalidade do conjunto de exemplos, mas cuja regra equivalente possui corpo vazio. Ou seja, a regra padrão possui condições otimizadas para cobertura, mas todos os bits de atividade estão desativados. Ela é construída

com base nos valores mínimos e máximos de cada um dos atributos quantitativos e na moda dos atributos qualitativos. Por exemplo, para um conjunto de dados baseado na população brasileira, no atributo “destreza”, a regra padrão muito provavelmente especificaria a condição “destro” ao invés de “canhoto” ou “ambidestro”. Isso é implementado para evitar que o algoritmo evolutivo insira nas regras condições pouco prováveis ou que não tenham chances de cobrir qualquer exemplo do conjunto de dados, tornando certas operações de mutação potencialmente fatais para o indivíduo, ao invés de atuarem como agente insersor de variabilidade.

Outras duas tarefas de grande importância são exercidas pela classe **ECMain**. Uma delas é exportar os indivíduos sob a forma de um objeto **RuleSet** para análise de eficiência da população. Essa tarefa é realizada através do método `export_rules_to_evaluation`. A outra tarefa de importância é a atualização das matrizes de contingência das regras previamente exportadas. Isso é realizado através do método `update_rules`. Juntos, esses dois métodos permitem que o algoritmo evolutivo possa trabalhar as regras, exportá-las para avaliação e então importar as matrizes de contingência, as quais são utilizadas pelos métodos de medida para avaliar a eficiência das regras geradas. Os principais métodos da classe **ECMain** são os descritos a seguir.

import_rules(<ruleset>): define a população inicial de acordo com o conjunto de regras do objeto de **RuleSet** passado como parâmetro. Para cada regra do **RuleSet** é criado um indivíduo correspondente na população inicial. Esse indivíduo correspondente contém todas as restrições impostas pela regra (condições ativas) acrescidas de todas as condições padrões para os atributos que não têm restrição definida pela regra (essas condições são iniciadas como inativas).

generate_default_rule: cria e armazena uma regra padrão conforme descrito anteriormente. O **ECMain** invoca automaticamente este método quando construído e armazena a regra padrão como uma propriedade. A regra padrão é fornecida a objetos **ECRule** sempre que instanciados.

get_default_rule: retorna uma cópia da regra padrão armazenada pelo **ECMain**.

size_of_population(<population>): retorna a quantidade de indivíduos armazenados em alguma população. O argumento **population** determina a população a ser examinada. Valores válidos são **Initial**, **Intermediary** e **Final**.

individual(<population>, <i>): retorna uma referência para o *i*-ésimo indivíduo da população especificada pelo argumento **population**. É importante que o retorno seja uma referência para que as alterações realizadas no indivíduo retornado se reflitam no indivíduo da população.

find_rule(<rule>, <population>): procura, na população definida pelo argumento **population**, uma regra igual à regra definida pelo argumento **rule**. Utiliza o método **equals** da classe **ECRule**.

append_rules(<rule_set>): adiciona à população o conjunto de regras definido pelo argumento **rule_set**. Esse argumento é um objeto da classe **RuleSet**.

update_rules(<rule_set>, <population>): atualiza a matriz de contingência da população definida pelo argumento **population** utilizando as matrizes de contingência presentes no objeto definido pelo argumento **rule_set**.

export_rules_to_evaluation(<rule>, <population>): esse método faz exatamente o oposto do método anterior, ou seja, exporta para um objeto **rule** as regras contidas na população definida pelo argumento **population**.

verbose([mode], [file_name]): este é um método de acesso que quando invocado sem argumentos retorna um valor lógico indicando se o modo de saída detalhada da execução do algoritmo evolutivo esta ativo. Quando invocado com o argumento **mode**, inabilita a saída detalhada dos resultados (mode=0) ou habilita (mode=1). Quando invocado com o argumento **file_name**, indica qual o arquivo a ser gravado os resultados.

6.2 Métodos de Seleção

Uma vez que os indivíduos da população tenham sido avaliados, o algoritmo evolutivo requer que sejam selecionados os mais aptos para a aplicação dos operadores evolutivos de cruzamento e mutação. Essa tarefa é realizada pelos métodos de seleção, que geram a população intermediária baseados na aptidão dos indivíduos da população inicial.

Similarmente às demais classes, as classes que implementam os métodos de seleção seguem uma hierarquia orientada a objeto com uma classe abstrata (**SelectionAbstract**) e classes derivadas concretas, cada uma especializa num método de seleção. A hierarquia de classes é ilustrada na Figura 18. Os objetos nessa hierarquia seguem um contrato semelhante ao definido para os métodos de seleção. Todos os construtores dessas classes esperam como parâmetro um objeto da classe **ECMain** para que os métodos possam atuar sobre a população de indivíduos. As classes e principais métodos que compõem o conjunto de classes destinados aos métodos de seleção são descritas a seguir.

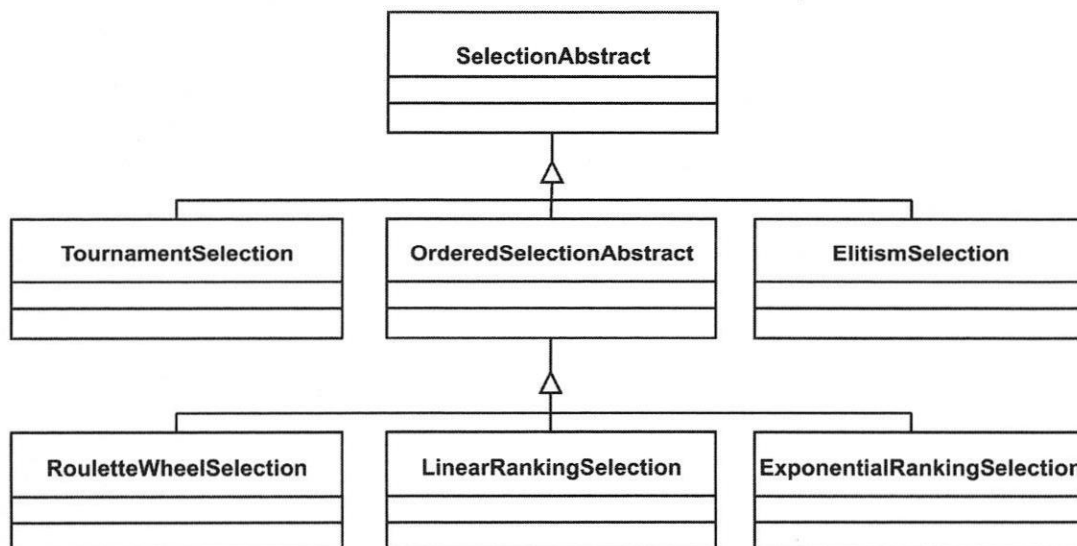


Figura 18: Diagrama de classes dos métodos de seleção

6.2.1 A classe **SelectionAbstract**

A classe **SelectionAbstract** implementa uma série de métodos que são herdados e sobrescritos utilizando polimorfismo. O principal método dessa classe é o **run**.

run: este método constrói toda a população intermediária com base nos indivíduos da população inicial. Ao final da execução desse método, a população intermediária possui o mesmo tamanho da população inicial.

6.2.2 A classe **OrderedSelectionAbstract**

A classe **OrderedSelectionAbstract** é a classe que implementa métodos e atributos herdados pelas classes descendentes que dependem da população inicial estar ordenada em ordem decrescente de aptidão. Os principais métodos desta classe são descritos a seguir.

order: responsável por ordenar os indivíduos em ordem decrescente de valor de aptidão. Esse processo é necessário para os métodos que requerem esse tipo de ordenação dos indivíduos.

get_individual: retorna o melhor indivíduo da população com base na aptidão. No caso de empate, um indivíduo é escolhido aleatoriamente entre os empatados.

6.2.3 A classe **ElitismSelection**

A classe **ElitismSelection** implementa o método de seleção chamado Elitismo, no qual o melhor indivíduo de uma população é copiado imediatamente para a população seguinte para garantir que ao menos uma cópia do indivíduo mais apto esteja presente na geração futura. O principal método dessa classe é o `get_individual`.

get_individual: retorna o melhor indivíduo da população com base na aptidão, o qual será automaticamente copiado para a geração seguinte. Como no caso anterior, em caso de empate, um indivíduo é escolhido aleatoriamente entre os empatados.

6.2.4 A classe **TournamentSelection**

A classe **TournamentSelection** implementa a seleção por torneio descrita na Seção 4.2.2 na página 26. Os principais métodos dessa classe são o `get_individual` e o `set_tournament_size(<size>)`.

get_individual: retorna o melhor indivíduo da população com base na aptidão.

set_tournament_size(<size>): este método ajusta o tamanho do torneio segundo o argumento `size`

6.2.5 A classe **RouletteWheelSelection**

A classe **RouletteWheelSelection** implementa a seleção por roda da roleta descrita na Seção 4.2.1 na página 25. O principal método dessa classe é o `calc_fitness` que calcula a aptidão de todos os indivíduos da população segundo os conceitos desse método. O método `get_individual` é herdado da classe ascendente.

calc_fitness: calcula a aptidão de todos os indivíduos da população.

6.2.6 A classe **LinearRankingSelection**

A classe **LinearRankingSelection** implementa a seleção por *ranking* linear descrita na Seção 4.2.3 na página 26. Os principais métodos dessa classe são descritos a seguir.

calc_fitness: calcula a aptidão de todos os indivíduos da população.

set_range(<min, max>): este método ajusta os valores mínimo e máximo para o cálculo do *rank* segundo os argumentos **min** e **max**.

6.2.7 A classe **ExponentialRankingSelection**

A classe **ExponentialRankingSelection** implementa a seleção por *ranking* exponencial descrita na Seção 4.2.3 na página 26. Os principais métodos dessa classe são descritos a seguir.

calc_fitness: calcula a aptidão de todos os indivíduos da população.

set_base(<base>): este método ajusta o valor da base da potenciação segundo o argumento **base**.

6.3 Operadores de *Crossover*

Os operadores desempenham papel fundamental nos algoritmos evolutivos por serem os agentes diretos na combinação dos indivíduos sub-ótimos em busca de indivíduos ótimos. O operador de *crossover* é normalmente o primeiro operador evolutivo a ser aplicado em indivíduos recém-selecionados. Na *ECLE*, essa tarefa cabe a um conjunto de classes que segue estrutura semelhante aos métodos já apresentado. Existe uma classe abstrata, **CrossoverAbstract**, responsável por fornecer funcionalidades úteis à maioria dos métodos de *crossover*. A partir dela, cada classe derivada concreta implementa funcionalidades específicas a cada método. A hierarquia de classes é ilustrada na Figura 19.

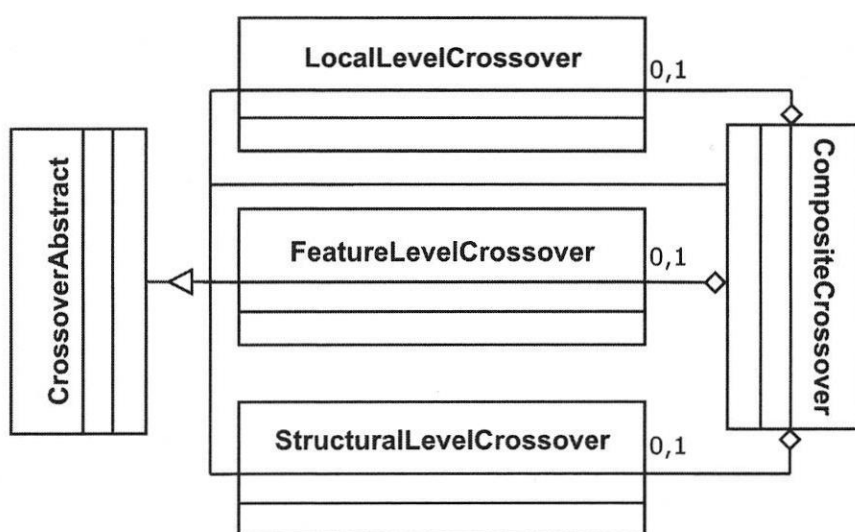


Figura 19: Diagrama de classes dos métodos *crossover*

Neste trabalho são propostos três tipos de *crossover*, cujas definições e classes são apresentados a seguir, bem como as classes abstratas utilizadas para construir a hierarquia de classes.

6.3.1 A classe **CrossoverAbstract**

A classe **CrossoverAbstract** implementa os dois principais métodos para as classes do conjunto de operadores de *crossover*. O método `run` é implementado nesta classe e utilizado através de polimorfismo de herança pelas classes derivadas. O método `do_cross` é implementado nas classes especializadas. Ambos os métodos são explicados a seguir.

run: este método submete os indivíduos da população intermediária dois-a-dois ao método de *crossover* e preenche a população final com seus descendentes. Vale ressaltar que este método, por trabalhar com pares de indivíduos, deve considerar a consistência no caso de populações com número ímpar de indivíduos.

do_cross(<father>, <mother>, <son>, <daughter>, <cross_ratio>): este método seleciona aleatoriamente um ponto de corte com probabilidade definida pelo argumento **cross_ratio**. Os argumentos **son** e **daughter** são dois objetos da classe **ECRule** que, após o processo, contém ponteiros para os dois novos indivíduos gerados a partir dos indivíduos pais passados como argumentos em **father** e **mother**. Nas classes descendentes esse método é sobrescrito para atender as funcionalidades específicas.

set_match_head(<match>): este método recebe como argumento um valor lógico que é utilizado para testar quando o *crossover* deve ser feito somente entre indivíduos que possuam a mesma cabeça (*head*).

set_crossover_rate(<cross_ratio>): o argumento **cross_ratio** é um número entre 0 e 1 que define a probabilidade de ocorrência de cruzamento cada vez que dois indivíduos são emparelhados.

6.3.2 A classe **StructuralLevelCrossover**

A classe **StructuralLevelCrossover** implementa o *crossover* de **nível estrutural**. Neste tipo de *crossover*, os indivíduos filhos são originados dos indivíduos pais que foram cruzados e tiveram seu material genético trocado no nível das condições. Dessa forma, pares inteiros de limitantes inferiores e superiores são trocados entre os pais para originarem os indivíduos filhos. Na Figura 20 é ilustrado este processo. Os indivíduos pais — Figura 20 (a) — são alinhados e o ponto de cruzamento aleatoriamente escolhido faz com que os indivíduos filhos — Figura 20 (b) — sejam gerados de forma que o primeiro indivíduo filho contenha a primeira porção do indivíduo pai inferior com a segunda porção do indivíduo pai superior. O segundo filho é obtido de maneira análoga. Para fins de clareza, no exemplo ilustrado pela Figura 20, as matrizes de contingência foram omitidas.

6.3.3 A classe **FeatureLevelCrossover**

A classe **FeatureLevelCrossover** implementa o *crossover* de **nível de atributo**. Neste tipo de *crossover*, os indivíduos filhos são originados dos indivíduos pais que foram cruzados e tiveram seu material genético trocado no nível dos atributos. Dessa forma, limitantes inferiores e superiores são trocados entre os pais para originarem os indivíduos filhos. Na Figura 21 é ilustrada este processo. Os indivíduos

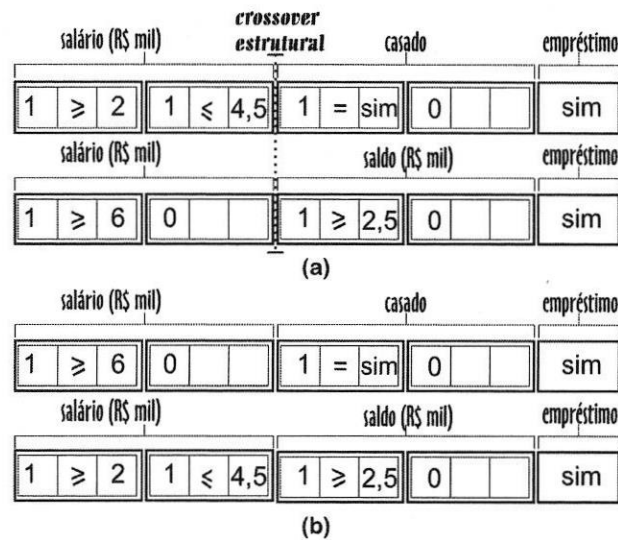


Figura 20: Operador de cruzamento em nível estrutural

pais — Figura 21 (a) — são alinhados e o ponto de cruzamento aleatoriamente escolhido faz com que os indivíduos filhos — Figura 21 (b) — sejam gerados de forma que o primeiro indivíduo filho contenha a primeira porção do atributo **salário** do indivíduo pai inferior com a segunda porção do atributo **salário** do indivíduo pai superior. O segundo filho é obtido de maneira análoga. Neste processo os demais atributos permanecem inalterados.

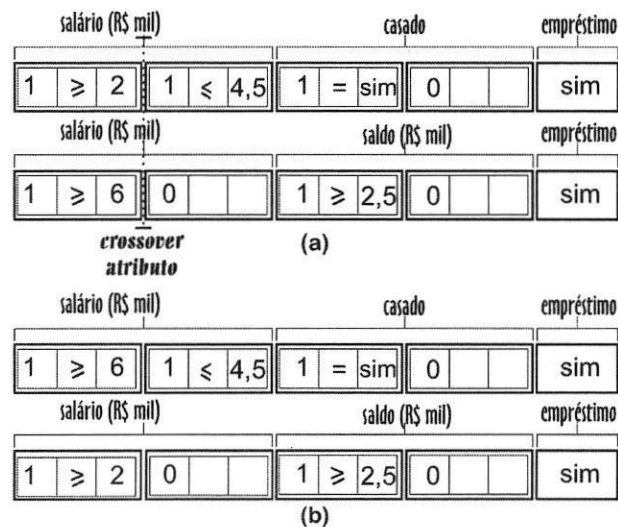


Figura 21: Operador de cruzamento em nível de atributo

6.3.4 A classe **LocalLevelCrossover**

A classe **LocalLevelCrossover** implementa o *crossover* de **nível local**. Neste tipo de *crossover*, os indivíduos filhos são originados dos indivíduos pais que foram cruzados e tiveram seu material genético trocado no nível dos valores locais que compõem as condições. Dessa forma, somente os valores de algum dos limitantes é recombinado. Na Figura 22 é representada este processo. Os indivíduos pais — Figura 22 (a) — são alinhados e o ponto de cruzamento aleatoriamente escolhido faz com que os indivíduos filhos — Figura 22 (b) — sejam gerados de forma que para o limitante selecionado ambos filhos

tenham uma mistura do valor inicial.

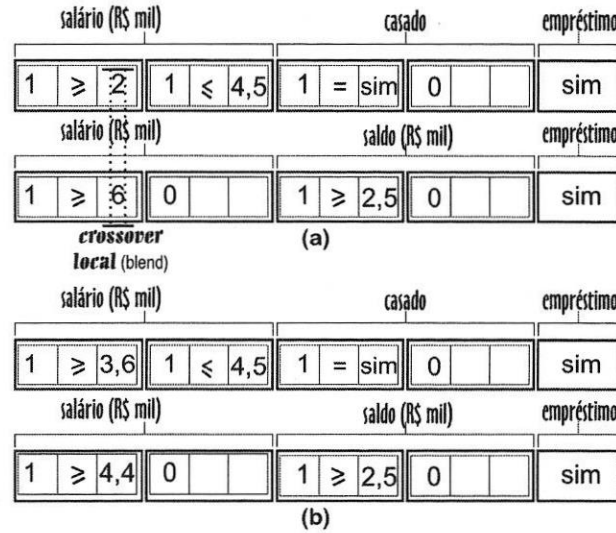


Figura 22: Operador de cruzamento em nível local

O *crossover* de nível local implementado neste trabalho é o *BLX- α* (Herrera et al., 2002). Neste *crossover*, dados dois números reais iniciais (x e y), dois novos números (x' e y') são gerados pela mistura (*blend*) que depende do fator α de mistura. Como pode ser observado na Equação 30, os novos números são calculados utilizando uma fração da diferença existente entre os números iniciais.

$$\left. \begin{aligned} x' &= x - \alpha \times (x - y) \\ y' &= y + \alpha \times (x - y) \end{aligned} \right\} \quad (30)$$

Para exemplificar, supondo $x = 8$, $y = 2$ e $\alpha = 0.3$. Aplicando a equação anterior, os valores de x' e y' seriam calculados conforme a Equação 31.

$$\left. \begin{aligned} x' &= x - \alpha \times (x - y) = 8 - 0.3(8 - 2) = 8 - 1.8 = 6.4 \\ y' &= y + \alpha \times (x - y) = 2 + 0.3(8 - 2) = 2 + 1.8 = 3.8 \end{aligned} \right\} \quad (31)$$

Esta classe provê especificamente o seguinte método.

set_alpha_rate(<alpha>): este método é utilizado para armazenar o valor da constante de mistura passada através do argumento **alpha**.

6.3.5 A classe **CompositeCrossover**

A classe **CompositeCrossover** é utilizada para implementar uma forma de gerenciar e permitir o uso de mais de um método de *crossover* para cada geração. Dessa forma, uma mesma população pode sofrer a influência de qualquer combinação dos métodos de *crossover* descritos anteriormente. Ainda permite que outros métodos de *crossover* criados posteriormente possam ser utilizados nessa composição. Esta classe provê especificamente os seguintes métodos.

add(<obj_crossover>): este método é utilizado para adicionar métodos de *crossover* que são

posteriormente executados sobre todos os indivíduos de uma mesma população. O argumento **obj_crossover** é um objeto de uma das classes de *crossover* apresentadas anteriormente.

run: este método executa seqüencialmente sobre a população todos os métodos de *crossover* que foram adicionados utilizando o método `add(<obj_crossover>)`.

6.4 Operadores de Mutação

O segundo operador evolutivo é o operador de mutação. Como mencionado, sua função é inserir variabilidade na população, alterando os indivíduos aleatoriamente após a aplicação do operador *crossover*. A estrutura de classes na *ECCE* designada a implementar essa funcionalidade segue a mesma hierarquia dos operadores de *crossover*. Uma classe abstrata, **MutationAbstract**, implementa funcionalidades comuns a todos os métodos, enquanto as demais classes implementam funcionalidades específicas. A hierarquia de classes é ilustrada na Figura 23.

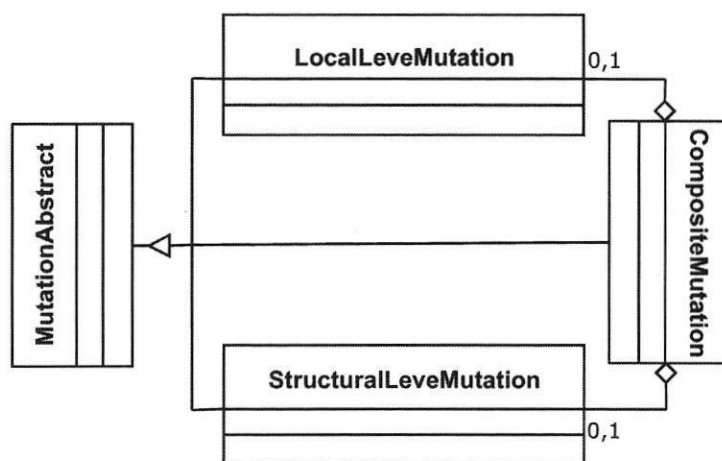


Figura 23: Diagrama de classes dos métodos de mutação

Neste trabalho são propostos dois tipos de mutação, cujas definições e classes são apresentados a seguir, bem como a classes abstrata utilizada para construir a hierarquia de classes.

6.4.1 A classe MutationAbstract

A classe **MutationAbstract** implementa os dois principais métodos para as classes do conjunto de operadores de mutação. O método `run` é implementado nesta classe e utilizado através de polimorfismo de herança pelas classes derivadas. O método `do_mut` é implementado nas classes especializadas. Ambos os métodos são explicados a seguir.

run: este método submete todos os indivíduos da população final ao método de mutação. A mutação é aplicada em todos os indivíduos com uma taxa de mutação que é constante para todas as gerações.

do_mut(<ind>, <mutation_ratio>): este método seleciona aleatoriamente um ponto do indivíduo para a mutação. O argumento **ind** é um objeto da classe **ECRule**. A mutação ocorre com taxa de mutação determinada pelo argumento **mutation_ratio**, o qual já foi inicialmente determinado pelo método descrito a seguir.

set_mutation_rate(<mutation_ratio>): o argumento **mutation_ratio** é um número entre 0 e 1 que define a probabilidade de ocorrência da mutação.

6.4.2 A classe **StructuralLevelMutation**

A classe **StructuralLevelMutation** implementa a mutação de **nível estrutural**. Neste tipo de mutação, o indivíduo sofre uma mudança *in-loco* e o *bit* de presença é ativado ou desativado, dependendo do estado inicial quando o método de mutação é invocado. Dessa forma, um indivíduo pode ter o número de condições aumentado ou diminuído. Na Figura 24 é representado este processo. O indivíduo — Figura 24 (a) — tem um ponto escolhido aleatoriamente. Este ponto deve coincidir com o *bit* de presença daquele indivíduo, o qual terá seu estado invertido — Figura 24 (b).

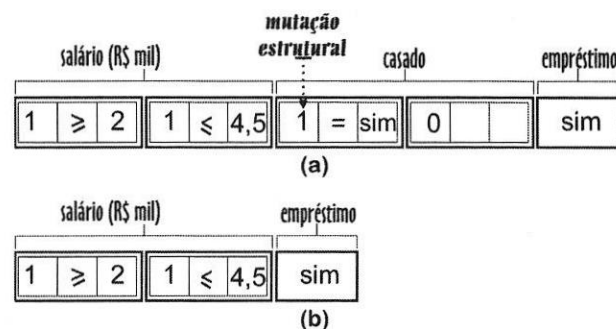


Figura 24: Operador de mutação em nível estrutural

6.4.3 A classe **LocalLevelMutation**

A classe **LocalLevelMutation** implementa a mutação de **nível local**. Neste tipo de mutação o indivíduo sofre uma mudança *in-loco* e o resultado é uma alteração no valor de comparação da condição selecionada. Dessa forma, somente o valor de algum dos limitantes é alterado. A alteração é feita considerando se o atributo que forma a condição é qualitativo ou quantitativo. No caso das condições com atributos qualitativos, a mutação é feita de forma que o valor passe a ser um dos outros possíveis valores que aquele atributo possa assumir. No caso das condições com atributos quantitativos, a mutação é feita de forma que o valor sofra uma alteração para mais ou para menos, dependendo de uma escolha aleatória. O valor a ser incrementado ou decrementado, depende da coleta amostral feita pela classe **FeatureStats** que guarda a menor diferença existente entre dois valores quaisquer dentre todos os valores de um determinado atributo.

Na Figura 25 é representado este processo. O indivíduo original — Figura 25 (a) — sofre mutação no limitante superior do atributo salário, passando a ser um indivíduo que possui outro valor no limitante superior do atributo salário — Figura 25 (b).

6.4.4 A classe **CompositeMutation**

A classe **CompositeMutation** é utilizada para implementar uma forma de gerenciar e permitir o uso de mais de um método de mutação para cada geração. Dessa forma, uma mesma população pode sofrer a influência de qualquer combinação dos métodos de mutação descritos anteriormente. Ainda, permite que

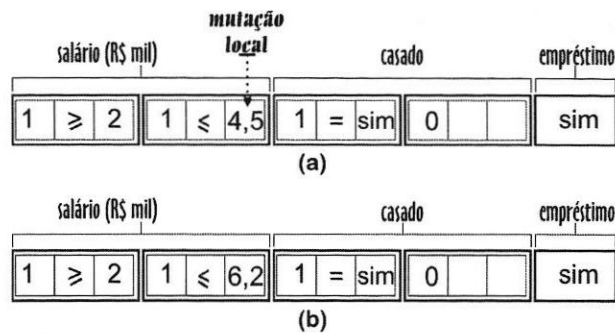


Figura 25: Operador de mutação em nível local

outros métodos de mutação criados posteriormente possam ser utilizados nessa composição. Esta classe provê especificamente o seguinte método.

add(<obj_mutation>): este método é utilizado para adicionar métodos de mutação que são posteriormente executados sobre todos os indivíduos de uma mesma população. O argumento **obj_mutation** é um objeto de uma das classes de mutação apresentadas anteriormente.

run: este método executa sequencialmente sobre a população todos os métodos de mutação que foram adicionados utilizando o método **add(<obj_mutation>)**.

6.5 Funções de Avaliação

Com o objetivo de proporcionar facilidades para a execução de experimentos variados e bastante personalizados, a *ECLE* disponibiliza uma grande quantidade de funções de avaliação. Todas essas funções utilizam como base as medidas descritas na Seção 2.6 na página 6. Cada medida é implementada na *ECLE* como uma classe cujo nome tem relação direta com alguma das medidas descritas na Seção 2.6 na página 6. Por exemplo, a função de avaliação que utiliza a medida de novidade (“novelty”, em inglês) está implementada na classe **NoveltyMeasure**.

As funções de avaliação são organizadas numa hierarquia de classe que possui uma classe abstrata (**MeasureAbstract**) como ancestral comum e classes derivadas que implementam as funcionalidades específicas de cada método. A hierarquia das classes de medidas de avaliação é ilustrada na Figura 26 na página seguinte.

Todas as classes de medidas instanciáveis possuem o método **calc(<ind>)**. Esse método recebe um objeto **ind** de **ECRule** como argumento e retorna um valor decimal que representa o valor calculado de sua aptidão de acordo com a medida da classe. Ele não realiza nenhuma modificação no indivíduo em si ou nas populações armazenadas pelo objeto **ECMain**. Isso é útil para a implementação de medidas genéricas que utilizem outras medidas como base.

6.5.1 A classe **MeasureAbstract**

Como dito anteriormente, a classe **MeasureAbstract** é utilizada para executar o cálculo das medidas de todos os indivíduos da população. Além do cálculo das medidas (aptidão dos indivíduos), os objetos

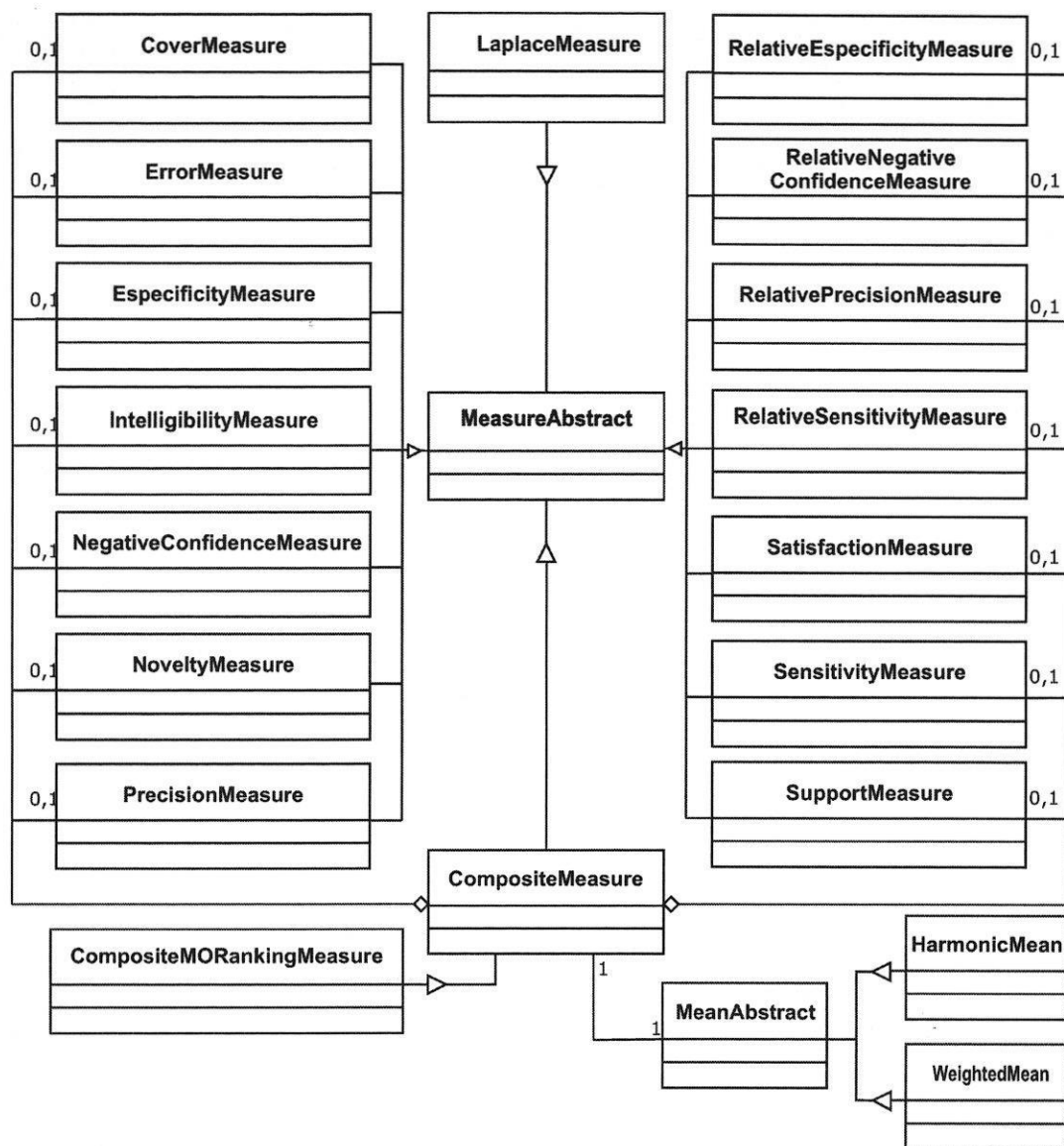


Figura 26: Diagrama de classes das medidas de avaliação

dessa classe armazenam na forma de atributos os valores de aptidão ótimo, médio e pior. Os principais métodos dessa classe são os descritos a seguir.

run: o objetivo desse método é automatizar a avaliação da população, calculando a aptidão do indivíduo através do método **calc** e armazenando o resultado em cada indivíduo. O processo é executado uma vez para cada regra contida na população inicial. Além disso, este método é o responsável por atualizar os atributos que armazenam os valores de aptidão ótimo, médio e pior.

analyze_fitness: este método retorna uma lista com os valores de aptidão ótimo, médio e pior.

set_mean: define um objeto da classe **MeanAbstract** para o cálculo da média da medida composta. O conceito de medida composta é explicado mais adiante.

6.5.2 As classes **Measure**

As classes que implementam as medidas definidas na Seção 2.6 são as listadas a seguir. Obviamente, as medidas equivalentes não foram implementadas e, portanto, não são listadas.

- **PrecisionMeasure** — Equação 1 na página 7.
- **LaplaceMeasure** — Equação 2 na página 7
- **ErrorMeasure** — Equação 3 na página 7.
- **NegativeConfidenceMeasure** — Equação 4 na página 7.
- **SensitivityMeasure** — Equação 5 na página 7.
- **SpecificityMeasure** — Equação 6 na página 7.
- **CoverMeasure** — Equação 7 na página 7.
- **SupportMeasure** — Equação 8 na página 8.
- **NoveltyMeasure** — Equação 9 na página 8.
- **SatisfactionMeasure** — Equação 10 na página 8.
- **RelativePrecisionMeasure** — Equação 11 na página 8.
- **RelativeNegativeConfidenceMeasure** — Equação 12 na página 8.
- **RelativeSensitivityMeasure** — Equação 13 na página 8.
- **RelativeSpecificityMeasure** — Equação 14 na página 9.
- **IntelligibilityMeasure** — Equação 20 na página 10.

6.5.3 A classe **CompositeMeasure**

A classe **CompositeMeasure** calcula a aptidão dos indivíduos com base em outras medidas de avaliação implementadas no sistema. A classe **CompositeMeasure** formula a aptidão do indivíduo com base na composição de outras medidas de avaliação implementadas na *ECLÉ*. Um objeto associado da classe **MeanAbstract** permite especificar como essas medidas são compostas e uma média é formulada. As possíveis duas médias usadas neste trabalho são as fornecidas pelas classes **WeightedMean** e **HarmonicMean**, relativas às médias aritmética e harmônica definidas pelas Equações 25 e 26 na página 20, respectivamente. Dessa forma, uma gama de funções de avaliação pode ser configurada com diferentes ponderações.

A adição e remoção de objetos de medida à composição é feita através do método `set_term`. A classe associa a cada objeto componente um valor que pode ser utilizado como peso no cálculo da aptidão combinada. O peso associado a algum objeto de medida pode ser recuperado através do método `get_term`. O objeto de média deve ser instanciado pelo usuário da classe e associado a esta classe através do método **MeasureAbstract::set_mean**.

A avaliação desta classe é feita através dos métodos `run` e `calc`. Do ponto de vista do usuário da classe, ambas funcionam como descrito para a classe **MeasureAbstract**. Segue uma breve descrição dos métodos das classes do conjunto de funções de avaliação.

calc(<individual>): calcula e retorna a aptidão do indivíduo segundo o método de seleção específico da classe. Para o caso específico da classe **CompositeMeasure**, efetua a avaliação de cada uma das medidas componentes e retorna a combinação dessas medidas através do objeto de média previamente associado.

run: chama o método `calc` de cada indivíduo da população inicial e armazena neles a aptidão medida.

set_term(<measure>, <value>): somente para a classe **CompositeMeasure**. Este método define o peso da medida `measure` com o valor especificado por `value`. Se o valor especificado for zero, o termo é removido do cálculo da média.

get_term(<measure>): somente para a classe **CompositeMeasure**. Retorna o peso definido para a medida desejada.

6.5.4 A classe **CompositeMORankingMeasure**

A classe **CompositeMORankingMeasure** calcula a aptidão dos indivíduos com base em outras medidas de avaliação implementadas no sistema. Entretanto, a aptidão é calculada utilizando uma composição dos *rank*s que cada uma das medidas assume, utilizando as médias aritmética e harmônica. Em resumo, o que o método `calc(<individual>)` dessa classe faz é o processo descrito na Seção 3.8 na página 18 para compor *rank*s.

6.6 Estatísticas — A Classe **FeatureStats**

As estatísticas relacionadas aos atributos são obtidas através da classe **FeatureStats**. Essa classe provê estatísticas importantes para outras classes da *ECCE*, como os nomes dos atributos do conjunto de dados e suas características principais, tipo do atributo (qualitativo ou quantitativo), valor mínimo da amostra, valor máximo da amostra e média amostral (para atributos quantitativos) ou moda (atributos qualitativos).

A classe **FeatureStats** é uma especialização da classe **BasicStats**. A classe **BasicStats** faz parte da DOL e também fornece estatísticas sobre o conjunto de dados. A diferença é que a primeira tem um foco muito mais voltado aos atributos do que aos valores da amostra. A classe **FeatureStats** abstrai uma série de passos para a obtenção de valores e os oferece para o resto do sistema através de métodos de acesso. Dos métodos presentes nessa classe, destacam-se os descritos a seguir.

refresh(): analisa o conjunto de dados, recolhendo as estatísticas que serão posteriormente utilizadas pelos métodos de acesso da classe. Este método é invocado automaticamente por objetos da classe logo após serem instanciados. O usuário da classe deve fazer chamadas explícitas a este método quando houver alterações no conjunto de dados — incluindo alteração de índice do Core.

feature_family(<atributo>): recebe o nome de um atributo da base de dados e retorna “qualitative” ou “quantitative”, dependendo se o atributo recebe valores qualitativos ou quantitativos, respectivamente.

feature_type(<atributo>): recebe o nome de um atributo da base de dados e retorna o tipo do atributo.

max(<atributo>): retorna o valor máximo amostral de um atributo quantitativo.

min(<atributo>): retorna o valor mínimo amostral de um atributo quantitativo.

std_dev(<atributo>): retorna o valor do desvio padrão amostral de um atributo quantitativo.

mean(<atributo>): retorna a média amostral de um atributo quantitativo.

mean_per_class(<atributo>, <classe>): retorna a média amostral de um atributo quantitativo considerando a distribuição das classes.

minor_diff(<atributo>): retorna a média amostral de um atributo quantitativo.

mode(<atributo>): retorna a moda (valor mais comum na amostra) de um atributo qualitativo.

get_feature_names("ordered"): retorna uma lista contendo os nomes de todos os atributos encontrados no conjunto de exemplos. O argumento **ordered** indica se os atributos devem ser ordenados alfabeticamente. Se não houver a necessidade de ordenação, basta omitir o argumento.

random_value(<atributo>): retorna um valor pseudo-aleatório relacionado ao atributo quantitativo.

random_value_prop(<atributo>): retorna um dos possíveis valores que considera a distribuição do atributo qualitativo.

possible_values(<atributo>): retorna uma lista contendo os possíveis valores amostrais de um atributo qualitativo.

O construtor da classe requer como parâmetro real uma única referência a um objeto da classe **Core**.

6.7 Iteradores

No paradigma da Programação Orientada a Objetos, *iteradores* são objetos que coordenam acesso sequencial em estruturas de dados, impondo restrições de consistência e ocultando detalhes de implementação. Na *ECLE*, uma abstração dessa idéia é a base do conjunto de iteradores, um grupo de classes que serve de abstração para a execução sequencial do algoritmo evolutivo.

O conjunto dos iteradores é formado de classes que seguem uma hierarquia semelhante àquela já apresenta em conjuntos anteriores. Uma classe abstrata, **IteratorAbstract**, fornece funcionalidades comuns a todos os iteradores. Classes descendentes mais especializadas implementam funcionalidades específicas. A hierarquia das classes dos iteradores é ilustrada na Figura 27 na página seguinte.

6.7.1 A classe **IteratorAbstract**

A classe **IteratorAbstract** possui a maioria dos métodos para que o algoritmo evolutivo funcione a contento. É nesta classe que ocorre a ordenação lógica da chamada de todos os demais métodos já apresentados e discutidos: seleção, *crossover*, mutação, medidas e estatísticas. Os critérios de parada

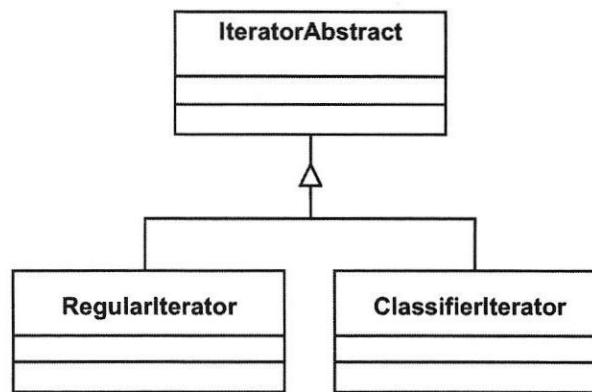


Figura 27: Diagrama de classes de iteração

também são definidos nesta classe. Vale ressaltar que, uma vez definidos os valores dos critérios de parada, o algoritmo evolutivo para quando pelo menos um deles é satisfeito. Isso dependerá de quantos critérios foram definidos. A classe **IteratorAbstract** possui os seguintes métodos disponíveis.

do_measure: utiliza o objeto de medida fornecido para avaliar a população inicial e calcular a aptidão de cada indivíduo.

do_select: utiliza o objeto de seleção fornecido para montar a população intermediária.

do_crossover: caso haja um objeto de *crossover* instanciado, seu método **run** é chamado para aplicar o operador de *crossover* nos indivíduos selecionados e montar a população final. Caso não haja um método de *crossover* selecionado, simplesmente copia os indivíduos da população intermediária para a final.

do_mutate: caso haja um objeto de mutação instanciado, seu método **run** é chamado para aplicar o operador de mutação na população final.

do_elitism: caso haja um objeto de elitismo instanciado, seu método **run** é chamado para aplicar o operador de elitismo na população intermediária.

set_measure(<obj>): define qual será o objeto de medida a ser utilizado na avaliação dos indivíduos. Esse objeto é passado através do argumento **obj**.

set_selection(<obj>): define qual será o objeto de seleção a ser utilizado na formação da população intermediária. Esse objeto é passado através do argumento **obj**.

set_crossover(<obj>): define qual será o objeto de *crossover* a ser utilizado. Esse objeto é passado através do argumento **obj**.

set_mutation(<obj>): define qual será o objeto de mutação a ser utilizado. Esse objeto é passado através do argumento **obj**.

set_elitism(<obj>): define qual será o objeto de elitismo a ser utilizado. Esse objeto é passado através do argumento **obj**.

set_stop_iterations(<num>): define o número máximo (**num** inteiro) de iterações que o algoritmo evolutivo deve executar.

set_stop_optimum(<num>): define o valor máximo (**num** real) a ser encontrado durante a execução do algoritmo evolutivo. Não há parada até que este valor seja alcançado.

set_stop_convergence(<num>): define o valor (**num** é um número entre 0 e 1) do critério de convergência. Este valor é utilizado para testar a convergência do algoritmo segundo a estabilidade da solução ótima. Se a diferença for menor que **num** então o critério é satisfeito.

6.7.2 A classe **RegularIterator**

A classe **RegularIterator** faz uso dos métodos herdados da classe ancestral e implementa o métodos `run` e `iterate` que controlam a execução do algoritmo evolutivo com base nos critérios de parada definidos.

A classe **RegularIterator** possui os seguintes métodos disponíveis.

iterate: este método executa uma única iteração do algoritmo evolutivo, aplicando seleção, *crossover*, mutação e a avaliação à uma população. Uma nova população inicial e, espera-se, mais apta, é obtida ao término desse ciclo.

run: este método invoca seguidamente o método `iterate` até que um dos critérios de parada previamente definidos seja satisfeito.

6.7.3 A classe **ClassifierIterator**

A classe **ClassifierIterator** tem o objetivo de utilizar o algoritmo evolutivo e a *ECLE* para a geração de um classificador. O classificador é gerado incrementalmente e de acordo com as medidas de qualidade de regras especificadas para a execução do algoritmo evolutivo, seguindo o Algoritmo 6.1. A classe **ClassifierIterator** implementa os seguintes métodos públicos:

Algoritmo 6.1 Algoritmo evolutivo para gerar um classificador

- 1: $D \leftarrow$ Exemplos do Conjunto de Dados
 - 2: $P(0) \leftarrow$ População Inicial de Regras
 - 3: $C \leftarrow \{\}$
 - 4: $Melhor_regra \leftarrow$ Algoritmo_evolutivo($P(0)$, D)
 - 5: **enquanto** ($\#Exemplos_cobertos_corretamente(Melhor_regra, D) > 1$) **faça**
 - 6: $Melhor_regra \leftarrow$ Algoritmo_evolutivo($P(0)$, D)
 - 7: $D \leftarrow D - \{Exemplos_cobertos(Melhor_regra)\}$
 - 8: Adicionar_regra(C , $Melhor_regra$)
 - 9: **fim-enquanto**
-

next_rule: obtém uma nova regra através da execução normal do algoritmo evolutivo. Caso a regra cubra algum exemplo, ela é adicionada à população de nome `Classifier_population` sob o objeto ativo da classe `ECMain`. O método retorna o número de exemplos cobertos pela regra.

run: invoca repetidamente o método `next_rule` até que o critério de parada definido para o iterador seja alcançado. O classificador é mantido sobre o objeto ativo da classe `ECMain` sob a população `Classifier_population`.

7 Implementação da Biblioteca *ECLL*

Segue-se a descrição de alguns detalhes sobre a implementação realizada do que foi proposto nas seções anteriores. Note que uma implementação completamente diferente da aqui apresentada poderia ser feita, contanto que toda as classes definidas no projeto tivessem seus métodos implementados.

A linguagem de programação PERL realiza gerenciamento automático de memória. Por isso, embora muitas classes aqui descritas façam alocação dinâmica de objetos de classes, tabelas *hash* e vetores, nenhum método de desalocação é fornecido. Essa tarefa é realizada automaticamente pelo *garbage collector* (Wall et al., 1996). Os detalhes de implementação são apresentados somente em casos para os quais for oportuna essa explanação. Portanto, nesta seção não serão discutidas todas as classes.

7.1 Convenções Utilizadas na Implementação

Na implementação das classes e métodos em PERL foram definidas algumas convenções que pudessem padronizar e facilitar a escrita e leitura do código por futuros usuários. O formato padronizado dos nomes dos identificadores visa reconhecer-los rapidamente a que tipo pertencem: métodos públicos, métodos privados, atributos, variáveis, classes ou objetos. Além do formato dos identificadores, existe uma convenção para a documentação interna das classes.

7.1.1 Nomes dos Identificadores

Os nomes dos identificadores de métodos, variáveis, objetos e classes seguem a seguinte convenção.

Métodos Públicos: os identificadores relacionados aos nomes dos métodos públicos são todos escritos em letras minúsculas, sendo que quando houver a necessidade de definir um nome composto, este deve ser separado pelo símbolo de grifo (`_`). Por exemplo, `analyse_fitness`.

Métodos Privados: os identificadores relacionados aos nomes dos métodos privados possuem a mesma convenção dos métodos públicos, adicionando o símbolo de grifo (`_`) no início do nome do identificado. Por exemplo, `_stat_per_class`.

Atributos: os identificadores relacionados aos nomes dos atributos devem sempre iniciar por uma letra maiúscula, sendo o restante do nome do identificador em letras minúsculas. Quando houver a necessidade de definir um nome composto, este deve ser separado pelo símbolo de grifo (`_`). Por exemplo, `Feature_name`.

Variáveis e Objetos: os identificadores relacionados aos nomes das variáveis e objetos em geral são todos em letras minúsculas.

Classes: os identificadores dos nomes das classes iniciam sempre por uma letra maiúscula e o restante do nome do identificador deve ser em letras minúsculas. Quando os nomes de classes forem formados por palavras compostas, utiliza-se letras maiúsculas infixas. Por exemplo, `LocalLevel-Crossover`.

7.1.2 Documentação Interna

Existem dois tipos de comentários que formam a documentação interna das classes: cabeçalho das classes e cabeçalho dos métodos. O cabeçalho das classes é formado pelo nome da biblioteca (Evolutionary Computing Learning Environment — *ECLE*), o nome da classe (*Package*), o autor da classe (*Author*) e o propósito da classe. Um exemplo é dado a seguir.

```
#####
# ECLE - Evolutionary Computing Learning Environment
#
# Package:      FeatureStats.pm
# Authors:      Adriano Donizete Pila [pila@icmc.usp.br]
#               Rafael Giusti [rg@grad.icmc.usp.br]
# Application:  FeatureStats class. Stores statistics for
#               each feature and provides some methods to easily
#               recover those stats.
#
#####
```

O cabeçalho dos métodos é formado pelo nome do método (*Sub name*), uma descrição sucinta do funcionamento (*Quick-info*), os parâmetros de entrada (*Input*), as saídas produzidas (*Output*) e uma descrição mais detalhada do funcionamento, quando conveniente (*Detailed info*). Um exemplo é dado a seguir.

```
#####
# Sub name:      feature_family
# Quick-info:    Returns the feature family
# Input:         Feature name.
# Output:        The feature family.
# Detailed info: --
#
#####
```

7.2 Indivíduos e População

O conjunto de indivíduos (população) possui as classes **ECRule** e **ECMain**. Ambas possuem detalhes específicos a esta implementação. Esses detalhes são explicados a seguir.

7.2.1 A Classe ECRule

A estrutura dos indivíduos descrita na Seção 6.1.1 na página 34 é implementada na *ECLE* através da classe **ECRule**. O atributo **Body** é uma tabela *hash* que correlaciona os nomes dos atributos do conjunto de dados às condições que as regras lhes impõem. Cada condição é, por sua vez, uma matriz 2×3

onde as linhas podem representar um intervalo, para valores quantitativos, ou uma literal, para valores qualitativos.

O código em PERL para acessar a coluna `$col` da linha `$line` do atributo `$attr` de uma regra referenciada pela variável `$rule` é o seguinte:

```
$rule->{Body}{$attr}{$line}[$col]
```

Note que essa referência é mostrada aqui apenas para conhecimento. É possível acessar esta mesma informação através dos métodos **get_feature** e **set_feature** sem violar o encapsulamento.

Cada regra armazena também uma ID que o *RuleSet* adiciona ao corpo das regras quando as imprime. As regras da *ECLE* também contêm um atributo que identifica à qual classe os exemplos que atendem às condições que ela impõe pertencem. Essas informações são armazenadas nos atributos **ID** e **Head**, respectivamente.

A matriz de contingência, por sua vez, é armazenada nos objetos da classe na forma de oito atributos, sendo quatro deles para exemplos com valores conhecidos e os outros quatro para exemplos com valores desconhecidos. Esses atributos são os seguintes:

- **Know_h_b**: exemplos para os quais o corpo da regra e cabeça correspondem. Ou seja, as restrições da regra aceitam os valores do exemplo e decidem corretamente sua classe.
- **Know_h_nb**: exemplos para os quais a classe da regra corresponde, mas o corpo não.
- **Know_nh_b**: exemplos para os quais o corpo da regra corresponde, mas a classe não.
- **Know_nh_nb**: exemplos não admitidos pela regra.
- **Know_n**: número de exemplos com valores conhecidos.
- **Unknow_h_b**: exemplos desconhecidos para os quais a cabeça e o corpo da regra correspondem.
- **Unknow_h_nb**: exemplos desconhecidos para os quais a cabeça corresponde, mas o corpo da regra não.
- **Unknow_nh_b**: exemplos desconhecidos para os quais o corpo da regra corresponde, mas a cabeça não.
- **Unknow_nh_nb**: exemplos desconhecidos não admitidos pela regra.
- **Unknow_n**: número de exemplos com valores desconhecidos.

Essas propriedades são utilizadas pelas classes de medidas para o cálculo da função de avaliação dos indivíduos. Não há funções de acesso.

Há somente duas características exclusivas de implementação na classe **ECRule**. A primeira delas está relacionada ao construtor. Como a linguagem PERL não controla a aridade das funções, o construtor da classe admite um segundo parâmetro opcional. Se ele for fornecido, o construtor o entende como um objeto de **ECRule** e se torna um construtor de cópia. Nesse caso, ao invés de um novo indivíduo com condições nulas, gera uma cópia da regra passada como parâmetro.

A outra característica está relacionada ao método **set_condition**. Nesse método, a natureza do parâmetro **value** é completamente indiferente ao projeto. Sua implementação, no entanto, precisa de uma natureza bem-definida.

Nesta implementação da *ECLE*, o parâmetro **value** é na verdade expandido nos seguintes três outros argumentos.

1. Operador de condição: uma string representando um dos possíveis operadores do conjunto {<, =, <=, <>, >, >=}. O operador implicitamente aponta a linha da matriz da condição: os quatro primeiros se referem à primeira linha, os restantes, à segunda.
2. Valor da condição: uma string, representando o valor literal do atributo, para o caso de atributos qualitativos ou um número real, para o caso de atributos quantitativos. A linguagem PERL admite que se passe, para este argumento, o valor **undef**, que representa um “valor indefinido”⁴. Se isso for feito, a função ignora o valor da condição e mantém o valor anterior ao acesso. Isso permite alterar o valor da condição sem modificar o bit de atividade.
3. Bit de atividade da condição: “1” para incluir a condição à regra. “0” para removê-la. É importante ressaltar que, no caso de atributos quantitativos, isso equivale a limitar um dos extremos do intervalos a um número real ou estendê-lo ao infinito. Este argumento também admite o valor **undef**. Isso permite ativar ou desativar a condição sem alterar seu conteúdo.

Assim, uma possível linha de comando, cuja finalidade é ativar o atributo “*GENDER*” com a condição “*MALE*”, seria a seguinte:

```
$rule->set_condition('GENDER', '=', 'MALE', 1)
```

O seguinte exemplo ilustra o uso do valor **undef**:

```
$rule->set_condition('AGE', '>', undef, 0)
```

Nesse exemplo, o atributo quantitativo “*AGE*” tem sua restrição desativada no extremo inferior, fazendo o intervalo se estender ao infinito negativo. Isso significa que a regra não impõe nenhuma idade mínima para o atributo “*AGE*” do conjunto de exemplos. O valor **undef** garante que o limitante inferior não seja alterado, permitindo que ele seja utilizado pelos operadores de *crossover* e *mutação*.

7.2.2 A Classe **ECMain**

A implementação das populações descritas na Seção 6.1.2 é feita através de três atributos da classe **ECMain**: **Initial_population**, **Intermediary_population** e **Final_population**, correspondentes às populações inicial, intermediária e final, respectivamente. Cada um destes atributos é um vetor de referências para objetos da classe **ECRule**.

⁴ A linguagem PERL não é “tipada”. Os dados podem assumir qualquer tipo e serem cambiados com poucas restrições. Um mesmo parâmetro formal é capaz de receber strings, números reais ou ponteiros nulos

O método de acesso **individual** retorna um indivíduo de alguma das três populações. É importante observar que o indivíduo armazenado é uma *referência* a um objeto **ECRule**. Assim, qualquer alteração realizada na variável retornada por **individual** será refletida na população.

Em qualquer função do **ECMain** que faça referência às populações, uma string é utilizada para identificar a população a que se refere. Os valores “*Initial*”, “*Intermediary*” e “*Final*” são utilizados para identificar as populações inicial, intermediária e final, respectivamente.

A chamada “regra padrão” é armazenada no atributo **Default_rule**. Assim que esta classe é instanciada, o método **generate_default_rule** é executado e a regra padrão é criada. Essa regra é passada como argumento para cada objeto de **ECRule** instanciado durante a importação de regras do *RuleSet*.

7.3 Métodos de Seleção

A implementação do conjunto de métodos de seleção possui sete classes, sendo cinco implementações de métodos de seleção e as classes abstratas **SelectionAbstract** e **OrderedSelectionAbstract**. A classe **SelectionAbstract** é ancestral comum a todos os métodos de seleção e implementa a função **run**. A classe **OrderedSelectionAbstract** é ancestral comum à todas as classes que implementam métodos de seleção que necessitam ter a população de indivíduos ordenada. Nessa classe, o método **order** ordena os indivíduos da população inicial por *fitness*.

A função **run** chama o método **get_individual**, presente em cada classe concreta do conjunto de métodos de seleção, e insere o indivíduo retornado na população intermediária.

As quatro classes responsáveis pelos métodos de seleção possuem a função **get_individual**. Quando chamada, essa função seleciona um indivíduo de acordo com as especificações do método adotado e retorna uma referência para o mesmo.

A adição de métodos de seleção é feita através da especialização da classe **SelectionAbstract**. É necessário também implementar um método **get_individual** para a escolha dos indivíduos mais adaptados.

7.4 Função de Avaliação

Somente a classe **ComposeMeasure** possui detalhes específicos de implementação, explicados a seguir.

A classe **ComposeMeasure** mantém uma tabela *hash* na qual são armazenados detalhes sobre os objetos de medidas utilizados na composição da função de avaliação. Para cada entrada na tabela, o nome do método é a chave de busca onde o objeto de medida instanciado e o peso dessa medida na função são armazenados. O atributo **Measure_list** contém a tabela.

As funções **set_term** e **get_term** gerenciam a lista de métodos de medida. Elas mantêm também o atributo **Total_weight** consistente. Esse atributo indica sempre a somatória dos pesos das medidas usadas na composição da função de avaliação.

No que diz respeito ao método **run**, ele é herdado de **MeasureAbstract** assim como as outras classes de medida. O método **calc**, por sua vez, simplesmente calcula a média ponderada de todos os métodos de medida selecionados.

De forma similar, a classe **ComposeMORankingMeasure** armazena os objetos de medidas a serem considerados na composição, uma vez que essa classe é uma descendente de **ComposeMeasure**. A diferença está na forma como o método **calc** utiliza os objetos de medidas para o cálculo do *fitness*.

8 O Ambiente Computacional SnifferECLE

O ambiente computacional SnifferECLE é composto por duas classes que em conjunto são capazes de executar experimentos configuráveis por meio de arquivos de configuração. Nesta seção são descritas as classes **SnifferECLE** e **Configure**, fornecidas por um arquivo, o qual contém uma gramática de descrição

8.1 A Classe SnifferECLE

A classe **SnifferECLE** é uma especialização da classe **Sniffer** pertencente à DOL (Batista, 2003a; Batista e Monard, 2003c). Esta classe reutiliza de sua classe ancestral alguns métodos e implementa serviços específicos da *ECLE*. Quando um objeto de **SnifferECLE** é instanciado, este objeto automaticamente invoca o seu método **run**. Este comportamento segue o contrato do construtor da classe **Sniffer**.

O método **run** configura as opções de execução do SnifferECLE, tais como o nome de diretório que identifica um experimento *ECLE* (valor **ecle** da propriedade **ecledir** por padrão) e o caminho do programa de testes do *ECLE* (valor `"/home/local/dle/ECLE/ecle_experiment.plx"` da propriedade **ecletestprogram**, por padrão).

Em seguida, o SnifferECLE inicia a busca por experimentos no diretório indicado para o programa `"sniffer.plx"` através de parâmetros. É importante que os experimentos sejam organizados de acordo com uma estrutura de diretórios padrão, porque é por meio dela que o SnifferECLE reconhece os sistemas que deve executar. Uma ilustração dessa organização é apresentada na Figura 28.

Os arquivos de configuração, o conjunto inicial de regras a ser utilizado e o conjunto de dados a serem particionados e validados devem estar armazenados num diretório com o nome **ecle** (ou outro configurado na propriedade **ecledir**). Se houver mais de um conjunto de dados ou de regras a ser validado, cada um deve ser mantido em seu próprio diretório **ecle**, de modo que um nível superior de diretórios se faz necessário. Por exemplo, para um experimento sobre os conjuntos de dados *breast* e *iris*, os arquivos necessários poderiam ser armazenados, respectivamente, nos diretórios `"/exp/breast/ecle"` e `"/exp/iris/ecle"`.

Quatro arquivos são essenciais para o SnifferECLE e o executor de testes da *ECLE*: um arquivo de nomes, um arquivo de dados, um arquivo de configuração e um ou mais arquivos de regras.

Arquivo de nomes (extensão .names): utilizado pelo **Core** para analisar os atributos do conjunto de exemplos.

Arquivo de dados (extensão .data): contém os dados que serão particionados através do método *5x2-cross-validation*, descrito mais adiante, em conjuntos de pares de arquivos de teste e dados.

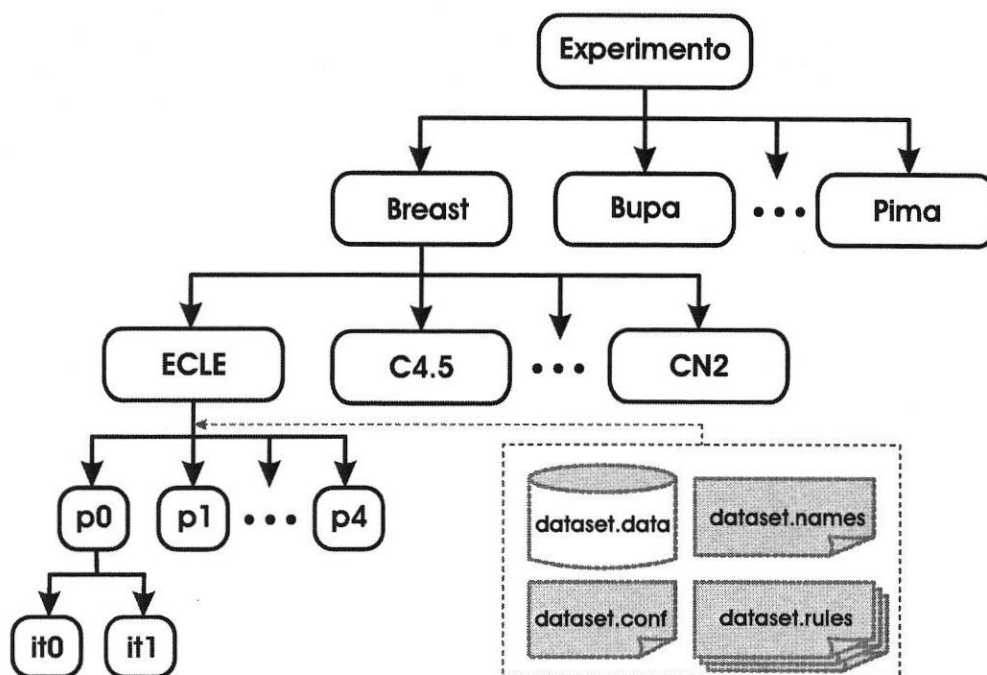


Figura 28: Esquema de funcionamento do SnifferECLE

Arquivo de configuração (extensão .conf): contém as instruções para que o SnifferECLE possa selecionar as populações de indivíduos do experimento e para a determinação dos parâmetros do algoritmo evolutivo.

Arquivo de regras (extensão .rules): pelo menos um arquivo de regras deve ser fornecido. Esse arquivo funciona como um conjunto de indivíduos.

Como pode ser visto na Figura 28, existem 5 populações subdivididas em 2 iterações. Essa divisão em 5 pastas subdivididas em outras 2 outras é conhecida como *5×2-cross-validation*. Na verdade o *5×2-cross-validation* é um método estatístico para validar a hipótese de consistência do modelo construído. Neste trabalho o modelo é uma regra de conhecimento com propriedades específicas. Nesse método, o conjunto de dados é dividido em 2 partes (conjunto de treinamento e conjunto de teste), sendo que a construção da regra de conhecimento é realizado utilizando os exemplos presentes no conjunto de treinamento. Quando o algoritmo evolutivo converge, o modelo é validado sobre o conjunto de teste. Feito isso, o processo se inverte: os exemplos pertencentes ao conjunto de treinamento passam a ser o conjunto de teste e os de teste passam a ser os exemplos do conjunto de treinamento. De forma análoga, a construção da regra de conhecimento é realizada e validada. Esse processo se repete outras 4 vezes. Ao final, tem-se 10 estatísticas de validação de diferentes regras que foram construídas sobre diferentes partições do conjunto de dados inicial.

Para cada instância do *5×2-cross-validation*, o SnifferECLE seleciona um conjunto de indivíduos (de acordo com as configurações do arquivo .conf) e monta a população a ser desenvolvida pelo algoritmo evolutivo. O arquivo de configuração é bastante flexível, permitindo muito mais do que simplesmente concatenar o conteúdo dos arquivos .rules. Ao encontrar os arquivos .names e .data, o SnifferECLE verifica a existência dos diretórios de instâncias do 5x2. Caso esses diretórios não sejam encontrados, o SnifferECLE cria cinco diretórios, de nomes p0, p1, ..., p4 e seleciona um conjunto de indivíduo de acordo com as regras do arquivo .conf. Em cada um desses diretórios é criado um arquivo .rules e feita

cópias dos arquivos `.data` e `.names`. Por fim, para cada instância do *5×2-cross-validation*, o método de `KFolding` do `Sniffer` é executado, particionado o conjunto de dados em dois sub-conjuntos. Com a estrutura de diretórios formada (ou já previamente formada) o `SnifferECLE` invoca o processo do executor de testes da *ECLE* (definido pelo parâmetro `ecletestprogram`). Esse programa lê e valida o conjunto de regras, instancia as classes da *ECLE* e inicia o experimento.

É importante destacar que toda a preparação e condução do experimento é definida pelo arquivo de configuração. A função de avaliação, os métodos de seleção, *crossover* e medida, bem como o critério de parada, estão definidos nesse arquivo. A leitura do arquivo e instanciação das classes é feita pela classe `Configure`. O arquivo de configuração define também o nível de informação que será impresso pela *ECLE* (`verbose` com valor 1 para informações detalhadas ou valor 0 para informações resumidas). O único método público desta classe é descrito a seguir.

run : inicia a execução dos experimentos de acordo com a configuração especificada pelo arquivo de configuração.

8.2 A Classe `Configure`

A classe `Configure` implementa uma série de métodos privados capazes de reconhecer uma gramática de descrição da configuração do experimento. Utilizando a linguagem definida por essa gramática, descrita mais adiante, é possível informar ao `SnifferECLE` os objetos que devem ser construídos relacionados aos possíveis métodos de seleção, *crossover*, mutação, avaliação e iteração do sistema. Além disso, é possível informar os parâmetros reais das chamadas a esses métodos. Esta classe prove publicamente os seguintes métodos.

new(<obj_ecmain>, <file_conf>) : este é o construtor da classe, ao qual devem ser passados como argumentos um objeto da classe `ECMain`, definido pelo argumento `obj_ecmain`, e o nome de um arquivo que contenha as configurações do experimento a ser executado pelo `SnifferECLE`, definido pelo argumento `file_conf`.

run: carrega as configurações definidas no arquivo passado como argumento para o construtor da classe.

8.3 O Arquivo de Configuração de Experimentos

A despeito da relativa facilidade em utilizar as classes da *ECLE*, a configuração de experimento através da criação de programas executáveis na linguagem PERL exige do usuário conhecimentos razoáveis dessa linguagem de programação. Por essa razão, a *ECLE* fornece uma classe que prepara experimentos através de arquivos de configuração que seguem uma linguagem muito mais simples e próxima da linguagem natural do que a linguagem PERL. Esses arquivos são chamados arquivos de configuração de experimento da *ECLE* (*E2CF*).

Cada linha de um arquivo *E2CF* possui um comando para configuração de algum parâmetro do experimento. Esses comandos definem a seleção dos indivíduos do conjunto de regras (comando `select`), escolhem os métodos utilizados pelo algoritmo evolutivo (comando `use`), adicionam parâmetros para os

métodos compostos, como **CompositeMeasure** (comando `add`) e determinam o critério de parada do algoritmo evolutivo (comando `stop`).

8.3.1 O comando **verbose**

A *ECCE* é capaz de gerar informação de depuração para todas as operações que executa. O comando **verbose** faz com que os comandos tenham ou não tenham sua execução registrada. A sintaxe para o comando **verbose** é a seguinte:

```
verbose on ['<arquivo>'] off
```

O parâmetro **on** *ativa* o registro de comandos subsequentes. Todos os comandos terão seus passos de execução detalhadamente registrados. Opcionalmente, é possível escolher um arquivo no qual os passos de execução serão gravados. Caso nenhum arquivo seja especificado, a *ECCE* grava o registro no arquivo `./verbose` do mesmo diretório onde se encontra o *dataset*. O parâmetro **off** *desativa* o registro de comandos.

O comando **verbose** pode aparecer inúmeras vezes ao longo do arquivo de configuração. Na ocasião de o registro ser desativado quando já se encontra desativado (ou o oposto, sendo ativado já estando ativado), o comando extra é ignorado. Isso quer dizer que, para alternar o arquivo do registro, o modo *verbose* deve ser explicitamente desativado e em seguida explicitamente ativado.

8.3.2 O comando **seed**

O comando **seed** permite a especificação de uma semente definida para o gerador de números pseudo-aleatório da *ECCE*. Isso permite a repetição de experimentos idênticos e a avaliação de parâmetros sem que a sequência pseudo-aleatória seja uma variável adicional ao experimento. A sintaxe para o comando **seed** é a seguinte:

```
seed <semente>
```

Sendo que *semente* deve ser um número inteiro. Caso o comando **seed** seja omitido, a *ECCE* gera uma semente automaticamente com base no *timer* do sistema. Para utilização real do sistema, é ideal que este comando *seja* omitido.

8.3.3 O comando **select**

Todo experimento da *ECCE* inicia com a seleção de um conjunto de indivíduos para compor a população inicial. O comando **select** é poderoso o suficiente para permitir que essa seleção seja realizada de uma maneira bastante flexível. É possível, por exemplo, realizar uma amostragem sem reposição de um determinado conjunto de regras e uma amostragem com reposição de um segundo conjunto de regras, desde que os conjuntos estejam em arquivos distintos. Duas sintaxes são permitidas para o comando **select**:

```
select all|<num> [individuals] [with reposition|without reposition] [from  
'<file>']
```

```
select all|<num> [individuals] [from '<file>'] [with reposition|without
```


reposition]

Sendo que `all` determina que todos os indivíduos do conjunto serão selecionados para a população inicial. Nesse caso, o parâmetro `reposition` deixa de ser significativo. A palavra `individuals` é opcional e serve para propósito puramente estético.

O parâmetro `num` determina a quantidade de indivíduos do conjunto que será selecionada para a população inicial.

O parâmetro `with reposition` especifica que a amostragem deverá ser feita com reposição. Isso significa que um mesmo indivíduo pode ser selecionado mais de uma vez de um conjunto. Este parâmetro é opcional e padrão.

O parâmetro `without reposition` especifica amostragem sem reposição. Isso significa que um mesmo indivíduo só pode ser selecionado uma vez de um conjunto pelo comando `select`. Entretanto, se mais de um comando `select` operar sobre um único conjunto ou se o indivíduo estiver duplicado em um ou mais conjuntos, ele poderá aparecer diversas vezes na população inicial. Além disso, é importante que a quantidade de indivíduos selecionada sem reposição seja menor ou igual ao total de indivíduos do conjunto, ou o experimento será abortado. Este parâmetro é opcional. Se omitido, a *ECLE* realizará amostragem com reposição.

O comando `from '<file>'` especifica o conjunto sobre o qual o comando `select` opera. Se omitido, a *ECLE* escolhe o arquivo de mesmo nome que o arquivo de configuração. Se o arquivo não for encontrado, o experimento será abortado. Além disso, é importante especificar o nome entre aspas simples, ou o comando não será reconhecido. Também, é importante observar que o comando `select` é obrigatório e deve aparecer pelo menos uma vez no arquivo de configuração. Exemplos de uso deste comando são:

select all: seleciona todos os indivíduos de arquivo cujo nome é idêntico ao nome do arquivo de configuração. Se o arquivo de configuração for `dataset.conf`, por exemplo, este comando operará sobre o arquivo `dataset.rules`.

select 10 individuals from 'iris.rules': seleciona dez indivíduos do arquivo `iris.rules`.
A amostragem é realizada com reposição.

select 9 without reposition from 'breast.rules': seleciona dez indivíduos sem reposição do arquivo `breast.rules`.

select 8 from 'more.rules' without reposition: seleciona oito indivíduos sem reposição do arquivo `more.rules`.

8.3.4 O comando use

O comando `use` especifica quais métodos serão utilizados pelo algoritmo evolutivo. Este comando determina métodos de avaliação, seleção, *crossover* e mutação, além de especificar o uso do elitismo. O comando `use` admite três sintaxes:

use <categoria> <método> [<lista de parâmetros>]

use <categoria> composite

use elitism

Sendo que categoria determina a categoria de ferramentas evolutivas configuradas pelo comando. Os quatro valores possíveis são **measure**, para métodos de medida; **selection**, para métodos de seleção; **crossover**, para métodos de *crossover*; **mutation**, para métodos de mutação; e **iterator** para iteradores da *ECCE*.

O comando **use** deve ser utilizado no máximo uma vez para cada categoria disponível. Não é necessário, por exemplo, especificar um método de *crossover*, mas no máximo um método de *crossover* pode ser especificado. É importante ter sempre em mente que um experimento da *ECCE* não pode ser executado sem um método de avaliação e um método de seleção, pelo menos.

O método especificado para cada categoria determina o algoritmo que será utilizado em cada etapa do algoritmo evolutivo. É importante que o método seja uma classe implementada pela *ECCE*. Por exemplo, para a categoria **selection**, o método **LinearRanking** pode ser escolhido, pois a classe **LinearRankingSelection** é implementada pela *ECCE*. É importante também respeitar letras maiúsculas e minúsculas, ou a *ECCE* pode não encontrar as classes que implementam os métodos desejados. O comando **use elitism** especifica que o algoritmo evolutivo deverá aplicar elitismo na seleção. O comando **use** pode precisar ser complementado por uma lista de parâmetros. Essa lista é explicada adiante.

8.3.5 O comando add

Quando o método escolhido para alguma categoria for de composição (*composite*), é necessário definir todos os métodos componente através do comando **add**. A sintaxe do comando **add** é a seguinte:

add <categoria> <método> [param <valor>] [<lista de parâmetros>]

Sendo que categoria, método e lista de parâmetros tem o mesmo significado que o comando **use**. A lista de parâmetros será explicada adiante, tanto para o comando **use** quanto para o comando **add**. Os parâmetros especificados no comando atuam sobre o método que será adicionado na composição. É interessante lembrar que o comando **add** funciona como configuração para a função membro **add** da classe que implementa o método composto de alguma categoria. Por isso, duas restrições devem ser sempre observadas.

Primeiramente, um comando **add** só pode ser especificado para uma categoria onde tenha sido selecionado um método de composição. Ou seja, é necessário um comando do tipo **use composite** antes de usar o comando **add**. Em segundo lugar, é possível que a adição de métodos à classe composta necessite parâmetros próprios. É o caso, por exemplo, da classe **MeasureAbstract**. Ao adicionar métodos de medida a essa classe, ela espera receber como parâmetro um valor real positivo, não nulo, que identifica o peso do método na composição. Esse valor é especificado pelo modificador **param**, entre colchetes.

Por exemplo, os seguintes comandos configuram a função de medida $f = \frac{2 \times \text{Error} + 3 \times \text{Precision}}{5}$.

use measure composite

add measure Error param [2]

add measure Precision param [3]

8.3.6 O comando stop

O comando `stop` pode ser utilizado para definir um ou mais critérios de parada. Se, a qualquer momento do experimento, algum critério `stop` for verificado, o experimento é finalizado, independente dos outros critérios. A sintaxe do comando `stop` é a seguinte:

stop [when] <atributo> <operador> <valor>

Sendo que `atributo` é uma expressão lógica envolvendo os seguintes critérios:

`iterations`: o número de iterações (ou gerações) passados desde o início do experimento.

`time`: o tempo de duração do experimento em segundos. O tempo é verificado imediatamente após o fim de cada iteração.

`convergence`: o desvio padrão do melhor indivíduo nas últimas iterações. Se o desvio padrão do melhor indivíduo nas últimas 20 gerações for menor que o valor informado, o algoritmo evolutivo convergiu.

`best`: o valor do melhor indivíduo para a função de avaliação.

`mean`: o valor da média para a função de avaliação.

A palavra `when` é utilizada apenas para fins estéticos e pode ser omitida.

O conjunto de operadores disponível é {>, <, >=, <=}. O operador de igualdade não está disponível, com o objetivo de diminuir a possibilidade de um critério de parada impedir o experimento de finalizar. Qualquer valor diferente de `operador` causará a parada do experimento.

O atributo `valor` deve ser um número real utilizando o ponto (.) como separador decimal.

8.3.7 Especificação de parâmetros

É importante observar que muitos métodos na *ECCE* exigem a especificação de parâmetros individuais. Por exemplo, o método de seleção **ExpontentialRanking** necessita de um parâmetro base cuja classe **ExpontentialRankingSelection** configura através da função membro `set_base`. Já os métodos de *crossover* necessitam do parâmetro `crossover_rate`, que é configurado na classe **CrossoverAbstract** através da função membro `set_crossover_rate`. Além disso, novos métodos, incrementados à *ECCE*, possivelmente, por outros desenvolvedores, podem necessitar de parâmetros bastante distintos.

A lista de parâmetros do comando `use` especifica esses valores. A lista utiliza a sintaxe descrita no exemplo a seguir.

```
with <parâmetro 1> <valor 1>
    <parâmetro 2> <valor 2> and
    ...
    <parâmetro N> <valor N>
```

Nesse ponto, é necessário que o responsável pelo experimento possua um conhecimento mínimo das classes que está utilizando. O nome do parâmetro é o nome da função membro a ser configurada, sem o prefixo `set_`. O valor do parâmetro é na verdade a lista de argumentos que a função recebe, entre colchetes. Exemplos de uso da especificação de parâmetros são:

use selection ExponentialRanking with base [0.7]: especifica o método de seleção *ranking* exponencial e configura o parâmetro *base* com o valor 0.7. Para realizar essa configuração, o responsável pelo experimento deve saber que o método de *ranking* exponencial é implementado na classe **ExponentialRankingSelection** e configurado pela função membro *set_base*, que recebe um único número real entre 0 e 1.

use selection LinearRanking with range [0.10]: especifica o método de seleção por *ranking* linear com intervalo de atuação [0, 10]. Para realizar essa configuração, o responsável pelo experimento deve saber que o método de *ranking* linear é implementado na classe **LinearRankingSelection** e configurado pela função membro *set_range*, que recebe como argumento dois valores inteiros, um mínimo e um máximo do intervalo de atuação.

use crossover LocalLevel with crossover_rate [0.8]: especifica o método de *crossover* local com taxa de *crossover* 0.8. Para realizar essa configuração, o responsável pelo experimento deve saber que o método de *crossover* local é implementado na classe **LocalLevelCrossover** e herda a função membro *set_crossover_rate* da classe **CrossoverAbstract**.

Note que todas as informações que devem ser de conhecimento do responsável pelo experimento podem ser encontradas neste documento, as quais encontram-se descritas detalhadamente na Seção 6 na página 31.

8.3.8 Inserção de Comentários

O arquivo de configuração permite a inclusão de comentários com o caractere especial #. Todo o restante da linha à direita deste caractere (a menos que ele apareça no nome de arquivo) será ignorado.

8.3.9 Exemplo de Arquivo de Configuração

A construção de um arquivo de configuração é bastante intuitiva. Para que os futuros experimentos baseados neste trabalho possam ser facilmente configurados, segue um exemplo de arquivo de configuração.

```
# Experimento #7
# Date   : 03/16/2006
# Author: Rafael Giusti / Adriano Donizete Pila

# Seleciona aleatoriamente indivíduos gerados pelo
# classificador e inclui regras de nossa autoria
# do arquivo 'personal.rules'
#

select 26 individuals without reposition from 'c4.5.rules'
select all from 'personal.rules'
```

```

# Função fitness:  $f(x) = (3 \cdot \text{Error} + 2 \cdot \text{Precision}) / 5$ 
#

use measure composite
add measure Error param [3]
add measure Precision param [5]


# Usa seleção por torneio e crossover do nível local
#

use selection Tournament with tournament_size [4]
use crossover LocalLevel with crossover_rate [0.5]


# Utiliza crossover composto com probabilidades idênticas
# para os níveis estrutural e de atributo e probabilidade
# um pouco maior para o crossover local
#

use crossover composite
add crossover StructuralLevel with crossover_rate [0.60]
add crossover FeatureLevel with crossover_rate [0.60]
add crossover LocalLevel with crossover_rate [0.80]


# Utiliza mutação composta com baixa probabilidade para
# o nível estrutural e uma probabilidade um pouco maior
# para o nível local
#

use mutation composite
add mutation StructuralLevel with mutation_rate [0.05]
add mutation LocalLevel with mutation_rate [0.2]


# Tenta parar somente quando o melhor indivíduo alcançar
# um fitness muito alto. Paralisa o experimento após uma
# hora, em caso não ocorrer convergência

```

#

```
stop when best >= 0.9  
stop when time >= 3600  
stop when convergence <= 0.005
```

9 Considerações Finais

Toda a biblioteca de classes *ECLE* foi implementada em PERL e esta disponível livremente sob a forma da licença da GNU/GPL (<http://www.gnu.org/copyleft/gpl.html>). Pode ser que seja necessária a instalação de alguns pacotes adicionais na versão do PERL que o usuário vá utilizar. Todos os pacotes foram instalados a partir da CPAN (<http://www.cpan.org>), mesmo aqueles adicionais. As funcionalidades de pré-processamento utilizadas são provenientes da DLE. A *ECLE* está disponível em <http://www.icmc.usp.br/~pila/ECLE.tar.gz>.

10 Conclusão

O objetivo deste trabalho é propiciar uma visão detalhada sobre a biblioteca de classes *ECLE*, a qual implementa algoritmos evolutivos destinados ao aprendizado de regras de conhecimento com propriedades específicas. Além dessa biblioteca, é também descrito o *SnifferECLE*, que é uma extensão do *Sniffer*, o qual implementa funcionalidades específicas para executar experimentos que envolvam algoritmos evolutivos e regras de conhecimento. O *SnifferECLE* permite configurar os experimentos de uma forma simples e intuitiva, utilizando um arquivo de configuração e uma classe que reconhece e interpreta a linguagem de descrição dos experimentos.

Embora as classes implementem uma vasta gama de métodos, sendo que esses métodos podem ser configurados com parâmetros próprios, a *ECLE* foi projetada para permitir a extensão de suas classes e métodos. Dessa forma, a implementação das classes utilizando sólidos conceitos de análise, projeto e programação orientados à objetos foi fundamental para o desenvolvimento da *ECLE*. Além de utilizar uma estrutura de dados apropriada para a representação e a evolução dos indivíduos, atualmente encontram-se definidos e implementados métodos de seleção, recombinação, mutação e avaliação de indivíduos utilizando medidas primitivas de avaliação de regras de conhecimento.

Experimentos estão sendo realizados para validar a implementação e a proposta do trabalho de pesquisa. Foram escolhidos diversos conjuntos de dados provenientes do repositório da UCI (Newman et al., 1998), para caracterizar vários cenários possíveis para a construção das regras. Os resultados obtidos serão publicados em artigos e num futuro relatório técnico.

Agradecimentos

Este trabalho foi desenvolvido com auxílio da FAPESP — processo nº 2005/01178-1 — destinado ao primeiro autor. O segundo autor agradece o suporte financeiro parcial da Anhanguera Educacional, Faculdade Comunitária de Santa Bárbara, Brasil – projeto FUNADESP nº 5500101.

Referências

- Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., e Verkamo, I. (1996). *Fast Discovery and Data Mining*, pag. 307–328. American Association for Artificial Intelligence, Menlo Park, CA. Citado na página 30.
- Alpaydin, E. (2004). *Introduction to Machine Learning*. MIT Press. Citado na página 2.
- Baranauskas, J. A. e Batista, G. E. A. P. A. (2000). O Projeto DISCOVER: Idéias Iniciais. Comunicação pessoal. Citado na página 28.
- Batista, G. E. A. P. A. (2001). Sintaxe padrão do arquivo de exemplos do projeto DISCOVER. <http://www.icmc.sc.usp.br/~gbatista/Discover/SintaxePadraoFinal.htm>. Citado na página 29.
- Batista, G. E. A. P. A. (2003a). Pré-processamento de Dados em Aprendizado de Máquina Supervisionado. Tese de Doutorado, ICMC-USP, <http://www.icmc.usp.br/~gbatista/pdfs/TeseDoutorado.pdf>. Citado na página 57.
- Batista, G. E. A. P. A. (2003b). Pré-processamento de Dados em Aprendizado de Máquina Supervisionado. Tese de Doutorado, ICMC-USP. <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06102003-160219/>. Citado na página 29.
- Batista, G. E. A. P. A. e Monard, M. C. (2003a). An Analysis of Four Missing Data Treatment Methods for Supervised Learning. *Applied Artificial Intelligence*, 17(5):519–533. <http://www.icmc.usp.br/~gbatista>. Citado na página 29.
- Batista, G. E. A. P. A. e Monard, M. C. (2003b). Descrição da Arquitetura e do Projeto do Ambiente Computacional DISCOVER LEARNING ENVIRONMENT — DLE. Technical Report 187, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_187.pdf. Citado nas páginas 29 e 30.
- Batista, G. E. A. P. A. e Monard, M. C. (2003c). Descrição da Arquitetura e do Projeto do Ambiente Computacional DISCOVER LEARNING ENVIRONMENT — DLE. Technical Report 187, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_187.pdf. Citado na página 57.
- Batista, G. E. A. P. A., Prati, R. C., e Monard, M. C. (2004). A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. *SIGKDD Explorations*, 6(1):20–29. Special issue on Learning from Imbalanced Datasets. <http://www.acm.org/sigs/sigkdd/explorations/>. Citado na página 29.
- Bäck, T., Fogel, D. B., e Michalewicz, T. (2000). *Evolutionary Computation 1 - Basic Algorithms and Operators*. Institute of Physics Publishing - IoP. Citado nas páginas 1, 67, 68 e 69.
- Bernardini, F. C. (2006). *Combinação de Classificadores Utilizando Medidas de Regras de Conhecimento e Algoritmos Genéticos*. PhD thesis, ICMC/USP, São Carlos, SP. Citado na página 10.
- Blickle, T. (2000). *Tournament selection*, chapter 24, pag. 181–186. Volume I of Bäck et al. (2000). Citado na página 26.
- Booch, G., Jacobson, I., Rumbaugh, J., e Rumbaugh, J. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley. Citado na página 31.
- Boswell, T. (1990). Manual for NewId version 4.1. Technical Report TI/P2154/RAB/4/2.3, The Turing Institute. Citado na página 30.
- Carvalho, A. C. P. L. F., Braga, A. P., e Ludermir, T. B. (2003). *Computação Evolutiva*, chapter 9, pag. 225–248. Volume 1 of Rezende (2003), 1 edition. Citado na página 27.

- Chiara, R. e Monard, M. C. (2003). Projeto e implementação de um filtro para transformar logs de servidores web em arquivos no formato padrão do sistema DISCOVER. Technical Report 183, ICMC-USP. Citado na página 30.
- Clark, P. e Boswell, R. (1991). Rule induction with CN2: Some recent improvements. In Kodratoff, Y., editor, *Proceedings of the 5th European Conference EWSL 91*, pag. 151–163. Springer-Verlag. Citado nas páginas 1 e 30.
- Coello, C. A. C. (2006). Evolutionary multi-objective optimization: A historical view of the field. *IEEE Comput. Int. Mag.*, pag. 28–36. Citado na página 10.
- Coello, C. A. C., Veldhuizen, D. A. V., e Lamont, G. B. (2002). *Evolutionary algorithms for solving multi-objective problems*. Kluwer Academic, New York, NY, USA. Citado nas páginas 10, 11 e 18.
- Darwin, C. R. (1859). *On the Origin of Species*. John Murray, London. Citado na página 1.
- Deb, K. (2000). *Introduction to selection*, chapter 22, pag. 166–171. Volume I of Bäck et al. (2000). Citado na página 25.
- Deb, K. (2001). *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, New York, NY, USA. Citado nas páginas 10, 11, 13, 15, 17 e 18.
- Dörre, J., Gerstl, P., e Seiffert, R. (1999). Text mining: Finding nuggets in mountains of textual data. In *Fifth International Conference on Knowledge Discovery and Data Mining (KDD-99)*, pag. 398–401. Citado na página 29.
- Fayyad, U., Piatetsky-Shapiro, G., e Smyth, P. (1996a). From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54. Citado na página 10.
- Fayyad, U. M., Piatetsky-Shapiro, G., e Smyth, P. (1996b). The kdd process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34. Citado na página 10.
- Flach, P. (2003). The geometry of ROC space: understanding machine learning metrics through ROC isometrics. In *Proc. 20th International Conference on Machine Learning (ICML'03)*, pag. 194–201. AAAI Press. Citado na página 18.
- Freitas, A. (2002). *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer-Verlag. Citado nas páginas 1, 23, 26 e 28.
- Freitas, A. A. (1998a). A multi-criteria approach for the evaluation of rule interestingness. In *Proc. of the International Conference on Data Mining*, pag. 7–20, Rio de Janeiro, RJ. Citado na página 9.
- Freitas, A. A. (1998b). On objective measures of rule surprisingness. In *Principles of Data Mining & Knowledge Discovery: Proc. of the Second European Symp. Lecture Notes in Artificial Intelligence*, volume 1510, pag. 1–9. Citado na página 9.
- Fuernkranz, J. e Flach, P. A. (2005). ROC 'n' rule learning – towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77. Citado na página 18.
- Gamma, E., Helm, R., Johnson, R., e Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. Citado na página 29.
- Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company. Citado na página 23.
- Gomes, A. K. (2002). Qualidade e interessabilidade de conhecimento adquirido utilizando aprendizado simbólico de máquina. Dissertação de Mestrado, ICMC-USP, <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-04072002-144610/>. Citado na página 10.

- Grefenstette, J. (2000a). *Proportional selection and sampling algorithms*, chapter 23, pag. 172–180. Volume I of Bäck et al. (2000). Citado na página 25.
- Grefenstette, J. (2000b). *Rank-based selection*, chapter 25, pag. 187–194. Volume I of Bäck et al. (2000). Citado na página 26.
- Haupt, R. L. e Haupt, S. E. (1998). *Practical Genetic Algorithms*. Wiley-interscience Publication. Citado nas páginas 24 e 26.
- Herrera, F., Lozano, M., Pérez, E., Sanchez, A. M., e Villar, P. (2002). Multiple crossover per couple with selection of the two best offspring: An experimental study with the blx-alpha crossover operator for real-coded genetic algorithms. In *IBERAMIA 2002: Proceedings of the 8th Ibero-American Conference on AI*, pag. 392–401, London, UK. Springer-Verlag. Citado na página 42.
- Horst, P. S. (1999). Avaliação do conhecimento adquirido por algoritmos de aprendizado de máquina utilizando exemplos. Dissertação de Mestrado, ICMC-USP. Citado na página 9.
- Kendall, M. G. (1990). *Rank correlation methods*. A Charles Griffin Book. Citado nas páginas 18 e 19.
- Lavrač, N., Flach, P., e Zupan, B. (1999). Rule evaluation measures: A unifying view. In Springer-Verlag, editor, *Lecture Notes in Artificial Intelligence*, pag. 174–185. Citado nas páginas 6 e 9.
- Michalewicz, Z. (1997). *Genetic Algorithms + Data Structures = Evolution Programs*. IE-Springer-Verlag. Citado nas páginas 1, 22 e 28.
- Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. The MIT Press. Citado na página 22.
- Mitchell, T. M. (1997). *Machine Learning*. WCB McGraw-Hill. Citado na página 29.
- Monard, M. C. e Baranauskas, J. A. (2003a). *Conceitos sobre Aprendizado de Máquina*, chapter 4, pag. 89–114. Volume 1 of Rezende (2003), 1 edition. Citado nas páginas 1 e 4.
- Monard, M. C. e Baranauskas, J. A. (2003b). *Indução de Regras e Árvores de Decisão*, chapter 5, pag. 115–140. Volume 1 of Rezende (2003), 1 edition. Citado na página 1.
- Newman, D., Hettich, S., Blake, C., e Merz, C. (1998). UCI repository of machine learning databases. <http://www.ics.uci.edu/~mllearn/MLRepository.html>. Citado nas páginas 2 e 66.
- Prati, R. C. (2003). *O Framework de Integração do Sistema DISCOVER*. Dissertação de Mestrado, ICMC-USP. <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-20082003-152116/>. Citado na página 29.
- Prati, R. C., Baranauskas, J. A., e Monard, M. C. (2001). Uma proposta de unificação da linguagem de representação de conceitos de algoritmos de aprendizado de máquina simbólicos. Technical Report 137, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_137.ps.zip. Citado nas páginas 29, 31 e 32.
- Prati, R. C., Batista, G. E. A. P. A., e Monard, M. C. (2004). Class imbalances versus class overlapping: an analysis of a learning system behavior. In *Mexican International Conference on Artificial Intelligence, LNAI 2972*, pag. 312–321. Springer-Verlag. Citado na página 29.
- Provost, F. e Fawcett, T. (2001). Robust classification for imprecise environments. *Machine Learning*, 42(3):203–231. Citado na página 18.
- Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106. Citado na página 30.
- Quinlan, J. R. (1987). Generating production rules from decision trees. Technical report, Italy. Citado na página 30.
- Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning*. Morgan Kaufmann Publishers, CA. Citado nas páginas 1 e 30.

- Rezende, S. O. (2003). *Sistemas Inteligentes: Fundamentos e Aplicações*. Editora Manole, Barueri, SP, Brasil. Citado nas páginas 67, 69 e 70.
- Rezende, S. O., Pugliesi, J. B., Melanda, E. A., e Paula, M. F. (2003). *Mineração de Dados*, chapter 12, pag. 307–336. Volume 1 of Rezende (2003), 1 edition. ISBN 85-204-1683-7. Citado na página 10.
- Russell, S. e Norvig, P. (2003). *Inteligência Artificial*. Campus Editora, São Paulo. Citado na página 4.
- Tsang, I. W., Kwok, J. T., e Cheung, P.-M. (2005). Core vector machines: Fast SVM training on very large data sets. *J. Mach. Learn. Res.*, 6:363–392. Citado na página 30.
- Wall, L., Christiansen, T., e Schwartz, R. L. (1996). *Programming Perl*. O'Reilly & Associates, 2 edition. Citado nas páginas 1, 29 e 52.
- Weiss, S. M. e Indurkha, N. (1998). *Predictive Data Mining: A Practical Guide*. Morgan Kaufmann, San Francisco, CA. Citado na página 29.