UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VAUX SANDINO DINIZ GOMES

UM CLASSIFICADOR BASEADO EM REGRAS DE DECISÃO E SUA IMPLEMENTAÇÃO NO WEKA

MOSSORÓ
2014

VAUX SANDINO DINIZ GOMES

UM CLASSIFICADOR BASEADO EM REGRAS DE DECISÃO E SUA IMPLEMENTAÇÃO NO WEKA

Monografia apresentada à Universidade Federal Rural do Semiárido – UFERSA, Departamento de Ciências Exatas e Naturais para a obtenção do título de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Sc. Helcio Wagner da Silva – UFERSA.

MOSSORÓ
2014

O conteúdo desta obra é de inteira responsabilidade de seus autores

Dados Internacionais de Catalogação na Publicação (CIP)

Biblioteca Central Orlando Teixeira (BCOT)

Setor de Informação e Referência

G633c Gomes, Vaux Sandino Diniz.

Um classificador baseado em regras de decisão e sua implementação no WEKA / Vaux Sandino Diniz Gomes. -- Mossoró, 2014.

65f.: il.

Orientador: Prof. Dr. Helcio Wagner da Silva.

Monografia (Graduação em Ciência da Computação) – Universidade Federal Rural do Semi-Árido. Pró-Reitoria de Graduação.

1. Minerador de dados. 2. Classificador. 3. Logical Analysis of data. 4. WEKA. I. Título.

RN/UFERSA/BCOT /752-14 CDD: 006.312

Bibliotecária: Vanessa Christiane Alves de Souza Borba

CRB-15/452

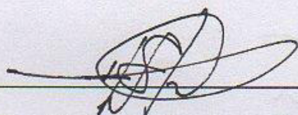
VAUX SANDINO DINIZ GOMES

UM CLASSIFICADOR BASEADO EM REGRAS DE DECISÃO E SUA IMPLEMENTAÇÃO NO WEKA

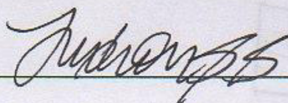
Monografia apresentada à Universidade Federal Rural do Semiárido – UFERSA, Departamento de Ciências Exatas e Naturais para a obtenção do título de Bacharel em Ciências da Computação.

APROVADA EM: 06 / 08 / 2014

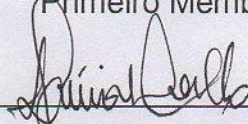
BANCA EXAMINADOR



Prof. Dr. Helcio Wagner da Silva - UFERSA
Orientador – Presidente



Prof. Dr. Judson Santos Santiago - UFERSA
Primeiro Membro



Prof. Msc. Flávia Estelia Silva Coelho - UFERSA
Segundo Membro

AGRADECIMENTOS

Agradeço à minha família e amigos próximos que sempre puxaram o meu saco e me que deram força e motivação, mesmo sem saber ao certo de que se trata o meu curso e muito menos este documento.

Agradeço aos meus grandes amigos da graduação, em especial Marisergio Angelo, Raimar de Raimundinho de Preto de Izauro, Rodrigo Kakashi e Danilo Ducelta, que compartilharam bons e maus momentos durante o tempo em que estudamos juntos.

Agradeço ao grande amigo Tibérius, que idealizou este projeto, sendo este documento um pequeno fruto da nossa colaboração. Agradeço ao revisor deste documento, Helcio pela força e dedicação que aplicou a este trabalho.

Também agradeço pelo incentivo por parte de amigos que vim a conhecer recentemente, Airton Ramiro Jr., Felipe Cego, Felipe Morto, Ademar Yesmadon, Maria Luzia, Ulysses Morgado e Paulo Rasgado. Obrigado a estes pelas gargalhadas e toda a amizade.

Por último e mais importante, agradeço a Deus cuja bondade tem feito grandes coisas em minha vida. A Ele e somente a Ele seja dada toda a glória nos céus e na terra. Agradeço porque o SENHOR tem me amado e quebrado o meu coração para que eu entendesse as coisas que parecem loucura aos olhos dos homens.

RESUMO

Aprendizado de máquina é uma área da ciência da computação que estuda maneiras pelas quais uma máquina pode encontrar ou reconhecer padrões em conjuntos de dados a fim de generalizar o conhecimento sobre os dados e poder tomar decisões sobre novas amostras de dados. Este trabalho se propõe a descrever como um dos mais utilizados pacotes de algoritmos de aprendizado de máquina, o WEKA, pode ser estendida para a inserção de novos algoritmos de classificação supervisionada; assim como propõe-se a descrever o algoritmo LAD (*Logical Analysis of Data*) e uma implementação particular deste algoritmo dentro do WEKA.

Palavras-chaves: Aprendizado de Máquina, Classificador, WEKA, *Logical Analysis of Data*.

ABSTRACT

Machine learning is a field within computer science that studies ways of how to program a machine to identify patterns within a labelled training set in order to infer a function that can predict the label of new instances. This document describes how to include a new classifier within WEKA, a free open source program that contains several machine learning algorithms. In addition to that, this document describes a version of LAD (Logical Analysis of Data) algorithm, a supervised learning classifier algorithm, and shows its implementation within WEKA.

Palavras-chaves: Machine Learning, Classifier, WEKA, Logical Analysis of Data.

SUMÁRIO

1 INTRODUÇÃO 10
1.1 OBJETIVOS 10
1.2 ORGANIZAÇÃO DO TRABALHO 11
2 ANÁLISE LÓGICA DE DADOS 12
2.1 PRINCIPAIS CONCEITOS 12
2.1.1 Outros conceitos e terminologias 13
2.2 BINARIZAÇÃO 13
2.2.1 Encontrando os pontos de corte 14
2.2.2 Binarizando os dados 16
2.3 SELEÇÃO DE ATRIBUTOS 17
2.3.1 O problema do recobrimento de conjuntos (PRC) 18
2.3.2 Heurística gulosa para o problema do recobrimento 18
2.3.3 Montando o PRC 20
2.3.4 Solucionando o PRC 22
2.4 IDENTIFICAÇÃO DE PADRÕES 22
2.4.1 Entendendo Regras 23
2.4.2 Geração de Regras 25
2.4.2.1 Parâmetros do algoritmo RRG 26
2.4.2.2 Algoritmo para geração de regras aleatórias 27
2.4.2.3 Outras considerações 29
2.4.2.4 Exemplo de geração de regras 30
2.4.3 Regras numéricas 30
2.4.4 Atribuindo pesos às regras 32
2.5 CLASSIFICAÇÃO 33
3 IMPLEMENTANDO UM CLASSIFICADOR NO WEKA 35
3.1 ESCRITOS SOBRE O WEKA 35
3.2 FERRAMENTAS 36
3.3 CRIANDO UM NOVO PROJETO NO ECLIPSE 36
3.4 IMPORTANDO O CÓDIGO-FONTE DO WEKA 36
3.5 CRIANDO UM CLASSIFICADOR SIMPLES 41
3.5.1 O tipo ARFF e as classes *Instances*, *Instance* e *Attribute* 41

3.5.3 Implementando *Serializable* 43
3.5.4 Treinamento 44
3.5.5 Classificando instâncias 44
3.5.6 Parâmetros da interface 45
3.5.7 Habilitando as capacidades do classificador 45
3.6 ARQUIVOS PROPS 47
3.7 EXPORTANDO O WEKA MODIFICADO 48
3.8 INSTALAÇÃO DO WEKA MODIFICADO 49
4 LAD-WEKA 51
4.1 AMBIENTE WEKA 51
4.1.1 Módulo *Explorer* 52
4.1.2 Selecionando um classificador 52
4.1.3 Acessando as configurações de um classificador 54
4.2 CONFIGURANDO O LAD 55
4.2.1 *Cutpoint Tolerance* 55
4.2.2 *Feature Selection* 56
4.2.3 *Minimum Purity* 56
4.2.4 *Rule Generator* 57
4.2.4.1 *Number of Rules* 57
4.2.4.2 *Number of Features* 58
4.2.4.3 *Minimum Relative Coverage of Own Class* 58
4.2.4.4 *Random Seed* 58
4.2.5 *Print File* 58
4.3 SAÍDA DO TREINAMENTO 59
4.4 TESTES DE COMPARAÇÃO 60
5 CONCLUSÕES 62
REFERÊNCIAS 63

1 INTRODUÇÃO

Aprendizado de máquina é uma área da ciência da computação que estuda maneiras pelas quais uma máquina pode encontrar ou reconhecer padrões em conjuntos de dados. Dentre as formas do aprendizado de máquina existe a classificação supervisionada. O objetivo principal é extrair informações de dados previamente separados em grupos, e, a partir destas informações, poder identificar a qual grupo uma nova amostra de dados pertence.

Algumas aplicações disponibilizam algoritmos de aprendizado de máquina. Estas ferramentas, em geral, são gerenciadas por grupos experientes de pesquisadores e programadores. Ainda, estas aplicações são bastante usadas por outros pesquisadores que desejam aplicar os algoritmos de aprendizado em problemas de cunho reais ou produzidos artificialmente para fins de estudos. Software que disponibilizam algoritmo de aprendizado de máquina também são desejáveis quando não se quer estudar e implementar todos os algoritmos necessitam em uma pesquisa. O pacote WEKA (*Waikato Environment for Knowledge Analysis*), disponibiliza vários algoritmos de aprendizado de máquina e ferramentas que possibilitam testes e comparações entre algoritmos. O WEKA caracteriza-se por possuir código aberto e licença GNU. Isto permite que qualquer um insira e/ou modifique os algoritmos da aplicação e distribua-os livremente.

O LAD (*Logical Analysis of Data*) é um algoritmo de classificação supervisionada que ainda não possui uma implementação dentro do pacote oficial de algoritmos do WEKA. Este algoritmo, que surgiu no final da década de 80 (Boros et al., 1997), combina elementos de funções booleanas, otimização e análise combinatória. O algoritmo já foi aplicado com sucesso a problemas nas áreas da medicina (Alexe et al., 2006), da economia (Hammer et al., 2006) e *business* (Hammer et al., 1999), etc. Este algoritmo possui poucas implementações públicas. Assim, a distribuição pública deste algoritmo pode ajudar a pesquisadores interessados na utilização e/ou no estudo do algoritmo. Em especial, a distribuição deste algoritmo dentro do WEKA possui vantagem sobre distribuições *standalone* do algoritmo, uma vez que se herda todas as demais vantagens e praticidades que o WEKA disponibiliza.

1.1 OBJETIVOS

Este trabalho propõe-se a descrever como se inserir um algoritmo de classificação dentro do WEKA a partir da construção de um classificador supervisionado simples e como exportar uma versão modificada do WEKA, e também propõe-se a descrever uma versão do algoritmo LAD com uma implementação particular da fase de identificação de padrões. Por último, como a união dos objetivos citados o documento mostra uma versão do WEKA contendo o algoritmo descrito.

1.2 ORGANIZAÇÃO DO TRABALHO

O Capítulo 2 descreve uma versão do LAD e mostra os principais conceitos do algoritmo. O Capítulo 3 descreve como se pode construir um classificador dentro do WEKA e, posteriormente, exportar uma versão modificada do WEKA. O Capítulo 4 exibe uma introdução ao uso do LAD dentro do WEKA e realiza um conjunto de testes comparativos com alguns algoritmos presentes na ferramenta. Por último, o Capítulo 5 traz as considerações finais sobre o que foi apresentado neste documento.

2 ANÁLISE LÓGICA DE DADOS

O algoritmo LAD (*Logical Analysis of Data*), ou em português, ALD (Análise Lógica de Dados), é um algoritmo de classificação supervisionada, baseado em regras. Tipicamente, algoritmos de classificação supervisionada utilizam amostras de dados para encontrar padrões e usá-los para classificar novas amostras de dados (Witten e Frank, 2005). O algoritmo surgiu em 1986, publicado por Peter L. Hammer, da Rutgers University, e foi, mais tarde, estudado e estendido por Hammer e outros co-autores (incluindo Alex Kogan, Endre Boros, Gabriela Alexe, Sorin Alexe, Tibérius Bonates e Toshihide Ibaraki).

O algoritmo LAD é composto de algumas etapas, a saber: binarização, seleção de atributos e identificação de padrões. Resumidamente, a fase de binarização recebe amostras de dados (ou, simplesmente, dados) e realiza um mapeamento do domínio original dos dados para um formato binário. A fase de seleção de atributos encarrega-se de simplificar o mapeamento da fase de binarização, tentando reduzir o número de atributos binários gerados. A fase de identificação de padrões é a responsável por procurar padrões nos dados e gerar regras de classificação.

Este capítulo tem o intuito de explicar o funcionamento das etapas do algoritmo LAD. A Seção 2.1 contém conceitos básicos para o entendimento do algoritmo, juntamente com os modelos matemáticos que serão utilizados no decorrer do capítulo. As demais seções contêm o detalhamento das fases do algoritmo LAD.

2.1 PRINCIPAIS CONCEITOS

Seja $D = \{(x^1, y^1), (x^2, y^2), \ldots, (x^d, y^d)\}$ um conjunto de dados rotulados, onde $x^i \in \mathbb{R}^n$ é um vetor que corresponde a uma instância dos dados, e $y^i \in \{-1, +1\}$ corresponde à classe (ou rótulo) à qual x^i pertence. Cada uma das n posições de x^i são chamadas de atributos. As coordenadas dos atributos são denotados por $X_1, X_2, \ldots, X_n$, onde X_j representa o atributo localizado na j-ésima posição do vetor x^i. Os valores dos atributos são denotados por $x_1, x_2, \ldots, x_n$, onde x_j é o valor do atributo X_j de uma instância x^i.

Também seja D^+ o subconjunto de todas as instâncias de D que possuem o valor de $y = +1$; e seja D^- o subconjunto de todas as instâncias de D que possuem $y = -1$. Diz-se que D^+ é o conjunto das instâncias positivas e D^- é o conjunto das instâncias negativas.

Análogo a D define-se $B = \{(x^1, y^1), (x^2, y^2), \ldots, (x^d, y^d)\}$ um conjunto de dados rotulados, onde $x^i \in \{0, 1\}^m$ é um vetor binário que corresponde a uma instância binária, e $y^i \in \{-1, +1\}$ corresponde à classe a qual x^i pertence. Também se define B^+ e B^- nos mesmos moldes de D^+ e D^-. Nas seções seguintes, será mostrado que B é obtido a partir de D pelo processo de binarização.

2.1.1 Outros conceitos e terminologias

D e B podem ser facilmente representado por uma matriz, onde cada par (x^i, y^i) representa uma linha da matriz. A matriz é composta por $n+1$ colunas, isto é, n colunas de atributos (derivados do vetor x) e uma coluna de classe (representando y).

Os atributos de um conjunto de dados podem ser de vários tipos como real, binário, nominal (por exemplo, A = {prédio, casa, tenda}), etc. Ainda, é possível que um ou mais valores dos atributos, para uma ou mais instâncias, estejam faltantes, ou seja, não possuam valor. Isso pode acontecer, por exemplo, quando houver erro na coleta de uma dada instância. Muitos classificadores procuram não descartar uma instância de valores faltantes, uma vez que estes ainda podem usar as demais informações contidas nos outros atributos da instância. O classificador LAD pode ser definido de forma a lidar com valores faltantes, mas, por questão de simplicidade, tais detalhes não serão discutidos neste trabalho.

A classe representa o rótulo de cada instância dos dados. Daí o termo "dados rotulados", bastante encontrado em artigos sobre aprendizado de máquina. A classe tem a função de informar a qual grupo de dados a instância pertence. Por exemplo, para um conjunto de dados de câncer de mama, cada nódulo mamário observado é uma instância, seus atributos podem ser o tamanho do nódulo, a consistência, etc. As classes ou grupos possíveis para este conjunto de dados podem ser "sim"ou "não", isto é, se o nódulo é ou não cancerígeno.

Os dados servem tanto para o aprendizado quanto para testar o classificador. Os dados usados para o aprendizado são chamados de dados de treinamento ou conjunto de treinamento. Enquanto os dados usados para testar o classificador são chamados de dados de teste ou conjunto de teste.

2.2 BINARIZAÇÃO

Internamente, o algoritmo LAD trabalha com dados binários (0 ou 1). A binarização é uma fase de pré-processamento dos dados. Esta técnica, essencialmente, mapeia atributos numéricos para atributos binários e faz com que apenas as variações mais significantes dos dados sejam levadas em consideração, além de facilitar o processamento interno das demais etapas do algoritmo.

Esta preparação dos dados é uma fase importante do algoritmo LAD, pois dados numéricos costumam estar sujeitos a imprecisões. Esta fragilidade dos dados pode conduzir o algoritmo a capturar minúcias ou detalhes que podem ter origem em imprecisões geradas na coleta dos dados. Ao se discretizar os dados desta forma, retém-se apenas as características mais proeminentes dos dados (Boros et al., 2000).

Para o entendimento desta seção e das demais, um conjunto de dados padrão será usado na demonstração dos resultados das aplicações das etapas do algoritmo LAD. A Figura 2.1 exibe o conjunto de dados de duas classes composto por seis instâncias, onde cada instância contém três atributos numéricos (atributos **A**, **B** e **C**). Não existem atributos com valores faltantes neste

conjunto de dados.

Figura 2.1: Conjunto de dados padrão (original).

A	B	C	Classe
3,5	3,8	2,8	Sim
2,6	1,6	5,2	Sim
2,3	2,1	1,0	Não
1,0	2,1	3,8	Sim
3,5	1,6	3,8	Não
4,2	1,8	2,0	Não

Fonte: autoria própria.

Na Figura 2.1, a classe denominada por *Sim* receberá valor +1 e a classe denominada por *Não* receberá valor -1. Estas classes serão chamadas, respectivamente, de classe positiva e classe negativa. Esta substituição de valores poderia se dar de diferentes maneiras, contanto que uma classe receba um valor diferente da outra. Também, as instâncias foram separadas, propositalmente, em grupos de acordo com seus valores classes, e numeradas para auxílio nos exemplos futuros. O resultado pode ser visto na Figura 2.2.

Figura 2.2: Conjunto de dados padrão (modificado).

<table>
<tr><th>Nº</th><th>A</th><th>B</th><th>C</th><th>Classe</th><th></th></tr>
<tr><td>1</td><td>3,5</td><td>3,8</td><td>2,8</td><td>1</td><td rowspan="3">D^+</td></tr>
<tr><td>2</td><td>2,6</td><td>1,6</td><td>5,2</td><td>1</td></tr>
<tr><td>3</td><td>1,0</td><td>2,1</td><td>3,8</td><td>1</td></tr>
<tr><td>4</td><td>2,3</td><td>2,1</td><td>1,0</td><td>-1</td><td rowspan="3">D^-</td></tr>
<tr><td>5</td><td>3,5</td><td>1,6</td><td>3,8</td><td>-1</td></tr>
<tr><td>6</td><td>4,2</td><td>1,8</td><td>2,0</td><td>-1</td></tr>
</table>

Fonte: autoria própria.

2.2.1 Encontrando os pontos de corte

O processo de binarização apóia-se no conceito de **pontos de corte**. Os pontos de corte são usados para converter os valores numéricos dos atributos em valores binários, ou, melhor dizendo, os pontos de corte transformam os atributos numéricos em um ou mais atributos binários.

Cada ponto de corte relaciona um atributo X_i a um valor v_k que consegue dividir parcialmente instâncias de classes diferentes. Isto é, observando a característica X_i de todas as instâncias de um conjunto de dados D, o valor v_k separa ao menos duas instâncias de classes diferentes (no nosso caso, de classes positiva e negativa). Ou seja, se v_k é um ponto de

corte para o atributo X_i, então existe pelo menos um par de observações $x^a \in D^+$ e $x^b \in D^-$, satisfazendo exatamente um dos seguintes casos:

i. o valor do atributo X_i em x^a é **menor** que v_k e o valor do atributo X_i em x^b é **maior** que v_k; ou

ii. o valor do atributo X_i em x^a é **maior** que v_k e o valor do atributo X_i em x^b é **menor** que v_k.

Logo, o valor v_k pode ser utilizado para distinguir entre as instâncias x^a e x^b.

O processo para encontrar os pontos de corte inicia-se por ordenar os vetores de valores de cada um dos atributos, separadamente. Para cada valor de cada vetor de valores de atributos pode-se determinar se aquele valor acontece em instância(s) de classe positiva e/ou de classe negativa. No exemplo da Figura 2.2, para o atributo **A** tem-se o seguinte vetor ordenado:

$$A_{ord} = \{1,0^{(+)}; 2,3^{(-)}; 2,6^{(+)}; 3,5^{(+,-)}; 4,2^{(-)}\}. \tag{2.1}$$

Os valores 1,0 e 2,6 do vetor ordenado pertencem a instâncias de classe positiva. O valor 2,3 pertence a uma instância de classe negativa. O valor 3,5 pode ser encontrado em instâncias de ambas as classes e, finalmente, o valor 4,2 pertence a uma instância de classe negativa.

Os pontos de corte são encontrados nos pontos de **transição de classe**, isto é, para dois valores consecutivos A_a e A_b no vetor ordenado A_{ord}, o ponto de corte existe se A_a acontece em uma instância de uma classe diferente de A_b. No exemplo do vetor ordenado 2.1 existem quatro pontos de transição de classe. Um ponto de transição entre $1,0^{(+)}$ e $2,3^{(-)}$. Outro ponto entre $2,3^{(-)}$ e $2,6^{(+)}$. Outro entre $2,6^{(+)}$ e $3,5^{(-)}$ (uma vez que o valor 3,5 está presente nas duas classes). Também existe um ponto de transição entre $3,5^{(+)}$ e $4,2^{(-)}$.

Por definição, o ponto de corte poderia ser qualquer valor entre os valores consecutivos A_a e A_b, pois, para estes valores, v_k seria maior que A_a e menor que A_b. Contudo, para este processo de binarização definiremos o valor do ponto de corte em:

$$v_k = \frac{A_a + A_b}{2}. \tag{2.2}$$

Aplicando-se o método para o vetor A_{ord}:

$$v_1 = \frac{(1,0+2,3)}{2} = 1.65 \tag{2.3a}$$

$$v_2 = \frac{(2,3+2,6)}{2} = 2.45 \tag{2.3b}$$

$$v_3 = \frac{(2,6+3,5)}{2} = 3.05 \tag{2.3c}$$

$$v_4 = \frac{(3,5+4,2)}{2} = 3.85. \tag{2.3d}$$

O processo deve ser repetido para cada um dos atributos. A lista de pontos de cortes gerados para os atributos **A**, **B** e **C** é mostrada na Figura 2.3.

Figura 2.3: Pontos de corte dos atributos **A**, **B** e **C**.

A		B		C	
v_1	(1,0 + 2,3)/2 = 1,65	v_5	(1,6 + 1,8)/2 = 1,70	v_8	(2,0 + 2,8)/2 = 2,40
v_2	(2,3 + 2,6)/2 = 2,45	v_6	(1,8 + 2,1)/2 = 1,95	v_9	(2,8 + 3,8)/2 = 3,30
v_3	(2,6 + 3,5)/2 = 3,05	v_7	(2,1 + 3,8)/2 = 2,95	v_{10}	(3,8 + 5,2)/2 = 4,50
v_4	(3,5 + 4,2)/2 = 3,85				

Fonte: autoria própria.

Vale observar que no vetor ordenado do atributo **C** o primeiro elemento é o $1,0^{(-)}$ seguido de $2,0^{(-)}$. Estes valores não formam uma transição de classe, portanto não geram ponto de corte.

2.2.2 Binarizando os dados

A binarização de um único valor numérico consiste na comparação deste valor com o valor de um ou mais pontos de corte. Em outras palavras, um valor numérico x_i deve ser comparado a todos os pontos de corte v_k encontrados no vetor ordenado de valores de um atributo X_i. Cada par formado por um valor numérico x_i e um ponto de corte v_k gera um valor binário $b(x_i, v_k)$ tal que:

$$b(v_k, x_i) = \begin{cases} 1, \text{ se } x_i \leq v_k \\ 0, \text{ se } x_i > v_k. \end{cases} \tag{2.4}$$

A Figura 2.4 mostra a binarização da instância x^1 (de acordo com a numeração da Figura 2.2). A relação entre o valor numérico de cada atributo com os pontos de corte encontrados para o conjunto de dados, assim como os respectivos valores binários derivados da relação, são exibidos na figura.

Por fim, a Figura 2.5 exibe o conjunto de dados binarizado.

Não se devem confundir os conceitos de ponto de corte e atributo binário, embora estes estejam bastante ligados. O ponto de corte é o valor de um ponto de transição de classes, enquanto um atributo binário é o valor que indica se o valor de um atributo numérico está antes ou depois da transição de classe, isto é, se o valor do atributo numérico é maior ou menor/igual ao valor de um ponto de corte. Por sua vez, denotaremos as posições dos atributos binários por $X_1, X_2, \ldots, X_m$.

Figura 2.4: Mapeamento da instância 1 da Figura 2.2.

Atributo Numéricos	A				B		C		
Valor numérico	3,5				3,8		2,8		
Pontos de Corte	1,65	2,45	3,05	3,85	1,85	2,95	2,40	3,30	4,50
Relação	>	>	>	≤	>	>	>	≤	≤
Valor Binário	0	0	0	1	0	0	0	1	1
Atributo Binário	b_{A1}	b_{A2}	b_{A3}	b_{A4}	b_{B1}	b_{B1}	b_{C1}	b_{C2}	b_{C3}

Fonte: autoria própria.

Figura 2.5: Conjunto de dados binarizado.

Nº	b_{A1}	b_{A2}	b_{A3}	b_{A4}	b_{B1}	b_{B2}	b_{C1}	b_{C2}	b_{C3}	
1	0	0	0	1	0	0	0	1	1	B^+
2	0	0	1	1	1	1	1	1	0	
3	1	1	1	1	0	1	0	0	1	
4	0	1	1	1	0	1	1	1	1	B^-
5	0	0	1	1	1	1	0	0	1	
6	0	0	0	0	1	1	1	1	1	

Fonte: autoria própria.

2.3 SELEÇÃO DE ATRIBUTOS

Alguns pontos de cortes gerados na binarização podem ser entendidos como irrelevantes ou até mesmo redundantes. Por definição, um ponto de corte já diferencia ao menos um par de instâncias de classes diferentes, visto que o ponto de corte é gerado em um ponto de transição de classe. Neste contexto, dizer que um ponto de corte é irrelevante ou redundante significa dizer que se este ponto de corte for desconsiderado (removido da lista de pontos de corte) ainda será possível diferenciar todos os pares x^a e x^b de instâncias de classes diferentes.

Na fase de seleção de atributos, o algoritmo LAD procura encontrar um **conjunto reduzido de pontos de corte** que permita diferenciar todos os pares de instâncias. Isto é, para cada par de instâncias x^a e x^b de classes diferentes, deve-se selecionar **pelo menos um ponto de corte** que permita distinguir entre x^a e x^b. Para que fique claro, um ponto de corte distingue duas instâncias binárias se para uma das instâncias o valor binário, gerado a partir daquele ponto de corte, for igual a zero, enquanto o valor binário da outra instância, gerado pelo mesmo ponto de corte, for igual a um.

Ter um conjunto reduzido de pontos de corte significa simplificar/reduzir o esforço compu-

tacional na fase de identificação de padrões. No entanto, é necessário que esta redução ainda mantenha um conjunto de pontos de corte que permita montar regras capazes de distinguir instâncias de classes diferentes. Portanto, busca-se simplificar sem perder informação essencial. A tarefa de encontrar um conjunto reduzido de pontos de corte é uma aplicação direta do Problema Não-ponderado do Problema de Recobrimento de Conjuntos (PNPRC).

As próximas subseções mostram o Problema de Recobrimento de Conjuntos (Seção 2.3.1), juntamente com heurística para a solução do problema (Seção 2.3.2). A Seção 2.3.3 mostra como o algoritmo LAD monta o problema do recobrimento a partir dos dados binarizados. Por fim, a Seção 2.3.4 mostra a solução para o problema montado.

2.3.1 O problema do recobrimento de conjuntos (PRC)

O Problema de Recobrimento de Conjuntos (PRC), ou simplesmente o Problema de Recobrimento, pode ser descrito da seguinte forma. Seja $E = \{e_1, e_2, \ldots, e_n\}$ um conjunto de elementos, $S = \{S_1, S_2, \ldots, S_m\}$ uma família de subconjuntos de E satisfazendo

$$E = \bigcup_{S_i \in S} S_i, \tag{2.5}$$

e $c : S \to \mathbb{R}^+$ uma função de custo definida sobre S.

O Problema de Recobrimento consiste em selecionar $F \subseteq \{S_1, S_2, \ldots, S_m\}$, de custo total mínimo

$$c_{\text{total}} = \sum_{S_i \in F} c(S_i), \tag{2.6}$$

satisfazendo:

$$E = \bigcup_{S_i \in F} S_i. \tag{2.7}$$

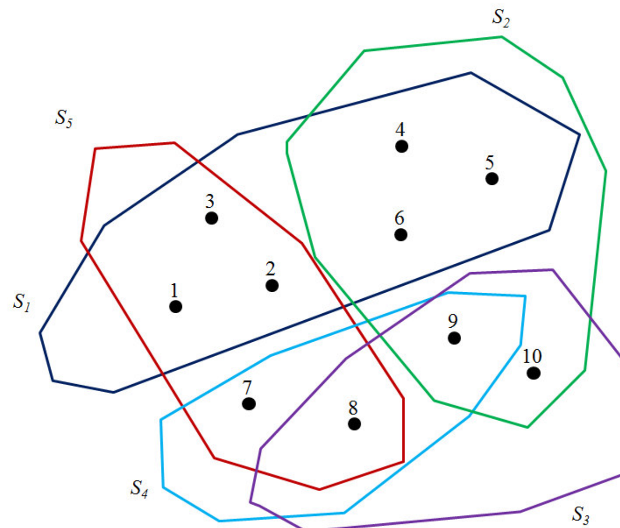
Na Figura 2.6, um exemplo (em representação geométrica) do problema é exibido (os pontos pretos representam os elementos de E).

Este problema possui inúmeras aplicações, incluindo cenários de planejamento de operações, tais como a localização ótima de centros de distribuição (denotado por S_i), cada um atendendo a certo conjunto de localidades (denotadas por e_i). O problema da Figura 2.6 será resolvido por meio do algoritmo descrito na próxima seção.

2.3.2 Heurística gulosa para o problema do recobrimento

A Figura 2.7 mostra um pseudocódigo do algoritmo guloso clássico para a solução do problema de recobrimento. O algoritmo é uma estratégia gulosa descrita por V. Chvátal em (Chvátal, 1979). É válido lembrar que, por ser uma heurística, o algoritmo de Chvátal não garante a solução ótima para o problema. Contudo, em tarefas de aprendizado de máquina, nem sempre o que se busca é o ótimo porque existe um risco de que o processo de aprendizagem

Figura 2.6: Exemplo do PRC.



Fonte: autoria própria.

memorize várias peculiaridades do conjunto de treinamento (fenômeno conhecido como superaprendizado) (Deitterich, 1995). Assim sendo, mesmo que a solução produzida pelo algoritmo de Chvátal não seja ótima, seu uso é recomendado, haja vista que tal solução estaria associada a um conjunto reduzido de pontos de corte.

O algoritmo da Figura 2.7 inicia com um conjunto $F = \emptyset$ e prossegue adicionando a F o subconjunto de S que é mais **atrativo** em cada iteração até que F componha uma solução completa para o problema.

A atratividade de um subconjunto S_i é dada por f, uma função que calcula a relação custo-benefício de um subconjunto. A função calcula a razão entre o custo associado a S_i e o número de elementos de E pertencentes a S_i que não pertencem a nenhum dos subconjuntos em F. Isto significa que estamos medindo a quantidade de custo adicional para cada novo item do conjunto E que é coberto por ocasião da inserção de S_i à solução atual. A função f pode ser denotada por:

$$f = \frac{c(S_i)}{\left| S_i \setminus (\bigcup_{S_j \in F} S_j) \right|}. \tag{2.8}$$

A cada iteração do algoritmo, os valores de custo-benefício são recalculados, tendo em vista que alguns dos elementos de E podem ter sido cobertos (passaram a pertencer a algum dos subconjuntos de F) na iteração passada. Para a versão Não-Ponderada do Problema do Recobrimento de Conjuntos (PNPRC), o custo $c(S_i) = 1, \forall S_i \in S$. Isto significa que o subconjunto S_i mais atrativo ou de melhor custo benefício será aquele que possuir menos elementos cobertos.

Para que a execução do algoritmo fique mais clara, o exemplo da Figura 2.6 será resolvido com o algoritmo da Figura 2.7. O algoritmo inicia adicionando S_1 a F (porque S_1 é o

Figura 2.7: Pseudocódigo do algoritmo guloso para o problema de recobrimento.

ALGORITMO GULOSO PARA O PROBLEMA DE RECOBRIMENTO

ENTRADA: Conjunto $E = \{e_1, e_2, \ldots, e_n\}$,
família $\mathscr{S} = \{S_1, S_2, \ldots, S_m\}$.
SAÍDA: Família F de custo mínimo.

1. $F = \emptyset$
2. **while** $(\bigcup_{S_i \in F} S_i \neq E)$
3. Seja S_j o subconjunto com menor "relação custo-benefício": $f(S_j) \leq f(S_i), \forall S_i \in S$
4. $S := S \setminus \{S_j\}$
5. $F := F \cup \{S_j\}$
6. **end**

Fonte: autoria própria.

subconjunto que contém o maior número de elementos de E que não pertencem a algum dos subconjuntos de F). Em seguida, o algoritmo pode adicionar S_3 ou S_4, pois ambos têm o mesmo custo-benefício, isto é, três elementos que ainda não pertencem a nenhum dos subconjuntos de F. O próximo conjunto a ser adicionado será o subconjunto S_5 ou S_2, dependendo de qual subconjunto foi adicionado a F na iteração anterior. Por fim, $F = \{S_1, S_3, S_5\}$ é suficiente para cobrir todos os elementos de E.

2.3.3 Montando o PRC

No algoritmo LAD, o problema de recobrimento é montado a partir do conjunto de dados binarizados. Cada subconjunto de S representa um atributo binário X_i. O que significa que o número de subconjuntos de S é igual ao número de atributos binários do conjunto de dados. Cada um dos elementos de E são representados por um par de instâncias de classes diferentes.

O algoritmo de montagem do PRC verifica os valores dos atributos de cada par de instâncias de classes diferentes para determinar a quais conjuntos de S o elemento de E, representado pelo par de instâncias, faz parte. Em outras palavras, para um par de instâncias x^a e x^b de classes diferentes, se o valor de um atributo X_i segue um dos seguintes casos:

i. o valor do atributo X_i em x^a é 1 e o valor do atributo X_i em x^b é 0; ou
ii. o valor do atributo X_i em x^a é 0 e o valor do atributo X_i em x^b a 1

diz-se que o elemento representado pelo par de instâncias x_a e x_b **pertence ao conjunto** S_i.

O algoritmo de montagem do PRC pode ser visto na Figura 2.8. Perceba que as condições (i) e (ii) são representadas pela condição da **linha 9** do algoritmo.

O PRC montado pode ser representado por uma matriz, como mostra a Figura 2.9. Cada coluna representa um dos conjuntos de S que, por sua vez, representam os atributos binários.

Figura 2.8: Pseudocódigo do algoritmo para montagem do PRC.

ALGORITMO PARA MONTAGEM DO PRC

ENTRADA: Conjunto de dados binários B.
SAÍDA: Instância S do PRC.

1. Seja B^+ e B^-, respectivamente, o conjunto de instâncias binárias positivas e negativas de B.
2. Seja m o número de atributos binários de B.
3. Seja $S := \{S_0, \ldots, S_m\}$ a família de conjuntos dos elementos de B
4. **foreach** $(p \in B^+)$
5.   **foreach** $(n \in B^-)$
6.   Seja e_j um novo elemento.
8.   **for** $(i := 1$ **to** $m)$
9.   **if** $(p_i \neq n_i)$
10. $S_i := S_i \cup \{e_j\}$
11.   **end**
12.   **end**
13.   **end**
14. **end**

Fonte: autoria própria.

Cada linha da matriz indica a quais conjuntos cada um dos elementos de E pertencem. Se o cruzamento de uma linha (elemento) com uma coluna (conjunto) contém o valor zero diz-se que o elemento em questão não pertence ao conjunto relacionado, e diz-se que o elemento pertence ao conjunto se o valor for um. No exemplo da Figura 2.9, o elemento e_1 representa o par de instâncias x^1 e x^4 da Figura 2.5. Este elemento pertence aos conjuntos S_2, S_3, S_6 e S_7.

Figura 2.9: PRC montado a partir do conjunto binarizado da Figura 2.5.

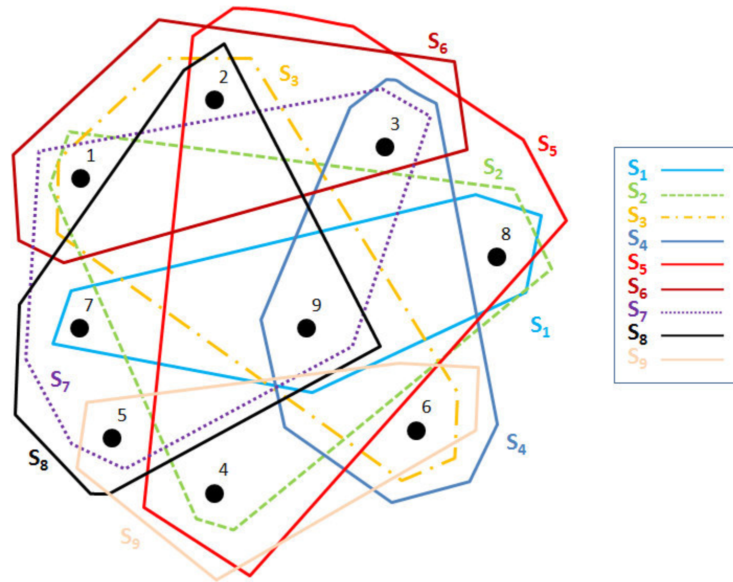
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_8
e_1	0	1	1	0	0	1	1	0	0
e_2	0	0	1	0	1	1	0	1	0
e_3	0	0	0	1	1	1	1	0	0
e_4	0	1	0	0	1	0	0	0	1
e_5	0	0	0	0	0	0	1	1	1
e_6	0	0	1	1	0	0	0	0	1
e_7	1	0	0	0	0	0	1	1	0
e_8	1	1	0	0	1	0	0	0	0
e_9	1	1	1	1	1	0	1	1	0

Fonte: autoria própria.

2.3.4 Solucionando o PRC

Uma visualização geométrica do problema montado na Figura 2.9 pode ser vista na Figura 2.10. Os elementos são os pontos pretos, e os conjuntos possuem cores diferentes e seus nomes estão exibidos na imagem.

Figura 2.10: PRC dos dados binarizados (representação geométrica da Figura 2.9).



Fonte: autoria própria.

A aplicação do algoritmo de Chvátal para o problema da Figura 2.9 (observando que o algoritmo implementado seleciona o subconjunto com o maior número de elementos não cobertos e que a busca é feita na ordem crescente dos índices dos subconjuntos) retorna:

$$F = \{S_5, S_7, S_3\}. \tag{2.9}$$

Os subconjunto contidos em F representam, respectivamente, os atributos binários b_{B1}, b_{C1} e b_{A3}. Os três atributos são suficientes para diferenciar todas as instâncias do conjunto de dados, como se pode observar na Figura 2.11. Perceba que o conjunto de dados foi bastante simplificado.

2.4 IDENTIFICAÇÃO DE PADRÕES

Os padrões são tendências encontradas nos dados de treinamento. Encontrar estas tendências e poder expressá-las em alguma estrutura é a missão desta fase. O LAD é um algoritmo que guarda padrões em formato de regras de decisão. Assim, torna-se interessante que entendamos sobre regras.

Esta seção foi dividida em duas subseções, uma sobre conceitos relacionados a regras (Seção 2.4.1), e outra sobre o algoritmo de geração de regras (Seção 2.4.2).

Figura 2.11: Seleção dos conjunto S_3, S_5 e S_7 da Figura 2.5.

Nº	b_{A3}	b_{B1}	b_{C1}	
1	0	0	0	
2	1	1	1	B^+
3	1	0	0	
4	1	0	1	
5	1	1	0	B^-
6	0	1	1	

Fonte: autoria própria.

2.4.1 Entendendo regras

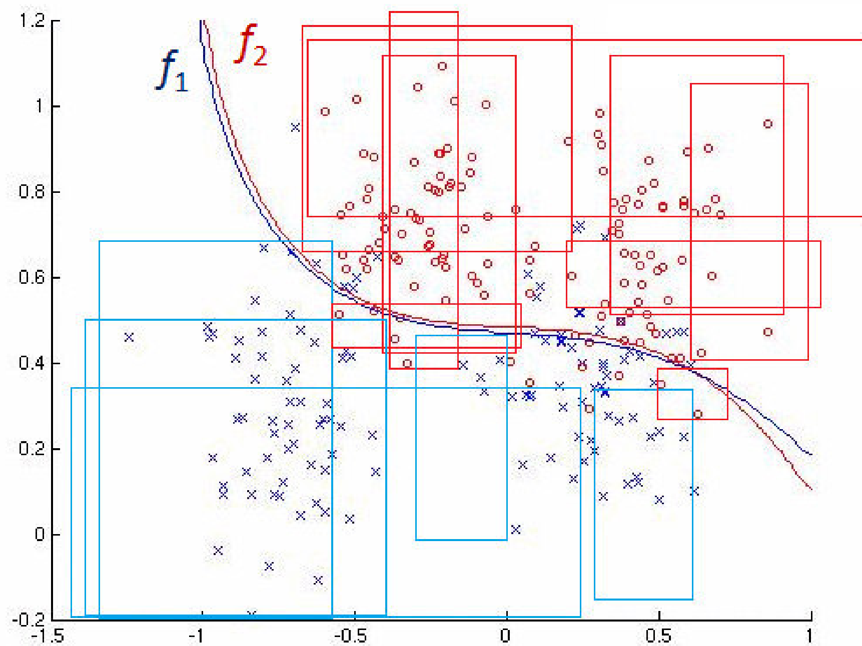
A tarefa de um classificador é aprender (inferir) uma função que possa separar os dados por suas classes (Deitterich, 1995). A função a ser aprendida será representada por um conjunto de regras pelo algoritmo LAD. A Figura 2.12 exibe um conjunto de dados definido em espaço de duas dimensões. As letras “x”em azul representam as instâncias de classe positiva, enquanto as letras “o”em vermelho representam as instâncias de classe negativa. Os retângulos representam regras (regras positivas em azul e regras negativas em vermelho). O intuito do algoritmo LAD é utilizar um conjunto de regras para criar uma função similar às funções f_1 e f_2, mostradas na figura.

Uma regra é um **conjunto de condições** sobre os valores de um subconjunto de atributos que restringe uma parte do espaço onde o conjunto de dados está definido. Cada regra possui uma cobertura, uma classe e um grau de pureza relacionado à porção dos dados que pertence à região do espaço definida pela regra.

A **cobertura** de uma regra é o valor referente ao número de instâncias do conjunto de dados cobertas pela regra. Diz-se que uma regra cobre uma instância, ou que uma instância é coberta por uma regra, se a instância se encontra dentro da porção do espaço restringida pela regra. Em outras palavras, uma instância é coberta por uma regra se, e somente se, todas as condições contidas na regra forem verdadeiras para os valores contidos na instância (o conceito de condição de regra será descrito nesta seção). Também se pode dizer que a instância satisfaz a regra ou que a regra é satisfeita pela instância.

Uma regra pode ser dita de **classe positiva** ou **negativa** de acordo com a classe majoritária dentre as instâncias do conjunto de dados que são cobertas pela regra. Isto é, se na porção do espaço restringida pela regra existem mais instâncias de classe positiva, a regra será dita positiva, e negativa se existirem mais instâncias de classe negativa.

Figura 2.12: Exemplo de conjunto de dados numérico.



Fonte: autoria própria.

A regra também possui um **grau de pureza**. O grau de pureza de uma regra é dado por:

$$\mathscr{P}(r) = \frac{\mathrm{N_c}}{\mathrm{N_t}}, \tag{2.10}$$

onde $\mathrm{N_c}$ é o número de instâncias que a regra r cobre e que são da mesma classe da regra, e $\mathrm{N_t}$ é o número total de instâncias que a regra cobre. Na Figura 2.12, as regras em vermelho e em azul são ditas puras, isto é, só cobrem elementos de uma cor (classe). Uma **regra pura** é aquela que tem grau de pureza igual a um, em outras palavras, uma regra é pura se $\mathrm{N_c} = \mathrm{N_t}$.

Como mencionado, a regra é composta por um conjunto de condições de regra. Uma vez que, neste ponto, trabalha-se com um conjunto de dados binários, as condições de regra serão sobre tais valores binários. Assim, para um atributo binário, existem duas condições possíveis: se o seu valor do atributo é "1" ou se é "0". Em modelo matemático, cada teste contido em uma regra verifica se o valor de um atributo X_i é igual a $v \in \{0,1\}$.

Por sua vez, uma regra pode conter várias condições, uma para cada atributo binário. Convém lembrar que não haveria sentido se a regra possuísse duas condições sobre o mesmo atributo. Se duas condições comparam o mesmo valor v para o mesmo atributo, estas condições são iguais, não acrescentam informação à regra. Se as duas condições comparam os dois valores possíveis de v não haveria como a instância satisfazer a regra.

Quanto menos condições a regra tiver, mais geral, ou relaxada esta será. De modo que se uma regra possuir m condições e m for igual ao número de atributos, esta regra, no máximo

cobrirá uma instância do conjunto de dados (assumindo que não existam instâncias repetidas). Por exemplo, para o conjunto de dados da Figura 2.11, uma regra:

$$r_1 = \{(b_{A3} = 1), (b_{B1} = 0), (b_{C1} = 1)\}, \tag{2.11}$$

cobriria somente a uma instância do conjunto de dados (a instância x^4). Enquanto uma regra com menos restrições, como:

$$r_2 = \{(b_{A3} = 1)\}, \tag{2.12}$$

cobriria quatro instâncias do conjunto de dados. Para este exemplo, cobrir uma única instância significa cobrir 12,5% do conjunto de dados, o que, dependendo do conjunto de dados, pode representar uma taxa alta de cobertura. Em geral, não se tem conjunto de dados tão pequenos, então uma regra que só cobre uma instância pode ser indesejável.

Adicionalmente, é válido observar que apesar de todas as regras da Figura 2.12 serem formadas por quatro condições (visualmente, cada um dos lados do retângulo), não necessariamente, uma regra, para este exemplo, teria que possuir quatro condições. Por exemplo, uma regra para o espaço definido pelo conjunto de dados, poderia ser representada por uma reta. Poder-se-ia traçar uma reta em x = -0,75, paralela ao eixo y, e assumir que todos os valores que estiverem à esquerda da reta serão ditos positivos.

Por último, é importante destacar uma característica vantajosa dos algoritmos que trabalham com regras: a classificação de uma instância pode ser plenamente justificada, pois é baseada em informações estruturadas, extraídas do conjunto de treinamento, na forma de condições sobre características individuais (Gomes e Bonates, 2011).

2.4.2 Geração de regras

Existem diferentes algoritmos para a geração de regras a partir de conjuntos de dados binários. Por exemplo, T. Bonates em (Bonates et al., 2008) expõe duas maneiras de se gerar regras de decisão. Neste trabalho, propomos um algoritmo de geração de regras desenvolvido com base nos métodos de geração de regras descritos em (Bonates et al., 2008), e introduzindo conceitos de Subespaços Aleatórios de (Ho, 1998). Ambos os conceitos são discussões extensas. Neste trabalho, estes conceitos não serão expostos. Ainda assim, não haverá perda quanto ao entendimento do algoritmo a ser descrito.

O algoritmo a ser exposto é chamado de *Random Rule Generator* (RRG) ("Gerador de Regras Aleatórias", em português). Com um método aleatório de geração de regras e observando pequenas regiões do espaço de regras (conceito a ser discutido nas próximas subseções), é possível se executar o algoritmo várias vezes, com cada execução sendo realizada em um curto intervalo de tempo. Esta é uma característica presente em algoritmos de estado-da-arte nas áreas de aprendizado de máquina (Breiman, 2001), otimização (Holland, 1992) e teste de propriedades (Ron, 2010). Também, pelo RRG ser um algoritmo aleatório, cada uma das suas execuções

têm a possibilidade de fazer escolhas bem diferentes (sem a necessidade de intervenção humana), que levam o algoritmo a explorar diferentes regiões do espaço de regras possíveis para o conjunto de dados em questão.

As subseções seguintes exibem o algoritmo RRG e detalham os seus parâmetros e funções internas. Por último, um exemplo de geração de regras é exibido.

2.4.2.1 Parâmetros do algoritmo RRG

O RRG possui vários parâmetros que controlam seu comportamento. Nesta seção, os parâmetros serão exibidos de forma breve. Isto é, sem maiores discussões sobre os seus usos. Logo após a exibição do algoritmo, detalhes sobre os parâmetros serão discutidos. Os parâmetros do RRG são:

- A classe da regra;
- A pureza mínima desejada da regra;
- A cobertura mínima sobre os elementos da classe da regra;
- O número de atributos aleatórios a cada iteração;
- Uma semente para o gerador de valores aleatórios.

O parâmetro da classe da regra indica, ao gerador de regras, a classe da regra que o algoritmo construirá. Este parâmetro modifica os cálculos da pureza e cobertura da regra (descritos mais à frente), uma vez que é necessária a informação de classe da regra para realizar tais cálculos.

O parâmetro da pureza da regra, como foi discutido na Seção 2.4.1, indica ao gerador que grau de pureza a regra deve possuir para que esta seja uma regra aceitável. Este parâmetro funciona como um limiar, onde somente regras com aquele grau de pureza (ou maior) podem fazer parte da lista de regras do classificador. Se não for possível à regra passar neste teste, então a regra não será aceita.

O parâmetro de cobertura mínima sobre os elementos da classe da regra é, assim como a pureza da regra, um limiar que indica se a regra deve ser rejeitada ou não. Por sua vez, a cobertura sobre os elementos da classe da regra pode ser dada por:

$$\mathscr{M}(r) = \frac{\mathrm{N_c}}{\mathrm{N_{tc}}}, \quad (2.13)$$

onde $\mathrm{N_c}$ é o número de instâncias **que a regra *r* cobre** e que são da mesma classe da regra, e $\mathrm{N_{tc}}$ é o número de instâncias **do conjunto de dados** que são da mesma classe da regra.

O parâmetro do número de atributos pseudoaleatórios em cada iteração remete ao conceito de subespaços aleatórios. Este indica a dimensão dos subespaços aleatórios no qual o algoritmo trabalhará. Este parâmetro será discutido mais à frente.

O parâmetro da semente serve para ajustar o gerador de valores aleatórios, que, por sua vez, é responsável por fazer o algoritmo executar diferentemente, mesmo com todos os outros parâmetros mantendo os mesmos valores.

2.4.2.2 Algoritmo para geração de regras aleatórias

O algoritmo RRG é visto na Figura 2.13. Este será dividido em dois momentos/fases para facilitar a compreensão. O primeiro momento é a escolha de uma condição de regra para ser adicionada à regra em construção. O segundo momento, é a verificação da pureza e da cobertura da regra montada para decidir se a regra está apta a fazer parte da lista de regras do classificador.

Para a primeira fase, alguns pontos devem ser observados (a numeração de linha utilizada é referente ao algoritmo da Figura 2.13):

- **Linha 5**: Gera-se q, uma cópia de r.
- **Linha 7**: T é um subconjunto de, no máximo, f elementos de S. Neste momento acontece a seleção da dimensão e da região do espaço onde o algoritmo irá trabalhar.
- **Linhas 10**: Uma nova condição h é montada. A condição é formada por um índice de atributo t e por um valor $v \in \{0,1\}$.
- **Linha 11**: Uma regra s é gerada a partir da união das condições da regra r com a condição h.
- **Linha 12**: Depois de gerada a nova regra s, esta é comparada com a regra corrente q. Por sua vez, s é considerada melhor que q se o grau de pureza de s for maior que o grau de pureza de q. A fórmula do cálculo do grau de pureza de uma regra foi definida na Equação 2.10.
- **Linhas 13 e 14**: Se o teste da **linha 12** for positivo, a regra recém gerada s é armazenada em q e t, usado para montar a condição h, é armazenado em U.

Primeiramente, deve-se observar que a ação da **linha 7** está ligada ao parâmetro f do algoritmo. A ação seleciona aleatoriamente, no máximo, f elementos do vetor de índices S. Como mencionado anteriormente, é neste momento que acontece a seleção da região do espaço onde o algoritmo irá trabalhar. Em uma análise rápida, para um conjunto de dados definido em um espaço de n dimensões, onde n é o número de atributos do conjunto de dados, faz-se uma seleção de m componentes, onde $m \leq n$, a fim de se ter uma representação em menor dimensão (ou igual) do espaço. Permitimos aqui a igualdade entre m e n por uma questão de completude, mas a vantagem desta abordagem, naturalmente, só existe quando m é menor que n, preferencialmente bem menor que n. Em seguida, dentro do laço da **linha 8**, cada um dos índices $t \in T$, juntamente com um dos valores de v, geram um nova condição de regra (representada por h). A cada iteração do laço, verifica-se se a regra $s := r \cup \{h\}$ é mais pura que a regra q. O laço busca uma condição h que gere uma regra s melhor que r. T representa a região que o algoritmo vai explorar.

Figura 2.13: Pseudocódigo do algoritmo aleatório de geração de um conjunto de regras binárias.

ALGORITMO ALEATÓRIO DE GERAÇÃO DE UM CONJUNTO DE REGRAS BINÁRIAS	
ENTRADA:	Conjunto de dados binários B de duas classes, A classe c da regra a ser gerada, A pureza mínima p aceitável, A cobertura mínima aceitável m da própria classe, O número f de índices de atributos a cada iteração.
SAÍDA:	Um conjunto R de regras.

1. Seja $R := \emptyset$ um conjunto de regras.
2. Seja $r := \emptyset$ uma regra de classe c.
3. Seja $S := \{1, 2, \ldots, k\}$ o conjunto dos índices dos atributos de B
4. **while** $(S \neq \emptyset)$
   `// Fase 1`
5. $\quad q := r$
6. $\quad$ Seja $U = \emptyset$ um índices de atributo.
7. $\quad$ Seja $T \subseteq S$, $|T| \leq f$, um subconjunto de f elementos de S selecionados aleatoriamente.
8. $\quad$ **foreach** $(t \in T)$
9. $\quad\quad$ Seja $v \in \{0, 1\}$, selecionado aleatoriamente, o valor de comparação da nova condição.
10. $\quad\quad$ Seja $h := g(v, t)$ uma condição de regra.
11. $\quad\quad$ Seja $s := r \cup \{h\}$ uma regra de classe c.
12. $\quad\quad$ **if** $(\mathscr{P}(s) > \mathscr{P}(q))$
13. $\quad\quad\quad q := s$
14. $\quad\quad\quad U := t$
15. $\quad\quad$ **end**
16. $\quad$ **end**
    `// Fase 2`
17. $\quad$ **if** $(U = \emptyset$ **or** $\mathscr{M}(q) < m)$
18. $\quad\quad$ **break**
19. $\quad$ **else**
20. $\quad\quad$ r := q
21. $\quad\quad S := S \setminus \{U\}$
22. $\quad\quad$ **if** $(\mathscr{M}(r) \geq m$ **and** $\mathscr{P}(r) \geq p)$
23. $\quad\quad\quad R := R \cup \{r\}$
24. $\quad\quad\quad p := \mathscr{P}(r) + (\frac{1 - \mathscr{P}(r)}{10})$
25. $\quad\quad$ **end**
26. $\quad$ **end**
27. **end**

Fonte: autoria própria.

Outro detalhe desta primeira parte do algoritmo, é o laço da **linha 4**. Este laço falha quando não houver mais conjuntos em S. Isto significa que a regra gerada possui todas as condições

possíveis. Mesmo que este comportamento possa não parecer desejável, pode ser que o usuário, propositalmente, configure o algoritmo de forma a obter tal resultado.

Sobre a segunda fase do algoritmo, podemos destacar os seguintes pontos:

- **Linha 17**: Neste momento, verifica-se se algum índice de atributo foi armazenado em U e, dependendo do resultado deste primeiro teste, se a cobertura $\mathscr{M}(q)$ é menor que a requerida pelo parâmetro m. Se algum dos testes for positivo, o algoritmo finaliza a geração de regras.
- **Linhas 20 e 21**: A regra q é armazenada em r e o índice de atributo U é removido de S.
- **Linha 22**: Verifica-se se $\mathscr{M}(r)$ e $\mathscr{P}(r)$ são aceitáveis.
- **Linhas 23 e 24**: A regra r é adicionada à lista de regras R e a pureza é incrementada.

As duas fases do algoritmo encontram-se dentro de um laço (**linha 4**) e têm a capacidade de serem repetidas várias vezes. A cada iteração do laço, a primeira fase gera uma regra derivada de r, enquanto, a segunda fase é responsável por adicionar ou não esta regra à lista de regras R (de acordo com as condições das **linhas 17** e **22**). O algoritmo deve ser executado diversas vezes. O parâmetro de classe deve ser alternado para que se gerem regras de ambas as classes.

Perceba que o algoritmo de geração de regra, descrito nesta seção, pode retornar mais de uma regra (não necessariamente uma única regra, como outros tipos de algoritmos (Bonates et al., 2008)), assim como pode retornar um conjunto vazio. A possibilidade do algoritmo retornar uma ou mais regras depende dos valores dos parâmetros e, obviamente, do conjunto de dados e da semente. Sendo assim, diz-se que o algoritmo **tentará gerar** um conjunto de regras.

2.4.2.3 Outras considerações

É interessante observar a diferença entre o grau de pureza e de cobertura de uma regra. Enquanto a pureza pode variar subindo ou descendo o seu valor, a cobertura da regra sobre os elementos da mesma classe da regra tende somente a decresce seu valor.

De fato, a pureza de uma regra pode aumentar ou diminuir ao se adicionar uma nova condição à regra. A regra pode passar a cobrir menos instâncias por causa da nova condição, e assim a proporção entre seu $\mathrm{N_c}$ e $\mathrm{N_t}$ muda. Já a cobertura da regra sempre vai diminuir, ou manter-se a mesma, pois, à medida que uma nova condição é adicionada à regra, o valor de $\mathrm{N_c}$ da regra só pode manter-se ou diminuir, enquanto $\mathrm{N_{tc}}$ sempre será o mesmo valor. Por exemplo, seja D um conjunto de dados com 100 instâncias, sendo 60 instâncias positivas e 40 negativas. Também seja r uma regra positiva que cobre 30 instâncias positivas e 10 negativas. Assim, a pureza de r é $\mathscr{P}(r) = 0,75$. A cobertura sobre os elementos da própria classe de r é $\mathscr{M}(r) = 0,5$. Por meio de adição de novas condições à regra r, pode-se criar dois cenários. No primeiro cenário adiciona-se uma condição à regra e esta passa a cobrir 9 instâncias negativas e 15 instâncias positivas, assim $\mathscr{P}(r) = 0,63$ e $\mathscr{M}(r) = 0,25$. No segundo cenário, a regra passa a cobrir 4 instâncias negativas e 15 instâncias positivas, assim $\mathscr{P}(r) = 0,79$ e $\mathscr{M}(r) = 0,25$.

Outra consideração interessante é que as possíveis regras geradas a partir de uma única execução do algoritmo são variações uma das outras. Pode ser que uma ou mais condições sejam adicionadas à regra r antes que esta consiga passar no teste da **linha 22**. Logo, é possível que se gere um conjunto de regras $R = \{r_1, r_2, \ldots, r_k\}$, onde $|r_i| < |r_{i+1}|$, $i < k$. Em outras palavras, r_i é formada por menos condições que r_{i+1}. Não necessariamente r_i terá uma única condição a menos que r_{i+1}, mas, certamente, r_i terá ao menos uma condição a menos.

Ainda, a pureza mínima aceitável para uma regra é sempre incrementada quando uma nova regra é adicionada a R. Ou seja, para que uma versão nova da regra seja adicionada ao conjunto de regras R, esta nova versão deve ser mais pura que a anterior. A fórmula do incremento da pureza vista na **linha 24** do algoritmo, faz incrementos de acordo com o valor da pureza da regra que foi recentemente adicionada.

2.4.2.4 Exemplo de geração de regras

Nesta subseção, um exemplo de geração de regras é exibido (verificar Figura 2.14). O conjunto de dados utilizado é o mesmo exibido na Figura 2.11. Para este exemplo, os índices dos atributos $\mathrm{b_{A3}}$, $\mathrm{b_{B1}}$ e $\mathrm{b_{C1}}$ serão representados por 1, 2 e 3, respectivamente. O exemplo será exibido mostrando apenas algumas informações dos passos do algoritmo.

2.4.3 Regras numéricas

Como foi dito na Seção 2.4.2, as regras geradas têm suas condições em formato binário. Este formato não é bastante amigável quanto a sua apresentação, pois é necessário que se rastreie o ponto de corte que gerou cada condição da regra. Uma vez que os dados eram, originalmente, numéricos, as instâncias que estarão no conjunto de teste estão em formato numérico. Assim, regras numéricas se tornam um formato mais adequado para a exibição das regras geradas.

O processo de transformação de regras binárias em regras numéricas dá-se pelo rastreamento do atributo binário da condição da regra e do valor de v ($v \in \{0,1\}$) que a condição usa para comparação. Com o rastreio do atributo binário da condição é possível descobrir qual atributo numérico gerou o atributo binário, e qual o valor do ponto de corte que gerou tal atributo binário. A partir do valor de v da condição descobre-se qual a relação de comparação que será usada na transformação da condição numérica. De modo que a relação numérica Re é dada por:

$$\mathrm{Re} = \begin{cases} > \text{, se } v = 0 \\ \leq \text{, se } v = 1 \end{cases} \tag{2.14}$$

Assim, a regra binária exibida na Equação 2.11 teria o seguinte formato numérico:

$$r_1 = \{(A \leq 3{,}05), (B > 1{,}70), (C \leq 2{,}40)\}. \tag{2.15}$$

Como resultado, observa-se que, assim como a regra binária da Equação 2.11, a regra

Figura 2.14: Exemplo de execução do RRG

EXEMPLO DO RRG	
PARÂMETROS:	$c = 0$ (classe negativa), $p = 0.95$, $m = 0,01$, $f = 2$
	// Fase 1
1.	$T := \{1,3\}$
2.	$t := 1, v := 0, h := (\mathrm{b}_{\mathrm{A3}} = 0)$
3.	$s := r \cup \{h\} = \{(\mathrm{b}_{\mathrm{A3}} = 0)\}$
4.	$\mathscr{P}(s) = 1/2 = 0,5 > \mathscr{P}(q) = 0$
5.	$q := s, U := t$
6.	$t := 3, v := 1, h := (\mathrm{b}_{\mathrm{C1}} = 1)$
7.	$s := r \cup \{h\} = \{(\mathrm{b}_{\mathrm{C1}} = 1)\}$
8.	$\mathscr{P}(s) = 0/1 = 0 < \mathscr{P}(q) = 0.5$
	// Fase 2
9.	$\mathscr{M}(q) = 1/3 = 0,3333 > m$
10.	$r := q, S := S \setminus \{U\} = \{2,3\}$
11.	$\mathscr{P}(r) < p$
	// Fase 1
12.	$T := \{2,3\}$
13.	$t := 2, v := 1, h := (\mathrm{b}_{\mathrm{B1}} = 1)$
14.	$s := r \cup \{h\} = \{(\mathrm{b}_{\mathrm{A3}} = 0), (\mathrm{b}_{\mathrm{B1}} = 1)\}$
15.	$\mathscr{P}(s) = 1/1 = 1 > \mathscr{P}(q) = 0,5.$
16.	$q := s, U := t$
17.	$t := 3, v := 0, h := (\mathrm{b}_{\mathrm{C1}} = 0)$
18.	$s := r \cup \{h\} = \{(\mathrm{b}_{\mathrm{A3}} = 0), (\mathrm{b}_{\mathrm{B1}} = 1), (\mathrm{b}_{\mathrm{C1}} = 0)\}$
19.	$\mathscr{P}(s) = 0/0 = 0 < \mathscr{P}(q) = 1.$
	// Fase 2
20.	$\mathscr{M}(q) = 1/3 = 0,3333 > m$
21.	$r := q, S := S \setminus \{U\} = \{3\}$
22.	$\mathscr{P}(r) > p$
23.	$R := R \cup \{r\}, p := 1$
	// Fase 1
24.	$T := \{3\}$
25.	$t := 3, v := 0, h := (\mathrm{b}_{\mathrm{C1}} = 0)$
26.	$s := r \cup \{h\} = \{(\mathrm{b}_{\mathrm{A3}} = 0), (\mathrm{b}_{\mathrm{B1}} = 1), (\mathrm{b}_{\mathrm{C1}} = 0)\}$
27.	$\mathscr{P}(s) = 0/0 = 0 < \mathscr{P}(q) = 1.$
28.	$U = \emptyset$

Fonte: autoria própria.

numérica mostrada acima cobre a instância x^4.

2.4.4 Atribuindo pesos às regras

Esta Seção e a Seção 2.5, que apresenta a maneira que as regras são usadas na classificação, estão bastante interligadas. De fato, a classificação é feita a partir dos valores de pesos atribuídos às regras nesta fase. Contudo, a classificação é uma fase posterior a esta e, por isso, esta fase é discutida primeiro.

Seja R o conjunto de regras obtido ao final da fase de geração de regras. $R = R^+ \cup R^-$, onde R^+ é o subconjunto das regras de classe positiva de R, e R^- é o subconjunto de regras de classe negativa de R. Deve-se considerar que, não necessariamente, o conjunto de regras R contém regras de ambas as classes, porque o algoritmo não garante isso. Também, deve-se considerar que os subconjuntos R^+ e R^- não possuem o mesmo número de regras. Em outras palavras $|R^+| \geq 0$, $|R^-| \geq 0$ e $|R^+| \neq |R^-|$. Para as definições a seguir, assumiremos que $|R^+| > 0$ e que $|R^-| > 0$. Os casos em que um destes subconjuntos, ou ambos, são vazios são casos especiais que não serão descritos neste trabalho.

Seja $w(r)$ uma função que retorna o peso de uma regra. Incialmente:

$$w(r) = \begin{cases} \dfrac{1}{|R^+|}, \text{ se } r \in R^+ \\ \dfrac{1}{|R^-|}, \text{ se } r \in R^-. \end{cases} \tag{2.16}$$

Desta forma, as regras de um subconjunto de regras possuirão o mesmo peso e o somatório de todos estes pesos será igual a 1. Assim, os conjuntos tornam-se equivalentes em peso. Contudo, a versão do algoritmo LAD implementada busca "premiar"as melhores regras dando a estas melhores pesos para que estas se destaquem entre as outras do mesmo conjunto, isto é, para que tais regras tenham maior influência no processo de classificação, como ficará evidente na Seção 2.5.

A premiação foi desenvolvida apenas para **ajustar levemente os pesos das regras**, não para procurar o melhor conjunto de pesos de acordo com algum critério específico. Uma vez que o processo de premiação usa o conjunto de treinamento e posteriormente as regras serão usadas em um conjunto de testes, provavelmente diferente do conjunto de treinamento, buscar algum processo de atribuição de pesos ótimos poderia levar o algoritmo ao superaprendizado (Deitterich, 1995).

O processo de premiação usa o conjunto de regras R para estimar a classe de cada instância do conjunto de treinamento e então compara o resultado da classificação (ver Seção 2.5 para detalhes sobre o processo de classificação) com o valor real da classe da instância. A cada instância testada, as regras que cobriram a instância são guardadas em um conjunto de regras satisfeitas por aquela instância. Dentre as regras satisfeitas por uma instância podem existir regras de ambas as classes, o que evidencia, por exemplo, que uma ou mais regras positivas estão afirmando que uma instância negativa é positiva. A premiação acontece incrementando o peso de cada regra que acertou a classificação da instância e decrementando o peso de cada

regra que errou a classificação. O valor do incremento e decremento é um valor pequeno, como, por exemplo, 0,01. Se, em algum momento, a regra tiver seu peso decrementado e este tornar-se negativo, o valor zero será atribuído ao peso da regra.

Após o processo de premiação os pesos das regras são normalizados. O cálculo utiliza os pesos que foram atribuídos às regras pelo processo de premiação. Para facilitar a notação, seja:

$$\mathscr{W}^+ = \sum_{r \in R^+} w(r), \tag{2.17}$$

o somatório dos pesos das regras do conjunto de regras positivas. Analogamente, define-se $\mathscr{W}^-$. O valores finais dos pesos das regras são calculados da seguinte maneira:

$$W(r) = \begin{cases} \dfrac{w(r)}{\mathscr{W}^+}, \text{ se } r \in R^+ \\ \dfrac{w(r)}{\mathscr{W}^-}, \text{ se } r \in R^-. \end{cases} \tag{2.18}$$

Novamente, os conjuntos tornam-se equivalentes em peso, mas agora, devido ao processo de premiação, algumas regras possivelmente tiveram seus pesos modificados.

2.5 CLASSIFICAÇÃO

Classificar uma instância é predizer com base no aprendizado a qual classe aquela instância pertence. A classificação é um processo bastante simples se comparado às outras fases e, como já foi mencionado, está ligada aos valores dos pesos das regras.

Seja $R = R^+ \cup R^-$ um conjuno de regras, onde $R^+ = \{P_1, P_2, \ldots, P_{|R^+|}\}$ é subconjunto das regras positivas de R, e $R^- = \{N_1, N_2, \ldots, P_{|R^-|}\}$ é o subconjunto das regras negativas de R. A classificação de uma instância numérica x usa $\Delta : \mathbb{R}^n \to \mathbb{R}$ especificado da seguinte maneira:

$$\Delta(x) = \sum_{P_i \in R^+} W(P_i) P_i(x) - \sum_{N_i \in R^-} W(N_i) N_i(x), \tag{2.19}$$

onde:

$$P_i(x) = \begin{cases} 1, \text{ se } x \text{ é coberto por } P_i \\ 0, \text{ caso contrário.} \end{cases} \tag{2.20}$$

O valor de $N_i(x)$ é definido de forma similar. Perceba que se o valor do peso de uma regra for 0, esta regra não influencia no valor de $\Delta(x)$.

A classificação de uma instância numérica x é dada por $\mathscr{Y} : \mathbb{R}^n \to \{+1, -1\}$ a seguir:

$$\mathscr{Y}(x) = \begin{cases} +1, \text{ se } \Delta(x) > 0 \\ -1, \text{ se } \Delta(x) < 0. \end{cases} \tag{2.21}$$

Se $\Delta(x) = 0$, a instância poderia ser considerada como sendo de qualquer uma das classes. Esta

versão do algoritmo LAD verifica qual foi a classe mais frequente no conjunto de **dados de treinamento** e utiliza esta informação para predizer a classe da instância.

Deve-se entender que a classificação baseia-se no conjunto de regras encontradas para um dado conjunto de treinamento, por isso não se pode afirmar que o conjunto de regras encontradas representa a função de separação dos dados com exatidão. Logo, assim como qualquer outro algoritmo de classificação existente, quando o classificador LAD prediz que uma instância é positiva ou negativa, significa dizer que, de acordo com o que foi possível aprender, a instância é **considerada positiva ou considerada negativa**.

3 IMPLEMENTANDO UM CLASSIFICADOR NO WEKA

O pacote WEKA (*Waikato Environment for Knowledge Analysis*) é uma coleção de algoritmos de aprendizado de máquina usados para mineração de dados (Hall et al., 2009). Dentre estes, encontram-se algoritmos de classificação, regressão, preprocessamento de dados e seleção de características. O software foi desenvolvido por um grupo de aprendizado de máquina na Universidade de Waikato, Nova Zelândia. O WEKA é bastante utilizado em pesquisas por permitir testes comparativos entre seus algoritmos de aprendizado e por sua facilidade de uso.

O WEKA foi desenvolvido em Java e é, portanto, multiplataforma. O WEKA possui código aberto e está registrado sobre a licença GNU (*GNU's Not Unix*) (GNU, 2007); logo, pode-se modificar seu código original e distribuí-lo livremente. O WEKA é um projeto maduro de cerca de 20 anos, mas que ainda está em evolução. Isto é, novos algoritmos são introduzidos como parte do pacote oficial de versão para versão.

WEKA é uma sigla, portanto será sempre vista neste trabaho escrita em letras maiúsculas. Contudo, os criadores do software preferem usar "Weka" nas janelas da aplicação. O formato pode ter sido escolhido por fatores preferenciais, estéticos, ou para referenciar a ave de mesmo nome que é o símbolo do programa. De todo modo, para que não haja confusão quanto a escrita, este trabalho sempre usará o formato "WEKA" quando se referir à ferramenta.

Este capítulo tem um tom propositál de tutorial. Neste capítulo mostra como criar um projeto Java a partir do código-fonte do WEKA, como criar um classificador dentro do WEKA, quais os principais métodos a serem implementados e com quais classes um classificador deve se relacionar, como funcionam os principais arquivos de configuração do WEKA, e, por fim, como exportar o projeto do WEKA.

3.1 ESCRITOS SOBRE O WEKA

Um dos poucos artigos escritos sobre o WEKA, em português, é o WEKA na Munheca (Santos, 2005). Este é citado aqui como forma de reconhecimento de tal trabalho, que foi pioneiro e tornou-se referência. Contudo, o artigo foi escrito como guia para uma disciplina de pós-graduação do INPE (Instituto Nacional de Pesquisas Espaciais), não aborda a adição de novos elementos ao WEKA. Logo, ele não detalha os itens que este capítulo abordará.

Em inglês, artigos sobre como implementar um novo classificador dentro do WEKA, são poucos. Os artigos disponíveis na Wikispace do WEKA (WEKA Wikispaces, 2014b) (a fonte principal de divulgação sobre o funcionamento interno da aplicação) são confusos e não proveem detalhes suficientes sobre alguns dos pontos que serão abordados aqui.

Diante da situação, é válido afirmar que este capítulo por si só já é uma contribuição considerável à comunidade científica. O que foi escrito é fruto de estudos pessoais, busca em foruns de discussões, e observações dos códigos já existentes dentro do WEKA.

3.2 FERRAMENTAS

Neste trabalho, o **Eclipse 4.2.1** (Eclipse, 2012) foi o IDE (*Integrated Development Environment*) utilizado para criar um projeto Java e posteriormente importar o código-fonte do WEKA. A versão a ser utilizada do WEKA será a **3.7.10** (Hall et al., 2013), facilmente encontrada no *site* oficial da aplicação. Esta é uma versão para desenvolvedores e encontra-se no topo das versões no momento em que este trabalho está sendo escrito. Contudo, os procedimentos descritos a seguir podem ser repetidos, com algumas poucas modificações, em versões passadas da aplicação.

Uma outra opção para a obtenção do código-fonte do WEKA é verificar na pasta de instalação do WEKA, se este estiver instalado em uma máquina. O endereço *default* para o *Microsoft Windows 7 e 8* é `\Program Files\Weka-3-6`. Neste endereço, é possível encontrar um arquivo chamado “weka-src.jar” que pode ser utilizado nos procedimentos a seguir (a versão 3.6.x, citada, é a última versão disponibilizada a usuários comuns, porém este trabalho se propõe ao uso da versão 3.7.10 para os estudos).

Também é necessário que se possua o *Java Development Kit* (JDK) instalada na máquina. A versão a ser utilizada é a 1.7.0 17 (Oracle, 2013).

As próximas subseções mostram como configurar o projeto Java no Eclipse, sua estrutura de construção e a importação de algumas bibliotecas necessárias para trabalhar com os códigos do WEKA.

3.3 CRIANDO UM NOVO PROJETO NO ECLIPSE

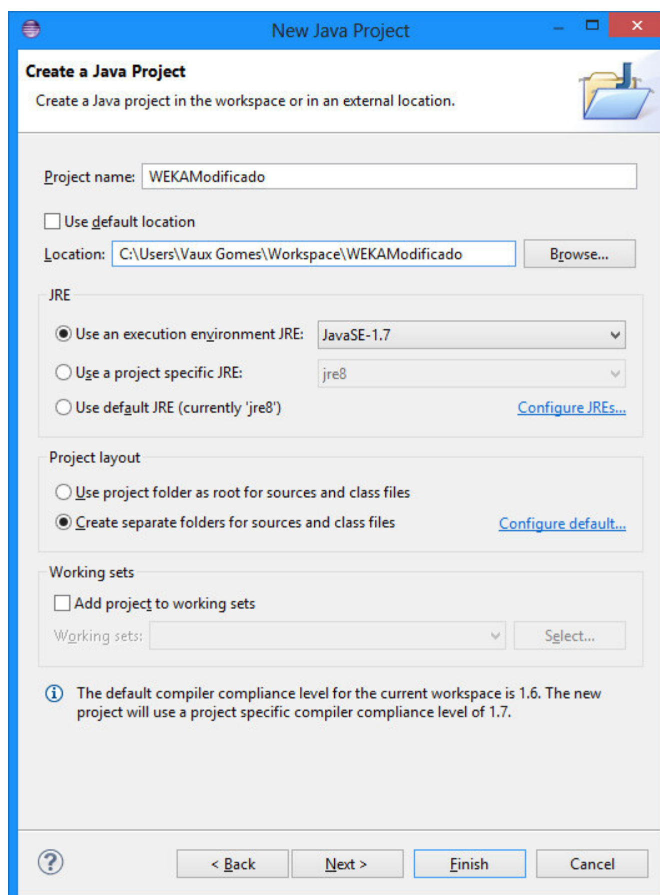
Para criar um projeto no Eclipse deve-se seguir pelos menus `File > New > Java Project`. Na janela *New Java Project*, exibida na Figura 3.15, deve-se escolher um nome para o projeto (WEKAModificado, por exemplo), e, opcionalmente, um caminho para a localização do código-fonte. Ao clicar no botão *Finish*, um novo projeto Java será criado e exibido na guia *Package Explorer* do Eclipse.

3.4 IMPORTANDO O CÓDIGO-FONTE DO WEKA

Um vez construído um projeto Java, deve-se importar o código-fonte do WEKA. Deve-se seguir pelos menus `File > Import`. Na janela *Import*, exibida na Figura 3.16, deve-se selecionar a opção `General > Archive File` e clicar em *Next*.

No passo seguinte, no campo *Into folder*, deve-se escolher o nome do projeto que receberá o código-fonte do WEKA, e no campo *From archive file* deve-se procurar pelo arquivo *weka-src.java*. O arquivo *weka-src.java* é encontrado no repositório do WEKA no *site* da aplicação ou dentro da pasta de instalação do WEKA, como mencionado na Seção 3.2. A Figura 3.17 mostra como configurar a importação do código.

Figura 3.15: Edição das configurações do novo projeto Java.



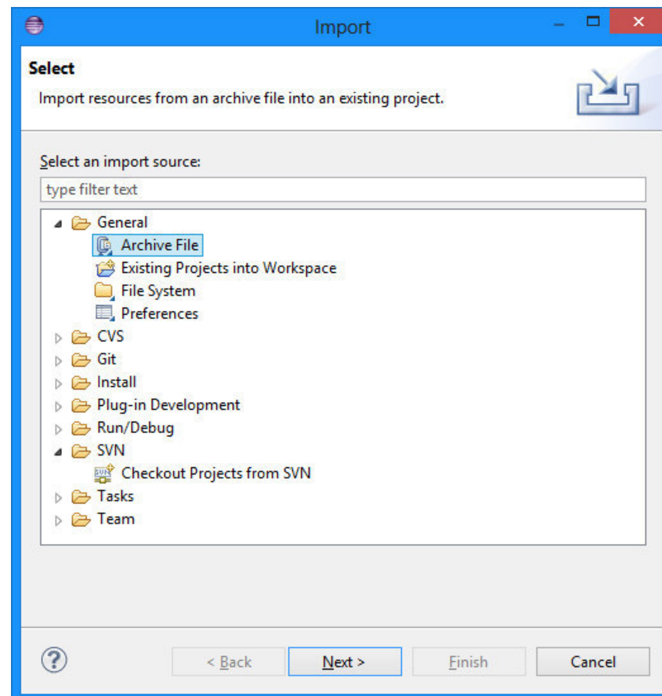
Fonte: autoria própria.

Após encontrar o arquivo de código-fonte do WEKA e escolher qual projeto receberá o código basta clicar no botão *Finish* e o Eclipse copiará as classes para dentro do projeto. Contudo, ainda não é o bastante para que se tenha um projeto executável. O projeto WEKA ainda precisa de bibliotecas adicionais e de um *Build Path* específico. Para efetuar tais mudanças deve-se ir à guia *Package Explorer*, e clicar com o botão direito sobre o nome do projeto e então clicar na opção *Properties*. Uma janela chamada *Properties for WEKAModificado* contendo uma série de submenus que possibilitam configurar o projeto, será mostrada.

Sob o submenu *Java Build Path*, na parte direita da janela encontram-se quadro abas (*Source*, *Projects*, *Libraries* e *Order and Export*). Primeiramente, deve-se selecionar a aba *Source* e então remover todos os itens exibidos no quadro *Source folders on build path* (a aba *Source* pode ser visualizada na Figura 3.19). Em seguida, deve-se clicar no botão *Add Folder* e adicionar os caminhos `src > main > java` e `src > test > java` como exibido na Figura 3.18. O resultado desta operação deve ser igual ao exibido na Figura 3.19.

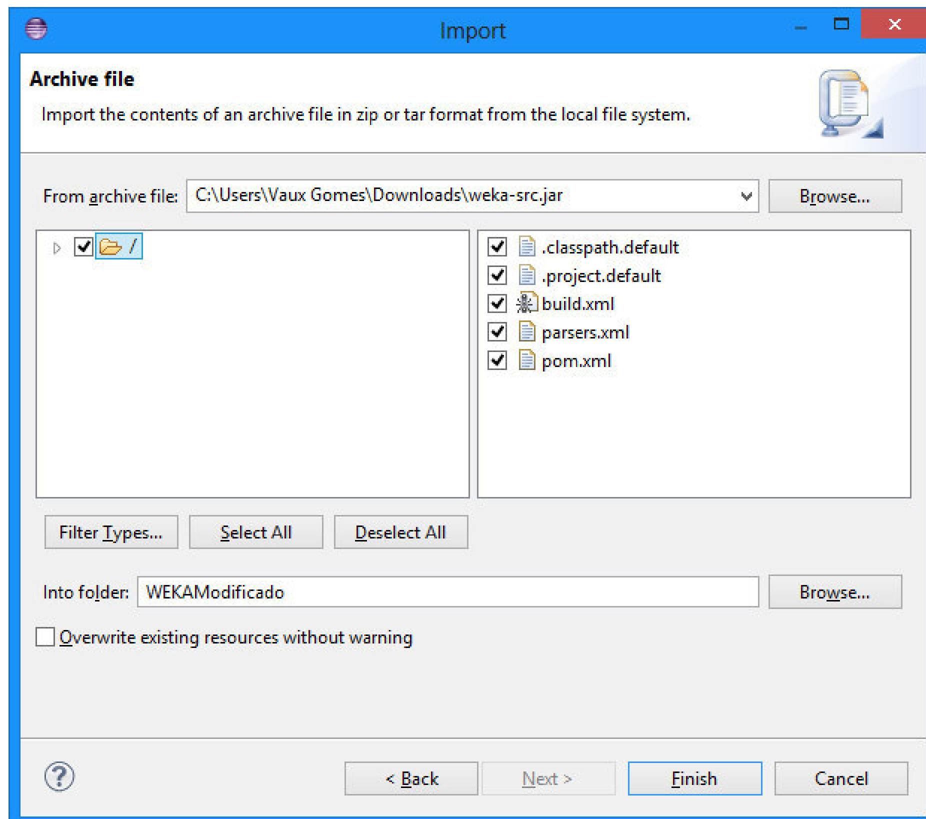
Na aba *Libraries* deve-se clicar no botão *Add JARs*, selecionar todas as bibliotecas encontradas na pasta *lib* do projeto WEKA, como mostrado na Figura 3.20, e clicar no botão *OK*. O resultado desta operação deve ser igual ao exibido na Figura 3.21. A janela *Properties for WEKAModificado* pode ser fechada.

Figura 3.16: Janela de importação 1.



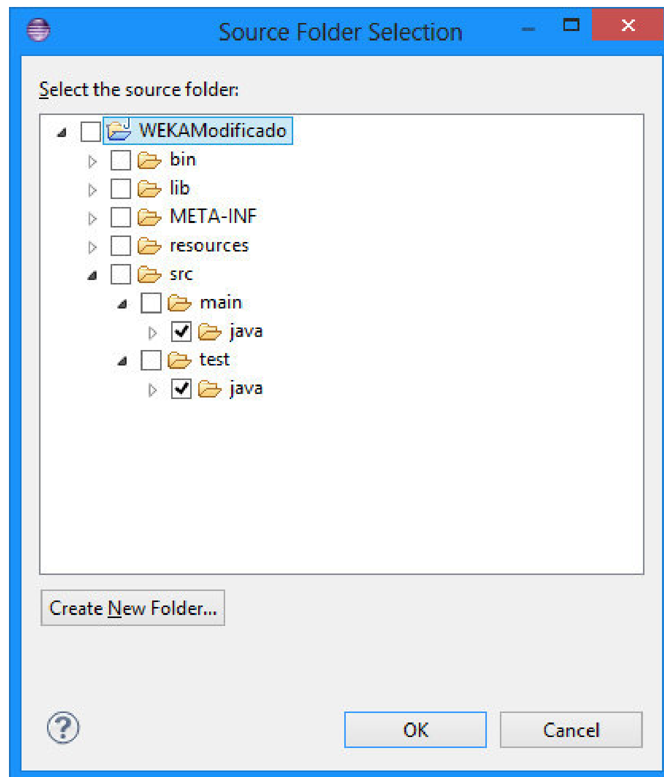
Fonte: autoria própria.

Figura 3.17: Janela de importação 2.



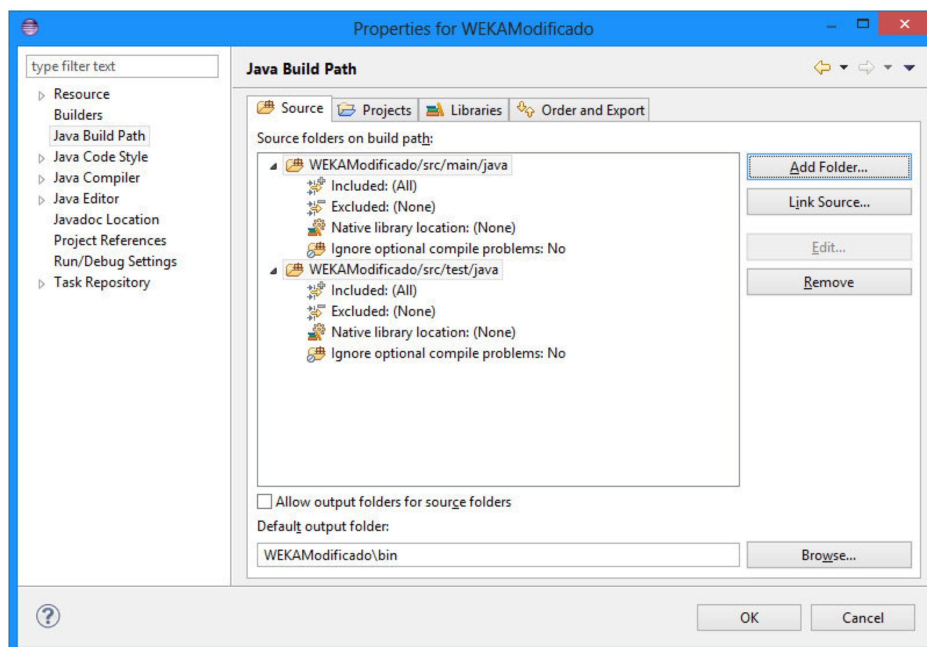
Fonte: autoria própria.

Figura 3.18: Adição dos novos caminhos de construção.



Fonte: autoria própria.

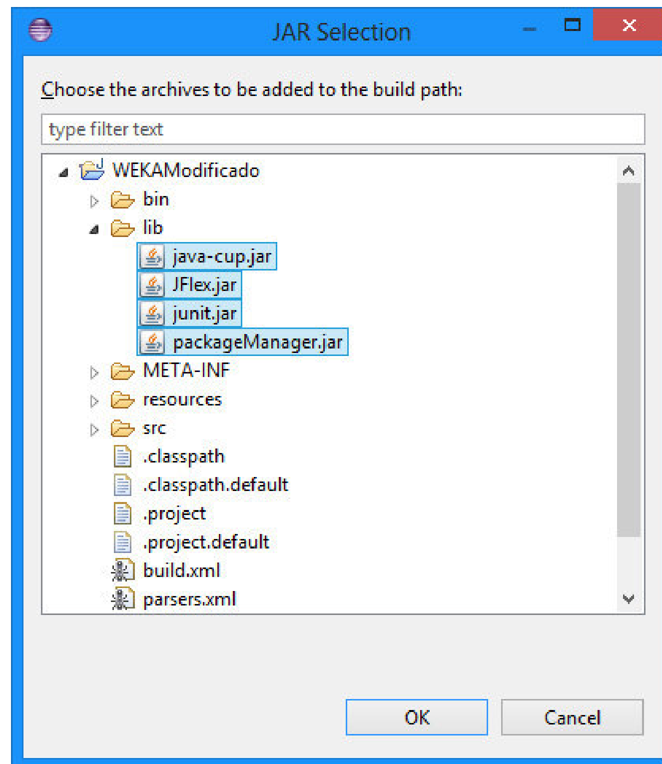
Figura 3.19: Resultado da adição dos novos caminhos de construção.



Fonte: autoria própria.

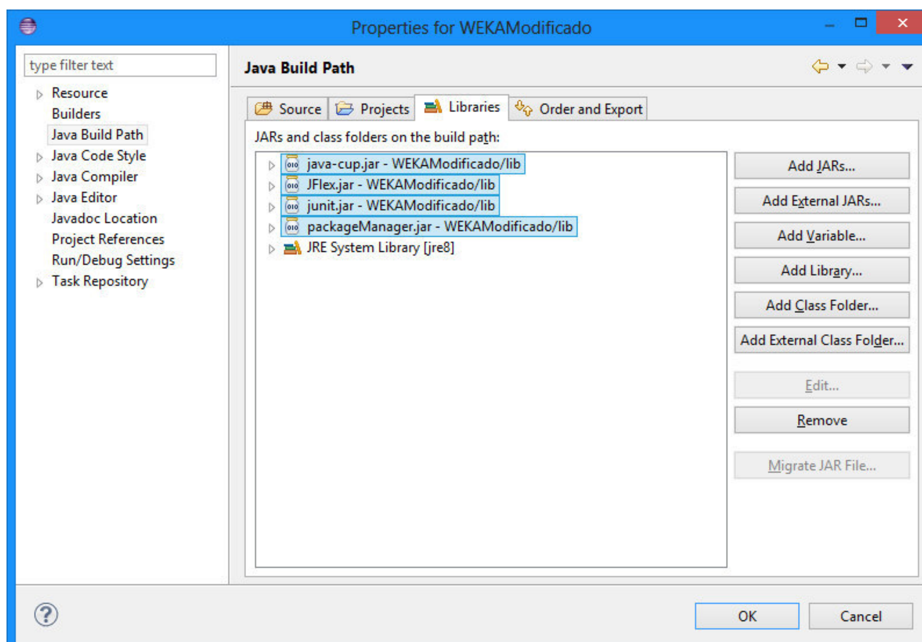
Ao fim deste processo, o projeto pode ser compilado e executado. A classe que contém o método *main* é a classe `weka.gui.GUIChooser`. A classe `GUIChooser` mostra a primeira janela do WEKA, onde se tem a chance de escolher qual módulo do WEKA se deseja usar.

Figura 3.20: Adição dos arquivos JAR ao projeto.



Fonte: autoria própria.

Figura 3.21: Resultado da adição dos arquivos JAR ao projeto.



Fonte: autoria própria.

3.5 CRIANDO UM CLASSIFICADOR SIMPLES

Antes de começar a codificar o classificador, deve-se conhecer um pouco da estrutura do WEKA. As próxima subseções mostrarão como o WEKA recebe, entende e estrutura os dados dentro de si. Em seguida, este documento mostra a construção de um classificador simples e de alguns detalhes relacionados à iteração do classificador com a *interface* do WEKA.

3.5.1 O tipo ARFF e as classes *Instances*, *Instance* e *Attribute*

O WEKA trabalha, principalmente, com um formato de dados específico chamado ARFF (*Attribute-Relation File Format*), usado para padronizar a entrada de dados. Informações sobre o formato ARFF podem ser encontradas em (WEKA Wikispaces, 2014b), mas, para que este trabalho não tenha muita dependência com fontes externas, o formato ARFF será discutido a seguir. A Figura 3.22, exibe um exemplo de um arquivo ARFF.

Figura 3.22: Exemplo de um arquivo ARFF.

```
1   @relation example
2
3   @attribute att1 numeric
4   @attribute att2 numeric
5   @attribute att3 numeric
6   @attribute att4 numeric
7   @attribute class {0, 1}
8
9   @data
10  5,1,1,3,0
11  5,4,10,3,1
12  3,1,2,3,0
13  6,8,4,3,1
14  4,1,1,3,0
15  8,10,10,9,1
16  1,1,10,3,0
17  2,2,1,3,0
18  2,1,1,1,0
```

Fonte: autoria própria.

Resumidamente, o padrão ARFF possui três *tokens*, ou seções, principais: `relation`, `attribute` e `data`. Os *tokens* são denotados pelo símbolo '@' seguidos do seu devido nome e de seus valores. O *token* `relation` usa o formato `@relation <dataset name>` e referencia o nome do conjunto de dados. O *token* `attribute` usa o formato `@attribute <att name> <att type>` e referencia o nome e o tipo de cada atributo. No caso da Figura 3.22, todos os atributos são numéricos, mas eles podem ser de tipo inteiro, real, nominal, *string*, etc.

Conjuntos de dados para aprendizado supervisionado, obrigatoriamente, têm o valor de classe para cada uma das instâncias. Para o WEKA, como visto na Figura 3.22, a classe também

é um atributo. Na figura, o atributo de classe recebe o nome de *class* ("classe", em português). Não necessariamente este atributo, por receber este nome, seria o atributo que referencia a classe. No WEKA, por definição, o último atributo da lista de atributos do ARFF será o atributo de classe (WEKA Wikispaces, 2014a). Adicionalmente, em conjuntos de dados de problemas de classificação supervisionada a classe deve ser nominal. Por este motivo, o atributo *class* apresenta seus valores no formato nominal ($\{$`classe`$_1$`, classe`$_2$`, ..., classe`$_n\}$).

Por último, o *token* `data` usa o formato `@data` seguido de i linhas, cada uma representando uma instância, no formato `<att value`$_1$`, att value`$_1$`, ..., att value`$_j$`>`, sendo j o número de atributos descritos pelos *tokens* `@attribute`.

Apesar do formato ARFF organizar e estruturar dos dados, é importante ressaltar que, internamente, o WEKA usa uma estrutura bem diferente, mas que está relacionada aos *tokens* presentes no formato. As classes `Attribute`, `Instance` e `Instances` do pacote `weka.core` são as classes responsáveis por representar, respectivamente, um atributo, uma instância com todos os valores dos seus atributos, e o conjunto total de instâncias contidas no conjunto de dados.

3.5.2 Inserindo um novo classificador

Ao criar um novo classificador, primeiramente deve-se escolher o endereço do classificador dentro do WEKA e o seu nome. Estas informações foram fixadas para o intuito deste trabalho e para que a escrita torne-se mais clara.

Para os passos seguintes, deve-se estar certo que se está trabalhando no diretório de montagem `src/`**main**`/java`, que é o diretório principal de algoritmos do WEKA, e não no `src/`**test**`/java`. O WEKA possui sua própria pasta de classificadores, referenciada pelo pacote `weka.classifiers`, encontrada em ambos diretórios de montagem do projeto.

A partir dos menus `File > New > Package` deve-se adicionar um novo pacote `weka.classifiers.classificadorSimples` ao projeto. Ainda, deve-se seguir pelos menus `File > New > Class` para se adicionar a classe do novo classificador, no endereço que acabou de ser de criado. Para o exemplo deste capítulo, esta classe será chamada `ClassificadorSimples`.

O nome e a localização da classe não são importantes neste momento, embora estas informações sejam necessárias mais à frente para que o WEKA possa carregar a classe, e o classificador possa aparecer na *interface* junto dos demais classificadores.

Após a criação da classe do classificador, deve-se identificá-la como sendo do tipo classificador do WEKA. Para isto, o WEKA possui uma *interface* chamada `Classifier`, localizada no pacote `weka.classifiers`, possui todos os métodos essenciais usados por um classificador no WEKA, mas, por praticidade, usaremos a classe abstrata `AbstractClassifier` (localizada no mesmo pacote) que é uma classe que implementa alguns dos métodos de `Classifier`. Esta classe deriva todos os demais classificadores do WEKA na versão 3.7.10.

A classe `AbstractClassifier` existe na versão 3.7.10 do WEKA, mas a versão 3.6.10 não a possui. Contudo, os passos que seriam necessários aplicar para `AbstractClassifier`

podem ser replicados para a `Classifier` na versão 3.6.10. Como observação, `Classifier` é uma classe, não uma *interface*, na versão 3.6.10.

As subseções seguintes abordam os principais métodos da classe `AbstractClassifier`. Estes métodos são o método `buildClassifier`, onde treina-se o classificador, os métodos `classifyInstance` e `distributionForInstance`, onde classificam-se as instâncias. Ainda discutiremos outro método que compõe `AbstractClassifier` para que se tenha um maior entendimento de um classificador do WEKA.

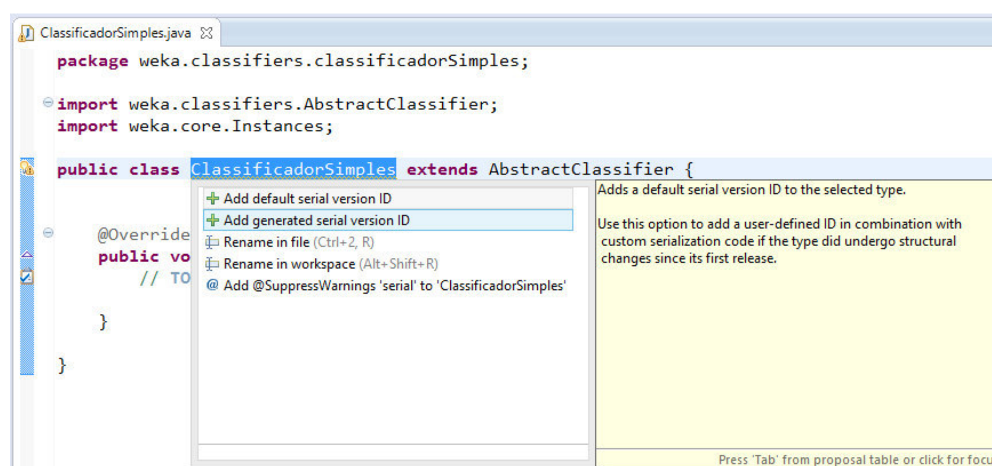
Ao modificar a classe `ClassificadorSimples` para um tipo classificador do WEKA, usando *extends* e fazendo referencia à classe `AbstractClassifier`, o Eclipse requisita a sobrecarga de apenas um dos métodos citados anteriormente, isto é, o método `buildClassifier`. Será necessário que também se sobrescreva o método `classifyInstance` ou do método `distributionForInstance`. Estes métodos são vistos em mais detalhes nas seções seguintes.

3.5.3 Implementando *Serializable*

Existe a necessidade de que o classificador implemente a *interface* `Serializable`. Esta *interface* serve para que o WEKA possa transformar os objetos em conjuntos de *bytes* de uma forma padronizada.

A classe `AbstractClassifier` já implementa esta interface. Assim, deve-se apenas selecionar o `serialVersionUID` para a classe do classificador. Esta variável indica a **versão da classe**. O valor padrão para esta variável é **1L**, mas o próprio Eclipse pode ajudar a gerar um identificador aleatório. Basta clicar no *warning* ao lado esquerdo da linha que define o nome da classe do classificador e requisitar um valor aleatório para o identificador de versão desta classe (opção *add generated serial version*). A Figura 3.23 mostra o exemplo de como gerar um identificador aleatório. O valor do `serialVersionUID` torna-se importante quando se tem várias versões da mesma classe.

Figura 3.23: Selecionando o `serialVersionUID`.



Fonte: autoria própria.

3.5.4 Treinamento

O treinamento de um classificador acontece no método `buildClassifier`. Cada classificador tem a sua maneira de aprender, e, portanto, tem implementação particular. A fim de evitar uma grande discussão a respeito da técnica e da codificação de um classificador mais complexo, este trabalho opta por construir um classificador bem simples conhecido como Zero. O próprio WEKA já implementa este classificador e o chama de `ZeroR`. O algoritmo Zero apenas busca a classe mais frequente no conjunto de treinamento, guarda o seu valor e o usa como saída para qualquer instância a ser classificada.

Existem várias maneiras de se implementar o Zero. Um exemplo de implementação é manter uma variável interna que indica qual foi a classe mais frequente nos dados de treinamento. Assim, uma passagem simples (dentro do método `buildClassifier`) por todas as instâncias do conjunto de treinamento e uma contagem das frequências das classes seria suficiente para realizar o aprendizado. Na Figura 3.24 um método específicos da classe `Instances` foi utilizado pra fazer a contagem dos valores, e a variável chamada `MFC` foi usada para armazenar o índice da classe mais frequente.

3.5.5 Classificando instâncias

A classificação de uma instância por qualquer classificador segue um mesmo formato, visto que a saída da classificação é padronizada pelo WEKA. Tal classificação se dá através de as ambas funções `distributionForInstance` ou `classifyInstance`. De fato, uma breve observação no código-fonte da classe `AbstractClassifier` mostra que um método faz menção ao outro (veja no código do projeto em construção). Assim, faz-se necessária a reimplementação de apenas um dos métodos.

A diferença entre os dois métodos é, basicamente, seu parâmetro de retorno. Um *double* para `classifyInstance` e um vetor de *double* para `distributionForInstance`. O método `distributionForInstance`, por vez, é um pouco mais detalhista. Cada posição do vetor de *double* informaria ao programa a probabilidade que a classe de índice i tem de ser a classe da instância que está sendo classificada. Isto é, se o vetor retornado pela função for $\{0{,}6;\ 0{,}4;\ 0\}$ significa dizer que, de acordo com o algoritmo, as classes de índice 0, 1 e 2 têm, respectivamente 60, 40 e 0 porcento de chance de ser a classe da instância em processo de classificação. Já o método `classifyInstance` retorna o valor índice da classe de maior probabilidade, ou seja, esta função simplesmente retornaria o valor 0 para o exemplo citado neste parágrafo. Vale lembrar que os índices das classes estão relacionados ao vetor de classes denotado no arquivo ARFF.

Por praticidade, o método `classifyInstance` foi implementado no exemplo da classe `ClassificadorSimples`. O método apenas retorna o valor do índice guardado na fase de treinamento, isto é, o valor de `MFC`. Neste ponto, a implementação da classe aparenta-se com o que é mostrado na Figura 3.24.

3.5.6 Parâmetros da *interface*

Alguns classificadores possibilitam ao usuário final o ajuste do valor de um ou mais dos seus parâmetros. Esta é uma função útil, pois é uma forma do usuário modificar o funcionamento do algoritmo para tentar fazer com que o algoritmo se adeque melhor aos dados. Tais ajustes são **baseados na crença** que o usuário tem de que os dados seguem certo formato. Um exemplo da janela de ajuste dos parâmetros pode ser vista na Figura 4.37

O WEKA fornece uma maneira para que o classificador especifique quais parâmetros disponibilizar na *interface* visual. Ao carregar as classes, O WEKA procura no código-fonte por métodos específicos relacionados à exibição de parâmetros na *interface* de usuário. São estes os métodos `get`, `set` e `tipText`. Os métodos devem seguir um padrão como a seguir:

- `<tipo do parâmetro> get<nome do parâmetro>(),`
- `set<nome do parâmetro>(<tipo do parâmetro> var),`
- `String <nome do parâmetro>tipText().`

Obrigatoriamente, os métodos `get` e `set` devem ser implementados para que, através da interface, os parâmetros possam ser modificados. Estes métodos são responsáveis por salvar e buscar o valor do parâmetro. Se um destes dois métodos não existir na classe do classificador, o parâmetro não será mostrado na *interface* do WEKA.

Por último, o método `tipText` é responsável por exibir uma mensagem na tela quando o *mouse* é posicionado sobre a caixa de texto relacionada ao parâmetro. Este método é apenas um método complementar que pode dar ao usuário alguma ideia da utilização do parâmetro.

Como exemplo, a Figura 3.25 mostra o resultado final da implementação (vista na Figura 3.24) dos três métodos para a variável `MFC` gerada na Seção 3.5.4. O parâmetro `debug`, que aparece na imagem, é derivado da classe `AbstractClassifier`.

3.5.7 Habilitando as capacidades do classificador

O WEKA permite que cada classificador especifique suas capacidades/habilidades, isto é, com que tipo de dados este classificador pode trabalhar. Se por acaso um classificador não pode trabalhar com instâncias que possuam valores faltantes, basta que este desative tal habilidade. O efeito de desativar uma habilidade de um classificador faz com que o WEKA impossibilite o classificador de executar (treinar e testar) um certo conjunto de dados, visto que o próprio classificador avisou que não poderia trabalhar com dados que possuem uma certa característica em particular.

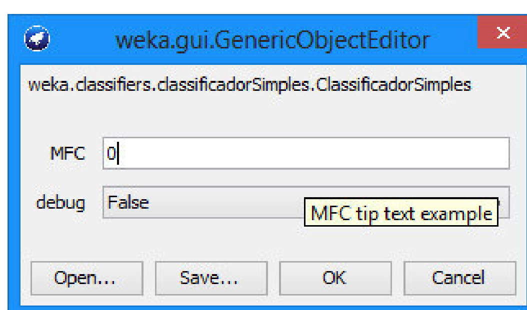
A classe `AbstractClassifier` implementa a *interface* `CapabilitiesHandler` que possui o método `getCapabilities`, onde deve-se listar as habilidades do classificador. A Figura 3.26 mostra a implementação deste método pela classe `AbstractClassifier`. O método foi implementado de forma a aceitar dados com quaisquer características.

Figura 3.24: Código resumido do classificador simples.

```
ClassificadorSimples.java
package weka.classifiers.classificadorSimples;

import weka.classifiers.AbstractClassifier;

public class ClassificadorSimples extends AbstractClassifier {

    /** SERIAL ID */
    private static final long serialVersionUID = 8015782592491444533L;

    private int MFC = 0; // Most Frequent Class

    /** Training */
    public void buildClassifier(Instances data) throws Exception {
        for (int i = 0; i < data.classAttribute().numValues(); i++)
            if (data.attributeStats(data.classIndex()).nominalCounts[MFC] < data
                    .attributeStats(data.classIndex()).nominalCounts[i])
                MFC = i;
    }

    /** Testing */
    public double classifyInstance(Instance instance) throws Exception {
        return MFC;
    }

    public int getMFC() {
        return MFC;
    }

    public void setMFC(int mFC) {
        MFC = mFC;
    }

    public String MFCTipText() {
        return "MFC tip text example";
    }
}
```

Fonte: autoria própria.

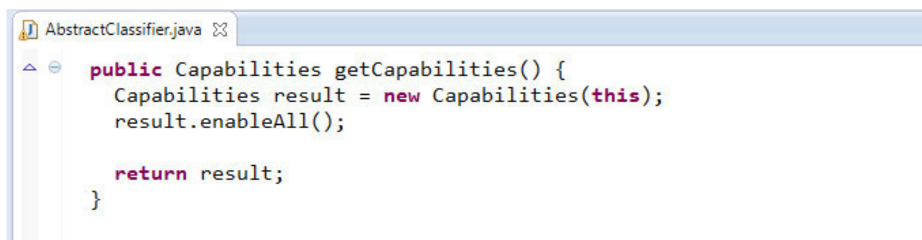
Figura 3.25: Adição de parâmetro à *interface* do classificador.



Fonte: autoria própria.

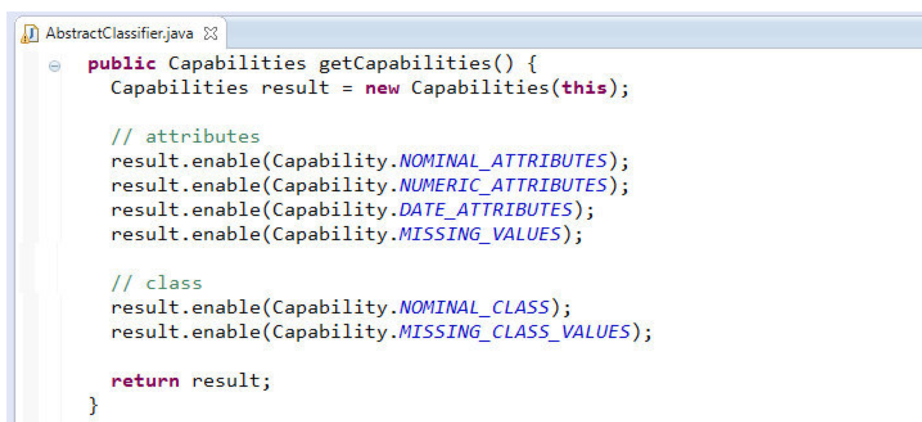
O método `getCapabilities` é importante dentro do WEKA e é utilizado para evitar erros em tempo de execução por parte dos classificadores, mas, para o classificador construído neste capítulo, este método não será modificado. Um outro exemplo de como ativar algumas habilidades que um classificador consegue especificar pode ser vista na Figura 3.27.

Figura 3.26: Métodos `getCapabilities` *default*.



Fonte: autoria própria.

Figura 3.27: Método `getCapabilities`.



Fonte: autoria própria.

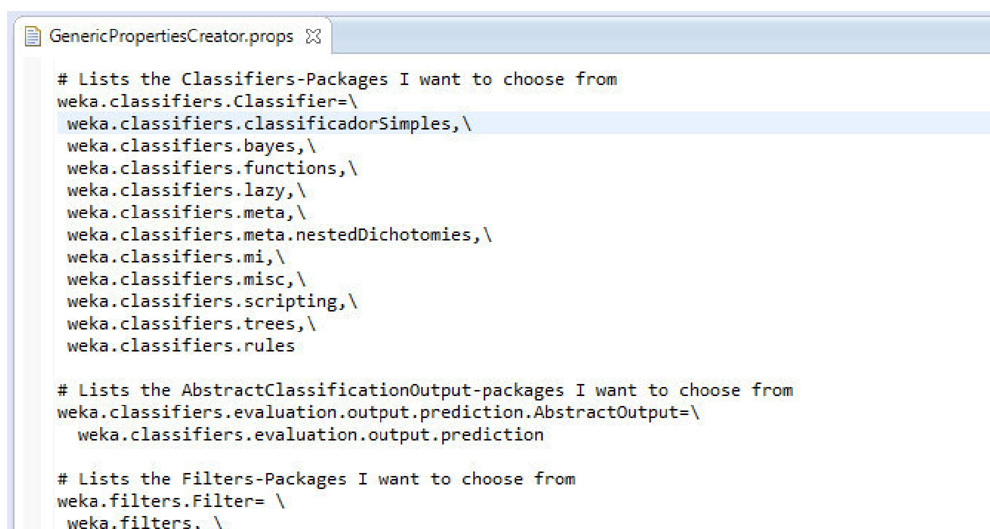
3.6 ARQUIVOS PROPS

Para que o classificador possa aparecer na *interface* visual do WEKA, é necessário modificar alguns dos arquivos de configuração do programa. O WEKA utiliza vários arquivos de configuração do tipo Props. Assim como os arquivos do tipo ARFF, os arquivos Props são arquivos de texto que seguem um padrão específico. Ambos os arquivos, a serem modificados, encontram-se no pacote `weka.gui` do projeto.

O primeiro arquivo a ser modificado é o arquivo `GenericPropertiesCreator.props`. Este arquivo, dentre outras coisas, é responsável por avisar ao WEKA onde (em que diretórios) procurar por objetos que são subclasses da classe `Classifier`. O segundo arquivo a ser modificado é o arquivo `GenericObjectEditor.props`. Este, dentre outras coisas, avisa ao WEKA quais classificadores devem ser exibidos na interface.

As modificações seguem o exemplo do nome da classe e do diretório escolhidos na Seção 3.5.2. A Figura 3.28 mostra as modificações para o arquivo `GenericPropertiesCreator`, enquanto a Figura 3.29 exibe as modificações para o arquivo `GenericObjectEditor` (destaque em azul claro).

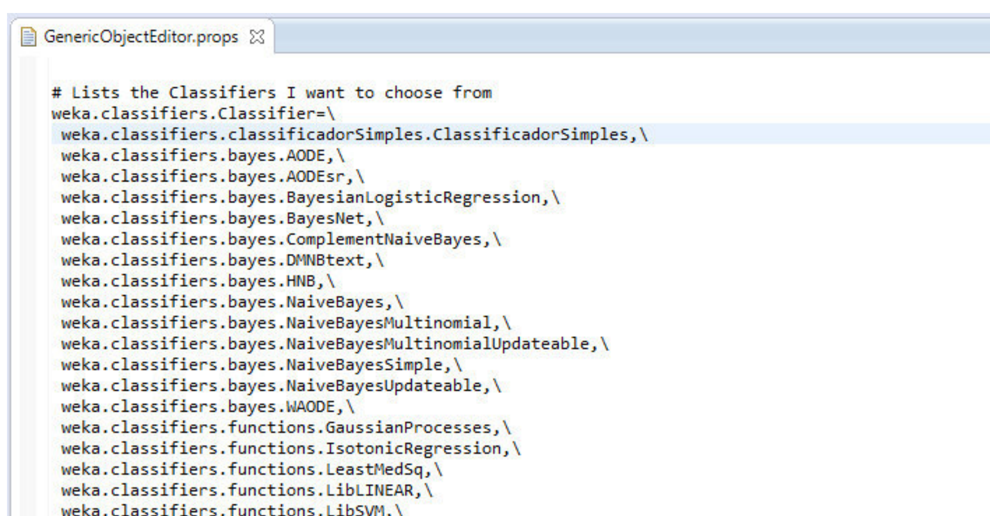
Figura 3.28: `GenericPropertiesCreator.props` modificado.



```
# Lists the Classifiers-Packages I want to choose from
weka.classifiers.Classifier=\
 weka.classifiers.classificadorSimples,\
 weka.classifiers.bayes,\
 weka.classifiers.functions,\
 weka.classifiers.lazy,\
 weka.classifiers.meta,\
 weka.classifiers.meta.nestedDichotomies,\
 weka.classifiers.mi,\
 weka.classifiers.misc,\
 weka.classifiers.scripting,\
 weka.classifiers.trees,\
 weka.classifiers.rules

# Lists the AbstractClassificationOutput-packages I want to choose from
weka.classifiers.evaluation.output.prediction.AbstractOutput=\
  weka.classifiers.evaluation.output.prediction

# Lists the Filters-Packages I want to choose from
weka.filters.Filter= \
 weka.filters, \
```

Fonte: autoria própria.

Figura 3.29: `GenericObjectEditor.props` modificado.



```
# Lists the Classifiers I want to choose from
weka.classifiers.Classifier=\
 weka.classifiers.classificadorSimples.ClassificadorSimples,\
 weka.classifiers.bayes.AODE,\
 weka.classifiers.bayes.AODEsr,\
 weka.classifiers.bayes.BayesianLogisticRegression,\
 weka.classifiers.bayes.BayesNet,\
 weka.classifiers.bayes.ComplementNaiveBayes,\
 weka.classifiers.bayes.DMNBtext,\
 weka.classifiers.bayes.HNB,\
 weka.classifiers.bayes.NaiveBayes,\
 weka.classifiers.bayes.NaiveBayesMultinomial,\
 weka.classifiers.bayes.NaiveBayesMultinomialUpdateable,\
 weka.classifiers.bayes.NaiveBayesSimple,\
 weka.classifiers.bayes.NaiveBayesUpdateable,\
 weka.classifiers.bayes.WAODE,\
 weka.classifiers.functions.GaussianProcesses,\
 weka.classifiers.functions.IsotonicRegression,\
 weka.classifiers.functions.LeastMedSq,\
 weka.classifiers.functions.LibLINEAR,\
 weka.classifiers.functions.LibSVM,\
```

Fonte: autoria própria.

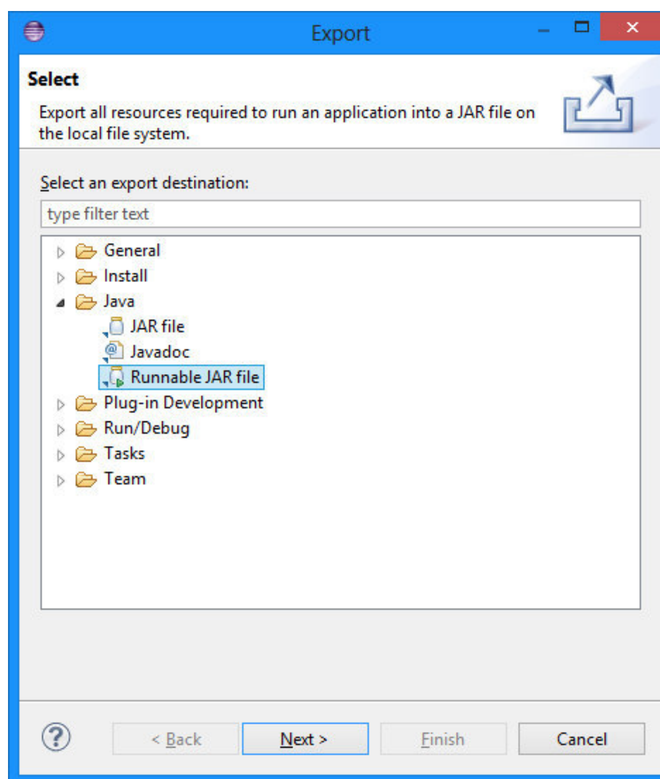
3.7 EXPORTANDO O WEKA MODIFICADO

Para finalizar a primeira parte do trabalho, o código do projeto WEKAModificado deve ser exportado em formato JAR. O Eclipse facilita um pouco o processo, mas há ainda alguns detalhes que devem ser discutidos.

Para iniciar o processo, deve-se seguir pelos menus `File > Export`. Na janela *Export*, deve-se expandir a pasta *Java*, selecionar a opção *Runnable JAR file*, como exibido na Figura 3.30, e clicar em *Next*.

Na janela *Runnable JAR File Export* é necessário usar a caixa de seleção do submenu *Launch configuration* para escolher a classe principal do projeto a ser exportado. Deve-se procurar pela classe `weka.gui.GUIChooser` do projeto. Algumas vezes, a classe principal do

Figura 3.30: Selecionando modo de exportação.



Fonte: autoria própria.

projeto que se quer não aparece na caixa de seleção, isto pode acontecer se o projeto nunca tiver sido executado. Se isto acontecer, basta abrir o projeto e executar a classe `GUIChooser`.

No submenu *Export destination*, deve-se selecionar o diretório de destino e o nome do arquivo a ser exportado. Em seguida, deve-se selecionar *Extract required libraries into generated JAR* no submenu *Library handling*. Este último passo permite exportar o projeto em apenas um arquivo que pode ser utilizado para substituir o arquivo de execução do WEKA instalado em uma máquina. Para exportar o arquivo deve-se clicar no botão *Finish*. A Figura 3.31 mostra detalhes da janela *Runnable JAR file*.

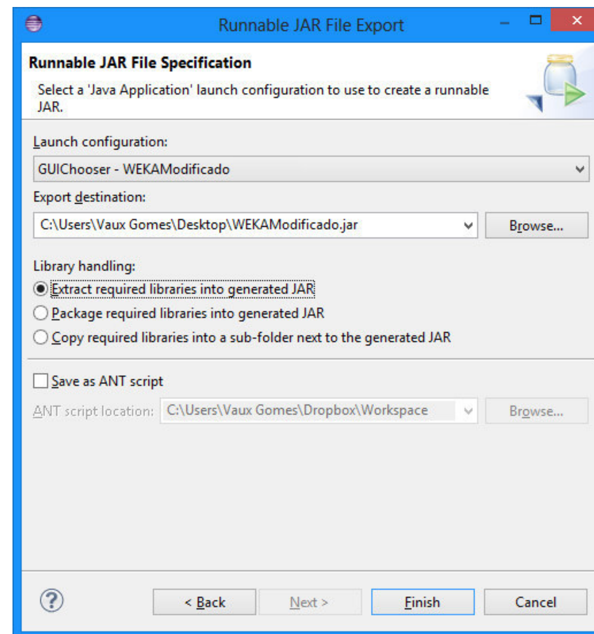
3.8 INSTALAÇÃO DO WEKA MODIFICADO

A instalação do WEKA modificado consiste da substituição do arquivo "weka.jar" do diretório de instalação do WEKA (por exemplo, o endereço descrito na Seção 3.2) pelo WEKA modificado exportado pelo Eclipse.

Como observação, é interessante manter a versão original do arquivo "weka.jar" para uso futuro (se necessário). A substituição pode ser vista na Figura 3.32. A imagem exibe o arquivo "weka - original.jar" (arquivo do WEKA que foi originalmente instalado e agora renomeado) e o arquivo "weka.jar" (arquivo WEKA exportado pelo Eclipse).

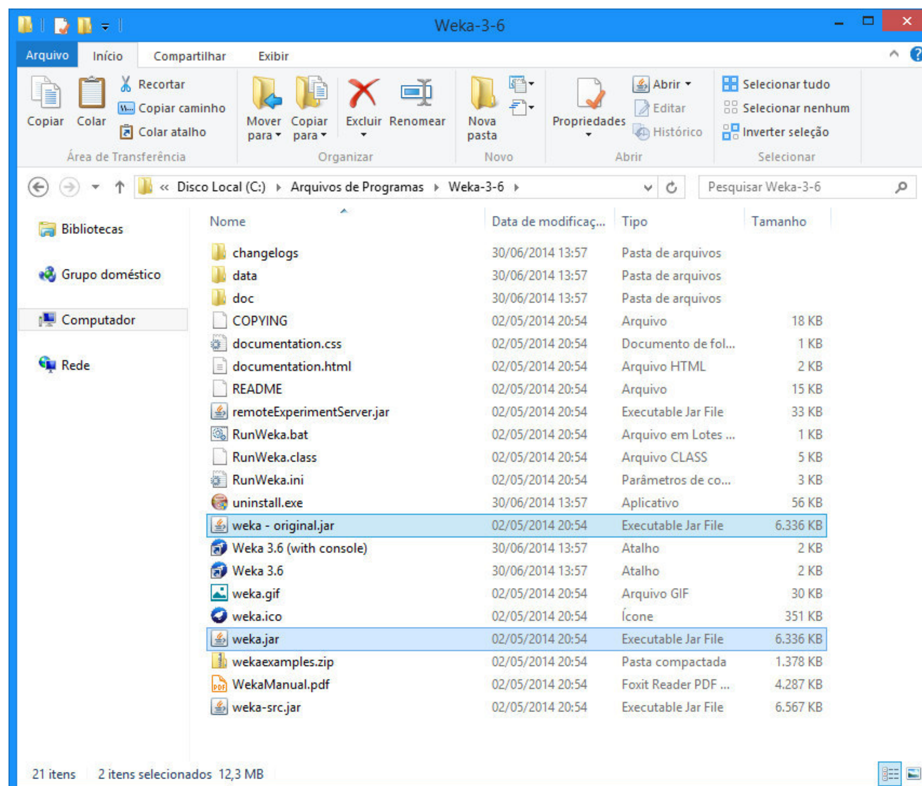
Com apenas este procedimento é possível executar o WEKA modificado normalmente como se ele fosse o arquivo originalmente instalado, bem como é possível desfazer o processo e

Figura 3.31: Exportando o projeto.



Fonte: autoria própria.

Figura 3.32: Substituindo instalação original do WEKA.



Fonte: autoria própria.

facilmente recuperar a versão original do WEKA.

4 LAD-WEKA

Este capítulo representa a soma dos conhecimentos exibidos nos Capítulos 1.2 e 2.5. O classificador LAD foi implementado dentro do WEKA, dando origem a uma versão do WEKA que chamamos de LAD-WEKA. O LAD-WEKA não ganha crédito pela implementação da ferramenta WEKA como um todo. Por isso, o LAD-WEKA preserva a *interface* original do WEKA. Deste modo, espera-se que o LAD-WEKA possa ser facilmente aceito por usuários do pacote WEKA oficial, e o classificador LAD seja incorporado a uma versão futura do WEKA.

O LAD-WEKA foi desenvolvido para ser público e de código aberto. Toda a implementação foi estruturada em módulos, de modo que é possível identificar as fases do algoritmo LAD e adicionar ou modificar os códigos existentes de cada fase separadamente.

A saber, este capítulo descreve como configurar os parâmetros da implementação do algoritmo LAD dentro do WEKA, como as regras são exibidas ao usuário, além de realizar alguns testes de comparação com alguns algoritmos já existentes no WEKA. Todo o capítulo é um mini-tutorial que visa passar conhecimentos sobre o uso da ferramenta. Por sua vez, detalhes avançados sobre o uso da ferramenta ou o código do algoritmo descrito no Capítulo 1.2 não serão exibidos. Tais detalhes não fazem parte do escopo deste documento, mas serão assunto de outra publicação preparada em paralelo a este trabalho.

4.1 AMBIENTE WEKA

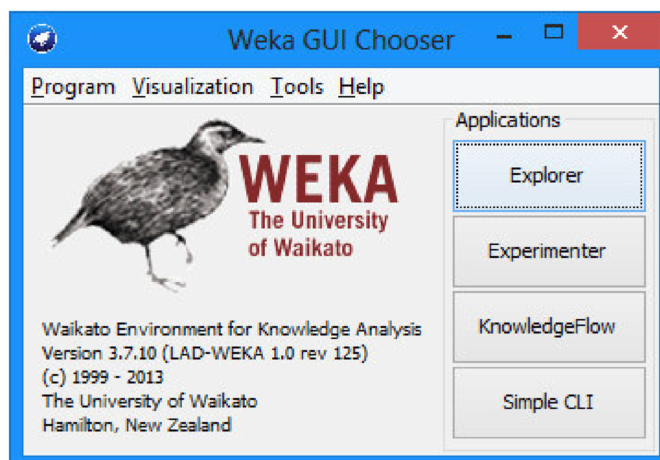
Antes de estudar os parâmetros do classificador LAD implementado, deve-se familiarizar com o módulo do WEKA em que os estudos serão realizados. O WEKA possui quatro módulos: *Explore*, *Experimenter*, *KnowledgeFlow* e *SimpleCLI*. Estes módulos podem ser acessados pelos botões da janela inicial do WEKA, observado na Figura 4.33. Dentre os módulos citados, somente o módulo *Explorer* será descrito com um pouco mais de detalhes. O objetivo é mostrar como a ferramenta funciona, e exemplificar o acesso e a configuração do classificador LAD.

A janela inicial do LAD-WEKA (veja a Figura 4.33) apresenta uma menção à versão do LAD-WEKA implementado e da revisão do repositório de controle de versão do código. Tais modificações são apenas estéticas, não influenciam no funcionamento do programa.

Todo o código interno do LAD-WEKA está em inglês, assim como a sua *interface* externa. Esta é uma medida considerada necessária para que o LAD-WEKA possa ser mais facilmente aceito entre os usuários do WEKA, e porque se espera que a versão implementada do classificador LAD torne-se parte do pacote oficial de classificadores em um futuro próximo.

As próximas subseções mostram como selecionar um classificador e acessar a sua janela de configuração de parâmetro usando o módulo *Explorer*.

Figura 4.33: Janela Weka GUI Chooser.



Fonte: autoria própria.

4.1.1 Módulo *Explorer*

A partir do módulo *Explorer*, visto na Figura 4.34, pode-se acessar muitos dos algoritmos do WEKA, incluindo os algoritmos de classificação (dentre os quais, encontra-se o LAD). O módulo necessita que um conjunto de dados seja carregado na aba *Preprocess* para que as demais abas tornem-se ativas. O botão *Open file* permite abrir uma janela de exploração para a busca do conjunto de dados (preferencialmente a janela procura por arquivos do tipo ARFF, mas é possível abrir arquivos em vários formatos).

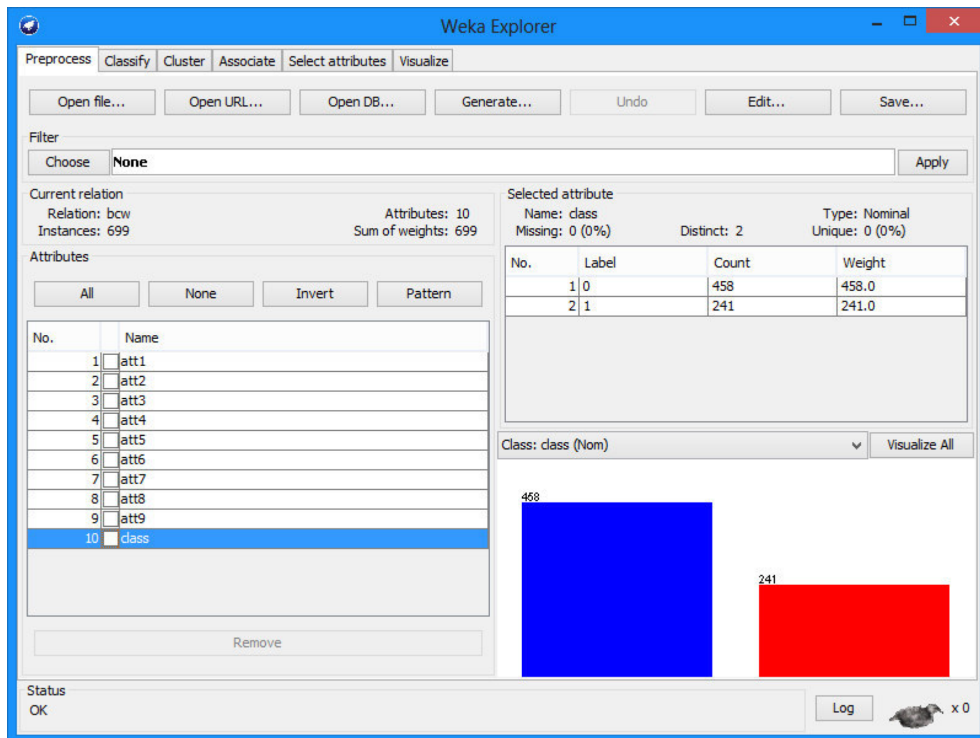
Para o exemplo deste capítulo, o conjunto de dados BCW (*Breast Cancer Wisconsin*) foi utilizado (Wolberg et al., 1995). O conjunto de dados foi transformado em ARFF. Todas as instâncias que possuíam valores faltantes foram excluídas. O conjunto de dados utilizado possui 684 instâncias e dez atributos (contando com o valor de classe). Este conjunto de dados possui somente duas classes. Ao carregar o arquivo, a janela exibe na tela todas as informações do conjunto de dados carregado e as outras abas podem ser acessadas, como mostrado na Figura 4.34.

4.1.2 Selecionando um classificador

Os classificadores podem ser acessados pela aba *Classify* (veja a Figura 4.35). Esta aba permite a execução de um algoritmo de classificação por vez, e também permite escolher entre algumas modalidades de teste e de treinamento. Para acessar os classificadores basta clicar no botão *Choose*, como mostrado na Figura 4.35. O classificador LAD foi configurado para aparecer dentro da pasta *rules* ("regras", em português) uma vez que este é um classificador baseado em regras de decisão.

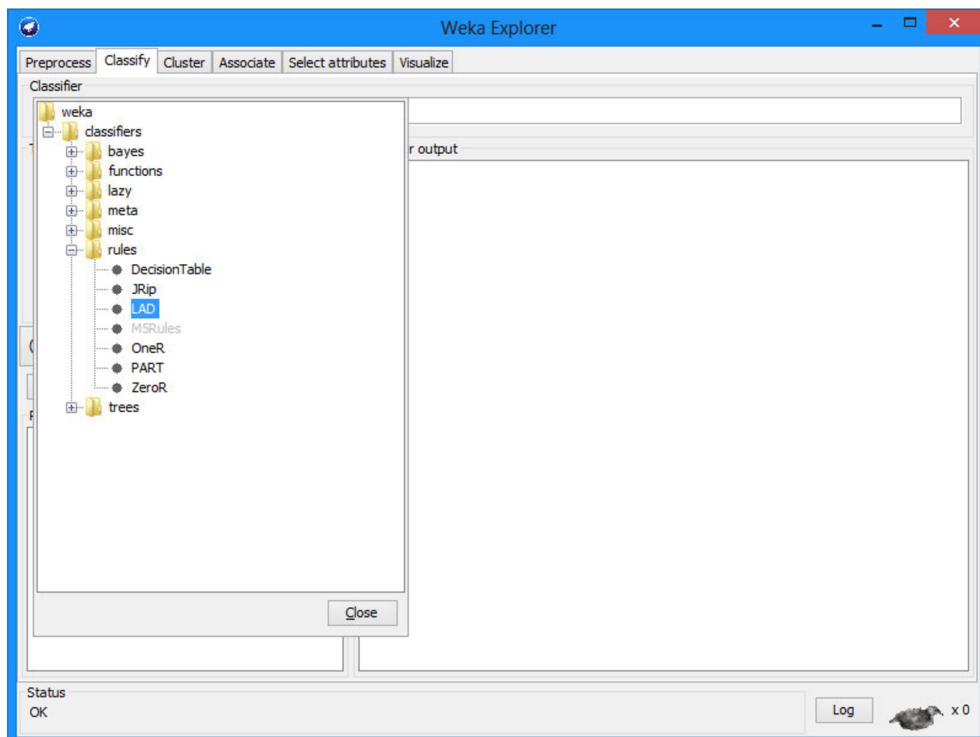
Depois de selecionado o algoritmo de classificação, o WEKA modifica um pouco a sua *interface*, como mostra a Figura 4.36. Ao lado do botão *Chooser* aparece uma série de *tokens* e parâmetros de configuração do LAD. Este mesmo código pode ser usado quando se deseja uti-

Figura 4.34: Janela Weka Explorer - BCW.



Fonte: autoria própria.

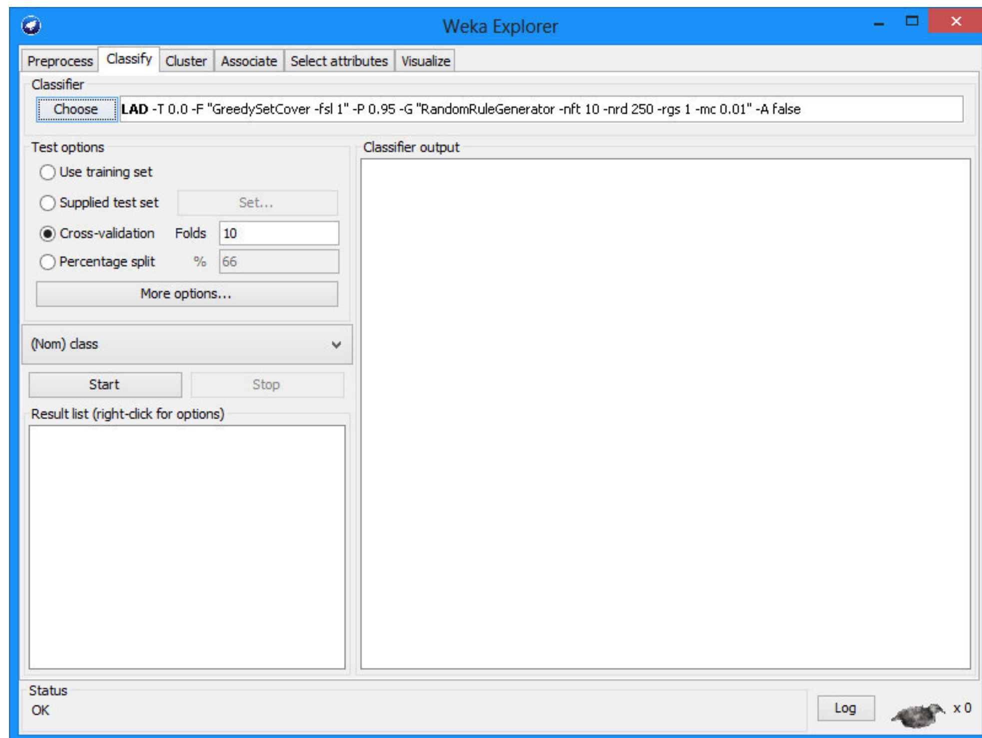
Figura 4.35: Janela *Weka Explorer* - Aba *Classify*.



Fonte: autoria própria.

lizar o algoritmo a partir do módulo *SimpleCLI*. Este módulo permite acessar os classificadores por linhas de comandos.

Figura 4.36: Aba *Classify* - Classificador LAD.



Fonte: autoria própria.

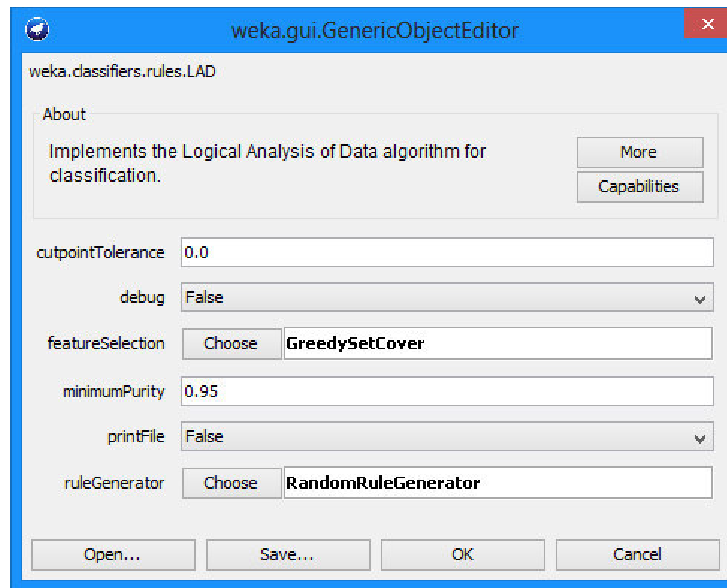
4.1.3 Acessando as configurações de um classificador

Para se configurar qualquer dos classificadores do WEKA basta clicar sobre a linha onde se exibe o nome do classificador e seus *tokens* de configuração. Para o LAD, a janela exibida na Figura 4.37 aparece na tela. Esta é uma janela de configuração frequentemente vista quando se trabalha com os classificadores do WEKA.

Dentre os parâmetros exibidos na Figura 4.37, somente o parâmetro `debug` não foi explicitamente adicionado pelo classificador LAD. Este parâmetro pertence à classe `Abstract Classifier`, discutida no Capítulo 2.5. O restante dos parâmetros serão discutidos nas subseções seguintes.

Antes de prosseguir, deve-se saber que não existe um controle quanto à exibição dos parâmetros pela *interface* do WEKA, nem quanto ao formato em que estes são escritos (sem espaços) nem quanto à sequência em que eles aparecem na janela de configuração. Os nomes dos parâmetros são definidos pelas funções **get** e **set** de cada parâmetro e aparecem em ordem alfabética. Nas próximas subseções, exibem-se os parâmetros em ordem lógica das etapas do algoritmo LAD, e não na sequência exibida na figura.

Figura 4.37: Janela de configuração do LAD.



Fonte: autoria própria.

4.2 CONFIGURANDO O LAD

4.2.1 *Cutpoint Tolerance*

O primeiro parâmetro em ordem lógica das etapas do algoritmo é o **Cutpoint Tolerance** ("Tolerância do Ponto de Corte", em português). Este parâmetro é usado como forma de diminuir o número de pontos de cortes gerados na binarização.

Na formação do ponto de corte, este é definido quando dois valores consecutivos A_a e A_b do vetor ordenado de valores de um atributo formam uma transição de classes. Mais especificamente, o valor do ponto de corte é calculado como na Equação 2.2.

A tolerância do ponto de corte verifica se os dois valores consecutivos que deveriam gerar um ponto de corte estão a uma **distância tolerável**, isto é, se

$$|X_{ia} - X_{ib}| \geq t, \tag{4.22}$$

onde t é a tolerância do ponto de corte. Assim, se esta diferença de valores não passar no teste o ponto de corte não será gerado. O valor padrão deste parâmetro é $t = 0,0$. Este valor faz com que o teste seja sempre verdadeiro, observando que o vetor é ordenado e não possui valores repetidos.

O bom uso deste parâmetro pode facilitar as outras fases do algoritmo, como a seleção de características e a busca de regras, visto que a tolerância pode fazer com que se gere um número menor de atributos binários, acelerando a execução. Contudo, este atributo é aplicado com o mesmo valor para o teste de todos os atributos. Assim, se um atributo tiver distâncias em magnitudes maiores, a tolerância pode não agir como se espera.

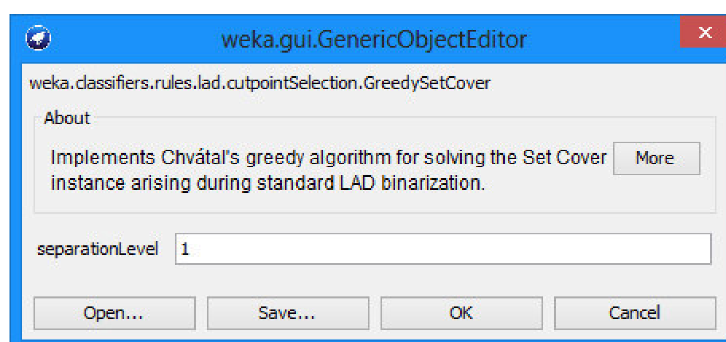
Para o uso correto deste parâmetro é bom que todos os atributos estejam devidamente normalizados. Normalizar os dados é um processo comum em mineração de dados e significa colocar os dados em uma faixa de valores específica. Digamos que um determinado atributo tem valores que vão de -1000 até 999. A normalização pode mapear os valores para uma faixa de, por exemplo, 0,0 até 1,0. Este processo pode fazer com que as possíveis distâncias entre os valores dos vetores ordenados estejam em uma magnitude similar.

A normalização é uma etapa de preprocessamento de dados, não cabe ao algoritmo LAD realizar este passo.

4.2.2 *Feature Selection*

O parâmetro **Feature Selection** ("Seleção de Características", em português), podendo ser chamado de **Attribute Selection** ("Seleção de Atributos", em português), controla uma pequena variação da seleção de atributos vista na Seção 2.3. Para acessar as configurações da seleção de atributos basta clicar sobre o nome **Greedy Set Cover**. Uma janela como a exibida na Figura 4.38 aparecerá.

Figura 4.38: Janela de configuração da Seleção de Atributos.



Fonte: autoria própria.

Na variação do modelo da Seção 2.3, o algoritmo seleciona o conjunto mais atrativo, mas os elementos pertencentes àquele conjunto não são excluídos a princípio (como aconteceria no modelo original). A diferença é que os elementos têm que ser "cobertos" (ter sido parte de um conjunto selecionado) ao menos k vezes antes de serem desconsiderados. Assim os elementos podem influenciar no custo-benefício de outros conjuntos. Por sua vez, k é representado na *interface* do WEKA como sendo *Separation Level*, como pode ser visto na figura.

O valor padrão da variável *Separation Level* é 1. Desta maneira, os elementos só precisam ser cobertos uma única vez para serem desconsiderados. Isto reflete o funcionamento original visto na Seção 2.3.

4.2.3 *Minimum Purity*

O parâmetro **Minimum Purity** ("Pureza Mínima", em português), representa o valor mínimo de pureza das regras a serem geradas. Como visto, o cálculo da pureza de uma regra varia de

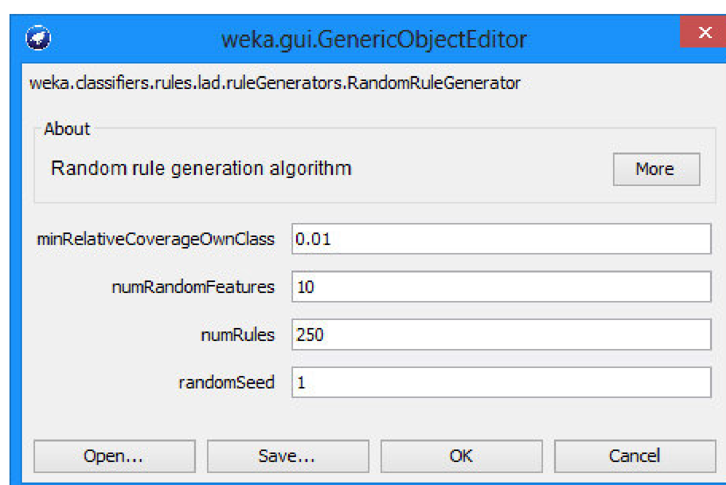
acordo com sua classe. Para uma regra de classe positiva, a pureza é a razão entre o número de instâncias de classe positiva cobertas pela regra, pelo número total de instâncias (isto é, positivas e negativas) cobertas por esta regra.

Este parâmetro é um limiar, um limite inferior. Para que uma regra seja adicionada à lista de regras do classificador, esta regra deve possuir pureza maior ou igual ao valor do parâmetro de pureza mínima.

4.2.4 *Rule Generator*

O parâmetro **Rule Generator** ("Gerador de Regras", em português) dá acesso ao controle dos parâmetros de configuração do gerador de regras *Random Rule Generator* (RRG). Para acessar as configurações do gerador de regras basta clicar sobre o nome **Random Rule Generator**. Uma janela como a exibida na Figura 4.39 aparecerá.

Figura 4.39: Janela de configuração do RRG.



Fonte: autoria própria.

Para que se entenda melhor sobre cada parâmetro, esta subseção será subdividida em quatro pequenas partes, de forma a explanar cada um dos parâmetros do gerador de regra separadamente. Os conceitos já foram discutidos no Capítulo 1.2, então serão apenas relembrados.

Novamente, os parâmetros não são controlados quanto à sequência de exibição. Nas subseções seguintes, os parâmetros serão discutidos em ordem lógica e não em ordem alfabética.

4.2.4.1 *Number of Rules*

O parâmetro **Number of Rules** ("Número de Regras", em português) permite que o usuário escolha o número de vezes que o algoritmo de geração de regras tentará gerar um conjunto de regra de cada classe. O usuário pode tentar gerar menos ou mais regras, dependendo dos seus motivos. Por exemplo, o usuário pode querer gerar menos regras para que o processamento do classificador seja mais rápido. O usuário também pode querer gerar menos regras porque considera o conjunto de dados fácil de aprender.

4.2.4.2 *Number of Features*

O parâmetro **Number of Features** ("Número de Características/Atributos", em português) especifica o número de atributos binários amostrados a cada iteração interna da geração de uma única regra. Este é o parâmetro f, visto na Figura 2.13.

4.2.4.3 *Minimum Relative Coverage of Own Class*

O parâmetro **Minimum Relative Coverage of Own Class** ("Cobertura Mínima da Própria Classe", em português), assim como a pureza mínima, é um limiar que determina se uma regra será ou não adicionada à lista de regras do gerador de regras.

A cobertura relativa mínima é a razão do número de instâncias que uma regra cobre de sua própria classe pelo número total de instâncias do conjunto de dados que são da mesma classe da regra.

Diferente da pureza mínima a cobertura relativa sempre usa todas as instâncias do conjunto de dados em seu cálculo. Outra característica da cobertura relativa é que, ao contrário da pureza, a cobertura relativa tende a decrescer, já que, na montagem da regra, uma regra não volta a cobrir uma instância que já foi descoberta.

4.2.4.4 *Random Seed*

O parâmetro **Random Seed** ("Semente Aleatória", em português) serve simplesmente como o valor da semente inicial do gerador de regras. Uma vez que este gerador de regras é aleatório, esta semente é necessária. A semente também é usada para que o usuário possa conduzir testes controlados. O usuário pode repetir resultados obtidos anteriormente apenas mantendo a mesma configuração.

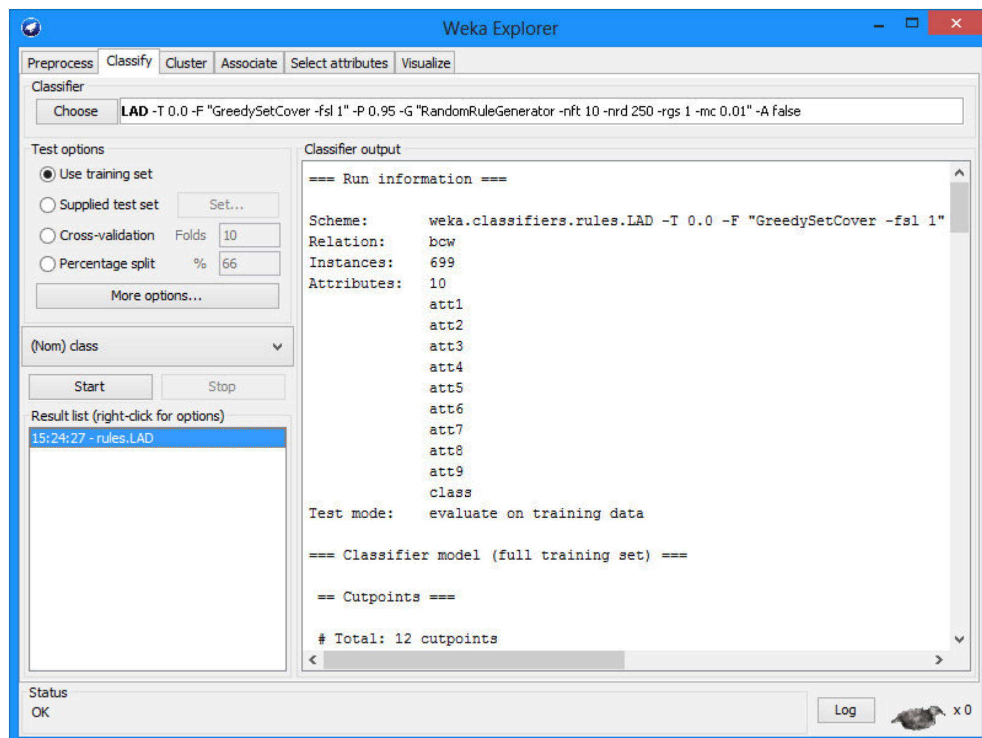
4.2.5 *Print File*

O parâmetro **Print File** é um booleano que faz com que o classificador gere um arquivo de saída com informações sobre seu aprendizado. Neste arquivo encontram-se as informações do conjunto de dados utilizado no treinamento, assim como as informações das configurações do algoritmo que foram utilizadas para o treinamento. O arquivo ainda exibe os pontos de corte e as regras geradas, além de informações sobre quais regras cobriram cada uma das instâncias do conjunto de testes e a classificação final do classificador para cada uma das instâncias. Embora o WEKA permita salvar a saída do treinamento (isto é, a saída exibida para qualquer dos classificadores, vista na Figura 4.40), este arquivo possui informações que não são exibidas na saída do WEKA (por exemplo, quais regras foram satisfeitas por cada uma das instâncias).

4.3 SAÍDA DO TREINAMENTO

A aba *Classify* exibe os resultados da execução de um classificador no quadro **Classifier output**, mostrado na Figura 4.40. O WEKA exibe a saída do classificador dentro de uma seção (no texto de saída) chamada **Classifier model**, possível de se visualizar ainda na Figura 4.40. Dentro desta seção, esta implementação do LAD exibe dois tipos de informações de saída. São estas, o número de pontos de corte e as regras geradas no treinamento.

Figura 4.40: Saída do LAD.



Fonte: autoria própria.

O número de pontos de corte exibido não representa o número de pontos de corte gerados na binarização. Este valor é, na verdade, o número final de pontos de cortes após a seleção de atributos.

As regras são escritas em seu formato numérico, como foi discutido anteriormente no Capítulo 1.2. Estas são mostradas na subseção *Summary of Patterns*, separadas por classe. Embora as regras possuam pesos que influenciam no processo de classificação das instâncias, os valores dos pesos não são exibidos na saída do classificador por motivos estéticos. A Figura 4.41 exibe o trecho de um exemplo saída do classificador LAD, mostrando o início das regras denotadas pela classe 0.

Figura 4.41: Exemplo de saída do classificador LAD.

```
=== Classifier model (full training set) ===

 == Cutpoints ===

 # Total: 12 cutpoints

 == Summary of Patterns ===

 @ Class "0" patterns:

 [att1 <= 3.5] [att6 <= 3.5]
 [att1 <= 3.5] [att3 <= 3.5] [att6 <= 3.5]
 [att2 <= 2.5]
 [att1 <= 3.5] [att2 <= 2.5]
 [att2 <= 2.5] [att4 <= 1.5]
 [att2 <= 2.5] [att4 <= 1.5] [att7 <= 4.5]
 [att2 <= 2.5] [att4 <= 1.5] [att6 <= 3.5] [att7 <= 4.5]
 [att3 <= 3.5] [att6 <= 3.5]
 [att3 <= 3.5] [att6 <= 3.5] [att8 <= 2.5]
 [att1 <= 6.5] [att3 <= 3.5] [att6 <= 3.5] [att8 <= 2.5]
```

Fonte: autoria própria.

4.4 TESTES DE COMPARAÇÃO

A fim de mostrar resultados da aplicação do algoritmo implementado, o classificador LAD foi testado sobre três conjuntos de dados. Os conjuntos de dados testados são públicos, e foram obtidos no repositório de dados da Universidade de Irvine (Frank e Asuncion, 2010). A saber, os conjuntos de dados são: *Breast Cancer Wisconsin*, *Heart Disease* (Janosi et al., 1988) e *Haberman's Survival* (Lim, 1999). O conjunto de dados *Breast Cancer Wisconsin* foi modificado nos moldes descritos na Seção 4.1.1, isto é as instâncias com valores faltantes foram removidas. Para o restante dos conjuntos de dados, nada foi modificado.

Além do LAD, outros três algoritmos foram usados nos testes. Foram estes: C4.5 (J48), *Support Machine Vector* (SVM) e *Multilayer Perceptrons* (MP). Estes algoritmos são bastante conhecidos e são implementações estáveis do pacote WEKA.

Os testes foram realizados com o uso do módulo *Explorer* do WEKA. A Figura 4.42 exibe os resultados dos testes. Cada entrada da tabela reflete a precisão média de 10 experimentos *10-fold cross validation* (Kohavi, 1995) aplicados a um classificador. Cada experimento toma um conjunto de dados e o divide em 10 partes de tamanhos iguais (cada parte é chamada de *fold* ou dobra). O experimento consiste em selecionar, por exemplo, as nove primeiras partes do conjunto de dados para serem usadas como conjunto de treinamento e a parte restante para ser usada como conjunto de teste. O processo é repetido 10 vezes, cada vez com uma partição é tomada aleatoriamente para ser usada como o conjunto de treinamento. Assim, cada um dos valores da tabela refletem a média de precisão de 100 testes.

Na leitura dos resultados, percebe-se que o classificador LAD implementado mantém-se competitivo, em relação aos demais algoritmos utilizados nos testes. Ao mesmo tempo, não se

Figura 4.42: Testes comparativos.

Algoritmos	Breast Cancer Wisc.	Heart Disease	Haberman's Survival
LAD	96,35 %	81,81 %	74,19 %
J48	95,19 %	78,15 %	74,05 %
MP	96,72 %	79,41 %	73,87 %
SVM	96,87 %	83,89 %	73,40 %

Fonte: autoria própria.

exigiu um ajuste intenso dos parâmetros para que obtivéssemos tais resultados, o que mostra a robustez do algoritmo. Como um último detalhe, pelo fato do algoritmo estar implementado dentro do próprio WEKA, os testes foram mais facilmente aplicados.

5 CONCLUSÕES

Algoritmos de aprendizado de máquina são algoritmos usados para extrair ou evidenciar padrões, e tendências que se repetem, em grandes conjuntos de dados. Este conhecimento pode ser apresentado de diversas formas como, por exemplo, hipóteses, regras, árvores de decisão. Algoritmos de classificação supervisionada como o LAD tentam inferir funções que separam os grupos de elementos contidos nos conjuntos de dados. Em especial, o algoritmo LAD usa um conjunto de regras para expressar tal função.

Softwares que disponibilizam algoritmos de aprendizado de máquina são frequentemente usados em pesquisas. Tais ferramentas tornam-se atrativas devido à quantidade de algoritmos disponíveis e por tais algoritmos serem bastante estáveis, pois geralmente a ferramenta é gerenciada por um grupo experiente. Em especial, podemos destacar o WEKA. Esta ferramenta oferece, além de um conjunto de algoritmos de aprendizado de máquina, facilidade de uso, métodos de experimentação e comparação entre algoritmos. O WEKA é conhecido mundialmente e possui uma grande quantidade de usuários.

Neste trabalho, uma versão do algoritmo LAD foi descrita, assim como os seus principais conceitos. É, portanto, necessário que se entenda qual é o verdadeiro impacto da descrição de tal algoritmo. Primeiro, é importante saber que não existem descrições detalhadas em português sobre o algoritmo LAD. Segundo, a versão do LAD descrita neste trabalho possui um novo procedimento de geração de regras que exibe resultados competitivos quando comparados a outros algoritmos do estado-da-arte. Além disso, a versão de geração de regras foi descrita pela primeira vez neste trabalho.

Também, neste trabalho, foi mostrado como inserir um novo classificador dentro do WEKA. Estes conhecimentos não existem, em tal nível de detalhamento, em inglês ou em português. Ainda, como foi descrito nos objetivos, um dos produtos deste trabalho é a disponibilização de uma versão do WEKA contendo o classificador LAD (versão que leva o nome de LAD-WEKA).

É necessário ressaltar alguns pontos sobre o LAD-WEKA. Primeiro, o LAD-WEKA possui uma versão estável do LAD e está disponível àqueles interessados no uso do classificador. Isto é interessante, uma vez que, até então, não existia uma versão pública do algoritmo. Além disso, o classificador, por estar dentro do WEKA, usufrui das várias vantagens da ferramenta. Segundo, o LAD ainda não faz parte da versão oficial do WEKA. Assim, espera-se que, a partir da distribuição do LAD-WEKA, do uso e da menção da ferramenta em pesquisas, e da publicação da ferramenta, o LAD venha a ser parte da lista de classificadores da versão oficial do WEKA.

Ao fim, todos os objetivos traçados neste trabalho foram atingidos.

REFERÊNCIAS

GNU (2007). General Public License, version 3. Disponível em: http://www.gnu.org/licenses/gpl.html. Acesso em: 10 Jun. 2014.

ALEXE, G. et al. (2006). Breast cancer prognosis by combinatorial analysis of gene expression data. *Breast Cancer Res.*, 8(8):R41

BONATES, T.O. et al. (2008). Maximum Patterns in Datasets. *Discrete Applied Mathematics*, 156(6):846–861.

BOROS, E. et al. (1997). Logical analysis of numerical data. *Mathematical Programming*, 79(1):163—190.

BOROS, E. et al. (2000). An implementation of logical analysis of data. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):292—306.

BREIMAN, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

CHVÁTAL, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235.

DEITTERICH, T. (1995). Overfitting and Undercomputing in Machine Learing. *ACM Computing Survey*, 27(3):326–327.

ECLIPSE (2012). Eclipse 4.2.1. Disponível em: http://www.eclipse.org. Acesso em: 12 Jun. 2014.

FRANK A.; ASUNCION A. (2010). UCI machine learning repository. Disponível em: http://archive.ics.uci.edu/ml. Acesso em: 12 Jun. 2014.

GOMES, V.S.D.; BONATES, T.O. (2011). Classificação supervisionada de dados via otimização e funções booleanas. *II Workshop Técnico-Científico de Computação*.

HALL, M. et al. (2009). The WEKA data mining software: An update. 11.

HALL, M. et al. (2013). Weka software 3.7.10. Disponível em: http://www.cs.waikato.ac.nz/ml/weka/. Acesso em: 12 Jun. 2014.

HAMMER, A. et al. (1999). Logical analysis of chinese labor productivity patterns. *Annals of Operations Research*, 87(0):165–176.

HAMMER, P. et al. (2006). Modeling country risk ratings using partial orders. *European Journal of Operational Research*, 175(2):836–859.

HO, T. (1998). The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844.

HOLLAND, J. H. (1992). Adaptation in natural and artificial systems.

JANOSI, A. (1988). Heart's disease, UCI machine learning repository. Disponível em: https://archive.ics.uci.edu/ml/datasets/Heart+Disease. Acesso em: 20 Jun. 2014.

KOHAVI, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. 1137–1143.

ORACLE (2013). Java 1.7.0 17. Disponível em: http://www.oracle.com/technetwork/java/javase/downloads/index.html. Acesso em 12 Jun. 2014.

RON, D. (2010). Algorithmic and analysis techniques in property testing. *Foundations and Trends in Theoretical Computer Science*, 5(2):73–205.

SANTOS, R. (2005). Weka na munheca - um guia para uso do WEKA em scripts e integração com aplicações em Java.

TJEN-SIEN LIM (1999). Haberman's suvival dataset, UCI machine learning repository, university of california, irvine, school of information and computer sciences. Disponível em: https://archive.ics.uci.edu/ml/datasets/Haberman's+Survival. Acesso em: 20 Jun. 2014.

WEKA WIKISPACES (2014a). Use Weka in your Java code. Disponível em: http://weka.wikispaces.com/Use+Weka+in+your+Java+code. Acesso em: 12 Jun 2014.

WEKA WIKISPACES (2014b). Use Weka in your Java code. Disponível em: http://weka.wikispaces.com/. Acesso em: 25 Mai 2014.

WITTEN, I.H. et al. (2005). Data Mining: Practical machine learning tools and techniques. *Morgan Kaufmann Publishers*.

WOLBERG, W.H. et al. (1995). Breast cancer wisconsin, UCI machine learning repository, university of california, irvine, school of information and computer sciences. Disponível em: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). Acesso em: 12 Jun 2014.