Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

Descrição da Implementação dos Métodos Estatísticos de Resampling do Ambiente Discovery

André Henrique Kemp

Gustavo Enrique de Almeida Prado Alves Batista

Maria Carolina Monard

N. 143

RELATÓRIOS TÉCNICOS DO ICMC

USP – São Carlos
Maio de 2001

Descrição da Implementação dos Métodos Estatísticos de Resampling do Ambiente Discovery[1]

André Henrique Kemp
Gustavo Enrique de Almeida Prado Alves Batista
Maria Carolina Monard

Universidade de São Paulo/ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP
e-mail:{akemp, gbatista, mcmonard}@icmc.sc.usp.br

Sumário

Neste trabalho nos concentramos em uma tarefa de aprendizado de máquina conhecida como classificação ou predição, cujo problema consiste na construção de um procedimento de classificação a partir de um conjunto de casos nos quais as classes verdadeiras são conhecidas, chamado de aprendizado supervisionado. O maior objetivo de um classificador é ser capaz de predizer com sucesso a respeito de novos casos. O desempenho de um classificador pode ser medido em termos da taxa de erro. Técnicas experimentais para estimar a taxa de erro para casos não vistos não somente provêm uma base para comparar objetivamente as performances de diversos algoritmos de aprendizado no mesmo conjunto de exemplos, mas também podem ser uma ferramenta poderosa para projetar um classificador. As técnicas para estimar a taxa de erro são baseados na teoria estatística de resampling.

Um ambiente chamado *Discover* está sendo implementado para ajudar na aplicação dos métodos de resampling em conjuntos de exemplos do mundo real. Foi projetado como uma coleção de programas independentes, os quais podem interagir entre si através de scripts pré-definidos ou de novos scripts criados pelo usuário. Inclui ferramentas para particionar conjuntos de exemplos em conjuntos de treinamento e teste utilizando diferentes métodos de resampling. Além do método holdout, que é o estimador de taxa de erro mais comum, suporta os métodos n-fold cross-validation — incluindo o leaning-one-out — e o método bootstrap.

O objetivo deste trabalho é fornecer um panorama geral do funcionamento do ambiente, e descrever a implementação dos métodos estatísticos de resampling pertencentes a esse ambiente.

Maio de 2001

[1]Trabalho realizado com auxílio da FAPESP.

Sumário

1 Introdução **1**

2 Sistemas de Aprendizado e Classificadores **1**

2.1 O Modelo de Classificação 2

2.2 Paradigmas de Aprendizado por Exemplos 3

2.2.1 Paradigma Simbólico 4

2.2.2 Paradigma Estatístico 4

2.2.3 Paradigma *Instance-based* 4

2.2.4 Paradigma Conexionista 5

3 Estimando o Desempenho dos Sistemas de Aprendizado **5**

4 Tipos de Erros **6**

5 Técnicas para Estimar a Taxa de Erro Verdadeira **7**

5.1 Estimativa da Taxa de Erro Através de Treinar-e-Testar 8

5.2 Métodos de *Resampling* 9

5.2.1 Random Subsampling 10

5.2.2 Cross-validation 10

5.2.3 Bootstraping 11

6 Sistemas de Aprendizado Simbólicos **12**

6.1 O Base de Dados *Iris* 13

6.2 O Sistema de Aprendizado C4.5 14

6.2.1 Principais Parâmetros 14

6.2.2 Arquivos 15

6.2.3 Sintaxe dos Arquivos de Entrada 15

6.2.4 Sintaxe dos Arquivos de Saída 17

6.3 O Sistema de Aprendizado C4.5rules 17

6.3.1 Principais Parâmetros 18

6.3.2 Arquivos 18

6.3.3 Sintaxe dos Arquivos de Entrada 18
6.3.4 Sintaxe dos Arquivos de Saída 18
6.4 O Sistema de Aprendizado C5.0 20
6.4.1 Principais Parâmetros 20
6.4.2 Arquivos 21
6.4.3 Sintaxe dos Arquivos de Entrada 22
6.4.4 Sintaxe dos Arquivos de Saída 22
6.5 O Sistema de Aprendizado CN2 22
6.5.1 Principais Parâmetros 24
6.5.2 Arquivos 24
6.5.3 Sintaxe dos Arquivos de Entrada 25
6.5.4 Sintaxe dos Arquivos de Saída 26

7 Projeto e Implementação de Alguns Métodos de Resampling 29
7.1 O programa Mix (*mix*) 29
7.2 O programa Holdout (*holdout*) 30
7.3 O programa Random Subsampling (*rsampling*) 32
7.4 O programa Cross Validation (*kfold*) 32

A Programas Fontes 35
A.1 *mix* 35
A.2 *holdout* 41
A.3 *rsampling* 43
A.4 *kfold* 45

Lista de Figuras

1 Sistema de Aprendizado 1
2 Sistema de Classificação 2
3 Representação da Classificação 3
4 Conjuntos de treinamento e teste 8
5 Número de Casos de Teste e Qualidade da Predição 9

6 Arquivo Iris.names 16
7 Parte do arquivo Iris.data 16
8 Árvore de decisão induzida pelo $C4.5$ para a base *Iris* 17
9 Conjunto de regras induzidas pelo $C4.5$rules para a base *Iris* 19
10 Árvore de decisão e conjunto de regras induzidos pelo C5.0 para a base *Iris* 23
11 Arquivo Iris.att 25
12 Parte do arquivo Iris.exs 26
13 Conjunto de regras ordenadas 27
14 Conjunto de regras não ordenadas 28
15 Algoritmo do programa `mix` 30
16 Algoritmo do programa `holdout` 31
17 Algoritmo do programa `rsamp` 32
18 Algoritmo do programa `kfold` 33

Lista de Tabelas

1 Exemplo de Matriz de Confusão para Três Classes 6
2 Tipos de Erros para Classificação com Duas Classes 7
3 Comparação entre Holdout e Random Subsampling 10
4 Comparação entre Métodos de Cross-validation 11
5 Características dos Estimadores Bootstrap 12

1 Introdução

Este relatório técnico tem como objetivo iniciar um estudo sobre Aprendizado de Máquina, com a finalidade de criar um ambiente de avaliação de performance. Neste estudo, são abordados os principais conceitos necessários para o desenvolvimento desse ambiente.

O estudo inicia com a definição de sistemas de aprendizado e os classificadores gerados por esses sistemas. Ainda, são abordados os principais paradigmas de aprendizado indutivo (Seção 2). Logo em seguida, é apresentada uma das medidas mais importantes para estimar o desempenho de sistemas de aprendizado, a *taxa de erro* (Seção 3). A taxa de erro possui a limitação de considerar todos os erros como igualmente importantes. Contudo, na maioria das aplicações reais, é necessário atribuir pesos para os diferentes erros cometidos (Seção 4). Para estimar uma taxa de erro não tendenciosa de um sistema de aprendizado é necessário utilizar métodos estatísticos de *resampling* (Seção 5).

O estudo continua com uma descrição do funcionamento de alguns dos principais algoritmos de Aprendizado de Máquina Simbólico (Seção 6). A descrição do funcionamento envolve identificar, para cada um dos sistemas de aprendizado, os principais parâmetros, os arquivos utilizados como entrada e saída e as sintaxes desses arquivos.

A seguir, o estudo descreve a implementação e o funcionamento dos algoritmos de *resampling* implementados neste projeto (Seção 7), terminando (Apêndice A) com os códigos fonte resultantes da implementação dos algoritmos na linguagem de programação PERL [Till, 1996].

2 Sistemas de Aprendizado e Classificadores

Sistema de aprendizado é um programa de computador que toma decisões baseado em experiências acumuladas contidas em casos resolvidos com sucesso [Weiss and Kulikowski, 1991]. Como ilustrado na Figura 1, o objetivo fundamental de um sistema de aprendizado é extrair conhecimento na forma de, por exemplo, regras de decisão, de um conjunto de dados de modo que essas regras possam ser aplicadas a novos dados. Um sistema de aprendizado típico é projetado para trabalhar sob algum paradigma de aprendizado, tal como o paradigma simbólico, estatístico, conexionista, etc. "Aprender" consiste em escolher ou adaptar parâmetros da representação do modelo, no paradigma escolhido, de forma que o modelo classifique corretamente os exemplos conhecidos bem como novos exemplos.

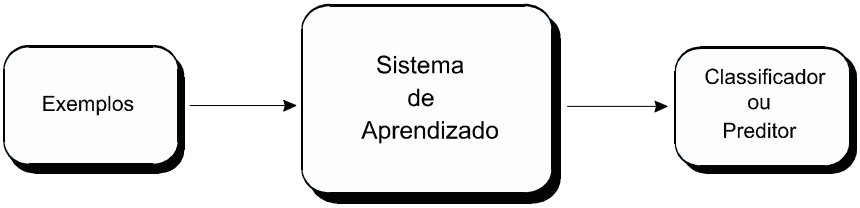

Figura 1: Sistema de Aprendizado

A grosso modo, classificação pode ser resumida em encontrar uma função que mapeie objetos

com características comuns, na mesma classe. O Aprendizado de Máquina pode possuir duas conotações diferentes:

1. Dado um conjunto de observações, tentar estabelecer a existência de classes ou *clusters* nos dados;
2. Dado um conjunto de observações no qual as classes são conhecidas, o objetivo está em encontrar regras capazes de classificar novas observações entre as classes já existentes.

O primeiro tipo é conhecido como *Aprendizado Não Supervisionado* ou *Clustering*, o segundo é conhecido como *Aprendizado Supervisionado*.

2.1 O Modelo de Classificação

Para problemas de classificação, um sistema de aprendizado pode ser visto como um sistema de alto nível que ajuda a construir um sistema de tomada de decisões chamado de *classificador*. A forma mais simples de representar um classificador seria como uma caixa preta que produz uma decisão para cada padrão de dados que for apresentado a ela. A Figura 2 ilustra a estrutura simples de um sistema de classificação o qual aceita um padrão de dados como entrada e produz uma decisão como saída.

Figura 2: Sistema de Classificação

Um sistema de aprendizado tem disponível um conjunto finito de exemplos de casos resolvidos. Os dados de cada caso consistem em um padrão de observações e a classificação correta correspondente. O objetivo do sistema de aprendizado é moldar a estrutura do classificador para um problema específico, encontrando uma forma genérica de relatar qualquer padrão particular de observações em uma das classes especificadas.

A representação básica do problema de classificação é simples. Cada exemplo de um problema resolvido consiste de observações e a correta classe correspondente. Geralmente, uma simples classe será resultado de um padrão de classes observado. Porém, podem haver problemas nos quais múltiplas classes estão associadas a um padrão de observações, mas freqüentemente tais problemas podem ser decompostos em vários subproblemas envolvendo classes individuais.

Desta forma, o conjunto de exemplos contém os dados que o sistema de aprendizado irá usar para encontrar as regras de decisão generalizadas para o classificador. A Figura 3 ilustra os elementos representacionais de uma coleção de casos resolvidos, e a relação entre o sistema de aprendizado e o classificador. Cada observação, do conjunto de observações, potencialmente relevante à solução do problema é chamada de *atributo*. Atributos também são chamados de *features*, variáveis, testes e medidas.

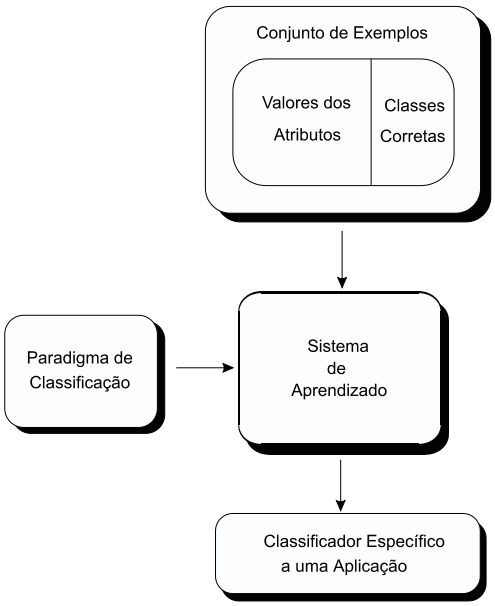

Figura 3: Representação da Classificação

Desta forma, o problema de classificação pode ser resumido da seguinte forma:

> dados exemplos compostos por n atributos, $x = x_1, \ldots, x_n$, e a classe associada a cada exemplo, y, a tarefa é encontrar uma função de mapeamento, f, tal que $y = f(x)$.

Vale lembrar que em problemas de classificação, a variável de saída y assume valores estritamente categóricos. Diferente da *regressão*, no qual a variável y assume valores contínuos. Desta perspectiva, classificação pode ser pensada como uma sub-categoria de regressão [Weiss and Indurkhya, 1995].

Como casos corretamente resolvidos serão utilizados na construção de um classificador específico, o padrão de valores de atributos para cada caso é associado com uma classificação correta. Dessa forma, aprender neste tipo de sistema pode ser visto como um processo de generalização das associações observadas empiricamente, mas sob restrições impostas pelo paradigma de aprendizado escolhido.

2.2 Paradigmas de Aprendizado por Exemplos

Dentro da área de aprendizado de máquina já foram propostos vários paradigmas, ou modelos de representação, diferentes. Um requisito básico para todos os que serão discutidos a seguir, é que o conhecimento a ser aprendido esteja na forma de exemplos e que cada exemplo esteja relacionado a uma classe corretamente classificada. Os paradigmas de aprendizado mais conhecidos serão brevemente apresentados a seguir.

2.2.1 Paradigma Simbólico

Os sistemas de aprendizado simbólico buscam aprender construindo representações simbólicas de um conceito observando exemplos e contra-exemplos desse conceito. As representações simbólicas estão tipicamente na forma de alguma expressão lógica, árvore de decisão, regras de produção ou rede semântica, enfim, uma representação que é diretamente compreensível ao ser humano.

Atualmente, entre as representações simbólicas mais estudadas estão as árvores e regras de decisão. Métodos de indução de árvores de decisão a partir de dados empíricos, conhecidos como *particionamento recursivo*, foram estudados por pesquisadores da área de Inteligência Artificial e Estatística. Os sistemas ID3 [Quinlan, 1986] e C4.5 [Quinlan, 1988] para indução de árvores de decisão tiveram uma importante contribuição sobre a pesquisa em Inteligência Artificial.

Os trabalhos com indução de regras de decisão surgiram com a simples tradução das árvores de decisão para regras, com a poda realizada sobre as regras; tal abordagem surgiu em [Quinlan, 1998, Quinlan, 1987]. Posteriormente, foram criados métodos que induziam regras diretamente a partir dos dados, um exemplo deste trabalho pode ser encontrado em [Michalski et al., 1986].

2.2.2 Paradigma Estatístico

Pesquisadores em estatística têm criado diversos métodos de classificação, muitos deles semelhantes aos métodos empregados em aprendizado de máquina. Por exemplo, o método CART [Breiman et al., 1984], para montar árvores de decisão, foi desenvolvido por estatísticos.

De maneira geral, as técnicas estatísticas tendem a focar tarefas em que os atributos têm valores contínuos ou ordinais. Muitos também são paramétricos, assumindo alguma forma de modelo, e então encontrando valores apropriados para os parâmetros do modelo a partir de dados. Por exemplo, um classificador linear assume que as classes podem ser expressas como combinação linear dos valores dos atributos, e então procura uma combinação linear particular que forneça a melhor aproximação sobre o conjunto de dados. Os classificadores estatísticos freqüentemente assumem que valores de atributos possuem distribuição normal, e então usam os dados fornecidos para determinar média, variância e co-variância da distribuição.

2.2.3 Paradigma *Instance-based*

Uma forma intuitiva que os seres humanos utilizam para classificar eventos é lembrar de um caso similar cuja resultado já conheçam e então assumir que o novo evento terá o mesmo resultado. Esta filosofia exemplifica os sistemas instance-based, que classificam casos nunca vistos através de casos similares conhecidos [Quinlan, 1988].

As principais características dos sistemas instance-based são:

- Quais casos de treinamento devem ser lembrados? Se todos os casos forem memorizados, o classificador pode se tornar lento e difícil de manusear. O ideal é reter casos típicos, que juntos resumam toda a informação importante; esta abordagem pode ser observada em livros médicos e legais.

- Como deve ser medida a similaridade entre os casos? Se todos os atributos forem contínuos, pode-se calcular a distância entre dois casos como a raiz quadrada da soma dos quadrados das diferenças dos atributos. Quando alguns atributos não são ordinais, essa interpretação de distância se torna mais problemática. Além do mais, se existem muitos atributos irrelevantes, dois casos similares podem aparentar serem muito diferentes pois eles podem possuir valores diferentes em atributos sem importância.

- Como um novo caso deve ser relacionado com casos armazenados? Existem duas alternativas, que são usar um simples caso armazenado o qual é o mais próximo do novo caso, ou usar vários casos levando-se em consideração os diferentes graus de similaridade entre cada caso.

2.2.4 Paradigma Conexionista

O maior representante deste paradigma são as *Redes Neurais*. Redes Neurais são construções matemáticas relativamente simples que foram inspiradas no modelo biológico do sistema nervoso. Sua representação envolve unidades altamente interconectadas, e desta forma o nome *conexionismo* é utilizado para descrever a área de estudo.

A metáfora com as conexões neurais do sistema nervoso tem interessado muitos pesquisadores, e tem fornecido muitas discussões sobre os méritos e as limitações desta abordagem de aprendizado. Em particular, as analogias com a biologia têm levado muitos pesquisadores a acreditar que as Redes Neurais possuem um grande potencial na resolução de problemas que requerem intenso processamento sensorial humano, tal como visão e reconhecimento de voz.

As pesquisas em Redes Neurais foram iniciadas com o trabalho pioneiro de McCulloch e Pitts em [McCulloch and Pitts, 1943]. McCulloch era um psiquiatra e pesquisou por 20 anos uma forma de representar um evento no sistema nervoso. Pitts era um jovem pesquisador e começou a trabalhar com McCulloch em 1942. Após a publicação desse livro, os estudos nessa área ficaram no esquecimento por 15 anos.

Dentre todos os estudos realizados, após o reinício das pesquisas na área de Redes Neurais, o artigo de Hopfield em 1982 e o livro de Rumelhart e McLelland [Rumelhart and McClelland, 1986], foram as publicações que mais influenciaram no resurgimento do interesse sobre Redes Neurais na década de 80. Redes Neurais tiveram um longo caminho desde McCulloch e Pitts, e continuarão a crescer em teoria, projetos e aplicações [Haykin, 1994].

3 Estimando o Desempenho dos Sistemas de Aprendizado

Quando um novo caso é apresentado ao classificador, o mesmo pode classificá-lo corretamente ou não. Seguramente, a taxa de erro é a forma mais utilizada para medir a performance de um classificador. Pode-se calcular a taxa de erro através da equação

$$taxa\ de\ erro = \frac{número\ de\ erros}{número\ de\ casos} \tag{1}$$

É conhecida por *taxa de erro aparente* de um classificador a taxa de erro calculada utilizando somente os exemplos de treinamento, ou seja, o sistema é treinado com um conjunto de exemplos e depois é verificado quantos erros o classificador cometeu classificando os mesmos exemplos com que foi treinado. A *taxa de erro verdadeira* é a taxa de erro obtida sobre um número muito grande de novos casos, selecionados independentemente dos casos usados para treinar o classificador. Ambas diferem porque a taxa de erro verdadeira exige um grande número de exemplos para testar o classificador, independentes dos exemplos de treinamento .

A taxa de erro verdadeira fornece uma excelente medida da performance de um classificador. Entretanto, conjuntos com grande número de exemplos são raros, o que inviabiliza o seu cálculo na maioria das vezes. O que faz-se, então, é contornar este problema por aproximar a taxa de erro verdadeira através de taxas calculadas sobre pequenos conjuntos.

Através de técnicas estatísticas que apresentam de diferentes formas os exemplos ao classificador é possível obter aproximações da taxas de erro verdadeira bem mais confiáveis que a taxa de erro aparente [Dietterich, 1997, Salzberg, 1997, Michie and Spiegelhalter, 1994]. Tais técnicas são objetos de discussão das próximas seções.

4 Tipos de Erros

Freqüentemente, são encontradas aplicações em que erros diferentes possuem pesos diferentes. Por exemplo, prever a ocorrência de uma chuva sem que essa aconteça não é tão mau quanto não prever a ocorrência da mesma e depois começar a chover. No primeiro caso, na pior das hipóteses, as pessoas sairiam de casa com guarda-chuvas, capas, etc, e não usariam, enquanto no segundo, as pessoas se molhariam, poderiam ficar doentes e, sendo um pouco trágicos, poderiam até morrer por causa de alguma doença adquirida enquanto estavam na chuva.

Assim, uma vez necessária a distinção entre os tipos de erros, uma *matriz de confusão* pode ser usada para separar os diferentes erros. A Tabela 1 mostra um exemplo desse tipo de matriz para três classes. O número de predições corretas para cada classe está representada ao longo da diagonal da matriz. Todos os outros números são erros de diferentes tipos. Por exemplo, na Tabela 1 a classe ♠ foi corretamente classificada 53 vezes, mas foi erroneamente classificada como classe ♢ 5 vezes.

	Classes Verdadeiras		
Classes Preditas	♣	♠	♢
♣	36	0	1
♠	0	53	2
♢	1	5	72

Tabela 1: Exemplo de Matriz de Confusão para Três Classes

A Tabela 2 mostra os tipos de erros que podem ocorrer através de uma matriz de confusão para duas classes.

Quando é necessário distinguir entre os diferentes tipos de erros, o *custo de classificação incorreta* é uma alternativa natural à taxa de erro. Tal custo é simplesmente uma penalidade pela ocorrência

	Classe Positiva	*Classe Negativa*
Predição Positiva	Verdadeiro Positivo	Falso Positivo
Predição Negativa	Falso Negativo	Verdadeiro Negativo

Tabela 2: Tipos de Erros para Classificação com Duas Classes

do erro. Um custo médio de classificação incorreta poderia ser obtido multiplicando cada custo pela respectiva taxa de erro. Por exemplo, no caso de duas classes, associa-se um custo ao erro falso positivo, e outro custo ao erro falso negativo. Para uma matriz de confusão qualquer, se $E_{(ij)}$ é o número de erros na matriz e $C_{(ij)}$ é o custo para cada tipo de erro, o custo total de classificação incorreta $C_{(total)}$ é dado pela equação 2, onde o custo para uma classificação correta ($C_{(ij)}$ para $i = j$) é zero.

$$C_{total} = \sum_{i=1}^{n} \sum_{j=1}^{n} E_{ij} C_{ij} \tag{2}$$

Com essas considerações, o que faz-se é primar por penalizar o classificador por cada classificação incorreta feita por ele. Contudo, é possível também *gratificá-lo*, estabelecendo ganhos, por classificações corretas. Esse processo é chamado de *análise de risco* ou *análise de decisão*, e nele, tanto custos como benefícios são levados em conta na análise da performance do classificador.

Neste processo, os custos são chamados de *riscos*. Se os custos de classificação incorreta são associados, por exemplo, a números negativos, e os ganhos para decisões corretas à números positivos, a equação 2 pode ser reescrita em termos de riscos, da seguinte forma

$$R_{total} = \sum_{i=1}^{n} \sum_{j=1}^{n} E_{ij} R_{ij} \tag{3}$$

No entanto, em casos reais, é muito difícil estabelecer os valores adequados para custos e riscos de forma avaliar corretamente a performance de um classificador. Por isso, a taxa de erro é mais utilizada como padrão para medir a performance de classificadores.

5 Técnicas para Estimar a Taxa de Erro Verdadeira

Dado que a taxa de erro aparente não é uma boa estimativa para a taxa de erro verdadeira, alguma alternativa deve ser encontrada.

Um dos requisitos para estimar a taxa de erro verdadeira é manter a amostra de exemplos sempre em ordem aleatória. Ou seja, a amostra de exemplos **nunca** deve ser pré-selecionada, de forma a evitar qualquer tipo de suposição quanto ao conjunto de exemplos. A seguir são descritos alguns métodos estatísticos utilizados na estimativa da taxa de erro verdadeira.

5.1 Estimativa da Taxa de Erro Através de Treinar-e-Testar

A idéia básica do método treinar-e-testar é dividir a amostra em dois conjuntos distintos e mutuamente excludentes:

- Conjunto de Treinamento
- Conjunto de Teste

O conjunto de treinamento é usado para treinar o sistema, gerando o classificador (ver seção 2.1). Já o conjunto de teste é usado para medir a taxa de erro do classificador, chamada de *taxa de erro do conjunto de teste*. A Figura 4 exemplifica bem como isso é feito.

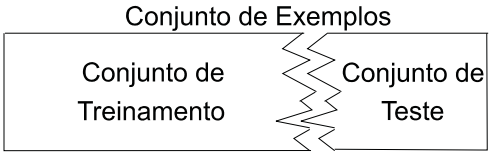

Figura 4: Conjuntos de treinamento e teste

Os dois conjuntos devem ser amostras aleatórias, apresentando somente casos independentes (entenda-se independente como a não existência de relação alguma entre os casos, a não ser o fato de pertencerem à mesma população). Por isso a questão da aleatoriedade é importante: para evitar que qualquer tipo de tendência apresentada por uma pré-seleção influencie na medição das taxas.

Da forma como foram abordadas até agora, as idéias do que são conjunto de treinamento e conjunto de testes parecem estar claras, contudo um tanto quanto abstratas, necessitando de parâmetros que as situem no mundo real. Com isso, precisamos responder à seguinte pergunta:

> *Quantos casos de teste são necessários para que a taxa de erro do conjunto de teste seja essencialmente a taxa de erro verdadeira?*

Surpreendentemente, pode-se obter excelentes aproximações da taxa de erro verdadeira, mesmo com amostras de exemplos pequenas. A Figura 5 [Weiss and Kulikowski, 1991] apresenta um gráfico que mostra a relação entre a taxa de erro do conjunto do teste e a mais alta possível taxa de erro verdadeira, para exemplos de teste de vários tamanhos. Estes valores possuem 95% de confiabilidade, isto é, não há mais de 5% de probabilidade que a taxa de erro exceda os valores apresentados. Por exemplo, para um conjunto de teste de tamanho 50, se a taxa de erro do conjunto do teste apresentar um valor de 0%, então há uma grande probabilidade que a taxa de erro verdadeira seja no máximo de 10%. Para 1000 casos de teste e taxa de erro do conjunto de teste de 0%, certamente a taxa de erro verdadeira ficará abaixo de 1%. Estes resultados não foram conjecturados, mas sim derivados de considerações básicas de probabilidade e estatística [Weiss and Kulikowski, 1991].

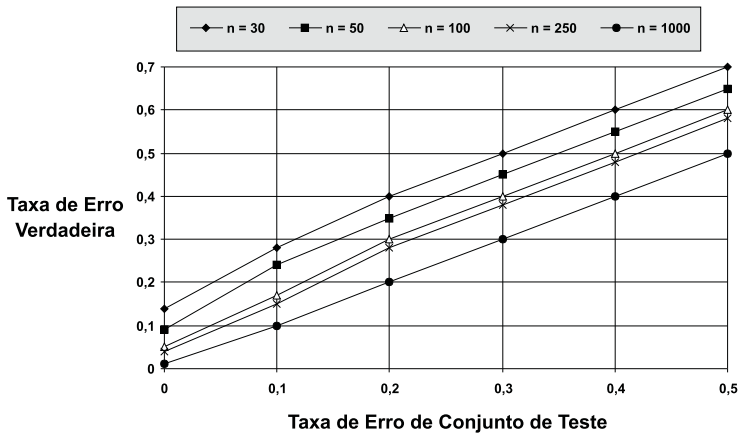

Figura 5: Número de Casos de Teste e Qualidade da Predição

Com isso, nota-se que a qualidade da estimativa por taxa de erro do conjunto de teste é diretamente dependente do número de casos do conjunto de teste.

Na prática o número de exemplos disponíveis é freqüentemente restrito. De modo que uma pergunta ainda precisa ser respondida:

Quantos casos deve conter o conjunto de teste e o de treinamento, respectivamente?

Existem várias formas conhecidas para dividir a amostra em conjuntos de treinamento e teste. O *Método Holdout* ou *Método H* divide fixamente a amostra de exemplos. É a técnica mais simples de estimativa de erro honesta, consistindo de um único esperimento treinar-e-testar.

Deve ser salientado que o método *holdout* apresenta uma taxa de erro do conjunto de teste próxima da taxa de erro verdadeira para grandes amostras de exemplos. Para pequenas amostras, a aproximação se torna pessimista e o método *holdout* se torna insatisfatório. Desta forma, para conjuntos pequenos, podem ser aplicadas variações de treinar-e-testar conhecidos por métodos *resampling*. Esses métodos serão discutidos a seguir.

5.2 Métodos de *Resampling*

Os métodos de resampling são variações dos métodos de treinar-e-testar nos quais são realizados vários experimentos de treinar-e-testar com diferentes partições dos exemplos. A seguir são descritos os seguintes métodos de resampling

- Random subsampling
- Cross-validation
- Bootstraping

5.2.1 Random Subsampling

Quando aplica-se o modelo *holdout* ao mesmo conjunto de dados várias vezes, são obtidos conjuntos de teste e de treinamento diferentes e independentes para cada uma das iterações.

Com isso, tem-se taxas de erros diferentes para cada um dos classificadores. Calculando a média entre essas diferentes taxas de erro encontra-se a taxa de erro estimada. Este método é conhecido como Random Subsampling e pode produzir melhores resultados que uma simples partição treinar-e-testar

A Tabela 3 mostra uma comparação entre os métodos *holdout* e *random subsampling* para uma amostra de n casos. Com o método *random subsampling* são criadas várias partições e o sistema é treinado e testado a cada iteração com uma partição diferente. No caso de uma partição ser pouco representativa, tal problema pode ser amenizado pois a taxa de erro final é calculada através das médias das taxas de erro de todas as partições.

	Holdout	*Random Subsampling*
Casos Treinamento	j	j
Casos de Teste	$n - j$	$n - j$
Iterações	1	Num. Iterações $\ll n$

Tabela 3: Comparação entre Holdout e Random Subsampling

5.2.2 Cross-validation

Cross-validation não é um único método, mas sim uma classe de métodos para estimar a taxa de erro verdadeira, também conhecidos como *k-fold cross-validation*. Nesses métodos, a amostra de exemplos é dividida em k partições mutuamente exclusivas. A cada iteração, uma partição diferente é usada para testar o sistema de aprendizado e todas as outras $k - 1$ partições são utilizadas para treinar o sistema. A taxa de erro é a média das taxas de erro calculadas para as diversas partições. Por exemplo, no caso do 10-fold cross-validation todo o conjunto de exemplos é dividido aleatoriamente em 10 partições diferentes mutuamente exclusivas. São realizadas 10 iterações, a cada iteração uma partição diferente é utilizada para testar e as outras 9 partições restantes são utilizadas para treinar o sistema. A taxa de erro é a média das taxas de erro das 10 iterações.

O método *Leaving-one-out* é um caso especial da classe de métodos cross-validation. Neste método, para uma amostra de n exemplos, são realizadas n iterações. A cada iteração um exemplo diferente é retirado do conjunto de exemplos e utilizado para testar o classificador. Os outros $n - 1$ exemplos são usados para treinar o sistema. A taxa de erro é calculada dividindo-se o número de erros observados por n.

O leaving-one-out é muito utilizado pois sua estimativa de taxa de erro verdadeira é praticamente não tendenciosa, isto é, quando usados vários conjuntos de exemplos, a média das estimativas tenderá a taxa de erro verdadeira.

Mas, por ser computacionalmente caro, o leaving-one-out é geralmente utilizado somente em pequenos conjuntos de exemplos. Para conjuntos maiores, o k-fold cross-validation, com $k = 10$,

apresenta uma aproximação quase tão apurada quanto o leaving-one-out, a um custo muito inferior. A Tabela 4 mostra uma comparação entre esses dois métodos de cross-validation.

	Leaving-One-Out	*10-fold Cross-Validation*
Casos Treinamento	$n-1$	90%
Casos de Teste	1	10%
Iterações	n	10

Tabela 4: Comparação entre Métodos de Cross-validation

5.2.3 Bootstraping

O problema de encontrar o melhor estimador para pequenas amostras é particularmente intrigante uma vez que pequenas amostras são bastante comuns. Tradicionalmente são consideradas pequenas amostras aquelas que possuem 30 ou menos casos. Por muitos anos, o leaving-one-out foi a técnica recomendada para avaliar a performace de classificadores para pequenas amostras. Uma vez que o leaving-one-out é virtualmente não tendencioso, ele pode ser aplicado a amostras muito maiores, apresentando resultados precisos.

Embora o leaving-one-out seja um estimador quase não tendensioso da taxa de erro verdadeira, ainda existem dificuldades com esta técnica. Tendência e variância de um estimador de taxa de erro contribuem para a imprecisão da taxa de erro estimada. Enquanto que o leaving-one-out é praticamente não tendensioso, sua variância para pequenas amostras é alta. Lembre-se que não tendensioso significa que o estimador, após várias aplicações, tende para a taxa de erro verdadeira. E variância significa dispersão dos dados em torno da média, ou seja, é o conjunto dos possíveis valores que podem ser atribuídos à média.

O efeito da variância tende a dominar em pequenas amostras. Desta forma, um estimador com baixa variância, mesmo que tendensioso, é potencialmente superior ao leaving-one-out para pequenas amostras.

Para amostras pequenas, o bootstrapping, um novo método de resampling, tem mostrado resultados promissores como estimador de taxa de erro. Esta é uma área de pesquisa ativa em estatística aplicada.

Existem vários estimadores bootstrap, mas dois estimadores em particular têm mostrado os melhores resultados, o *e0 bootstrap* e o *.632 bootstrap*. Para o estimador $e0$, o conjunto de treinamento consiste de n casos *amostrados com substituição* a partir de uma amostra de tamanho n. Amostrados com substituição significa que os exemplos de treinamento são retirados do conjunto inicial de exemplos mas cópias dos elementos retirados permanecem no conjunto inicial, de forma que elementos repetidos podem ser amostrados.

A fim de ilustrar o método, considera-se a seguinte amostra de casos contendo apenas sete casos

Amostra de casos: $\boxed{1\ 2\ 3\ 4\ 5\ 6\ 7}$

Para contruir o conjunto de treinamento deve-se sortear um caso aleatóriamente da amostra de casos. Considerando que o caso número 6 é sorteado, ele é copiado para o conjunto de

treinamento. Os casos devem ser copiados, e não retirados da amostra inicial, desta forma a amostra inicial deve sempre conter os mesmos casos. Novamente, outros casos são sorteados aleatóriamente e copiados, existindo a possibilidade do caso número 6 ser copiado mais de uma vez para o conjunto de treinamento. Os casos são copiados até que o conjunto de treinamento possua o mesmo número de casos que a amostra de casos inicial, ou seja, sete casos. Por exemplo, o conjunto de treinamento poderia conter

Conjunto de treinamento: $\boxed{1\ 2\ 3\ 3\ 3\ 6\ 6}$

Finalmente, o conjunto de teste é composto por todos os casos que não pertencem ao conjunto de treinamento, neste exemplo esse conjunto seria composto pelos seguintes casos

Conjunto de teste: $\boxed{4\ 5\ 7}$

Esta técnica possui uma fração média de casos não repetidos para o conjunto de treinamento de 0.632, e uma fração média de 0.368 para casos do conjunto de teste. O estimador .632 bootstrap é a combinação linear simples de

$$.368 * app + .632 * e0$$

Onde *app* é a taxa de erro aparente para todos os casos (treinamento e teste). Deve ser notado que $e0$ pode ser aproximando através da repetição do método 2-fold cross-validation, isto é, a divisão meio a meio da amostra inicial para treinamento e teste.

A taxa de erro final é calculada através da média das taxas de erro das várias iterações. Cerca de 200 iterações são consideradas necessárias para obter uma boa estimativa com o bootstrap. Portanto, este método pode ser considerado mais caro computacionalmente que o método leaving-one-out. A Tabela 5 resume as características básicas do método bootstrap.

	Bootstrap
Casos Treinamento	n (com j casos não repetidos)
Casos de Teste	$n - j$
Iterações	200

Tabela 5: Características dos Estimadores Bootstrap

6 Sistemas de Aprendizado Simbólicos

O objetivo desta seção é iniciar um estudo de alguns dos mais importantes sistemas de Aprendizado de Máquina Simbólico, mais especificamente os algoritmos **C4.5** [Quinlan, 1988], **C4.5rules** [Quinlan, 1988], **C5.0** [Rulequest-Research, 1999] e **CN2** [Clark and Niblett, 1989]. Através desse estudo serão identificadas as principais características do funcionamento desses algoritmos, incluindo as sintaxes do arquivo de entrada, de saída e os principais parâmetros. O objetivo final

é integrar, através de *scripts*, os programas de particionamento de dados para a estimativa de erro com alguns dos principais algoritmos de Aprendizado de Máquina. Com isso, será possível que o usuário estime o erro de um conjunto de dados sobre um algoritmo de aprendizado sem a necessidade de se preocupar em converter as sintaxes do arquivo de entrada ou chamar o algoritmo com os parâmetros corretos. Além disso, o sistema irá buscar na saída desses algoritmos pela taxa de erro. No caso do usuário utilizar um método de resampling, o sistema calculará as principais medidas de dispersão e centralidade.

6.1 O Base de Dados *Iris*

Antes de iniciar a discussão acerca dos algoritmos de aprendizado, descrevendo seu funcionamento, seus principais parâmetros, tipos de arquivos gerados, etc; falaremos brevemente sobre a base de dados que será fornecida a esses algoritmos, como exemplo de execução. A base escolhida é uma base bastante conhecida, a base *Iris* [Merz and Murphy, 1998].

Essa base, composta por um total de 150 instâncias, retrata observações feitas sobre uma espécie de planta chamada *iris*. São três as classes apresentadas pela base, cada uma representado um "tipo" de iris:

- setosa;
- versicolor;
- viginica.

Os atributos obervados foram 4:

1. Comprimento da sépala (sepal-length);
2. Largura da sépala (sepal-width);
3. Comprimento da pétala (petal-length);
4. Largura da sépala (petal-width).

A base de dados não apresenta ruídos (nenhum valor de atributo foi perdido) e possui uma distribuição de 50 instâncias para cada classe (33.3% para cada uma das três classes).

Como será visto nas seções seguintes, cada algoritmo exige uma sintaxe específica para seus arquivos de entrada e apresenta uma sintaxe também específica para seus arquivos de saída. Assim, embora apresentando uma forma de representação diferente, a base de dados é a mesma para todos os algoritmos. Ou seja, os atributos e seus respectivos valores se mantém inalterados todo o tempo.

6.2 O Sistema de Aprendizado C4.5

O sistema $C4.5$ pertence à família *TDIDT (Top Down Induction of Decision Trees)*, isto é, o $C4.5$ cria árvores de decisão para representar as regras de classificação [Quinlan, 1988].

Uma vez criada a árvore de decisão, é possível executar o algoritmo $C4.5$rules, que cria regras de decisão utilizando, inicialmente, a árvore gerada com a execução do $C4.5$. Maiores detalhes encontram-se na Seção 6.3

Dado um *conjunto de treinamento*, o $C4.5$ começa a dividí-lo em subconjuntos de acordo com os valores dos atributos apresentados, até que seja atingido o maior grau de refinamento possível, ou seja, todas as instâncias presentes em um dado subconjunto pertencem à mesma classe. É importante destacar, a essa altura, que a eficiência de uma árvore de decisão depende em muito do critério usado para escolher os atributos que serão usados na "quebra" do conjunto de treinamento em subconjuntos. Se um atributo é um forte indicador para a separação dos casos em suas devidas classes, este deve aparecer o quanto antes na árvore - perto da raiz.

A maioria dos algoritmos de árvores de decisão usam uma heurística para estimar o melhor atributo. O $C4.5$, em particular, usa uma versão modificada de medida de entropia da teoria da informação. Essa medida retorna um número entre 0 e 1 para cada atributo, onde 0 representa um conjunto uniforme (só há casos de uma classe presentes no conjunto) e 1 indica a presença de casos de todas as classes no conjunto.

6.2.1 Principais Parâmetros

Os comandos e parâmetros para executar o $C4.5$ são os seguintes:

`c4.5` Este comando inicia a construção da árvore de decisão.

`consult` Após a indução da árvore de decisão pelo $C4.5$, este comando pode ser usado para consultar o classificador gerado. É necessário que o usuário forneça os valores para os atributos.

Para executar o $C4.5$ o comando de linha assume a seguinte forma:

`c4.5`[*opções*]

As opções podem aparecer em qualquer ordem, as mais importantes são:

- `-f` *arquivo*
  Especifica o conjunto de dados de entrada. Não deve ser fornecida a extensão (ver Seção 6.2.2).

- `-u`
  Diz ao $C4.5$ para usar o arquivo de teste (`.test`). O *default* é não usar uma arquivo teste.

- `-s`
  Agrupa os valores discretos (em conjuntos) para os nós de teste na árvore de decisão. O *default* é não agrupar, para que dessa forma, cada caso seja tratado separadamente.

- `-m` *weight*
  Geralmente usado para dados com ruído, para "esvaziar" árvores com pouco poder preditivo.
- `-c` *CF*
  O valor de *CF* afeta a poda da árvore. Valores pequenos causam uma poda maior do que valores grandes. Se esse parâmetro não for informado, o valor *default* assumido é 25%.

Existem, além dessas, algumas outras opções, que são descritas detalhadamente em [Baranauskas and Monard, 2000

6.2.2 Arquivos

Todos os arquivos lidos e escritos pelo $C4.5$ assumem a forma *nome.extensão*, onde *extensão* caracteriza o tipo de dado envolvido. As extensões possíveis são as seguintes:

`.names` Este arquivo texto fornece os nomes das classes, atributos e enumera possíveis valores para atributos. Os valores para atributos são:

- **ignore** faz com que o atributo seja completamente ignorado
- **continuous** indica valores inteiros ou do tipo ponto flutuante
- **discrete** N onde $N > 0$ é um inteiro especificando que o atributo não pode assumir mais que N valores discretos
- uma lista de nomes separados por vírgulas, indicando que o atributo pode assumir esses valores discretos

`.data` Cada linha deste arquivo descreve uma instância, fornecendo valores para cada atributo. Valores desconhecidos podem ser especificados com o símbolo "?". A classe a que a instância pertence deve estar na última coluna.

`.test` Arquivo texto opcional que tem a mesma forma do `.data`. É usado para medir a precisão do classificador gerado. Geralmente este arquivo não deve apresentar casos que fazem parte do arquivo `.data`

`.unpruned` Arquivo de saída binário que contém as árvores não podadas geradas pelo $C4.5$. Este arquivo é usado pelo $C4.5$rules.

`.tree` Arquivo de saída binário que contém a árvore final podada. Se mais de uma árvore for gerada, a melhor é escolhida.

6.2.3 Sintaxe dos Arquivos de Entrada

Como visto na Seção 6.1, a base selecionada para exemplo foi a base *Iris*. Nas Figuras 6 e 7 pode ser observada a sintaxe desses arquivos.

No arquivo `.names` cada linha é separada por um ponto final. Linhas em branco, espaços e *tabs* podem ser usados para clareza. Qualquer texto à direita de uma barra vertical é ignorado e pode

ser considerado como um comentário. Além disso, a primeira entrada deste arquivo deve ser os valores da classe, separados por vírgula e terminando com um ponto final. As linhas restantes do arquivo consistem de uma linha para cada atributo. Uma descrição de atributo começa com o nome do atributo seguido por "dois pontos" e então os possíveis valores que o atributo pode assumir (terminando com um ponto final).

Vale lembrar que a ordem de declaração das atributos entre si não é relevante, mas a sua posição no arquivo `.data` deve ser a mesma do `.names`.

```
Iris−setosa, Iris−versicolor,
Iris−virginica.                | classes

sepal−length:                  continuous.
sepal−width:                   continuous.
petal−length:                  continuous.
petal−width:                   continuous.
```

Figura 6: Arquivo Iris.names

No arquivo `.data` cada valor deve ser separado usando-se vírgulas e o fim pode opcionalmente ser marcado por um ponto final. Como comentado anteriormente, os valores dos atributos devem aparecer na mesma ordem como eles foram declarados no arquivo `.names`. Valores desconhecidos podem ser especificados pelo símbolo de interrogação (?).

```
5.1,3.5,1.4,0.2,Iris−setosa
4.9,3.0,1.4,0.2,Iris−setosa
4.7,3.2,1.3,0.2,Iris−setosa
4.6,3.1,1.5,0.2,Iris−setosa
5.0,3.6,1.4,0.2,Iris−setosa
5.4,3.9,1.7,0.4,Iris−setosa
7.0,3.2,4.7,1.4,Iris−versicolor
6.4,3.2,4.5,1.5,Iris−versicolor
6.9,3.1,4.9,1.5,Iris−versicolor
5.5,2.3,4.0,1.3,Iris−versicolor
6.5,2.8,4.6,1.5,Iris−versicolor
5.7,2.8,4.5,1.3,Iris−versicolor
6.3,3.3,6.0,2.5,Iris−virginica
5.8,2.7,5.1,1.9,Iris−virginica
7.1,3.0,5.9,2.1,Iris−virginica
6.3,2.9,5.6,1.8,Iris−virginica
6.5,3.0,5.8,2.2,Iris−virginica
7.6,3.0,6.6,2.1,Iris−virginica
```

Figura 7: Parte do arquivo Iris.data

6.2.4 Sintaxe dos Arquivos de Saída

O arquivo de saída (`.tree`) é um arquivo binário, o que impossibilita sua exibição. Contudo, ao executar o programa $C4.5$ a árvore de decisão gerada é exibida. Na Figura 8 o classificador gerado pode ser observado.

```
C4.5 [release 8] decision tree generator        Thu Jan  4 10:22:47 2001
-------------------------------------------

    Options:
        File stem <iris>

Read 150 cases (4 attributes) from iris.data

Decision Tree:

petal-width <= 0.6 : Iris-setosa (50.0)
petal-width > 0.6 :
|   petal-width > 1.7 : Iris-virginica (46.01.0)
|   petal-width <= 1.7 :
|   |   petal-length <= 4.9 : Iris-versicolor (48.01.0)
|   |   petal-length > 4.9 :
|   |   |   petal-width <= 1.5 : Iris-virginica (3.0)
|   |   |   petal-width > 1.5 : Iris-versicolor (3.01.0)


Tree saved


Evaluation on training data (150 items):

         Before Pruning           After Pruning
         ----------------  ----------------------------
         Size      Errors   Size      Errors   Estimate

            9    3( 2.0%)      9    3( 2.0%)    ( 6.5%)   <<
```

Figura 8: Árvore de decisão induzida pelo $C4.5$ para a base *Iris*

6.3 O Sistema de Aprendizado C4.5rules

O $C4.5$rules é um sistema que depende integralmente do $C4.5$, pois usa, como ponto de partida, a árvore de decisão gerada por ele para gerar as sua regras. A razão para a criação do $C4.5$rules é que árvores de decisão muito grandes são de difícil leitura e compreensão, porque cada nó da árvore está inserido dentro de um contexto estabelecido pelos nós pais. É importante ressaltar que as regras criadas pelo $C4.5$rules nem sempre são uma tradução direta da árvore de decisão

em regras. O $C4.5$rules realiza uma série de processamentos, como poda das regras, que são diferentes daqueles feitos pelo $C4.5$.

6.3.1 Principais Parâmetros

Para executar o $C4.5$rules o comando de linha assume a seguinte forma:

`c4.5rules`[*opções*]

As opções podem aparecer em qualquer ordem. As três primeiras são idênticas às do $C4.5$ (Seção 6.2.1):

- `-f` *arquivo*

- `-u`

- `-c` *CF*

- `-r` *redundância*
  Esta opção permite ao usuário informar ao indutor que existem muitos atributos redundantes no conjunto de treinamento. Deve ser usada somente quando o usuário conhece muito bem a base com que está trabalhando.

6.3.2 Arquivos

O arquivos lidos e escritos pelo $C4.5$rules assumem a forma *nome.extensão*, onde *extensão* caracteriza o tipo de dado envolvido. Os arquivos são:

`.rules` Arquivo binário contendo conjunto final de regras gerado pelo $C4.5$rules

`.unpruned` Arquivo binário contendo as árvores não podadas geradas pelo $C4.5$

6.3.3 Sintaxe dos Arquivos de Entrada

Os dados de entrada para o $C4.5$rules são os mesmos usados para o $C4.5$ (Seção 6.2.3) mais o arquivo `.unpruned`.

6.3.4 Sintaxe dos Arquivos de Saída

O arquivo de saída do $C4.5$rules também é um arquivo binário. Assim, é listado o conjunto de regras exibido pelo programa quando o mesmo é executado. Na Figura 9 encontra-se o conjunto de regras geradas a partir da árvore exibida na Seção 6.2.4.

```
C4.5 [release 8] rule generator Thu Jan  4 10:19:23 2001
-------------------------------------

    Options:
        File stem <iris>

Read 150 cases (4 attributes) from iris

--------------------
Processing tree 0

Final rules from tree 0:

Rule 1:
        petal-width <= 0.6
        ->  class Iris-setosa  [97.3%]

Rule 2:
        petal-length <= 4.9
        petal-width > 0.6
        petal-width <= 1.7
        ->  class Iris-versicolor  [94.6%]

Rule 5:
        petal-width > 1.7
        ->  class Iris-virginica  [94.4%]

Rule 3:
        petal-length > 4.9
        ->  class Iris-virginica  [91.8%]

Default class: Iris-setosa


Evaluation on training data (150 items):

Rule  Size  Error  Used  Wrong          Advantage
----  ----  -----  ----  -----          ---------
   1     1   2.7%    50    0 (0.0%)       0 (0|0)   Iris-setosa
   2     3   5.4%    48    1 (2.1%)      47 (47|0)  Iris-versicolor
   5     1   5.6%    46    1 (2.2%)       5 (5|0)   Iris-virginica
   3     1   8.2%     6    2 (33.3%)      4 (4|0)   Iris-virginica

Tested 150, errors 4 (2.7%)   <<


          (a)  (b)  (c) <-classified as
         ---- ---- ----
           50           (a): class Iris-setosa
                47    3 (b): class Iris-versicolor
                 1   49 (c): class Iris-virginica
```

Figura 9: Conjunto de regras induzidas pelo $C4.5$rules para a base *Iris*

6.4 O Sistema de Aprendizado C5.0

O $C5.0$ trabalha como seus predecessores, $C4.5$ e $C4.5$rules, com a diferença de que agora os dois estão juntos em um único programa. Como trata-se de uma *versão*, serão discutidas somente as novas características incorporadas.

6.4.1 Principais Parâmetros

Os comandos e parâmetros para executar o $C5.0$ são os seguintes:

`c5.0` Este comando inicia a construção da árvore de decisão ou inicia a indução das regras, dependendo das opções usadas na linha de comando.

`predict` Após a indução da árvore ou das regras, este comando pode ser usado para consultar os classificadores gerados.

As seguintes opções são válidas para `predict`:

- `-f` *arquivo*
  Especifica o arquivo a ser lido.
- `-p`
  Mostra árvores ou conjunto de regras.
- `-r`
  Usa conjunto de regras ao invés de árvores.
- `-h` *weight*
  Imprime mensagem de ajuda.

Para executar o $C5.0$ o comando de linha assume a seguinte forma:

`c5.0`[*opções*]

As opções podem aparecer em qualquer ordem, as mais importantes são:

- `-f` *arquivo*
  Especifica o conjunto de dados de entrada. Não deve ser fornecida a extensão.
- `-u`
  Inicia o *adaptative boosting*: a idéia é criar vários classificadores ao invés de um único. Quando um novo caso está para ser classificado, cada classificador gerado dá o seu voto, dizendo como classificaria aquele caso. Os votos são contados e assim determina-se a classe final.
- `-v` *número*
  Permite ao usuário escolher o número v de classificadores que deseja criar.

- `-r`
  Esta opção cria regras ao invés de árvores.

- `-s`
  Agrupa os valores discretos (em conjuntos) para os nós de teste na árvore de decisão. O default é não agrupar, para que dessa forma, cada caso seja tratado separadamente.

- `-m` *weight*
  Geralmente usado para dados com ruído, para "limpar" árvores com pouco poder preditivo.

- `-c` *CF*
  O valor de *CF* afeta a poda da árvore. Valores pequenos causam uma poda maior do que valores grandes. Se esse parâmetro não for informado, o valor default assumido é 25%.

- `-S` *v*
  Indica a porcentagem do conjunto de treinamento a ser usada.

- `-h`
  Imprime mensagens de ajuda.

6.4.2 Arquivos

Todos os arquivos lidos e escritos pelo $C5.0$ assumem a forma *nome.extensão*, onde *extensão* caracteriza o tipo de dado envolvido. As extensões possíveis são as seguintes:

`.names` Este arquivo possui basicamente o mesmo formato do $C4.5$. Opções adicionais podem ser encontradas em [Baranauskas and Monard, 2000].

`.data` Este arquivo assume o mesmo formato do $C4.5$.

`.test` Este arquivo opcional assume a mesma forma do `.data`. É usado para medir a precisão do classificador gerado.

`.cases` Este arquivo é opcional e tem a mesma forma do `.data`. Ele difere do `.test` somente por permitir que a classe nas instâncias seja desconhecida. Esses casos serão rotulados segundo o modelo (árvore ou regras) gerado

`.costs` Este arquivo também é opcional e é usado quando quer-se atribuir pesos diferentes a tipos de erros diferentes. Cada entrada do arquivo tem a seguinte forma:

```
classe-predita, classe-verdadeira:  peso
```

onde `peso` é um número real não negativo.

`.tree` Arquivo de saída binário que contém a árvore podada final. Este arquivo é usado pelo `predict`

`.rules` Arquivo de saída contendo o conjunto final de regras gerado pelo $C5.0$. Este arquivo também é usado pelo programa `predict`.

6.4.3 Sintaxe dos Arquivos de Entrada

Tanto o arquivo `.names` como o `.data` assumem, basicamente, o mesmo formato dos usados pelo $C4.5$ (Seção 6.2.3).

6.4.4 Sintaxe dos Arquivos de Saída

Os dois arquivos de saída gerados pelo $C5.0$ (`.tree` e `.rules`) são binários. Contudo, quando o programa é executado, a árvore de decisão e o conjunto de regras são exibidos e ficam em um arquivo texto de extensão `.out`. A Figura 10 mostra a listagem desse arquivo.

6.5 O Sistema de Aprendizado CN2

$CN2$ é um algoritmo de aprendizado de máquina que induz regras da forma *if* <condição> *then* <classe>.

O $CN2$ é capaz de gerar regras ordenadas ou não ordenadas. Regras ordenadas são regras da forma:

if condição *then* classe = C1

else if condição *then* classe = C2

else if condição *then* classe = C3

...

Nessa situação, um novo caso começa a ser classificado a partir da primeira regra. Caso ele não satisfaça a condição dessa regra ele entra no *else* (conseqüentemente em um novo *if*) e assim ele sucessivamente até encontrar uma regra que o classifique. Dessa forma, um determinado caso pode ser coberto somente por uma única regra.

No mecanismo que gera regras não ordenadas, as regras de classificação são do tipo:

if condição *then* classe = C1

if condição *then* classe = C2

if condição *then* classe = C3

...

Aqui, um novo caso a ser classificado considera **todas** as regras de classificação, mesmo que em algum momento já tenha sido coberto por alguma regra. Após isso, o algoritmo utiliza alguma política para decidir a classe do caso, baseado nas regras que cobriram o caso. Uma forma de fazer isso é fazer com que cada regra que cobre o exemplo seja considerada um "voto" para a classificação desse exemplo. A classe mais votada será a classe atribuída a esse caso.

```
See5 INDUCTION SYSTEM [Release 1.09a]    Thu Jan 04 11:24:04 2001
-------------------------------------

    Options:
        Generating rules

Read 150 cases (4 attributes) from iris.data

Decision tree:

petal-width <= 0.6: Iris-setosa (50.0)
petal-width > 0.6:
:...petal-width > 1.7: Iris-virginica (46.01.0)
    petal-width <= 1.7:
    :...petal-length <= 4.9: Iris-versicolor (48.01.0)
        petal-length > 4.9:
        :...petal-width <= 1.5: Iris-virginica (3.0)
            petal-width > 1.5: Iris-versicolor (3.01.0)

Extracted rules:

Rule 1: (cover 50)
        petal-width <= 0.6
        -> class Iris-setosa  [0.981]

Rule 2: (cover 54)
        petal-width > 0.6
        petal-width <= 1.7
        -> class Iris-versicolor  [0.893]

Rule 3: (cover 46)
        petal-width > 1.7
        -> class Iris-virginica  [0.958]

Rule 4: (cover 46)
        petal-length > 4.9
        -> class Iris-virginica  [0.938]

Default class: Iris-setosa

Evaluation on training data (150 cases):

          Decision Tree           Rules
        ----------------    ----------------
        Size      Errors      No      Errors

          5    3( 2.0%)       4    4( 2.7%)   <<

         (a)   (b)   (c)      <-classified as
        ----  ----  ----
         50                   (a): class Iris-setosa
               47     3       (b): class Iris-versicolor
                1    49       (c): class Iris-virginica
```

Figura 10: Árvore de decisão e conjunto de regras induzidos pelo C5.0 para a base *Iris*

6.5.1 Principais Parâmetros

O comando executar o $CN2$ é o seguinte:

`cn` Este comando inicia a indução das regras.

Diferente dos outros sistemas, vistos até agora, o programa que implementa o $CN2$ é um ambiente, e não um programa executado na forma de linha de comando. Cada comando no `cn` é executado digitando-se a letra inicial (maiúscula ou minúscula) do comando, exceto para `eXecute` que é chamado por `x`. O comportamento do $CN2$ pode mudar à medida que esses parâmetros vão sendo ajustados, mas todos eles já possuem um valor *default* caso não seja informado um valor específico. Com isso, é imprescindível para o funcionamento do $CN2$ ajustar estes parâmetros antes de processar os dados.

As principais opções são:

- `Read`
  Lê o arquivo de atributos, exemplos (instâncias), atributos-e-exemplos ou o arquivo de regras. Vale ressaltar que o $CN2$ pode armazenar na memória somente um conjunto de atributos, e um de exemplos, de cada vez.

- `Induce`
  Executa o indutor de regras $CN2$ sobre o conjunto de atributos e instâncias lido mais recentemente.

- `Write`
  Após usar a opção `Induce`, é possível usar esta opção para escrever o conjunto de regras em um arquivo.

- `eXecute`
  Com este comando procede-se à análise da performance das regras geradas avaliando um conjunto de exemplos. Neste modo tem-se as seguintes opções:

  1. `All:` Avalia o conjunto de regras como um todo.
  2. `Individual:` Avalia as regras individualmente.
  3. `Help/?:` Mostra o menu de comandos.
  4. `<RET>:` Retorna ao prompt do $CN2$.

- `Algorithm`
  Especifica se o $CN2$ produzirá um conjunto de regras ordenados ou não ordenadas.

- `Quit`
  Sai do $CN2$

6.5.2 Arquivos

Todos os arquivos lidos e escritos pelo $CN2$ assumem a forma *nome.extensão*, mas, diferente dos outros sistemas, a *extensão* pode ser qualquer string escolhida pelo usuário. Em geral, as

pessoas que trabalham com o $CN2$ seguem uma nomenclatura para os nomes de arquivo que foi a sugerida pelo autor do $CN2$:

`.att` Este arquivo texto possui os nomes das classes, atributos e enumera possíveis valores para os atributos.

`.exs` Arquivo de exemplos, contendo os casos usados para gerar o classificador.

`.tst` É um arquivo opcional e possui a mesma forma do arquivo `.exs`. É usado para medir a precisão do classificador gerado.

`.rules` É um arquivo de saída contendo o conjunto de regras induzidas pelo $CN2$.

6.5.3 Sintaxe dos Arquivos de Entrada

Utilizando a nomenclatura sugerida pelo autor do $CN2$, para a base *Iris*, tem-se os arquivos `.att` e `.exs` como arquivos de entrada.

No arquivo `.att`, mostrado na Figura 11, cada fim de linha é terminado como um "ponto-e-vírgula" (;). Linhas em branco, espaços e *tabs* são usados para clareza. Qualquer texto à direita do símbolo % é ignorado e pode ser usado como comentário. A primeira linha do arquivo, deve ser a texto

```
**ATRIBUTE FILE**
```

As linhas restantes são constituídas de uma linha para cada atributo, sendo que a descrição de um atributo começa com o nome seguido de "dois pontos" (:) e então os possíveis valores que o atributo pode assumir. Esses valores podem ser:

- **(INT)** indicando valores inteiros
- **(FLOAT)** indicando valores do tipo ponto flutuante
- uma lista de nomes separados por espaços ou *tabs*, indicando que o atributo pode assumir valores discretos

A ordem das declarações deve seguir a posição exata dos dados no arquivo `.exs`. A última coluna indica a classe, que é seguida por um "ponto-e-vírgula" (;).

```
**ATTRIBUTE FILE**

sepal−length: (FLOAT)
sepal−width: (FLOAT)
petal−length: (FLOAT)
petal−width: (FLOAT)
class: Iris−setosa Iris−versicolor Iris−viginica;
```

Figura 11: Arquivo Iris.att

No arquivo `.exs`, mostrado na Figura 12, a primeira linha deve ser o texto

```
**EXAMPLE FILE**
```

Cada linha descreve uma instância, ou caso, com os valores para cada atributo. Cada valor deve ser separado do outro por espaços ou *tabs* e a linha termina com um "ponto-e-vírgula". Os atributos devem aparecer na mesma ordem em que foram declarados no arquivo `.att`. Valores desconhecidos podem ser especificados pelo símbolo de interrogação (?).

```
**EXAMPLE FILE**

5.1 3.5 1.4 0.2 Iris−setosa;
4.9 3.0 1.4 0.2 Iris−setosa;
4.7 3.2 1.3 0.2 Iris−setosa;
4.6 3.1 1.5 0.2 Iris−setosa;
5.0 3.6 1.4 0.2 Iris−setosa;
5.4 3.9 1.7 0.4 Iris−setosa;
6.4 3.2 4.5 1.5 Iris−versicolor;
6.9 3.1 4.9 1.5 Iris−versicolor;
5.5 2.3 4.0 1.3 Iris−versicolor;
6.5 2.8 4.6 1.5 Iris−versicolor;
5.7 2.8 4.5 1.3 Iris−versicolor;
6.3 3.3 4.7 1.6 Iris−versicolor;
5.7 2.8 4.1 1.3 Iris−versicolor;
6.3 3.3 6.0 2.5 Iris−viginica;
5.8 2.7 5.1 1.9 Iris−viginica;
7.1 3.0 5.9 2.1 Iris−viginica;
6.3 2.9 5.6 1.8 Iris−viginica;
6.5 3.0 5.8 2.2 Iris−viginica;
7.6 3.0 6.6 2.1 Iris−viginica;
```

Figura 12: Parte do arquivo Iris.exs

6.5.4 Sintaxe dos Arquivos de Saída

O único arquivo de saída do $CN2$ é o `.rules` e ele se caracteriza pela presença, na primeira linha do arquivo, da frase:

```
**RULE FILE**
```

Nas Figuras 13 e 14, podem ser observados dois exemplos de arquivo de saída gerados pelo $CN2$. Na Figura 13 encontra-se o conjunto de regras ordenado, enquanto que na Figura 14, encontra-se o conjunto de regras não ordenadas, ambos induzidos para base de dados *Iris*.

```
**RULE FILE**
@
Time: [ Mon Dec 25 11:37:35 2000 ]
Examples: iris.exs
Algorithm: ORDERED
Error_Estimate: LAPLACIAN
Threshold: 0.00
Star: 5
@

ORDERED-RULE-LIST

IF    petal-length < 2.45
THEN  class = Iris-setosa  [50 0 0]
ELSE
IF    petal-length < 4.75
  AND petal-width < 1.65
THEN  class = Iris-versicolor  [0 44 0]
ELSE
IF    sepal-length > 6.05
  AND petal-width > 1.75
THEN  class = Iris-viginica  [0 0 38]
ELSE
IF    sepal-length < 6.20
  AND sepal-width < 3.10
  AND petal-width > 1.65
THEN  class = Iris-viginica  [0 0 8]
ELSE
IF    petal-length < 4.95
THEN  class = Iris-versicolor  [0 4 0]
ELSE
IF    petal-width < 1.55
THEN  class = Iris-viginica  [0 0 3]
ELSE
IF    sepal-length < 6.95
THEN  class = Iris-versicolor  [0 2 0]
ELSE
(DEFAULT) class = Iris-viginica  [0 0 1]
```

Figura 13: Conjunto de regras ordenadas

```
**RULE FILE**
@
Time: [ Mon Dec 25 12:12:59 2000 ]
Examples: iris.exs
Algorithm: UNORDERED
Error_Estimate: LAPLACIAN
Threshold: 0.00
Star: 5
@

UNORDERED-RULE-LIST

IF    petal-length < 2.45
THEN  class = Iris-setosa  [50 0 0]

IF    sepal-length > 4.95
  AND 2.45 < petal-length < 4.95
  AND petal-width < 1.75
THEN  class = Iris-versicolor  [0 46 0]

IF    2.35 < sepal-width < 2.45
THEN  class = Iris-versicolor  [0 3 0]

IF    sepal-length > 6.25
  AND sepal-width > 2.45
  AND petal-length > 5.05
THEN  class = Iris-viginica  [0 0 35]

IF    sepal-width < 3.10
  AND petal-width > 1.75
THEN  class = Iris-viginica  [0 0 28]

IF    sepal-length < 6.15
  AND sepal-width < 2.65
  AND petal-length > 4.45
THEN  class = Iris-viginica  [0 0 4]

IF    petal-length > 5.25
THEN  class = Iris-viginica  [0 0 32]

(DEFAULT) class = Iris-setosa  [50 50 50]
```

Figura 14: Conjunto de regras não ordenadas

7 Projeto e Implementação de Alguns Métodos de Resampling

Nesta seção é discutida a implementação de alguns dos métodos de resampling. Vale ressaltar que aqui serão exibidos os algoritmos em alto nível de tais implementações. O códigos fonte dessas implementações, na linguagem de programação PERL, podem ser observados no Apêndice A.

Os métodos de resampling implementados neste trabalho são:

- Holdout
- Random Subsampling
- Cross Validation

Auxiliarmente a estes métodos foi implementado um programa chamado *mix* para "embaralhar" as instâncias dentro da base, para preservar uma característica fundamental das amostras, a aleatoriedade. A seguir são explicitados cada um dos algoritmos, inclusive o do programa *mix*, que embaralha o arquivo de exemplos, e o funcionamento de cada um é discutido em linhas gerais.

7.1 O programa Mix (*mix*)

Este programa, executado na forma de linha de comando, tem a seguinte forma:

`mix` [*arquivo de entrada*] [*arquivo de saída*]

Como o arquivo manipulado é um arquivo do tipo *texto*, em que cada linha representa uma instância, o que este programa faz é simplesmente trocar as linhas de lugar aleatoriamente, misturando-as.

Dado o arquivo a ser *misturado*, o programa cria 4 (quatro) arquivos auxiliares, chamados *aux0* ... *aux3*, que serão usados no processo de *partição* do arquivo original. Neste processo é utilizada uma função que gera número aleatórios. Essa função é disparada de tal modo que gere números entre 0 e 3. Assim, é lida uma linha do arquivo original e, dependendo do número gerado, a linha lida é gravada em um dos arquivos auxiliares. Por exemplo, se o número gerado for 2, a linha é gravada no arquivo *aux2*, e assim sucessivamente até o fim do arquivo original. Com isso, no final desse processo, tem-se o arquivo original divido, de maneira aleatória, em quatro arquivos auxiliares.

Depois de gerados os arquivos auxiliares é feito um *merge* desses arquivos dois a dois. A forma como acontece a escolha dos pares de arquivos também é aleatória. Mais uma vez a função que gera números aleatórios é usada para gerar números entre 0 e 3. Supondo que, nesse estágio da execução do programa, os dois primeiros números gerados sejam 3 e 0, então o primeiro *merge* acontecerá entre os arquivos *aux3* e *aux0*, e conseqüentemente o outro *merge* será realizado entre *aux1* e *aux2*. Vale salientar que a função que faz esta escolha, trata de possíveis casos especiais, como por exemplo o número 3 ser gerado mais de uma vez, o que faria com que *aux3* participasse

```
início
        ler(nomearq_entrada);
        ler(nomearq_saída);

        fonte := abrir(nomearq_entrada);
        destino := criar(nomearq_saida);
        cria_arquivos_auxiliares(aux0,aux1,aux2,aux3,merge1,merge2);

        enquanto não eof(fonte)
        início
                linha := próxima_linha(fonte);
                i := sorteia (0, 3);
                concatena("aux", i, linha);
        fim;

        i := sorteia (0, 3);
        j := sorteia (0, 3);

        merge("aux", i, j, "merge1");

        complementar (i, j, k, l, 0, 3);

        merge("aux", i, j, "merge1");
        merge("merge1", "merge2", "destino");

        fechar(fonte, destino, aux0, aux1, aux2, aux3, merge1, merge2);

        remover(aux0, aux1, aux2, aux3, merge1, merge2);
fim.
```

Figura 15: Algoritmo do programa `mix`

em mais de um *merge* ou fizesse um *merge* com ele mesmo. Os arquivos resultantes de ambos os *merges* são gravados em arquivos auxiliares de nomes *merge1* e *merge2*.

Finalmente, esses dois arquivos passam por um último *merge* e o arquivo final é gerado. Após isso, todos os arquivos auxiliares gerados pelo programa são removidos e os arquivos original e final são fechados.

7.2 O programa Holdout (*holdout*)

Este programa, executado na forma de linha de comando, tem a seguinte forma:

`holdout` [*arquivo de entrada*] [*porcentagem arq. treinamento*] [*diretório de destino*]

O método holdout divide fixamente uma amostra em duas, uma para treinamento e outra para teste. É por isso que um dos parâmetros desse programa é a porcentagem, que deve ser fornecida

pelo usuário como um número real entre 0 e 100 (não incluindo os valores 0 e 100). Os arquivos de treinamento e teste são colocados em um diretório já existente, que pode ser especificado pelo usuário. Caso o usuário não informe nenhum caminho, os arquivos são gerados no diretório corrente.

O programa começa por "misturar" o arquivo original, através do programa *mix*, gerando o arquivo *arqaux*. Em seguida *arqaux* é aberto e é obtido dele o seu número de linhas. Tendo o número de linhas e a porcentagem que fará parte do arquivo de treinamento, calcula-se a quantidade de linhas (variável *treinamento*) a copiar para o arquivo de treinamento.

```
início
        ler(nome_arq);
        ler(porcentagem);
        ler(diretório);

        "mix nome_arq arqaux";

        original := abrir(arqaux);
        treinar := criar(treinar, diretório);
        testar := criar(testar, diretório);

        numlinhas := número_linhas(arqaux);
        treinamento := (numlinhas * porcentagem)/100;

        i := 0;
        enquanto i < treinamento
        início
                aux := próxima_linha(arqaux);
                concatenar(treinar, linha);
                i := i + 1;
        fim

        enquanto i < numlinhas
        início
                aux := próxima_linha(arqaux);
                concatenar(testar, linha);
                i := i + 1;
        fim

        fechar(original, treinar, testar);
        remover(arqaux);
fim.
```

Figura 16: Algoritmo do programa `holdout`

Com isso, é iniciado um laço, que vai de 0 a *treinamento*, que lê uma linha de cada vez de *arqaux* e a copia para o arquivo de treinamento. Terminado esse laço, outro é iniciado, indo da linha de número *treinamento + 1* até o final de *arqaux*, copiando as linhas lidas para o arquivo de teste. Gerados os arquivos, todos os arquivos são fechados e *arqaux* é removido.

7.3 O programa Random Subsampling (*rsampling*)

Este programa, executado na forma de linha de comando, tem a seguinte forma:

`rsampling` [*núm. iterações*] [*porc. arquivo de treinamento*] [*arquivo de entrada*] [*dir. saída*]

À excessão do número de iterações (k), todas as outras entradas tem o mesmo significado das entradas descritas no programa holdout. No *rsampling*, se nenhum diretório é informado, subdiretórios *default* são criados e devidamente indexados com índices que vão de 0 a k-1.

O programa consiste de um único laço, que vai de 0 a k-1, sendo que, cada passagem pelo laço se resume à chamda do programa holdout descrito anteriormente (Seção 7.2). Ao executar o programa holdout, são passados o nome do arquivo e a porcentagem fornecidos pelo usuário. Já no que se refere à passagem do diretório de destino, tomou-se o cuidado de, em cada iteração, passar o endereço de diretório devidamente indexado, para que não ocorram perdas de dados, entre outros problemas.

```
início
        ler(k)
        ler(porcentagem);
        ler(diretório);
        ler(nome_arq);


        cria_diretórios(diretório, k);

        i := 0;
        enquanto i < k
        início
                "holdout nome_arq porcentagem diretório"
        fim
fim.
```

Figura 17: Algoritmo do programa `rsamp`

7.4 O programa Cross Validation (*kfold*)

Este programa, executado na forma de linha de comando, tem a seguinte forma:

`kfold` [*núm. partições*] [*arquivo de entrada*] [*diretório de saída*]

O objetivo é implementar o método estatístico k-fold cross validation. Devem ser fornecidos ao programa o número de iterações (k), o arquivo de entrada e o diretório de saída. Se nenhum nome de diretório de saída for fornecido, são criados subdiretórios *default*, de nome *it*, devidamente indexados com índices que vão de 0 a k-1.

Feitas essas considerações, pode-se discutir o programa propriamente dito. Primeiramente, o arquivo original passa pelo programa *mix* para ser embaralhado. Depois, são criados k arquivos auxiliares, *aux0 ... aux(k-1)*, que serão usados na *partição* do arquivo original. Nessa partição o arquivo original é igualmente dividido pelos arquivos auxiliares. Ou seja, a primeira linha do arquivo original é lida e gravada em $aux0$, a segunda em $aux1$ e assim por diante, até que a linha k-1 seja gravada em aux(k-1). Após isso, volta-se ao arquivo $aux0$, que recebe a linha k, $aux1$ recebe a linha (k+1), e assim sucessivamente até atingir o fim do arquivo original.

```
início
        ler(k);
        ler(nome_arq);
        ler(diretório);

        "mix nome_arq arqaux";

        cria_diretórios(diretório, k);

        fonte := abrir(arqaux);
        abrir_v"srios(k, "aux");

        i := 0;

        enquanto não eof(fonte)
        início
                linha := proxima_linha(fonte);
                concatena("aux", i, linha);
                i := i + 1;
                se i = k então
                        i := 0;
        fim

        para i:= até i = k − 1
        início
                treinar := criar(treinar, diretório);
                testar  := criar(testar, diretório);
                merge("aux", 0...i−1, i+1...k−1, treinar, diretório);
                copia("aux", i, testar, diretório);
                fechar(treinar, testar, diretório)
        fim

        fechar(aux0, ..., auxk−1, fonte);
        remover(aux0, ..., auxk−1);
fim.
```

Figura 18: Algoritmo do programa `kfold`

Com os arquivos auxiliares prontos, começa o processo de geração dos arquivos dentro dos subdiretórios. Dentro de cada subdiretório, os arquivos de treinamento e teste tem o nome do arquivo original com as extensões `.data` e `.test`, respectivamente. A criação desses arquivos é simples.

Para o subdiretório de índice i, foi convencionado que o arquivo $auxi$ seria o arquivo de teste, enquanto que todos os outros $aux's$ formariam o arquivo de treinamento. Assim, para um subdiretório de índice 2, por exemplo, o arquivo $aux2$ seria o arquivo de teste daquela iteração e com isso seria copiado para dentro dele como o `.test`. Os demais arquivos são simplesmente copiados para o `.data` e passarão a constituir o conjunto de treinamento daquela iteração. Finalmente, após todas as iterações terem sido finalizadas, os arquivos são fechados e os arquivos auxiliares são removidos.

A Programas Fontes

Neste apêndice são listados os programas discutidos na seção 7. Como já mencionado antes, foram implementados na linguagem de programação PERL devido aos recursos que esta apresenta para manipulação de arquivos texto.

A.1 *mix*

```
#! /usr/bin/perl

# Este programa faz o "shuffle" de um arquivo fornecido, ou seja, embaralha suas
# linhas. Para tanto, ele trabalha com quatro arquivos auxiliares(aux's 1,2,3,4)
# Como funciona:
#    Gera, através da função RAND() um número aleatório entre 0 e 3 (inclusive)
#    É lida uma linha do arquivo orignal. Este número designa para qual dos
#    arquivos auxiliares a linha será mandada. Depois, é feito um "merge" dos
#    arquivos também seguindo este raciocínio de número aleatórios, pra decidir
#    quais arquivos serão os primeiros a realizar o merge.
# Entradas: O nome do arquivo que se deseja embaralhar e o nome que se deseja
#    dar ao arquivo de saída.
# Saídas: O novo arquivo embaralhado.

# Leitura do terminal, dos nomes dos arquivos na forma command-line.

$nomearq  = $ARGV[0];
$nome_out = $ARGV[1];

# Variáveis auxiliares que serão usadas como passagem de parâmetros às subrotinas

$k;
$nome;

# Aviso, case o usuário não passe, ou passe errado os parametros.

if (($nomearq eq "") and ($nome_out eq "") )
{
    die ("Forma : mixd [arquivo de entrada] [arquivo de saida]\n");
}

# Abertura dos arquivos, e verificação e verificação de possíveis erros
# de abertura.

unless   (open (original, $nomearq))
{
    if (-e $nomearq)
```

```
    {
        die("Arquivo $nomearq existe mas não pode ser aberto! \n");
    }
    else
    {
        die("Arquivo $nomearq não existe! \n");
    }
}

unless (open (saida, ">$nome_out"))
{
    die("Arquivo $nome_out não pode ser aberto! \n");
}

# Abre K arquivos auxiliares para escrita.

$k = 3;
$nome = "aux";
&abre_n ($k, $nome);

# Geração dos arquivos auxiliares de modo ramdômico.

&gera_aux;

# Fechando os arquivos auxiliares.

&fecha_n ($k, $nome);

# Escolhendo randomicamente qual a ordem dos arquivos em que será realizado o
# merge.

&escolhe_ordem;

# Abrindo os arquivos auxiliares para leitura, na ordem determinada acima.

$k = 3;
$nome = "aux".$arqmerge;
&abrel_n ($k, $nome);

# Depois de escolhida a ordem dos arquivos, dois arquivos auxiliares são criados para a
# realizacao de dois merges, de dois arquivos, que gerarão merge0 e merge0. Com estes dois,
# faço mais um merge e gero o arquivo final.

$k    = 1;
$nome = "merge";
&abre_n ($k, $nome);
```

```
# Agora com os arqivos de merge abertos, o merge é feito.
# Primeiro com os dois primeiros arquivos (aux0 e aux1) sendo gravados em
# "merge0" e depois os restantes (aux2 e aux3) sendo gravados em "merge1".

$aux   = 1;
$merge = "merge0";
$string0 = "aux0";
$string1 = "aux1";

while ($aux <= 2)
{
    $line0 = <$string0>;
    $line1 = <$string1>;
    while ($line0 ne "" || $line1 ne "")
    {
        if ($line0 ne "")
        {
            print $merge ($line0);
            $line0 = <$string0>;
        }
        if ($line1 ne "")
        {
            print $merge ($line1);
            $line1 = <$string1>;
        }
    }
    $aux = $aux + 1;
    if ($aux == 2)
    {
        $merge = "merge1";
        $string0 = "aux2";
        $string1 = "aux3";
    }
}

# Fechando os dois arquivos de merge e reabrido-os para leitura.
$k    = 1;
$nome = "merge";
&fecha_n ($k, $nome);

&abrel_n ($k, $nome);

# Com os dois arquivos de merge prontos, é feito um merge dos dois e o arquivo
# de saída é gerado.
```

```
$line0 = <merge0>;
$line1 = <merge1>;
while ($line0 ne "" || $line1 ne "")
{
    if ($line0 ne "")
    {
        print saida ($line0);
        $line0 = <merge0>;
    }
    if ($line1 ne "")
    {
        print saida ($line1);
        $line1 = <merge1>;
    }
}

# Fechando todos os arquivos

close(original);
close(saida);

$k = 3;
$nome = "aux";
&fecha_n ($k, $nome);

$k = 1;
$nome = "merge";
&fecha_n ($k, $nome);

# Executando chamadas ao sistema para deletar os arquivos auxiliares.

$k = 3;
for ($aux = 0; $aux <= $k; $aux++)
{
system("rm aux$aux");
}
system("rm merge0");
system("rm merge1");


#  S U B R O T I N A S

# Rotina que abre k arquivos para escrita, com nome dado por $nome e índices
# que vão de 0 à $k - 1.
sub abre_n
{
```

```
    my ($k, $nome) = @_;
    my ($aux);
    for ($aux = 0; $aux <= $k; $aux++)
    {
        unless (open ($nome.$aux, ">$nome".$aux))
        {
            die("Arquivo $nome.$aux não pode ser aberto! \n");
        }
    }
}

# Rotina que realiza a mesma função da rotina acima, com a diferença de que
# abre os arquivos para leitura.
sub abrel_n
{
    my ($k, $nome) = @_;
    my ($aux);

    for ($aux = 0; $aux <= $k; $aux++)
    {
        unless (open ($nome.$aux, "$nome".$aux))
        {
            if (-e "$nome".$aux)
            {
                die("Arquivo $nome$aux existe mas não pode ser aberto! \n");
            }
            else
            {
                die("Arquivo $nome$aux não existe! \n");
            }
        }
    }
}

# Rotina que fecha k arquivos de nome $nome seguindo a definições acima para $k e $nome.
sub fecha_n
{
    my ($k, $nome) = @_;
    my ($aux);
    for ($aux = 0; $aux <= $k; $aux++)
    {
        close($nome.$aux);
    }
}

# Rotina que gera os arquivos auxiliares de modo ramdômico.
```

```
sub gera_aux
{
    my ($line, $randomico);

    srand();
    $line = <original>;
    $randomico = 0;
    $randomico = int(rand(4));

    while ($line ne "")
    {
        if ($randomico == 0)
        {
            print aux0 ($line);
            $randomico = int(rand(4));
            $line = <original>;
        }
        elsif ($randomico == 1)
        {
            print aux1 ($line);
            $randomico = int(rand(4));
            $line = <original>;
        }
        elsif ($randomico == 2)
        {
            print aux2 ($line);
            $randomico = int(rand(4));
            $line = <original>;
        }
        else
        {
            print aux3 ($line);
            $randomico = int(rand(4));
            $line = <original>;
        }
    }
}


# Subrotina que determina qual a ordem em arquivos aux's serão pegos para realizar o merge.
sub escolhe_ordem
{
    srand();

    $arqmerge0 = int(rand(4));
    $arqmerge1 = int(rand(4));
```

```
    while ($arqmerge1 == $arqmerge0)
    {
        $arqmerge1 = int(rand(4));
    }
    $arqmerge2 = int(rand(4));
    $arqmerge3 = int(rand(4));

    while (($arqmerge2 == $arqmerge0) or ($arqmerge2 == $arqmerge1))
    {
        $arqmerge2 = int(rand(4));
    }

    while (($arqmerge3 == $arqmerge0)or($arqmerge3 == $arqmerge1)or($arqmerge3 == $arqmerge2))
    {
        $arqmerge3 = int(rand(4));
    }
}
```

A.2 *holdout*

```
#! /usr/bin/perl

#   Este programa faz a partição de um arquivo fornecido em 2 subarquivos segundo uma
# porcentagem pré-definida pelo usuário.
#   A porcentagem informada pelo usuário será a do arquivo de treinamento.
#   Para realizar essa tarefa carregamos o arquivo em um array e dele tiramos o número
# de linhas.
#   Uma vez com o número de linhas, basta calcular o número de linhas do arq. de treinamento
# e do arq. de teste e fazer as devidas  transferêncuas.

# Leitura do terminal, do nome do arquivo, da porcentagem e do diretório de destino.

$nomearq   = $ARGV[0];
$porcent   = $ARGV[1];
$diretorio = $ARGV[2];

# Verfifcando se todos os parâmetros são nulos

if ($nomearq eq "")
{
  print("\nForma: holdout [arq. entrada] *[porc. arq. treinam.] **[diret. de destino]\n\n");
  print("* = (1% - 100%)\n");
```

```
  print("** = caminho completo de um diretório já existente onde gerar os arquivos
  (inclua no final /).\n");
  die("Se nada é informado, os arquivos são gerados no diretório corrente.\n");
}

# Verificando se a porcentagem foi fornecida
if (($porcent eq "") or ($porcent == 0))
{
    die ("Forneça a porcentagem do arquivo de treinamento!");
}

# Chamada do programa "mix" para embaralhar o arquivo original

system ("./mix $nomearq arqaux");

# Abertura do arquivo "misturado" para a realização do holdout.
unless (open(original,"arqaux"))
{
    if (-e "arqaux")
    {
        die("Arquivo arqaux existe mas não pode ser aberto!");
    }
    else
    {
        die("Arquivo arqaux não existe!");
    }
}

# Carregando o arquivo em um vetor para extrair o número de linhas.

@vetarq = <original>;

# Extraindo o número de linhas do vetor.

$numlinhas = @vetarq;

# Calculando o número de linhas do arquivo de treinamento.

$treinamento = ($numlinhas * $porcent)/100;
$treinamento = int ($treinamento);


#  Gerando os arquivos de treinamento e de teste

# Abrindo os arquivos de treinamento e teste a serem gerados.
```

```
unless (open(treinar, ">$diretorio$nomearq.data"))
{
    die("Arquivo 'treinamento' não pode ser aberto !");
}

unless (open(test, ">$diretorio$nomearq.test"))
{
    die("Arquivo 'test' não pode ser aberto !");
}

# Gerando o arquivo de treinamento

$i = 0;

while ($i < $treinamento)
{
    print treinar ($vetarq[$i]);
    $i = $i + 1;
}

# Gerando o arquivo de testes

while ($i < $numlinhas)
{
    print test ($vetarq[$i]);
    $i = $i + 1;
}

# Fechando todos os arquivos

close (treinar);
close (test);
close (original);

# Removendo o arquivo auxiliar

system ("rm arqaux");
```

A.3 *rsampling*

```
#! /usr/bin/perl

# Este programa implementa o método estatístico random subsampling.
```

```
# Tal método consiste tão somente da execução por k-vezes do método
# holdout. Com isso teremos k-iterações totalmente independentes para
# treinamento e teste.
# Vale lembrar que se o diretório não informado um default rs indexado com um
# índice numérico.

# Leitura do terminal, dos nomes dos arquivos

$k =  $ARGV[0];
$porc = $ARGV[1];
$nomearq = $ARGV[2];
$diretorio = $ARGV[3];

# Verfifcando se todos os parâmetros são nulos
if (($k eq "") or ($k == 0))
{
    print ("nForma: rsampling [num. iter.] [porc. arq. trein.] [arq. entrada]
    *[dir. saída]\n\n");
    die("* = se não informado, subdiretórios default devidamente indexados
    com o número da iteração serão gerados)\n" );
}

if (\$diretorio eq "")
{
    diretorio = "./rs";
}

# Criação de k subdiretórios para geração dos k arquivos
for($i = 0 ; $i < $k; $i++)
{
    system("mkdir $diretorio$i");
}

# Gerando os arquivos de treinamento e de teste dentro dos respectivos subdiretorios

$i = 0;

while ($i < $k)
{
    system("./holdout $nomearq $porc $diretorio$i/ ");
    $i = $i + 1;
}
```

A.4 *kfold*

```
#! /usr/bin/perl

#   Este programa faz a partição de um arquivo fornecido em "k" subarquivos e
# gera depois, usando-as, "k" séries diferentes de dois arquivos, um para
# treinamento e outro para teste.
#   Este programa deve gerar todas as combinações possíveis de arquivos para
# teste e treinamento(que nesse caso serão k).
#   A ordem das entradas serão: k, Arquivo de entrada, diretório de saída.
#   Nota: ao nome do diretório de saída será indexado um índice que vai de
#   0 a k-1.


# Leitura do terminal, dos nomes dos arquivos

$k = $ARGV[0];
$nomearq = $ARGV[1];
$diretorio = $ARGV[2];

# Variável auxiliar usada como parâmetro na chamada de subrotinas.
$nome;

# Verfifcando se todos os parâmetros são nulos
if (($k eq "") or ($k == 0))
{
    print ("\nForma: kfold [num. partições] [arq. entrada] *[dir. saída]\n\n");
    print("* = informe o caminho completo do diretório a ser criado.\n");
    die("Se nada for informado subdiretórios default indexados são gerados.\n" );
}

# Chamada do programa "mix" para embaralhar o arquivo original

system ("./mix $nomearq arqaux");

# Abertura do arquivo original para a realização do fold.
unless (open(original,"arqaux"))
{
    if (-e "arqaux")
    {
        die("Arquivo arqaux existe mas não pode ser aberto!");
    }
    else
    {
        die("Arquivo arqaux não existe!");
```

```
    }
}

# Criação de k subdiretórios para geração dos k arquivos

$diretorio = &diretorios ($k, $diretorio);

# Abertura de k arquivo auxiliares para a geração dos arquivos definitivos.

$nome = "aux";
&abre_n ($k, $nome);

# Gerando k subarquivos aux's para a realização do k-fold. Para a geração desses
# subarquivos, lemos a primeira linha do arquivo "original" e copiamos para
# o arquivos aux1, a segunda para aux2 e assim sucessiva e ciclicamente.

&particiona ($k, $nome);

# Gerando os arquivos de treinamento e de teste dentro dos respectivos  subdiretorios

# Fechando todos os arquivos auxiliares e em seguida reabrindo-os

$nome = "aux";
&fecha_n ($k, $nome);
&abrel_n ($k, $nome);

# Começando um loop para os arquivos

for ($i = 0; $i < $k; $i++)
{

# Abrindo os arquivos necessários para o uso

    unless (open(treinar, ">$diretorio$i/$nomearq.data"))
    {
        die("Arquivo 'treinamento' não pode ser aberto !");
    }
    unless (open(test, ">$diretorio$i/$nomearq.test"))
    {
        die("Arquivo 'test' não pode ser aberto !");
    }

    # Gerando o arquivo de teste

    $concat = aux.$i;
    $line = <$concat>;
```

```
    while ($line ne "")
    {
        print test ($line);
        $line = <$concat>;
    }

    # Gerando o arquivo de treinamento

    $aux = 0 ;
    while ($aux < $k)
    {
        if ($aux == $i) # descarto o aux que ja foi usado (o de testes)
        {

            $aux = $aux + 1;
        }
        else
        {
            $concat = aux.$aux;
            $line = <$concat>;
            while($line ne "")
            {
                print treinar ($line);
                $line = <$concat>;
            }
            $aux = $aux + 1;
            }
    }

    # É preciso fazer o ponteiro do arquivo voltar a apontar para o seu comeco.
    # Uso o comando "seek( , , )".

    $aux = 0;
    while ($aux < $k)
    {
        seek(aux.$aux, 0, 0); # Volto para o comeco do arquivo.
        $aux = $aux + 1;
    }
    close (treinar);
    close (test);
}

#Fechando os arquivos restantes

$nome = "aux";
&fecha_n ($k, $nome);
```

```
close (original);

# Removendo os arquivos auxiliares
$aux = 0;
while($aux < $k)
{
    system ("rm ./aux$aux");
    $aux = $aux + 1;
}
system ("rm arqaux");


#  S U B R O T I N A S

# Rotina que abre k arquivos para escrita, com nome dado por $nome e índices que
# vão de 0 à $k - 1.
sub abre_n
{
    my ($k, $nome) = @_;
    my ($aux);
    for ($aux = 0; $aux < $k; $aux++)
    {
        unless (open ($nome.$aux, ">$nome".$aux))
        {
            die("Arquivo $nome.$aux não pode ser aberto! \n");
        }
    }
}

# Rotina que realiza a mesma função da rotina acima, com a diferença de que abre
# os arquivos para leitura.
sub abrel_n
{
    my ($k, $nome) = @_;
    my ($aux);
    for ($aux = 0; $aux < $k; $aux++)
    {
        unless (open ($nome.$aux, "$nome".$aux))
        {
            if (-e "$nome".$aux)
            {
                die("Arquivo $nome$aux existe mas não pode ser aberto! \n");
            }
            else
            {
                die("Arquivo $nome$aux não existe! \n");
```

```
            }
        }
    }
}

# Rotina que fecha k arquivos de nome $nome seguindo a definições acima
# para $k e $nome.
sub fecha_n
{
    my ($k, $nome) = @_;
    my ($aux);
    for ($aux = 0; $aux < $k; $aux++)
    {
        close($nome.$aux);
    }
}

# Rotina que cria os subdiretórios para armazenar os arquivos gerados pelo programa.
sub diretorios
{
    my ($k, $diretorio) = @_;
    my ($i);

    if ($diretorio eq "")
    {
        $diretorio = "./it";
    }

    for($i = 0; $i < $k; $i++)
    {
        system("mkdir $diretorio$i")
    }
    $diretorio;  #retorno da função
}

# Rotina que particina igualmente a base de dados em K arquivos auxiliares
sub particiona
{
    my ($k, $nome) = @_;
    my ($line, $i);

    $line = <original>;
    $i = 0;
    while($line ne "")
    {
        $concat = $nome.$i;
```

```
        print $concat ($line);
        $line = <original>;
        $i = $i + 1;
        if ($i >= $k)
        {
            $i = 0;
        }
    }
}
```

Referências

[Baranauskas and Monard, 2000] Baranauskas, J. A. and Monard, M. C. (2000). An unified overview of six supervised symbolic machine learning inducers. Technical Report 103, ICMC-USP, São Carlos, SP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_103.ps.zip.

[Breiman et al., 1984] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth & Books, Pacific Grove, CA.

[Clark and Niblett, 1989] Clark, P. and Niblett, T. (1989). The CN2 Induction Algorithm. *Machine Learning*, 3(4):261–283.

[Dietterich, 1997] Dietterich, T. G. (1997). Statistical Tests for Comparing Supervised Classification Learning Algorithms. ftp://ftp.cs.orst.edu/pub/tgd/papers/stats.ps.gz.

[Haykin, 1994] Haykin, S. (1994). *Neural Networks A Comprehensive Foundation*. College Publishing Company Inc.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A Logical Calculus of the Ideas Immanet in Nervous Activity. *Bulletim of Mathematical Biophysics*, 5:115–133.

[Merz and Murphy, 1998] Merz, C. J. and Murphy, P. M. (1998). UCI Repository of Machine Learning Datasets. http://www.ics.uci.edu/ mlearn/MLRepository.html.

[Michalski et al., 1986] Michalski, R., Mozetic, I., Hong, J., and Lavrac, N. (1986). The Multipurpose Incremental Learning System AQ15 and Its Testing Application to Three Medical Domains. In *Proceedings of the Fifth Annual National Conference on Artificial Inteligence*, pages 1041–1045, Philadelphia.

[Michie and Spiegelhalter, 1994] Michie, D. and Spiegelhalter, D. J. (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, 1:81–106. Reprinted in Shavlik and Dieterich (eds.) Readings in Machine Learning.

[Quinlan, 1987] Quinlan, J. R. (1987). Simplifying Decision Trees. *International Journal of Man-Machine Studies*, 27:221–234.

[Quinlan, 1988] Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning*. Morgan Kaufmann, CA.

[Quinlan, 1998] Quinlan, J. R. (1998). Generating Production Rules from Decision Trees. In *Proceedings of the Thent International Joint Conference on Artificial Inteligence*, pages 304–307, Italy.

[Rulequest-Research, 1999] Rulequest-Research (1999). Data Mining Tools See5 and C5.0. http://www.rulequest.com/see5-info.html em 03/1999.

[Rumelhart and McClelland, 1986] Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel Distributing Processing: Exploration in the Microstruture of Cognition*. MIT Press, Cambridge.

[Salzberg, 1997] Salzberg, S. L. (1997). On Comparing Classifiers: Pitfalls to Avoid and a Recommended Approach. http://www.cs.jhu.edu/~salzberg/critique.ps.

[Till, 1996] Till, D. (1996). *Teach Yourself Perl 5 in 21 Days.* Sams Publishing.

[Weiss and Indurkhya, 1995] Weiss, S. M. and Indurkhya, N. (1995). Rule-Based Machine Learning Methods for Functional Prediction. *Journal of Artificial Inteligence Research*, 3:383–403.

[Weiss and Kulikowski, 1991] Weiss, S. M. and Kulikowski, C. A. (1991). *Computer Systems that Learn.* Morgan Kaufmann, San Mateo, CA.