ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Detecção de Novidades em Problemas de Classificação com Redes Neurais Artificiais Probabilísticas Construtivas

Trabalho de Conclusão de Curso

Engenharia da Computação

Flávio Roberto Gomes da Costa
Orientador: Prof. Adriano Lorena Inácio de Oliveira

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Detecção de Novidades em Problemas de Classificação com Redes Neurais Artificiais Probabilísticas Construtivas

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Flávio Roberto Gomes da Costa
Orientador: Prof. Adriano Lorena Inácio de Oliveira

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

Flávio Roberto Gomes da Costa

Detecção de Novidade em Problemas de Classificação com Redes Neurais Artificiais Probabilísticas Construtivas

Resumo

Detectar novidades é uma característica importante quando se pensa em aprendizado de máquina. Vários modelos para representação dos dados já foram utilizados em outros trabalhos, para bases de dados artificiais e também extraídas de dados do mundo real. Este trabalho apresenta a avaliação de um algoritmo para treinamento construtivo de Redes Neurais Probabilísticas (PNN–DDA), no trabalho de detecção de novidades, adotando uma nova forma de normalização da saída da rede. Os resultados obtidos a partir do modelo estatístico NNDD são usados para fins de comparação, uma vez que ele é usado como referência em trabalhos similares. Além da nova abordagem, uma técnica de diminuição de complexidade é proposta e avaliada neste trabalho. Para avaliação, tomamos como parâmetro de referência (1) a variação na detecção de padrões novos, e classificação errada de padrões conhecidos e (2) a melhoria da capacidade de generalização das redes neurais obtidas. Para a análise dos resultados deste trabalho, são usadas Curvas ROC, obtidas para cada classificador, bem como o respectivo valor de AUC (*Area Under Curve*). As simulações foram realizadas através de um programa, chamado *Novelty Detector*, desenvolvido neste trabalho, usando a linguagem Java, com os modelos de representação dos dados e as técnicas de treinamento propostas. Os resultados obtidos demonstram que o modelo PNN com poda, proposto aqui, possui uma boa capacidade para detecção de novidades, associada com uma capacidade de generalização melhorada através da diminuição da complexidade final da rede.

Abstract

Detecting novelties is an important issue in machine learning research and practical applications. Some theoretical models for representing information have already been used in others works, for both artificial or real world datasets. This work presents an evaluation of a constructive training algorithm for Probabilistic Neural Networks (PNN-DDA), when detecting novelties, through a new approach for network output normalization. Results from the Nearest Neighbor Data Description (NNDD) are used for comparison, since this model is commonly used in works from this research area. Besides the new approach, a new technique for decreasing complexity is evaluated. For the evaluation, we considered (1) the correct detection of novel patterns and misclassification of known ones, and (2) the improvement of generalization capability of obtained neural networks. For analysis of results, ROC Curves are gathered from each classifier, along with the respective AUC (Area under Curve) value. Simulations were carried out by using software called *Novelty Detector*, developed in this work using the Java language, with data models and the proposed training techniques. The obtained simulation results show that the PNN model with proposed pruning has good novelty detection ability; in addition, it has generalization quality improved through a decrease on the final network complexity.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

IDS – Intrusion Detection System (Sistema para Detecção de Intrusão)

NNDD – Nearest Neighbor Data Description (Descrição de Dados do Vizinho mais Próximo)

PNN – Probabilistic Neural Network (Rede Neural Probabilística)

DDA – Dynamic Decay Adjustment (Ajuste Dinâmico do Decaimento)

ROC – Receiver Operating Characteristic

HMM – Hidden Markov Models (Modelos Ocultos de Markov)

kNN – k-Nearest Neighbor (k-Vizinho mais Próximo)

MLP – Multi-layer Perceptron

RBF – Radial Basis Function (Função de base radial)

SOM – Self-organizing Maps

GMM - Gaussian Mixture Modeling

XOR – eXclusive OR (Ou Exclusivo)

SVM - Support Vector Machine (Máquina de vetor de suporte)

ART - Adaptive Resonance Theory

PD – Probability of Detection (Probabilidade de Detecção)

PFA – Probability of False Alarm (Probabilidade de Alarme Falso)

FP – False Positive (Positivo Falso)

FN – False Negative (Negativo Falso)

TN – True Negative (Negativo Verdadeiro)

FP – False Positive (Positivo Falso)

AUC – Área Under Curve (Área sob a curva)

DDTOOLS – Data Description Tools (Ferramenta para descrição de dados)

JDK – Java Development Kit (Kit para Desenvolvimento em Java)

SNNS – Stuttgart Neural Network Simulator

ASCII – American Standard Code for Information Interchange

API – Application Programming Interface - Interface para Programação de Aplicativos

LGPL - Lesser General Public License

Agradecimentos

"Em tudo daí graças"
Paulo, I Tes. 5:18

Gostaria de agradecer em primeiro lugar a Deus, por em tudo ter colocado um pouco da Sua grandeza, nos permitindo descortinar o conhecimento como crianças brincando com conchas na praia, alheias ao grande oceano a ser descoberto[1].

Passando por esta etapa, gostaria de agradecer a todos aqueles que residem em meu coração como família, por todo incentivo, apoio e amparo nos diversos momentos durante minha vida, em especial na universidade. Minhas constantes e sinceras desculpas à minha esposa Cláudia, e meus dois mais bem sucedidos experimentos, Guilherme e Lucas, por toda ausência sentida por eles.

Agradecimento especial tem de ser feito a todos os colegas do Tribunal Regional Eleitoral, que sempre me deram apoio logístico, nas infindáveis horas de preparação das Eleições.

A todo o colegiado do DSC, pelo apoio pessoal, nas figuras dos Professores Carlos Alexandre e Ricardo Massa, também cobaias, e pelo belo trabalho feito para o crescimento do Curso, do Departamento e da Faculdade. Obrigado e Parabéns!

Ao meu orientador Adriano Lorena, por além de direcionado o trabalho, ter incentivado seu nascimento e execução. Agradeço a confiança depositada em mim.

A todos os colegas das várias turmas por onde passei. Em especial a Laureano e José Guilherme, companheiros cobaias das primeiras aventuras, e Moacir, amigo das sinapses aleatórias.

[1] Parafraseando Isaac Newton, gênio inglês.

Capítulo 1

Introdução

"Ancora Imparo" (Ainda estou aprendendo)
Michelangelo Buonarroti

1.1 Motivação

Considerando à parte as discussões filosóficas sobre o que é inteligência, podemos afirmar que um sistema dotado da capacidade de pensar deve estar apto a se adaptar a situações consideradas novas, alterando seu comportamento através de experiências ou práticas, e armazenando esse novo conhecimento. Assim, esse é um ponto chave: um sistema inteligente deve ser capaz, entre outras coisas, de discernir a informação recebida, classificando-a como já existente, ou considerá-la novidade.Contudo, em problemas de classificação não existem critérios pré-definidos para saber se uma situação é nova ou pode ser classificada como algo já existente.

Uma das possíveis aplicações de detecção de novidades é na área de detecção de ataques em redes de computadores. Sistemas de Detecção de Intrusão (*Intrusion Detection Systems* - IDS) são responsáveis por detectar e relatar atividades maliciosas agindo em computadores e recursos da rede [8]. Para execução dessa atividade, existem duas formas de ação: detecção de mau-uso e detecção de anomalias. Para o primeiro caso, se usa informações de ataques já conhecidos, para barrar essas ocorrências.

Para localizar atividades maliciosas, normalmente se analisa o tráfego comum da rede a fim de se obter um padrão de tráfego normal. Esta informação é usada como base para se localizar uma intrusão. Para isso, é essencial que haja algum modelo adequado para representar a informação conhecida e definir fronteiras limítrofes para separação dos dados considerados novos.

Da mesma forma, outras áreas como aquelas de sistemas usados para detecção de falhas em funcionamento de máquinas, análise de imagens, reconhecimento de objetos em vídeos [17] [18], segmentação de dígitos manuscritos [20] [24] e controle de tráfego aéreo, por exemplo, são

áreas onde a capacidade de identificar novos objetos, isto é, objetos que não pertencem a nenhuma das classes existentes na fase de treinamento, é muito importante e pode melhorar o trabalho de classificação, bem como os resultados obtidos.

1.2 Objetivos e Trabalhos Relacionados

O presente trabalho tem por objetivo contribuir com a área de detecção de novidades, avaliando o uso de Redes Neurais Probabilísticas Construtivas (PNN-DDA) [2] para a modelagem de dados em problemas de classificação, com ênfase à detecção de padrões desconhecidos (novidades). Apesar de já ter apresentado bons resultados para problemas de classificação, esse modelo ainda não foi avaliado na área em questão, principalmente com a abordagem proposta por Berthold e Diamond para a normalização da saída da rede.

Além disso, pretende-se avaliar o impacto obtido com a aplicação de uma técnica recente proposta por Perfetti e Ricci [26] para diminuição da complexidade, ou o número de elementos constituintes das redes neurais obtidas após o treinamento. Tal diminuição é bem vinda especialmente por conta da necessidade de recursos computacionais para a implementação. E como cada vez mais é prática comum desenvolver sistemas voltados a dispositivos móveis e as limitações de hardware são comuns nesses tipos de aparelhos, a complexidade das redes finais pode ser usada como função de custo a ser minimizada para obtenção das redes neurais.

Como redes neurais probabilísticas, treinadas de forma construtiva, conseguem produzir modelos de forma otimizada [2]; isso é um fato indicativo de que o modelo pretendido para representação é adequado para os objetivos propostos.

Outra linha de pesquisa, seguida por Harmeling *et al.* [9], propõe uma nova forma de modelagem mais adequada para dados com múltiplas dimensões, em que são definidas três novas variáveis κ, γ e δ, responsáveis por aprimorar o modelo estatístico *k-Nearest Neighbor* (k-NN) [4]. Os resultados obtidos pelos idealizadores do trabalho confirmaram que as variáveis propostas podem ser usadas com sucesso para modelagem de dados do mundo real, que são multi-variáveis por natureza, e ser utilizadas para o trabalho de detecção de novidades.

Além de objetivos semelhantes, a forma adotada para avaliação dos modelos é a mesma que a do trabalho atual, as curvas ROC [6] [13]. Essa técnica já demonstrou ser um bom critério para comparação de classificadores, fornecendo dados importantes ao pesquisador.

1.3 Contribuições

O presente trabalho apresenta as seguintes contribuições às pesquisas na área de detecção de novidades:

1. Avaliação das redes PNN-DDA para detecção de novidades, incluindo a avaliação da influência do parâmetro de treinamento θ- no desempenho;

2. Proposta e avaliação de um novo método para detecção de novidades baseado em PNN-DDA com poda;
3. Desenvolvimento de um simulador para detecção de novidades em Java, considerando as técnicas: (a) PNN-DDA; (b) PNN-DDA com poda (proposta neste trabalho); e (c) NNDD;
4. Avaliação experimental das técnicas propostas e comparação com NNDD.

1.4 Estrutura do Trabalho

O presente trabalho está estruturado em sete capítulos. Este Capítulo apresenta a motivação para o desenvolvimento do trabalho, bem como os objetivos a que se propõe.

O Capítulo 2 apresenta a área de detecção de novidades, incluindo os princípios básicos a serem seguidos por modelos de dados. Além disso, são apresentados outros modelos já utilizados nessa área de pesquisa. O terceiro Capítulo se atém a dois dos modelos existentes, o NNDD e o PNN-DDA, por serem os escolhidos para execução das simulações. Também é apresentada a técnica proposta neste trabalho para redução de complexidade de redes neurais.

O Capítulo 4 traz uma explanação sobre Curvas ROC, técnica utilizada para comparação de classificadores. O quinto Capítulo apresenta detalhes sobre a implementação do simulador desenvolvido como fruto do presente trabalho.

O Capítulo 6 mostra os resultados obtidos com as simulações e realiza uma discussão, comparando-os. O Capítulo 7 apresenta as conclusões do trabalho, trazendo algumas sugestões de ações a serem tomadas como trabalhos futuros.

Capítulo 2

Detecção de Novidade

2.1 Caracterização do Problema

A capacidade de detectar novidades é uma característica desejável em sistemas responsáveis por tarefas de classificação. Um classificador, por exemplo, que analisa faces e é responsável pelo controle de acesso a um ambiente, deve, além de classificar corretamente os rostos previamente cadastrados, identificar faces desconhecidas, impedindo o acesso.

De modo geral, como o mundo vive em constante mudança, não podemos prover um sistema com todas as informações necessárias para garantir uma perfeita e eterna capacidade de classificação. Assim, saber tratar dados novos, aprender, é importante principalmente no trato com dados do mundo real, pois nem sempre é possível conhecer, a priori, todas as informações necessárias para o treinamento de classificadores. Alguns exemplos de sistemas em que é necessário trabalhar com dados cujas classes não estavam presentes na fase de treinamento são: sistemas de detecção de falhas em máquinas [15], sistemas de detecção de objetos em radar [15], e segmentação de dígitos manuscritos [24]. Outro exemplo envolve a detecção de intrusão em redes de computadores, área de pesquisa que gera inúmeros interesses comerciais por parte de empresas [8].

Porém, além de importante, detectar novidades se mostra uma atividade complexa. Não existe um modelo completo, capaz de detectar novidades em qualquer meio. Diferentes modelos já foram testados em diversas bases de dados, sendo os resultados dependentes também das propriedades estatísticas das informações a serem representadas. Além disso, não existem bases de dados prontas com novidades, pois erros e falhas são eventos raros e a existência dessa informação seria equivalente a "conhecer o desconhecido". Para contornar as inúmeras dificuldades que envolvem a pesquisa nessa área, foi desenvolvida a idéia de modelagem dos dados normais, conhecidos, e definição de alguma função de distância, bem como um limiar, para localizar anormalidades. Contudo, essa abordagem tende a atingir níveis altos de complexidade

em virtude da multiplicidade de classes, alta dimensionalidade dos dados, "ruídos" na informação e, geralmente, quantidade pequena de dados para análise, situações comuns para problemas do mundo real.

Por fim, alguns aspectos importantes para sistemas de detecção de novidade devem ser considerados [15]:

- **Princípio da robustez e ajustabilidade**: um método de detecção de novidade deve ser robusto o suficiente para detectar, o mais corretamente possível, padrões novos, bem como não considerar novo um padrão conhecido. Além disso, essa robustez deve ser previsível e controlável pelo experimentador;
- **Princípio dos parâmetros mínimos**: um método deve possuir o menor número possível de parâmetros para ajuste do usuário;
- **Princípio da generalização**: o sistema deve ser capaz de fazer generalizações, mas sem confundir com padrões novos;
- **Princípio da complexidade computacional**: a maior parte das aplicações de detecção de novidades possui execução *on-line*, assim, os sistemas devem ser tão simples quanto possível.

Essas são algumas das características que devem ser seguidas no desenvolvimento de modelos para detectar novidades.

2.2 Abordagens Existentes

Para modelar os dados, várias abordagens já foram feitas através de métodos estatísticos como HMM (*Hidden Markov Models*) [4], mistura de funções Gaussianas [4], kNN (*k-Nearest Neighbor*) [4], e através do uso de algumas arquiteturas de redes neurais como MLP (*Multi-layer Perceptron*) [4], RBF (*Radial Basis Function*) [4] ou SOM (*Self-organizing Maps*) [4].

2.2.1 Métodos Estatísticos para Detecção de Novidade

A idéia principal por trás da abordagem estatística é modelar os dados de treinamento e usar essa informação para definir se padrões de teste pertencem ou não à mesma distribuição. A técnica a ser usada vai depender principalmente das propriedades estatísticas dos dados, bem como de aspectos como qualidade e quantidade dos dados existentes para a modelagem.

Métodos estatísticos podem ser parametrizados ou não parametrizados. A abordagem paramétrica assume uma distribuição conhecida para os dados, como a normal, e procura calcular os parâmetros do modelo, como valores padrões e covariância, para que os dados de treinamento sejam explicados da melhor forma. Essa abordagem é simples de ser utilizada, pois já parte de um princípio, que é a distribuição dos dados, fazendo com que o trabalho se restrinja ao cálculo dos parâmetros. Porém, para dados do mundo real, ela tende a se tornar complicada, pois

normalmente a estrutura dos dados (distribuição de probabilidade) é pouco conhecida, e não segue uma distribuição simples.

Nos métodos não parametrizados não há a necessidade de se assumir uma distribuição de probabilidade específica para os dados, pois ela é obtida através da sua própria estrutura, bem como os parâmetros que definem o modelo. Essa abordagem provê uma maior flexibilidade à modelagem, especialmente em dados reais, mas requer um maior custo computacional. Contudo, em ambas as abordagens, é necessária a definição de um limiar, responsável pela definição da classe dos padrões de teste. Esse limiar deve ser ajustado pelo pesquisador para garantir um melhor acerto na detecção de novidades, e uma baixa taxa de erro, ou de alarmes falsos.

Alguns exemplos de métodos que usam uma abordagem paramétrica são:

- GMM (*Gaussian Mixture Modeling*) [4], onde alguns núcleos baseados em funções gaussianas, em quantidade menor que o montante de dados de treinamento, são ajustados para representar a distribuição de probabilidade dos dados. Caso a dimensão dos dados seja alta, uma quantidade maior de informações é necessária para a execução do treinamento;
- HMM (*Hidden Markov Models*) [4], que é um método estocástico que pode ser usado para modelar dados que possuem estrutura temporal, que sejam conectados, ou seja o estado *t* depende de *t - 1* [4], como fonemas em conversas, ou seqüências de comandos em sistemas computacionais;
- Teste de Hipótese [4], onde é avaliado se a distribuição dos dados de teste seguem a mesma estrutura dos dados de treinamento.

Entre os métodos não parametrizados, temos os seguintes exemplos:

- kNN (*k-Nearest Neighbor*) é uma técnica que procura estimar a função de densidade dos dados a serem modelados e usar essa informação para definir a classe dos padrões de teste. Para isso, dois conceitos importantes são usados, distância e vizinhança entre padrões. Eles são melhores explorados mais adiante;
- *Parzen Windows* [4], onde a estimativa da função de densidade dos dados é feita a partir de núcleos para cada dado de treinamento, extrapolando aos dados de teste;
- Abordagens baseadas em *Strings* [4], em que os dados de treinamento são modelados como seqüências de caracteres e alguma medida compara os dados de treinamento e teste. Essa técnica permite o uso de algoritmos genéticos para essa análise;
- Abordagens baseadas em *Clusters* [4], é um método onde é feita uma partição dos dados em *clusters,* e um grau de inclusão é associado a cada padrão. Se for estipulado um limiar para a inclusão no *cluster*, os padrões novos serão aqueles que não atingirem o valor mínimo em nenhuma das classes.

Todas as soluções apresentadas já foram usadas em diversas pesquisas e constituem importante área de interesse para detecção de novidade.

2.2.2 Uso de Redes Neurais em Detecção de Novidade

Redes neurais [4] podem ser definidas como a união estruturada de elementos básicos chamados neurônios que possuem funções de ativação não lineares. A união dos elementos produz um efeito global mais complexo na rede, seguindo o princípio conexionista da área de inteligência artificial, onde processos sofisticados emergem da interconexão de elementos simples. Um exemplo da Neurociência vem do funcionamento do cérebro, onde a interconexão de neurônios biológicos é capaz de produzir fenômenos como a mente e o pensamento humanos.

A forma de interconexão e a função de ativação dos neurônios definem o tipo da rede neural. Redes MLP (*Multi-layer Perceptron*) [4] são constituídas através de entidades de processamento simples chamadas Perceptrons [4], interligadas em três camadas, normalmente, com conexões em uma única direção de um neurônio de uma camada para a seguinte, conforme apresentado na Figura 2-1. O diagrama apresenta uma rede MLP conectada e treinada para fornecer a resposta da função binária XOR entre duas variáveis *x* e *y*.

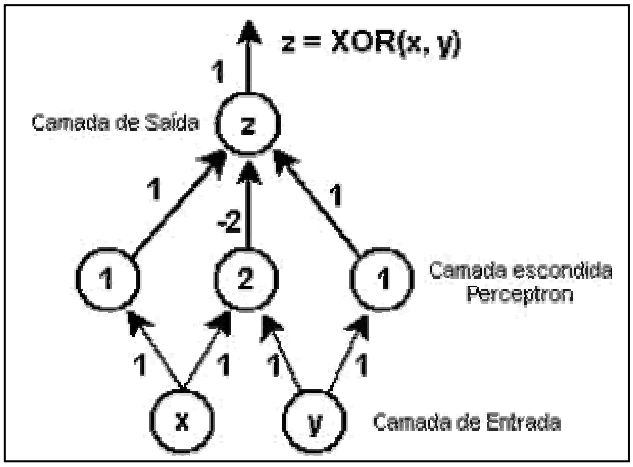

Figura 2-1. Diagrama básico de uma rede neural MLP fornecendo a resposta da função XOR como saída.

Além das características já apresentadas, outro fator é fundamental para diferenciar as redes neurais: a forma de treinamento. As técnicas podem ser classificadas em três categorias, supervisionadas, não supervisionadas e por reforço.

Na área de detecção de novidade, várias arquiteturas já foram analisadas em diversas aplicações diferentes. Algumas características importantes das redes neurais fazem com que essa abordagem seja diferente dos métodos estatísticos, tais como: sua capacidade de generalização, uma maior demanda computacional para treinamento, e o custo no caso de agregar mais informações após a conclusão do treinamento. Porém, podem ser vistas como vantagens o

pequeno número de parâmetros necessários para a realização do treinamento das redes, e a ausência da necessidade de assumir propriedades dos dados a serem usados.

Algumas arquiteturas e métodos de redes neurais já utilizados para detectar novidades estão listados abaixo.

- MLP, onde algumas adaptações devem ser feitas para adequar o modelo ao trabalho de detecção de novidade, uma vez que seu uso é mais voltado à classificação comum;
- Máquina de Vetor de Suporte (*Support Vector Machine* – SVM) [4], que é baseada na idéia de utilizar hiperplanos para separar classes distintas entre si. Essa separação pode ser usada para encapsular os dados apresentados durante o treinamento das demais informações existentes no universo de possibilidades;
- ART (*Adaptive Resonance Theory*) [4], que consegue gerar classificadores específicos para detectar novidades;
- Redes com funções de bases radiais (RBF - *Radial basis function*) [4], onde a ativação dos neurônios é definida pela distância entre o padrão fornecido e o centro de cada elemento.

2.3 Abordagens Propostas

Neste trabalho são apresentados dois métodos para modelagem no problema de detecção de novidades: um método estatístico que toma como referência a idéia do vizinho mais próximo (NNDD) e um método com base em redes neurais, utilizando uma modelagem através de redes neurais probabilísticas (PNN-DDA). Além disso, é proposto um novo método para detecção de novidades, que combina PNN-DDA com poda.

2.3.1 *Nearest Neighbor Data Description* (NNDD)

Quando não existe uma descrição completa da distribuição probabilística dos padrões existentes, apenas o conhecimento obtido a partir de informações pré-existentes, a decisão entre classificar um padrão **x** como pertencente a uma classe **A** vai depender tão somente da correta classificação de padrões dessa classe. Esse é o foco principal do método estatístico não-paramétrico chamado *Nearest Neighbor Data Description* (NNDD), que trabalha com o conceito de vizinhança entre padrões para obter as informações necessárias para a classificação.

A idéia de vizinhança é um conceito simples que pode ser introduzido através de um exemplo da geometria Euclidiana. Considerando dois pontos no espaço tridimensional **q** e **r**, localizados nas coordenadas (x_q, y_q, z_q) e (x_r, y_r, z_r), respectivamente, pode-se calcular a distância Euclidiana entre eles através da equação (1).

$$Dist_{AB} = \sqrt{(x_q - x_r)^2 + (y_q - y_r)^2 + (z_q - z_r)^2} \qquad (1)$$

Tomando uma classe A, se for necessário saber se um padrão x pertence a essa classe, podemos considerar o conjunto de pontos presentes na classificação. Calculando as distâncias de **x** a esses pontos, se **x'** (**x'** ∈ A) for um ponto com o menor valor para a distância euclidiana, será considerado que **x** também pertence à classe A. Assim, **x'** é o vizinho mais próximo do padrão **x**. Essa regra é conhecida como regra do vizinho mais próximo. A Figura 2-2 apresenta um exemplo onde a regra é ampliada para avaliar a classe não somente do vizinho mais próximo, mas de um número **v** deles. Assim, a classe que possuir maior número de vizinhos será definida para o padrão desejado. No exemplo mostrado, para **v** = 3, o círculo será definido como pertencente à classe A, pois dois dos seus três vizinhos mais próximos estão nessa classe.

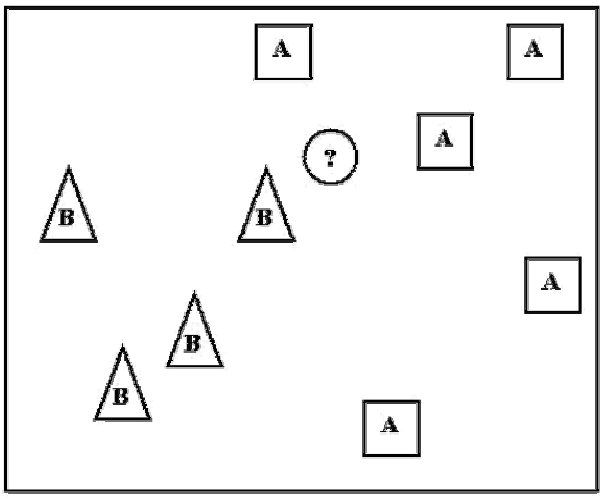

Figura 2-2. Classificação de padrão através do método do vizinho mais próximo, considerando três vizinhos.

Neste trabalho, é usada um abordagem ligeiramente diferente daquela apresentada na Figura 2-2, por se trabalhar num problema de detecção de novidade. Nessa situação, apenas duas classes existem, a classe conhecida e a novidade. Assim, o problema consiste em saber se um padrão **x** pertence à mesma classe dos elementos já analisados. Se for utilizada a idéia acima, sempre haverá uma resposta positiva à classificação do padrão como conhecido, pois qualquer vizinho pré-existente já é conhecido. Para isso, a técnica apresentada por Tax em sua Tese de Doutorado [29] é implementada neste trabalho. Sua idéia consiste em avaliar além da distância euclidiana entre **x** e seu vizinho mais próximo (chamado de **x'**), também a distância entre **x'** e seu vizinho mais próximo, chamado de **x''**. Se a razão entre essas distâncias for menor que um determinado parâmetro **k**, **x** é considerado conhecido, caso contrário, **x** será tratado como novidade. A Figura 2-3 ilustra a técnica, mostrando um padrão classificado como novidade. Normalmente, o valor de **k** é tomado igual a 1, a fim de manter a sensitividade local do método. Contudo, a fim de se permitir a avaliação da capacidade de detecção de novidade do modelo NNDD, uma variação do valor desse parâmetro será realizada entre dois limiares, definidos

empiricamente para este trabalho como iguais respectivamente a 0,0 (valor em que todos os padrões diferentes dos utilizados para o treinamento são considerados novos) e 6,0 (onde nenhum padrão é considerado novidade). Essa faixa de valores foi escolhida por representar os dois extremos possíveis de detecção. Foi possível avaliar que nas bases de dados usadas neste trabalho, o espalhamento dos dados no espaço é tal que nenhum padrão possui distância euclidiana ao vizinho mais próximo que produza um valor de **k** maior que 6,0. Adotando valores superiores ao adotado, não alteraria os resultados obtidos, apenas aumentaria o tempo de execução das simulações.

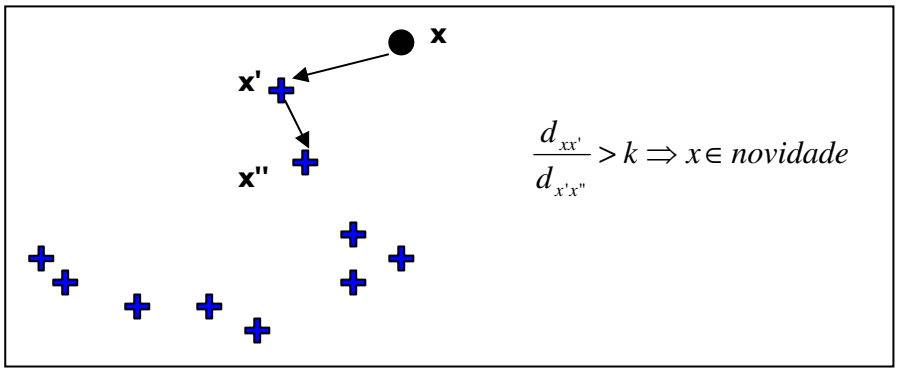

Figura 2-3 Exemplo de classificação de um padrão x, com o método proposto por Tax.

2.3.2 Redes Neurais Probabilísticas

Além dos métodos de origem estatística, as redes neurais são amplamente usadas na área de detecção de novidades, como já visto no Capítulo anterior. Berthold e Diamond [2] propuseram uma nova forma de treinamento para redes neurais probabilísticas, bem como uma maneira diferente de normalização da saída obtida, sugerindo a aplicação de suas idéias em problemas onde houvesse a necessidade de detectar novidades, além de classificar objetos em classes conhecidas.

Estrutura da Rede

As chamadas Redes de Funções com Bases Radiais (*Radial Basis Function Networks* - RBF) são estruturadas em unidades de processamento localmente ajustadas. Os neurônios são definidos a partir de funções gaussianas centrada em um ponto específico no espaço das informações tratadas e a sua distribuição se dá numa única camada escondida. Uma segunda camada se encarrega de receber os resultados de cada neurônio e realizar uma soma ponderada entre eles, atribuindo pesos às conexões entre essa camada e a primeira [1]. As redes RBF são a base para o desenvolvimento de uma série de modelos capazes de fornecer uma melhor estrutura para compreensão do sistema modelado.

As Redes Neurais Probabilísticas (*Probabilistic Neural Networks* - PNN) são um tipo especial de rede RBF introduzida em 1990 por Specht [2] [28], que permitem fazer a associação

entre a estrutura da rede e funções de densidade de probabilidades, bem como consegue produzir resultados melhores que obtidos através de outros classificadores.

O objetivo principal do modelo PNN é fornecer em sua saída a probabilidade do padrão recebido pertencer a cada uma das classes, quantidade que será denotada por ***p*(classe *k*|*x*)**, ou seja, a probabilidade do padrão **x** pertencer à classe **k**. A Figura 2-4 apresenta a arquitetura básica da rede PNN. Como uma rede RBF comum, os padrões são fornecidos como vetores $(x_1,..., x_n)$ de dimensão igual a *n*. Após ser recebido pelo neurônio de entrada, o vetor é fornecido a todos os neurônios da primeira camada escondida. Essa camada apresenta $\mathbf{m_k}$ neurônios para cada classe **k**, e cada um deles possui uma função de ativação do tipo gaussiana, que produz a saída $\boldsymbol{p_j^k(x)}$, mostrada na equação (2).

$$p_j^k(x) = \exp\left(-\frac{\left\|x-\mu_j^k\right\|^2}{\sigma_j^2}\right) \qquad (2)$$

Na equação (2), μ_j^k representa o centro do neurônio **j**, e σ_j^k determina o valor de seu desvio padrão. A variável **j** varia entre 1 e $\mathbf{m_k}$, desse modo haverá $\mathbf{m_k}$ distribuições de probabilidade para cada classe **k**.

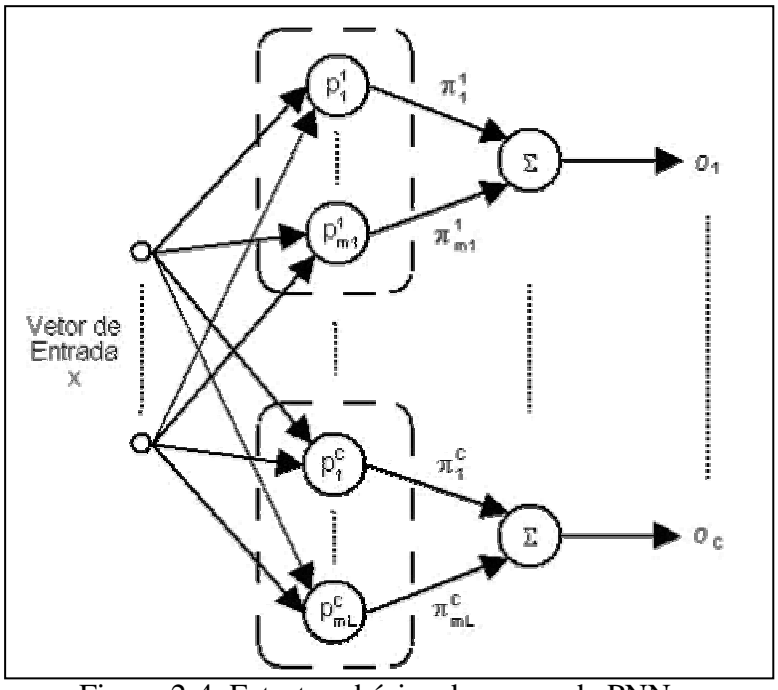

Figura 2-4. Estrutura básica de uma rede PNN

A principal diferença entre as redes RBF comum e as PNNs está no fato de que a saída da primeira camada não é conectada completamente à segunda, pois os neurônios são divididos em classes. Assim, no cálculo mostrado acima, só participam os neurônios de cada classe **k**. Os pesos usados devem ser normalizados, e representam a participação de cada neurônio na "mistura" das

saídas. É possível a definição de uma terceira camada, não mostrada na Figura 2-4, que é opcional e fornece informação sobre o custo da classificação errada de um padrão da classe k numa outra classe l, dando uma função de risco na decisão para o uso da rede [1] [2]. Para os fins deste trabalho, essa terceira camada não será usada.

Treinamento Construtivo

Para simplificar o uso das redes PNN, Berthold e Diamond propuseram uma nova forma de treinamento, usando um procedimento construtivo, ao invés da forma idealizada por Specht que envolve a inclusão de um neurônio para cada padrão de treinamento da rede [28]. A técnica é chamada de Ajuste Dinâmico do Decaimento (*Dynamic Decay Adjustment* - DDA), e seu princípio básico para a construção da rede consiste em inserir neurônios na medida da necessidade, fazendo ajustes nos seus parâmetros para garantir que um mesmo neurônio possa responder de forma satisfatória a mais de um padrão. Assim, cada neurônio escondido da PNN-DDA possui dois parâmetros: o centro (μ); e o raio, que é ajustado dinamicamente para garantir a não classificação de padrões de classe diferente do neurônio.

Para a execução da rotina de treinamento apenas dois parâmetros devem ser definidos: $\theta+$ e $\theta-$. Esses parâmetros se encarregam de definir limiares para a resposta dada pela função gaussiana de cada neurônio. O valor definido como $\theta+$ representa a resposta mínima que um padrão deve gerar pela função gaussiana de um neurônio para ser considerado classificado. A quantidade $\theta-$ indica o valor máximo que um padrão deve gerar num neurônio de classe diferente da sua (conflitante). Essa idéia é usada para garantir uma área de conflito, onde nenhum neurônio deve se encontrar, e a modelagem da separação entre vizinhos conflitantes. A Figura 2-5 apresenta os limiares juntamente com a saída de um neurônio. Como a saída é normalizada, os valores para os limiares estão compreendidos entre 0 e 1.

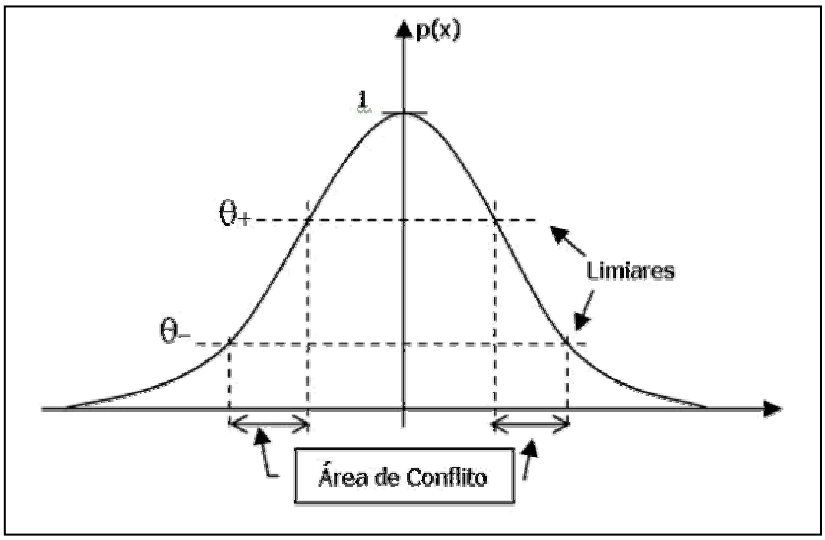

Figura 2-5. Representação gráfica da saída de um neurônio com a indicação dos limiares do treinamento DDA.

A técnica DDA proposta apresenta algumas características [2]:

1. **Treinamento Construtivo**: novos neurônios são adicionados à rede de acordo com a necessidade, durante o treinamento. Tanto a quantidade quanto o raio de cada neurônio são calculados dinamicamente, sem nenhuma definição prévia em relação à estrutura da rede.
2. **Rapidez**: poucas iterações são necessárias para a conclusão do treinamento, normalmente 4 ou 5.
3. **Convergência garantida**: quando se usa um número limitado de padrões para o treinamento, é possível garantir sua convergência.
4. **Simplicidade no treinamento**: apenas dois parâmetros devem ser ajustados pelo usuário para a realização do treinamento.
5. **Zonas de classificação distintas**: com a separação fornecida pelos limiares, após o treinamento temos três possibilidades bem distintas para classificação.
    a. Inclusão da classe - classificação correta possuirá valor acima de $\theta+$
    b. Exclusão da classe - não-classificação produzirá valores abaixo de $\theta-$
    c. Incerteza - padrões presentes nas áreas de conflito serão “classificados” como "novidade"

Essas características tornam o uso das redes PNN interessante em problemas do mundo real, principalmente por não haver a necessidade de definição prévia da estrutura da rede, nem de parâmetros aleatórios ajustados manualmente. Como essas bases de dados tendem a ser grandes e redundantes, o treinamento construtivo e a rapidez tornam mais fortes a propensão ao seu uso.

O algoritmo DDA é apresentado na Figura 2-6 onde os passos principais estão colocados em pseudo-código. Cada época de treinamento produz alterações na rede PNN de forma a incluir neurônios para novos padrões, ou ajustar os já existentes. Ele opera da seguinte maneira:

(1) Antes de qualquer época de treinamento, os pesos existentes são igualados a zero para não haver acúmulo de informações duplicadas entre as épocas;

(2) Todos os padrões **x**, bem como suas classes **k,** são apresentados à rede neural;

(3) Caso haja algum neurônio que classifique corretamente o padrão (3), fornecendo um retorno da função de ativação maior que o limiar $\theta+$, seu peso deve ser incrementado para registrar a informação;

(4) Caso haja mais de um neurônio nessa situação, deve ser incrementado o peso do neurônio que forneça maior retorno;

(5) Caso não haja classificação correta para o padrão, um novo neurônio deve ser incluído na rede;

(6) O centro da função de ativação gaussiana é definido como o centro do novo padrão;

(7) O peso desse neurônio é colocado como igual a 1;

(8) O valor para o raio do neurônio é igual ao maior valor que não classifica como pertencente à classe **k** padrões de classes diferentes. Esse valor é calculado avaliando as distâncias entre os centros do novo neurônio e os demais com classes diferentes, escolhendo-se o menor valor calculado;

(9) Por fim, os neurônios de classe diferente do padrão fornecido devem ser avaliados para garantir a não classificação errônea. Para isso, a distância entre os centros é calculada e comparada com o raio atual do neurônio. Caso haja necessidade, o raio deve ser substituído pelo menor dos valores.

```
        // pesos são definidos como zero
(1)     FORALL neurônios p_i^k DO
              A_i^k = 0,0
        ENDFOR
        // época completa de treinamento
(2)     FORALL padrões de treinamento x, da classe k DO
(3)           IF existir p_i^k : p_i^k > θ+ THEN
(4)                 A_i^k += 1,0
              ELSE
(5)                 // introduzir novo neurônio
                    m_k += 1
(6)                 μ_mk^k = x
(7)                 A_mk^k = 1,0
(8)                 σ_mk^k = min_{l<>k} { sqrt( -||μ_j^l - μ_mk^k||^2 / ln θ- ) }
              ENDIF
(9)           // ajuste dos neurônios em conflito
              FORALL l <> k, 1 <= j < = m_l DO
                    σ_j^l = min { σ_j^l, sqrt( -||x - μ_j^l||^2 / ln θ- ) }
              ENDFOR
        ENDFOR
```

Figura 2-6. Algoritmo de treinamento DDA em pseudo-código para uma época.

Ao final de algumas épocas de treinamento, normalmente quatro ou cinco, não haverá mais inclusões de neurônios, nem ajustes, indicando o fim da fase de treinamento, e haverá uma rede de neurônios interligados de acordo com a estrutura apresentada na Figura 2-4, com pesos e raios calculados de forma a atender todos os padrões apresentados durante essa fase. Os pesos normalizados das saídas podem ser calculados a partir dos pesos A_j^k, através da equação (3).

$$\forall k, 1 \le k \le c, \forall i, 1 \le j \le m_k : \pi_j^k = \frac{A_j^k}{\sum_{j=1}^{m_k} A_j^k} \qquad (3)$$

Por fim, a saída o_k da rede PNN é separada por classe, e pode ser calculada através da soma ponderada entre o retorno de cada neurônio p_j^k e os pesos normalizados π_j^k. O cálculo é feito através da equação (4).

$$o_k(x) = \sum_{j=1}^{m_k} \pi_j^k \, p_j^k(x) \qquad (4)$$

Normalização da Saída

Como a saída da rede PNN deve fornecer probabilidades, se faz necessária uma normalização dos valores obtidos dos neurônios. O procedimento comum consiste do cálculo apresentado nas equações (5) e (6), onde os valores apresentados nas saídas da rede são somados e essa soma é usada como fator para normalização de cada saída para obtenção de $p(\text{class } k|x)$. Desse modo, garantimos que a saída esteja em valores percentuais, com a soma das probabilidades obtidas igual a 1.

$$p(class\ k \mid x) = \frac{o_k(x)}{\sum_{l=1}^{c} o_l(x)} \qquad (5)$$

$$\forall x: \quad \sum_{k=1}^{c} p(class\ k \mid x) = 1 \qquad (6)$$

Contudo, tal procedimento gera alguns problemas. Agindo dessa maneira, generalizamos a todo o espaço os resultados obtidos com os padrões de treinamento. Assim, perdemos a característica da localidade, que é uma das vantagens do modelo PNN, pois será dado a todos os padrões a classificação de alguma das classes existentes, mesmo que o novo padrão seja, de fato, uma “novidade”.

Para contornar esse problema com as redes PNN, Berthold e Diamond propuseram a inclusão de uma saída extra na rede, chamada de $o_?$. Essa saída se encarrega de fornecer a informação de "novidade" na rede, e seu valor deve ser superado por alguma das saídas para que o padrão possa ser classificado em alguma das classes existentes. Para garantir essa característica, seu valor deve ser constante e igual a θ-, pois se algum padrão não for identificado como pertencente a nenhuma das classes existentes, o valor de ativação atingido por ele será menor que θ- em todos os neurônios da rede, fazendo com que todas as saídas sejam menores que θ- [2]. As equações (7), (8) e (9) com as alterações propostas estão mostradas abaixo:

$$\forall x: \quad \sum_{k=1}^{c} [p(class\ k \mid x)] + p(? \mid x) = 1 \qquad (7)$$

$$p(class\ k \mid x) = \frac{o_k(x)}{\sum_{l=1}^{c} o_l(x) + o_?} \qquad (8)$$

$$p(? \mid x) = \frac{o_?}{\sum_{l=1}^{c} o_l(x) + o_?} \qquad (9)$$

Dessa forma, fica garantido que os padrões não ajustados para nenhuma das classes sejam separados dos demais, fornecendo um tratamento mais adequado aos que se encontram na área de conflito, pois esses não podem ser simplesmente classificados nem descartados. A Figura 2-7 apresenta um gráfico com um exemplo de saída de uma rede PNN considerando duas classes A e B (a). Com a normalização, a saída se comporta como mostrado em (b), onde a linha p(?|x) identifica a área onde padrões seriam tratados como desconhecidos ou novidades.

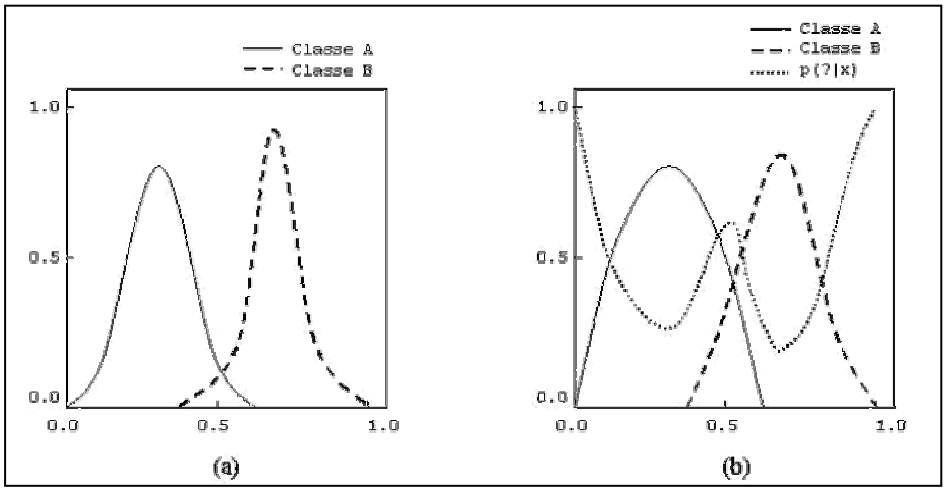

Figura 2-7. (a) Exemplo de saída de uma rede PNN com duas classes A e B. (b) Exemplo de saída normalizada com a inclusão da saída extra para padrões desconhecidos.

2.3.3 Método Proposto para Treinamento de Redes PNN para Detecção de Novidade

Apesar das propostas mostradas nas Seções anteriores, Berthold e Diamond não avaliaram o modelo PNN com a atenção voltada para a normalização apresentada. Dessa forma, não foi analisada a capacidade de trabalhar a detecção de novidade nesse tipo de rede neural. Além disso, estudos posteriores trouxeram novas técnicas e soluções que ainda não foram completamente avaliadas e integradas.

Trabalhos recentes avaliaram o impacto da variação do parâmetro θ- na resposta de classificação de redes RBF treinadas com a técnica DDA. Apesar de haver uma afirmação de Berthold e Diamond [2] de que a variação desse parâmetro não altera significativamente a capacidade de classificação da rede após o treinamento, inclusive indicando para ele os valores de

0,1 ou 0,2, os resultados atuais indicam que uma alteração da ordem de grandeza do limiar θ- é capaz de produzir variações grandes nas taxas de erro de classificação em diversas bases de dados como de reconhecimento de caracteres [20] e outras [21] [22] [23]. A Tabela 2-1 apresenta uma comparação de taxas de erro na classificação envolvendo quatro bases de dados [23].

Tabela 2-1. Comparação entre RBF-DDA e RBF-DDA com seleção de θ-. São mostrados as taxas de erro na classificação e o número de neurônios escondidos.

Método Treinamento	Optdigits	Pendigits	Letter	Satimage
RBF-DDA (Padrão)	10,18% (1953)	8,12% (1427)	15,60% (7789)	14,95% (2812)
RBF-DDA (sel. θ-)	2,78% (3812)	2,92% (5723)	5,30% (12861)	8,55% (4099)

É possível ver nos resultados da Tabela 2-1 a forte melhoria do desempenho causada pela variação desse limiar (θ-). Essa amostra dos resultados mostra a influência que esse ajuste pode apresentar durante a fase de treinamento.

Contudo, a variação do limiar θ- gera um aumento na área de conflito, fazendo com que mais padrões fiquem sem classificação, acarretando a inclusão de mais neurônios durante o treinamento DDA. É definida a complexidade de uma rede neural como a quantidade de neurônios que compõem essa rede, e um aumento nessa quantidade produz um problema chamado sobre-treinamento, quando a rede não "aprende" com os padrões, mas os "decora". Pode-se dizer também que a rede teve sua capacidade de generalização, de entendimento do problema abordado, diminuída, pois ela não consegue retirar dos dados a estrutura do problema, mas simplesmente mantém um registro das situações existentes.

Para contornar o problema acima exposto abordagens diferentes já foram propostas. Alguns trabalhos aplicados a redes RBF-DDA incluem a eliminação aleatória de um percentual dos elementos constituintes da rede ao final do treinamento [21]. Outra idéia envolve a diminuição dos dados usados durante o treinamento, com a retirada de alguns padrões que não influenciam a separação entre as diferentes classes [22]. Perfetti e Ricci [26] propuseram a retirada de neurônios após a fase de treinamento, baseando-se na avaliação da resposta de cada um deles.

No método DDA, nas primeiras épocas de treinamento, alguns neurônios são inseridos na rede para atender os padrões fornecidos, mas eles são substituídos por melhores exemplares em fases subseqüentes. Como não existe nenhum procedimento de eliminação no treinamento DDA, esses neurônios continuam na rede, apesar de suas limitações e redundâncias, e acabam não contribuindo muito para o trabalho de classificação, chegando algumas vezes a atrapalhar, podendo ser responsáveis por produzir erros.

Para avaliar o grau de redundância entre dois neurônios i e j quaisquer de uma determinada classe **k**, é associado a cada neurônio um vetor de dimensão igual ao número de

padrões (N_k) da mesma classe. Cada elemento do vetor é calculado então a partir do retorno dado pela função de ativação $\mathbf{R(x_n)}$ do neurônio, para cada padrão $\mathbf{x_n}$, com n variando entre 1 e N_k. As equações (10) e (11) mostram a criação dos chamados vetores de ativação, que possuirão a maior parte de seus elementos próximos a zero, por conta da característica de localidade dos neurônios que produzem apenas respostas significativas para os padrões próximos ao seu centro.

$$\Phi_i^k = \left[R_i^k(x_1)\ R_i^k(x_2)\ldots R_i^k(x_{N_c})\right] \qquad (10)$$

$$\Phi_j^k = \left[R_j^k(x_1)\ R_j^k(x_2)\ldots R_j^k(x_{N_c})\right] \qquad (11)$$

O grau de sobreposição ou redundância é obtido através de uma operação de produto interno entre esses dois vetores. Baseado na localidade dos neurônios, o resultado será próximo a zero para aqueles que não estejam ligados aos mesmos padrões, e aumenta de acordo com o número de padrões compartilhados. Como o resultado da operação é um valor escalar, pode-se usar esse número como um parâmetro de decisão, onde seu valor servirá de critério para a exclusão do neurônio.

A operação de produto interno deve ser realizada para cada par de vetores da rede após o fim do treinamento, e os valores calculados devem ser preservados. Assim, com a escolha de um valor η como limiar para exclusão de redundâncias, deve-se realizar uma verificação dos produtos internos calculados para avaliar quais pares de neurônios devem passar pela fase de exclusão, ou poda. Para definir qual deve ser o elemento do par a ser eliminado, leva-se em consideração os pesos de cada neurônio, ou o raio, caso haja igualdade nos pesos. A Figura 2-8 apresenta o algoritmo de avaliação para a poda, onde o neurônio com menor representatividade para a rede neural é eliminado, e o remanescente possuirá peso, ou raio, maior que o eliminado, indicando uma maior "cobertura" dos padrões de treinamento.

```
Considerando os pesos Ai e Aj:
      IF Ai > Aj(Ai < Aj)
            remover neurônio pj (pi)
      IF Ai = Aj
            Avaliar os raios ri e rj:
            IF ri > rj (ri < rj)
                  remover neurônio pj (pi)
            ENDIF
      ENDIF
      ENDIF
```

Figura 2-8. Algoritmo de poda a ser implementado. Para considerar um neurônio como menos representativo são avaliados os pesos e os raios.

Essa técnica da poda de neurônios apresentou bons resultados quanto à taxa de acerto na classificação e melhora da capacidade de generalização das redes neurais obtidas, tomando-se como critério para essa análise a complexidade final da rede [26].

Dessa forma, pretende-se, neste trabalho, implementar o treinamento DDA para o modelo PNN de rede neural, usando a normalização da saída proposta por Berthold e Diamond [2] para avaliar a capacidade de detectar novidade dessa arquitetura, com a variação do parâmetro θ- [20] [23]. É importante ressaltar que Berthold e Diamond propuseram essa técnica de normalização, porém não avaliaram a capacidade de detectar novidades de PNN-DDA com esse tipo de normalização.

Além disso, propomos neste trabalho uma nova técnica para detecção de novidades que combina redes PNN-DDA com uma técnica de poda. A idéia é adaptar a técnica de poda de neurônios proposta por Perfetti e Ricci para redes RBF-DDA [26] para o caso de redes PNN-DDA. O objetivo é diminuir a complexidade e conseqüentemente melhorar a capacidade de generalização da rede final obtida após o treinamento.

2.4 Técnica para análise de classificadores: Curvas ROC

Dentre as abordagens existentes para avaliação de classificadores, foi escolhida a análise de curvas ROC (*Receiver Operating Characteristics*) [6] [13]. Essa escolha foi motivada pelo crescente interesse despertado pela técnica na comunidade acadêmica de aprendizagem de máquina [6] [7] [10].

2.4.1 Definição

Curvas ROC são utilizadas para avaliar e quantificar o desempenho de classificadores. A idéia principal está na comparação da classificação correta dos padrões desejados com a classificação incorreta de padrões não desejados. Para fins deste trabalho são usadas duas siglas para representar esses valores, PD (*Probability of Detection*), para identificar a probabilidade de um padrão desejado ser classificado de forma correta, e PFA (*Probability of False Alarm*), para os padrões de classes diferentes das desejadas, identificados erroneamente.

Considerando padrões separados em duas classes (positivo e negativo) e um sistema de classificação, há quatro possibilidades de resultados [6]:

- Padrões da classe positiva, classificados corretamente como pertencentes à classe. Essa quantidade é chamada de TP (*true positive*);
- Padrões da classe positiva, classificados como pertencentes à outra classe. Essa quantidade é chamada de FN (*false negative*);

- Padrões da classe negativa, classificados de maneira acertada como negativos. Essa quantidade é intitulada de TN (*true negative*);
- Padrões da classe negativa, classificados erroneamente como positivos. Essa quantidade é dita FP (*false positive*).

Tomando os valores acima, podemos definir algumas métricas para análise de classificadores conforme as equações mostradas na Figura 2-9. PD e PFA foram definidos anteriormente, enquanto (utilizando a terminologia inglesa) *precision* é a probabilidade de um padrão classificado como positivo ser realmente desta classe, e *accuracy* é a probabilidade de classificar corretamente os padrões considerando as duas classes. A métrica *precision* trata apenas a capacidade de detecção dos padrões da classe positiva. *Accuracy*, por sua vez, avalia de modo geral a capacidade de detecção e exclusão de um classificador [6].

$$PD = \frac{TP}{P} = \frac{TP}{TP + FN} \quad \text{(a)}$$

$$PFA = \frac{FP}{N} = \frac{FP}{TN + FP} \quad \text{(b)}$$

$$precision = \frac{TP}{TP + FP} \quad \text{(c)}$$

$$accuracy = \frac{TP + TN}{P + N} \quad \text{(d)}$$

Figura 2-9. Equações de quatro métricas que são utilizadas para análise de desempenho: (a) PD, (b) PFA, (c) *precision* e (d) *accuracy*.

Tomando a saída de um classificador, pode-se obter um par ordenado com os valores de PFA e PD. Assim, ao se colocar os resultados obtidos por diferentes classificadores num gráfico onde o eixo das ordenadas representa PFA e o eixo das abscissas, PD, como apresentado na Figura 2-10, tem-se uma maneira de verificação visual dos resultados obtidos. Um resultado igual a (0,0; 0,0), por exemplo, representa um limite com um classificador extremamente limitado, pois não aceita nenhum padrão nem como positivo nem como negativo. O outro extremo, com resultado igual a (1,0; 1,0) apresenta um classificador onde a probabilidade de se detectar os padrões desejados é máxima, mas também o de emitir um alarme falso. Isso equivale a um classificador que identifica qualquer padrão como do tipo desejado. O resultado (0,0; 1,0) equivale ao obtido de um classificador ideal, pois a probabilidade de emissão de um alarme falso é nula, e o de detectar corretamente o padrão desejado é máxima. Como regra geral, quanto mais à noroeste no gráfico o resultado estiver, melhor será a capacidade de classificação de um sistema.

Os pontos mostrados na Figura 2-10 representam situações diferentes. O ponto D identifica um classificador ideal, como já mostrado. O ponto C mostra um resultado possível para um classificador aleatório. Ele mostra que esse classificador atribui cerca de 60% dos padrões a classe positiva, gerando, tanto para PD quanto para PFA, o valor de 60%. Qualquer classificador com essa característica aleatória, não possui informações sobre as classes, e terá um resultado que

se encontrará ao longo da diagonal mostrada. Por sua vez, o triângulo inferior representa um classificador com informações sobre as classes, mas usadas de maneira invertida, pois PFA é sempre maior PD. Nesse caso, uma inversão na saída de classificação mostrada no ponto E corrige essa falha, fazendo com que todos os classificadores se encontrem no triângulo superior esquerdo do plano, como os mostrados nos pontos A e B. Uma outra informação advinda da análise da curva está ligada à localização do resultado da classificação. Quanto mais à direita no plano, mais "liberal" pode ser considerado o classificador, pois terá valores altos de PD, mas com maiores chances de emissão de um alarme falso (alto PFA). Por conta disso, o classificador A pode ser considerado mais "conservador" que o B, pois um valor pequeno do PFA impede a classificação equivocada de padrão negativos.

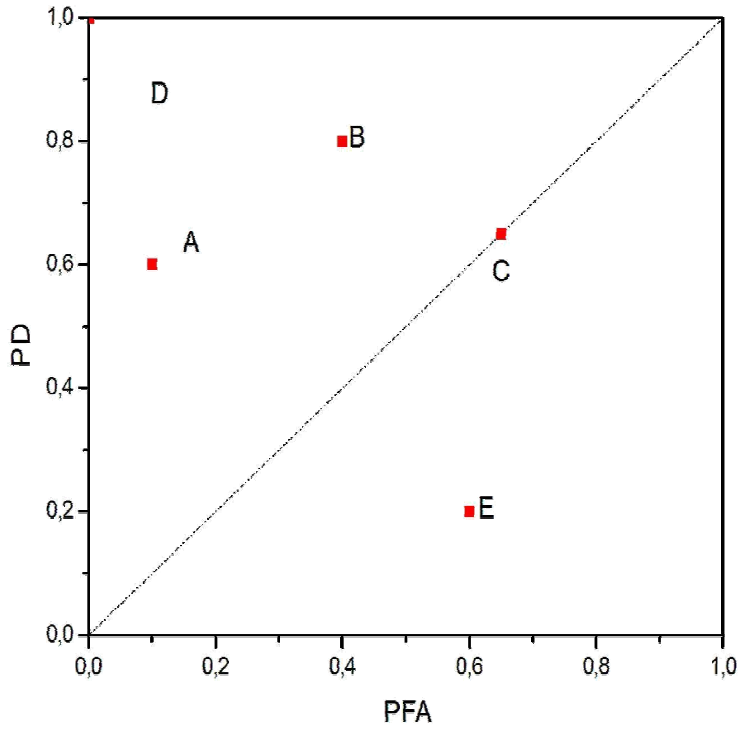

Figura 2-10. Gráfico representando o plano ROC, com cinco exemplos de resultados de classificadores

A curva ROC é montada a partir da obtenção de vários pares ordenados (PFA, PD). Esses valores são obtidos de forma dinâmica, através da variação de algum parâmetro que controla os resultados apresentados pelo sistema de classificação. Tomando como exemplo redes neurais MLP (*Multi-layer Perceptron*) [4], onde normalmente se usa o princípio do *winner-takes-all* [4] para definir qual classe será a resposta dada pela rede, pode-se adotar uma outra técnica onde a resposta da rede é definida a partir da avaliação do resultado obtido em uma das saídas. A comparação desse resultado com um limiar pré-definido λ qualquer, definirá a saída fornecida pela rede neural. Dessa maneira, se for realizada uma variação no limiar λ, entre 0,0 e 1,0, no caso da saída normalizada, é possível ter diferentes classificações para os mesmos padrões, tendo

pares distintos (PFA, PD) para cada λ utilizado na análise. O limiar λ do exemplo acima está ligado à arquitetura MLP, mas cada modelo de simulação permite esse tipo de avaliação, através da variação de algum parâmetro específico do modelo. Essa idéia de avaliação dinâmica pode ser utilizada em qualquer classificador que produza saídas probabilísticas ou numéricas não normalizadas, como métodos estatísticos de classificação e redes neurais.

Um conceito importante para comparação de curvas ROC é o de dominação [10]. Uma curva X domina uma curva Y caso todos os pontos de X estejam acima e à esquerda de todos os pontos de Y. Nesses caso, podemos dizer que o classificador representado pela curva X produz sempre melhores resultados que o classificador Y. Mas nem sempre essa dominância acontece, por isso, como a curva ROC é uma representação bi-dimensional da capacidade de classificação, é interessante haver uma quantidade escalar para auxiliar a comparação de classificadores.

Para a limitação apresentada acima é possível calcular a área sob a curva ROC, denominada AUC, do termo em inglês *Area Under Curve*. Como a curva está limitada a um quadrado de lado igual a 1,0, o valor de AUC sempre é menor que 1,0. Como nenhum classificador real fica abaixo da linha diagonal, a curva ROC se localiza acima dessa linha, fazendo com que AUC fique com seu valor restrito aos limites 0,5 e 1,0. Com a comparação dos valores de AUC conseguimos comparar curvas como as mostradas na Figura 2-11, onde ocorre variação do comportamento dos classificadores para situação específicas. Em algumas oportunidades, o classificador identificado pela letra **B** consegue atingir melhores resultados de classificação que o classificador **A**. Contudo, ao compararmos seus valores para o AUC, temos que a capacidade de classificação de **A** apresenta na média melhores resultados.

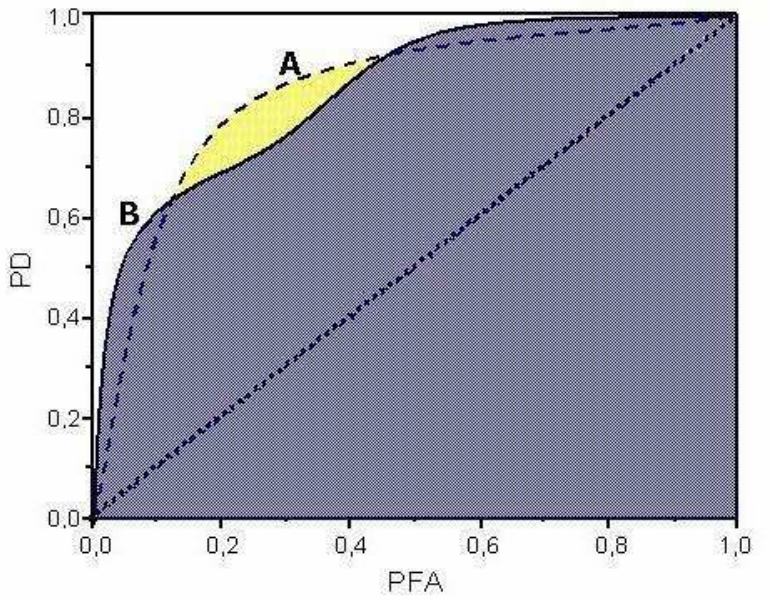

Figura 2-11. Exemplo de curvas ROC. As curvas mostram que o classificador **B** apresenta melhores resultados que **A** em algumas condições. Contudo, com o valor AUC de **A** maior que o de **B**, o primeiro classificador é o que apresenta melhores resultados de maneira geral.

Por ser uma técnica simples e poderosa, a análise de curvas ROC é bastante utilizada, em especial, em áreas onde os classificadores binários, ou discretos, são mais comuns como a área médica ou de transmissão de sinais.

Trabalhos recentes demonstraram que o uso de curvas ROC, e seu respectivo valor da AUC, é uma boa métrica para avaliação de algoritmos de aprendizagem, no lugar da comparação do percentual de acerto de classificação [6] [10]. Uma das vantagens existentes no seu uso é a análise da capacidade de classificação sem a necessidade do estabelecimento de limiares, uma vez que a curva apresenta o comportamento do sistema como um todo, pois a obtenção dos pontos da curva é feita através da variação de algum parâmetro de forma dinâmica, ao longo de toda a faixa de possibilidades.

Devido a essa melhor capacidade de análise, o valor da AUC pode ser utilizado nos trabalhos de avaliação, comparação e projetos de algoritmos de aprendizado. Um exemplo está em uma técnica chamada ROCCER [7], um algoritmo proposto para seleção de regras baseado na análise da curva ROC. Os resultados alcançados pelos proponentes mostraram que o conjunto de regras é bem menor que os obtidos de outros métodos, com valores de AUC compatíveis entre si.

Capítulo 3

Simulador Desenvolvido

Para os fins propostos neste trabalho, foi tomada a decisão de se desenvolver uma ferramenta para simulação dos modelos de dados escolhidos, no problema da detecção de novidades. Além da implementação dos métodos que envolvem redes neurais probabilísticas, foi tomada a decisão de implementar no simulador o modelo estatístico NNDD. Antes de tal decisão, foi avaliada uma biblioteca de funções chamada *Data Description Toolbox* (DDTOOLS) [3], desenvolvida para uso com o programa MATLAB [19]. Essa ferramenta provê uma série de modelos prontos para uso, bem como fornece algumas métricas de avaliação como a curva ROC e a AUC.

Contudo, como o principal objetivo da DDTOOLS é o de fornecer funcionalidades úteis a pesquisadores, o foco de desenvolvimento não ficou ligado a aspectos de desempenho e uso de recursos da memória, pelo menos na versão 1.11, usada nos testes. Desse modo, quando foram usadas bases de dados com informações do mundo real, por sua alta complexidade em relação ao número de padrões e devido ao grande número de dimensões de cada padrão, facilmente se alcançou o limite físico de endereçamento de memória, causando falta de recursos e falha na execução das simulações. Esse problema foi sentido em dois dos modelos mais simples, o NNDD e a mistura de Gaussianas, não sendo facilmente contornável, pois envolveria alguma técnica para agrupamento prévio dos dados em *clusters*, ou uma alteração na implementação da DDTOOLS, para execução das rotinas internas em etapas separadas.

Assim, a ferramenta *Novelty Detector*, desenvolvida neste trabalho, fornece a capacidade de treinamento e teste com os modelos NNDD, PNN com treinamento DDA e implementa a nova abordagem do treinamento DDA do modelo PNN, aplicando a técnica proposta de poda para diminuição do número de neurônios da rede treinada (complexidade). Além disso, os resultados obtidos são fornecidos através de curvas ROC e seus respectivos valores da AUC.

Foi escolhida a linguagem Java [11] para desenvolvimento da aplicação. Foi também escolhida a plataforma Eclipse [5] para o trabalho, juntamente com o *plug-in Visual Editor* [31], para criação da interface gráfica. A versão do *Java Development Kit* (JDK) usada foi a 5.0, porém foi mantida a compatibilidade do projeto com a versão 1.42, uma vez que essa é uma versão

estável e conhecida, e foi usada no desenvolvimento de alguns protótipos anteriores ao presente trabalho.

O fluxo de atividades necessários para a execução das simulações está apresentado na Figura 3-1. O diagrama mostra a seqüência de passos necessárias para a obtenção da curva ROC e seu respectivo valor de AUC. A idéia foi desenvolver o programa pensando numa execução monotônica, onde as etapas são executadas numa ordem definida. A principal desvantagem dessa abordagem ficou sendo a obtenção e comparação de resultados, uma vez que o foco está na implementação dos algoritmos de treinamento. Caso haja a necessidade de execução de uma série de simulações para comparação de resultados, os dados devem ser obtidos e guardados para análise em outra ferramenta. Nessa etapa do projeto, não é possível, por exemplo, colocar duas curvas ROC distintas num mesmo gráfico para uma comparação visual. Por ser um problema contornável, sua solução será deixada para uma etapa futura de desenvolvimento.

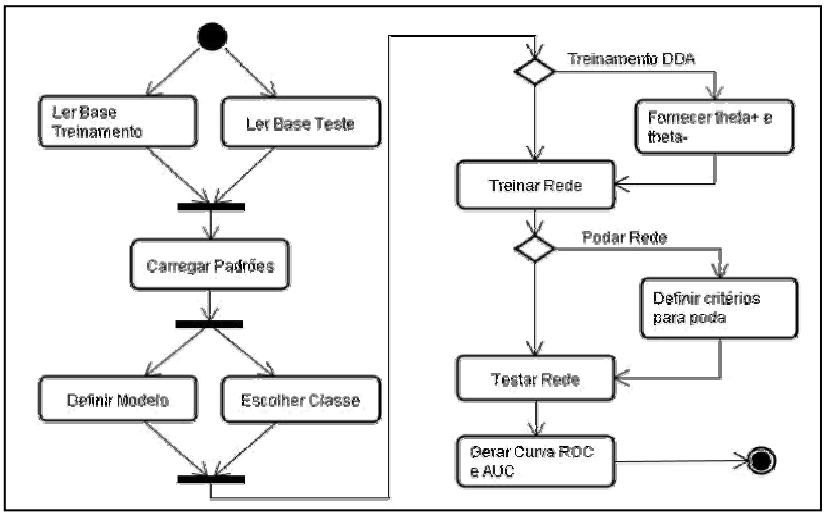

Figura 3-1. Diagrama da atividade do simulador *Novelty Detector*.

Apesar do fluxo único, a idéia básica no desenvolvimento do simulador, é que haja diferentes implementações de modelos para treinamento e teste. Ficou definido que seriam incluídos os modelos de dados NNDD, PNN-DDA e PNN-DDA com poda, sendo este último a proposta apresentada neste trabalho. Os três modelos seguirão um padrão definido para as classes, com a estrutura similar à apresentada na Figura 3-2, onde são mostradas algumas classes do projeto e relacionamentos de herança existentes. Um pacote chamado `common` será utilizado para agrupar classes com funcionalidades acessadas por outras classes, além daquelas responsáveis por indicar a estrutura que deve ser seguida para a implementação de um modelo de representação de dados. Essas classes são abstratas e chamadas `NeuralNetwork`, `NeuralElement` e `Roc`.

`NeuralNetwork` deve ser usada como superclasse da classe que representará a estrutura necessária para armazenamento do modelo de dados a ser implementado. Apesar do nome, essa

classe no presente trabalho é usada tanto para redes neurais, quanto para modelos estatísticos, uma vez que a estrutura criada atende a ambos. Essa classe possui um atributo privado chamado `net`, implementado através do tipo Java *Vector*, responsável por armazenar os elementos constituintes do modelo. Um método existente nessa superclasse é o `getComplexity()`, responsável por informar a complexidade da rede. Normalmente, esse valor é inteiro e igual ao número de elementos presentes na rede.

`NeuralElement` é uma classe sem atributos e métodos, apenas encarregada de garantir a definição de elementos para povoar a rede de dados. `Roc`, por sua vez, é responsável por identificar dois métodos importantes para a obtenção dos resultados esperados. O método `getPontosCurva()` retorna uma matriz de pontos (PFA, PD) para criação da curva e o método `getAUC()` informa o valor da AUC da curva obtida. Esses métodos são abstratos e sua implementação deve ser dada por qualquer classe filha, dependendo da estrutura de dados definida.

Como exemplo, a Figura 3-2 apresenta algumas características da implementação do modelo PNN.

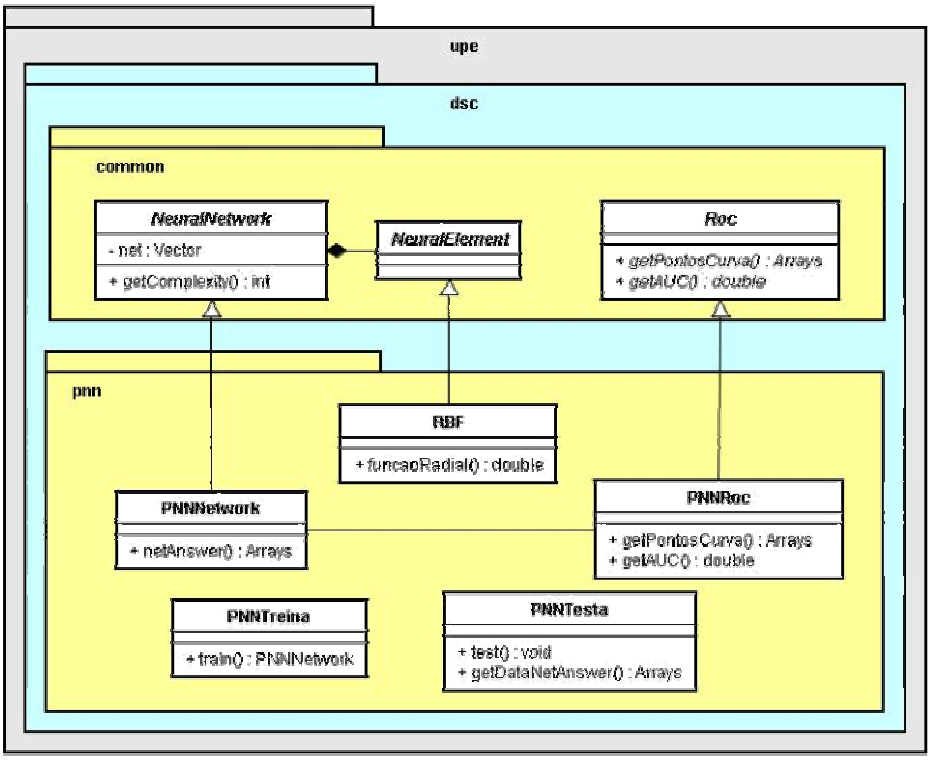

Figura 3-2. Diagrama reduzido de algumas classes do simulador *Novelty Detector*, apresentando a estrutura básica a ser seguida na implementação do projeto.

A classe RBF herda da classe `NeuralElement` os relacionamentos e representa a unidade básica da rede PNN que é um elemento criado com base numa função radial (*radial basis*

function - RBF). Essa classe possui um método chamado `funcaoRadial()`, dentre outros, que fornece a resposta dada pelo elemento a um padrão fornecido. A rede PNN é obtida através da classe `PNNNetwork`, filha da superclasse `NeuralNetwork`, possuindo uma nova função `netAnswer()`, responsável por informar o retorno obtido a partir da rede a um padrão. Essa função é obtida através da análise dos valores obtidos a partir de cada RBF isoladamente, sendo fornecido um valor para cada classe representada. A classe `PNNRoc` se encarrega de implementar as funções abstratas herdadas, usando os valores de retorno da rede PNN para calcular as probabilidades PFA e PD, bem como o valor de AUC para a curva final gerada.

Além dos pacotes mostrados na Figura 3-2, existem outros com estrutura similar para os modelos NNDD e PNN-DDA com poda. Como cada um apresenta peculiaridades específicas, as funções internas variam de acordo com a necessidade, mas os relacionamentos apresentados se mantêm. Além de permitir uma organização das classes obtidas, a estrutura criada permite que se possa expandir o trabalho atual, agregando novos modelos e funcionalidades no futuro.

A interface desenvolvida para o projeto é apresentada na Figura 3-3, onde vários campos são mostrados desabilitados.

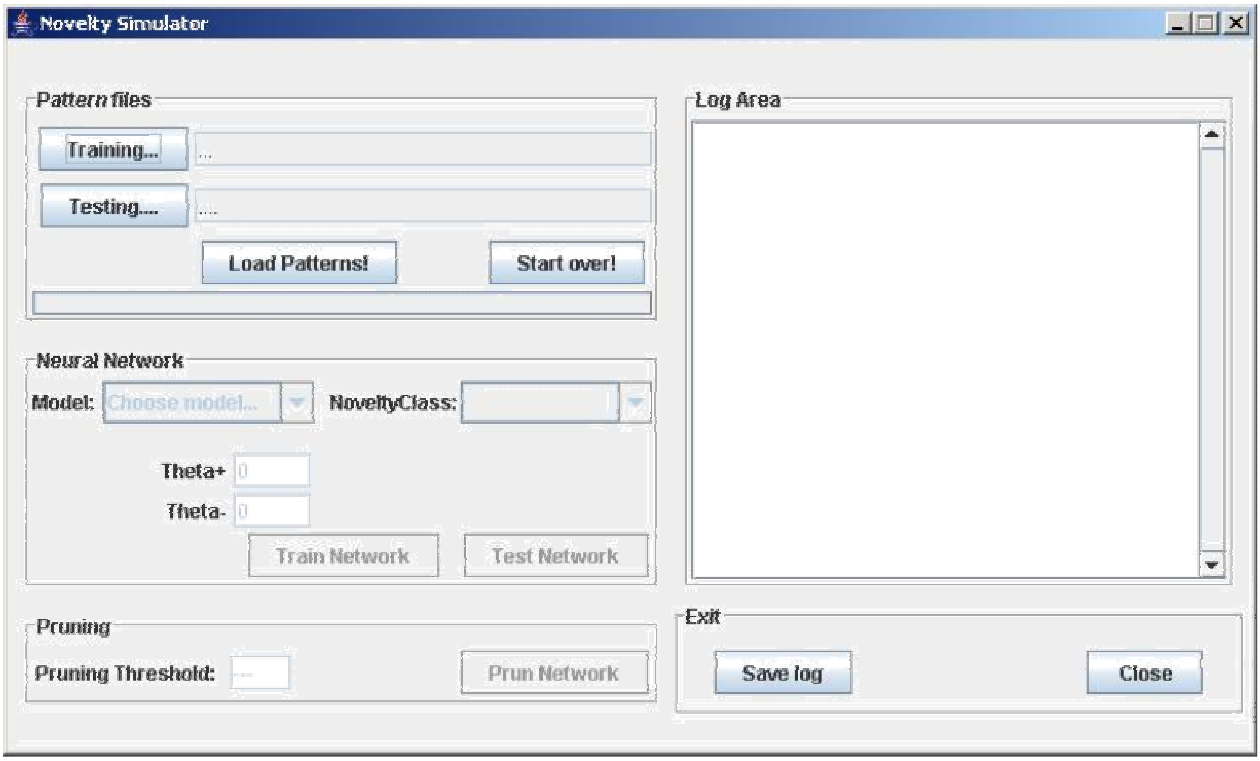

Figura 3-3. Interface do simulador *Novelty Detector* desenvolvido.

Para garantir a correta execução dos passos do programa, a habilitação dos botões e campos só é feita com a conclusão efetiva da etapa anterior. Assim, na primeira execução, apenas os botões "*Training...*" e "*Testing...*" estão funcionais, pois são neles que se definem quais são os

arquivos com as bases de dados para treinamento e teste, respectivamente. Após a leitura dos arquivos através do botão "*Load Patterns!*" as listas de seleção "*Model:*" e "*Novelty Class*" ficam disponíveis para uso, e assim segue a execução do programa. Apenas três botões estão sempre disponíveis para uso a qualquer momento: "*Start over!*", responsável pela reinicialização do simulador; "*Save log*", que se encarrega do salvamento das informações contidas na área de texto do programa; e "*Close*", que encerra a aplicação.

3.1 Estrutura das Bases de Dados

Para estruturar as bases de dados em arquivos e permitir sua leitura pela aplicação, foi escolhido o formato usado pela ferramenta SNNS [27]. Esse simulador usa arquivos com extensão *pat*, para representar os dados a serem usados nas simulações, tanto nos treinamentos quanto nos testes. Essa escolha se deu pelo uso já difundido da ferramenta no meio acadêmico, o que faz com que várias bases já estejam disponíveis nesse formato.

Os arquivos *pat* são constituídos de texto simples no formato ASCII, e sempre iniciam com o cabeçalho mostrado na Figura 3-4.

```
SNNS pattern definition file V1.4
generated at Mon Oct 27 18:31:49 2003


No. of patterns     : 7200
No. of input units  : 21
No. of output units : 3

0.73 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0.0006 0.015 0.12 0.082 0.146 0 0 1
0.62 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0.011 0.008 0.073 0.074 0.098 0 1 0
....
```

Figura 3-4. Exemplo de arquivo de padrões do simulador SNNS.

A primeira linha do cabeçalho do arquivo indica a versão do formato representado. A segunda linha indica o momento de geração do arquivo com os padrões. As três últimas mostram valores importantes para o treinamento ou teste da rede, que são: (1) a quantidade de padrões constante do arquivo; (2) o número de entradas de cada padrão; e (3) o número de classes ou saídas.

Quando é informado qual arquivo deve ser lido no *Novelty*, as primeiras informações obtidas são as constantes nas três linhas, pois são essenciais a todas as operações.

Após o cabeçalho do arquivo, os padrões são fornecidos na área de dados. A Figura 3-4 mostra um exemplo obtido da base Thyroid, composta de 7200 padrões, divididos entre treinamento, validação e teste, montada com 21 entradas e classificada entre 3 classes (saídas). Nele, cada linha representa um padrão, e cada elemento da linha representa o valor em cada

entrada e a saída obtida na classificação. Assim, cada padrão deve possuir uma quantidade de valores igual à soma dos números de entradas e classes, informados no cabeçalho do arquivo. Comentários são iniciados através do caractere '#' e não devem ser processados. Outra forma válida de apresentação dos dados pode ser vista na Figura 3-5.

```
# input
0.73 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0.0006 0.015 0.12 0.082 0.146
#target
0 0 1
# input
0.62 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0.011 0.008 0.073 0.074 0.098
#target
0 1 0
.....
```

Figura 3-5. Outra forma de representação dos dados nos arquivos de padrões do SNNS.

O simulador *Novelty Detector* foi desenvolvido pensando em bases com entradas numéricas. Assim, caso a base selecionada possua atributos booleanos (*true/false*), ou entradas com referências a características textuais, se faz necessário um pré-processamento dos dados para fornecer os dados na forma esperada pelo simulador. Para tanto, valores textuais podem ser associados a quantidades numéricas, bem como podemos atrelar os valores booleanos aos números 1 (true) e 0 (false). Ao final, uma normalização em cada entrada pode ser realizada.

3.2 JFreeChart

A fim de facilitar a obtenção de curvas ROC ao final de cada rodada de simulação (treinamento e teste) iremos utilizar uma biblioteca desenvolvida para a linguagem Java chamada JFreeChart [12]. Seu objetivo é facilitar para desenvolvedores a obtenção de gráficos e curvas, através de uma API bem desenvolvida, documentada e facilmente expansível. Além disso, sua característica *free software*, fornecida através da licença *LGPL - GNU Lesser General Public License* [14], torna fácil seu uso, permitindo a ampla utilização em aplicação acadêmicas.

JFreeChart permite a obtenção de gráficos do tipo pizza, curvas, barras, dispersão, dando também a opção de obtê-las em 3D. A biblioteca também possui recursos específicos para o desenho de séries temporais, bem como curvas da área financeira. A Figura 3-6 mostra alguns exemplos de gráficos que podem ser feitos com a biblioteca.

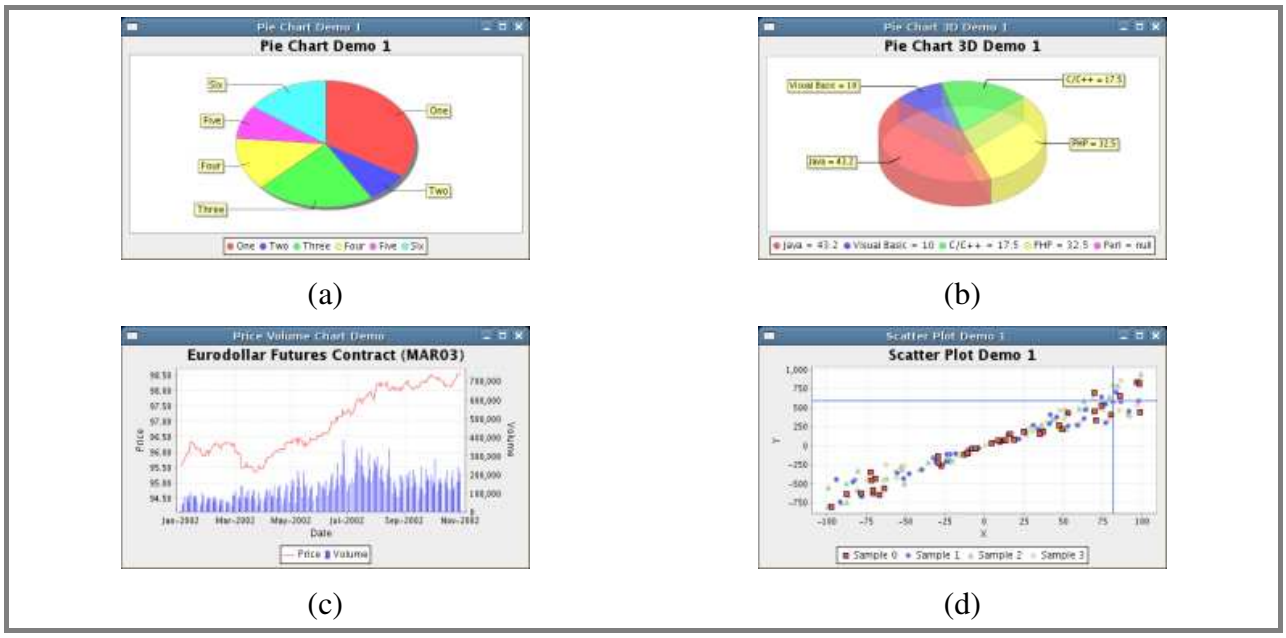

Figura 3-6. Exemplos de gráficos obtidos através da biblioteca JFreeChart. Gráfico em formato pizza (a), Gráfico pizza 3D (b), série temporal (c) e gráfico XY (d).

Como a curva ROC é obtida a partir dos pares (PFA, PD), foi utilizada na implementação a classe `XYDataset`, para adequação dos valores obtidos após os testes para o formato utilizado pela biblioteca. Além disso, foi usado o método `createXYLineChart()` da classe `ChartFactory` para definir o tipo de gráfico que se pretende obter. Por fim, foi definido que a janela com os resultados deve possuir dois botões: um para salvar a imagem obtida com a curva e outro para salvamento dos pontos da curva. Essas opções são importantes para permitir a análise dos dados em ferramentas específicas. A Figura 3-7 apresenta um exemplo de curva ROC obtida através da ferramenta *Novelty* Detector, usando a biblioteca JFreeChart.

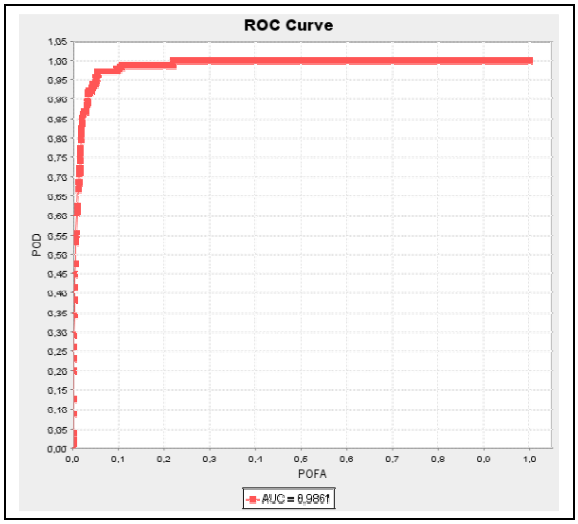

Figura 3-7. Exemplo de curva ROC obtida após simulação com a ferramenta *Novelty Detector*. Além da curva, é informado o valor do AUC.

Capítulo 4

Experimentos e Resultados

Os experimentos foram realizados através da ferramenta *Novelty Detector*, desenvolvida neste trabalho. As bases de dados listadas na Tabela 4-1 foram escolhidas a partir do repositório UCI [30] e são normalmente usadas para *benchmarking*. Em termos de tamanho, a escolha recaiu em uma base pequena (soybean), três médias e uma grande (letter), conforme listado na Tabela 4-1.

Tabela 4-1. Bases selecionadas para o trabalho com as quantidade de atributos, classes e padrões.

Base	# Atributos	# Classes	# Padrões Treinamento	# Padrões Teste
Soybean	82	19	513	170
Optdigits	64	10	3823	1797
Pendigits	16	10	7494	3498
Letter	16	26	15000	5000
Satimage	36	6	4827	1608

As bases selecionadas são usadas normalmente em problemas de classificação com múltiplas classes. Mas como a intenção é utilizá-las para avaliar detecção de novidade, foi utilizado um artifício de considerar uma das classes existentes como "novidade", sendo as demais consideradas normais, a fim de simular a estrutura normal/novidade presente no problema. Dessa maneira, a fase de treinamento é feita sem considerar dados da classe escolhida, que somente é apresentada durante a fase de testes, para a análise da capacidade de detectar corretamente padrões considerados novos. Esse artifício já foi utilizado em outros trabalhos que envolvem classificadores de uma única classe [29] e detecção de protótipos [9].

O simulador desenvolvido realiza essa exclusão de forma dinâmica, a partir da definição pelo operador do programa, da classe a ser considerada novidade. Essa forma de operação permite que várias execuções sejam realizadas facilmente na mesma base, sem a necessidade de tratamento prévio dos dados. A Tabela 4-2 apresenta as bases selecionadas com as classes escolhidas como novidades, além da divisão dos padrões entre normais e novidades. Vale

ressaltar o fato de que somente na base de teste é que se encontram os padrões novos, pois somente nessa fase, o detector deve tomar conhecimento dos dados, para fins de teste.

Tabela 4-2. Características das bases de dados usadas nos experimentos de detecção de novidade

Base	Classe Novidade	Treinamento	Teste	
		# Normal	# Normal	# Novidade
Soybean	17	445	150	20
Optdigits	6	3436	1616	181
Pendigits	3	6715	3134	364
Letter	3	14407	4827	173
Satimage	1	3755	1147	461

Para cada simulação foi anotado o valor da AUC da curva ROC obtida, bem como a do número final de neurônios na rede treinada (complexidade) do detector de novidades. Esses valores foram usados para avaliação e comparação dos métodos.

Apesar da ferramenta desenvolvida possuir a capacidade de fornecer curvas ROC, como apenas uma curva é obtida por vez, foi utilizado o programa Origin [25], em sua versão 6.0, para obtenção de gráficos com múltiplas curvas. Tal necessidade é importante para avaliar o impacto das propostas a serem analisadas nas curvas ROC.

4.1 Simulações com Modelo NNDD

O modelo NNDD foi avaliado através da ferramenta desenvolvida, *Novelty Detector*. A biblioteca de ferramentas DDTOOLS não foi usada por não conseguir executar as simulações para bases médias/grandes, mas serviu como base na validação dos resultados obtidos para a base Soybean.

4.1.1 Resultados

Foram executadas simulações com quatro das bases presentes na Tabela 4-1 (Soybean, Optdigit, Pendigit e Letter). Os resultados obtidos estão listados na Tabela 4-3, onde apenas os valores de AUC obtidos são apresentados. As curvas ROC serão mostradas mais adiante quando da comparação dos resultados com outros modelos.

Tabela 4-3. Resultados obtidos com o modelo NNDD.

Base	Complexidade	AUC
Soybean	448	0,5706
Optdigits	3447	0,9233
Pendigits	6714	0,8707
Letter	14435	0,9162

4.2 Simulações com Modelo PNN-DDA

Para o modelo PNN-DDA, foi escolhido o valor de θ+ igual a 0,4 para todas as simulações, e a variação do parâmetro θ- foi feita de forma sucessiva a partir do valor 10^{-1} até 10^{-6}, conforme procedimento adotado em outros trabalhos [20] [21] [22] [23]. O valor inicial igual a 10^{-1} foi usado por Berthold e Diamond no trabalho com redes RBF [1] e a variação do limiar é feita na sua ordem de grandeza, porque pequenas variações do valor não influenciam os resultados obtidos [20].

4.2.1 Resultados

Após as execuções das simulações com as bases testadas, com a escolha das classes listadas na Tabela 4-2, foram obtidos os resultados elencados na Tabela 4-4. Os valores da AUC estão listados, juntamente com o número de neurônios da rede treinada, ou complexidade, entre parênteses. Cada coluna apresenta os resultados de acordo com o parâmetro θ -.

Tabela 4-4. Resultados obtidos com o modelo PNN-DDA.

Base	θ- = 10^{-1}	θ- = 10^{-2}	θ- = 10^{-3}	θ- = 10^{-4}	θ- = 10^{-5}	θ- = 10^{-6}
Soybean	0,6575 (170)	0,9018 (240)	0,9173 (282)	0,9133 (293)	0,9054 (301)	0,9015 (315)
Optdigits	0,9143 (1680)	0,9778 (2952)	0,9884 (3323)	0,9910 (3410)	0,9910 (3435)	0,9910 (3435)
Pendigits	0,8393 (1076)	0,8924 (2455)	0,8970 (3590)	0,9066 (4360)	0,9020 (4898)	0,9021 (5276)
Letter	0,7318 (7460)	0,8113 (10522)	0,8510 (11700)	0,8857 (12301)	0,8959 (12634)	0,9010 (12879)

As curvas ROC obtidas são apresentadas na Figura 4-1, Figura 4-2, Figura 4-3, bem como na Figura 4-4. O que pode ser prontamente percebido é que o parâmetro θ- é capaz de influenciar o desempenho na detecção de novidades para as quatro bases de dados analisadas, variando apenas o grau de influência. Para todas elas, com θ- = 10^{-1} o detector se comporta da pior maneira, se aproximando mais da linha diagonal que identifica um classificador aleatório. Através da Tabela 4-4 também é possível constatar esse fato, através da comparação do valor de AUC, que é sempre menor na primeira coluna. O aumento do parâmetro θ- faz com as curvas ROC se aproximem cada vez mais do ponto considerado ideal no gráfico (0,0, 1,0)

Nas bases de dados Soybean e Pendigits é possível identificar também uma queda no valor de AUC a partir de determinados valores de θ-. Esse detalhe pode ser associado ao treinamento excessivo, gerado pela inclusão de cada vez mais neurônios, o que faz com que a rede comece a "decorar" os padrões de dados usados no treinamento, perdendo parte de sua capacidade de generalização, em outras palavras, sua capacidade de "entender" o problema.

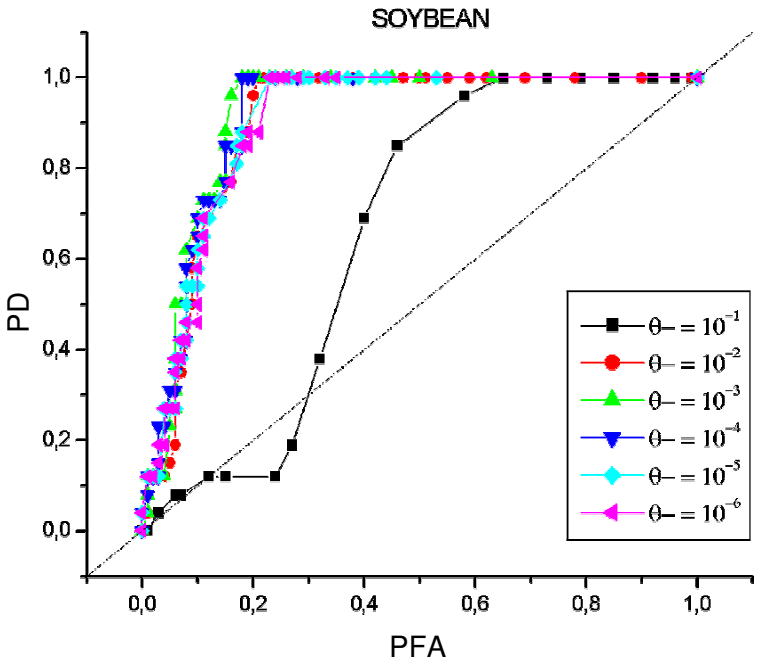

Figura 4-1. Curvas ROC para a base Soybean

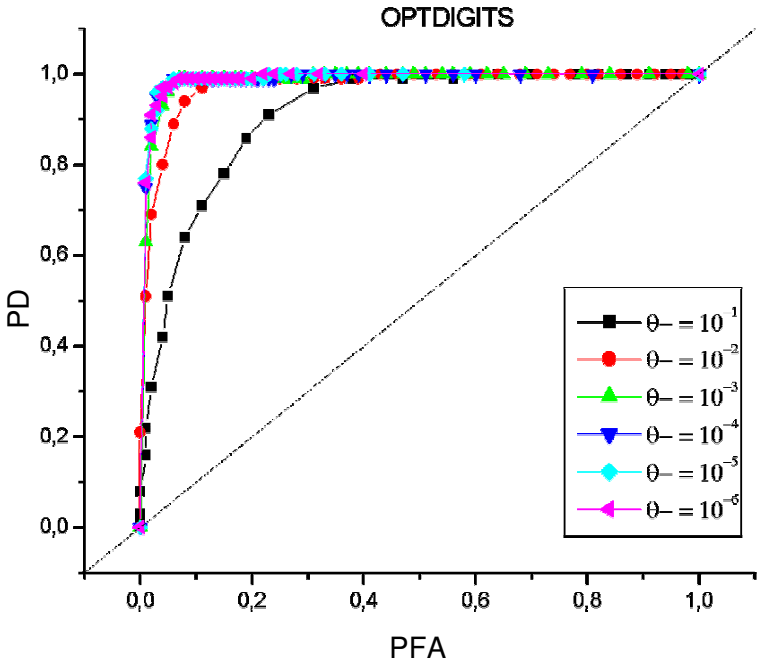

Figura 4-2. Curvas ROC para a base Optdigits

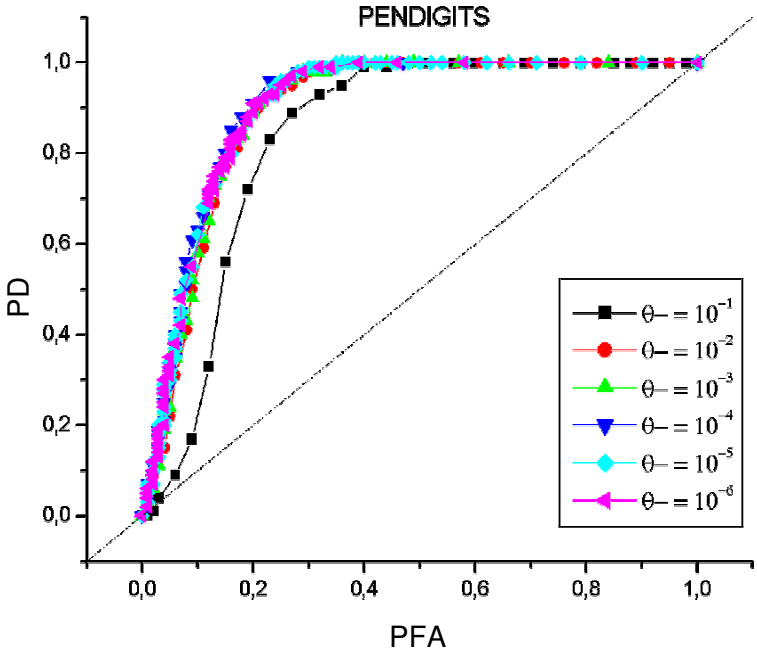

Figura 4-3. Curvas ROC para a base Pendigits

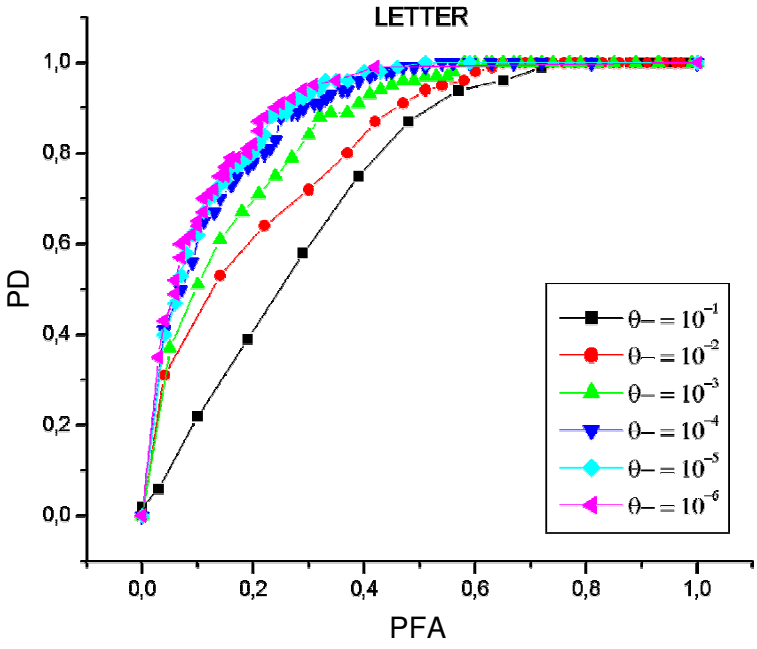

Figura 4-4. Curvas ROC para a base Letter

4.3 Simulações com Modelo PNN-DDA com Poda

Para inclusão da poda após o treinamento DDA, um novo limiar η é introduzido, para definir o nível de poda a ser realizado na rede. Como esse parâmetro estabelece um balanceamento entre o grau de especialização e a capacidade de generalização, seu valor será obtido a partir de uma faixa que parte de 0,01 e vai até 10,0, passando pelos valores mostrados na Tabela 4-5.

Essa faixa foi produzida baseado nos resultados experimentais preliminares obtidos, que mostraram que no modelo PNN-DDA implementado, o valor de η não possui um valor máximo geral para qualquer base de dados. Perfetti e Ricci propuseram para as redes RBF o uso do valor de θ+ como limite máximo para o parâmetro η [26]. A partir deste limite, deve haver uma diminuição em seu valor até se atingir os patamares de poda desejados pelo pesquisador. Nos experimentos realizados, ao se tomar $\eta = 0{,}4$, que foi o valor adotado para θ+, foi obtida uma diminuição da complexidade da base Soybean de cerca de 35%, o que contrasta com a informação fornecida pelos pesquisadores. Assim, através de teste nos valores, foram escolhidos um de uma ordem de grandeza superior ($\eta = 10$), onde praticamente não há poda, e outro inferior ($\eta = 0{,}01$), quando a diminuição da complexidade atingiu valores superiores a 90%. Entre essas duas quantidades, foram escolhidos aleatoriamente alguns valores para avaliar o comportamento da poda, com a variação do limiar.

Tal diferença provavelmente está ligada à estrutura da rede PNN, que inclui uma normalização nos pesos entre a camada escondida e a camada de saída, bem como a normalização proposta para as saídas, com a inclusão de uma específica para "novidades".

Tabela 4-5. Valores usados para o limiar de poda η.

Limiar η													
0,01	0,05	0,1	0,2	0,3	0,4	0,5	0,8	1,0	2,0	3,0	4,0	5,0	10,0

Para a análise do algoritmo de poda, além das bases já usadas nas simulações anteriores, foi incluída a base Satimage, também do repositório UCI [30]. Foram feitas as simulações necessárias para obtenção do valor de θ- que produz o melhor valor para AUC, chegando-se aos seguintes valores para esta base:

- Classe Novidade = 1
- $\theta- = 10^{-6}$ (melhor valor para AUC)
- AUC = 0,8161
- Complexidade da rede = 3458

A partir dai, foram feitas rodadas consecutivas de simulação com variação do fator de poda η, nas cinco bases selecionadas, com os parâmetros de acordo com a Tabela 4-6.

Tabela 4-6. Bases selecionadas para análise do algoritmo de poda, com os parâmetros de treinamento obtidos com o treinamento PNN-DDA sem poda.

Bases	θ+	θ-	Classe "Novidade"	Complexidade	AUC
Soybean	0,4	10^{-3}	17	282	0,9173
Optdigits	0,4	10^{-4}	6	3410	0,9910
Pendigits	0,4	10^{-4}	3	4360	0,9066
Letter	0,4	10^{-6}	3	12879	0,9010
Satimage	0,4	10^{-6}	1	3458	0,8161

4.3.1 Resultados

Os resultados obtidos após as simulações se encontram na Tabela 4-7, Tabela 4-8, bem como na Tabela 4-9.

Tabela 4-7. Resultados da poda para as bases Soybean e Optdigits.

	Soybean		Optdigits	
η	Complexidade	AUC	Complexidade	AUC
0,01	70	0,7889	505	0,9869
0,05	107	0,8515	1220	0,9879
0,1	137	0,8924	1630	0,9895
0,2	158	0,8898	2167	0,9891
0,3	170	0,8908	2478	0,9894
0,4	184	0,8906	2741	0,9900
0,5	197	0,8914	2926	0,9898
0,8	225	0,9112	3228	0,9904
1,0	237	0,9099	3308	0,9907
2,0	268	0,9145	3408	0,9909
3,0	275	0,9183	3410	0,9910
4,0	279	0,9173	3410	0,9910
5,0	280	0,9173	3410	0,9910
10,0	281	0,9173	3410	0,9910
Sem poda	282	0,9173	3410	0,9910

Tabela 4-8. Resultados da poda para as bases Pendigits e Letter.

	Pendigits		Letter	
η	**Complexidade**	**AUC**	**Complexidade**	**AUC**
0,01	154	0,5000	6334	0,9302
0,05	590	0,6721	8411	0,9337
0,1	404	0,7293	9399	0,9324
0,2	973	0,8023	10370	0,9277
0,3	1305	0,8302	10976	0,9285
0,4	1594	0,8561	11369	0,9281
0,5	1844	0,8766	11662	0,9278
0,8	2545	0,8960	12251	0,9226
1,0	2846	0,8923	12472	0,9163
2,0	3697	0,9001	12764	0,9094
3,0	3994	0,9044	12836	0,9047
4,0	4133	0,9057	12854	0,9042
5,0	4211	0,9071	12862	0,9041
10,0	4345	0,9068	12879	0,9039
Sem poda	4360	0,9066	12879	0,9010

Tabela 4-9. Resultados da poda para a base Satimage

η	**Complexidade**	**AUC**
0,01	2470	0,8051
0,05	2895	0,8166
0,1	3052	0,8174
0,2	3185	0,8162
0,3	3253	0,8168
0,4	3300	0,8178
0,5	3321	0,8172
0,8	3360	0,8167
1,0	3378	0,8166
2,0	3402	0,8163
3,0	3413	0,8163
4,0	3420	0,8163
5,0	3425	0,8163
10,0	3439	0,8163
Sem poda	3458	0,8161

4.4 Análise dos Resultados

Com os dados obtidos a partir das simulações, algumas conclusões importantes puderam ser obtidas.

4.4.1 NNDD x PNN-DDA

Comparando os resultados obtidos com o modelo NNDD e as redes PNN-DDA, pode-se afirmar que melhores resultados foram obtidos no geral com as redes PNN-DDA. A Figura 4-5 apresenta as curvas ROC das sete simulações realizadas (1 NNDD + 6 PNN-DDA) com a base Soybean. Através do gráfico, é possível ver uma significativa melhoria no desempenho do detector de novidades através das redes PNN-DDA. Com o valor $\theta- > 10^{-1}$ o valor de AUC já aumenta bastante, em relação ao modelo NNDD. Além disso, também é possível constatar que a capacidade de detectar novidade aumenta com a diminuição de θ-, sendo essa melhora mais sentida quando o valor é diminuído de 10^{-1} a 10^{-2}.

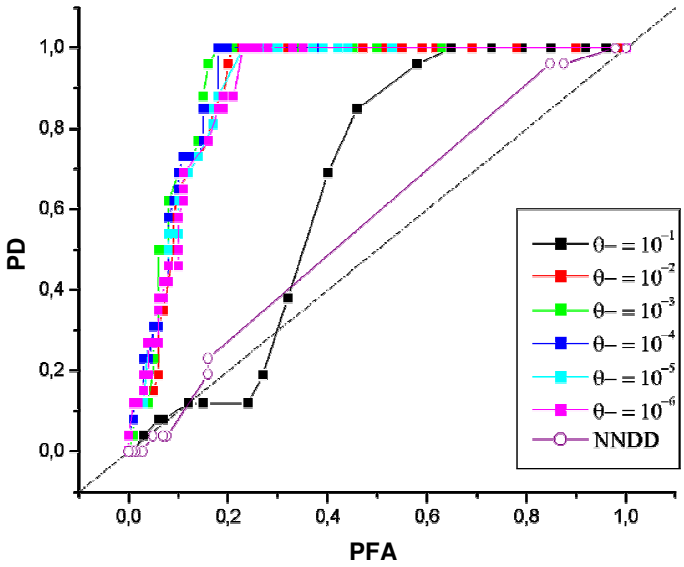

Figura 4-5. Curvas ROC para a base Soybean, construídas usando NNDD e PNN- DDA com variação de θ-.

A mesma melhoria dos resultados por conta da variação de θ- pode ser percebida nas simulações envolvendo as bases Optdigits (Figura 4-6), Pendigits (Figura 4-7), bem como a Letter (Figura 4-8). Essa conclusão é importante pois corrobora a obtida por outros trabalhos envolvendo classificadores comuns [23]. Com exceção da base Letter, também é possível verificar uma melhor capacidade de detectar novidade em relação ao modelo NNDD, como na base Soybean.

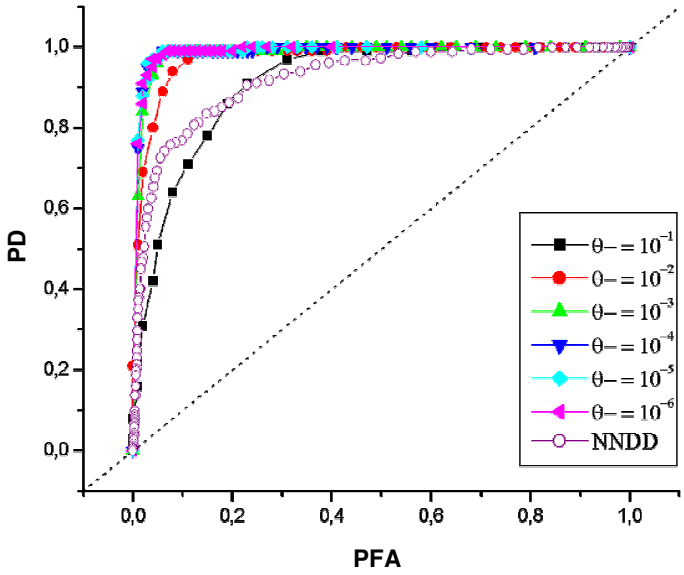

Figura 4-6. Curvas ROC para a base Optdigits, construídas usando NNDD e PNN-DDA com variação de θ-

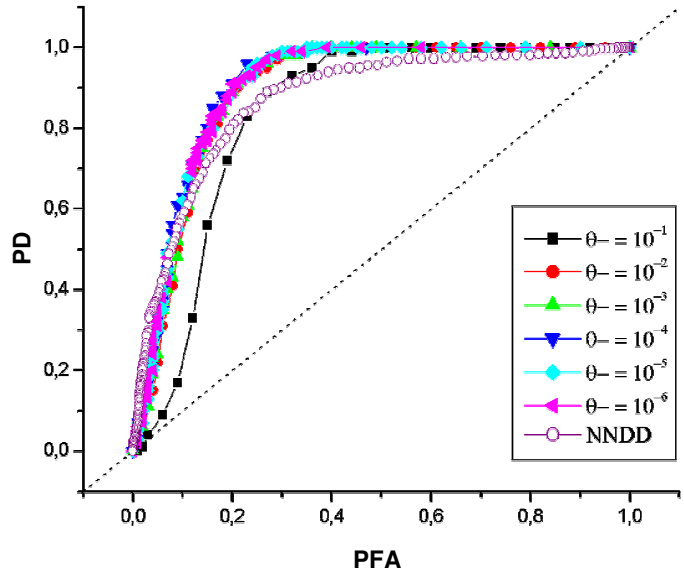

Figura 4-7. Curvas ROC para a base Pendigits, construídas usando NNDD e PNN-DDA com variação de θ-

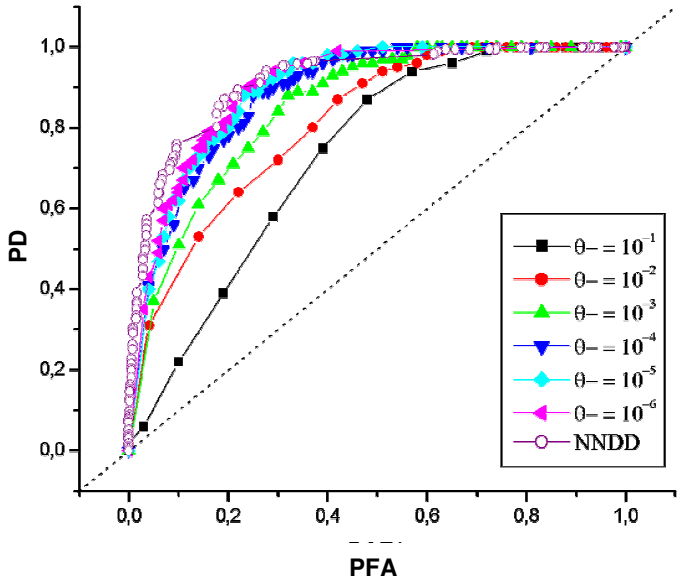

Figura 4-8. Curvas ROC para a base Letter, construídas usando NNDD e PNN-DDA com variação de θ-.

Outro fato importante a ser registrado envolve a complexidade da rede final. A Tabela 4-10 apresenta os resultados obtidos com o modelo NNDD e os resultados da rede PNN-DDA com o valor de θ- que produziu os melhores resultados. Além da melhora da AUC em três das quatro bases analisadas (Soybean, Optdigits e Pendigits), é possível observar que a complexidade final de todas as redes diminui quando são utilizadas redes PNN-DDA, mesmo com os ajustes no valor de θ-. A base Letter, por exemplo, apresentou resultados de acerto na detecção similares aos do modelo NNDD, porém com uma complexidade menor na rede PNN-DDA, mesmo com o valor de θ- igual a 10^{-6}, o que já ocasiona a inclusão de um grande número de neurônios na rede.

A complexidade é um fator importante principalmente para a implementação dos detectores de novidades, pois sistemas mais complexos demandam um maior esforço computacional, bem como uma maior necessidade de recursos físicos para processamento e armazenamento dos dados.

Tabela 4-10. Comparação dos resultados entre NNDD e PNN-DDA com θ- que produz melhores valores

Base	NNDD	PNN-DDA (melhor θ-)
Soybean	0,5706 (448)	0,9171 (282) ($\theta- = 10^{-3}$)
Optdigits	0,9233 (3447)	0,9910 (3410) ($\theta- = 10^{-4}$)
Pendigits	0,8707 (6714)	0,9059 (4360) ($\theta- = 10^{-4}$)
Letter	0,9162 (14435)	0,9009 (12879) ($\theta- = 10^{-6}$)

4.4.2 PNN-DDA x PNN-DDA com poda

Analisando os resultados obtidos com a técnica de poda, é possível verificar que ocorrem melhorias quanto à capacidade de detecção de novidades, bem como quanto ao aspecto de complexidade da rede neural. A Tabela 4-11 apresenta uma comparação entre algumas situações nas cinco bases selecionadas. Para essa análise, foram considerados os resultados da AUC e complexidade para cinco casos possíveis:

1. Simulação NNDD normal;
2. Simulação PNN-DDA, usando θ- igual a 0,2;
3. Simulação PNN-DDA, usando o valor de θ- que produziu os melhores resultados;
4. Simulação PNN-DDA com poda, procurando um limiar η que forneça um valor para AUC próximo ao obtido na simulação 3;
5. Simulação PNN-DDA com poda, procurando um limiar η que forneça um valor para a complexidade similar ao atingido com a execução 2.

O caso (1) foi apresentado para servir de base de referência às demais simulações. A configuração (2) foi usada porque o valor de θ- igual a 0,2 é o proposto por Berthold e Diamond [2], e servirá também como critério de comparação com os resultados obtidos com a poda. A idéia é poder comparar os resultados atuais, com os obtidos da mesma maneira proposta.

Tabela 4-11. Comparação dos resultados das bases em cinco configurações diferentes

Base	**(1) NNDD**	**(2)PNN-DDA θ-= 0,2**	**(3)PNN-DDA [melhor θ-]**	**(4)PNN-DDA com poda**	**(5)PNN-DDA com poda**
Soybean (c:17)	0,5706 (448)	0,3137 (138)	0,9181 (282) [θ-=10^{-3}]	0,9125 (261) η=1,5	0,8933 (137) η=0,1
Optdigits (c: 6)	0,9233 (3447)	0,8484 (961)	0,9910 (3410) [θ-=10^{-4}]	0,9895 (1630) η=0,1	0,9832 (972) η=0,032
Pendigits (c: 3)	0,8707 (6714)	0,8276 (659)	0,9059 (4360) [θ-=10^{-4}]	0,9002 (3697) η=2	0,7485 (658) η=0,121
Letter (c: 3)	0,9162 (14435)	0,6876 (5463)	0,9009 (12879) [θ-=10^{-6}]	0,9325 (9399) η=0,1	0,9215 (5484) η=0,043
Satimage (c: 1)	0,8863 (3755)	0,7990 (2052)	0,8161 (3458) [θ-=10^{-6}]	0,8139 (2775) η=0,03	0,8665 (2047) η=0,0023

A comparação entre as simulações (2) e (3) indica, como mostrado na seção anterior, a melhoria que pode ser obtida através da seleção de θ-, já que o valor para AUC aumenta consideravelmente em relação às redes com θ- = 0,2, demonstrando o ganho na capacidade de detecção de novidade. As melhorias são bem latentes nas bases Soybean e Letter, onde um forte

aumento pode ser visto. Além do aumento para AUC, é possível notar também o aumento na complexidade final da rede, fato este que pode ser fonte de problemas em aplicações práticas.

Para contornar esse problema, a aplicação da técnica de poda aqui proposta é bem interessante. As simulações (4) mostram que valores de AUC próximos a (3) podem ser alcançados mesmo após a diminuição do número de neurônios da rede. Como essa queda é feita através da eliminação de redundâncias, é possível se conseguir redes menores a partir das obtidas em (3), com capacidades de detectar novidade similar. Em algumas bases, como a Letter, houve até aumento no valor de AUC, devido, possivelmente, à eliminação de neurônios responsáveis por classificação incorreta de padrões. Um detalhe importante é que a escolha do limiar de poda η foi feita de forma empírica, através de tentativa e erro, para obtenção da comparação desejada em cada base.

Pode-se também ver que os resultados apresentados mostram melhorias significativas com o uso da poda, quando se objetiva o controle da complexidade. A simulação (5) foi realizada para comparar a capacidade de detecção em redes com complexidade próximas às obtidas em (2). Como pode ser visto na Tabela 4-11, redes com valores próximos de complexidade, possuem valores para AUC melhores em (5) que em (2) na maioria das bases, com exceção da base Pendigits, onde ligeira queda pode ser vista.

De maneira geral, podemos afirmar que resultados melhores na classificação e detecção de novidades são obtidos através do uso das redes PNN-DDA, se compararmos com outros métodos já estabelecidos. E com o uso da técnica de poda analisada, é possível melhorar a capacidade de generalização das redes neurais, com a diminuição da complexidade da rede. Além disso, para algumas bases existe a possibilidade até de melhorar a detecção de novidade, como pode ser visto pelo aumento da métrica AUC na base Letter.

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho teve como objetivos (1) a avaliação do modelo PNN-DDA no tarefa de detecção de novidades, usando a normalização proposta por Berthold e Diamond, bem como (2) a proposta de uma nova técnica para detecção de novidades, envolvendo o uso de PNN-DDA com poda e sua avaliação e comparação com outras técnicas, a saber, NNDD e PNN-DDA sem poda. O trabalho apresenta contribuições científicas, pois confirma que a nova abordagem para redes PNN com o treinamento DDA também pode ser usada com sucesso na área de detecção de novidades. Além disso, o método proposto (PNN-DDA com poda) é também original e mostra-se bastante eficiente em tarefas de detecção de novidades.

Os resultados obtidos para os modelos analisados confirmam que as redes PNN-DDA com poda são ótimas candidatas para detecção de novidade. Seu uso se mostrou robusto na classificação de padrões novos. Além disso, o treinamento é simples, sendo necessários apenas o ajuste de dois parâmetros (θ- e η). Por fim, a capacidade de generalização das redes neurais é conhecida, e a técnica de poda consegue melhorá-la através da diminuição na complexidade das redes obtidas após o treinamento.

Foi revista e utilizada a técnica conhecida como curva ROC para análise e comparação dos resultados obtidos com as simulações. Essa técnica tem se mostrado poderosa para avaliação de classificadores binários e também de uma única classe (*one-class*), sendo seu uso bastante difundido em áreas como a medicina e comunicação de dados.

Para a execução de algumas das simulações, foi utilizada uma biblioteca de funções para Matlab chamada DDTools. Seu uso se mostrou ineficaz para as bases de dados escolhidas, contudo serviu de ponto de partida para a decisão de implementar todas as técnicas de treinamento e teste necessárias. Assim, foi desenvolvido o simulador *Novelty Detector*, que agrega os modelos NNDD, PNN-DDA e PNN-DDA com poda. Além de realizar o treinamento e teste, o simulador fornece a curva ROC e o respectivo valor para AUC.

O desenvolvimento da ferramenta foi importante não só para as simulações, mas a estrutura de classes definida pode servir como base para o desenvolvimento de qualquer nova técnica, ou aprimoramento das já implementadas.

Por fim, todo o trabalho foi desenvolvido com o pensamento voltado para uma área de detecção de novidades. A idéia principal foi dar uma contribuição ao seu desenvolvimento, provendo uma nova técnica e mais dados para análise de pesquisadores.

5.1 Trabalhos Futuros

Como proposta de trabalho futuro existe a intenção de agregar a técnica avaliada a um problema de classificação comum. Assim, seria possível haver um classificador que além de classificar objetos em classes conhecidas teria a capacidade de detectar novidades.

Como os resultados obtidos com as bases *Optdigits* e *Pendigits* mostram uma boa capacidade de detecção, ligadas a reconhecimento de caracteres, e a área de processamento de imagens de documentos históricos [24] tem se desenvolvido trazendo inúmeras oportunidades para aplicação de classificadores com essa funcionalidade, pretende-se aplicar nessa área as técnicas propostas e analisadas neste trabalho.

Outra vertente de atuação se direciona para uma melhor estruturação da ferramenta desenvolvida (*Novelty Detector)*. Isso se deve à necessidade de melhor adequação do projeto a técnicas de programação que permitam uma maior participação em conjunto.

Bibliografia

[1] BERTHOLD, M.; DIAMOND, J. Boosting the Performance of RBF Networks with Dynamic Decay Adjustment. **Advances in Neural Information Processing Systems**. Cambridge, MA, MIT Press, v. 7, p.521–528, 1995.

[2] BERTHOLD, M.; DIAMOND, J. Constructive training of probabilistic neural networks. **Neurocomputing**. Holanda, v. 19, n. 1–3, p.167–183, abr. 1998.

[3] DATA DESCRIPTION TOOLBOX: http://ict.ewi.tudelft.nl/~davidt/dd_tools.html, último acesso em 15 set. 2006.

[4] DUDA, R. O.; HART P. E.; STORK D. G. **Pattern Classification**. 2. ed. : Nova Iorque: Wiley Interscience, 2000. 680p.

[5] ECLIPSE Project: http://www.eclipse.org/, último acesso em: 20 out. 2006.

[6] FAWCETT, T. An introduction to ROC analysis. **Pattern Recognition Letters**. Holanda, v. 27, n. 8, p. 861–874, jun. 2006.

[7] FLACH, P. A.; PRATI R. C. ROCCER: an Algorithm for Rule Learning Based on ROC Analysis. In: International Joint Conference on Artificial Intelligence, 19., 2005, Edimburgo, Escócia. **Proceedings of 19th International Joint Conference on Artificial Intelligence** (IJCAI'2005), São Francisco, USA: Morgan Kaufmann Publishers, ago. 2005. p. 823-828.

[8] GOMES, D. A. O., **Detecção de Intrusão em Redes de Computadores utilizando Classificadores One-Class**, 2006. Trabalho de conclusão de curso de Engenharia da Computação do Departamento de Sistemas Computacionais (DSC) da UPE (http://tcc.dsc.upe.br/20061/DanielGomes.pdf).

[9] HARMELING, S. *et al*. From outliers to prototypes: Ordering data. **Neurocomputing**. Holanda, v. 69, n. 13–15, p.1608–1618, ago. 2006.

[10] HUANG, J.; LING, C. X. Using AUC and accuracy in evaluating learning algorithms. **IEEE Transactions on Knowledge and Data Engineering**. USA, v. 17, n. 3, p. 299–310, mar. 2005.

[11] JAVA: http://java.sun.com/, último acesso em: 16 out. 2006.

[12] JFREECHART, Java Chart Library: http://www.jfree.org/jfreechart/, último acesso em 07 out. 2006

[13] LASKO, T. A. *et al*. The use of receiver operating characteristic curves in biomedical informatics. **Journal of Biomedical Informatics**. Holanda, v. 38, n. 5, p. 404–415, out. 2005.

[14] LGPL, GNU Lesser General Public License: http://www.gnu.org/licenses/lgpl.html, último acesso em 07 out. 2006

[15] MARKOU, M.; SINGH, S. Novelty detection: a review - part 1: statistical approaches. **Signal Processing**. Holanda, v. 83, n. 12, p. 2481-2497, dez. 2003.

[16] MARKOU, M.; SINGH, S. Novelty detection: a review - part 2: neural network based approaches. **Signal Processing**. Holanda, v. 83, n. 12, p. 2499-2521, dez. 2003.

[17] MARKOU, M.; SINGH, S. An Approach to Novelty Detection Applied to the Classification of Image Regions. **IEEE Transactions on Knowledge and Data Engineering**. USA, v. 16, n. 4, p. 396-407, abr. 2004.

[18] MARKOU, M.; SINGH, S. A Neural Network-Based Novelty Detector for Image Sequence Analysis. **IEEE Transactions on Pattern Analysis and Machine Intelligence**. USA, v. 28, n. 10, p. 1664-1677, out. 2006.

[19] MATLAB: http://www.mathworks.com/, último acesso em 15 set. 2006.

[20] OLIVEIRA, A. L. I. *et al*. Improving RBF-DDA Performance on Optical Character Recognition Through Parameter Selection. In: International Conference on Pattern Recognition, 17. , 2004, Cambridge, UK. **Proceedings of the 17th International Conference on Pattern Recognition** (ICPR'2004). USA: IEEE Computer Society Press, 2004. v. 4. p. 625-628.

[21] OLIVEIRA, A. L. I. *et al*. Improving constructive training of RBF networks through selective pruning and model selection. **Neurocomputing**. Holanda, v. 64, p. 537–541, mar. 2005.

[22] OLIVEIRA, A. L. I. *et al*. Integrated method for constructive training of radial basis function networks. **Electronics Letters**. Reino Unido, v. 41, n. 7, p. 429–430, mar. 2005.

[23] OLIVEIRA, A. L. I. *et al*. On the Influence of Parameter θ- on Performance of RBF Neural Networks Trained with the Dynamic Decay Adjustment Algorithm. **International Journal of Neural Systems**. New Jersey, USA, v. 16, n. 4, p. 271-281, ago. 2006.

[24] OLIVEIRA, A. L. I. *et al*. Optical Digit Recognition for Images of Handwritten Historical Documents. In: SIMPÓSIO BRASILEIRO DE REDES NEURAIS, 9., 2006, Ribeirão Preto. **Anais...** Los Alamitos, CA, USA: IEEE Computer Society, 2006. p. 29.

[25] ORIGIN, OriginLab – Scientific Graphing and Analysis Software, versão 6.0, http://www.originlab.com/, último acesso em 21 out. 2006.

[26] RICCI, E.; PERFETTI, R. Improved pruning strategy for radial basis function networks with dynamic decay adjustment. **Neurocomputing**. Holanda, v. 69, n. 13-15, p. 1728–1732, ago. 2006.

[27] SNNS - Stuttgart Neural Network Simulator: http://www-ra.informatik.uni-tuebingen.de/SNNS/, último acesso em 14 set. 2006.

[28] SPECHT, D. F. Probabilistic neural networks and the polynomial Adaline as complementary techniques for classification. **IEEE Transactions on Neural Networks**. USA, v. 1, n. 1, p. 111–121, mar. 1990.

[29] TAX, D. M. J. **One-class classification:** Concept-learning in the absence of counter-examples. 2001. 202f. Tese de Doutorado (Ciência da Computação - Inteligência Artificial). Technische Universiteit Delft, Delft, Holanda, 2001.

[30] UCI Machine Learning Repository: http://www.ics.uci.edu/~mlearn/MLRepository.html, último acesso em 14 set. 2006.

[31] VEP, Visual Editor Project: http://www.eclipse.org/vep, último acesso em 20 out. 2006.

Apêndice A - Código fonte - Treinamento

Será apresentado aqui o código fontes das classes responsáveis pelas rotinas de treinamento dos modelos NNDD, PNN-DDA e PNN-DDA com poda no projeto desenvolvido. Para cada modelo, a classe fornece uma rede de elementos treinados de acordo com a arquitetura. O objetivo é mostrar as funções implementadas neste trabalho.

Classe NNDDTreina.java

```
package upe.dsc.nndd;
/*
 * Created on 13/04/2006
 *
 * A classe NNDDTreina se encarrega de treinar uma rede com
 * os padrões de treinamento oferecidos. Para isso, durante o
 * inicio do treinamento o usuário deve informar qual a classe
 * será considerada como novidade. Os padrões que pertencerem
 * a essa classe não serão armazenados no treinamento. Após o
 * treinamento, a rede conterá todos os padrões que não são da
 * classe "novidade".
 */

public class NNDDTreina{

    private NNDDNetwork rede;
    private LeitorArquivoPat dados;
    private int novidade;
    private Log log;

    /*
     * Metodos inicializadores da classe
     */
    public NNDDTreina(){
        this.rede = null;
        this.dados = null;
    }

    public NNDDTreina(NNDDNetwork r, LeitorArquivoPat d, int nov, Log l){
        this.rede = r;
        this.dados = d;
        this.novidade = nov;
        this.log = l;
```

```
    }

    public NNDDNetwork getTrainedNetwork(){
        return rede;
    }

    public NNDDNetwork train(){
        int numPadroes = dados.getNumPadroes();  //numero de padroes
        int numEntradas = dados.getNumEntradas();  //numero de entradas de cada padrão
        double[] ent = new double[numEntradas];   //variável para o padrao de entrada

        // variável indicativa da classificação correta de um padrao de entrada
        boolean continua;
        int classeAtual; // variavel para indicar classe associada ao padrao lido

        for(int i=0; i < numPadroes; i++){ //treina com todos os padroes
            /* saída para o padrão atual sendo treinado
             */

            classeAtual = dados.getClasse(i);
            continua = true;
            if (classeAtual == novidade){
                continua = false;
            }
            if(continua){
                /* cria um vetor com o padrão de entrada atual para
                 * ser usado na criação dos RBFs no treinamento atual
                 * e na redução dos desvios das rbfs conflitantes
                 */

                double[][] entradaAtual = dados.getEntradas();
                /* armazena o valor do padrao de entrada sendo treinado na variavel
                 */
                for(int m=0; m < numEntradas ; m++){
                    ent[m] = entradaAtual[i][m];
                }

                // Adiciona padrao a rede NNDD, através de um Nodo
                rede.addNode(new Node(ent));
            }
        }
        log.println("   Treinamento concluido!!");
        log.println("   Complexidade da Rede NNDD: " + rede.getComplexity());

        return rede;
    }
}
```

Classe PNNTreina.java

```
package upe.dsc.pnn;

public class PNNTreina {

    // variável que guarda dados de treinamento
    private LeitorArquivoPat dadosTreinamento;
    // variáveis que indicam os parametros theta+ e theta-
    private double thetaMais, thetaMenos;
    // numero de unidades de entrada da rede
    private int numEntradas;
    private PNNNetwork net;
    // identificador da classe escolhida para a deteccao de novidade
    private int novidade;
    private Log log;

    /* ********************************************
     * Funcao set para os parametros theta+ e
     * theta-.
     **********************************************/
    public void setThetaPlus(double valor){ // ajuste do theta+
        this.thetaMais = valor;
    }

    public void setThetaMinus(double valor){ // ajuste do theta-
        this.thetaMenos = valor;
    }

    /* ********************************************
     * Função para definição do valor inicial do desvio padrao na criacao de
     * um novo RBF. As entradas sao um inteiro representando uma classe e um
     * RBF "rb" e a saída eh o menor desvio padrão dentre todos os calculados.
     * O menor desvio eh calculado atraves do calculo entre o centro do novo
     * RBF e os centros dos RBFs de classe diferente. O que produzir um
     * menor valor será o escolhido.
     **********************************************/
    private double getMaiorDesvio(int classe, RBF rb){
        int i,j;
        double menor = 100;
        double dEuclidiana = 0.0;

        for(i=0; i < net.getComplexity(); i++){
            double d;
            RBF r = net.rbfAt(i);
            if (r.getClasse() != classe){
                dEuclidiana=0.0;
                for(j=0;j<numEntradas;j++){
                    dEuclidiana += ( (r.getCentro(j) - rb.getCentro(j)) * (r.getCentro(j) - rb.getCentro(j)) ) ;
                }
                d = Math.sqrt( -((dEuclidiana))/Math.log(thetaMenos));

                if(i==0){
                    menor = d;
                }
                if(d < menor){
```

```
                menor = d;
            }
        }
    }
    return menor;
}

/* ********************************************
 * Função para fazer o ajuste nos RBFs de classe diferente
 * do ultimo padrao analisado. Esse ajuste equivale a uma
 * diminuicao do desvio padrao para que todos os padroes de
 * classe diferente estejam com valores abaixo de theta-.
 * A funcao retorna o numero de RBFs que sofreram a variacao
 * no desvio padrao.
 ********************************************/
private int shrink(double[] entrada, int classe){
    int i,j;
    double d;
    double dEuclidiana;
    int contracao = 0;

    for(i=0; i < net.getComplexity(); i++){
        RBF rb = net.rbfAt(i);
        if (rb.getClasse() != classe){
            dEuclidiana = 0;
            for(j=0;j<numEntradas;j++){
                dEuclidiana += ( (entrada[j] - rb.getCentro(j)) * (entrada[j] - rb.getCentro(j)) ) ;
            }
            d = Math.sqrt( -((dEuclidiana))/Math.log(thetaMenos));
            if(d < rb.getDesvio()){
                rb.setDesvio(d);
                contracao++;
            }
        }
    }
    return(contracao);
}

/* ********************************************
 * Funcao para eliminar da rede  os  elementos
 * com pesos igual a zero. Esses elementos sao
 * incluidos nas fases iniciais do treinamento
 * mas são substituidos por elementos melhores
 ********************************************/
private void eliminaElementosDescnecessarios(){
    int numRBFs = net.getComplexity();
    RBF rb;

    for (int i = numRBFs-1; i>=0; i--){
        rb = net.rbfAt(i);
        System.out.println(rb.getPeso());
        if (rb.getPeso() == 0){
            net.removeRBF(rb);
        }
    }
    numRBFs = net.getComplexity();
}
```

```
/* ********************************************
 * Funcao para normalizar o valor dos pesos dos
 * RBFs pertencentes a uma mesma classe
 ********************************************/
private void normalizaPesos(){
    int numClasses = dadosTreinamento.getNumSaidas();
    int numRBFs = net.getComplexity();
    double tempPesos[] = new double[numRBFs];
    int somaPesos;
    RBF rb;

    for (int classe = 0; classe < numClasses; classe++){
        somaPesos = 0;
        for (int i = 0; i<numRBFs; i++){
            rb = net.rbfAt(i);
            if (rb.getClasse() == classe)
                somaPesos += rb.getPeso();
        }
        for (int i = 0; i<numRBFs; i++){
            rb = net.rbfAt(i);
            if (rb.getClasse() == classe)
                tempPesos[i] = (rb.getPeso()) / somaPesos;
        }
    }
    for (int i = 0; i<numRBFs; i++){
        rb = net.rbfAt(i);
        rb.setPeso(tempPesos[i]);
    }
}

/* ********************************************
 * Funcao para zerar o valor no atributo que armazena
 * os pesos das conexoes aos RBFs.
 ********************************************/
private void resetaPesos(){
    for(int i=0; i< net.getComplexity(); i++){
        RBF r = net.rbfAt(i);
        r.setPeso(0.0);
    }
}

private int[] treina1Epoca(){
    int ajusteRede[] = {0, 0, 0};
    //numero de padroes
    int numPadroes = dadosTreinamento.getNumPadroes();
    // numero total de classes
    int numSaidas = dadosTreinamento.getNumSaidas();
    this.net.setNumTotalClasses(numSaidas);
    //variável para o padrao de entrada
    double[] ent = new double[numEntradas];
    //variável indicativa da classificação correta de um padrao de entrada
    boolean continua, classificado;
```

```
for(int i=0; i < numPadroes; i++){ //treina com todos os padroes
    /* saída para o padrão atual sendo treinado
     */
    int classeAtual = dadosTreinamento.getClasse(i);
    continua = true;
    if (classeAtual == novidade){
        continua = false;
    }
    if(continua){
        /* cria um vetor com o padrão de entrada atual para
         * ser usado na criação dos RBFs no treinamento atual
         * e na redução dos desvios das rbfs conflitantes
         */

        double[][] entradaAtual = dadosTreinamento.getEntradas();
        /* armazena o valor do padrao de entrada sendo treinado na variavel
         */
        for(int m=0; m < numEntradas ; m++){
            ent[m] = entradaAtual[i][m];
        }

        /* se não existe nenhuma RBF , cria a primeira
        */
        if(net.getComplexity() == 0){
            BF rb = new RBF(numEntradas);

            // configura o centro da RBF com o padrao de entrada atual
            for(int n=0; n < numEntradas ; n++){
                rb.setCentro(n,ent[n]);
            }
            // peso eh definido com o valor padrao
            b.setPeso(1.0);
            // classe do RBF eh definida como a atual
            b.setClasse(classeAtual);
            // o desvio do RBF é colocado com um valor alto para sofrer ajuste
            rb.setDesvio(100000);
            // RBF criado é incluido no vetor
            net.addRBF(rb);
            ajusteRede[0]++;
            log.println("RBF created for pattern " + i);
        }
        else {
         /* flag não deixa criar outra RBF se já existe uma RBF
          * da mesma classe e a função de ativacao dela for maior
          * que theta+ (threshold positivo)
          */
            classificado = false;
         /* para cada RBF testa se o padrão dado é da mesma classe
          * e testa se o R(x) é maior que o theta+ , se
          * sim incrementa o peso da RBF de maior valor da funcao
          */
            double maiorResultado = 0.0;  // maior saida da f gaussiana
            // indice do RBF que retorna a maior funcao de ativacao
            int indiceRBF =0;
            for(int j=0; j < net.getComplexity(); j++){
                RBF rb = net.rbfAt(j);
                if( rb.getClasse() == classeAtual ){
```

```
                    double resultadoAtual = rb.funcaoRadial(dadosTreinamento, i);
                    if((resultadoAtual >= this.thetaMais) && (maiorResultado < resultadoAtual)){
                        maiorResultado = resultadoAtual;
                        indiceRBF = j;
                        classificado = true;
                    }
                }
            }
            /* se flag == true, já existe RBF que cobre o valor do
             * padrao e não eh necessario criar outro RBF. O peso
             * do RBF que o classificou é incrementado de um e a
             * iteração vai para o próximo padrão de entrada.
             */
            if(classificado == true){
                RBF temp = net.rbfAt(indiceRBF);
                temp.setPeso(temp.getPeso() + 1);
                ajusteRede[1]++;
            }
            /* caso contrario, nenhum RBF obteve uma funcao de ativacao
             * com R(x) ( onde x é o padrão atual de entrada ) maior que
             * theta+. Neste caso, cria-se outra RBF para a classe atual
             * com centro no padrao
             */
            else{
                RBF rb = new RBF(numEntradas);
                // configura o centro da RBF com o padrao de entrada atual
                for(int n=0; n < numEntradas; n++){
                    rb.setCentro(n,ent[n]);
                }
                // peso é definido com o valor padrao
                rb.setPeso(1.0);
                // classe do RBF eh definida como a atual
                rb.setClasse(classeAtual);
                /* o desvio do RBF é calculado pela funcao que retorna
                 * o maior valor para o desvio que não classifica
                 * corretamento os padroes de classe diferente
                 */
                rb.setDesvio(getMaiorDesvio(classeAtual, rb));
                // RBF criado é incluido no vetor
                net.addRBF(rb);
                ajusteRede[0]++;
            } //end else
        }
        ajusteRede[2] += shrink(ent, classeAtual);
    }
  }
  log.println("   Amount of used RBFs: " + net.getComplexity() +
          "    Adjustments made: " + ajusteRede[0] + ", " + ajusteRede[1] + ", " + ajusteRede[2]);
  return ajusteRede;
}

private void treinaNEpocas(){
  int j=0;
  int[] mudancas;
  while(true){
      resetaPesos();
      log.println("Traning age: " + (++j));
```

```
            mudancas = treina1Epoca();
            if((mudancas[0] + mudancas[2]) == 0)
                break;
        }
    }

    public PNNTreina(PNNNetwork _net, LeitorArquivoPat _trainData, int _nov, Log _log){
        this.dadosTreinamento = _trainData;
        this.numEntradas = this.dadosTreinamento.getNumEntradas();

        this.novidade = _nov;

        this.log = _log;

        this.net = _net;
    }

    public PNNNetwork train(){
        this.treinaNEpocas();

        this.eliminaElementosDescnecessarios();

        this.normalizaPesos();

        return net;
    }
}
```

Classe PrunPNNTreina.java

```
package upe.dsc.prun_pnn;

public class PrunPNNTreina {

    private static PrunPNNTreina ref;
    private LeitorArquivoPat dadosTreinamento;      // variável que guarda dados de treinamento
    private double thetaMais, thetaMenos;           // variáveis que indicam os parametros theta+ e theta-
    private PrunPNNNetwork net;
    private PrunPNNNetwork prunedNet;
    private int novidade;                           // identificador da classe escolhida para a deteccao de novidade
    private Log log;

    /* ********************************************
     * Funcoes set para os parametros theta+ e
     * theta-.
     **********************************************/
    public void setThetaPlus(double valor){ // ajuste do theta+
        this.thetaMais = valor;
    }

    public void setThetaMinus(double valor){ // ajuste do theta-
        this.thetaMenos = valor;
    }

    /**
     * @param dadosTreinamento the dadosTreinamento to set
     */
    public void setDadosTreinamento(LeitorArquivoPat dadosTreinamento) {
        this.dadosTreinamento = dadosTreinamento;
    }

    /**
     * @param log the log to set
     */
    public void setLog(Log log) {
        this.log = log;
    }

    /**
     * @param net the net to set
     */
    public void setNet(PrunPNNNetwork net) {
        this.net = net;
    }

    /**
     * @param novidade the novidade to set
     */
    public void setNovidade(int novidade) {
        this.novidade = novidade;
    }
```

```
/* ********************************************
 * Função para definição do valor inicial do desvio padrao na criacao de
 * um novo RBF. As entradas sao um inteiro representando uma classe e um
 * RBF "rb" e a saída eh o menor desvio padrão dentre todos os calculados.
 * O menor desvio eh calculado atraves do calculo entre o centro do novo
 * RBF e os centros dos RBFs de classe diferente. O que produzir um
 * menor valor será o escolhido.
 ********************************************/
private double getMaiorDesvio(int classe, PrunRBF rb){
    int i,j;
    double menor = 100;
    double dEuclidiana = 0.0;

    PrunRBF r;
    for(i=0; i < net.getComplexity(); i++){
        double d;
        r = net.rbfAt(i);
        if (r.getClasse() != classe){
            dEuclidiana=0.0;
            for(j=0;j<dadosTreinamento.getNumEntradas();j++){
                dEuclidiana += ( (r.getCentro(j) - rb.getCentro(j)) * (r.getCentro(j) - rb.getCentro(j)) ) ;
            }
            d = Math.sqrt( -((dEuclidiana))/Math.log(thetaMenos));
            if(i==0){
                menor = d;
            }
            if(d < menor){
                menor = d;
            }
        }
    }
    r = null;
    return menor;
}

/* ********************************************
 * Função para fazer o ajuste nos RBFs de classe diferente
 * do ultimo padrao analisado. Esse ajuste equivale a uma
 * diminuicao do desvio padrao para que todos os padroes de
 * classe diferente estejam com valores abaixo de theta-.
 * A funcao retorna o numero de RBFs que sofreram a variacao
 * no desvio padrao.
 ********************************************/
private int shrink(double[] entrada, int classe){
    int i,j;
    double d;
    double dEuclidiana;
    int contracao = 0;

    PrunRBF rb;
    for(i=0; i < net.getComplexity(); i++){
        rb = net.rbfAt(i);
        if (rb.getClasse() != classe){
            dEuclidiana = 0;
            for(j=0;j<dadosTreinamento.getNumEntradas();j++){
                dEuclidiana += ( (entrada[j] - rb.getCentro(j)) * (entrada[j] - rb.getCentro(j)) ) ;
            }
```

```
                d = Math.sqrt( -((dEuclidiana))/Math.log(thetaMenos));
                if(d < rb.getDesvio()){
                    rb.setDesvio(d);
                    contracao++;
                }
            }
        }
        rb = null;
        return(contracao);
    }

    /* ********************************************
     * Funcao para calcular o produto interno de dois
     * vetores
     **********************************************/

    public double dot_product(double[] vector1, double[] vector2){
        double dot_prod = 0;
        for (int i = 0; i < vector1.length; i++){
            dot_prod += (vector1[i] * vector2[i]);
        }
        return dot_prod;
    }
    /* ********************************************
     * Funcao para normalizar o valor dos pesos dos
     * RBFs pertencentes a uma mesma classe
     **********************************************/
    private void normalizaPesos(PrunPNNNetwork network){
        //int numClasses = dadosTreinamento.getNumSaidas();
        int numRBFs = network.getComplexity();
        double tempPesos[] = new double[numRBFs];
        int somaPesos;
        PrunRBF rb;

        for (int classe = 0; classe < dadosTreinamento.getNumSaidas(); classe++){
            somaPesos = 0;
            for (int i = 0; i<numRBFs; i++){
                rb = network.rbfAt(i);
                if (rb.getClasse() == classe)
                    somaPesos += rb.getPeso();
            }
            for (int i = 0; i<numRBFs; i++){
                rb = network.rbfAt(i);
                if (rb.getClasse() == classe)
                    tempPesos[i] = (rb.getPeso()) / somaPesos;
            }
        }
        for (int i = 0; i<numRBFs; i++){
            rb = network.rbfAt(i);
            rb.setPeso(tempPesos[i]);
        }
        tempPesos = null;
        rb = null;
    }
```

```
/* ******************************************
 * Funcao para zerar o valor no atributo que armazena
 * os pesos das conexoes aos RBFs.
 ******************************************/
private void resetaPesos(){
    PrunRBF r;
    for(int i=0; i< net.getComplexity(); i++){
        r = net.rbfAt(i);
        r.setPeso(0.0);
    }
    r = null;
}

private int[] treina1Epoca(){
    int ajusteRede[] = {0, 0, 0};
    double[] ent = new double[dadosTreinamento.getNumEntradas()];        //variável para o padrao de entrada
    /* variável indicativa da classificação correta de um padrao de entrada
     */
    boolean continua, classificado;

    for(int i=0; i < dadosTreinamento.getNumPadroes(); i++){ //treina com todos os padroes
        /* saída para o padrão atual sendo treinado
         */
        int classeAtual = dadosTreinamento.getClasse(i);
        continua = true;
        if (classeAtual == novidade){
            continua = false;
        }
        if(continua){
            /* cria um vetor com o padrão de entrada atual para
             * ser usado na criação dos RBFs no treinamento atual
             * e na redução dos desvios das rbfs conflitantes
             */

            double[][] entradaAtual = dadosTreinamento.getEntradas();
            /* armazena o valor do padrao de entrada sendo treinado na variavel
             */
            for(int m=0; m < dadosTreinamento.getNumEntradas() ; m++){
                ent[m] = entradaAtual[i][m];
            }

            /* se não existe nenhuma RBF , cria a primeira
            */
            if(net.getComplexity() == 0){
                PrunRBF rb = new PrunRBF(dadosTreinamento.getNumEntradas());

                // configura o centro da RBF com o padrao de entrada atual
                for(int n=0; n < dadosTreinamento.getNumEntradas() ; n++){
                    rb.setCentro(n,ent[n]);
                }
                // peso eh definido com o valor padrao
                rb.setPeso(1.0);
                // classe do RBF eh definida como a atual
                rb.setClasse(classeAtual);
                // o desvio do RBF é colocado com um valor alto para sofrer ajuste
                rb.setDesvio(100000);
                // RBF criado é incluido no vetor
```

```
    net.addRBF(classeAtual, rb);
    ajusteRede[0]++;
    log.println("Criado RBF no padrao " + i);
}
else {
 /* flag não deixa criar outra RBF se já existe uma RBF
  * da mesma classe e a função de ativacao dela for maior
  * que theta+ (threshold positivo)
  */
    classificado = false;
 /* para cada RBF testa se o padrão dado é da mesma classe
  * e testa se o R(x) é maior que o theta+ , se
  * sim incrementa o peso da RBF de maior valor da funcao
  */
    double maiorResultado = 0.0;        // maior saida da função de ativação gaussiana
    int indiceRBF =0;                   // indice do RBF que retorna a maior funcao de ativacao
    PrunRBF tempRbf;
    for(int j=0; j < net.getComplexity(); j++){
        tempRbf = net.rbfAt(j);
        if( tempRbf.getClasse() == classeAtual ){
            double resultadoAtual = tempRbf.funcaoRadial(dadosTreinamento, i);
            if((resultadoAtual >= this.thetaMais) && (maiorResultado < resultadoAtual)){
              maiorResultado = resultadoAtual;
              indiceRBF = j;
              classificado = true;
            }
        }
    }
    /* se flag == true, já existe RBF que cobre o valor do
     * padrao e não eh necessario criar outro RBF. O peso
     * do RBF que o classificou é incrementado de um e a
     * iteração vai para o próximo padrão de entrada.
     */
    if(classificado == true){
        tempRbf = net.rbfAt(indiceRBF);
        tempRbf.setPeso(tempRbf.getPeso() + 1);
        ajusteRede[1]++;
    }

    /* caso contrario, nenhum RBF obteve uma funcao de ativacao
     * com R(x) ( onde x é o padrão atual de entrada ) maior que
     * theta+. Neste caso, cria-se outra RBF para a classe atual
     * com centro no padrao
     */
    else{
        PrunRBF rb = new PrunRBF(dadosTreinamento.getNumEntradas());
        // configura o centro da RBF com o padrao de entrada atual
        for(int n=0; n < dadosTreinamento.getNumEntradas(); n++){
            rb.setCentro(n,ent[n]);
        }
        // peso é definido com o valor padrao
        rb.setPeso(1.0);
        // classe do RBF eh definida como a atual
        rb.setClasse(classeAtual);
        /* o desvio do RBF é calculado pela funcao que retorna
         * o maior valor para o desvio que não classifica
         * corretamento os padroes de classe diferente
```

```
                    */
                    rb.setDesvio(getMaiorDesvio(classeAtual, rb));
                    // RBF criado é incluido no vetor
                    net.addRBF(classeAtual, rb);
                    ajusteRede[0]++;
                } //end else
                tempRbf = null;
            }
            ajusteRede[2] += shrink(ent, classeAtual);
        }
    }
    log.println("   Quantidade atual de RBFs: " + net.getComplexity() +
            "    Ajustes realizados: " + ajusteRede[0] + ", " + ajusteRede[1] + ", " + ajusteRede[2]);
    return ajusteRede;
}

private void treinaNEpocas(){
    int j=0;
    int[] mudancas;
    while(true){
        resetaPesos();
        log.println("Training age: " + (++j));
        mudancas = treina1Epoca();
        if((mudancas[0] + mudancas[2]) == 0)
            break;
    }
}

public PrunPNNNetwork getPrunedNetwork(double pruningFactor){
    this.prunedNet = (PrunPNNNetwork)net.clone();

    int classe;
    PrunRBF rbf;
    double[] actvFunctVector;
    int pos;

    for (int r = 0; r < this.prunedNet.getComplexity(); r++){
        rbf = this.prunedNet.rbfAt(r);
        classe = rbf.getClasse();
        actvFunctVector = new double[this.dadosTreinamento.getNumPadroesNaClasse(classe)];
        pos = 0;
        for(int pat = 0; pat < dadosTreinamento.getNumPadroes(); pat++){
            if (this.dadosTreinamento.getClasse(pat) == classe){
                actvFunctVector[pos] = rbf.funcaoRadial(this.dadosTreinamento, pat);
                pos++;
            }
        }
        rbf.setAFVector(actvFunctVector);
    }
    actvFunctVector = null;

    double dotProduct;
    PrunRBF rbi;
    PrunRBF rbj;
    for(int cl = 0; cl < this.prunedNet.getNumTotalClasses(); cl++){
        for(int i = 0; i < (this.prunedNet.getNumRBFnaClasse(cl) - 1); i++){
            for(int j = (i + 1); j < this.prunedNet.getNumRBFnaClasse(cl); j++){
```

```
                    rbi = this.prunedNet.rbfAt(cl, i);
                    rbj = this.prunedNet.rbfAt(cl, j);
                    dotProduct = this.dot_product(rbi.getAFVector(), rbj.getAFVector());
                    if (dotProduct > pruningFactor){
                        if (rbi.getPeso() > rbj.getPeso()){
                            rbj.setPrunning(true);
                        }
                        else{
                            if (rbi.getPeso() < rbj.getPeso()){
                              rbi.setPrunning(true);
                            }
                            else{
                              if (rbi.getDesvio() > rbj.getDesvio()){
                                        rbj.setPrunning(true);

                              }
                              else{
                                        rbi.setPrunning(true);
                              }
                            }
                        }
                    }
                }
            }
        }
        rbi = null;
        rbj = null;

        // Após identificar os RBFs a serem retirados, é só realizar a poda na rede
        this.prunedNet.prune();
        this.normalizaPesos(prunedNet);
        return prunedNet;
    }

    private PrunPNNTreina(){
    }

    public static PrunPNNTreina getPrunPNNTreinaObject(){
        if (ref == null)
            // it's ok, we can call this constructor
            ref = new PrunPNNTreina();
        return ref;
    }

    public PrunPNNNetwork getTrainedNetwork(){
        this.net.setNumTotalClasses(this.dadosTreinamento.getNumSaidas());
        this.treinaNEpocas();
        return net;
    }
}
```

Apêndice B - Código fonte – Teste

Será apresentado aqui o código fontes das classes responsáveis pelas rotinas de teste das redes dos modelos de dados após treinamento. Ao final do treinamento, a classe se encarrega de fornecer a saída gerada pelos padrões fornecidos. O objetivo aqui é mostrar as funções implementadas.

Classe NNDDTeste.java

```
package upe.dsc.nndd;

public class NNDDTeste{

    private NNDDNetwork rede;
    private LeitorArquivoPat dados;
    private Log log;
    private double retorno[];

    public NNDDTeste(){
        this.rede = null;
        this.dados = null;
    }

    public NNDDTeste(LeitorArquivoPat dt, NNDDNetwork r, int nov, Log l){
        this.rede = r;
        this.dados = dt;
        this.log = l;
        this.retorno = new double[dt.getNumPadroes()];
    }

    public double[] getDataNetAnswer(){
        return this.retorno;
    }

    public void test(){
        int numPadroes = dados.getNumPadroes();  //numero de padroes
        int numEntradas = dados.getNumEntradas(); //numero de entradas de cada padrao

        double[] ent = new double[numEntradas];       //variável para o padrao de entrada

        /* cria um vetor com o padrão de entrada atual para
         * ser usado no treinamento atual
         */
        double[][] entradaAtual = dados.getEntradas();
```

```
        for(int pat=0; pat < numPadroes; pat++){ //treina com todos os padroes
            // armazena o valor do padrao de entrada sendo treinado na variavel
            for(int j=0; j < numEntradas ; j++){
                ent[j] = entradaAtual[pat][j];
            }
            retorno[pat] = rede.netAnswer(new Node(ent));
        }
    }
}
```

Classe PNNTesta.java

```
package upe.dsc.pnn;

public class PNNTesta {
    private PNNNetwork rede;
    private int numPadroesTeste;
    private int classeNovidade;
    private double probNovidade;
    private LeitorArquivoPat dadosTeste;
    private double[][] retornoRede;

    public double[][] getDataNetAnswer(){
        return retornoRede;             }

    private void normalizacao(double[] vetor){
        int tamanhoVetor = vetor.length;
        double soma = 0;
        for (int i = 0; i < tamanhoVetor; i++){
            soma += vetor[i];
        }
        for (int i = 0; i < tamanhoVetor; i++){
            vetor[i] = vetor[i]/soma;
        }
    }

    public void test(){
        double[] saidaRede;
        double[] ent = new double[this.dadosTeste.getNumEntradas()];
        double[][] entradas = this.dadosTeste.getEntradas();
        for (int entrada = 0; entrada < numPadroesTeste; entrada++){
            for (int i = 0; i < this.dadosTeste.getNumEntradas(); i++){
                ent[i] = entradas[entrada][i];
            }
            saidaRede = rede.netAnswer(ent);//this.getSaidaCalculada(entrada);
            /*
             * Como o modelo PNN nao usa padroes da classe escolhida como novidade durante o treinamento,
             * todos os padroes pertencentes a essa classe terao como saida o valor zero na classe. Como essa saida
             * será usada para avaliar novidade, seu valor é definido durante o teste como sendo igual ao valor de
             * theta-, antes da normalizacao da saida, conforme especificado por Berthold em seu artigo. */
            saidaRede[this.classeNovidade] = this.probNovidade;
            this.normalizacao(saidaRede);
            for (int i = 0; i < this.dadosTeste.getNumSaidas(); i++){
                this.retornoRede[entrada][i] = saidaRede[i];
            }
        }
    }

    public PNNTesta(LeitorArquivoPat _testData, PNNNetwork _net, int novelty, double thetaMinus, Log _log){
        this.rede = _net;
        this.dadosTeste = _testData;
        this.numPadroesTeste = _testData.getNumPadroes();
        this.classeNovidade = novelty;
        this.probNovidade = thetaMinus;
        this.retornoRede = new double[this.numPadroesTeste][rede.getNumTotalClasses()];
    }
}
```

Classe PrunPNNTesta.java

```
package upe.dsc.prun_pnn;

public class PrunPNNTesta {
    private PrunPNNNetwork rede;
    private int numPadroesTeste;
    private int classeNovidade;
    private double probNovidade;
    private LeitorArquivoPat dadosTeste;
    private double[][] retornoRede;

    public double[][] getDataNetAnswer(){
        return retornoRede;             }

    private void normalizacao(double[] vetor){
        int tamanhoVetor = vetor.length;
        double soma = 0;
        for (int i = 0; i < tamanhoVetor; i++){
            soma += vetor[i];
        }
        for (int i = 0; i < tamanhoVetor; i++){
            vetor[i] = vetor[i]/soma;
        }
    }

    public void test(){
        double[] saidaRede;
        double[] ent = new double[this.dadosTeste.getNumEntradas()];
        double[][] entradas = this.dadosTeste.getEntradas();
        for (int entrada = 0; entrada < numPadroesTeste; entrada++){
            for (int i = 0; i < this.dadosTeste.getNumEntradas(); i++){
                ent[i] = entradas[entrada][i];
            }
            saidaRede = rede.netAnswer(ent);
            /*
             * Como o modelo PNN nao usa padroes da classe escolhida como novidade durante o treinamento,
             * todos os padroes pertencentes a essa classe terao como saida o valor zero na classe. Como essa
             * saida será usada para avaliar novidade, seu valor é definido durante o teste como sendoigual ao
             * valor de theta-, antes da normalizacao da saida, conforme especificado por Berthold em seu artigo. */
            saidaRede[this.classeNovidade] = this.probNovidade;
            this.normalizacao(saidaRede);
            for (int i = 0; i < this.dadosTeste.getNumSaidas(); i++){
                this.retornoRede[entrada][i] = saidaRede[i];
            }
        }
    }

    public PrunPNNTesta(LeitorArquivoPat _testData, PrunPNNNetwork _net, int nov, double thMinus, Log _log){
        this.rede = _net;
        this.dadosTeste = _testData;
        this.numPadroesTeste = _testData.getNumPadroes();
        this.classeNovidade = nov;
        this.probNovidade = thMinus;
        this.retornoRede = new double[this.numPadroesTeste][this.rede.getNumTotalClasses()];
    }
}
```

Apêndice B - Código fonte – Curvas ROC

Será apresentado aqui o código fontes das classes responsáveis por gerar os pontos necessários para o desenho das curvas ROC, bem como os valores de AUC para a curva. Devido à similaridade entre as classes, pois as funções implementadas são as mesmas, apenas será colocada a classe pertencente ao pacote que implementa PNN-DDA com poda (`package upe.dsc.prun_pnn`).

Classe PrunPNNRoc.java

```
package upe.dsc.prun_pnn;

public class PrunPNNRoc extends Roc{

    private double[][] retornoDados;

    public PrunPNNRoc(LeitorArquivoPat d, int nov, double ret[][]) {
        super(d, nov, new double[1]);
        this.retornoDados = ret;
    }

    public double[][] getPontosCurva(double limite1, double limite2, int numPontos){
        double truePositive, falsePositive, trueNegative, falseNegative;
        double pod, pofa;
        int classeAtual;//, novidade;

        /* a variavel pontosRoc representa uma tabela com 3 colunas. O numero de linhas
        /* identifica a quantidade desejada de pontos para formar a curva. A primeira
         * coluna representa o valor de limiar desejado para a analise. A segunda indica
         * a probabilidade de um falso alarme (PFA). A terceira mostra a probabilidade de
         * se acertar na deteccao de novidade.
         */
        pontosRoc = new double[numPontos][3];
        double intervalo;

        if (numPontos > 1)
            intervalo = (limite2 - limite1) / (double)(numPontos - 1);
        else
            intervalo = 0;

        pontosRoc[0][0] = limite1;
```

```
        for (int p = 1; p < (numPontos - 1); p++ ){
            pontosRoc[p][0] = pontosRoc[p-1][0] + intervalo;
        }
        pontosRoc[numPontos-1][0] = limite2;

        for (int lim = 0; lim < numPontos; lim++){
            truePositive = 0;
            falseNegative = 0;
            trueNegative = 0;
            falsePositive = 0;

            for (int i = 0; i < dados.getNumPadroes(); i++){
                classeAtual = dados.getClasse(i);
                if (classeAtual == this.novidade){
                    if (retornoDados[i][this.novidade] > pontosRoc[lim][0]){
                        truePositive++;
                    }
                    else{
                        falseNegative++;
                    }
                }
                else{
                    if (retornoDados[i][this.novidade] > pontosRoc[lim][0]){
                        falsePositive++;
                    }
                    else{
                        trueNegative++;
                    }
                }

                pofa = (falsePositive)/(trueNegative + falsePositive);
                pod = (truePositive)/(truePositive + falseNegative);
                pontosRoc[lim][1] = pofa;
                pontosRoc[lim][2] = pod;
            }
        }
        return pontosRoc;
    }

    public double getAUC(){
        double auc = 0;
        double altura;
        double somaDasBases;
        if (pontosRoc != null){
            for (int lim = 0; lim < pontosRoc.length - 1; lim++){
                altura = pontosRoc[lim][1]-pontosRoc[lim+1][1];
                somaDasBases = pontosRoc[lim][2] + pontosRoc[lim+1][2];
                auc += (somaDasBases * altura / 2);
            }
        }
        return auc;
    }
}
```