Diplomarbeit

Entwurf und Implementierung eines UNIX–basierten Softwaretools zur Realisierung des Neuro–Fuzzy–Klassifikationsansatzes NEFCLASS

von

cand. inform. Thomas Hoferichter

Technische Universität Braunschweig

Institut für Betriebssysteme und Rechnerverbund

Aufgabenstellung und Betreuung:

Prof. Dr. R. Kruse und

Dr. D. Nauck

Braunschweig, 9. August 1996

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, 9. August 1996

Kurzfassung

In der vorliegenden Arbeit wird eine Softwareimplementierung des NEFCLASS–Modells vorgestellt. NEFCLASS ist ein Neuro–Fuzzy–Ansatz, der auf einem generischen Fuzzy–Perceptron basiert und im Bereich der Datenklassifikation eingesetzt werden kann. Der Vorteil dieses Modells gegenüber anderen in diesem Bereich einsetzbarer Verfahren wie Neuronaler Netze oder statistischer Verfahren ist, daß ein NEFCLASS–System in Form von linguistischen Wenn–Dann–Regeln interpretiert werden kann. Außerdem kann vorhandenes Vorwissen leicht in ein NEFCLASS–System integriert werden. In dieser Arbeit wird zunächst die Architektur des NEFCLASS–Modells sowie dessen überwachter Lernalgorithmus vorgestellt. Im Anschluß daran wird die Implementierung des UNIX–basierten Softwaretools NEFCLASS erläutert. Abschließend werden die Ergebnisse einiger Datenanalysen präsentiert, die mit diesem Softwaretool erzielt werden konnten.

Abstract

This thesis presents a software implementation of the NEFCLASS model. NEFCLASS is a neuro-fuzzy approach based on a generic fuzzy perceptron that can be used for the classification of data. The advantage of the NEFCLASS model compared with other approaches like neural networks or statistical proceedings that can be used for data classification, is that a NEFCLASS system can be interpreted in form of linguistic If–then–rules. Furthermore, prior knowledge about the learning task can be integrated in the system easily. This thesis first presents the architecture of the NEFCLASS model and its supervised learning algorithm. In the following the implementation of the UNIX–based software tool NEFCLASS will be explained. Finally, the obtained results of some data analysis using NEFCLASS will be presented.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In der vorliegenden Arbeit wird ein Neuro–Fuzzy–Ansatz zur Datenklassifikation, das NEFCLASS–Modell (**NE**uro–**F**uzzy–**CLASS**ification), vorgestellt. Dieses Modell vereint Konzepte von Neuronalen Netzen und Fuzzy–Systemen und wurde insbesondere im Hinblick auf eine gute Interpretierbarkeit entwickelt. Darüberhinaus erlaubt das Modell die einfache Integration von Vorwissen.

Im Rahmen dieser Diplomarbeit wurde das NEFCLASS–Modell in einem kommandogesteuerten Softwaretool unter dem Betriebssystem UNIX implementiert und anhand von Beispieldaten getestet. Beim Entwurf dieses kommandogesteuerten Softwaretools wurde besonderer Wert auf eine möglichst gute Portierbarkeit der Software auf andere Rechner– und Betriebssystemplattformen gelegt, so daß das Softwaretool nicht nur auf UNIX–basierten Rechnern benutzt werden kann. Darüberhinaus wurde speziell für das X–Window–System eine graphische Benutzeroberfläche für das kommandogesteuerte Softwaretool entwickelt, die eine einfache Bedienung des Programms erlaubt.

Zum Aufbau der Diplomarbeit

Im Kapitel 2 werden zunächst die grundsätzlichen Konzepte von Neuronalen Netzen und Fuzzy–Systemen erläutert, wobei es nicht das Ziel dieser Einleitung ist, eine gründliche systematische Einführung in diese Themengebiete zu geben. Der interessierte Leser sei hier auf die weiterführende Literatur verwiesen, wenn er nicht bereits mit den Grundlagen Neuronaler Netze und den Konzepten von Fuzzy–Systemen vertraut ist. Im Kapitel 2 finden sich mehrere Hinweise auf weiterführende Literatur. Am Ende dieses Kapitels werden die vielversprechenden Möglichkeiten der Kopplung von Neuronalen Netzen mit Fuzzy–Systemen aufgezeigt, durch die es möglich ist, lernfähige, interpretierbare Systeme zu konstruieren.

Im Anschluß an diese Einleitung führt Kapitel 3 das formale NEFCLASS–Modell und das dazugehörige Lernverfahren ein, wobei die Grundlagen über Neuronale Netze und

Fuzzy–Systeme als bekannt vorausgesetzt werden.

Im Rahmen dieser Arbeit wurde das NEFCLASS–Modell prototypisch in einem Softwaretool implementiert. Das Kapitel 4 beschreibt den Systemaufbau sowie die wichtigsten Datenstrukturen des im Rahmen dieser Arbeit entwickelten Softwaretools. Dieses Kapitel richtet sich daher in erster Linie an Leser und Leserinnen, die die softwaretechnische Umsetzung des Modells verstehen wollen, z.B. weil sie die Software verbessern und / oder erweitern wollen.

Das sich daran anschließende Kapitel 5 stellt dann Ergebnisse von Datenklassifikationen vor, die mit dem Programm NEFCLASS bei der Klassifikation ausgewählter Datensätze erreicht werden konnten. Dabei wird gezeigt, daß die Leistungsfähigkeit des NEFCLASS–Modells durchaus vergleichbar ist mit der Leistungsfähigkeit anderer im Bereich der Datenklassifikation einsetzbarer Verfahren wie Neuronaler Netze oder klassischer statistischer Verfahren. Der Vorteil des NEFCLASS–Ansatzes gegenüber diesen anderen Verfahren liegt dabei in der guten Interpretierbarkeit des erzeugten NEFCLASS–Klassifikators.

Abschließend wird in den Anhängen A und B die Benutzung des Softwaretools NEFCLASS und der speziell für dieses Tool entwickelten graphischen Benutzeroberfläche NEFCLASS–X erläutert. Diese beiden Benutzerhandbücher (user manuals) sind in englischer Sprache abgefaßt.

Kapitel 2

Einführung in Neuronale Netze und Fuzzy–Systeme

2.1 Neuronale Netze

Neuronale Netze, die auch als *konnektionistische Systeme* bezeichnet werden, waren in den vergangenen Jahren Gegenstand einer großen Anzahl wissenschaftlicher Veröffentlichungen und haben mittlerweile auch in industrielle Anwendungen Einzug gehalten. Besondere Faszination geht dabei von der Lernfähigkeit Neuronaler Netze aus. Neuronale Netze sind mathematische Modelle, die versuchen, einige der im menschlichen Gehirn erkannten oder vermuteten Strukturen nachzubilden. Sie bestehen aus mehreren, sehr einfachen informationsverarbeitenden Einheiten, den *Neuronen*, die über gewichtete Verbindungen miteinander kommunizieren. Die ersten Arbeiten auf diesem Gebiet waren daher auch neurobiologisch motiviert, in der heutigen Forschung spielt die neurobiologische Plausibilität von künstlichen Neuronalen Netzen jedoch nur noch zum Teil eine Rolle.

Das erste künstliche Neuronale Netz, das berühmte *Perceptron* (von engl. perception = Wahrnehmung), wurde bereits 1958 von Frank Rosenblatt vorgestellt [Rosenblatt, 1958] und nährte bei vielen Forschern die Hoffnung, die Vision von der Konstruktion „lernfähiger Maschinen“ könnte sich in Zukunft erfüllen. Das Perceptron war ein sehr einfaches Neuronales Netz, das lediglich aus einer Verarbeitungseinheit bestand und trotzdem mit Hilfe einer Lernregel aus einer Menge von Beispielen lernte, bestimmten Eingabemustern die erwünschte Ausgabe zuzuordnen. Das Perceptron konnte somit zur Musterklassifikation eingesetzt werden. Der Lernalgorithmus des Perceptrons verändert dabei die Gewichte der Eingänge und führt garantiert zum Erfolg, was durch das sogenannte Perceptron–Konvergenztheorem nachgewiesen werden konnte. Voraussetzung ist dabei allerdings, daß das zu lösende Problem in eine bestimmte Problemklasse fällt, d.h. überhaupt mit einem Perceptron gelöst werden *kann* (dies erkannte man jedoch erst später).

1960 stellten Widrow und Hoff mit dem ADALINE (ADAptive LInear NEuron) ein erstes lineares Neuronales Netzmodell vor, das ebenfalls zur Musterklassifikation eingesetzt werden konnte. Ihren Namen erhalten lineare Neuronale Netze durch die Verwendung linearer Aktivierungsfunktionen in ihren Verarbeitungseinheiten. Das beim ADALINE eingesetzte Lernverfahren, die *Delta–Regel*, ist eine Weiterentwicklung der Perceptron–Lernregel, mit dem ein ADALINE weitaus schneller lernt als ein Perceptron.

Beiden Modellen ist gemeinsam, daß sie (streng genommen) nur aus einem Neuron mit mehreren gewichteten Eingängen und einem Ausgang bestehen. Man kann diese Architektur aber auch als zweischichtig ansehen, indem man sich die beiden Modelle bestehend aus einer mehrelementigen *Eingabeschicht* und einer einelementigen *Ausgabeschicht* vorstellt. Die Neuronen in der Eingabeschicht nehmen dabei nur die externe Eingabe entgegen und leiten diese über die gewichteten Verbindungen an das Ausgabeneuron weiter, d.h. die Neuronen der Eingabeschicht führen im eigentlichen Sinne keine Verarbeitung durch. Die Ausgabeeinheit berechnet ihre Aktivierung als gewichtete Summe der Eingänge. Die Ausgabe des Neuronalen Netzes ist dann die Aktivierung des Ausgabeneurons (ggf. noch durch eine Ausgabefunktion verändert, um z.B. als Ausgabewerte nur -1 und 1 zuzulassen).

Die Hoffnungen, die viele Forscher mit Neuronalen Netzen verbanden, wurden Ende der sechziger Jahre durch Minsky und Papert in ihrem Buch *Perceptrons* [Minsky und Papert, 1969] jäh zerstört. Minsky und Papert wiesen nach, daß ein Perceptron grundsätzlich nicht in der Lage ist, daß Paritätsproblem zu lösen, d.h. zu bestimmen, ob ein gegebenes, aus Nullen und Einsen bestehendes Muster eine gerade oder ungerade Anzahl von Einsen enthält (im zweidimensionalen Fall ist das Paritätsproblem identisch mit der XOR–Funktion). Diese Funktion gehört zur Klasse der *nicht linear separablen* Probleme, die allesamt nicht durch ein Perceptron gelöst werden können.

In dieser Veröffentlichung wurde auch aufgezeigt, daß ein Perceptron zur Lösung nicht linear separabler Probleme über mindestens eine *innere Schicht (hidden units)* zwischen der Ein– und der Ausgabeschicht verfügen muß; eine Lernregel für ein solches mehrschichtiges Neuronales Netz *(Multilayer–Perceptron)* konnten Minsky und Papert jedoch nicht angeben. Nach dieser Veröffentlichung war das Interesse an Neuronalen Netzen daher zunächst stark rückläufig.

Den Durchbruch in der Forschung an Neuronalen Netzen bewirkten dann 1986 David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams, als sie eine Verallgemeinerung der Delta–Regel von Widrow und Hoff für Multilayer–Perceptrons veröffentlichten, das *Backpropagation*–Verfahren [Rumelhart et al, 1986]. Die Kernidee dieses Lernverfahrens ist es, den Fehler, den die Einheiten in der Ausgabeschicht machen, rückwärts durch das Netz zu propagieren (d.h. ausgehend von der Ausgabeschicht durch die inneren Schichten in Richtung Eingabeschicht) und dieses Fehlersignal zur Bestimmung des Fehlers der inneren Einheiten und damit zur Bestimmung der Gewichtsänderung zu verwenden.

Multilayer–Perceptrons sind vielfältig einsetzbar z.B. in der Mustererkennung und –klassifikation. Man kann auch nachweisen, daß Multilayer–Perceptrons *universelle Approximatoren* sind, d.h. sie können jede beliebige stetige Funktion annähern. Allerdings ist der Lernerfolg bei Multilayer–Perceptrons keineswegs garantiert, ein Konvergenztheorem vergleichbar mit dem des einfachen Perceptrons gibt es nicht. Dies ist darin begründet, daß das Backpropagation–Verfahren ein heuristisches Verfahren ist, daß einen Gradientenabstieg vollzieht, der sich anschaulich folgendermaßen beschreiben läßt: Der Algorithmus versucht, daß tiefste Tal innerhalb eines Gebirges (der Fehlerfläche über den Gewichten) dadurch zu erreichen, daß er nur bergab geht. Offensichtlich muß dieses Vorgehen nicht zur Bestimmung des absoluten Minimums führen, der Algorithmus kann sich in lokalen Minima verfangen. Auch die Bestimmung der Anzahl der inneren Schichten sowie der Anzahl Neuronen innerhalb dieser Schichten ist schwierig, da hierfür kein geschlossenes Verfahren existiert. Hier müssen sich Anwender zumeist auf Probieren und „Faustregeln“ verlassen.

Die im vorangegangenen Abschnitt kurz skizzierte Entwicklung der Forschung an Neuronalen Netzen ist sehr ausführlich in dem Band *Neurocomputing* von James A. Anderson und Edward Rosenfeld [Anderson et al, 1988] beschrieben, hier finden sich auch die wichtigsten Veröffentlichungen der letzten Jahrzehnte im Original. Einen guten Überblick über die verschiedenen Netztypen und die zugehörigen Lernverfahren bietet auch das Buch von Detlef Nauck, Frank Klawonn und Rudolf Kruse [Nauck et al, 1996].

Die Lernverfahren der gängigen Neuronalen Netztypen versuchen eine Lösung des gestellten Problems durch eine geeignete Veränderung der Verbindungsgewichte zu finden, die eigentliche Netzwerkstruktur (Anzahl Schichten, Anzahl Neuronen, Existenz oder Fehlen von Verbindungen) bleibt in der Regel unberührt. Gelingt es dem Lernalgorithmus, die Gewichte der einzelnen Verbindungen so zu bestimmen, daß die Lernaufgabe erfüllt wird, so ist die Lösung also implizit in den Gewichten kodiert und damit einer sinnvollen Interpretation nicht zugänglich.

Dies ist ein großer Nachteil Neuronaler Netze: Das erlernte „Wissen“ ist verteilt in dem Neuronalen Netz gespeichert und kann nicht in expliziter Form aus dem Netz extrahiert werden. Neuronale Netze weisen also ein „Black–Box“–Verhalten auf; sie können eine bestimmte Funktion erfüllen, *wie* sie dies tun ist aber nicht zugänglich (und damit auch nicht verifizierbar). Analoge Schwierigkeiten ergeben sich bei dem Versuch, bereits bestehendes Vorwissen über die Lernaufgabe in ein Neuronales Netz zu integrieren, um die Lernaufgabe zu erleichtern. Auch dies ist aus demselben Grund nicht möglich.

Dennoch lassen sich Neuronale Netze hervorragend im Bereich der Musterklassifikation einsetzen. Ihre Stärken liegen in der Fehlertoleranz, der Fähigkeit auch für unvollständige oder gestörte Eingaben akzeptable Ausgaben produzieren zu können sowie in ihrer Eigenschaft, aus Beispielen „lernen“ zu können. Weiterhin läßt ihre verteilte Struktur hochgradig parallele Software–Implementierungen und Hardware–Realisierungen zu.

2.2 Fuzzy–Systeme

Komplexe technische Systeme werden üblicherweise durch mathematische Modelle (z.B. Differentialgleichungssysteme) beschrieben. Eine großes Problem bei der Bestimmung einer adäquaten mathematischen Modellierung einer konkreten Problemstellung besteht darin, daß oftmals erhebliche Idealisierungen des realen Systems gemacht werden müssen, weil mathematische Modelle stets nach exakten Konzepten verlangen.

Eine Möglichkeit, die Komplexität von Systemen zu reduzieren und damit ihre Handbarkeit zu verbessern, besteht in der Tolerierung eines kalkulierbaren Anteils an *Impräzision* und *Vagheit*. Bei einer solchen Modellierung wird ganz bewußt ein gewisser Informationsverlust in Kauf genommen, so daß derartige Systeme nicht perfekt im Sinne der Exaktheit sind, aber dennoch die geforderte Aufgabe bewältigen können.

In der alltäglichen Kommunikation wird *Vagheit* gezielt benutzt, um Informationen auf das Wesentliche zu beschränken und die Kommunikation so zu vereinfachen. Beschreibt z.B. eine Person die Temperatur einer Herdplatte als „heiß", so wird dies bei einer zuhörenden Person in der Regel das Fernbleiben von der Herdplatte zur Folge haben, unabhängig davon, ob die exakte Temperatur der Herdplatte 80.4^oC oder 91.7^oC beträgt. *Impräzision* dagegen entsteht dadurch, daß Informationen (z.B. Meßwerte) häufig nur bis zu einem gewissen Genauigkeitsgrad beobachtet oder gemessen werden können.

Die Grundidee von Fuzzy–Systemen besteht nun darin, klassische zweiwertige Konzepte und Eigenschaften („Prädikate") wie *jung*, *heiß*, usw. dahingehend zu erweitern, daß solche Prädikate auch *graduell* erfüllt sein können. Danach kann dann z.B. ein Mensch nicht nur als *jung* oder *nicht jung* eingestuft werden, sondern dieses Prädikat zu einem gewissen Grad zwischen 0 und 1 erfüllen.

Mitte der sechziger Jahre erweiterte L. A. Zadeh, der Begründer der Fuzzy–Theorie, das klassische mathematische Mengenkonzept in diesem Sinne, indem er für die „ist–Element–von"–Beziehung graduelle Zugehörigkeitswerte einführte [Zadeh, 1965]. Diese Überlegung führt auf das Konzept der *Fuzzy–Menge*, mit dem z.B. linguistische Terme wie *jung*, *heiß*, *schnell* usw. sehr gut modelliert werden können.

Die meisten Anwendungen von Fuzzy–Systemen finden sich im Bereich der Regelungstechnik, dem *Fuzzy Control*. Auf diesem Gebiet sind auch die bemerkenswertesten Erfolge dieser Technologie zu verzeichnen, z.B. der Einzug von Fuzzy–Methoden in Konsumgüter wie z.B. Fotoapparate.

Diese regelungstechnischen Anwendungen des Fuzzy Control basieren auf Wenn–Dann–Regeln, deren Prämissen linguistische Terme (als sprachliche Beschreibung unscharfer Eingangsgrößen bzw. impräziser Meßwerte) und deren Konklusion einen für diese Eingabe– bzw. Meßwerte geeigneten (ggf. ebenfalls unscharfen) Stellwert enthält. Die linguistischen Terme in den Regelprämissen werden durch entsprechend festgelegte Fuzzy–Mengen repräsentiert. Diese *Fuzzy–Regeln* sind dabei nicht als logische Implikation, sondern als Angabe einer Funktion durch Fallunterscheidungen aufzufassen.

In der klassischen Regelungstechnik steht das mathematisch–physikalische Modell des zu regelnden Prozesses im Vordergrund (z.B. in Form von Differentialgleichungen), was sicherlich ein sinnvoller Ansatz ist. Oftmals ist jedoch eine solche mathematische Prozeßbeschreibung nicht leicht anzugeben. Auf der anderen Seite gibt es eine Vielzahl von Prozessen, die durch Menschen geregelt werden können, ohne daß sich diese Menschen jemals Gedanken über das zugrundeliegende mathematische Modell gemacht haben (beispielsweise skilaufen, Fahrrad fahren etc.).

Die grundsätzliche Idee der Fuzzy–Regelung ist es nun, einen solchen menschlichen Bediener eines Regelungsprozesses („den Experten") zu *simulieren (kognitive Regelung)*. Das Wissen des Experten über den zu regelnden Prozeß kann entweder direkt durch eine Befragung (d.h. der Experte sieht sich in der Lage, sein Wissen in Form von linguistischen Wenn–Dann–Regeln zu formulieren) oder aber durch eine Beobachtung des Experten akquiriert werden.

Ein generelles Problem bei Fuzzy–Systemen ist die Festlegung konkreter Zugehörigkeitsgrade, durch die angegeben wird, in welchem Maße ein bestimmtes Element einer Grundmenge (z.B. ein Meßwert) ein Konzept erfüllt. Durch die Festlegung dieser Zugehörigkeitsgrade wird jedoch das Verhalten eines Fuzzy–Controllers entscheidend bestimmt, so daß nach der Inbetriebnahme eines solchen Reglers häufig ein aufwendiges „Fine–Tuning", d.h. eine nachträgliche Anpassung der Zugehörigkeitsgrade, erfolgen muß.

Das Konzept der Regelung von technischen Systemen unter Benutzung von Fuzzy–Methoden wurde zuerst von L. A. Zadeh entwickelt [Zadeh, 1972b]. Die ersten Anwendungen des Fuzzy Control wurden von E. H. Mamdani und S. Assilian [Mamdani et al, 1975, Mamdani, 1974] sowie von M. Sugeno [Sugeno, 1985] vorgestellt. Noch heute basieren die meisten Anwendungen des Fuzzy Control auf den nach diesen Pionieren benannten Mamdani– bzw. Sugeno–Controllern. Einen guten Überblick über das gesamte Gebiet der Fuzzy–Systeme (inklusive Fuzzy Control, vor allem auch zu semantischen Fragen der Theorie der Fuzzy–Mengen) bietet das Buch von R. Kruse, J. Gebhardt und F. Klawonn [Kruse et al, 1993].

2.3 Neuronale Fuzzy–Systeme

Nach den Ausführungen in den beiden vorangegangenen Abschnitten ist offensichtlich, daß eine Kopplung von Neuronalen Netzen und Fuzzy–Systemen von großem Interesse ist. Die Vorteile des einen Konzepts sind gerade die Nachteile des anderen und umgekehrt. So sind Neuronale Netze zwar in der Lage, aus gegebenen Beispielen zu lernen, indem sie generalisieren; das in einem trainierten Neuronalen Netz enthaltene Wissen kann jedoch nicht extrahiert werden („Black–Box–Verhalten“). Aus demselben Grund kann eventuell vorhandenes a–priori–Wissen nicht in das Neuronale Netz integriert werden.

Auf der anderen Seite sind Fuzzy–Systeme leicht zu interpretieren, da sie auf linguistischen Wenn–Dann–Regeln basieren. Fuzzy–Systeme können daher leicht mit Vorwissen über die Problemstellung initialisiert werden. Problematisch ist hingegen die Festlegung konkreter Zugehörigkeitsgrade für die betrachteten Elemente (z.B. Meßwerte). Bei größeren Fuzzy–Systemen ist eine Anpassung der Fuzzy–Mengen zum Zweck der Optimierung von Hand kaum zu bewältigen. Hier wäre ein Lernverfahren, daß aus Beispieldaten die Anpassung der Fuzzy–Mengen vornehmen kann, von großem Nutzen.

Solche Kopplungen von Neuronalen Netzen mit Fuzzy–Systemen, die sog. *Neuronalen Fuzzy–Systeme* (oder kurz *Neuro–Fuzzy–Systeme*), die zu interpretierbaren, lernfähigen Systemen führen, sind daher in den letzten Jahren Gegenstand intensiver Forschungsbemühungen. Unter den verschiedenen Ansätzen, Neuronale Netze mit Fuzzy–Systemen zu kombinieren, lassen sich zwei verschiedene Ansätze unterscheiden.

Beim ersten Ansatz, den *kooperativen Neuro–Fuzzy–Systemen* arbeiten ein Neuronales Netz und ein Fuzzy–System grundsätzlich unabhängig voneinander. Die Kopplung der beiden Architekturen besteht darin, daß das Neuronale Netz einige Parameter des Fuzzy–Systems im Sinne einer Optimierung erlernt.

Der zweite Ansatz basiert auf einer einheitlichen Architektur, die sich sowohl als Neuronales Netz als auch als Fuzzy–System interpretieren läßt. Eine derartige Form der Kopplung wird als *hybrides Neuro–Fuzzy–System* bezeichnet. In der Regel lehnt sich die Architektur eines solchen Systems an ein Neuronales Netz an, dessen Verbindungsgewichte jedoch nicht durch reelle Zahlenwerte, sondern durch Fuzzy–Mengen realisiert werden. Die Fuzzy–Gewichte können mit linguistischen Termen assoziiert werden, so daß ein hybrides Neuro–Fuzzy–System ähnlich wie ein herkömmliches Fuzzy–System interpretierbar ist.

Die meisten Neuro–Fuzzy–Systeme stammen aus dem Bereich der Regelungstechnik, d.h. es wird versucht, einen herkömmlichen Fuzzy–Regler in die Lage zu versetzen, seine Fuzzy–Mengen selbst zu erlernen.

Eine ausführlichere Einführung in Neuro–Fuzzy–Systeme sowie Hinweise auf weiterführende Literatur zu diesem Themengebiet findet sich in [Nauck et al, 1996]. Dort wird neben anderen Neuro–Fuzzy–Ansätzen auch das hier behandelte NEFCLASS–Modell vorgestellt und diskutiert.

Das in dieser Arbeit vorgestellte NEFCLASS–Modell ist ein hybrides Neuro–Fuzzy–System, das zur Musterklassifikation verwendet werden kann. Die formalen Grundlagen des Modells werden im folgenden Kapitel erläutert.

Kapitel 3

Das NEFCLASS–Modell

In diesem Kapitel werden die formalen Grundlagen des Neuro–Fuzzy–Modells NEFCLASS erläutert, das zur Klassifikation von Daten verwendet werden kann. Für das Verständnis dieses Kapitels werden Kenntnisse über die grundsätzliche Arbeitsweise Neuronaler Netze (insbesondere Multilayer–Perceptrons) sowie die Grundlagen der Theorie der Fuzzy–Mengen vorausgesetzt. Der interessierte Leser findet im Kapitel 2 einige Hinweise auf entsprechende Literatur. Einen guten Überblick über beide Gebiete (mit dem Schwerpunkt auf Neuronalen Netzen) findet man in [Nauck et al, 1996]. Das NEFCLASS–Modell wurde von den Autoren dieses Buches entwickelt; die Ausführungen in diesem Kapitel lehnen sich stark an die dort zu findende Diskussion des Modells an.

Nach einer kurzen Einführung in die Datenklassifikation im folgenden Abschnitt wird in Abschnitt 3.2 das formale NEFCLASS–Modell vorgestellt. In Abschnitt 3.3 wird das Propagationsverfahren dann ausführlich diskutiert und an einem Beispiel erläutert. Die beiden sich daran anschließenden Abschnitte dieses Kapitels gehen eingehend auf den Regelerzeugungsprozeß (Abschnitt 3.4) und auf das Erlernen der den Regeln zugrundeliegenden Fuzzy–Mengen (Abschnitt 3.5) ein.

3.1 Einführung

Für den Bereich der Datenklassifikation (im folgenden wird hierfür auch häufig synonym der Begriff *Musterklassifikation* verwendet) gibt es eine Vielzahl von Anwendungen in den verschiedensten Bereichen. Grundsätzlich kann man z.B. jede menschliche Entscheidung, durch die ein bestimmtes Objekt einer von mehreren möglichen (disjunkten) Klassen zugeordnet wird, als Klassifikation aufgefaßt werden. Ein Objekt wird dabei in der Regel nicht als Ganzes gesehen, sondern (im Sinne einer Konzentration auf des Wesentliche, d.h. im Sinne einer Datenreduktion) durch im Hinblick auf die Entscheidungsfindung besonders wichtige Attribute (Merkmale) charakterisiert.

Ein solches Beispiel ist der Kauf eines Gebrauchtwagens: Hier ordnet der potentielle Käufer jedes der begutachteten Autos anhand von Merkmalen wie Kaufpreis, Laufleistung, Motorleistung etc. einer der beiden Klassen „interessante Angebote“ bzw. „uninteressante Angebote“ zu. Das Beispiel des Gebrauchtwagenkaufs ist simpel; es ist jedoch unmittelbar einsichtig, daß Klassifikationen von Objekten mit sehr vielen Attributen bzw. unter Berücksichtigung vieler verschiedener Klassen oder aber Klassifikationen einer großen Anzahl von Objekten manuell kaum durchzuführen sind.

Beispiele für derartige Anforderungen finden sich in sehr vielen Bereichen u.a. in Medizin, Technik und Wirtschaft. Man denke z.B. an einen Produktionsprozeß, in dem ein Bauteil mehrere Produktionsschritte durchläuft (und dabei jeweils ein charakteristisches Merkmal, beispielsweise ein Durchmesser o.ä., bestimmt werden kann) und am Ende des Produtionsprozesses jedes Bauteil einer bestimmten Qualitätsklasse zugeordnet werden soll. Andere denkbare Anwendungen sind die automatische Bearbeitung von Kreditanträgen auf der Basis charakteristischer Merkmale des Antragstellers oder die Bestimmung der richtigen Therapie bei bestimmten Kranheitssymptomen.

Es besteht daher ein großer Bedarf an rechnergestützen Systemen zur automatischen Objektklassifikation. Solche Systeme, die unter Verwendung eines bestimmten Klassifikationsverfahrens eine Objektklassifikation selbständig bestimmen können, werden auch als *Klassifikatoren* bezeichnet.

Um einen Klassifikator für eine bestimmte Problemstellung zu bestimmen, muß in der Regel eine genügend große Menge an repräsentativen, bereits korrekt klassifizierten Beispieldaten vorliegen. Durch eine Analyse dieser Beispieldaten, deren konkrete Durchführung abhängig ist von dem zu verwendenden Klassifikationsverfahren, werden dann Struktur und / oder Parameter des Klassifikators bestimmt. Der so erstellte Klassifikator kann dann zur Klassifikation neuer Daten, deren Klassifikation unbekannt ist, herangezogen werden.

Bevor man den Klassifikator zur Klassifikation unbekannter Daten verwendet, sollte man die Leistungsfähigkeit des Klassifikationssystems anhand bereits klassifizierter Muster (jedoch nicht derselben Beispieldaten, die bereits zur Bestimmung des Klassifikators dienten) zunächst testen. Üblicherweise werden daher die verfügbaren Beispieldaten nur zur Hälfte zur Bestimmung des Klassifikators verwendet, die andere Hälfte wird zum Testen des erhaltenen Klassifikators benutzt (die korrekte Klassifikation der Beispieldaten ist schließlich bekannt). Dabei ist bei der Aufteilung der Beispieldatenmenge in Trainings– und Testdatenmenge unbedingt darauf zu achten, daß beiden Mengen tatsächlich möglichst repräsentative Muster enthalten, da dies wesentlich die Güte des Klassifikationssystems beeinflußt.

Nach dieser kurzen motivierenden Einführung in die Musterklassifikation wird in der folgenden Definition festgelegt, was in dieser Arbeit formal unter eine Klassifikationsaufgabe (im folgenden häufig auch als *Lernaufgabe* bezeichnet) zu verstehen ist.

Definition 3.1 (Klassifikationsaufgabe) *Eine Klassifikationsaufgabe* $\mathcal{L} \subseteq \mathbb{R}^n \times \{0,1\}^m$ *ist eine Menge von Ein–/Ausgabepaaren* $(\mathbf{x}, \mathbf{t})$. *Die Eingabemuster sind reelle Vektoren* $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ *, bei denen wir voraussetzen, daß sie sich in disjunkte Klassen einteilen lassen. Die einzelnen Komponenten* x_i *eines Eingabemusters* $\mathbf{x}$ *werden wir als Merkmale bezeichnet. Eine Klasse* $C_j \subseteq \mathbb{R}^n$ *ist die Menge von Eingabemustern, die die Klasse* j *repräsentieren. Das Ausgabemuster* $\mathbf{t}$ *eines Paares* $(\mathbf{x}, \mathbf{t})$ *stellt eine binäre Kodierung der Klassenzugehörigkeit von* $\mathbf{x}$ *dar, d. h. es gilt*

$$\mathbf{t} = (t_1, \ldots, t_m) \in \{0,1\}^m \text{ mit } t_j = \begin{cases} 1, & \text{falls } \mathbf{x} \in C_j \\ 0, & \text{sonst.} \end{cases}$$

Bemerkung:

(i) Die vorangegangene Definition einer Klassifikationsaufgabe erlaubt auch das Vorhandensein *widersprüchlicher* Musterpaare, in denen dasselbe Eingabemuster unterschiedlichen Klassen zugeordnet wird.

Im Gegensatz dazu gilt bei einer *eindeutigen* Klassifikationsaufgabe für alle Musterpaare $p = (\mathbf{x}^{(p)}, \mathbf{t}^{(p)})$, $q = (\mathbf{x}^{(q)}, \mathbf{t}^{(q)}) \in \mathcal{L}$:

$$(\mathbf{x}^{(p)} = \mathbf{x}^{(q)}) \Longrightarrow (\mathbf{t}^{(p)} = \mathbf{t}^{(q)}).$$

Bei der Verwendung nicht eindeutiger Klassifikationsaufgaben mit einem regelbasierten Klassifikationsansatz werden alle Musterpaare mit gleichem Eingabemuster ein und derselben Klasse zugeordnet, d.h. die Widersprüche können nicht aufgelöst werden und führen zu Fehlklassifikationen (im Sinne der Lernaufgabe).

(ii) Im Sprachgebrauch der Neuronalen Netze wird eine solche Klassifikationsaufgabe auch als *feste Lernaufgabe* bezeichnet, da die gewünschte Ausgabe zu einem bestimmten Eingabemuster $\mathbf{x}$ im Vorfeld bekannt ist. Einige neuronale Netztypen können auch *freie Lernaufgaben* erfüllen, bei denen die korrekte Ausgabe zu einem gegebenen Eingabemuster nicht bekannt ist. Ziel des Lernvorgangs ist es dann, im Sinne eines geeigneten Abstandsmaßes ähnliche Eingaben auf ähnliche Ausgaben abzubilden.

(iii) Die vorangegangene Definition legt fest, daß die Eingabemuster reelle Vektoren sind. Häufig sollen bei der Datenanalyse aber auch diskrete Merkmale (wie z.B. „ledig“, „verheiratet“, „geschieden“) oder auch (als Spezialfall der diskreten Merkmale) binäre Merkmale (z.B. „männlich“/ „weiblich“, Wahrheitswerte) verwendet werden.

Sollen Daten, die solche nicht reellwertige Merkmale beeinhalten, mit Hilfe eines NEFCLASS–System klassifiziert werden, so müssen diese Merkmale geeignet in reelle Merkmale transformiert werden.

Für die Aufgabe der Musterklassifikation im Sinne der Definition 3.1 existieren eine Reihe mächtiger Verfahren. Zum einen ist diese Form der Datenanalyse eine typische Anwendung Neuronaler Netze, zum anderen können selbstverständlich auch klassische statistische Verfahren wie die Cluster– oder Regressionsanalyse verwendet werden.

Die Benutzung eines Neuro–Fuzzy–Ansatzes wie NEFCLASS bietet jedoch zwei wesentliche Vorteile gegenüber den vorgenannten Verfahren:

- Das Klassifikationssystem ist für den Benutzer *interpretierbar* durch die Verwendung sog. *Fuzzy–Regeln*.
- Verfügt der Anwender im Vorfeld der Datenanalyse bereits über (partielles) Vorwissen über die Klassifikation, so kann er dieses Vorwissen (gerade aufgrund der Interpretierbarkeit des Systems) in NEFCLASS integrieren und damit den Lernprozeß verkürzen und / oder das Klassifikationsergebnis verbessern.

Die von NEFCLASS verwendeten Fuzzy–Regeln haben die Form

$$R_r : \textbf{If } x_1 \text{ is } A^{(1)}_{j_1,r} \text{ and } \ldots \text{ and } x_n \text{ is } A^{(n)}_{j_n,r} \textbf{ then } (x_1, x_2, \ldots, x_n) \in C_j,$$

wobei $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ gemäß Definition 3.1 ein Eingabemuster darstellt; $C_j \subseteq \mathbb{R}^n$ ist die Menge von Mustern, die zur Klasse j gehören. $A^{(1)}_{j_1,r}, \ldots, A^{(n)}_{j_n,r}$ sind linguistische Terme (wie *jung*/*alt*, *kalt*/*warm*), die durch Fuzzy–Mengen $\mu^{(1)}_{j_1,r}, \ldots, \mu^{(n)}_{j_n,r}$ repräsentiert werden. Diese Fuzzy–Mengen werden durch eine Partitionierung der Wertebereiche der einzelnen Merkmale x_i erzeugt, d.h. der Wertebereich jedes Merkmals x_i wird durch q_i Fuzzy–Mengen $\mu^{(i)}_1, \ldots, \mu^{(i)}_{q_i}$ partitioniert. Die Regelbasis eines NEFCLASS–Systems wird durch k solcher Fuzzy–Regeln $R_1, \ldots, R_k$ beschrieben.

NEFCLASS ist in der Lage, seine die Regelbasis bildenden Fuzzy–Regeln durch Analyse der Musterdaten vollständig selbst zu erzeugen, es können jedoch auch Regeln vom Benutzer angegeben werden. In einem zweiten Schritt versucht NEFCLASS dann, die Parameter der den linguistischen Termen zugeordneten Fuzzy–Mengen zu optimieren.

3.2 Das formale Modell

Das NEFCLASS–Modell basiert auf dem in [Nauck et al, 1996] vorgestellten dreischichtigen *Fuzzy–Perceptron*, das ebenso die Grundlage für das verwandte Neuro–Fuzzy–Modell NEFCON (NEural Fuzzy CONtroller) [Nauck et al, 1996] darstellt, welches im Bereich der Fuzzy–Regelung eingesetzt werden kann.

Bevor das formale NEFCLASS–Modell eingeführt werden kann, wird daher in der folgenden Definition zunächst die Struktur und Arbeitsweise eines Fuzzy–Perceptrons definiert.

Definition 3.2 (Dreischichtiges Fuzzy–Perceptron) *Ein dreischichtiges Fuzzy–Perceptron ist ein vorwärtsbetriebenes Neuronales Netz (U, W, NET, A, O, ex) mit den folgenden Spezifikationen:*

(i) $U = \bigcup_{i \in M} U_i$ *ist eine nicht-leere Menge von Verarbeitungseinheiten (Neuronen) und* $M = \{1, 2, 3\}$ *heißt Indexmenge von* U*. Für alle* $i, j \in M$ *gilt* $U_i \neq \emptyset$ *und* $U_i \cap U_j = \emptyset$ *für* $i \neq j$*.* U_1 *heißt Eingabeschicht,* U_2 *Regelschicht (hidden layer) und* U_3 *Ausgabeschicht.*

(ii) Die Struktur des Fuzzy–Perceptrons ist gegeben durch $W : U \times U \to \mathcal{F}(\mathbb{R})$*, wobei ausschließlich Verbindungen von der Eingabeschicht zur Regelschicht und von der Regelschicht zur Ausgabeschicht existieren.*

(iii) A *ordnet jeder Einheit* $u \in U$ *eine Aktivierungsfunktion* A_u *zu, mit der die Aktivierung* a_u *einer Einheit berechnet wird. Dabei gilt*

(a) für Eingabe- und Regeleinheiten $u \in U_1 \cup U_2$*:*

$$A_u : \mathbb{R} \to \mathbb{R}, \quad a_u = A_u(\mathrm{net}_u) = \mathrm{net}_u,$$

(b) für Ausgabeeinheiten $u \in U_3$*:*

$$A_u : \mathcal{F}(\mathbb{R}) \to \mathcal{F}(\mathbb{R}), \quad a_u = A_u(\mathrm{net}_u) = \mathrm{net}_u.$$

(iv) O *ordnet jeder Einheit* $u \in U$ *eine Ausgabefunktion* O_u *zur Berechnung der Ausgabe* o_u *zu. Es gilt*

(a) für Eingabe- und Regeleinheiten $u \in U_1 \cup U_2$*:*

$$O_u : \mathbb{R} \to \mathbb{R}, \quad o_u = O_u(a_u) = a_u,$$

(b) für Ausgabeeinheiten $u \in U_3$*:*

$$O_u : \mathcal{F}(\mathbb{R}) \to (\mathbb{R}), \quad o_u = O_u(\mathrm{net}_u) = DEFUZZ_u(\mathrm{net}_u),$$

wobei $DEFUZZ_u$ *ein geeignetes Defuzzifizierungsverfahren ist.*

(v) NET *ordnet jeder Einheit* $u \in U$ *eine Netzeingabefunktion* NET_u *zur Berechnung der Netzeingabe* net_u *zu. Es gilt*

(a) für Eingabeeinheiten $u \in U_1$*:*

$$\mathrm{NET}_u : \mathbb{R} \to \mathbb{R}, \quad \mathrm{net}_u = \mathrm{ex}_u,$$

(b) *für Regeleinheiten $u \in U_2$:*

$$\begin{aligned}\mathrm{NET}_u &: (\mathbb{R} \times \mathcal{F}(\mathbb{R}))^{U_1} \to [0,1],\\ \mathrm{net}_u &= \top_{u' \in U_2}(W(u',u)(o_{u'})),\end{aligned}$$

wobei $\top$ eine t-Norm ist.

(c) *für Ausgabeeinheiten $u \in U_3$:*

$$\begin{aligned}\mathrm{NET}_u &: ([0,1] \times \mathcal{F}(\mathbb{R}))^{U_2} \to \mathcal{F}(\mathbb{R}),\\ \mathrm{net}_u &: \mathbb{R} \to [0,1], \quad \mathrm{net}_u(x) = \perp_{u' \in U_2} \top(o_{u'}, W(u',u)(x))),\end{aligned}$$

wobei $\perp$ eine t-Conorm ist.
Sind die Fuzzy–Mengen $W(u',u)$, $u' \in U_2$, $u \in U_3$, über ihren Trägermengen monoton, und gilt $W^{-1}(u',u)(\tau) = x \in \mathbb{R}$ mit $W(u',u)(x) = \tau$, so kann die Netzeingabefunktion für eine Ausgabeeinheit $u \in U_3$ alternativ definiert werden als:

$$\mathrm{net}_u(x) = \begin{cases} 1, & \text{falls } x = \dfrac{\sum\limits_{u' \in U_2} o_{u'} \cdot W^{-1}(u',u)(o_{u'})}{\sum\limits_{u' \in U_2} o_{u'}} \\ 0, & \text{sonst.} \end{cases}$$

Für die Ausgabe o_u gilt dann abweichend von (iv.b)

$$o_u = x, \ \mathit{mit}\ \mathrm{net}_u(x) = 1.$$

(vi) ex: $U_1 \to \mathbb{R}$, *ordnet jeder Eingabeeinheit ihre externe Eingabe* $\mathrm{ex}(u) = \mathrm{ex}_u$ *zu.* ex *ist ausschließlich für Eingabeeinheiten definiert.*

Ein Fuzzy–Perceptron, wie es in Definition 3.2 beschrieben wird, kann als eingeschränkte „Fuzzifizierung“ eines herkömmlichen Multilayer–Perceptrons interpretiert werden. Lediglich die Gewichte der Verbindungen sowie die Netzeingabe und die Aktivierung der Ausgabeeinheiten werden mit Hilfe von Fuzzy–Mengen modelliert.

Ein Fuzzy–Perceptron kann daher ebenso wie ein normales Multilayer–Perceptron zur Approximation beliebiger stetiger Funktionen dienen. Der Vorteil des Fuzzy–Perceptrons gegenüber einem herkömmlichen Multilayer–Perceptron besteht darin, daß die Netzstruktur in Form linguistischer Regeln interpretiert werden kann, indem die Fuzzy–Gewichte mit linguistischen Termen assoziiert werden.

Ein NEFCLASS–System soll zur Klassifikation von Mustern eingesetzt werden und muß somit die Klasse eines gegebenen Eingabemusters bestimmen können. Dies kann man auch im Sinne einer Funktionsinterpolation auffassen, d.h. die Regelbasis eines NEFCLASS–Systems dient zur Approximation einer (unbekannten) Funktion φ, die

die Klassifikationsaufgabe repräsentiert und ein Eingabemuster $\mathbf{x}$ seiner Klasse C_i zuordnet:

$$\varphi : \mathbb{R}^n \to \{0,1\}^m, \quad \varphi(\mathbf{x}) = (c_1, \ldots, c_m), \text{ mit } c_i = \begin{cases} 1, & \text{falls } \mathbf{x} \in C_i \\ 0, & \text{sonst.} \end{cases}$$

Aufgrund des Propagationsverfahrens von NEFCLASS kann die Regelbasis — wie wir noch in Abschnitt 3.3 sehen werden — nicht φ direkt, sondern nur die Funktion $\tilde{\varphi} : \mathbb{R}^n \to [0,1]^m$ approximieren. Wir erhalten $\varphi(\mathbf{x})$ durch $\varphi(\mathbf{x}) = \psi(\tilde{\varphi}(\mathbf{x}))$, wobei ψ die Interpretation des Klassifikationsergebnisses darstellt. Wir verwenden für NEFCLASS eine Winner–Take–All–Interpretation und bilden die größte Komponente des Ausgabevektors $\mathbf{c}$ auf 1 und alle anderen Komponenten von $\mathbf{c}$ auf 0 ab.

$$\psi : \mathbb{R}^m \to \{0,1\}^m, \ \psi(\mathbf{c}) = \mathbf{o} = (o_1, \ldots, o_m)$$

$$\text{mit} \quad o_i = \begin{cases} 1, & \text{falls } c_i = \max\limits_{i=1,\ldots,n} \{c_i\} \\ 0, & \text{sonst.} \end{cases} \tag{3.1}$$

In Abbildung 3.1 ist ein NEFCLASS–System zu sehen, das Eingabemuster mit je zwei Merkmalen in zwei disjunkte Klassen einteilt und dazu fünf linguistische Regeln verwendet. Die folgende Definition beschreibt ein NEFCLASS–System formal.

Definition 3.3 (NEFCLASS–System) *Ein NEFCLASS–System ist ein dreischichtiges Fuzzy–Perceptron gemäß Definition 3.2, das den folgenden Bedingungen genügt:*

(i) $U_1 = \{x_1, \ldots, x_n\}$, $U_2 = \{R_1, \ldots, R_k\}$, *und* $U_3 = \{c_1, \ldots, c_m\}$.

(ii) Jede Verbindung zwischen $x_i \in U_1$ *und* $R_r \in U_2$ *ist mit einem linguistischen Term* $A_{j_r}^{(i)}$ $(j_r \in \{1, \ldots, q_i\})$ *benannt.* q_i *bezeichnet dabei wie bisher die Anzahl der Fuzzy–Mengen, durch die der Wertebereich des Merkmals* x_i *partitioniert ist.*

(iii) Verbindungen, die von derselben Eingabeeinheit x_i *wegführen und identische Benennungen aufweisen, tragen zu jeder Zeit dasselbe Gewicht. Die Verbindungen heißen* gekoppelt.

(iv) Sei $L_{x,R}$ *die Benennung der Verbindung zwischen den Einheiten* $x \in U_1$ *und* $R \in U_2$. *Für alle* $R, R' \in U_2$ *gilt*

$$((\forall x \in U_1) \ L_{x,R} = L_{x,R'}) \Longrightarrow R = R'.$$

(v) Für alle Verbindungen zwischen Einheiten $R \in U_2$ *und Einheiten* $c \in U_3$ *gilt* $W(R,c) \in \{0,1\}$, *d.h. die Gewichte zwischen der Regel– und der Ausgabeschicht haben eine reine Verbindungsfunktion.*

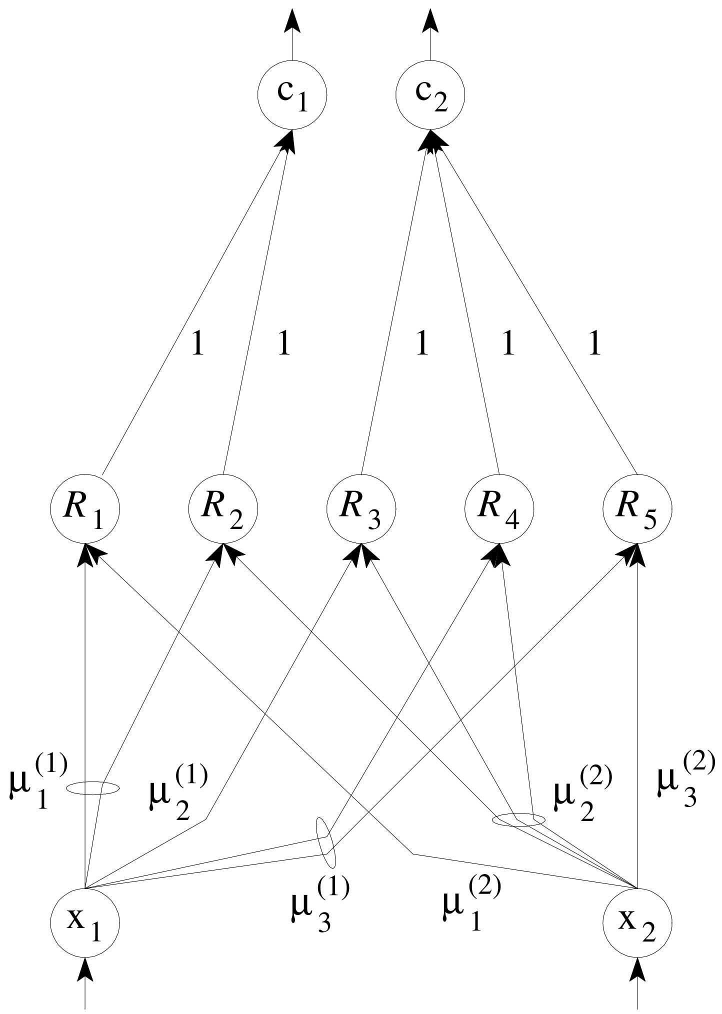

Abbildung 3.1: Ein NEFCLASS–System mit zwei Eingangsgrößen, fünf Regeln und zwei Ausgabeklassen

(vi) Für alle Regeleinheiten $R \in U_2$ und alle Ausgabeeinheiten $c, c' \in U_3$ gilt

$$(W(R,c) = 1) \wedge (W(R,c') = 1) \Longrightarrow c = c'.$$

Durch diese Bedingung ist sichergestellt, daß jede Regeleinheit mit nur einer Ausgabeeinheit verbunden sein darf. Darüber hinaus gilt auch

$$\forall_{R \in U_2} \exists_{c \in U_3} : W(R,c) = 1,$$

d.h. es existiert keine Regel, die mit keiner Ausgabeeinheit verbunden ist. Zusammengefaßt ist also jede Regel $R_1, \ldots, R_k$ mit genau *einer Ausgabeeinheit $c \in U_3$ verbunden.*

(vii) *Für alle Ausgabeeinheiten gilt* $c \in U_3$ *und* $o_c = a_c = \mathrm{net}_c$.

(viii) *Für alle Ausgabeeinheiten* $c \in U_3$ *ergibt sich die Netzeingabe* net_c *zu*

$$\mathrm{net}_c = \frac{\sum\limits_{R \in U_2} W(R,c) \cdot o_R}{\sum\limits_{R \in U_2} W(R,c)} \quad \textit{(gewichtete Summe)} \tag{3.2}$$

Alternativ kann die Netzeingabe der Ausgabeeinheiten $c \in U_3$ *auch bestimmt werden durch:*

$$\mathrm{net}_c = \max_{R \in U_2} \{W(R,c) \cdot o_R\} \quad \textit{(Maximum)} \tag{3.3}$$

Die Bedingung (iii) der Definition 3.3 stellt sicher, daß gleiche linguistische Werte einer Eingabevariablen durch dieselbe Fuzzy–Menge repräsentiert werden. Die Bedingung (iv) sorgt dafür, daß es keine zwei Regeln mit identischer Prämisse innerhalb eines NEFCLASS–Systems geben darf.

3.3 Das Propagationsverfahren

In diesem Abschnitt wird das im NEFCLASS–Modell verwendete Propagationsverfahren, das formal durch die Definitionen 3.2 und 3.3 festgelegt ist, noch einmal detailliert dargestellt und an einem Beispiel erläutert.

Zur Auswertung jeder Regel R_r der Regelbasis wird zunächst der Erfüllungs- oder Akzeptanzgrad bestimmt, zu dem die Prämisse bei dem betrachteten Eingabemuster $\mathbf{x} = (x_1, \ldots, x_n)$ erfüllt ist. Zu diesem Zweck wird für $\nu = 1, \ldots, n$ der Wert $\mu^{(\nu)}_{i_\nu,r}(x_\nu)$ berechnet, der den Zugehörigkeitsgrad des Merkmals x_ν zu der mit dem linguistischen Term $A^{(\nu)}_{i_\nu,r}$ assoziierten Fuzzy–Menge $\mu^{(\nu)}_{i_\nu,r}$ angibt. Da in der Prämisse der Regel R_r verlangt ist, daß jedes Merkmal $x_1, \ldots, x_n$ den zugehörigen linguistischen Term $A^{(1)}_{i_1,r}, \ldots, A^{(n)}_{i_n,r}$ erfüllt, müssen die Werte $\mu^{(\nu)}_{i_\nu,r}(x_\nu)\,(\nu = 1, \ldots, n)$ in geeigneter Weise konjunktiv verknüpft werden. In NEFCLASS findet an dieser Stelle die min-Funktion Verwendung, grundsätzlich kann hier aber auch jede andere t-Norm zur Auswertung der Konjunktion benutzt werden.

Bezeichnen wir den Erfüllungsgrad der Prämisse einer Regel R_r mit α_r, so berechnet sich dieser Erfüllungsgrad zusammenfassend zu

$$\alpha_r = \min\{\mu^{(1)}_{i_1,r}(x_1), \ldots, \mu^{(n)}_{i_n,r}(x_n)\} \quad (r = 1, \ldots, k). \tag{3.4}$$

In einem NEFCLASS–System wird dieser Erfüllungsgrad einer Regel durch die Aktivierung der zugehörigen Regeleinheit repräsentiert, d.h. für alle $R_r \in U_2$ gilt

$$\alpha_r = a_{R_r}, \quad r = 1, \ldots, k.$$

Nachdem die α_r–Werte in der eben beschriebenen Weise bestimmt wurden, können die Aktivierungen der Ausgabeeinheiten $c \in U_3$ nach Definition 3.3 auf zwei verschiedene Arten berechnet werden (siehe Gleichung 3.2 und 3.3):

$$a_c = \text{net}_c = \frac{\sum\limits_{R \in U_2} W(R,c) \cdot \alpha_R}{\sum\limits_{R \in U_2} W(R,c)} \quad \text{(gewichtete Summe)}$$

bzw.

$$a_c = \text{net}_c = \max_{R \in U_2} W(R,c) \cdot \alpha_R \quad \text{(Maximum)}$$

Der Anwender eines NEFCLASS–Systems kann zwischen diesen beiden alternativen Formeln zur Bestimmung der Aktivierungen in der Ausgabeschicht wählen (z.B. je nach Beschaffenheit des Datensatzes).

Um den eben beschriebenen Propagationsvorgang einmal exemplarisch darzustellen und die formalen Definitionen 3.2 und 3.3 zu veranschaulichen, folgt nun ein ausführliches Beispiel einer NEFCLASS–Anwendung.

Beispiel 3.4 Ein großer Konzern plant, das Auswahlverfahren bei der Neueinstellung von Hochschulabsolventen effizienter zu gestalten, da sich das herkömmliche Verfahren, d.h. das manuelle Prüfen der schriftlichen Bewerbungsunterlagen durch einen Sachbearbeiter bzw. eine Sachbearbeiterin, als zu zeitaufwendig erwiesen hat. Aus diesem Grund soll zum Zweck der Vorsortierung der Bewerber ein NEFCLASS–System angewendet werden.

Für die Klassifikation sind drei Eingabemerkmale vorgesehen, d.h. jede Bewerbung wird zu einem Vektor $\mathbf{x} = (x_1, x_2, x_3)$ komprimiert, wobei die einzelnen Komponenten jedes Vektors $\mathbf{x}$ folgende Bedeutung haben:

x_1 : Diplomnote (als arithmetisches Mittel aller Diplomprüfungen und der Diplomarbeit, auf zwei Nachkommastellen gerundet)
x_2 : Studiendauer (Anzahl Fachsemester)
x_3 : Alter des Bewerbers / der Bewerberin

Die Bewerber sollen durch eine geeignete Regelbasis den folgenden drei (disjunkten) Klassen zugeordnet werden:

C_1 : Bewerber/-in wird zu einem Vorstellungsgespräch eingeladen
C_2 : Bewerbung muß manuell geprüft werden
C_3 : Bewerber/-in wird eine Absage geschickt

Die gewählte Partitionierung der drei Eingabemerkmale *Diplomnote*, *Studiendauer* und *Bewerberalter* zeigt die Abbildung 3.2.

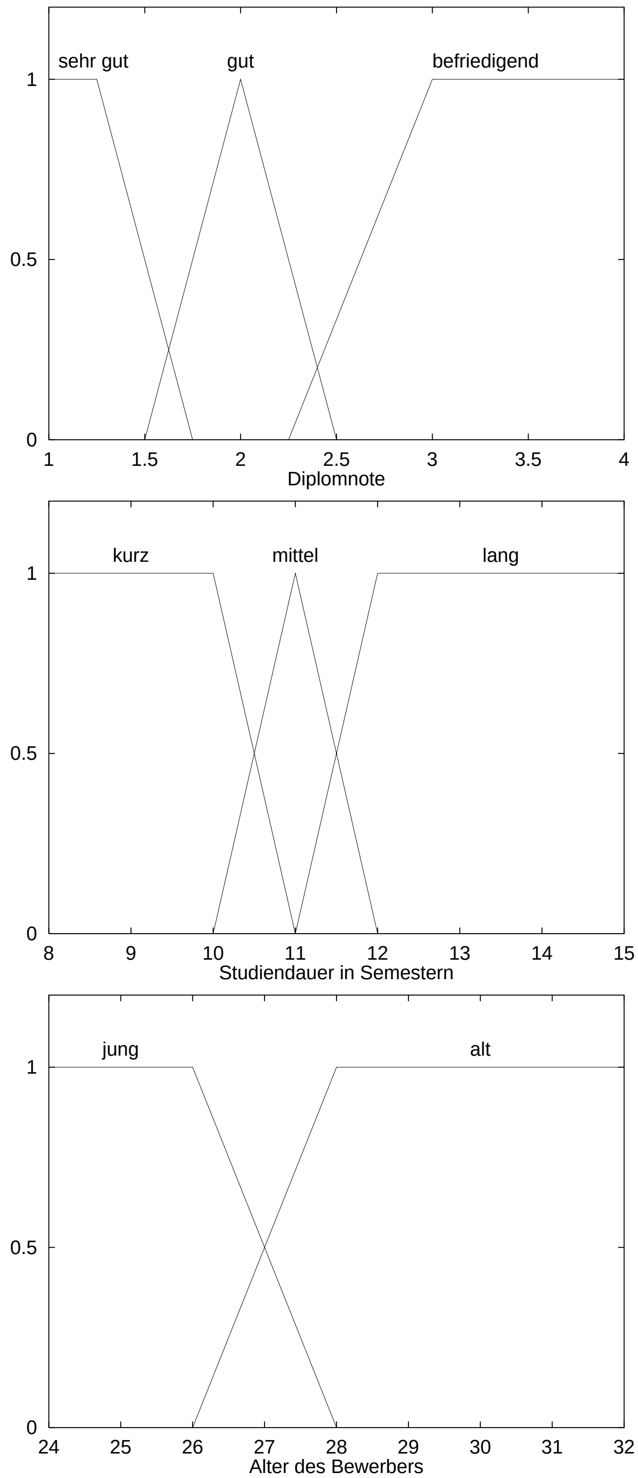

Abbildung 3.2: Partitionierung der Eingangsgrößen *Diplomnote*, *Studiendauer* und *Alter des Bewerbers*.

Der Wertebereich des Merkmals $x_1 = \mathit{Diplomnote}$ wird dabei in $q_1 = 3$ Fuzzy–Mengen $\mu_1^{(1)}, \ldots, \mu_3^{(1)}$ partitioniert, welche die linguistischen Terme $A_1^{(1)} = \mathit{sehr\_gut}$, $A_2^{(1)} = \mathit{gut}$ und $A_3^{(1)} = \mathit{befriedigend}$ repräsentieren.

Analog sind die Wertebereiche der Merkmale $x_2 = \mathit{Studiendauer}$ und $x_3 = \mathit{Bewerberalter}$ partitioniert; hier gilt $q_2 = 3$ mit den linguistischen Termen $A_1^{(2)} = \mathit{kurz}$, $A_2^{(2)} = \mathit{mittel}$ und $A_3^{(2)} = \mathit{lang}$ bzw. $q_3 = 2$ mit $A_1^{(3)} = \mathit{jung}$, $A_2^{(3)} = \mathit{alt}$.

Die von der Personalabteilung des Konzerns erstellte Regelbasis umfaßt die folgenden neun Fuzzy–Regeln:

R_1: **if** *(sehr_gut, kurz, *)*	**then** *Einladung*
R_2: **if** *(sehr_gut, mittel, jung)*	**then** *Einladung*
R_3: **if** *(sehr_gut, mittel, alt)*	**then** *Prüfen*
R_4: **if** *(sehr_gut, lang, *)*	**then** *Prüfen*
R_5: **if** *(gut, kurz, *)*	**then** *Einladung*
R_6: **if** *(gut, mittel, jung)*	**then** *Prüfen*
R_7: **if** *(gut, mittel, alt)*	**then** *Absage*
R_8: **if** *(gut, lang, *)*	**then** *Absage*
R_9: **if** *(befriedigend, *, *)*	**then** *Absage*

Tabelle 3.1: Die Regelbasis zur Klassifikation von Bewerberdaten

Aus Gründen der Übersichtlichkeit wurden die Prämissen in einer verkürzten Schreibweise notiert. Die Regel R_6

R_6: **if** *(gut, mittel, jung)* **then** *Prüfen*

ist z.B. zu lesen als

if	*Note*	is *gut* and
	Studiendauer	is *mittel* and
	Alter	is *jung*
then	*Prüfen*	

Enthält eine Regelprämisse für ein (oder mehrere) Merkmale statt eines linguistischen Terms einen „*“, so kann bei der Auswertung dieser Regel das mit „*“ bezeichnete Merkmal unberücksichtigt bleiben *(don't care)*. Dies ist sinnvoll, wenn bei einer bestimmten Merkmalskombination andere Merkmale gleichgültig werden. Davon wird z.B. in Regel R_1 Gebrauch gemacht, d.h. Bewerber, die nach einer kurzen Studienzeit einen sehr guten Abschluß erreicht haben, werden unabhängig von ihrem Alter zu einem Vorstellungsgespräch eingeladen.

Durch die Verwendung von *don't cares* in den Prämissen einiger Regeln kann die Anzahl der Regeln in der Regelbasis gegebenenfalls deutlich verringert werden, allerdings müssen solche Regeln durch den Benutzer vorgegeben werden, da NEFCLASS solche Regeln nicht selbständig erzeugen kann (siehe hierzu auch Abschnitt 3.5).

Nachdem nun sowohl die (initialen) Fuzzy–Partitionen als auch eine Regelbasis festgelegt worden sind, kann der Propagationsvorgang einmal exemplarisch durchgeführt werden.

Zu diesem Zweck soll die Bewerbung eines (fiktiven) Hochschulabgängers klassifiziert werden. Wir nehmen an, daß der Bewerber $26\frac{1}{2}$ Jahre alt ist und sein Studium nach 11 Semestern mit einer Diplomnote von 1.35 abgeschlossen hat, d.h.

$$x_0 = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)}) = (1.35, 11.0, 26.5).$$

Zunächst muß für jede der neun Regeln $R_1, \ldots, R_9$ der Regelbasis der Erfüllungsgrad der Prämisse nach Gleichung 3.4 bestimmt werden. Hierzu werden zunächst die Zugehörigkeitsgrade der Merkmalsausprägungen $x_i^{(0)}, i = 1, 2, 3$ bzgl. der Fuzzy–Mengen, die mit den in den Regelprämissen enthaltenen linguistischen Termen assoziiert werden, bestimmt. Dabei ergeben sich für unser Beispiel folgende Zugehörigkeitsgrade (die Parameter a, b, c der einzelnen Fuzzy–Mengen können der Abbildung 3.2 entnommen werden):

$$\begin{array}{lllllllll}
\mu_1^{(1)}(1.35) & = & 0.8, & \mu_2^{(1)}(1.35) & = & 0.0, & \mu_2^{(1)}(1.35) & = & 0.0, \\
\mu_1^{(2)}(11.0) & = & 0.0, & \mu_2^{(2)}(11.0) & = & 1.0, & \mu_3^{(2)}(11.0) & = & 0.0, \\
\mu_1^{(3)}(26.5) & = & 0.75, & \mu_2^{(3)}(26.5) & = & 0.25. & & &
\end{array}$$

Für die Regel R_2

R_2: **if** *(sehr_gut, mittel, jung)* **then** *Einladung*

ergibt sich damit z.B.:

$$\begin{array}{lll}
\alpha_2 & = & \min\{\mu_1^{(1)}(x_1^{(0)}), \mu_2^{(2)}(x_2^{(0)}), \mu_1^{(3)}(x_3^{(0)})\} \\
& = & \min\{\mu_1^{(1)}(1.35), \mu_2^{(2)}(11.0), \mu_1^{(3)}(26.5)\} \\
& = & \min\{0.8, 1.0, 0.75\} \\
& = & 0.75.
\end{array}$$

Die Erfüllungsgrade der anderen Regeln ergeben sich analog, wobei für die in einigen Regeln enthaltenen *don't cares* ein Zugehörigkeitsgrad von 1.0 eingesetzt wurde, so daß die betreffenden Merkmale auf die Minimumbildung in Gleichung 3.4 keinen Einfluß nehmen (also unberücksichtigt bleiben).

Damit ergeben sich die folgenden Erfüllungsgrade für die Regeln $R_1, \ldots, R_9$:

$$
\begin{aligned}
\alpha_1 &= \min\{\mu_1^{(1)}(1.35), \mu_1^{(2)}(11.0), 1.0\} &&= \min\{0.8, 0.0, 1.0\} &&= 0.0\\
\alpha_2 &= \min\{\mu_1^{(1)}(1.35), \mu_2^{(2)}(11.0), \mu_1^{(3)}(26.5)\} &&= \min\{0.8, 1.0, 0.75\} &&= 0.75\\
\alpha_3 &= \min\{\mu_1^{(1)}(1.35), \mu_2^{(2)}(11.0), \mu_2^{(3)}(26.5)\} &&= \min\{0.8, 1.0, 0.25\} &&= 0.25\\
\alpha_4 &= \min\{\mu_1^{(1)}(1.35), \mu_3^{(2)}(11.0), 1.0\} &&= \min\{0.8, 0.0, 1.0\} &&= 0.0\\
\alpha_5 &= \min\{\mu_2^{(1)}(1.35), \mu_1^{(2)}(11.0), 1.0\} &&= \min\{0.0, 0.0, 1.0\} &&= 0.0\\
\alpha_6 &= \min\{\mu_2^{(1)}(1.35), \mu_2^{(2)}(11.0), \mu_1^{(3)}(26.5)\} &&= \min\{0.0, 1.0, 0.75\} &&= 0.0\\
\alpha_7 &= \min\{\mu_2^{(1)}(1.35), \mu_2^{(2)}(11.0), \mu_2^{(3)}(26.5)\} &&= \min\{0.0, 1.0, 0.25\} &&= 0.0\\
\alpha_8 &= \min\{\mu_2^{(1)}(1.35), \mu_3^{(2)}(11.0), 1.0\} &&= \min\{0.0, 0.0, 1.0\} &&= 0.0\\
\alpha_9 &= \min\{\mu_3^{(1)}(1.35), 1.0, 1.0\} &&= \min\{0.0, 1.0, 1.0\} &&= 0.0
\end{aligned}
$$

Nach Definition 3.3 ergeben sich die Aktivierungen der Ausgabeeinheiten aus den Aktivierungen der Regeleinheiten wie folgt:

(i) Unter Anwendung der gewichteten Summe (Gleichung 3.2):

$$
\begin{aligned}
a_{c_1}^{(sum)} &= \frac{\sum\limits_{R \in U_2} W(R, c_1) \cdot o_R}{\sum\limits_{R \in U_2} W(R, c_1)} = \frac{o_{R_1} + o_{R_2} + o_{R_5}}{3} = \frac{0 + 0.75 + 0}{3} = 0.25\\
a_{c_2}^{(sum)} &= \frac{\sum\limits_{R \in U_2} W(R, c_2) \cdot o_R}{\sum\limits_{R \in U_2} W(R, c_2)} = \frac{o_{R_3} + o_{R_4} + o_{R_6}}{3} = \frac{0.25 + 0 + 0}{3} = 0.083\\
a_{c_3}^{(sum)} &= \frac{\sum\limits_{R \in U_2} W(R, c_3) \cdot o_R}{\sum\limits_{R \in U_2} W(R, c_3)} = \frac{o_{R_7} + o_{R_8} + o_{R_9}}{3} = \frac{0 + 0 + 0}{3} = 0
\end{aligned}
$$

(ii) Bei Verwendung der Maximumfunktion (Gleichung 3.3):

$$
\begin{aligned}
a_{c_1}^{(max)} &= \max_{R \in U_2}\{W(R, c_1) \cdot o_R\} = \max\{o_{R_1}, o_{R_2}, o_{R_5}\} = \max\{0, 0.75, 0\} = 0.75\\
a_{c_2}^{(max)} &= \max_{R \in U_2}\{W(R, c_2) \cdot o_R\} = \max\{o_{R_3}, o_{R_4}, o_{R_6}\} = \max\{0.25, 0, 0\} = 0.25\\
a_{c_3}^{(max)} &= \max_{R \in U_2}\{W(R, c_3) \cdot o_R\} = \max\{o_{R_7}, o_{R_8}, o_{R_9}\} = \max\{0, 0, 0\} = 0
\end{aligned}
$$

Der Ausgabevektor des NEFCLASS–Systems ist damit also — je nach Wahl der Berechnungsvorschrift für die Aktivierung der Ausgabeeinheiten — gegeben durch

$$\mathbf{c}^{(sum)} = (o_{c_1}^{(sum)}, o_{c_2}^{(sum)}, o_{c_3}^{(sum)}) = (0.25, 0.083, 0) \quad \text{bzw.}$$

$$\mathbf{c}^{(max)} = (o_{c_1}^{(max)}, o_{c_2}^{(max)}, o_{c_3}^{(max)}) = (0.75, 0.25, 0).$$

Die Anwendung der Winner–Take–All–Interpretation ψ auf $\mathbf{c}$ (siehe Gleichung 3.1) führt jedoch in beiden Fällen zu einer Zuordnung des Eingabemusters $x^{(0)}$ in die Klasse C_1, d.h. der Bewerber wird zu einem Vorstellungsgespräch eingeladen.

3.4 Der Regelerzeugungsprozeß

Im vorangegangenen Abschnitt wurde der Propagationsvorgang des NEFCLASS–Modells ausführlich dargestellt und an einem Beispiel exemplarisch durchgeführt. Dabei wurde eine vollständige Regelbasis als vorgegeben angenommen. Normalerweise verfügt man im Vorfeld einer Datenanalyse jedoch bestenfalls (wenn überhaupt) über partielles Vorwissen über die zu klassifizierenden Eingabemuster. Ein NEFCLASS–System kann einerseits aus solchem partiellem Wissen aufgebaut und dann durch Lernen verbessert werden. Ebenso vermag ein NEFCLASS–System jedoch seine Regelbasis auch vollständig selbst zu erlernen („learning from scratch").

Zur Initialisierung braucht ein NEFCLASS–System lediglich eine initiale Fuzzy–Partitionierung für jede Eingabegröße (das Softwaretool NEFCLASS erzeugt auch diese selbst) und die Angabe der maximalen Anzahl von Regeleinheiten in der inneren Schicht. Mit diesen Vorgaben kann ein NEFCLASS–System seine Regelbasis inkrementell erlernen, da es eine feste Lernaufgabe verwendet.

Zum Zweck der Regelerzeugung werden alle Muster nacheinander propagiert. Wird ein Muster dabei nicht oder falsch klassifiziert, so läßt sich aus dem Eingabemuster und seiner in der Klassifikationsaufgabe festgelegten Klassenzugehörigkeit direkt eine Fuzzy–Regel ableiten und in das NEFCLASS–Netzwerk einfügen. Um eine neue Fuzzy–Regel zu bilden, wird für jedes Merkmal eines gegebenen Eingabemusters $\mathbf{x}$ die Fuzzy–Menge $\mu_{max}^{(i)}$ bestimmt, die für den aktuellen Wert x_i des Merkmals den größten Zugehörigkeitswert ergibt.

$$\mu_{max}^{(i)} = \mu_j^{(i)} \quad \text{mit } \mu_j^{(i)}(x_i) = \max_{j \in \{1,\ldots,q_i\}} \{\mu_j^{(i)}(x_i)\}, \quad (i = 1, \ldots, n).$$

Die Konjunktion der mit diesen Fuzzy–Mengen $\mu_{max}^{(i)}$ $(i = 1, \ldots, n)$ assoziierten linguistischen Terme bildet die Prämisse der gesuchten Fuzzy–Regel; die Konklusion wird durch die in der Lernaufgabe angegebene Klasse des Musters festgelegt. Existiert noch keine Regel mit dieser Prämisse in der Regelbasis, so wird die Regelbasis um die neue Regel erweitert, sofern nicht schon die vom Benutzer spezifizierte Maximalanzahl an Regeln erreicht ist.

Der Algorithmus zur Bildung der Regelbasis eines NEFCLASS–Systems existiert in den drei Varianten „einfaches", „optimales" und „klassenoptimales" Regellernen. Diese drei Varianten werden formal in der Definition 3.5 erläutert. Die eben gemachten Ausführungen gelten für das „einfache" Regellernen.

Mit der beschriebenen Vorgehensweise ist es möglich, durch einen einzigen Durchlauf durch die Lernaufgabe eine initiale Regelbasis aufzubauen. In dieser Form hängt der Regelerzeugungsprozeß jedoch von der Reihenfolge ab, in der die zu propagierenden Muster dem NEFCLASS–System präsentiert werden. Eine auf diese Weise bestimmte Regelbasis wird nur in seltenen, einfachen Fällen zufriedenstellende Ergebnisse liefern. Dazu müßten die Muster bezüglich ihrer Klassen ungefähr gleichverteilt sein und idealerweise klassenweise abwechselnd propagiert werden.

Sind diese Bedingungen nicht gegeben, können vor allem die folgenden Probleme auftreten:

- Die maximale Regelanzahl wird bereits erreicht, bevor alle Klassen propagiert sind. Dies hat zur Folge, daß eine oder mehrere Klassen überhaupt nicht erkannt werden.
- Ein „Ausreißer", d.h. ein Muster, das durch die Klassifikationsaufgabe einer Klasse C_i zugeordnet ist, von seinen Merkmalsausprägungen aber eher zur Klasse C_j gehört, erzeugt eine Regel, deren Prämisse eigentlich zur Klassifikation von Mustern der Klasse C_j benötigt würde. Diese Gefahr ist umso größer, je stärker sich die Klassen überlappen.

Um solche Fehler bei der Bestimmung der Regelbasis zu vermeiden, ist es notwendig, die Regeln in einem zweiten Durchgang durch die Lernaufgabe bezüglich ihrer Klassifikationsleistung zu bewerten. Dabei wird sich bei einer von einem „Ausreißer" erzeugten Regel herausstellen, daß die Konklusion dieser Regel geändert werden muß, da die Klassifikationsleistung dieser Regel für eine andere Klasse höher ist, als für die Klasse, zu der der „Ausreißer" gehört. Schließlich kann man die Regeln nach dem Erfolg ihres Klassifikationsergebnisses bewerten und nur die besten auswählen, während die übrigen aus der Regelbasis entfernt werden.

Eine solche Regelbewertung mit anschließender Auswahl wird beim „optimalen" und beim „klassenoptimales" Regellernen durchgeführt. Im Gegensatz zum „einfachen" Regellernen werden bei diesen beiden Varianten im ersten Durchlauf durch die Lernaufgabe soviele Regeln wie möglich erzeugt. Die Reduktion auf die vom Benutzer gewünschte Maximalanzahl erfolgt dann durch die sich an die Bewertung der Regeln anschließende Auswahl der Regeln.

Die folgende Definition beschreibt noch einmal formal die Regelerzeugung in einem NEFCLASS–System:

Definition 3.5 (Algorithmus zur Bildung der Regelbasis)
Gegeben sei ein NEFCLASS–System mit n Eingabeeinheiten $x_1, \ldots, x_n$ und m Ausgabeeinheiten $c_1, \ldots, c_m$. Weiterhin sei eine feste Lernaufgabe $\mathcal{L} = \{(\mathbf{i}_1, \mathbf{t}_1), \ldots, (\mathbf{i}_s, \mathbf{t}_s)\}$ bestehend aus s Mustern gegeben, wobei jedes aus einem Eingabemuster $\mathbf{i} \in \mathbb{R}^n$, und einem Ausgabemuster $\mathbf{t} \in \{0,1\}^m$ besteht. Die gewünschte Maximalanzahl an Regeleinheiten sei mit k_{max} bezeichnet. Zu Beginn des Regelerzeugungsprozesses enthalte das System keine Regeln.

Der Lernalgorithmus, der die Regeleinheiten des NEFCLASS–Systems bestimmt, besteht aus den folgenden Schritten:

(i) Wähle das nächste Muster $(\mathbf{i}, \mathbf{t})$ *aus* $\mathcal{L}$ *und propagiere es.*

(ii) Für jede Eingabeeinheit $x_i \in U_1$ *finde die Zugehörigkeitsfunktion* $\mu_{j_i}^{(i)}$ *mit*

$$\mu_{j_i}^{(i)}(a_{x_i}) = \max_{j \in \{1,\ldots,q_i\}} \{\mu_j^{(i)}(a_{x_i})\}.$$

(iii) Wenn noch keine Regeleinheit R *mit*

$$W(x_1, R) = \mu_{j_1}^{(1)}, \ldots, W(x_n, R) = \mu_{j_n}^{(n)}$$

existiert, dann erzeuge eine solche Einheit und verbinde sie mit der Ausgabeinheit c_l*, falls* $t_l = 1$ *ist.*

Im Falle des „einfachen Regellernens“ beende den Algorithmus, falls die Maximalanzahl an Regeleinheiten k_{max} *erreicht ist, ansonsten fahre mit Schritt (iv) fort.*

(iv) Falls es noch nicht verarbeitete Muster in $\mathcal{L}$ *gibt, fahre mit Schritt (i) fort, ansonsten mit Schritt (v).*

(v) Im Falle des „optimalen“ bzw. „klassenoptimalen“ Regellernens“ bestimme auf eine der beiden folgenden Arten eine Regelbasis aus den erzeugten Regeln:

- **Optimales Regellernen***: Durchlaufe* $\mathcal{L}$ *erneut und akkumuliere für jede Regeleinheit ihre Aktivierungen getrennt nach den Klassen der propagierten Muster. Falls eine Regel* R *für eine Klasse* C_j *eine höhere akkumulierte Aktivierung aufweist, als für die Klasse* C_R*, die durch ihre Konklusion spezifiziert wird, dann ändere die Konklusion der Regel auf* C_j*.*

  Durchlaufe $\mathcal{L}$ *erneut und bestimme für jede Regeleinheit* R *den Wert*

  $$V_R = \sum_{p \in \mathcal{L}} a_R^{(p)} \cdot e_p, \quad e_p = \begin{cases} 1, & \text{falls } p \text{ korrekt klassifiziert wurde,} \\ -1, & \text{sonst.} \end{cases}$$

  Behalte nur die k_{max} *Regeleinheiten mit den höchsten* V_R*–Werten und entferne die übrigen aus dem NEFCLASS–System.*

- **Klassenoptimales Regellernen***: Verfahre wie beim optimalen Regellernen, behalte jedoch je Klasse* C_j *die* $\left\lfloor \frac{k_{max}}{m} \right\rfloor$ *besten Regeln, deren Konklusionen der Klasse* C_j *entsprechen (*$\lfloor \cdot \rfloor$ *bezeichnet die nächstkleinere ganze Zahl).*

Wie in Definition 3.5 beschrieben, bietet der NEFCLASS–Lernalgorithmus dem Anwender drei verschiedene Möglichkeiten, um eine Regelbasis heranzubilden.

Normalerweise wird sich ein Benutzer aufgrund der oben bereits erwähnten Probleme beim „einfachen“ Regellernen für das „optimale“ oder „klassenoptimale“ Regellernen entscheiden. Das „klassenoptimale“ Regellernen ist immer dann vorzuziehen, wenn vermutet wird, daß die Muster sich je Klasse auf eine etwa gleichgroße Anzahl von Clustern verteilen. Da jedes Cluster durch eine Fuzzy–Regel repräsentiert wird, sollte in diesem Fall die gleiche Anzahl von Regeln pro Klasse eingesetzt werden.

Ist die Clusteranzahl mindestens einer Klasse vermutlich signifikant größer als die der anderen Klassen oder sind die Clusteranzahlen der Klassen generell sehr unterschiedlich, so sollte das „optimale“ Regellernverfahren eingesetzt werden. Dabei muß jedoch darauf geachtet werden, daß die Maximalananzahl $k_{\max}$ so groß gewählt wird, daß nach der Auswahl der besten Regeln auch alle Klassen abgedeckt werden.

Das Algorithmus zur Bestimmung der Regelbasis eines NEFCLASS–Systems ist sehr viel weniger aufwendig als das z.B. das Verfahren des verwandten NEFCON–Modells, da NEFCLASS eine feste Lernaufgabe verwendet und die Regeln daher inkrementell gebildet werden können. In Abhängigkeit von der gewählten Alternative zur Regelauswahl („einfach“, „optimal“, „klassenoptimal“) liegt nach spätestens drei Epochen eine Regelbasis vor.

Die Anzahl möglicher Regeln ist nach oben zum einen durch die Anzahl s an Musterpaaren $(\mathbf{i}, \mathbf{t}) \in \mathcal{L}$ beschränkt, d.h. im Extremfall kann jedes der Eingabemuster die Bildung einer Regel verursachen. Außerdem ist die maximale Regelanzahl beschränkt durch die Maximalzahl möglicher Kombinationsmöglichkeiten der linguistischen Terme, da nach Definition 3.3, Bedingung (iv) keine zwei Regeln der Regelbasis identische Prämissen haben dürfen. Ist der Wertebereich jedes Eingabemerkmals x_i durch jeweils q_i Fuzzy–Mengen partitioniert, so lassen sich genau $\prod_{i=1}^{n} q_i$ verschiedene Kombinationen der mit den linguistischen Termen assoziierten Fuzzy–Mengen bilden. Zusammengefaßt kann die Regelbasis eines NEFCLASS–Systems also im ungünstigsten Fall $k = \min\{s, \prod_{i=1}^{n} q_i\}$ Regeln enthalten. Im Regelfall, d.h. unter der Voraussetzung, daß die Muster eine „einigermaßen gutartige“ Verteilung aufweisen, werden sich jedoch immer sehr viel weniger Regeln heranbilden, also $k << \min\{s, \prod_{i=1}^{n} q_i\}$.

Die Erzeugung der Regelbasis eines NEFCLASS–Systems hat den Aufwand $O(s \cdot n \cdot q)$, wobei n die Anzahl der Merkmale eines Eingabemusters und q die Anzahl der Fuzzy–Mengen pro Fuzzy–Partition ist. In der Regel wird die Anzahl der Muster s hierbei den größten Einfluß nehmen. Die sich beim „optimalen“ bzw. „klassenoptimalen“ Regellernen anschließende Bewertung der zunächst erzeugten $\tilde{k}$ Regeln zur Auswahl der besten k Regeln hat den Aufwand $O((s + k) \cdot \tilde{k})$ [Nauck et al, 1996].

3.5 Erlernen der Fuzzy–Mengen

Wie wir im vorangegangenen Abschnitt gesehen haben, erzeugt ein NEFCLASS–System zu Beginn des Lernprozesses zunächst eine Regelbasis $R_1, \ldots, R_k$ (alternativ kann diese Regelbasis natürlich auch — partiell oder vollständig — vom Anwender vorgegeben werden, wie es in Beispiel 3.4 demonstriert wurde). Die in den Prämissen der Fuzzy–Regeln enthaltenen linguistischen Terme $A_j^{(i)}$ werden dabei durch die Fuzzy–Mengen $\mu_j^{(i)}$, $i = 1, \ldots, n$, $j = 1, \ldots, q_i$ repräsentiert, die durch eine Partitionierung des Wertebereichs jedes Eingabemerkmals x_i durch q_i Fuzzy–Mengen erzeugt werden.

Nachdem die Regelbasis erzeugt wurde, paßt der Lernalgorithmus in einem zweiten Schritt diese Zugehörigkeitsfunktionen der Prämissen an. In einem NEFCLASS–System finden für die Fuzzy–Mengen $\mu_j^{(i)}$ Dreiecksfunktionen Verwendung, die jeweils durch drei Parameter a, b und c mit $a < b < c$ festgelegt sind. Der Parameter b bestimmt dabei die Spitze des Dreiecks (d.h. den Abszissenwert, für den die betrachtete Fuzzy–Menge den Zugehörigkeitsgrad 1 liefert), die Parameter a und c bestimmen den linken bzw. rechten Fußpunkt. Der Zugehörigkeitsgrad $\mu(x)$ berechnet sich dann wie folgt:

$$\mu : \mathbb{R} \to [0,1], \quad \mu(x) = \begin{cases} \frac{x-a}{b-a} & \text{falls } x \in [a,b), \\ \frac{c-x}{c-b} & \text{falls } x \in [b,c], \\ 0 & \text{sonst.} \end{cases} \tag{3.5}$$

Um die Parameter der Fuzzy–Mengen erlernen zu können, muß zunächst ein geeignetes Fehlermaß definiert werden. Wie beim verwandten NEFCON–Modell soll auch hier ein Fuzzy–Fehlermaß [Nauck et al, 1996] Anwendung finden, wobei wir beim NEFCLASS–Modell natürlich die aus der Klassifikationsgabe bekannte korrekte Ausgabe auf ein Eingabemuster zur Definition des Fuzzy–Fehlers benutzen können. Die folgende Definition legt dieses Fehlermaß fest.

Definition 3.6 *Gegeben seien ein NEFCLASS–System und eine Klassifikationsaufgabe $\mathcal{L}$. Der* Fuzzy–Fehler *einer Ausgabeeinheit c des NEFCLASS–Systems bezüglich eines Musters $p = (\mathbf{i}^{(p)}, \mathbf{t}^{(p)}) \in \mathcal{L}$ ist*

$$E_c^{(p)} = 1 \; - \; e^{-\beta(t_c^{(p)} - o_c^{(p)})^2}, \quad (c = 1, \ldots, n)$$

mit $\beta \in \mathbb{R}, \beta > 0$.

Mit dem Parameter β kann die Fehlersensitivität beeinflußt werden. Wird β relativ klein gewählt ($\beta \to 0$), werden auch größere Abweichungen toleriert, d.h. führen zu einem geringen Fehler. Bei größeren β–Werten werden dagegen schon kleinere Abweichungen einer Ausgabeeinheit vom gewünschten Ergebnis mit einem großen Fehlerwert $E_c^{(p)}$ bestraft.

Nachdem das Fehlermaß festgelegt ist, kann nun der Algorithmus zur Adaption der Fuzzy–Mengen vorgestellt werden.

Definition 3.7 Lernalgorithmus zur Adaption der Fuzzy–Mengen
Gegeben seien ein NEFCLASS–System und eine Lernaufgabe wie in Definition 3.5. Weiterhin seien Lernraten $\sigma_a, \sigma_b, \sigma_c \in \mathbb{R}$ $(\sigma_a, \sigma_b, \sigma_c > 0)$ gegeben. Der überwachte Lernalgorithmus eines NEFCLASS–Systems zur Anpassung seiner Fuzzy–Mengen läuft zyklisch durch die Lernaufgabe $\mathcal{L}$ und führt dabei die folgenden Schritte solange wiederholt aus, bis eine gegebene Abbruchbedingung erfüllt ist :

(i) *Wähle das nächste Muster $(\mathbf{i}, \mathbf{t})$ aus $\mathcal{L}$, propagiere es durch das NEFCLASS–System und bestimme den Ausgabevektor $\mathbf{c}$.*

(ii) *Bestimme für jede Ausgabeeinheit $c \in U_3$ den Deltawert*

$$\delta_c^{(p)} = \operatorname{sgn}\left(t_c^{(p)} - o_c^{(p)}\right) \cdot E_c^{(p)},$$

wobei $E_c^{(p)}$ der Fuzzy–Fehler gemäß Definition 3.6 ist. sgn *bezeichnet dabei die übliche Signum–(Vorzeichen) Funktion.*

(iii) *Für jede Regeleinheit $R \in U_2$ mit $o_R > 0$:*

(a) *Bestimme den Deltawert*

$$\delta_R^{(p)} = o_R^{(p)}\left(1 - o_R^{(p)}\right) \sum_{c \in U_3} W(R, c)\, \delta_c^{(p)}.$$

(b) *Finde die Eingabeeinheit x' so daß*

$$W(x', R)(o_{x'}^{(p)}) = \min_{x \in U_1} \{W(x, R)(o_x^{(p)})\}.$$

gilt.

(c) *Bestimme für die Fuzzy–Menge $W(x', R)$ die Deltawerte ihrer Parameter a, b, c unter Verwendung der Lernraten $\sigma_a, \sigma_b, \sigma_c > 0$:*

$$\begin{aligned}
\delta_b^{(p)} &= \sigma_a \cdot \delta_R^{(p)} \cdot (c - a) \cdot \operatorname{sgn}(o_{x'}^{(p)} - b), \\
\delta_a^{(p)} &= -\sigma_b \cdot \delta_R^{(p)} \cdot (c - a) + \delta_b^{(p)}, \\
\delta_c^{(p)} &= \sigma_c \cdot \delta_R^{(p)} \cdot (c - a) + \delta_b^{(p)}.
\end{aligned}$$

Addiere diese Deltawerte zu den Parametern von $W(x', R)$, falls dies nicht gegen gegebene Randbedingungen Φ verstößt.

(iv) *Falls am Ende einer Epoche das Abbruchkriterium erfüllt ist, dann halte an, ansonsten fahre mit Schritt (i) fort.*

Das Lernverfahren für Fuzzy–Mengen ist eine einfache Heuristik: Nach der Propagation eines Muster $p \in \mathcal{L}$ wird für jede Regeleinheit der Regelbasis, für deren Aktivierung

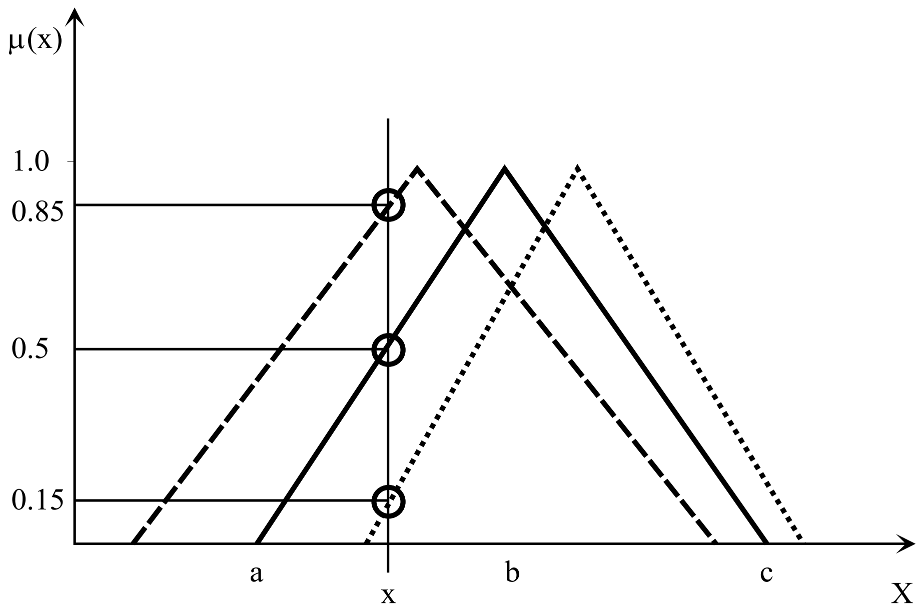

Abbildung 3.3: Das Lernverfahren für Fuzzy–Mengen in NEFCLASS: Die mit durchgezogenen Linien gezeichnete Funktion stellt die Ausgangssituation dar. Soll der Treffergrad erhöht werden, so gelangt man zur gestrichelt dargestellten Situation, andernfalls zur gepunktet gezeichneten.

$a_R > 0$ gilt, anhand der korrekten Ausgabe $\mathbf{t}^{(p)}$ ermittelt, ob die Aktivierung dieser Regeleinheit für das aktuelle Muster verstärkt oder abgeschwächt werden muß.

Hierzu wird für jede Regel die Fuzzy–Menge in der Prämisse der Regel bestimmt, die aufgrund ihres minimalen Zugehörigkeitsgrades (für das aktuelle Merkmal) für die Höhe der Regelaktivierung verantwortlich ist. Die Parameter dieser Fuzzy–Menge werden dann derart geändert, daß der Zugehörigkeitsgrad des entsprechenden Merkmals des Musters p bei der nächsten Propagation von p entsprechend höher (bzw. niedriger) ist, so daß die Aktivierung der betrachteten Regel dann auch höher (niedriger) ausfallen wird. Die Änderung der Parameter a, b, c der Fuzzy–Menge verursacht eine Verschiebung und eine Verbreiterung bzw. Verengung der entsprechenden Dreiecksfunktion. Die Vorgehensweise ist in Abbildung 3.3 dargestellt.

Bemerkung:

(i) Da ein bestimmter linguistischer Term in den Prämissen mehrerer Regeln vorkommen kann, wird die mit diesem linguistischen Term assoziierte Fuzzy–Menge nach der Propagation eines Musters gegebenfalls mehrfach verändert.

(ii) Der Faktor $o_R\,(1 - o_R)$ in Schritt (iii.a) des Lernalgorithmus für Fuzzy–Mengen soll bewirken, daß Regeleinheiten, die eine Aktivierung von ungefähr 0.5 bei der Propagation eines bestimmten Musters aufweisen, „gezwungen“ werden, sich für oder gegen eine Reaktion (durch ihre Aktivierung) bezüglich dieses Musters zu entscheiden.

Durch diesen Faktor wird erreicht, daß die Änderungen an der die Aktivierung der Regeleinheit bestimmende Fuzzy–Menge umso größer sind, je näher die Aktivierung einer Regeleinheit bei einem Wert von etwa 0.5 liegt und umso kleiner, je näher sich die Aktivierung den optimalen Werten 1 bzw. 0 nähert.

(iii) Gegeben sei die Fuzzy–Partitionierung $\mu_1, \ldots, \mu_q$ eines abgeschlossenen Intervalls $[x_l, x_r]$. Die Parameter a, b, c der Dreiecksfunktionen seien analog zu den Fuzzy–Mengen bezeichnet mit $(a_1, b_1, c_1), \ldots, (a_q, b_q, c_q)$. Die in Schritt (iii.c) des Lernverfahrens für Fuzzy–Mengen genannten Randbedingungen Φ legen fest, daß

(a) die Dreiecksform der Zugehörigkeitsfunktionen erhalten bleibt:

$$\forall_{i=1,\ldots,q}: \; a_i < b_i < c_i,$$

(b) die Fuzzy–Mengen den Wertebereich ihres Merkmals nicht verlassen dürfen:

$$\forall_{i=1,\ldots,q}: \; ((a_i, b_i, c_i \geq x_l) \wedge (a_i, b_i, c_i \leq x_r)),$$

(c) ihre relative Position zueinander erhalten bleiben muß:

$$(x_l \leq a_1 < a_2 < \ldots < a_q) \wedge (c_1 < c_2 < \ldots < c_q \leq x_r).$$

Von einem Neuro–Fuzzy–System wie NEFCLASS können wir keine exakten Ausgaben erwarten. Daher ist auch das Abbruchkriterium des Lernvorgangs nicht einfach anzugeben. Der Gesamtfehler, den das System bezogen auf die gesamte Lernaufgabe $\mathcal{L}$ macht, ergibt sich bei einem System mit m Ausgabeeinheiten zu:

$$E = \sum_{p \in \mathcal{L}} E^{(p)} = \sum_{p \in \mathcal{L}} \sum_{c=1}^{m} E_c^{(p)} = \sum_{p \in \mathcal{L}} \sum_{c=1}^{m} 1 \; - \; e^{-\beta(t_c^{(p)} - o_c^{(p)})^2} \tag{3.6}$$

Dieser Gesamtfehler kann aufgrund der Definition von net_c in Definition 3.3 normalerweise nicht 0 werden. Das Abbruchkriterium sollte daher keine Fehlerschranke, sondern z.B. eine Maximalzahl zulässiger Fehlklassifikationen sein; alternativ kann der Lernvorgang auch abgebrochen werden, wenn der Gesamtfehler während einer gewissen Zahl von Epochen nicht weiter fällt.

Kapitel 4

Die Implementierung des Softwaretools NEFCLASS

Im Rahmen dieser Diplomarbeit wurde das NEFCLASS–Modell vom Autor in einem Softwaretool realisiert. Zielsetzung war es dabei, ein leicht zu handhabendes Tool zu entwickeln, das aus Beispieldaten ein NEFCLASS–System erzeugen und dann unbekannte Daten klassifizieren kann. Weiterhin sollte der Benutzer die Möglichkeit haben, sich die Fuzzy–Partitionen der Eingangsvariablen graphisch darstellen zu lassen und eigene Regeln (d.h. Vorwissen) in das System integrieren zu können.

Ausgehend von dieser Zielsetzung und dem Bestreben, ein möglichst *portables* Tool zu entwickeln, fiel dann die Entscheidung, das Tool in zwei Programme derart aufzuteilen, daß ein Programm das eigentliche NEFCLASS–Modell und die dazugehörenden Algorithmen vollständig implementiert, und das andere Programm lediglich als graphisches „Frontend“ für das erste Programm fungiert. Diese Aufteilung trägt der Tatsache Rechnung, daß bei der Softwareportierung häufig die graphische Oberfläche *(graphical user interface, GUI)* die größte Hürde darstellt. Die Idee der Aufteilung des Tools besteht also darin, den Kern portierbar zu gestalten und lediglich die graphische Ergebnisausgabe (bzw. eine graphische Benutzeroberfläche) jeweils abhängig vom Betriebssystem zu implementieren.

Konkret wurde so das kommandogesteuerte Softwaretool *NEFCLASS* entwickelt, das das NEFCLASS–Modell vollständig realisiert und ähnlich zu bedienen ist wie z.B. *gnuplot*. Genau wie viele andere kommandogesteuerte Programme kennt auch NEFCLASS einen interaktiven und einen Batch–Modus, d.h. die Möglichkeit, Kommandos in eine Datei zu schreiben und diese dann sequentiell von NEFCLASS abarbeiten zu lassen. Dieser Batch–Modus ist vor allem bei großen oder hochdimensionalen Datensätzen sehr sinnvoll, da der Lernvorgang in diesen Fällen viel Rechenzeit benötigt und ein interaktiver Modus somit wenig sinnvoll ist. Das NEFCLASS–Tool wurde in C++ implementiert und sollte somit auf alle gängigen Betriebssystemplattformen, auf denen ein C++–Compiler zur Verfügung steht, ohne größere Probleme portiert werden können.

Der C++–Compiler sollte allerdings in der Lage sein, *templates* (generische Objekte, siehe z.B. [Stroustrup, 1995] oder [Lipman, 1990]) zu handhaben, da die Implementierung von Vektoren und Matrizen in NEFCLASS (Datentypen *TVector* und *TMatrix*) von diesem Konstrukt Gebrauch macht. Die Übersetzung der Quelltexte mit einem nicht template–fähigen Compiler erfordert die Aufteilung des *TVector*– und des *TMatrix*–Objekts in jeweils zwei Objekte: jeweils eines für reelle und eines für ganze Zahlen.

Der Autor selbst hat die Quelltexte problemlos unter SUN–OS, Linux (jeweils mit dem GNU–Compiler *gcc* in der Version 2.7.2) sowie unter MS–DOS unter Verwendung von Borland's Turbo C++ (Version 3.0) übersetzen können.

Neben dem portablen NEFCLASS–Tool wurde auch ein prototypisches graphisches Frontend für das X–Window–System entwickelt, das über Menüs zu bedienen ist und auch die Fuzzy–Mengen, die die linguistischen Terme repräsentieren, am Bildschirm graphisch darstellt. Dieses mit *NEFCLASS–X* bezeichnete graphische Frontend wurde mit Hilfe des frei verfügbaren (public domain) Tcl/Tk–Toolkits *(Tool command language / Toolkit)* von John K. Ousterhout entwickelt. Tcl ist eine leistungsfähige Skriptsprache, mit der recht einfach und auf einem sehr hohen Abstraktionsniveau Applikationen entwickelt werden können. Tk stellt als Erweiterung zu Tcl GUI–Konstrukte bereit, mit denen es möglich ist, OSF–Motif–ähnliche graphische Benutzeroberflächen für das X–Window–System zu erstellen.

Applikationen, die auf Tcl/Tk basieren, werden zur Laufzeit vom Tcl–Interpreter *interpretiert*, wodurch sie in der Ausführung sicherlich etwas langsamer sind als Anwendungen, welche auf herkömmlichem Wege in Maschinencode übersetzt wurden. Gerade im Bereich der graphischen Benutzeroberflächen sowie unter Berücksichtigung der heute zumeist verfügbaren Rechnerleistung kann dies jedoch vernachlässigt werden. Insgesamt erlaubt Tcl/Tk eine sehr zügige Entwicklung graphischer Oberflächen (oder auch ganzer Anwendungen) und ist damit insbesondere für das *Rapid Prototyping* prädestiniert. Eine gute und ausführliche Beschreibung von Tcl und Tk findet sich in dem von John K. Ousterhout selbst verfaßten Referenzhandbuchs [Ousterhout, 1994].

Für die Implementierung des graphischen Programmteils gibt es grundsätzlich zwei Alternativen bezüglich des Funktionsumfangs, die sich nach den Möglichkeiten des verwendeten Betriebssystems richten:

- Unter den heute gängigen mehrprozeßfähigen Betriebssystemen (UNIX, OS/2, Windows etc.) kann der graphische Programmteil neben der eigentlichen Visualisierung der Fuzzy–Mengen tatsächlich auch eine graphische Benutzeroberfläche zur Verfügung stellen, so daß der Benutzer keine Kommandos mehr eingeben muß, um den Klassifikator zu bedienen. Beide Programmteile laufen dann parallel ab und kommunizieren über einen geeigneten, vom Betriebssystem bereit-gestellten Mechanismus (im Fall von NEFCLASS–X über eine *pipe*).

- Verfügt das angestrebte Betriebssystem nicht über diese Mehrprozeßfähigkeit (wie z.B. MS–DOS), so kann der graphische Programmteil die vom Klassifikator erzielten Ergebnisse lediglich darstellen (insbesondere natürlich die Fuzzy–Partitionen der Eingangsvariablen), der Klassifikator muß dann über die Kommandosprache bedient werden.

  Speziell für MS–DOS gibt es jedoch eine Alternative: Ulrike und Detlef Nauck haben bereits im Vorfeld dieser Arbeit speziell für MS–DOS ein Softwaretool für das NEFCLASS–Modell entwickelt, das über eine menügeführte Benutzeroberfläche verfügt und auch die Fuzzy–Partitionen auf dem Bildschirm darstellt, also beide Anforderungen in einem Programm integriert [NEFCLASS-PC].

Die eben gemachten grundsätzlichen Einschränkungen bezüglich der Portabilität graphischer Benutzeroberflächen können für den speziellen Fall der Tcl/Tk–Implementierung von NEFCLASS–X jedoch (zumindest teilweise) abgeschwächt werden. John K. Ousterhout hat sich mit seinem Team zum Ziel gesetzt, Tcl/Tk zukünftig auch für andere Plattformen als UNIX/X11 zur Verfügung zu stellen. Mit Erscheinen dieser Arbeit sollte mindestens eine Tcl/Tk–Version für sämtliche Microsoft Windows–Versionen (3.x, NT, 95) verfügbar sein. Kurz vor Abschluß dieser Arbeit war bereits eine erste MS–Windows–Version von Tcl/Tk verfügbar, der Autor konnte diese jedoch aus Zeitgründen nicht mehr testen.

Aufgrund dieser Entwicklung besteht die begründete Hoffnung, daß auch der graphische Programmteil in einer zukünftigen Version basierend auf ein und demselben Quelltext auf verschiedenen Betriebssystem–Plattformen lauffähig sein wird.

4.1 Vorbemerkungen zum „Programmierstil“

Dieser Abschnitt erläutert kurz den Programmierstil des Autors, um das Verständnis der Quelltexte zu erleichern.

4.1.1 Bezeichner

Bezeichner von Variablen und Funktionen sind durchgängig in englischer Sprache gehalten, wobei versucht wurde, die Bezeichner so zu wählen, daß der Verwendungszweck des bezeichneten Objekts aus dem Bezeichner hervorgeht. Dabei beginnen die Bezeichner (mit Ausnahme von z.B. Laufvariablen) stets mit einem Großbuchstaben. Besteht ein Bezeichner aus mehreren Teilwörtern, so beginnt jedes Teilwort mit einem Großbuchstaben (Beispiel: *CreateRuleBase*).

Die Bezeichner von selbstdefinierten Typen und Objektklassen sind ebenfalls nach diesen Richtlinien gebildet, zusätzlich beginnen diese mit einem vorangestellten *T* (wie *type*). Beispiele hierfür sind Bezeichner wie *TRule*, *TPartition* und *TFuzzySet*. Soweit für solche selbstdefinierten Typen eigene Zeigertypen deklariert wurden, beginnen diese mit einem *P* (wie *pointer*) statt des *T*. Ein Beispiel hierfür findet sich in der Datei *fuzzynet.h*:

```
/* define 'pointer to rule' */
typedef TRule *PRule;
```

Eine letzte Besonderheit findet sich bei der Benennung von Konstanten: Diese beginnen in der Regel mit einem Präfix, der aus zwei bis drei Kleinbuchstaben besteht und das Auffinden der Deklaration erleichtern soll. So sind z.B. Konstanten für alle in NEFCLASS existierenden Kommandos definiert, deren Bezeichner alle mit dem Präfix „cm" beginnen (z.B. *cmLoad, cmSave* usw.).

4.1.2 Vektoren

Da im NEFCLASS–Modell sehr häufig Vektoren verwendet werden (z.B. die Eingabemuster in der Lernaufgabe), ist in der Implementierung von NEFCLASS hierfür eine eigene Klasse vorgesehen. Diese mit *TVector* bezeichnete Klasse bietet einige Vorteile gegenüber der Verwendung eines normalen Arrays:

- In C++ besteht die Möglichkeit, *Operatoren* wie +, – usw. klassenspezifisch zu überschreiben (siehe [Stroustrup, 1995] oder [Lipman, 1990]). Für die Klasse *TVector* wurden daher die wichtigsten Operatoren in folgender Weise überschrieben:
  `+` (Vektoraddition),
  `-` (Vektorsubtraktion),
  `*` (Multiplikation eines Vektors mit einem Skalar),
  `=` (Vektorzuweisung),
  `==` (Vergleich zweier Vektoren).

  Durch die Verwendung dieser Operatoren können Operationen auf Vektoren deutlich lesbarer formuliert werden.

- Weiterhin wurde auch der Operator `[]` für die Klasse *TVector* eingeführt. Dadurch konnten gleich mehrere Ziele realisiert werden:

  – Auf Vektoren kann damit zugegriffen werden wie auf Arrays.

  – Die Indizierung der Elemente von Vektoren kann — wie in der Mathematik üblich — mit 1 beginnen (siehe hierzu auch den folgenden Abschnitt).

  – Durch die Einführung des Operators `[]` für die *TVector*–Klasse kann beim Zugriff auf ein Vektorelement zuvor eine Bereichsüberprüfung des angegebenen Indizes (durch das Vektor–Objekt selbst) stattfinden. Eine derartige

Bereichsüberprüfung für das *TVector*–Objekt ist realisiert und kann optional eingeschaltet werden.

Zu diesem Zweck müssen die Quelltexte mit dem *define* (Symbol für die bedingte Kompilierung) „__RANGECHECK__“ übersetzt werden. Die Bereichsüberprüfung kann während der Entwicklungsphase und bei der Fehlersuche gute Dienste leisten.

4.1.3 Indizierung von Arrays und dynamische Arrays

Normalerweise werden Arrays mit n Elementen in der C/C++–Programmierung von 0 an beginnend bis $n-1$ indiziert. Ist ein Array z.B. durch die Deklaration

```
int IntArray[3];
```

gegeben, so ist das erste Element des Arrays in `IntArray[0]` und das letzte Element in `IntArray[2]` enthalten. Dies ist wenig intuitiv (andere Programmiersprachen wie z.B. PASCAL sind hier flexibler) und nach Ansicht des Autors auch sehr fehlerträchtig im Hinblick auf Bereichsüberschreitungen (*range check errors*).

Aus diesem Grund werden in der Implementierung von NEFCLASS Arrays in der Regel von 1 bis n indiziert, das Element [0] bleibt normalerweise ungenutzt.

Im obigen Beispiel des `IntArray` wurde dessen Größe statisch, d.h. zur Kompilationszeit, festgelegt. C/C++ bietet darüber hinaus auch die Möglichkeit, die Größe eines Arrays erst zur Laufzeit des Programms festzulegen. Ein Beispiel soll dies verdeutlichen:

```
int *IntArray;

    IntArray = new int[4+1];
    for (i = 1; i <= 4; i++)
         IntArray[i] = i*i;
```

Die Array–Variable wird also zunächst als einfacher Zeiger deklariert. Diesem Zeiger wird dann dynamisch (durch den Aufruf von *new* in C++ bzw. *malloc* in C) der entsprechende Speicherplatz zugewiesen, danach kann eine solche Variable wie ein gewöhnliches statisches Array verwendet werden. Von dieser Möglichkeit wird in der Implementierung von NEFCLASS häufig Gebrauch gemacht. Derartige Arrays werden im folgenden als *dynamische Arrays* bezeichnet.

4.2 Beschreibung der wichtigsten Datenstrukturen in NEFCLASS

In diesem Abschnitt werden die zentralen Datenstrukturen der NEFCLASS–Implementierung genauer beschrieben. Diese Datenstrukturen stellen eine mehr oder weniger direkte Umsetzung der in den Definitionen des Kapitels 3 vorgestellten formale Konstrukte dar. Die Diskussion der zentralen Datenstrukturen soll dazu dienen, dem interessierten Leser das Verständnis der Quelltexte des Programms zu erleichtern.

Die folgende Tabelle gibt einen kurzen Überblick über die Datenstrukturen, die auf den nächsten Seiten beschrieben werden.

Datentyp	realisierte Struktur	formale Beschreibung siehe
TPattern	Musterpaar	Def. 3.1
TLearningTask	Klassifikationsaufgabe (Lernaufgabe)	Def. 3.1
TFuzzySet	Fuzzy–Menge	Gleichung 3.5
TPartition	Fuzzy–Partitition	Seite 17
TRule	Fuzzy–Regel	Seite 17
TFuzzyPerceptron	NEFCLASS–System	Def. 3.2, Def. 3.3

Tabelle 4.1: Überblick über die wichtigsten Datenstrukturen in NEFCLASS

Bei der folgenden Beschreibung der Klassen wurde auf die Erläuterung eher „technischer“ Methoden bewußt verzichtet, da diese für alle Objekte den gleichen Aufbau und die gleiche Funktion haben.

Zu diesen „technischen“ Methoden zählen:

- **Default–Konstruktoren:** Ein Default–Konstruktor ist ein Objektkonstruktor, der beim Aufruf kein Argument erhält. Ein solcher Konstruktor wird immer dann benötigt, wenn ein Array aus Objekten einer Klasse erzeugt werden soll. Beim Erzeugen eines solchen Arrays ist es nicht möglich, den gewünschten Konstruktor mitanzugeben, so daß die Initialisierung eines Objekt–Arrays in zwei Schritten erfolgen muß.

- **Kopierkonstruktoren:** Ein Kopierkonstruktor für eine beliebige Klasse X hat die Form

  ```
  X (const X &ObjectToCopy)
  ```

und wird benutzt, um ein neues Objekt mit dem bestehenden Objekt *ObjectToCopy* zu initialisieren. Dabei wird der Kopierkonstruktor einer Klasse nicht nur bei einem expliziten Aufruf benötigt, vielmehr wird dieser durch den Compiler auch immer dann aufgerufen, wenn ein Objekt dieser Klasse einer beliebigen Funktion als *Werte*parameter übergeben wird. Durch den Kopierkonstruktor wird in diesem Fall eine automatisch generierte temporäre Variable initialisiert, die vor Eintritt in die Funktion erzeugt und nach Beendigung dieser Funktion durch einen Destruktor–Aufruf wieder entfernt wird.

Aus diesem Grund existiert für jede Klasse, die dynamisch erzeugte Datenelemente beinhaltet, ein solcher Kopierkonstruktor. Einzige Ausnahme von dieser Regel sind solche Klassen, deren Objekte garantiert nicht als Werteparameter einer Funktion auftreten.

- **Destruktoren:** Der Destruktor einer Klasse ist das Gegenstück zum Konstruktor und sorgt für die Freigabe der von einem Objekt der Klasse belegten Ressourcen. Jede Klasse, die dynamisch erzeugte Datenelemente beinhaltet, hat daher einen Destruktor.

- **Überschriebene Operatoren:** Dazu zählen vor allem Zuweisungsoperatoren (zwingend für Klassen mit dynamisch erzeugten Datenelementen), Gleichheitsoperatoren und benutzerdefinierte Ein–/Ausgabeoperatoren (<</>>).

Weitergehende Ausführungen zu derartigen Methoden sind in [Stroustrup, 1995] und in [Lipman, 1990] zu finden.

Die auf den nächsten Seiten folgende Beschreibung der in Tabelle 4.1 aufgeführten zentralen Datenstrukturen ist für jedes Objekt wie folgt gegliedert:

- Titelzeile (Name des beschriebenen Objekts und Name der Datei, in der die Deklaration zu finden ist).

- verbale Beschreibung des Objekts mit Verweisen zu den entsprechenden formalen Konstrukten in Kapitel 3.

- Deklaration des Objekts als C++–Quelltext

- Beschreibung der einzelnen Datenelemente des Objekts

- Beschreibung der einzelnen Methoden des Objekts

Die Implementierung des jeweiligen Objekts findet sich in der Datei, deren Namen in der Titelzeile angegeben ist, wobei als Dateiendung '.h' durch '.cc' zu ersetzen ist.

TPattern — lrntask.h

Der Datentyp *TPattern* implementiert ein Musterpaar $(\mathbf{x}, \mathbf{t})$ mit $\mathbf{x} \in \mathbb{R}^n$ und $\mathbf{t} \in \{0,1\}^m$, wie es in Definition 3.1 beschrieben ist. Für $\mathbf{t} = (t_1, \ldots, t_m)$ gilt

$$t_j = \begin{cases} 1, & \text{falls } \mathbf{x} \in C_j \\ 0, & \text{sonst.} \end{cases}$$

$C_j \subseteq \mathbb{R}^n$ ist dabei die Menge von Eingabemustern, die die Klasse j repräsentieren.

Deklaration

```
typedef struct TPattern {
    TVector<Real> Input;
    int           Class;
};
```

Daten–elemente

Input

TVector<Real> **Input**

Der *Input*–Vektor stellt das Eingabemuster $\mathbf{x}$ des Musterpaares $(\mathbf{x}, \mathbf{t})$ dar, d.h. *Input* enthält die Merkmale $x_1, \ldots, x_n$.

Class

int **Class**

Die Variable *Class* gibt die Klasse an, zu der der *Input*–Vektor gehört. Die Klasseninformation wird dabei aus der Musterdatei zwar noch in Form eines binären Vektors eingelesen, danach wird die Klassenzugehörigkeit aber nur noch in der integer–Variablen *Class* gespeichert, um Speicherplatz zu sparen.

Ist *Class* $= 0$, so handelt es sich um ein Muster, dessen Klassenzugehörigkeit unbekannt ist (d.h. durch den Klassifikator bestimmt werden soll, vgl. Option *„NOCLASS“* in Abschnitt A.6).

Bemerkung:
Wird das NEFCLASS–Modell zu einem späteren Zeitpunkt dahingehend erweitert, daß ein gegebenes Eingabemuster zu *mehreren* Klassen gehören kann (d.h. der Vektor $\mathbf{t}$ kann mehr als eine 1 enthalten), so muß die Klasseninformation eines Musters dann wieder als Vektor gespeichert werden (z.B. als *TVector<int>* oder als Bitvektor).

TLearningTask lrntask.h

TLearningTask realisiert die in Definition 3.1 formal beschriebene Klassifikationsaufgabe $\mathcal{L} = \{(\mathbf{i}_1, \mathbf{t}_1), \ldots, (\mathbf{i}_s, \mathbf{t}_s)\}$ mit $\mathbf{i}_i \in \mathbb{R}^n$ und $\mathbf{t}_i \in \{0,1\}^m, (i = 1, \ldots, s)$.

Die Muster der Lernaufgabe können dabei entweder bereits klassifizierte Muster („Beispieldaten") oder aber Muster unbekannter Klassenzugehörigkeit sein. Erstere dienen zum Erzeugen und Testen eines NEFCLASS–Systems, letztere können durch ein zuvor erstelltes und trainiertes NEFCLASS–System klassifiziert werden.

Dieses Objekt wird immer aus einer *Daten–* bzw. *Musterdatei* eingelesen. Das Dateiformat für Musterdateien ist im Anhang in Abschnitt A.6 beschrieben.

Deklaration

```
class TLearningTask {
 public:
    int             InDimension,
                    OutDimension;
    int             Count;
    char            Name[80];
    TVariableName   *InputName,
                    *ClassName;
    Boolean         ContainsClassInfo;
    TPattern        *Pattern;
    TBound          *Bounds;
    TVector<int>    PatternPerClass;
    Real            MissingValue;
    int             MissingValues;

    TLearningTask (istream     &s,
                   TLoadStatus &Status,
                   ostrstream  &Msg);

    TLearningTask (const TLearningTask &T);
    ~TLearningTask();

    Boolean NamesAvail (void) const;
```

```
    void PrintInfo (ostream    &s,
                    const char *Keyword,
                    const char *Filename);
 private:
    void ComputeBounds (TBound *BoundsSpecified);
}; /* TLearningTask */
```

Daten–elemente

InDimension, OutDimension — int **InDimension, OutDimension**

Die Ein–/bzw. Ausgabedimension der Klassifikationsaufgabe, d.h. Anzahl Merkmale pro Eingabemuster bzw. Anzahl der Klassen.

Count — int **Count**

Gibt die Anzahl an Musterpaaren in der Lernaufgabe an.

Name — char **Name[80]**

Optional zu vergebender Name der Lernaufgabe (siehe hierzu auch den Befehl *NAME* in Abschnitt A.6).

InputName, ClassName — TVariableName ***InputName, *ClassName**

Dynamische Arrays, die die (optionalen) Bezeichnungen für die Eingabemerkmale und die Klassen enthalten. Diese Namen können durch den Benutzer mittels der *VARNAMES*–Anweisung in der Musterdatei angegeben werden (siehe Anhang A.6). Werden vom Benutzer keine Namen spezifiziert, so gilt *InputName* = *ClassName* = *NULL*. Der Typ *TVariableName* ist ebenfalls in der Datei lrntask.h definiert:

```
    typedef char TVariableName[30];
```

Contains–ClassInfo — Boolean **ContainsClassInfo**

Die boolesche Variable *ContainsClassInfo* gibt an, ob die Klassenzugehörigkeit der in der Klassifikationsaufgabe enthaltenen Musterpaare bekannt ist, d.h. ob es sich um Musterpaare handelt, die zum Lernen und Testen geeignet sind. Enthält die Lernaufgabe Muster, deren Klassenzugehörigkeit unbekannt ist (die also mittels eines trainierten NEFCLASS–Systems klassifiziert werden sollen), so gilt

ContainsClassInfo = *False*, und für jedes Muster p der Lernaufgabe gilt $p.Class = 0$.

Eine Klassifikationsaufgabe enthält immer entweder nur klassifizierte oder nur zu klassifizierende Muster, niemals beide Musterarten zugleich.

Pattern

TPattern ***Pattern**

Dynamisch erzeugtes Array, das die eigentlichen Musterpaare enthält. *Pattern[1]* enthält das erste Musterpaar, *Pattern[Count]* das letzte.

Siehe auch: *TPattern.*

Bounds

TBound ***Bounds**

Die Variable *Bounds* ist ein dynamisch erzeugtes Array, das für jedes Eingabemerkmal die Grenzen des Wertebereichs enthält. Diese Grenzen können vom Benutzer mittels der *INRANGES*–Klausel in der Musterdatei angegeben werden, andernfalls werden sie durch Analyse der Muster automatisch bestimmt. Der Typ *TVariableName* ist in der Datei *fuzzy.h* definiert:

```
typedef struct TBound {
    Real  Min, Max;
};
```

Pattern–PerClass

TVector<int> **PatternPerClass**

Enthält für jede Klasse die Anzahl der ihr zugeordneten Eingabemuster. Dieser Vektor wird nur für statistische Zwecke (von der Methode *PrintInfo*) benötigt.

MissingValue

Real **MissingValue**

Die Variable *MissingValue* gibt die reelle Zahl an, die als „fehlender Wert“ *missing value)* interpretiert werden soll und in der Musterdatei durch die *MISSINGVALUE*–Anweisung spezifiziert werden kann. Dies ist sinnvoll, wenn bei einigen (wenigen) Eingabemustern die Merkmalsausprägung eines bestimmten Merkmals unbekannt ist.

MissingValues — int **MissingValues**

Gibt die Gesamtzahl von *missing values* in der Lernaufgabe an. Diese Variable wird nur für statistische Zwecke (von der Methode *PrintInfo*) benötigt.

Methoden

Stream–Konstruktor — **TLearningTask** (istream &s,
TLoadStatus &Status,
ostrstream &Msg)

Erzeugen eines *TLearningTask*–Objekts durch Einlesen der Daten aus dem Stream *s* (in der Regel aus einer Datei). Das Datenformat für Musterdateien ist im Anhang in Abschnitt A.6 beschrieben. Der Parameter *Status* gibt nach Beendigung des Konstruktors Aufschluß über den Erfolg der Operation. Der Datentyp *TLoadStatus* ist in der Datei *stdutils.h* als Aufzählungstyp definiert:

```
typedef enum {
    ldOk      =  0,
    ldWarning = -1,
    ldError   = -2
} TLoadStatus;
```

Ist *Status* ungleich *ldOk*, so befindet sich die entsprechende Fehler– bzw. Warnmeldung in der Streamvariablen *Msg*. Im Fall von *Status* == ldError ist das Objekt unbrauchbar und darf nicht weiter verwendet werden.

NamesAvail — Boolean **NamesAvail** (void) const

Die Funktion liefert *True*, wenn Bezeichnungen für die Eingabemerkmale und die Klassen verfügbar sind, anderenfalls *False*.

Siehe auch: *InputName, ClassName.*

PrintInfo — void **PrintInfo** (ostream &s,
const char *Keyword,
const char *Filename)

PrintInfo schreibt die wichtigsten Angaben über die Lernaufgabe (Anzahl Eingabemerkmale, Anzahl Klassen, Anzahl Muster, Anzahl *missing values*, etc.) in den Stream *s*. Der Parameter *Filename* ist dabei der Dateiname der Musterdatei, aus dem die Daten gelesen wurden (das Objekt verfügt selbst nicht über diese Information, siehe Konstruktor).

Ist die übergebene Zeichenkette *Keyword* ungleich NULL, so wird diese zuerst (gefolgt von einem Zeilenvorschub) in den Stream geschrieben.

Compute–Bounds

void **ComputeBounds** (TBound *BoundsSpecified)

Diese Methode berechnet die Intervallgrenzen $[x^{(i)}_{min}, x^{(i)}_{max}]$ des Wertebereichs für jedes Eingabemerkmal ($i = 1, \ldots, n$). Der Parameter *BoundsSpecified* gibt dabei die vom Benutzer mittels der *INRANGES*–Anweisung (siehe Abschnitt A.6 des Anhangs) angegebenen Grenzen an (falls der Benutzer keine solche Angabe gemacht hat, gilt *BoundsSpecified == NULL*).

Sind die vom Benutzer gewünschten Intervalle größer als eigentlich notwendig (z.B. auf- bzw. abgerundet auf die nächste ganze Zahl), so bleiben diese erhalten, lediglich zu klein gewählte Intervalle werden durch die berechneten ersetzt (anderenfalls würden Merkmale einzelner Muster außerhalb des Wertebereichs liegen).

TFuzzySet fuzzy.h

TFuzzySet ist die Implementierung einer Fuzzy–Menge μ, die den Zugehörigkeitsgrad $\mu(x)$ zu einem gegebenen $x \in \mathbb{R}$ wie in Formel 3.5 (Dreiecksfunktion) berechnet und durch die drei Parameter a, b, c festgelegt wird. Die Fuzzy–Menge kann optional links– oder rechtsseitig (auch beidseitig) geschultert sein.

Deklaration

```
class TFuzzySet {
 public:
    char    Name[20];
    Real    a, b, c;
    Boolean LeftShouldered,
            RightShouldered;

    TFuzzySet();
    TFuzzySet (const char *AName,
               Real a, Real b, Real c,
               Boolean ShoulderedLeft,
               Boolean ShoulderedRight);
    TFuzzySet (istream& s);

    /* degree of membership of value x */
    Real Membership (Real x) const;

    /* change the parameters a, b, c */
    void Adapt (Real Delta_a,
                Real Delta_b,
                Real Delta_c);

    /* overloaded stream operators */
    friend ostream& operator << (ostream         &s,
                                 const TFuzzySet &F);

}; /* TFuzzySet */
```

Daten–elemente

Name

char **Name[20]**

Linguistischer Term, mit dem die Fuzzy–Menge assoziiert wird.

a, b, c — Real **a, b, c**

Die Parameter a, b, c der Dreiecksfunktion.

Left–/Right–Shouldered — Boolean **LeftShouldered, RightShouldered**

Boolesche Variablen, die angeben, ob die Fuzzy–Menge links– bzw. rechtsseitig geschultert ist (siehe hierzu auch die Beschreibung der Methode *Membership*). Eine Fuzzy–Menge kann auch beidseitig geschultert sein; dies macht jedoch nur dann Sinn, wenn der gesamte Wertebereich eines Merkmals durch eine Fuzzy–Menge überdeckt werden soll (*don't care*), so daß jeder Wert x des Wertebereichs den Zugehörigkeitsgrad $\mu(x) = 1$ liefert.

Methoden

Konstruktor — **TFuzzySet** (const char *AName,
Real a, Real b, Real c,
Boolean ShoulderedLeft,
Boolean ShoulderedRight)

Der Konstruktor der Klasse *TFuzzySet* erzeugt eine Fuzzy–Menge mit den Dreiecksparameter *a, b, c*. Die Fuzzy–Menge kann dabei rechts–, links– oder auch beidseitig „geschultert" sein, was durch die beiden *ShoulderedLeft* und *ShoulderedRight* festgelegt wird. Der linguistische Term, der mit der Fuzzy–Menge assoziiert wird, kann durch den Parameter *AName* angegeben werden.

Stream–Konstruktor — **TFuzzySet** (istream &s)

Der Stream–Konstruktor erzeugt eine Fuzzy–Menge durch Einlesen der Parameter aus dem Stream s. Das Datenformat für eine einzelne Fuzzy–Menge ist im Abschnitt A.6 unter dem Schlüsselwort *FUZZY* beschrieben.

Membership — Real **Membership** (Real x)

Die Methode *Membership* berechnet den Zugehörigkeitsgrad $\mu(x)$ zu einem gegebenen $x \in \mathbb{R}$ nach folgender Formel:

$$\mu(x) = \begin{cases} \dfrac{x-a}{b-a} & \text{falls } x \in [a, b), \\ \dfrac{c-x}{c-b} & \text{falls } x \in [b, c], \\ 0 & \text{sonst.} \end{cases}$$

Gilt *LeftShouldered* = *True*, so gilt abweichend von obiger Formel

$$\mu(x) = 1 \quad \text{für } x \in [a, b]$$

und analog für *RightShouldered* = *True*

$$\mu(x) = 1 \quad \text{für } x \in [b, c].$$

Adapt

void **Adapt** (Real Delta_a, Real Delta_b, Real Delta_c)

Die Methode *Adapt* dient dazu, die Parameter a,b und c der Fuzzy–Menge anzupassen. Die übergebenen Delta–Werte werden zu diesem Zweck auf die bisherigen Parameter aufaddiert, d.h. die neuen Parameter der Fuzzy–Menge $a^{(neu)}$,$b^{(neu)}$,$c^{(neu)}$ ergeben sich durch:

$$\begin{aligned} a^{(neu)} &= a^{(alt)} + Delta\_a, \\ b^{(neu)} &= b^{(alt)} + Delta\_b, \\ c^{(neu)} &= c^{(alt)} + Delta\_c. \end{aligned}$$

Parameteränderungen, die eine Verletzung der Randbedingung

$$a^{(neu)} < b^{(neu)} < c^{(neu)} \quad \text{(Einhaltung der Dreiecksform)}$$

zur Folge haben würden, werden von der Methode zurückgewiesen.

TPartition fuzzy.h

TPartition ist das Objekt zur Realisierung von Fuzzy–Partitionen. In der vorliegenden Implementierung kann eine Fuzzy–Partition zwischen einer und sieben Fuzzy–Mengen enthalten. Eine Fuzzy–Partition mit nur einer Fuzzy–Menge kann zur Modellierung einer *don't care*–Variablen verwendet werden. Die Obergrenze von maximal sieben Fuzzy–Mengen pro Partition wurde ausschließlich vor dem Hintergrund der sprachlichen Unterscheidbarkeit der zugehörigen linguistischen Terme gewählt und kann im Bedarfsfall erhöht werden.

Die Deklaration der entsprechenden Konstanten ist in der Datei *fuzzy.h* zu finden:

```
/* min. and max. number of fuzzy sets per partition */
#define pttMinFuzzySet  1
#define pttMaxFuzzySet  7
```

Deklaration

```
class TPartition {
 public:
    int        Count;
    TBound     Bounds;
    TFuzzySet  *FuzzySet;

    TPartition();
    TPartition (TBound ABound, int NFuzzySets);
    TPartition (istream &s);
    TPartition (const TPartition &P);
    ~TPartition();

    TPartition& operator = (const TPartition& P);

    TVector<Real> GetMembershipVector (Real x) const;
    Real MaxMembership (Real x, int& Index) const;

    void Adapt (int i,
                Real Delta_a,
                Real Delta_b,
                Real Delta_c);

    void AdaptConstrained (int i, Real x,
                           Real Delta_a,
                           Real Delta_b,
                           Real Delta_c);

    /* overloaded stream operators */
    friend ostream& operator << (ostream         &s,
                                 const TPartition &P);
};
```

Daten–elemente

Count

int **Count**

Die Variable *Count* gibt die Anzahl der Fuzzy–Mengen in der Partition an.

Bounds

TBound **Bounds**

In der Variablen *Bounds* ist die obere und untere Grenze des Wertebereichs der Partition gespeichert.

FuzzySet

TFuzzySet ***FuzzySet**

Die Variable *FuzzySet* ist ein dynamisch erzeugtes Array, das die *Count* Fuzzy–Mengen enthält.

Methoden

Konstruktor

TPartition (TBound ABound, int NFuzzySets)

Der Konstruktor erzeugt eine Fuzzy–Partition aus *NFuzzySets* Fuzzy–Mengen für das durch den Parameter *ABound* angegebene Intervall $[ABound.Min, ABound.Max]$. Dabei werden die Parameter der einzelnen Fuzzy–Mengen so gewählt, daß sich benachbarte Fuzzy–Mengen bei einem Zugehörigkeitsgrad von 0.5 schneiden. Zusätzlich ist die erste Fuzzy–Menge linksseitig und die letzte Fuzzy–Menge rechtsseitig geschultert.

Die zugehörigen linguistischen Terme der Fuzzy–Mengen werden automatisch (in Abhängigkeit von der Anzahl der Fuzzy–Mengen in der Partition) durch ein in der Datei *fuzzy.cc* deklariertes konstantes Array festgelegt.

Siehe auch: Konstruktor von *TFuzzySet*.

Stream–Konstruktor

TPartition (istream &s)

Der Stream–Konstruktor erzeugt eine Fuzzy–Partition durch Einlesen der Parameter und der die Partition bildenden Fuzzy–Mengen aus dem Stream *s*. Zu diesem Zweck liest die Methode zunächst die Anzahl der Fuzzy–Mengen und die Intervallgrenzen ein. Im Anschluß daran wird für die eingelesene Anzahl von Fuzzy–Mengen jeweils der Stream–Konstruktor der Klasse *TFuzzySet* aufgerufen. Das Datenformat für eine Fuzzy–Partition ist im Anhang in Abschnitt A.6 unter dem Schlüsselwort *FUZZY* beschrieben.

Siehe auch: Stream–Konstruktor von *TFuzzySet*.

GetMembershipVector

TVector<Real> **GetMembershipVector** (Real x) const

GetMembershipVector ermittelt den Zugehörigkeitsgrad eines gegebenen $x \in \mathbb{R}$ bezüglich aller Fuzzy–Mengen der Partition. Rückgabewert ist daher ein Vektor, dessen i–te Komponente der Zugehörigkeitsgrad $\mu_i(x)$ ist $(i = 1, \ldots, Count)$.

Bemerkung:
Die Methode *GetMembershipVector* ist laufzeitoptimiert und setzt

voraus, daß die in der Partition enthaltenen Fuzzy–Mengen ihre relative Position zueinander beibehalten (siehe auch Punkt (iii).c der Bemerkung zur Definition 3.7). Unter dieser Voraussetzung muß die Methode nicht für alle Fuzzy–Mengen in der Partition deren Methode *Membership* aufrufen, sondern nur für die Fuzzy–Mengen, deren Parameter a kleiner und deren Parameter c größer ist als x, d.h. $x \in [a, c]$. Die anderen Fuzzy–Mengen liefern für x einen Zugehörigkeitsgrad von 0.

Ist diese Randbedingung nicht gegeben, liefert diese Methode in der jetzigen Implementierung nicht für alle Fuzzy–Mengen der Partition den korrekten Zugehörigkeitswert.

Siehe auch: *TFuzzySet.Membership*.

Max–Membership

Real **MaxMembership** (Real x, int &Index) const

Die Methode *MaxMembership* ermittelt die Fuzzy–Menge μ_i in der Partition, die für den gegebenen Wert x den größten Zugehörigkeitsgrad liefert. Der Index $i \in \{1, \ldots, Count\}$ dieser Fuzzy–Menge innerhalb der Partition wird dem Aufrufer in der Variablen *Index* zurückgegeben. Der Rückgabewert der Funktion ist der entsprechende Zugehörigkeitsgrad $\mu_i(x)$.

Adapt

void **Adapt** (int i, Real Delta_a, Real Delta_b, Real Delta_c)

Adapt ermöglicht die Veränderung der Parameter der i–ten Fuzzy–Menge μ_i um die Parameter *Delta_a*, *Delta_b* und *Delta_c* (vergleiche *TFuzzySet.Adapt*). Außerdem wird gewährleistet, daß die Parameter durch die Veränderung nicht den in der Variablen *Bounds* spezifizierten Wertebereich verlassen.

Adapt–Constrained

void **AdaptConstrained** (int i, Real x,
Real Delta_a, Real Delta_b,
Real Delta_c)

AdaptConstrained ermöglicht wie *Adapt* die Veränderung der Parameter der i–ten Fuzzy–Menge μ_i um die Parameter *Delta_a*, *Delta_b* und *Delta_c*. Im Unterschied zu *Adapt* ändert diese Methode jedoch die benachbarten Fuzzy–Mengen der zu ändernden Fuzzy–Menge $\mu_i^{(j)}$ derart mit, daß sich benachbarte Fuzzy–Mengen auch nach der Änderung bei einem Zugehörigkeitsgrad von 0.5 schneiden.

Der zusätzliche Parameter x ist der Wert x_j des betrachteten Merkmals des Eingabemusters. Dieser Wert wird benötigt, um festzustellen, ob der rechte oder linke Dreiecksschenkel (d.h. $x \in [a, b]$

oder $x \in (b, c]$) der Fuzzy–Menge zu ändern ist. *AdaptConstrained* ändert stets nur die zwei betroffenen Parameter der Fuzzy–Menge.

Siehe auch: *TFuzzySet*.

TRule fuzzynet.h

Beschreibung

Die Datenstruktur *TRule* implementiert eine Fuzzy–Regel, wie sie auf Seite 17 beschrieben ist.

Deklaration

```
typedef struct TRule {
    TVector<int>   Antecedents;
    int            Node;
    Real           ConclusionWeight;
    TRule(void);
    TRule& operator = (const TRule &R);
};
/* define 'pointer to rule' */
typedef TRule *PRule;
```

Daten–elemente

Antecedents

TVector<int> **Antecedents**

In dem Vektor *Antecedents* sind die Indizes der mit den linguistischen Termen einer Regel assoziierten Fuzzy–Mengen gespeichert, d.h. für eine Regel

$$R : \textbf{If } x_1 \text{ is } A^{(1)}_{j_1} \text{ and } \ldots \text{ and } x_n \text{ is } A^{(n)}_{j_n} \textbf{ then } (x_1, x_2, \ldots, x_n) \in C_j,$$

gilt $Antecedents = (j_1, \ldots, j_n)$ mit $j_i \in \{1, \ldots, q_i\}$ $(i = 1, \ldots, n)$.

Node

int **Node**

Gibt die Ausgabeeinheit (d.h. die Klasse) an, mit der die Regel verbunden ist. Existieren m disjunkte Klassen, so gilt also $Node \in \{1, \ldots, m\}$.

Conclusion–Weight

Real **ConclusionWeight**

Der Konklusion einer Regel kann optional ein trainierbares Gewicht zugeordnet werden, das durch die Variable *ConclusionWeight* wiedergegeben wird. Durch gewichtete Konklusionen leidet jedoch die Interpretierbarkeit eines NEFCLASS–Systems stark, normalerweise gilt deshalb für alle Regeln *ConclusionWeight* = 1.0 (siehe auch Parameter *Learning_Weights* in Tabelle A.1).

TFuzzyPerceptron fuzzynet.h

TFuzzyPerceptron ist die zentrale Objektklasse in der Objekthierachie der NEFCLASS–Implementierung. Ein *TFuzzyPerceptron*–Objekt realisiert ein NEFCLASS–System gemäß Definition 3.3 und verwendet dabei alle auf den vorherigen Seiten beschriebenen Datenstrukturen.

Deklaration

```
class TFuzzyPerceptron {
 public:

  TVariableName     *InputName,
                    *ClassName;
  TVector<Real>     InputLayer,
                    RuleLayer,
                    OutputLayer;
  int               MaxRules;
  int               NumberOfRules;
  PRule             *Rules;
  TPartition        *Partition;
  TVector<int>      MinAnt;

  TAggregationFunc AggregationFunc;

  friend ostream& operator << (ostream &s,
                               const TFuzzyPerceptron &N);

  TFuzzyPerceptron (istream     &s,
                    TLoadStatus &Status,
                    ostrstream  &Msg);
```

```
 TFuzzyPerceptron (const TLearningTask &T,
                   int              MaxRulesToCreate,
                   TRuleMode        RuleSelectMode,
                   TVector<int>     FuzzySetsPerVariable,
                   TAggregationFunc AggFunction,
                   ostream          &LogStream);

 virtual ~TFuzzyPerceptron();

 void EvaluateRules (const TLearningTask &T,
                     TRuleMode          Mode,
                     ostream            &LogStream);

 TVector<Real>& Propagate (const TVector<Real> &Pattern,
                           Real MissingValue);

 Boolean CheckOutput (int WishedClass,
                      int &ComputedClass);

 Real ComputePatternError (int WishedClass);

 Real FuzzyErrorBP (int WishedClass,
                    const TLearningParameters &LParam,
                    Boolean &IsCorrect);

 int Learn (const TLearningTask       &T,
            const TLearningParameters &LParam,
            Real                      &TotalError);

 int Test (const TLearningTask &T,
           Real                &TotalError,
           ostream             &LogStream);

 void PrintPartitions (ostream    &s,
                       const char *Keyword=NULL)const;
 void PrintRules (ostream    &s,
                  const char *Keyword = NULL) const;

private:

 void CreateRuleBase (const TLearningTask &T,
                      TRuleMode RuleSelectMode,
                      int       &RulesWanted,
                      ostream   &LogStream);
```

```
  void CheckRuleConnections (const TLearningTask &T,
                             ostream &LogStream);


  void CreateVariableNames (int InputDim,
                            int OutputDim);

}; /* TFuzzyPerceptron */
```

Daten–elemente

InputName ClassName — TVariableName ***InputName, *ClassName**

Dynamische Arrays, die die Bezeichnungen für die Eingabemerkmale und die Klassen enthalten (nicht zu verwechseln mit den linguistischen Termen, die die „Namen" für die Fuzzy–Mengen darstellen). Diese Bezeichnungen können vom Anwender mittels der *VARNAMES*–Anweisung in der Musterdatei festgelegt werden, das Objekt *TFuzzyPerceptron* übernimmt diese Namen dann aus der Lernaufgabe, die dem Konstruktor übergeben wird.

Sind vom Benutzer keine Namen spezifiziert wurden, werden Standardbezeichner generiert („Input_1",„Input_2", usw. für die Eingangsgrößen sowie „Class_1", „Class_2" usw. für die Klassen). Der Datentyp *TVariableName* ist in der Datei lrntask.h definiert:

```
typedef char TVariableName[30];
```

InputLayer, RuleLayer, OutputLayer — TVector<Real> **InputLayer, RuleLayer, OutputLayer**

Diese drei Vektoren geben die Aktivierungen der Eingabe–, Regel– bzw. Ausgabeeinheiten wieder, d.h. es gilt:

$$\begin{aligned} InputLayer &= (a_{x_1}, \ldots, a_{x_n}), \\ RuleLayer &= (a_{R_1}, \ldots, a_{R_k}) \text{ und} \\ OutputLayer &= (a_{c_1}, \ldots, a_{c_m}). \end{aligned}$$

MaxRules — int **MaxRules**

Die Variable *MaxRules* gibt die vom Benutzer angegebene Maximalzahl an Regeln an, die erzeugt werden sollen. Diese Variable wird nur während des Regelerzeugungsprozesses benötigt.

NumberOf–Rules — int **NumberOfRules**

Gibt die Anzahl der Regeln in der Regelbasis an.

Rules

PRule ***Rules**

Rules ist ein dynamisch erzeugtes Array, das für jede Regel einen Zeiger auf eine Struktur vom Typ *TRule* (siehe vorangegangenen Abschnitt) enthält. Das Array enthält Zeiger und nicht direkt die Struktur *TRule*, weil die Zeiger in der Methode *EvaluateRules* einfacher „umgehängt“ werden können.

Partition

TPartition ***Partition**

Partition ist ein dynamisch erzeugtes Array, das für jedes Eingabemerkmal x_i ein Objekt vom Typ *TPartition* (siehe Seite 51) enthält.

MinAnt

TVector<int> **MinAnt**

MinAnt enthält nach der Propagation eines Musters via *Propagate* für jede Regel $R_i, i = 1, \ldots, k$ den Index der Fuzzy–Menge, die den geringsten Zugehörigkeitsgrad lieferte und damit für die Höhe der Aktivierung der Regeleinheit verantwortlich ist.

Aggregation–Func

TAggregationFunc **AggregationFunc**

Die Variable *AggregationFunc* gibt an, nach welcher Berechnungsvorschrift die Aktivierung der Ausgabeeinheiten bestimmt werden sollen (siehe die Gleichungen 3.2 und 3.3). Der Datentyp *TAggregationFunc* ist in der Datei *param.h* als Aufzählungstyp definiert:

```
typedef enum {
        aggWeightedSum,
        aggMaximum
} TAggregationFunc;
```

Methoden

Stream–Konstruktor

TFuzzyPerceptron (istream &s,
TLoadStatus &Status,
ostrstream &Msg)

Der Stream–Konstruktor dient zur Erzeugung eines *TFuzzyPerceptron*–Objekts durch Einlesen der Daten aus dem Stream s (in der Regel aus einer Datei). Das Datenformat für Netzdateien ist im Anhang in Abschnitt A.6 beschrieben.

Der Parameter *Status* gibt nach Beendigung des Konstruktors Aufschluß über den Erfolg der Operation. Der Datentyp *TLoadStatus* ist in der Datei *stdutils.h* als Aufzählungstyp definiert:

```
typedef enum {
    ldOk      =  0,
    ldWarning = -1,
    ldError   = -2
} TLoadStatus;
```

Ist *Status* ungleich *ldOk*, so befindet sich die entsprechende Fehler– bzw. Warnmeldung in der Streamvariablen *Msg*. Im Fall von *Status == ldError* ist das Objekt unbrauchbar und darf nicht weiter verwendet werden.

Konstruktor

TFuzzyPerceptron (const TLearningTask &T,
int MaxRulesToCreate,
TRuleMode RuleSelectMode,
TVector<int> FuzzySetsPerVariable,
TAggregationFunc AggFunction,
int &RulesWanted,
ostream &LogStream)

Der Konstruktor dient zur Erzeugung eines NEFCLASS–Systems, wobei die Anzahl der Ein– bzw. Ausgabeeinheiten den Dimensionen der übergebenen Lernaufgabe T entnommen wird. Dann wird zunächst für den Wertebereich jedes Eingabemerkmals eine initiale Fuzzy–Partitionierung erzeugt. Die Anzahl der Fuzzy–Mengen wird für jede Partition einzeln durch den Vektor *FuzzySetsPerVariable* bestimmt, d.h. es gilt $FuzzySetsPerVariable[i] = q_i$, $(i = 1, \ldots, T.InDimension)$. Für die q_i muß dabei $1 \leq q_i \leq 7$ gelten; die zugehörigen linguistischen Terme werden automatisch (in Abhängigkeit von der Anzahl der Fuzzy–Mengen in der Partition) durch ein in der Datei *fuzzy.cc* deklariertes konstantes Array festgelegt (sie können dann aber im Anschluß in der Netzdatei editiert werden).

Im Anschluß wird durch den Aufruf von *CreateRuleBase* eine initiale Regelbasis mit maximal *MaxRulesToCreate* Regeln erzeugt, wie es in Definition 3.5 beschrieben ist. Die sich anschließende Auswahl der Regeln (siehe Punkt (v) der Definition 3.5) wird über den Parameter *RuleSelectMode* gesteuert, dessen Datentyp als Aufzählungstyp in der Datei *param.h* deklariert ist.

```
typedef enum {
    lrSimple,
    lrBest,
    lrBestPerClass
} TRuleMode;
```

Die Funktion, die zur Berechnung der Aktivierung der Ausgabeeinheiten angewandt wird (vgl. Punkt (viii) in Definition 3.3), wird durch den Parameter *AggFunction* („Aggregierungsfunktion") bestimmt (siehe auch *TFuzzyPerceptron.AggregationFunc*).

Durch den Referenzparameter *RulesWanted* wird dem Aufrufer mitgeteilt, wie viele Regeln noch hätten erzeugt werden können, *nachdem* bereits *MaxRulesToCreate* Regeln erzeugt worden sind. Bei der Beurteilung dieses Wertes ist zu beachten, daß einige dieser zusätzlich möglichen Regeln möglicherweise doppelt gezählt wurden, da hierbei auf die Einhaltung der Bedingung (iv) in Definition 3.3 (es dürfen keine zwei Regeln mit identischer Prämisse existieren) nicht geachtet wurde.

Details über die Regelerzeugung und den Regelauswahlprozeß werden von der Konstruktor–Methode in den Stream *LogStream* geschrieben.

Siehe auch: *CreateRuleBase*, *CheckRuleConnections*, *EvaluateRules*.

Evaluate–Rules

void **EvaluateRules** (const TLearningTask &T,
TRuleMode Mode,
ostream &LogStream)

EvaluateRules bestimmt die weiter zu verwendenden Regeln aus der initialen Regelbasis. Zu diesem Zweck wird jedes Muster der Lernaufgabe T propagiert und so für jede Regel der Wert V_R aus Definition 3.5, Punkt (v), bestimmt. In Abhängigkeit vom Parameter *Mode* (Beschreibung siehe Konstruktor) werden die Regeln dann entweder nach dem Prinzip „optimales Regellernen" oder mittels „klassenoptimalem Regellernen" ausgewählt.

Siehe auch: Konstruktor.

Propagate

TVector<Real>& **Propagate** (const TVector<Real> &Pattern, Real MissingValue)

Die Methode *Propagate* propagiert das übergebene Eingabemuster *Pattern* durch das NEFCLASS–System und gibt als Rückgabewert das Ergebnis der Berechnungen, d.h. einen Vektor, der die Aktivierungen der Ausgabeeinheiten $c_1, \ldots, c_m$ enthält, zurück. Der übergebene Parameter *MissingValue* gibt den reellen Wert an, der von Propagate als „fehlende Angabe" interpretiert werden soll, falls das Muster eine Merkmalsausprägung mit diesem Wert besitzt.

Durch die Propagation werden die natürlich die Aktivierungen der Einheiten in allen drei Schichten geändert. Diese veränderten Werte bleiben nach Beendigung von *Propagate* bestehen, worauf einige andere Methoden des Objekts aufbauen.

Siehe auch: *FuzzyErrorBP, CheckOutput, Learn, Test.*

CheckOutput

Boolean **CheckOutput** (int WishedClass, int &ComputedClass)

CheckOutput bestimmt, ob ein zuvor via *Propagate* propagiertes Muster korrekt (Rückgabewert *True*) oder falsch (Rückgabewert *False*) klassifiziert wurde. In *CheckOutput* ist also die in Formel 3.1 beschriebene Winner–Take–All–Interpretation der von NEFCLASS berechneten Ausgabe gekapselt. Soll die Interpretation des Ausgabevektors einmal geändert werden, so muß nur diese Methode entsprechend modifiziert werden, da alle anderen Methoden des *TFuzzyPerceptron*–Objekts diese Methode verwenden, um das Ergebnis zu bewerten.

Ein Muster $p \in C_j$ wird z.Z. nur dann als „korrekt klassifiziert" betrachtet, wenn nach der Propagation

$$a_{c_j} = \max_{i \in \{1,\ldots,m\}} \{a_{c_i}\} \quad \text{und } \nexists_{i \neq j} \;\; a_{c_i} = a_{c_j}.$$

gilt, d.h. die Aktivierung der Ausgabeeinheit c_j muß das (eindeutige) Maximum der Aktivierungen in der Ausgabeschicht sein.

WishedClass ist dabei die Nummer der korrekten Klasse des zuvor propagierten Musters, in *ComputedClass* wird die Klasse, die durch das NEFCLASS–System bestimmt wurde, zurückgegeben.

Compute–PatternError

Real **ComputePatternError** (int WishedClass)

ComputePatternError berechnet den Fehler $E^{(p)}$, der bei der Propagation eines Musters $p \in \mathcal{L}$ gemacht wurde. Zur Bestimmung des

Fehlers werden die aktuellen Aktivierungen der Ausgabeeinheiten verwendet, d.h. das Muster muß *zuvor* via *Propagate* propagiert worden sein. Die korrekte Klasse für p wird vom Aufrufer durch den Parameter *WishedClass* angegeben.

FuzzyErrorBP

```
Real FuzzyErrorBP (int                         WishedClass,
                   const TLearningParameters   &LParam,
                   Boolean                     &IsCorrect)
```

FuzzyErrorBP (fuzzy error backpropagation) ist die Umsetzung des in Definition 3.7 beschriebenen Verfahrens zur Adaption der Fuzzy–Mengen. Direkt vor dem Aufruf von *FuzzyErrorBP* sollte ein Muster propagiert werden, da die Methode die aktuellen Aktivierungen in der Regel– und Ausgabeschicht zur Grundlage der durchzuführenden Änderungen macht.

Die zu verwendenden Lernraten werden über den Parameter *LParam* mitgeteilt (der Typ *TLearningParameters* ist in der Datei *param.h* deklariert). *WishedClass* gibt die korrekte Klasse des zuvor propagierten Musters an; durch den Referenzparameter *IsCorrect* wird dem Aufrufer mitgeteilt, ob die zuvor durchgeführte Propagation zu einer korrekten Klassifikation geführt hat oder nicht.

Siehe auch: *Propagate, CheckOutput, Learn.*

Learn

```
int Learn (const TLearningTask         &T,
           const TLearningParameters   &LParam,
           Real                        &TotalError)
```

Learn führt eine komplette Lernepoche mit der Lernaufgabe T durch, d.h. jedes Muster aus T wird einmal propagiert. Im Anschluß an jede Musterpropagation wird eine Anpassung der Fuzzy–Mengen gemäß der übergebenen Lernparameter *LParam* durch den Aufruf von *FuzzyErrorBP* vorgenommen.

Der Rückgabewert von *Learn* ist die Anzahl der fehlklassifizierten Muster, der Gesamtfehler $\sum_{p \in T} E^{(p)}$ (siehe Gleichung 3.6) wird in der Variablen *TotalError* gespeichert.

Test

```
int Test (const TLearningTask   &T,
          Real                  &TotalError,
          ostream               &LogStream)
```

Die Methode *Test* führt eine komplette Epoche mit der Lernaufgabe T durch, d.h. jedes Muster aus T wird einmal propagiert. Im

Gegensatz zur Methode *Learn* werden dabei keine Fuzzy–Mengen adaptiert, diese Methode dient zum Testen eines bereits trainierten Klassifikators.

Detaillierte Testergebnisse bezüglich jedes einzelnen Musters werden in den übergebenen *LogStream* geschrieben (also z.B. in eine Datei). Der Rückgabewert von *Test* ist die Anzahl der fehlklassifizierten Muster, der Gesamtfehler $\sum_{p \in T} E^{(p)}$ wird in der Variablen *TotalError* gespeichert.

Print–Partitions

void **PrintPartitions** (ostream &s,
const char *Keyword = NULL) const

Diese Methode schreibt die Fuzzy–Partitionen, die in dem Array *Partition* gespeichert sind, auf den Stream *s*. Das Format für Fuzzy–Partitionen ist in der Beschreibung des Formats für Netzdateien im Abschnitt A.6 unter dem Schlüsselwort *FUZZY* zu finden.

Ist die optional anzugebende Zeichenkette *Keyword* ungleich *NULL*, so wird diese zuvor (gefolgt von einem Zeilenvorschub) auf den Stream geschrieben.

PrintRules

void **PrintRules** (ostream &s,
const char *Keyword = NULL) const

Diese Methode schreibt die Fuzzy–Regeln, die in dem Array *Rules* gespeichert sind, in sprachlicher Form in den Stream *s*. Dabei werden die Bezeichnungen der Eingabemerkmale, die Klassennamen sowie die den in den Regelprämissen enthaltenen Fuzzy–Mengen zugeordneten linguistischen Terme verwendet.

Ist die optional anzugebende Zeichenkette *Keyword* ungleich *NULL*, so wird diese zuvor (gefolgt von einem Zeilenvorschub) auf den Stream geschrieben.

Siehe auch: *InputName, ClassName, TFuzzySet.Name.*

Create–RuleBase

void **CreateRuleBase** (const TLearningTask&T,
TRuleMode RuleSelectMode,
int &RulesWanted,
ostream &LogStream)

CreateRuleBase erzeugt eine initiale Regelbasis für die Lernaufgabe *T*, wobei maximal *MaxRules* erzeugt werden (die Variable *MaxRules* ist zuvor durch den Konstruktor initialisiert wurden, welcher auch die Methode *CreateRuleBase* aufruft). Nach der eigentlichen Erzeugung der Regeln werden die Regeln gemäß dem in der Variablen *RuleSelectMode* angegebenen Verfahren ausgewählt (vgl. Definition 3.5). In der Variablen *RulesWanted* wird die Anzahl der Regeln zurückgegeben, die noch hätten erzeugt werden können, nachdem bereits *MaxRules* erzeugt worden sind.

Details bezüglich des Regelerzeugungs– und Regelauswahlprozesses werden in den übergebenen *LogStream* geschrieben.

CheckRule–Connections

void **CheckRuleConnections** (const TLearningTask &T,
ostream &LogStream)

Die Methode *CheckRuleConnections* führt die in der Definition 3.5, Unterpunkt (v).b angesprochene Überprüfung durch und ändert gegebenenfalls die Konklusion einiger Regeln.

Create–Variable–Names

void **CreateVariableNames** (int InputDim, int OutputDim)

Erzeugt Default–Namen für die Eingabemerkmale und die Klassen und wird aufgerufen, sofern in der Musterdatei keine Bezeichnungen spezifiziert wurden (siehe auch *VARNAMES*–Anweisung in der Beschreibung der Dateiformate im Anhang, Abschnitt A.6). Die Default–Namen haben die Form „Input_1“,„Input_2“, ... für die Eingabemerkmale sowie „Class_1“, „Class_2“ usw. für die Klassen.

Kapitel 5

Ergebnisse ausgewählter Datenanalysen mit NEFCLASS

Das Softwaretool NEFCLASS, dessen Implementierung im vorangegangenen Kapitel diskutiert wurde, wurde anhand ausgewählter, bekannter Datensätze vom Autor getestet. Die Ergebnisse von zwei dieser Beispielklassifikationen sowie die Vorgehensweise werden im folgenden vorgestellt. Weitere Beispiele für Datenanalysen, die mit dem vom Autor erstellten Tool NEFCLASS erzielt wurden, sind in einer Arbeit von Matthias Kanne zu finden [Kanne, 1996]. M. Kanne nimmt in dieser Arbeit eine vergleichende Betrachtung verschiedener Klassifikationsansätze vor und berücksichtigt dabei auch detailliert das NEFCLASS–Modell sowie das zugehörige Softwaretool.

Ausführliche Benutzeranleitungen für NEFCLASS und NEFCLASS–X sind in den Anhängen A und B zu finden. An dieser Stelle soll nur kurz der grundsätzliche Umgang mit dem Softwaretool beschrieben werden, bevor auf die Testergebnisse eingegangen wird.

Die zu verwendende Lernaufgabe muß in einer Datei vorliegen, deren Dateiformat im Anhang A.6 beschrieben ist. NEFCLASS verwendet für alle Dateitypen (Netz–, Daten–, Regel–, Error–, Logdatei) Textdateien, die mit jedem normalen Texteditor eingesehen bzw. editiert werden können. In der Datendatei (d.h. in der Datei, die die Lernaufgabe enthält) können den Eingangsmerkmalen und den Klassen Namen zugeordnet werden; diese werden dann beim Erzeugen des zugehörigen NEFCLASS–Systems in die Netzdatei übernommen.

Liegt die gewünschte Lernaufgabe in der oben beschriebenen Form vor, so muß sie zunächst eingelesen werden, wodurch auch die Ein– und Ausgabedimension des zu erstellenden NEFCLASS–Systems festgelegt werden (Anzahl Eingangsmerkmale = Anzahl Eingabeeinheiten und Anzahl Klassen = Anzahl Ausgabeeinheiten). Bevor nun ein für die Muster der Lernaufgabe geeignetes NEFCLASS–System erstellt werden kann, muß der Benutzer noch wählen, durch wieviele Fuzzy–Mengen der Wertebereich jeder Eingangsgröße partitioniert werden soll. Dieser Wert q_i kann sowohl für jede Eingangsgröße einzeln, als auch für alle Eingangsgrößen gleich ($q = q_1 = \cdots = q_n$) angegeben werden.

Für diese Werte muß $q_{min} = 1 \leq q_i \leq 7 = q_{max}$ für alle i gelten, wobei eine Partitionierung eines Wertebereichs mit nur einer Fuzzy–Menge einer *don't care*–Variablen entspricht, da eine solche Fuzzy–Menge beidseitig geschultert ist (graphisch eine Parallele zur x–Achse) und für jeden x–Wert des Wertebereichs einen Zugehörigkeitsgrad von 1 liefert. Die Obergrenze von maximal $q_{max} = 7$ Fuzzy–Mengen pro Partition wurde im Hinblick auf die Interpretierbarkeit und die linguistische Differenzierbarkeit der Fuzzy–Mengen gewählt. Liegen im Vorfeld der Klassifikation keinerlei Kenntnisse über die Verteilung der einzelnen Eingangsmerkmale vor, so sollten alle Fuzzy–Partitionen aus einem mittleren Wert (etwa $q = 3$) an Fuzzy–Mengen gebildet werden.

Die mit den Fuzzy–Mengen linguistische Terme wie *small, large* usw. werden durch NEFCLASS automatisch in Abhängigkeit von der Gesamtanzahl erzeugt, sie können später jedoch in der Netzdatei nach Belieben editiert werden. Weiterhin kann der Benutzer bei der Erzeugung des NEFCLASS–Systems bestimmen, wieviele Regeln *höchstens* erzeugt werden sollen. Unterbleibt diese Angabe, werden von NEFCLASS soviele Regeln wie möglich nach dem in Definition 3.5 beschriebenen Algorithmus erzeugt.

Verfügt der Anwender über a–priori–Regelwissen über die Klassifikationsaufgabe, so kann er diese Regeln in die Netzdatei eintragen. Solche vom Benutzer vorgegebenen Regeln müssen — wie wir schon im Beispiel 3.4 in Abschnitt 3.3 gesehen haben — nicht alle Eingangsgrößen in der Regelprämisse verwenden. Die vom System erzeugten Regeln verwenden jedoch grundsätzlich alle Eingangsvariablen.

Nachdem das NEFCLASS–System erzeugt worden ist, kann der Lernprozeß beginnen, in dem das System durch Adaption der Fuzzy–Mengen versucht, den Gesamtfehler zu verringern. Der Lernvorgang kann dabei über eine Vielzahl von Parametern beeinflußt werden, die alle in Anhang A beschrieben sind. Als Abbruchkriterien für den Lernvorgang können eine Maximalanzahl zu durchlaufender Epochen oder eine erreichte Anzahl zulässiger Fehlklassifikationen (absolut oder prozentual) spezifiziert werden.

NEFCLASS legt bei der Erzeugung jedes neuen Systems (optional) eine sogenannte *Log–Datei* an, die detaillierte Informationen über den Regelerzeugungsprozeß enthält. Die dort zu findende Bewertung der einzelnen Regeln („rule performance“) stellt ein wesentliches Hilfsmittel bei der Wahl einer möglichst geringen Anzahl von Regeln dar.

Im den folgenden beiden Abschnitten soll die Vorgehensweise der Datenklassifikation mit dem NEFCLASS–Tool anhand von zwei bekannten Datensätzen beispielhaft durchgeführt werden. Die Kommandodateien (mit allen wichtigen Parametern) für diese beiden Beispiele sind der für die Installation des NEFCLASS–Tools erforderlichen Archivdatei beigelegt, so daß sie vom interessierten Leser nachvollzogen werden können. Die Bezugsquelle dieser Datei ist in Anhang A angegeben.

5.1 Iris–Datensatz

Der Iris–Datensatz ist ein Standard–Datensatz im Bereich der Musterklassifikation. Er wurde bereits 1936 von R.A. Fisher aufgenommen [Fisher, 1936] und enthält 150 Muster, die Merkmale von Irisblumen beschreiben. Jedes der Muster enthält vier Merkmale (jeweils die Breite und Länge eines Blumen– und eines Kelchblattes), anhand derer die Muster den drei disjunkten Klassen *Iris Setosa*, *Iris Versicolour* und *Iris Virginica* zugeordnet werden können.

Der Iris–Datensatz ist recht einfach zu klassifizieren, da die Klasse *Setosa* linear separabel zu den beiden anderen Klassen ist, während sich *Versicolour* und *Virginica* ein wenig überlappen. Viele der gängigen Klassifikationsverfahren (statistische Ansätze, Neuronale Netze etc.) liefern daher gute Resultate bei der Klassifikation der Iris–Daten, in der Regel werden etwa fünf Muster fehlklassifiziert (bezogen auf den gesamten Datensatz von 150 Mustern) [Nauck et al, 1996].

Um die Iris–Daten mit Hilfe von NEFCLASS zu klassifizieren, wird der Datensatz zunächst in zwei gleichgroße Teildatensätze (jeweils 75 Muster) aufgeteilt, wobei darauf geachtet wurde, daß der relative Anteil jeder Klasse bezogen auf den Gesamtdatensatz von 150 Mustern in den beiden Teildatensätzen erhalten bleibt. Im Fall des Iris–Datensatzes geht dies sogar exakt: Der vollständige Datensatz enthält 50 Muster pro Klasse, so daß nach der Aufteilung beide Teilmengen genau 25 Muster pro Klasse enthalten. Einen der Teildatensätze verwenden wir als *Trainingsdatensatz*, den anderen entsprechend zum *Testen* des trainierten NEFCLASS–Systems.

Wir gehen davon aus, daß wir keine Kenntnisse über den Iris–Datensatz besitzen und lassen die Wertebereiche aller vier Eingangsmerkmale durch jeweils $q = 3$ Fuzzy–Mengen partitionieren. Weiterhin soll NEFCLASS nach dem Verfahren des „optimalen“ Regellernens (siehe Definition 3.5) zunächst soviele Regeln wie möglich erzeugen.

NEFCLASS bestimmt daraufhin 19 Regeln (potentiell möglich wären bei 75 Mustern mit jeweils 4 Merkmalen und 3 Fuzzy–Mengen pro Partition $\min\{75, 3^4\} = 75$ Regeln). An die Erzeugung der Regeln schließen wir sofort das Erlernen der Fuzzy–Mengen an. Gleich zu Beginn des Lernvorgangs werden nur 2 Muster fehlklassifiziert. Das Training wird nach 140 Epochen abgebrochen, da sich der Gesamtfehler, d.h. die Summe der quadrierten Abweichungen zwischen korrekter und tatsächlicher Ausgabe, seit etwa 50 Epochen nicht mehr wesentlich verändert hat. Am Ende des Lernens macht das NEFCLASS–System auf den Trainingsdaten jetzt 3 Fehler, von den Testdaten werden nur 2 Muster fehlklassifiziert. Das NEFCLASS–System liefert also auf Anhieb ein sehr gutes Ergebnis, nämlich 5 Fehler bezogen auf den gesamten Datensatz von 150 Mustern, d.h. 96,7% der Muster werden korrekt klassifiziert.

Unsere weiteren Bemühungen gelten nun dem Versuch, den erhaltenen Klassifikator durch Reduktion der Regelanzahl zu vereinfachen, um seine Interpretierbarkeit noch weiter zu verbessern.

Zunächst können wir der Log–Datei entnehmen, daß in obigem Beispiel die Performance der Regeln nach den besten fünf Regeln deutlich abfällt. Unter diesen Regeln ist jeweils eine Regel für die Klasse *Iris Setosa* und *Iris Versicolour*, sowie drei Regeln für die Klasse *Iris Virginica*. Wir wiederholen daher den Versuch, erlauben dem Programm jedoch diesmal nur drei Regeln nach dem „klassenoptimalen“ Verfahren zu erzeugen, d.h. wir wollen sehen, wie gut das System mit nur einer Regel pro Klasse noch klassifizieren kann.

Diesmal macht das System am Anfang 12 Fehler, steigert sich aber während des Lernvorgangs deutlich und macht bereits nach 30 Epochen nur noch 3 Fehler. Nach 235 Epochen wird das Training abgebrochen, da der Gesamtfehler nicht mehr weiter abnimmt. Der Fehlerverlauf (Gesamtfehler und fehlklassifizierte Muster) während des Trainings ist in Abbildung 5.1 zu sehen.

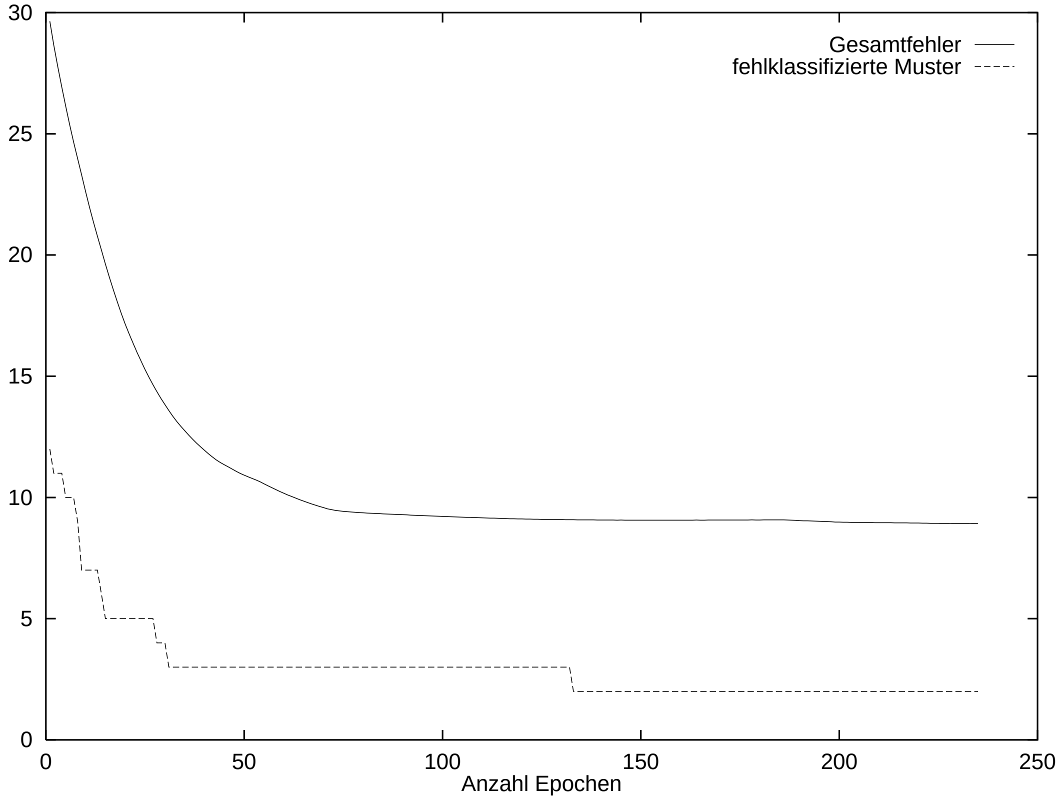

Abbildung 5.1: Die Entwicklung des Gesamtfehlers und der Anzahl fehlklassifizierter Muster während der Lernphase für den Iris-Datensatz.

Nach Abschluß der Lernphase werden durch das System zwei Muster der Trainingsdaten und zwei Muster der Testdaten fehlklassifiziert, d.h. insgesamt erhalten wir also nur vier Fehler bezogen auf den gesamten Datensatz. Der Klassifikator besteht dabei nur noch aus drei Regeln und ist damit erheblich kompakter.

Die drei Regeln sind in der Tabelle 5.1 angegeben. In der Tabelle sind die Regelprämissen in einer verkürzten Schreibweise angegeben, dabei bedeutet *sm* = *small*, *md* = *medium* und *lg* = *large*.

if x_1 is *sm*	and	x_2 is *md*	and	x_3 is *sm*	and	x_4 is *sm*	**then**	*Setosa*
if x_1 is *md*	and	x_2 is *sm*	and	x_3 is *md*	and	x_4 is *md*	**then**	*Versicolour*
if x_1 is *lg*	and	x_2 is *sm*	and	x_3 is *lg*	and	x_4 is *lg*	**then**	*Virginica*

Tabelle 5.1: Die von NEFCLASS bestimmte Regelbasis zur Klassifikation der Irisdaten (*sm* = *small*, *md* = *medium*, *lg* = *large*)

Die erlernten Fuzzy–Mengen, die die linguistischen Terme *small*, *medium* und *large* bezüglich der Eingangsmerkmale $x_1, \ldots, x_4$ repräsentieren, sind in der Abbildung 5.2 dargestellt.

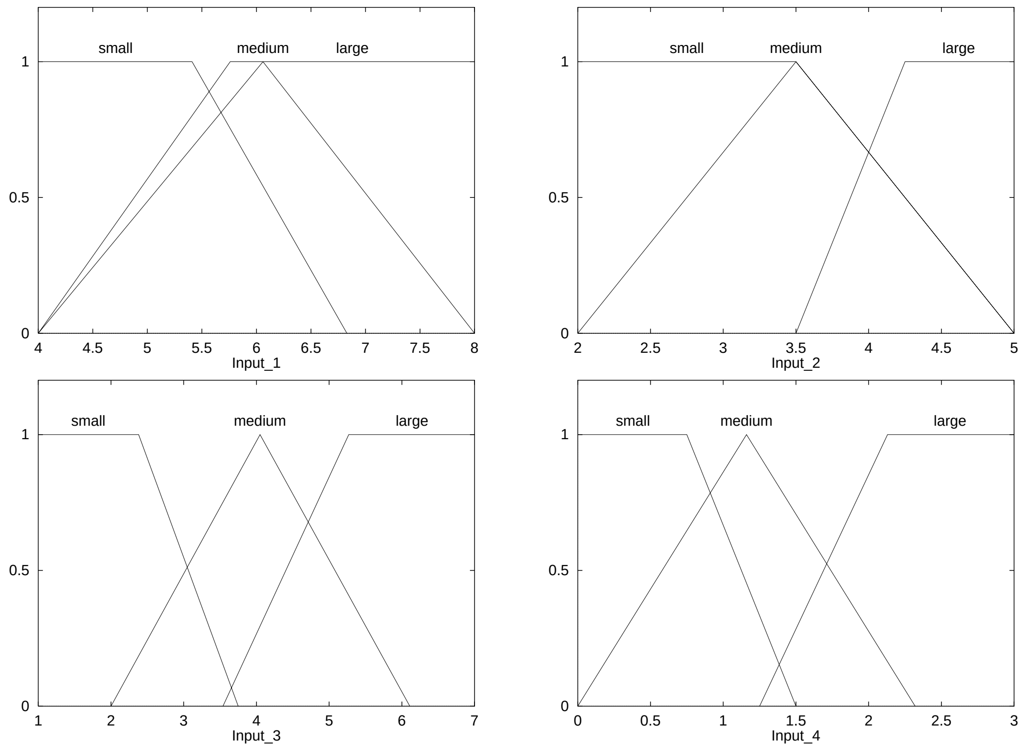

Abbildung 5.2: Die erlernte Partitionierung der Eingangsgrößen $x_1, \ldots, x_4$ für den Iris-Datensatz bei einer aus drei Regeln bestehenden Regelbasis und $q = 3$ Fuzzy–Mengen pro Eingabegröße

Die Partitionierung der Eingangsgrößen x_1 und x_2 kann sicherlich noch verbessert werden. Wie man in der Abbildung 5.2 erkennen kann, wird die Fuzzy–Menge *medium* in der Partitionierung von x_1 vollständig durch die Fuzzy–Menge *large* überdeckt. Ebenso wird in der Fuzzy–Partition der Eingangsgröße x_2 die Fuzzy–Menge *medium* durch die Fuzzy–Menge *small* überdeckt, darüberhinaus wird die Fuzzy–Menge *large* dieser Eingangsgröße von keiner Regel benutzt (siehe Tabelle 5.1). Dies läßt darauf schließen, daß für x_1 höchstens zwei Fuzzy–Mengen benötigt werden und x_2 ganz unberücksichtigt bleiben kann.

In einem abschließenden Versuch spezifizieren wir daher gleich beide Eingangsgrößen x_1 und x_2 als don't–care–Variablen, d.h. wir setzen $q_1 = q_2 = 1$. Wieder erlauben wir dem System die Erzeugung von drei Regeln nach dem „klassenoptimalen"Verfahren.

Tatsächlich verringert sich die Leistung des Klassifikators nur unwesentlich durch das Weglassen dieser beiden Variablen. Nach einer Lernphase von 300 Epochen macht das System auf den Trainingdaten zwei und auf den Testdaten drei Fehler, insgesamt also nur fünf Fehler (gegenüber vier Fehlern bei der Berücksichtigung aller vier Eingangsgrößen).

Die in dieser Konstellation erlernten Regeln sind in der Tabelle 5.1 aufgeführt, wobei die don't–care–Variablen x_1 und x_2 aus Gründen der Übersichtlichkeit weggelassen wurden.

if x_3 is *small*	and x_4 is *small*	**then** *Setosa*
if x_3 is *medium*	and x_4 is *medium*	**then** *Versicolour*
if x_3 is *large*	and x_4 is *large*	**then** *Virginica*

Tabelle 5.2: Die Regelbasis zur Klassifikation der Irisdaten bei ausschließlicher Berücksichtigung der Eingangsgrößen x_3 und x_4

5.2 Breast–Cancer–Datensatz

Als zweites Beispiel wird ein komplexerer Datensatz verwendet, der „Wisconsin–Breast–Cancer"–Datensatz, der von Dr. William H. Wolberg an den University of Wisconsin Hospitals in Madison erhoben wurde [Wolberg, 1990]. Dieser Datensatz enthält 699 Musterdaten von Patientinnen, die bezüglich eines eventuell vorhandenen Brustkrebses untersucht wurden. Die Muster enthalten dabei je 9 diskrete Merkmale von 1–10. Die Daten verteilen sich auf die beiden Klassen *gutartig* und *bösartig*, wobei 458 Fälle gutartig und 241 Fälle bösartig sind.

Der Breast–Cancer–Datensatz ist ebenfalls schon häufig untersucht wurden, so wurde dieser Datensatz z.B. von H. Wolberg und X. Mangasarian unter Benutzung von drei Hyperebenenpaaren mit einer Genauigkeit von 95,9% klassifiziert [Mangasarian et al, 1990]. In [Nauck et al, 1996] geben die Autoren als Ergebnis einer Diskriminanzanalyse mit dem bekannten Statistikprogramm SPSS eine Fehleranzahl von 27 (d.h. eine Klassifikationsleistung von 96,05%) bezogen auf den gesamten Datensatz an.

Für die Verarbeitung mit NEFCLASS wurden aus dem ursprünglichen Datensatz 16 Muster entfernt, da sie fehlende Angaben beeinhalten. Zur Verarbeitung gelangen also 683 Muster, die wie schon beim Iris–Datensatz in einen Trainingsdatensatz (342 Muster) und einen Testdatensatz (341 Muster) aufgeteilt werden.

Im ersten Versuch lassen wir NEFCLASS wie schon beim Iris–Datensatz nach dem Verfahren des „optimalen" Regellernens soviele Regeln wie möglich erzeugen, wobei wir die Wertebereiche aller neun Eingangsmerkmale durch jeweils $q = 2$ Fuzzy–Mengen partitionieren.

NEFCLASS bestimmt daraufhin 85 Regeln (potentiell möglich wären $\min\{342, 2^9\} =$ 342 Regeln). An die Erzeugung der Regeln schließen wir wieder sofort das Erlernen der Fuzzy–Mengen an. Am Anfang des Lernvorgangs werden nur 10 Muster fehlklassifiziert, diese Fehlerzahl verschlechtert sich jedoch durch das Training auf 40 Fehler nach 10 Epochen (obwohl der Gesamtfehler weiter fällt). Wir brechen den Lernvorgang daher an dieser Stelle ab.

Die Regelperformance stellt sich diesmal wie folgt dar: Die beiden (mit Abstand) besten Regeln klassifizieren Muster für die Klasse *gutartig*, danach folgen fast nur noch Regeln für die Klasse *bösartig*, deren Performance jedoch nicht sehr stark unterschiedlich ist (unter den 85 Regeln gibt es insgesamt nur sieben für die Klasse *gutartig*).

Um herauszufinden, wieviele Regeln für ein akzeptables Ergebnis notwendig sind, erlauben wir dem Programm im folgenden Versuch nur eine Regel pro Klasse, um zu sehen, wie gut die Klassifikationsleistung mit nur zwei Regeln ist. Zu Beginn des Trainings werden zunächst 95 Muster fehlklassifiziert, nach nur 30 Epochen sind es jedoch nur noch zehn Fehler. Nach 500 Epochen brechen wir den Lernvorgang ab. Das System macht dann auf den Trainingsdaten nur noch fünf Fehler, von den Testdaten werden 17 Muster fehlklassifiziert. Insgesamt erreichen wir also eine Anzahl von 22 fehlklassifizierten Mustern bezogen auf den gesamten Datensatz von 683 Mustern (d.h. 96,8% der Eingabemuster werden korrekt klassifiziert).

Die beiden Klassifikationsregeln sind in Tabelle 5.2 angegeben.

R_1: **if**	(lg, lg, lg, lg, lg, lg, lg, lg, sm)	**then** *bösartig*,
R_2: **if**	(sm, sm, sm, sm, sm, sm, sm, sm, sm)	**then** *gutartig*.

Tabelle 5.3: Die von NEFCLASS bestimmte Regelbasis zur Klassifikation des Breast–Cancer–Datensatzes (*sm* = *small*, *lg* = *large*)

Wie man leicht erkennt, haben sowohl R_1 als auch R_2 für das Merkmal x_9 den linguistischen Term *small* in ihrer Prämisse. Das Merkmal x_9 ist somit für die Klassifikation (unter Benutzung der o.g. Regelbasis) nicht relevant und kann zur don't–care–Variablen erklärt werden. Die Partitionierung der Wertebereiche der Merkmale $x_1, \ldots, x_8$ ist in der Abbildung 5.3 auf Seite 74 dargestellt.

Aus der Abbildung kann man entnehmen, daß es bei den meisten der Partitionen Bereiche gibt, in denen sowohl die jeweilige Fuzzy–Menge *small* als auch die Fuzzy–Menge *large* den Zugehörigkeitsgrad 1 liefern. Solche Partitionen sind jedoch häufig unerwünscht.

Wir machen daher einen letzten Versuch, in dem wir das Programm anweisen, die Fuzzy–Mengen so zu verändern, daß sich benachbarte Fuzzy–Mengen bei einem Zugehörigkeitsgrad von 0.5 schneiden. Die Regelbasis bleibt dabei unverändert. In dieser Konstellation können wir den Lernvorgang bereits nach 250 Epochen beenden, da sich der Gesamtfehler seit 50 Epochen nicht mehr wesentlich verändert hat. Die Klassifikationsleistung ist durch die Restriktion bezüglich der Fuzzy–Mengen auf 95.3% gesunken (10 Fehler in den Trainings– und 22 Fehler in den Testdaten). Dennoch ist die Leistung sehr zufriedenstellend, zumal das System nur zwei Regeln benötigt und dadurch (und durch die Gestalt der Fuzzy–Partitionen) sehr gut interpretiert werden kann. Die bei diesem Lernvorgang erlernten Fuzzy–Mengen zeigt Abbildung 5.4 auf Seite 75.

In diesem Kapitel wurde gezeigt, wie sich das NEFCLASS–Modell im Aufgabengebiet der Musterklassifikation einsetzen läßt. Die Leistung des NEFCLASS–Modells ist dabei bei den hier vorgeführten Beispielen durchaus vergleichbar mit den Leistungen anderen in diesem Bereich einsetzbarer Verfahren wie z.B. Neuronale Netze oder statistische Verfahren. Der große Vorteil des NEFCLASS–Modells gegenüber diesen anderen Verfahren liegt dabei zum einen in der guten Interpretierbarkeit des erhaltenen Klassifikators und zum anderen in der Möglichkeit, den Klassifikator mit Vorwissen zu initialisieren.

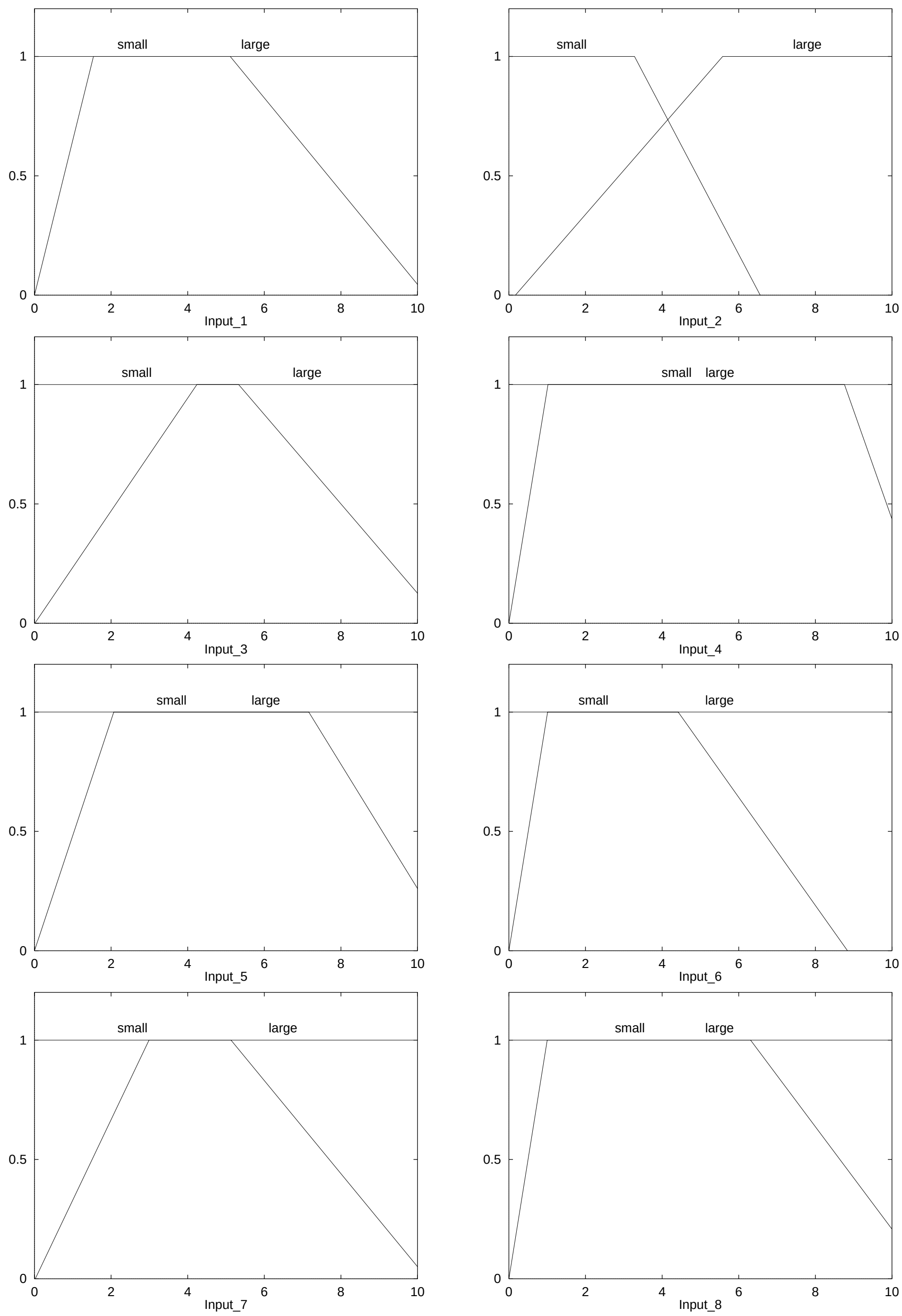

Abbildung 5.3: Partitionierung der Eingangsgrößen $x_1, \ldots, x_8$ des Breast–Cancer–Datensatzes nach dem Lernvorgang ohne Restriktionen bezüglich der Änderung der Fuzzy–Mengen

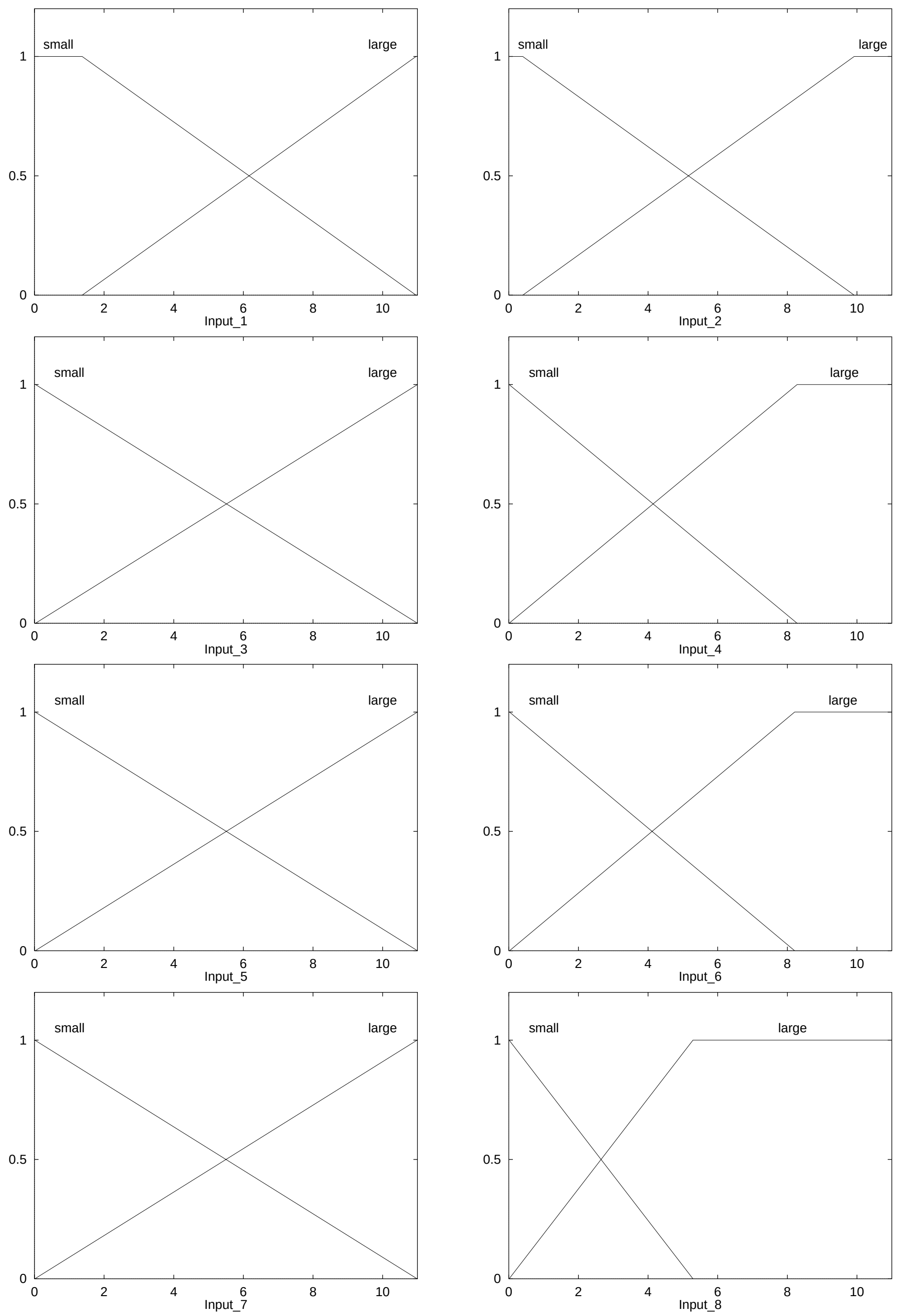

Abbildung 5.4: Partitionierung der Eingangsgrößen $x_1, \ldots, x_8$ des Breast–Cancer–Datensatzes nach dem Lernvorgang, wenn sich benachbarte Fuzzy–Mengen bei einem Zugehörigkeitsgrad von 0.5 schneiden müssen

Kapitel 6

Zusammenfassung

In der vorliegenden Arbeit wurde das hybride Neuro–Fuzzy–Modell NEFCLASS vorgestellt, das im Bereich der Musterklassifikation eingesetzt werden kann. Es basiert auf dem generischen Modell eines dreischichtigen Fuzzy–Perceptrons, wodurch das Modell in Form linguistischer Regeln interpretiert werden kann. Durch den Einsatz gekoppelter Verbindungen werden semantische Probleme bezüglich der repräsentierten Fuzzy–Regeln verhindert, da eine multiple Repräsentation identischer linguistischer Terme vermieden wird.

Der Lernalgorithmus des NEFCLASS–Modells ist in der Lage, sowohl die Fuzzy–Regeln als auch die Parameter der den Regeln zugrundeliegenden Fuzzy–Mengen zu erlernen. Die Bildung der Regelbasis kann durch die Verwendung einer festen Lernaufgabe inkrementell erfolgen und verursacht deshalb nur geringen Aufwand. Das Erzeugen neuer Regeln, deren Bewertung und die Auswahl einer Regelbasis sind — je nach Wahl der verwendeten Variante zur Regelerzeugung — nach spätestens drei Epochen beendet. Allerdings ist das hier vorgestellte NEFCLASS–Modell nur in der Lage, einfache Fuzzy–Regelsysteme zu erzeugen. Insbesondere verwendet jede der durch das System gebildeten Regeln grundsätzlich alle Eingangsgrößen in ihrer Prämisse. Für eine einfache Interpretierbarkeit des Systems ist es wünschenswert, Regeln mit möglichst wenigen Eingangsgrößen in den Prämissen zu finden. Im Kapitel 2 wurde im Rahmen eines ausführlichen Beispiels einer NEFCLASS–Anwendung gezeigt, daß Regeln, die don't–care–Variablen in ihrer Prämisse verwenden, sehr effizient eingesetzt werden können. Allerdings müssen derartige Regeln durch den Anwender vorgegeben werden.

Das überwachte Lernverfahren zur Änderung der verwendeten Fuzzy–Mengen setzt ein Fuzzy–Fehlermaß ein und ist eine einfache Heuristik. Das Erlernen der Parameter der Fuzzy–Mengen während der Trainingsphase kann leicht Randbedingungen unterworfen werden, so daß Modifikationen der Parameter nur dann vorgenommen werden, wenn diese Randbedingungen dabei eingehalten werden. Im Hinblick auf eine spätere Interpretation des Lernergebnisses ist man daran interessiert, Fuzzy–Partitionen zu erhalten, die sich möglichst leicht mit linguistischen Termen belegen lassen. Eine sinnvolle Randbedingung besteht daher in der Beibehaltung der relativen Positionen der Fuzzy–Mengen einer Variable zueinander (vgl. hierzu auch die Bemerkung zu Definition 3.7). Ebenso kann es sinnvoll sein, den Grad der Überschneidung zweier benachbarter Fuzzy–Mengen festzulegen. Bei der Festlegung solcher Randbedingungen muß man jedoch beachten, daß derartige Randbedingungen die Freiheitsgrade des Lernalgorithmus einschränken und damit die Bestimmung einer guten Lösung erschweren können.

Aufgrund des in Abschnitt 3.3 ausführlich diskutierten Propagationsverfahrens kann ein NEFCLASS–System keine Ausgaben von exakt 0 bzw. 1 erzielen, sondern nur Werte aus dem Intervall $[0,1]$. Exakte Ausgaben erhält man erst durch die Interpretation der Ausgaben; beim NEFCLASS–Modell findet an dieser Stelle eine Winner–Take–All–Interpretation Anwendung.

Um direkt exakte Ausgaben zu erreichen, müßten die Regeln mit beliebigen reellen Faktoren gewichtet werden. Die Verwendung gewichteter Regeln beeinträchtigt jedoch die Semantik des Modells erheblich.

Würde man als Gewichte reelle Zahlen aus dem Einheitsintervall $[0,1]$ verwenden, so ließen sich diese ggf. als eine Art „Vertrauensgrad“ (in die Regel) interpretieren. Um exakte Ausgaben zu erhalten, müssen diese Gewichte jedoch normalerweise deutlich größer als 1 sein. Eine sinnvolle Interpretation des erhaltenen Klassifikators ist dann jedoch nicht mehr möglich. Das im Rahmen dieser Arbeit entwickelte Softwaretool NEFCLASS bietet dennoch alle drei Möglichkeiten:

- Verzicht auf Regelgewichte,
- Regelgewichte aus $[0,1]$,
- Regelgewichte aus $\mathbb{R}$.

Im Normalfall, d.h. wenn der erhaltene Klassifikator in Form linguistischer Wenn–Dann–Regeln interpretiert werden soll, sollte man auf die Verwendung trainierbarer Gewichte zwischen Regel– und Ausgabeschicht gänzlich verzichten.

In Kapitel 5 wurde die Datenanalyse mit dem Softwaretool NEFCLASS anhand zweier ausgewählter Datensätze beispielhaft durchgeführt. Dabei wurde aufgezeigt, daß die erzielten Ergebnisse durchaus vergleichbar sind mit den Ergebnissen, die z.B. Neuronale Netze oder statistisches Verfahren bei der Musterklassifikation liefern.

Der Vorteil eines Neuro–Fuzzy–Klassifikationsverfahrens wie NEFCLASS gegenüber diesen anderen Verfahren besteht vor allem darin, den erhaltenen Klassifikator auf der Grundlage linguistischer Regeln interpretieren und — daraus resultierend — mit a–priori–Wissen über die Lernaufgabe initialisieren zu können.

Diese beiden Punkte sind neben der Einfachheit des NEFCLASS–Ansatzes von nicht zu unterschätzender Bedeutung in den Bereichen, in denen menschliche Entscheidungsträger Systeme zur Entscheidungsfindung und –unterstützung benötigen. Beispiele für mögliche Anwendungen solcher Systeme sind bereits in Kapitel 2 genannt worden. Derartige Systeme werden von Anwendern eher akzeptiert, wenn die Entscheidungen des Systems nachvollzogen werden können.

Das hier vorgestellte Softwaretool NEFCLASS erlaubt es dem Anwender, auf einfache Weise einen Klassifikator aus Beispieldaten aufzubauen und eventuell vorhandenes Vorwissen über die Lernaufgabe in das System zu integrieren.

Der erzeugte Klassifikator ist dabei aufgrund der semantischen Eigenschaften des NEFCLASS–Modells vollständig transparent. Das Softwaretool kann daher als interaktives Werkzeug zur Datenanalyse verstanden werden. Ausgehend von der durch das System erzeugten initialen Regelbasis wird die Datenanalyse in mehreren Schritten durchgeführt, wobei der Anwender durch z.B. die Elimination einzelner Variablen, individuelle Änderungen der Fuzzy–Partitionen, Reduktion der Regelanzahl usw. den Lernvorgang sehr effektiv beeinflussen kann. Dieses schrittweise Vorgehen wurde in Kapitel 5 exemplarisch demonstriert.

Literaturverzeichnis

[Anderson et al, 1988] James A. Anderson, Edward Rosenfeld. *Neurocomputing. Foundations of Research.* MIT Press, Cambridge, 1988.

[Fisher, 1936] R.A. Fisher. *The use of multiple measurements in taxonomic problems.* Annual Eugenics, 7, Part II, 179-188, 1936.

[Kanne, 1996] Matthias Kanne. *Neuro–Fuzzy–Methoden in der Datenanalyse.* Diplomarbeit, TU Braunschweig, 1996.

[Kruse et al, 1993] Rudolf Kruse, Jörg Gebhardt, Frank Klawonn. *Fuzzy–Systeme.* Teubner–Verlag, Stuttgart, 1993.

[Lipman, 1990] Stanley B. Lipman. *C++ Einführung und Leitfaden.* Addison–Wesley, Bonn, 1990.

[Mamdani, 1974] E.H. Mamdani. *Applications of Fuzzy Algorithms for simple Dynamic Plant.* In: Proc. IEEE 121, pp. 1585–1588, 1974.

[Mamdani et al, 1975] E.H. Mamdani, S. Assilian. *An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller.* In: International Journal of Man Machine Studies, 7:1–13, 1975.

[Mangasarian et al, 1990] O. L. Mangasarian, W. H. Wolberg. *Cancer diagnosis via linear programming.*, SIAM News, Volume 23, Number 5, September 1990.

[Minsky und Papert, 1969] L. Minsky, S. Papert. *Perceptrons.* MIT Press, Cambridge, 1969.

[Nauck et al, 1996] Detlef Nauck, Frank Klawonn, Rudolf Kruse. *Neuronale Netze und Fuzzy–Systeme.* Vieweg–Verlag, 2. Auflage. Braunschweig, 1996.

[NEFCLASS-PC] Detlef Nauck, Ulrike Nauck. *NEFCLASS–PC.* Eine Implementierung des NEFCLASS–Modells für MS–DOS–PC's. TU Braunschweig, erscheint demnächst.

[Ousterhout, 1994] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison–Wesley, 1994.

[Rosenblatt, 1958] F. Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.* Psychological Review, 65:386–408, 1958.

[Rumelhart et al, 1986] D.E. Rumelhart, G.E. Hinton, R.J. Williams. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition. Foundations*, Band 1. MIT Press, Cambridge, 1986.

[Stroustrup, 1995] Bjarne Stroustrup. *Die C++–Programmiersprache: Erweitert um Entwürfe zur ANSI–/ISO–Standardisierung.* 2., überarbeitete Auflage. Addison–Wesley, Bonn, 1995.

[Sugeno, 1985] M. Sugeno. *Industrial Applications of Fuzzy Control.* North–Holland, Amsterdam, 1985.

[Wolberg, 1990] William H. Wolberg, O.L. Mangasarian. *Multisurface method of pattern separation for medical diagnosis applied to breast cytology.* Proceedings of the National Academy of Sciences, U.S.A., Volume 87, December 1990, pp 9193-9196.

[Zadeh, 1965] L.A. Zadeh. *Fuzzy Sets.* Information and Control, 8:338–353, 1965.

[Zadeh, 1972b] L.A. Zadeh. *A Rational for Fuzzy Control.* In: Dynamic Systems, Measurements and Control, Series 6, 94:3–4, 1972.

Appendix A

NEFCLASS User Manual

NEFCLASS (**NE**uro **F**uzzy **CLASS**ification) is a command–driven interactive simulation program to develop, train and test neuro–fuzzy networks for data classification.

This document provides an overview of the NEFCLASS neural network simulator. First it explains how to install and configure the simulator. After that all the NEFCLASS commands are explained. The last part describes the file format of the files that are used by NEFCLASS.

A.1 Introduction

NEFCLASS is a neuro-fuzzy model based on a generic 3-layer fuzzy perceptron, and it is used for classifying data (patterns).

A NEFCLASS system is trained with a set of example patterns, where each pattern belongs to one of a number of distinct classes (crisp classification). A single input pattern is a vector of floating point numbers.

NEFCLASS finds fuzzy rules by scanning the data, and later optimizes these rules by learning the parameters of the fuzzy sets that are used to partition the domain of the input variables (features of the patterns).

After the learning process the resulting NEFCLASS system can be used for classifying new, previously unknown data. One advantage of using NEFCLASS for data classification is that the system can be interpreted in form of fuzzy rules like

$$R_r : \textbf{If } x_1 \text{ is } A^{(1)}_{j_1} \text{ and } \ldots \text{ and } x_n \text{ is } A^{(n)}_{j_n} \textbf{ then } (x_1, x_2, \ldots, x_n) \in C_j,$$

where the $A^{(i)}_{j_i}$ are linguistic terms (like *small*, *large* etc.) represented by fuzzy sets. $x_1, \ldots, x_n$ are the input features of the pattern $\mathbf{x}$. C_j is a (crisp) subset of $\mathbb{R}^n$ that represents the class j.

To obtain more information about the NEFCLASS model please read the postscript file nefclass.ps that is included in the distribution.

The new NEFCLASS user should begin by reading about 'commands' (type '`help commands`').

A.2 Installing NEFCLASS

To install NEFCLASS on your system, you need the file nfcXXsrc.zip which contains the whole source code of the NEFCLASS project ('XX' is the version number; the current version of nefclass is 1.0, so the right name for the archive is 'nfc10src.zip'). This archive can be obtained from

Institut fuer Betriebssysteme und Rechnerverbund
Technische Universitaet Braunschweig,
Braunschweig, Germany.
FTP: ftp.ibr.cs.tu-bs.de in the directory pub/local/nefclass.
WWW: http://www.tu-bs.de/ibr/fuzzy

Furthermore you need a C++–compiler that is able to handle templates (most C++–compilers do). The author compiled the sources on UNIX–systems under SUN–OS and Linux using the gnu–compiler gcc (version 2.7.2) and on a MS–DOS–system using Borland's Turbo C++ 3.0. The archive contains several makefiles for these systems. If you use another platform, you have to modify one of these makefiles.

Now we assume you have got the archive file and an appropriate compiler. First, you have to unpack the archive in a directory of your choice (be sure to reconstruct the directory structure that is given in the archive). Secondly, copy the appropriate makefile template to a file 'makefile' and adapt it for your system, if this is necessary. Now you should be able to compile the source files by typing

```
make
```

If the compilation was successful, you obtained the executable file which is called '*nefclass.exe*' on DOS–systems respectively '*nefclass*' on UNIX–systems. You may now delete the object files the compiler needed for linking. You can remove them using the following command:

UNIX–system: `rm *.o`

DOS–system: `del *.obj`

At this point the installation process is completed, if you always want to start the simulator from the directory where you installed the source code. Otherwise, move the executable file to a directory of your choice (e.g. 'C:\NEFCLASS' on DOS–systems or '/usr/local/bin' on UNIX–systems) that is contained in the path list given by the environment variable 'PATH' (if you don't know about the 'PATH'–variable, refer to the documentation of your operating system). In this case you have to set an environment variable 'NFCPATH', which must point to the directory, where NEFCLASS should look for the help file ('nefclass.hlp') and the (optional) default configuration file ('nefclass.cfg'). Note that you don't have to move these two files in the same directory where the executable can be found, it might be another (but the environment string 'NFCPATH' must contain the right directory in both cases). After that, the installation is complete.

A.3 Command–line Arguments

If you want to use one of the command–line arguments, you have to use the following syntax:

`nefclass` $\{[-p \mid --pipe]\}$ $\{< file_1 >< file_2 > \ldots < file_n >\}$

$< file_i >$	name of an existing file containing NEFCLASS commands
$-p \mid --pipe$	optional switch that cause NEFCLASS to make more detailed outputs to stdout that can be easily evaluated by other programs (like graphical frontends)

If no filename is given at the command line, NEFCLASS enters interactive mode.

Using NEFCLASS in Batch Mode

A command–line argument without a hyphen (–) is assumed to be a filename of an existing file containing NEFCLASS commands. Each command in this file will be executed in the specified order. After the last command file has been processed, NEFCLASS exits.

In a command file you may use every command that is valid in the interactive mode, except the shell ('!') command (the construct '!<shell command> is allowed anyway). Furthermore, comment lines (lines beginning with '#' or '%', or blank lines) are allowed in command files and will be ignored.

Using NEFCLASS in batch mode is recommended especially if the pattern file contains a high number of patterns, or if the classification problem is high–dimensional (i.e. the input patterns contain many features and / or there are many classes). In this cases you have to expect that the learning process takes a longer time, so that it is not convenient to use the interactive mode. The following file is a typical example for a NEFCLASS command file:

```
# Example for a NEFCLASS command file

# load the pattern file (data set for training)
load pattern iris1.dat

# make some settings for the rule creating and learning process...
set FSetsPerVariable = 3
set Learning_Rules   = Best
set Learning_Weights = Fixed
set FsIntersect_0.5  = No
set sigma_a = 0.0075
set sigma_b = 0.0075
set sigma_c = 0.0075

# create a new network based on the loaded pattern
create net iris-1st.net -r0

# start the training process
set maxepoch = 100
run learning
# save the net after learning !
save net

# load the test data set...
load pattern iris2.dat
# and test the network
run test
# it's possible to use shell commands in command files, e.g.
# move the detailed result file to another directory
!mv iris1.res ~/results
```

The available commands and configuration parameters are described in the following sections.

A.4 Configuration

Frequently used settings (learning parameters in the first place) for NEFCLASS can be written in a configuration file. An example configuration file is shown below (comments and blank lines are allowed as well as in command files).

```
# NEFCLASS 1.0 example configuration file
# General learning parameters
MaxEpoch              = 100
Stop_At_Percent       = 0
Stop_At_Misclassified = 0

# Update cycle for screen update
Update_Cycle = 10

# Create a logfile during net creation?
CreateLogFile = Yes

# Learning rates
# (for the conclusion weights and
# the parameters a, b, c of the fuzzy sets):
sigma = 0.001
sigma_a = 0.01
sigma_b = 0.01
sigma_c = 0.01

# Fuzzy sets per input variable (1 <= x <= 7)
FSetsPerVariable = 3

# Fuzzy sets must intersect at 0.5 ?
FsIntersect_0.5 = No

# Learning conclusion weights
# Possible values are: fixed, normalized or free
Learning_Weights = Fixed

# Learning rules
# Possible values are: simple, best or bestperclass
Learning_Rules = Best
```

For the parameter names and parameter values, NEFCLASS does not distinguish between lowercase and uppercase letters. Boolean values may be abbreviated with 'Y' for 'Yes' and 'N' for 'No'. For a description of the available parameters see Table A.1. An easy way to create a complete parameter file is to save the current configuration using the *save config* command (see Section A.5.2 for details). After you have started NEFCLASS, type the following command to create a complete parameter file:

```
nefclass> save config <filename>
```

Now you can edit this file using your favourite text editor.

By default, the configuration file 'nefclass.cfg' will be loaded at the program start. Any other configuration file can be loaded with the command

```
nefclass> load config <filename>
```

The Table A.1 provides an overview of all available configuration and learning parameters. These parameters can be set in a configuration file (as described above) or interactively using the *set* command (refer to Section A.5.7 for details).

MaxEpoch	int	Maximum number of epochs to learn
Stop_At_Percent	float	Stop learning, if at most *Stop_At_Percent* percent of the patterns have been misclassified
Stop_At_Misclassified	int	Stop learning, if at most *Stop_At_Misclassified* patterns have been misclassified
Update_Cycle	int	Update cycle for screen update during learning (*Update_Cycle* = 0 means no output to screen between the first epoch and the end of the learning process)
CreateLogFile	boolean	Create a logfile during net creation?
sigma	float	Learning rate for the conclusion weights (only considered if $Learning\_Weights \neq Fixed$)
sigma_a	float	Learning rate for parameter a of each fuzzy set
sigma_b	float	Learning rate for parameter b of each fuzzy set
sigma_c	float	Learning rate for parameter c of each fuzzy set
FSetsPerVariable	int	Fuzzy sets per input variable ($1 \leq x \leq 7$)
FsIntersect_0.5	boolean	Fuzzy sets must intersect at 0.5 ?
Learning_Weights	string	Possible values are: *fixed*: rule weights are fixed at 1.0 *normalized*: rule weights are from $[0,1]$ *free*: rule weights are from $\mathbb{R}$
Learning_Rules	string	Possible values are: *simple*: no selection after rule creation *best*: select the best rules after creation *bestperclass*: select the best rules per class

Table A.1: Overview of all available configuration parameters

A.5 Command Description

This section describes all available commands in NEFCLASS. Note that the commands have to be written in lowercase. No matter if you are using the interactive or the batch mode, there is only one command per line allowed (otherwise, an error will occure).

Available Commands

The following commands are available in NEFCLASS 1.0:

```
load      run       help
save      stop      !(OS shell)
set       abort     exit
show      cd        quit
create
```

In this documentation, curly braces ({}) denote optional arguments to some commands, and a vertical bar (|) separates mutually exclusive choices. Angle brackets (<>) are used to mark replaceable tokens.

To get detailed information about a single command, refer to the next section.

A.5.1 load

With the *load* command you can load a configuration, pattern or previously defined NEFCLASS network file.

Syntax:

```
load [pattern | net | config] <filename>
```

Examples:

```
load pattern test.dat
load config ~/nefclass/myconfig.cfg
load net ../net/foo.net
```

If there is currently a pattern file loaded or a network defined, it will be replaced without asking you for confirmation.

For information about the file formats of pattern and network files, see Section A.6. For details about configuration files refer to Section A.4.

A.5.2 save

The *save* command allows you to save a previously created (and trained) NEFCLASS network and the according fuzzy rules to a file. Furthermore, the current configuration can be saved to a file, too.

Syntax:

```
save [net | rules | config] {<filename>}
```

Examples:

```
save net test.net
save rules
save config ~/nefclass/myconfig.cfg
```

If no filename is given, the name of the last loaded file (of the same type, of course) will be used.

If you use the *save rules* command without specifying a filename, then the rules will be saved in a file where the filename is derived from the name of the network file (without extension), appending the extension '.rul' (e.g. if you have a network file 'iris.net', then the rules will be saved in a file 'iris.rul').

Important note:
If the specified file already exists, it will be overwritten *without* asking you for confirmation.

A.5.3 create

The *create* command creates a new NEFCLASS network based on the current loaded pattern file, where

$$\begin{aligned} <\text{number of input units}> &= <\text{number of input features per pattern}> \quad \text{and} \\ <\text{number of output units}> &= <\text{number of distinct classes}>. \end{aligned}$$

If no pattern file is loaded at this time, you will get an error message.

Syntax:

```
create net {<filename>} -r<(maximum) number of rule units>
```

The parameter <number of rule units> specifies the maximum number of rules to be created. If this parameter is set to 0, then NEFCLASS creates as many rules as possible for the given data set.

Examples:

```
# create a net with a maximum of 25 rules and save it into foo.net
create net foo.net -r25

# create a net with as many rules as possible for the given data set
create net -r0
```

If you don't specify a filename when using the *create* command, the new net will be saved automatically in a file in the current working directory, where the filename is derived from the name of the pattern file (without path and file extension), appending the extension '.net'.

By default, NEFCLASS creates a *log file* during the net creation process. This log file contains detailed information about

- which pattern was responsible for the creation of a distinct rule,
- whether a rule's conclusion had to be reconnected (to another output unit) or not,
- the performance of each rule.

Especially the rule performance is a very good indicator if you want to reduce the number of rules (and so make the interpretation of the system easier).

NEFCLASS creates fuzzy rules as follows:

First, a uniform fuzzy set partitioning of the domain of each input variable has to be determined. Then an initial rule base can be created by scanning the data. There are three ways to create an initial rule base, depending on the value of the variable *Learning_Rules* (see Table A.1).

A.5.4 run

This command starts the learning process, or the test cycle respectively, in case you have some patterns and a network loaded (or created) before. If not, or if the patterns are not compatible to the net, an error message will inform you.

During the *learning process* NEFCLASS tries to optimize the fuzzy rules by learning the parameters of the fuzzy sets that build the premise of these rules. The learning process ends if at least one of the end criterions *MaxEpoch*, *Stop_At_Percent* or *Stop_At_Misclassified* is fulfilled.

To test a previous trained NEFCLASS network, you should load a set of test patterns (previously classified patterns, but not the same patterns as the patterns used for learning) and execute the command *run test*. NEFCLASS then propagates every pattern of the test data set once, and informs you about the result. The test cycle can also be used to classify unkwown data, i.e. patterns where the correct class has not been determined. The NEFCLASS simulator recognizes those patterns by the keyword 'NOCLASS' specified in the pattern file (see Section A.6).

Syntax:

```
run [learning | test]
```

During the learning process, and at the end of the test cycle, you will be informed about the performance of the network (number of misclassified patterns, total error, mean error etc.).

During the test cycle, NEFCLASS creates a so-called *result file* (file extension '.res') in which you can find detailed informations about the propagation result (network output, pattern error) for each single pattern.

On UNIX–systems, executing the command *run learning* in the interactive mode creates a new process, so that the original process is able to continue handling commands, especially the commands *stop* and *abort*.

This feature allows you to influence a running learning process which is very useful, because (depending on the dimensions of input, rule and output units) the learning process can take much time.

A.5.5 stop / abort (UNIX–systems only)

The commands *stop* and *abort* can be used to terminate a running learning process in interactive mode.

The *stop* command can be used to stop the learning process, e.g. because you see that the results will not get better, and you do not want to wait until all epochs you specified are computed. The modifications of the fuzzy sets will be preserved after you have stopped the learning process, so you can save the network or start a new learning process with different learning parameters.

Using the *abort* command will also stop the learning process, but the network will be reset to the status before the *last run learning* command, i.e. the modifications of the fuzzy sets made by NEFCLASS will be lost.

Note: These two commands are only available on UNIX–systems.

A.5.6 show

Execute the *show* command to get informations about the current configuration. or to get specific informations about the currently loaded pattern set or network.

Syntax:

```
show [config             |
      <parameter name> |
      NumberOfFSets      |
      patterninfo        |
      rules              |
      fuzzy              |
      ranges             |
      inputnames         |
      classnames]
```

The command *show config* prints all parameter values, that can be set in configuration files, to stdout. *show <parameter name>* prints the value of the specified parameter to stdout, where <parameter name> is one of the parameters described in Section A.4.

The command *show NumberOfFSets* is useful to check the settings you made using the *set NumberOfFSets[<Index>]* command (see below). It prints the number of fuzzy sets for each input variable as a vector, enclosed in angle brackets, to stdout. This vector is only defined, if a pattern set is loaded, because the number of input variables determines the dimension of this vector. Note that the *NumberOfFSets* vector specifies the number of fuzzy sets for each partition, that *will* be used by the next *create net* command. It does *not* specify the number of fuzzy sets of the currently loaded network, if there is one.

Using the *show patterninfo* command will cause NEFCLASS to print some statistics about the currently loaded pattern set to stdout. The *show rules* command prints the rule base of the currently loaded network in form of linguistic IF–THEN–rules to stdout.

The arguments *fuzzy*, *ranges*, *inputnames* and *classnames* of the *show* command are in the first place useful for graphical frontends to get information about the currently loaded network without reading the network file itself.

show fuzzy prints the *FUZZY* section of the network file (see Section A.6) to stdout. Executing the *show ranges* command, you will get the lower and upper bound of each fuzzy partition. The *show inputnames/show classnames* command form prints the labels, that are assigned to the input variables and classes, to stdout.

A.5.7 set

The *set* command allows you to change a single parameter of the current configuration, where <parameter name> is one of the parameters described in Section A.4 (see Table A.1 for details) or 'NumberOfFSets[<index>]'.

Syntax:

```
set [<parameter name> | NumberOfFSets[<index>]] = <value>
```

Note that there has to be at least one blank character between <parameter name> and '=' and between '=' and <value> (this is not necessary in configuration files like 'nefclass.cfg').

The command form *set NumberOfFSets[<index>]* is useful (and the only way) to create fuzzy partitions with different numbers of fuzzy sets per partition. *<index>* is the index of the input feature and *<value>* is the number of fuzzy sets you want to have for it. The following command sequence illustrates this. We assume that we have input patterns with 5 features $x_1, \ldots, x_5$ per pattern. All domains of this input features are to be partitionated by 2 fuzzy sets, except x_2 that is a don't–care–variable, and x_4 that has to be partitionated by 3 fuzzy sets.

```
set FSetsPerVariable = 2
set NumberOfFSets[2] = 1
set NumberOfFSets[4] = 3
```

Note that the effect of this these commands depends on its sequence. If the command *set FSetsPerVariable = 2* would be the last, then it would overwrite the settings you have made for x_2 and x_4.

A.5.8 cd

The *cd* command allows you to change the working directory.
Syntax:

```
cd <directory name>
```

<directory name> may be any path that is valid in the underlying operating system (including relative pathes, tilde substitution in UNIX etc.). On DOS–Systems you can alternatively use the appropriate shell command ('*!cd <directory name>*), but this does not work on UNIX–systems (so you have to use the built–in *cd* command on UNIX–systems).
Examples:

```
cd ..\test      # (MS-DOS)
cd ~/data/iris  # (UNIX)
```

A.5.9 help

The *help* command displays an online–help. If you're interested in information about a particular topic (e.g. about how to use a particular command), just type *help <topic>*, where <topic> is the name of the topic that you are interested in. A list of all available topics can be obtained by typing *help contents*.

A.5.10 ! (OS shell)

The shell command creates a shell of the underlying operating system, so that it's possible for you to execute some shell commands without leaving the NEFCLASS simulator. To return to NEFCLASS, just type *exit*.

A single shell command may be executed by preceding it with the '!' character at the beginning of the command line (note: whitespaces between the '!' character and the command are not allowed). After the command is executed, the control will return immediately to NEFCLASS.

Examples:

```
# print the list of pattern files in the current directory (MS-DOS)
!dir *.dat

# view the fuzzy rules in file 'iris.rul' (UNIX)
!less iris.rul
```

A.5.11 exit/quit

The commands *exit* and *quit* and the END-OF-FILE character (if you are using NEFCLASS in batch mode) will exit NEFCLASS. Note that you will *not* be asked to confirm this action, even if there is still any data unsaved (e.g. a trained net or a modified configuration)!

A.6 File Formats Used by NEFCLASS

This section describes the file formats of *net files* (files that contain the complete description of a NEFCLASS system) and *pattern files* (files that contain the learning task) that are used by NEFCLASS, so that you are able to understand and modify those files.

Both file types are subdivided in sections. A section begins with a keyword (always written in uppercase) and ends when the next keyword appears. It is possible to insert comment lines in these files to make them more readable. A line *beginning* with '%' or '#' (or a blank line) is assumed to be a comment line and will be ignored. Note that comment lines are only allowed *between* sections, comment lines within a section cause an error.

The tables on the following pages describe the file formats formally. The description of each file format is followed by a short example file.

A.6.1 Pattern File Format

<pattern file>	:	<header> <pattern> END
<header>	:	<dimensions> [<name>] [<inranges>] [<varnames>] [<misvalue>]
<dimensions>	:	DIMENSIONS <newline> <input feature per pattern> <number of classes> <newline>
<name>	:	NAME <newline> <description of the data set (one line only)> <newline>
<inranges>	:	INRANGES <newline> <min> <max> of the domain of input feature x_1 <newline> ⋮ <min> <max> of the domain of input feature x_n <newline>
<varnames>	:	VARNAMES <newline> <name of input feature x_1 > <newline> ⋮ <name of input feature x_n > <newline> <name of class C_1 > <newline> ⋮ <name of class C_m > <newline>
<misvalue>	:	MISSINGVALUE <newline> <floating point number that is to be interpreted as missing value (must be $\neq 0$!)> <newline>
<pattern>	:	<example> | <unkwown>
<example>	:	PATTERNS <newline> <number of patterns> <newline> $< x_1 \ldots x_n > \quad < t_1 \ldots t_m >$<newline> (first pattern) ⋮ $< x_1 \ldots x_n > \quad < t_1 \ldots t_m >$<newline> (last pattern)
<unkwown>	:	NOCLASS <newline> PATTERNS <newline> <number of patterns> <newline> $< x_1 \ldots x_n >$<newline> (first pattern) ⋮ $< x_1 \ldots x_n >$<newline> (last pattern)

Table A.2: The Pattern File Format

Example:

```
# Example of a pattern file (iris data with only 3 patterns)

# Dimensions: number of input features per pattern and
# number of classes (required)
DIMENSIONS
4   3

# Short description of the data set (optional)
NAME
Iris data set, first part, 3 patterns only

# Ranges of the 4 input variables (optional)
INRANGES
4.0 8.0
2.0 5.0
1.0 7.0
0.0 3.0

# (optional) names of the input features and classes
VARNAMES
Sepal length in cm
Sepal width in cm
Petal length in cm
Petal width in cm
Iris-Setosa
Iris-Versicolour
Iris-Virginica

# This are 3 patterns,
# first given is the the number of patterns (required)
PATTERNS
3
5.1  3.5  1.4  0.2     1 0 0
7.0  3.2  4.7  1.4     0 1 0
6.3  3.3  6.0  2.5     0 0 1
END
```

A.6.2 Net File Format

<net file> : <header> <data section> END

<header> : <parameters> [<varnames>] [<agg_func>]

<parameters> : PARAMETERS <newline>
$< n >< k >< m >< k_m ax >$<newline>
with
$< n >$: Number of input units
$< k >$: Number of rule units
$< m >$: Number of output units
$< k_{max} >$: Maximum number of rule units specified
by the user at creation time

<varnames> : VARNAMES <newline>
<name of input feature $x_1 >$ <newline>
⋮
<name of input feature $x_n >$ <newline>
<name of class $C_1 >$ <newline>
⋮
<name of class $C_m >$ <newline>

<agg_func> : USESUM <newline> | USEMAXIMUM <newline>
(aggregation function to be used to determine the
activation of the output units)

<data section> : <partitions> | <connections> | <weights>

<partitions> : FUZZY <newline>
<partition> of the domain of input feature x_1
⋮
<partition> of the domain of input feature x_n

<partition> : <number of fuzzy sets in the partition (q)> <newline>
<lower bound> <upper bound> of the partition <newline>
<fuzzy set> (fuzzy set 1) <newline>
⋮
<fuzzy set> (fuzzy set q) <newline>

Table A.3: The Net File Format

<fuzzy set> : $<< a >< b >< c >< ls >< rs >< name >>$ <newline>
with
$< a >$: left point,
$< b >$: peak,
$< c >$: right point,
$< ls >= 1$ if left shouldered (0 otherwise)
$< rs >= 1$ if right shouldered (0 otherwise)
$< name >$ is the linguistic term associated with the fuzzy set

<connections> : MATRIX <newline>
$< I_1^{(1)} > \ldots < I_k^{(1)} >$ <newline>
$\vdots$
$< I_1^{(n)} > \ldots < I_k^{(n)} >$ <newline>
with
$< I_j^{(i)} >$: Index of the fuzzy set (relative to
the fuzzy partition of feature x_i) that is used in
the premise of rule j.
$< I_j^{(i)} >= 0$ means: don't care about this feature.

<weights> : WEIGHTS <newline>
<class> <conclusion weight> (rule 1) <newline>
$\vdots$
<class> <conclusion weight> (rule k) <newline>

Table A.4: The Net File Format (continuation)

Example:

```
# NEFCLASS 1.0 example net file
# Network contains 3 input, 3 rule and 2 output units.
#
PARAMETERS
3       3       2       3

VARNAMES
Input_1
Input_2
Input_3
Class_1
Class_2

USESUM

FUZZY
1
4       8
4.000000        6.000000        8.000000        1 1  dont_care

1
2       5
2.000000        3.500000        5.000000        1 1  dont_care

3
1       7
1.000000        2.370867        3.547293        1 0  small
2.370867        3.547293        6.345948        0 0  medium
3.547293        6.345948        7.000000        0 1  large

MATRIX
1       1       1
1       1       1
1       3       2

WEIGHTS
   1   1.000000
   2   1.000000
   2   1.000000

END
```

Appendix B

NEFCLASS–X User Manual

B.1 Introduction

NEFCLASS–X is a graphical user interface (GUI) for the command–driven NEFCLASS program. NEFCLASS–X can be used by menus as you know it from many other applications, so you do not have to write commands on the NEFCLASS command line. In addition to all features of NEFCLASS you are able to look at the fuzzy partitions of a network even during learning. You can use all features of NEFCLASS except using NEFCLASS in batch mode.

This chapter provides an overview of the menu structure and the most important user dialogs of NEFCLASS–X. It is strongly recommended that you read the NEFCLASS User Manual in Appendix A before you read this documentation of NEFCLASS–X.

B.2 Installation

To install NEFCLASS–X on your system, you need the file 'xnfcXX.zip' which contains the whole source code of NEFCLASS–X ('XX' is the version number; the current version of NEFCLASS–X is 1.0, so the right name for the archive is 'xnfc10.zip'). This archive can be obtained from

Institut fuer Betriebssysteme und Rechnerverbund
Technische Universitaet Braunschweig,
Braunschweig, Germany.
FTP: ftp.ibr.cs.tu-bs.de in the directory pub/local/nefclass.
WWW: http://www.tu-bs.de/ibr/fuzzy

To run NEFCLASS–X on your system, you need the Tcl/Tk–package (Tcl version 7.4 or higher, Tk version 4.0 or higher) and the command–driven NEFCLASS program, or course. About installation of NEFCLASS, refer to Section A.2.

First, you have to unpack the contents of the archive *xnfc10.zip* in a directory of your choice, e.g. '/usr/local/NEFCLASS–X'. The archive contains a file *xnfc* and several files with the file extension '.tcl'. All these files are text files containing Tcl/Tk–commands. The main file *xnfc* be executable. Change the file mode for *xnfc*, if this is necessary, using the following UNIX command:

```
chmod a+x xnfc
```

Now change to the directory where you installed the NEFCLASS–X source files, if you not already did. Start Tk's *wi*ndowing *sh*ell *wish* and type the following command at the *wish* prompt:

```
auto_mkindex . xnfc *.tcl
```

This command will create a file *tclIndex* in the current directory, which is needed by Tcl/Tk to find all procedures that are called by NEFCLASS–X.

At last, you have to edit the file *xnfc* to set the application path. The file *xnfc* begins with the following lines:

```
#!/bin/sh
#the next line starts the wish \
exec wish "$0"
set auto_path [linsert $auto_path 0 <NEFCLASS-X directory>]
```

In line 3 Tk's *wish* is started automatically. You only have to modify this line if the *wish* has another filename on your system.

In any case you have to modify line 4 to set the path where Tcl/Tk searches for script files. Replace the token '<NEFCLASS–X directory>' with the name of the directory where you installed the NEFCLASS–X package. So if we assume you installed the NEFCLASS–X files in the directory '/usr/local/NEFCLASS–X', the correct line must be:

```
set auto_path [linsert $auto_path 0 /usr/local/NEFCLASS-X]
```

Moreover, it is necessary that the *PATH* environment variable contains the directory where the executable file of the command–driven NEFCLASS program can be found. After that, the installation is complete.

Test your installation by starting NEFCLASS–X from any directory typing the command `xnfc` on the UNIX shell command line.

B.3 The Main Window

This section describes the components of the NEFCLASS–X main window that is shown in Figure B.1. At the top of the main window you find the *menu bar* which

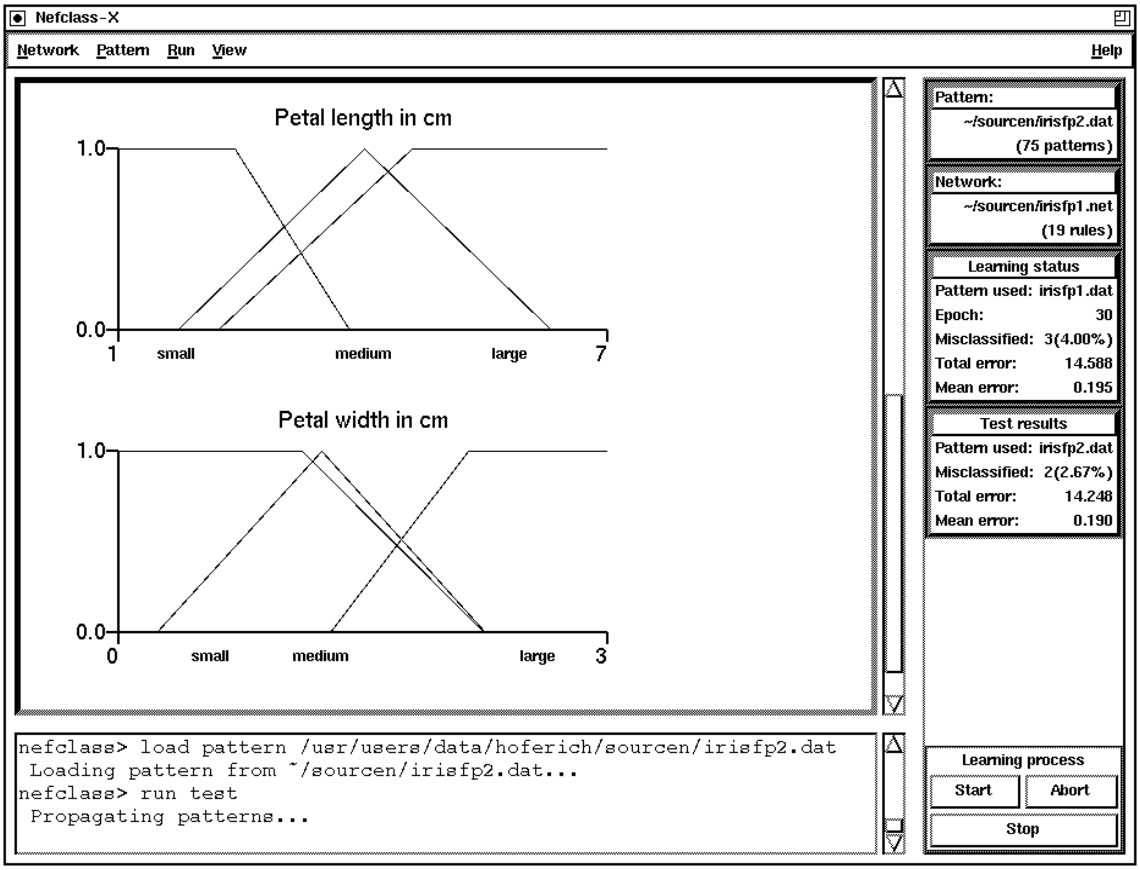

Figure B.1: The main window of NEFCLASS–X.

contains several submenus. These submenus are described in the following section. The large area below the menu bar is the *main display*, where the fuzzy partitions are displayed, whenever a network is loaded. After starting NEFCLASS–X, this display is empty. Below the main display there is a protocol window where you will see status messages from NEFCLASS. Many of the messages from NEFCLASS will be shown in pop-up dialog boxes, but others will be written into this protocol window.

At the right side you can find several status displays that inform you about the current loaded pattern file and network, and about the learning and test results you obtained with the current network.

At the bottom of the status area there is the *Learning Process Control Panel*, that contains the three buttons *Start*, *Stop* and *Abort*. Pressing the *Start* button will start the learning process, and is identical to the menu command *Run|Start learning*. The *Stop* button can be used to stop the learning process, e.g. because you see that the results will not get better, and you do not want to wait until all epochs you specified are computed. The modifications of the fuzzy sets will be preserved after you have stopped the learning process, so you can save the network or start a new learning process with different learning parameters.

Using the *Abort* button will also stop the learning process, but the network will be reset to the status before the *last* 'run learning' command, i.e. the modifications of the fuzzy sets made by NEFCLASS will be lost.

B.4 Menu Description

In this section the NEFCLASS–X menu structure will be described. If you are familiar with the NEFCLASS commands, it will be easy to find the appropriate menu item in the menu tree of NEFCLASS–X. The following subsections describes the submenus of the menubar.

Menu items that end on three dots (...) will open a dialog, other menu items will perform their command immediately (like the *Network|Save* menu item).

Menu Network

The *Network* menu contains menu items for creating a new network, load or save a network, reset the network and quit the application.

New... This item opens the *Create Network Dialog* that is described in Section B.5. Note that before you may create a new network, you have to load a pattern file. The pattern file will determine the number of input and output units by its dimensions, so that the created network is compatible to the loaded pattern set.

Load... This menu item opens a file selection dialog, where you may choose the network file you want to load.

Save Selecting this menu item will save the current network to disk using the filename that is displayed in the *Network Display*.

Save as... Use this menu item, if you want to save the current network to another file as the file that is displayed in the *Network Display*. Selecting this item will open the file selection dialog described in Section B.5, where you may specify the new filename.

View rules This menu item allows you to view the linguistic rules that build the rule base of the current network. After selecting this item, a text window will be opened in which the linguistic rules will be displayed.

Save rules... Selecting this menu item will save the rules of the current network to disk. The filename can be specified in a file selection dialog.

Change directory... Use this menu item to change the current working directory. Selecting it will open a dialog box, where you may specify the name of the directory you want to change to.

Exit Exit NEFCLASS–X.

Menu Pattern

Load pattern... This menu item opens a file selection dialog, where you may choose the pattern file you want to load.

View info This menu item allows you to get detailed information about the currently loaded pattern set. Selecting it will cause NEFCLASS–X to send the *show patterninfo* command to NEFCLASS. The obtained informations will be displayed in a text window.

Menu Run

The *Run* menu contains items for parameter settings of the learning process, and to start learning or testing a network. Moreover, there are items to load a predifined configuration file, and to save the current configuration to disk.

Learning control... This menu item opens the *Learning Control Dialog* that is described in Section B.5. In this dialog you may specify several stop criterions for the learning process, and how often the graphical display will be updated during the learning process.

Learning parameters... This menu item opens the *Learning Parameters Dialog* that is described in Section B.5. In this dialog you may specify the learning rates that are to be used during learning.

Start learning If you select this menu item, the learning process will be started using the currently loaded network and patterns (in case you have loaded patterns and loaded or created a network before). The learning process can also be started by pressing the *Start* button in the *Learning Process Control Panel* in the main window.

During the learning process the *Learning Status Display* will be updated each <*Update_Cycle*> epochs to inform you about the current learning results (number of misclassified patterns, total and mean error). The fuzzy partitions displayed in the main display will also be updated, so that you have an impression of how the fuzzy sets change during learning. The user interface is fully usable while the learning process is running, so that you are able to stop or abort the learning process, or to scroll in the main display to view another partition.

Test network The *Test network* menu item will cause NEFCLASS–X to test the current loaded network with the current loaded pattern set. After testing the network, the *Test Results Display* will contain the results of the test cycle.

Save configuration... Selecting this menu item will save the current configuration to disk. The name of the configuration file can be specified in a file selection dialog.

Load configuration... This menu item opens a file selection dialog where you may choose the configuration file you want to load.

Menu View

All menu items of the *View* menu open the file selection dialog described in Section B.5, where you can choose a filename to view a file of the desired type. The difference between the items in this menu is the pre–setting for the file filter in the file selection dialog. After selecting a file a text window will be opened to display the file.

Note that you may open more than one of these file windows, but only one of a type (it is not possible to view more than one log file, for example).

Rule file... Select this item if you want to view a rule file (file extension '*.rul'). Rule files contain the rule base of a NEFCLASS network in form of linguistic rules, and can be created using the menu item *Network|Save rules*. If you want to view the rules of the current loaded network, you may use the *Network|View rules* menu item.

Log file... Select this item if you want to view a log file (file extension '*.log'). Log files are created automatically when a new network is created, and contain detailed information about the net creation process, especially the performance of each rule.

Error file... Select this item if you want to view an error file (file extension '*.err'). Log files are created automatically during the learning process. An error file contains the number of misclassified patterns and the total and mean error for each epoch the network was trained. Error files can also be plotted easily to visualize the error curve. Use *gnuplot* for example to plot an error file.

Result file... Select this item if you want to view a result file (file extension '*.res'). A result file is created automatically, when a network is tested. It contains detailed results for each pattern of the pattern set that was used for testing.

Menu Help

Help index... After selecting this menu item a window containing all available help topics will be shown. Press mouse button 1 on the desired topic to get detailed help on that topic.

About... Selecting this menu item will open a window that contains informations about the version number and the author of NEFCLASS–X.

B.5 Dialogs

This section provides a description of the most important dialogs in NEFCLASS–X. All dialogs can be used with the keyboard or the mouse. Using the keyboard you can change the input focus from one widget to another with the tab–character (tabulator).

Nearly all dialogs contain an *Ok* button and a *Cancel* button. The *Ok* button closes the dialog, and accepts the values from this dialog. The *Cancel* button closes the dialog and discards the values.

Create Network Dialog

In the *Create Network* dialog you can make the settings that are needed for creating a new network. The dialog is shown in Figure B.2. It contains dialog elements for the network parameters (number of input, rule and output units), the number of fuzzy sets per input variable, the rule learning method and the aggregation function to be used.

The number of input and output units are set to the appropriate values of the current loaded pattern set and can not be edited. The input field for the number of rule units is set to 0, which will cause NEFCLASS to create as many rules as possible for the pattern set (see the description of the *create* command in Section A.5.3). You can change this value to another positive integer value, to give a limit for the maximum number of rules to be created, of course.

If you want to define individual fuzzy partitions, i.e. if you want different numbers of fuzzy sets per partition for each input variable, select the checkbutton labeled *Individually*. Selecting this checkbutton will change the state of the *Define* button to enabled. Press the *Define* button to open the *Define individual partitions* dialog that is described in the following subsection. If you want to have the same number of

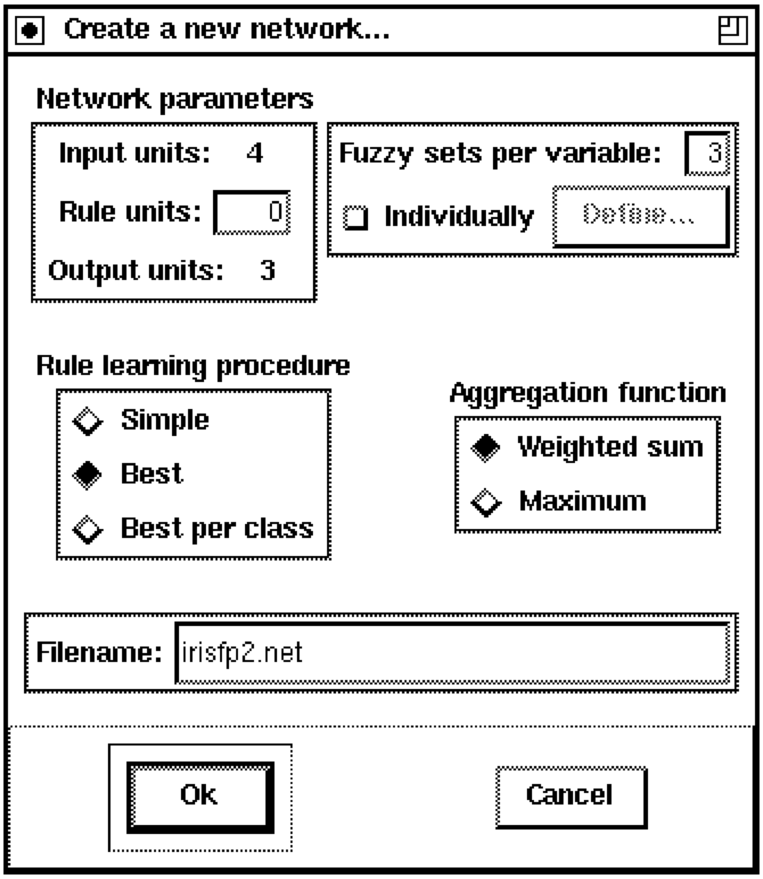

Figure B.2: The *Create Network* dialog.

fuzzy sets for each input variable, just deactivate the *Individually* checkbutton and set the value *Fuzzy sets per variable* to the desired value.

Define Partitions Dialog

This dialog is useful if you want to define individual fuzzy partitions for the input variables. It contains a listbox on the left side, where the input variables are listed (see Figure B.3). Select the desired variable in the listbox by clicking on it with mouse button 1 or press Return. The selected variable is shown on the right side. Now you can edit the value for the number of fuzzy sets. At last, press the *Set* button to save the modified value. If you are ready with defining fuzzy partitions, press the *Close* button to return to the *Create Network* dialog.

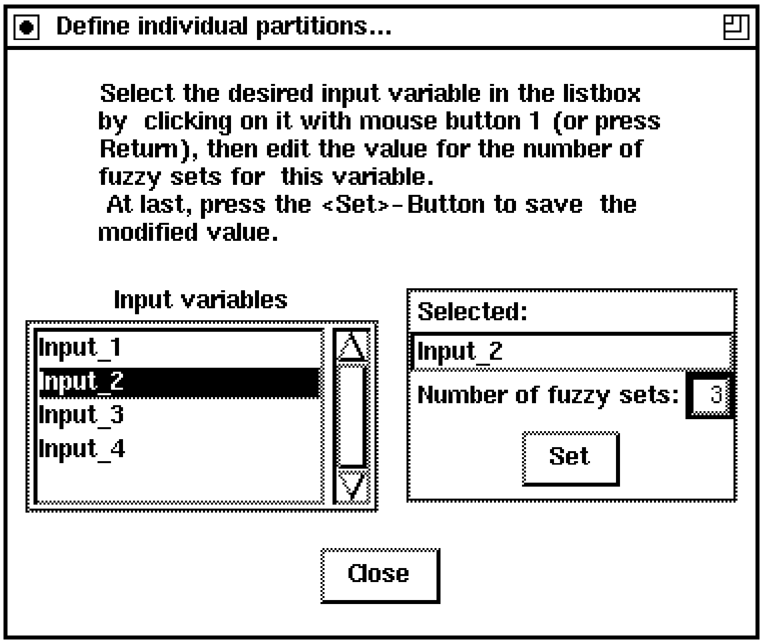

Figure B.3: The dialog to define individual fuzzy partitions.

Learning Control Dialog

In the *Learning Control* dialog you can specify different stop criterions for the learning process. The first of these criterions that is fulfilled during the learning process will terminate it (a description of these parameters can be found in Section A.4). With the parameter *Update cycle* you can specify the frequency for updating the display during the learning process (in number of epochs).

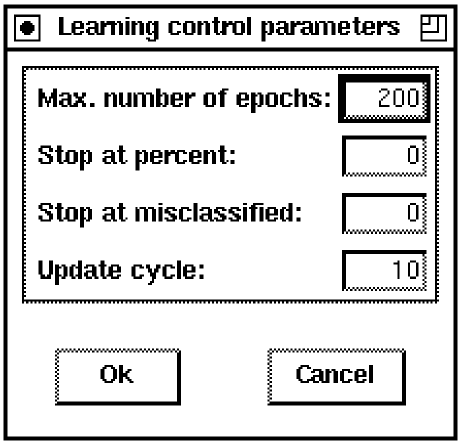

Figure B.4: The *Learning Control* dialog.

Consider that very small values for the update cycle will cause NEFCLASS–X to become slow.

Learning Parameters Dialog

This dialog allows you to specify the learning rates to be used during learning (see Section A.4 for details). Moreover, you can select whether conclusion weights should

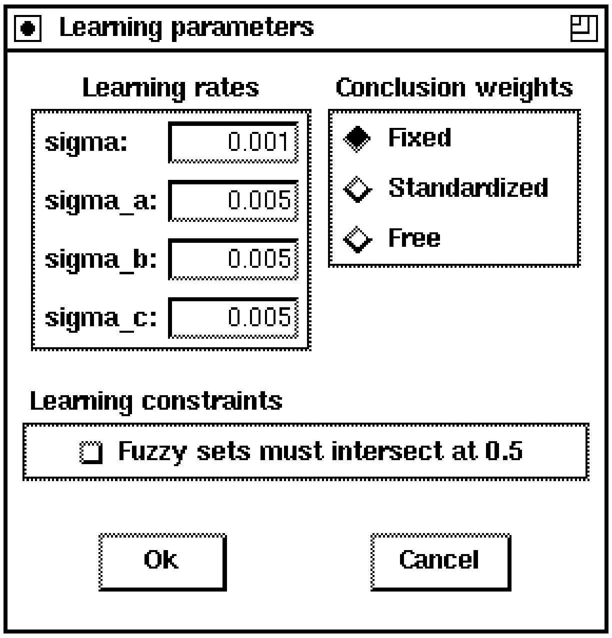

Figure B.5: The *Learning Parameters* dialog.

be learned or not. The recommended setting is that no conclusion weights are to be used, i.e. conclusion weights are *Fixed*. Furthermore, you can choose if the fuzzy sets have to be intersect at 0.5 when fuzzy set parameters are changed during learning. This leads to fuzzy partitions that are easy to interprete, but may prevent the learning algorithm from determining a good result.

File Selection Dialog

The *File Selection* dialog is used whenever you want to load or save a file. Figure B.6 shows this dialog. It contains a filter input line at the top, a listbox where the files matching the filter are displayed, and several buttons at the bottom.

The title bar of the dialog shows what kind of file will be loaded or saved. You can edit the filter to choose another set of files, or to change to another directory. You may use the same rules as you use with the UNIX–shell (e.g. tilde–substitution and

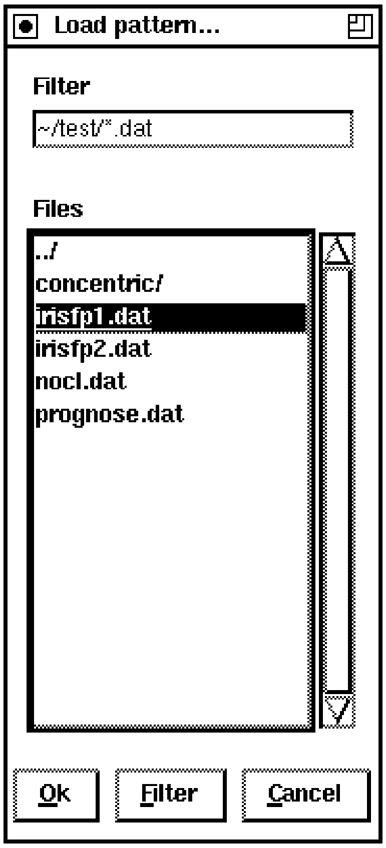

Figure B.6: The file selection dialog.

wildcards). After editing the filter press *Return* in the filter input line or press the *Filter* button at the bottom to update the contents of the listbox.

An easy way to walk through the directory tree is to double click on the desired directory with mouse button 1. All subdirectories in the directory given by the filter will be displayed at first in the listbox, indepent from the filter value. The directories and the following files that match the filter are sorted alphabetically.

To select a file for loading or saving, just double–click on it in the listbox, or mark it as selected and press *Ok*. To save to a new file, just type the new filename in the filter and then press *Return* in the filter or press the *Ok* button.