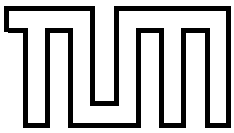
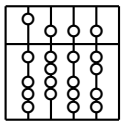


Technische Universität München
Fakultät für Informatik

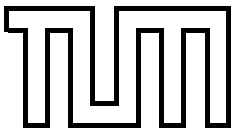


Diplomarbeit

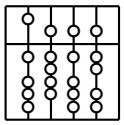
Benutzerorientierter Datenfilter für Umgebungsinformationen als Erweiterung eines Navigationssystems für sehbehinderte und blinde Mitmenschen

NAVI
Navigation **A**id for **V**isually **I**mpaired

Florian Reitmeir



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Benutzerorientierter Datenfilter für Umgebungsinformationen als Erweiterung eines Navigationssystems für sehbehinderte und blinde Mitmenschen

NAVI
Navigation **A**id for **V**isually **I**mpaired

Florian Reitmeir

Aufgabensteller:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inf. Martin Bauer Dipl.-Inf. Martin Wagner
Abgabedatum:	15. Dezember 2003

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Dezember 2003 Florian Reitmeir

Inhaltsverzeichnis

1 Aufgabenstellung **7**
- 1.1 Problemstellung . . . 7
- 1.2 Forschungsziel . . . 7

2 Anforderungsanalyse **9**
- 2.1 Erste Idee . . . 9
- 2.2 Meta–Daten, Informationsfilter . . . 11
- 2.3 Existierende Systeme . . . 13

3 Maschinelles Lernen **16**
- 3.1 Einführung . . . 16
  - 3.1.1 Agenten . . . 17
  - 3.1.2 Einfache Suchverfahren . . . 18
  - 3.1.3 Probleme . . . 19

4 Lösungsstrategie **21**
- 4.1 GPS–Navigation . . . 21
- 4.2 Taktile Ausgabe . . . 22
- 4.3 Karten, Routenplanung . . . 23
  - 4.3.1 Meta–Karte . . . 23
  - 4.3.2 Straßenkarte . . . 24
- 4.4 Informationsfilter . . . 25
  - 4.4.1 Vermeidung von Wiederholungen . . . 27
  - 4.4.2 Klassifizierung der Eingabe . . . 27
  - 4.4.3 Lernalgorithmen . . . 29
  - 4.4.4 Unterschiede beim Klassifizieren . . . 32

5 Konkrete Umsetzung **34**
- 5.1 Bibliotheken, Framework . . . 34
  - 5.1.1 CORBA . . . 34
  - 5.1.2 DWARF . . . 35
  - 5.1.3 Torch . . . 37
- 5.2 NAVI . . . 39
  - 5.2.1 Komponentensicht . . . 41
  - 5.2.2 Filter–Service . . . 46
  - 5.2.3 SVM Parameter . . . 50

6 Ergebnisse, Beispiel **52**
6.1 Szenario . . . 52
6.2 Lernmodell . . . 54
6.3 Weg durch das Szenario . . . 58

7 Zusammenfassung und Aussichten **61**
7.1 Aussichten . . . 61
7.2 Status . . . 63

A Appendix **64**
A.1 IDL – DWARF Filter Interface . . . 64
A.1.1 NaviMeta . . . 64
A.1.2 InputDataString . . . 65
A.2 Service – XML Beschreibung . . . 66
A.2.1 Filter–Service . . . 66
A.3 SVM – Test . . . 66
A.3.1 Parameter . . . 66
A.3.2 Beispiele . . . 68
A.4 Filter – Trainingsdaten . . . 68

Literaturverzeichnis **70**

Abbildungsverzeichnis

1.1 NAVI, Komponenten des Systems 8

2.1 NAVI – Funktionsidee, dem Nutzer wird das Restaurant gemeldet 10
2.2 Spamfilter – CRM114, SBPH 14

3.1 Entscheidungsbaum 18

4.1 Vibratoren, Taktile–Ausgabegeräte 22
4.2 Meta Karte, München Schwabing 24
4.3 Straßenkarte, Graphenmodell 25
4.4 NAVI [14] Sql Schema 26
4.5 Filter, Funktionsweise 27
4.6 Szenario – Positions–Hysterese 28
4.7 SVM 32
4.8 SVM – Klassifizierung in zwei Dimensionen 33

5.1 DWARF – NAVI mit DIVE–Debugger betrachtet 42
5.2 UML Serviceübersicht 43
5.3 NAVI – KDE Debugausgabe 44
5.4 UML Ablaufdiagramme 45
5.5 UML – Needs/Abilities des Filter Service 47
5.6 UML – NaviMeta Datenstruktur 47
5.7 UML – Filter–Service Objektübersicht 49
5.8 Ablaufdiagramm – Filter–Service 49

6.1 Szenario – Karte 53

1 Aufgabenstellung

1.1 Problemstellung

In unserer heutigen Zeit wird die Integration von behinderten Mitmenschen, in unserem Fall von sehbehinderten Menschen, immer größer geschrieben. Es werden vermehrt Hilfsmittel erforscht und entwickelt, um die bei einer Behinderung im Alltag auftretenden Schwierigkeiten besser bewältigen zu können.

Das Ziel dieser Diplomarbeit ist es, blinden und sehbehinderten Menschen eine sprechende Straßenkarte zur Verfügung zu stellen, die es den Zielpersonen ermöglicht, sich in unbekannter Umgebung zurechtzufinden. Dadurch kann die Selbständigkeit erhöht und die Abhängigkeit von behilflichen Mitmenschen vermindert werden.

Taktile und akustische Reize sind für blinde und sehbehinderte Menschen die wichtigste Möglichkeit, Informationen der Umwelt zu erfahren. Wird die Umwelt verstärkt akustisch wiedergegeben, stellt dies eine Flut von Informationen dar.

Der Schwerpunkt dieser Arbeit liegt darin, dem Benutzer eine Hilfe zu geben, um in der Flut der Umgebungsinformationen Interessantes von Uninteressantem zu trennen. Der verwendete Filter soll sich während der Navigation an die Bedürfnisse des Benutzers dynamisch anpassen.

1.2 Forschungsziel

Das Forschungsziel besteht aus zwei miteinander verknüpften Zielen. Zum einen soll ein lauffähiger Prototyp entstehen und zum anderen liegt der Schwerpunkt dieser Arbeit auf sich selbst optimierenden Informationsfiltern.

Forschungsziel NAVI

Der folgende Arbeitsablauf soll vom Projekt NAVI [14] nach dem Ende der Realisierung vom Benutzer durchgeführt werden können:

Ein Benutzer sitzt vor einem geeigneten [1] PC und wählt sich den gewünschten Stadtkartenausschnitt seiner Wunschstadt aus einer Liste aus. Im Anschluss überträgt er diese Karte auf seinen PDA [2].

Am PDA kann der Benutzer seine Tagesziele festlegen. Das Navigationssystem (Abb. 1.1, S. 8) bietet ihm ab diesem Zeitpunkt optimale Routen zu diesen Zielen an. Des weiteren

[1] Ein für Blinde und Sehbehinderte ausgestatteter Computer.
[2] PDA = Personal Digital Assistant

Abbildung 1.1: NAVI, Komponenten des Systems

repräsentiert das System die für den Benutzer relevanten Umgebungsinformationen durch eine Sprachausgabe.

Forschungsziel Informationsfilter

Das Projekt NAVI [14] soll als Grundlage für einen Informationsfilter dienen, welcher sich dynamisch an die Bedürfnisse des Benutzers anpasst. Um dies zu erreichen, müssen zuerst entsprechende Daten erfasst und aufbereitet werden.

Für den Filter soll ein sinnvolles Verfahren ausgewählt und anhand dessen das Gesamtsystem erprobt werden.

Als Vereinfachung (da nur zwei Entwickler für das gesamt System verantwortlich und sechs Monate Zeit waren) wird nur eine in weiten Teilen fest verdrahtete Implementierung durchgeführt, um die Ein-/Ausgabe– und Routenplanermöglichkeiten zu testen und das Potential des Filtersystems zu zeigen.

2 Anforderungsanalyse

2.1 Erste Idee

Es soll eine *sprechende Straßenkarte* entwickelt werden. Normale Karten sind für blinde– und sehbehinderte Nutzer nicht zu gebrauchen und Spezialangebote wie ertastbare Karten [39] [32] sind für den praktischen Feldeinsatz ungeeignet.

Dem Benutzer soll ein autonomes System geboten werden, mit dem er sich auch in unbekannter Umgebung zurechtfinden kann.

Das System sollte folgende Teile beinhalten:

- Standortdaten
- Routenplaner
- Karten
- Sprachausgabe

Um ein solches System zu realisieren sind eine Reihe von Komponenten notwendig, im Groben werden dies **Navigation, Benutzerinteraktion, Informationsaufbereitung und ein Informationsfilter** sein.

Aus diesen recht groben Anforderungen ergeben sich eine Menge anderer:

Positionsbestimmung

NAVI [14] soll ein Navigationssystem für Fußgänger werden. Die Positionsbestimmung muss unter freiem Himmel funktionieren und die Genauigkeit sollte in etwa einer Straßenbreite entsprechen (16m), damit festgestellt werden kann, auf welcher Straßenseite der Benutzer sich gerade befindet.

Die Auswertung der Daten muss in Echtzeit möglich sein, um die Bewegungsgeschwindigkeit und die Gehrichtung des Benutzers schnell bestimmen zu können. Dies ist wichtig, um den Benutzer möglichst rasch auf Fehlgänge seinerseits hinzuweisen und die Route zu korrigieren.

Eine Positionsbestimmung innerhalb von Gebäuden wird für dieses System nicht erforderlich sein, da sich hier SB–Personen [1] einfacher mit herkömmlichen Hilfsmitteln zurecht finden.

[1] SB–Personen steht für sehbehinderte und blinde Personen.

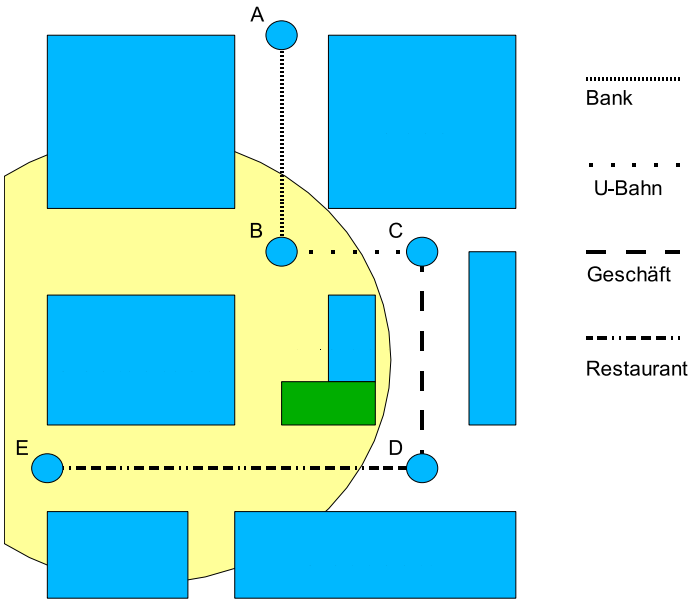


Abbildung 2.1: NAVI – Funktionsidee, dem Nutzer wird das Restaurant gemeldet

Der **Informationsfilter** benötigt ebenfalls sehr genaue Positionsdaten. Bei seiner Anwendung sind ungenaue Daten nicht tragisch, aber sehr störend. Auf Grund der ermittelten Koordinaten werden die möglichen Umgebungsinformationen zusammengestellt. Abweichungen wirken sich direkt auf diese Zusammenstellung aus und verzerren so das Bild der Umgebung (Abb. 2.1, S. 10).

Betriebssystem, Bibliotheken

Für das **Betriebssystem** gilt als Anforderung, dass es während der NAVI [14] Prototyp Implementierung auf einem normalen Notebook verfügbar sein muss, da uns nicht alle Hardwarekomponenten zur Verfügung stehen, um das gesamte Projekt für einen PDA zu entwerfen.

Falls NAVI [14] weiterentwickelt werden soll, ist die Unterstützung des Betriebssystems für Handheld–PCs sehr wichtig. Es muss bei der Auswahl der verwendeten Bibliotheken drauf geachtet werden, dass diese wenig Arbeitsspeicher und Rechenlast benötigen.

NAVI [14] wird aus mehreren Komponenten bestehen und es ist noch nicht abzusehen, wie aufwendig die einzelnen Teile genau werden. Eine vernetzte Struktur ist deshalb anzustreben, um im Zweifelsfalle Teile des Systems auf andere Rechner zu verlagern.

Netzwerk

Falls NAVI [14] auf einer vernetzten Struktur aufbauen sollte, muss das Netzwerk, welches die Komponenten verbindet, einigen Ansprüchen genügen. Der Benutzer ist **mobil**, falls zur Laufzeit Vernetzung erforderlich ist, muss das Netz an allen erreichbaren Stellen für den

Benutzer verfügbar sein. Sollte es Grenzen der geographischen Verfügbarkeit geben, muss das System den Benutzer rechtzeitig auf diese hinweisen.

Ein-/Ausgabe

Das System soll natürlich **bedienbar** sein. SB–Personen haben in diesem Zusammenhang besondere Bedürfnisse. Sehbehinderte könnten zwar mit einem Bildschirm entsprechender Grösse arbeiten, das Projekt soll jedoch mobil bleiben.

NAVI [14] muss ohne Bildschirm bedienbar sein. Sprachausgabe, wie sie in vielen ähnlichen Projekten häufig zum Einsatz kommt, wird sich nicht vermeiden lassen. Bei der Realisierung soll jedoch versucht werden möglichst wenig auf dieses Ausgabemedium zurückzugreifen [29], da SB–Personen das Gehör für andere Zwecke ebenfalls benötigen.

2.2 Meta–Daten, Informationsfilter

Dem Benutzer soll nicht nur ermöglicht werden zu navigieren, er soll sich ein akustisches *Bild* seiner Umgebung machen können. Dazu bekommt er zusätzliche Informationen, wie z.B. ob sich ein Restaurant in seiner Nähe befindet oder eine U–Bahn Station etc. (Abb. 2.1, S. 10), mitgeteilt.

Um nun zu wissen, was um ihn herum geschieht und ihn navigieren zu können, benötigt das Projekt NAVI [14] digital aufbereitete Karten, zum einen müssen diese in eine passende Datenstruktur gebracht werden, um eine effiziente Anwendung von Wegesuchalgorithmen zu ermöglichen. Zum anderen benötigt das System noch eine Reihe weiterer Informationen, welche über die üblichen Routenplanerdaten hinaus gehen.

Da diese Karte eine Vielzahl von Informationen beinhalten muss, teilen wir sie zunächst in verschiedene **Schichten** [26] ein

- **Straßenkarte:**
  Dieser Schicht ordnen wir alle Daten zu, welche später zur Wegfindung gebraucht werden, also Straßen, deren Länge und Größe, Kreuzungen (siehe 4.3.2) und andere Straßenend- und Schnittpunkte.

- **Gebäude Metainformationen:**
  Da wir dem Benutzer ein Bild seiner Umgebung *ansagen* wollen, benötigten wir eine Vielzahl von aufbereiteten Informationen (Gebäude, U–Bahnstationen, Parks, etc.).

  Der zu erfassende Punkt auf der Karte (z.B. ein Gebäude) hat eine gewisse räumliche Ausdehnung, eine äußere Beschreibung und meist auch einen Inhalt (siehe 4.3.1). Falls es sich um ein Geschäft oder mehrere handelt, ist auch die Art, der Name, Öffnungszeiten etc. interessant.

- **Sonstiges:**
  Andere Informationen, welche Karten enthalten können, werden vorerst nicht verwendet. Für zukünftige Erweiterungen könnten aber noch andere Informationen benötigt werden, z.B.:

Höhenunterschied: Geht der Benutzer einen Berg lieber zu Fuß hinauf, oder ist ein Taxi angebrachter?

Bevölkerungsdichte: Es ist nicht Jedermanns Sache, spät Abends in verlassenen Stadtteilen zu spazieren.

Mobilfunkempfang: Ist es möglich, auf meiner gewählten Route zu telefonieren? Hilfe zu holen? Oder bin ich sehr wahrscheinlich auf mich allein gestellt?

...

Es werden Kartendaten gesucht, welche wie beschrieben in Schichten zerlegbar sind und genug Informationen bieten um aus ihnen Umgebungsdaten zu extrahieren.

Ein Problem für die Zukunft ist sicher die Aktualität dieser Kartendaten. Der Benutzer kann die Aktualität der Daten nur schwer nachvollziehen, Hinweise auf eventuelle Änderungen sind deshalb wünschenswert.

Die Informationen zu sammeln ist eine Sache, es dem Benutzer zu ermöglichen mit ihnen zu arbeiten und sich aus ihnen ein Bild der Umgebung zu machen ist eine andere. Die voraussichtliche Anzahl der Informationen setzt einen **Filter** voraus, der den Benutzer vor zu viel des Guten bewahrt.

2.3 Existierende Systeme

NAVI [14] ist nicht das erste Projekt ist, das versucht, SB–Personen die Navigation zu erleichtern. Hier kurz ein Überblick zu anderen bestehenden Systemen. An dieser Stelle möchte ich auch auf die Parallelarbeit von Günther Harrasser [35] zum Projekt NAVI [14] verweisen, welche die Routenplanung und Benutzer Ein–/Ausgabeaspekte der Blindennavigation näher betrachtet.

Hier soll das Hauptaugenmerk vor allem auf Produkte und Projekte gelegt werden, welche interessante Filteralgorithmen umsetzen.

Navigationssystem – TREKKER

TREKKER [22] ist ein Gerät der kanadischen Firma Visuaide, welches bereits kommerziell am Markt verfügbar ist.

Das Gerät beruht auf einem Compaq–PDA, der mit Unterstützung von GPS die aktuelle Position bestimmt. Es kann derzeit angenommen werden, dass TREKKER das am weitesten entwickelte Produkt auf dem Gebiet der Navigation für Blinde und Sehbehinderte ist. Hier eine Featureübersicht:

- Positionsbestimmung via GPS.
- Aktuelle Karten können vom Benutzer über das Internet bezogen werden.
- Es können **Wegmarken** auf der Karte hinterlegt werden (sogenannte "Punkte von Interesse"), auf diese wird der Benutzer während der Navigation hingewiesen.
- Ausgabe des aktuellen Straßennamens.
- Das Touchpad kann in verschiedene Eingabemodi gebracht werden, je nach Vorliebe des Benutzers, als Braille- oder Telefontastatur.
- Die komplette Ausgabe erfolgt akustisch.

Das Benutzerverhalten wird vom Gerät nicht erlernt, es liefert auch keine detaillierteren Umgebungsinformationen an den Benutzer (außer die aktuellen Straßennamen).

Touristenführer – Nomadic Information System

[5] Das **Nomadic Information System** ist ein Projekt des Fraunhofer Instituts.

Das Projekt ist eine Weiterführung von anderen Projekten wie dem **virtual Tourist Guide**. Im Prinzip geht es darum, einem Touristen seine Umgebung mittels Augmented Reality näher zu bringen. Der Nutzer des **virtual Tourist Guide** setzt sich eine Brille auf, sieht sich die Umgebung an, z.B. in der Münchner Altstadt das Rathaus, möchte mehr Informationen und bekommt diese in sein Sichtfeld eingeblendet. Das System hat den Vorteil, dass dem Nutzer nicht nur die aktuelle Ansicht des Gebäudes gezeigt werden kann, sondern auch eine Rekonstruktion aus vergangener Zeit.

Das **Nomadic Information System** verfolgt ähnliche Beweggründe wie der *virtual Tourist Guide*, ist aber nur mehr an Alltagsbedürfnissen orientiert. Grundüberlegungen und Projektvoraussetzungen des Systems sind:

- Der Benutzer bewegt sich durch physikalischen Raum.
- Er benötigt spezielle Ausrüstung.
- Der Benutzer hat keinen ständigen Zugang zum Internet.

Das System soll nach Fertigstellung folgende Funktionen bieten:

- Standortbestimmung des Benutzers.
- Interessen des Benutzers erkennen.
- Die Umgebung erkennen.

Um diese ehrgeizigen Ziele zu erreichen wurde das Projekt in mehrere Subprojekte aufgeteilt. Leider ist aus den Projektübersichten nicht erkennbar welche genauen Verfahren zum Einsatz kommen, um die verschiedenen Ziele zu erreichen.

Spamfilter – SpamAssassin

SpamAssassin [19] macht genau das, was NAVI [14] auch erreichen will, nämlich den Benutzer vor unerwünschten Informationen [34] [40] zu schützen.

Die Arbeitsweise ist regelbasiert. SpamAssassin besteht aus einer Reihe von bewertenden Regeln. Diese vergeben eine gewisse Anzahl von Punkten falls sie zutreffen.

Zusätzlich wird ein naiver Bayes–Lernalgorithmus eingesetzt (die E–Mail wird in Token zerlegt, die mit Bayes ausgewertet werden), der ebenfalls Punkte vergibt. Bekommt eine E–Mail mehr als 5 Punkte (Standardeinstellung), wird sie als Spam klassifiziert.

Das Positive an SpamAssassin, im Gegensatz zu vielen anderen Informationsfiltern ist, dass in der Anfangsphase umfangreiche Regeln E–Mails beurteilen und erst mit genug Lernmaterial ein Bayes–Klassifizierer angewandt wird.

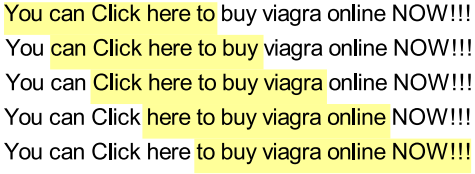


Abbildung 2.2: Spamfilter – CRM114, SBPH

Spamfilter – CRM114

CRM114, **the Controllable Regex Mutilator** [2] ist ebenfalls ein Spamfilter [34]. Im Gegensatz zu anderen Programmen, die nur den naiven Bayes Algorithmus [40] verwenden, hat der Autor sich eine Erweiterung zu diesem erdacht und mit CRM114 implementiert.

Getauft wurde diese Erweiterung **Sparse Binary Polynomial Hashing (SBPH)** [45]. Grundsätzlich wird Bayes zum Lernen verwendet, d.h. der Text wird in Tokens aufgespalten die gezählt werden. Um auch Aussagen über Phrasen im Text zu erhalten, wird SBPH auf den Text angewandt. Dies ist ein Fenster [2], das über eine konstante Anzahl von Wörtern geschoben wird (Abb. 2.2, S. 14). Die dabei entstehenden Phrasen werden als neue Tokens zu den einfachen Wort–Tokens hinzugenommen. Um Speicherplatz zu sparen, wird für jedes Token ein Hashwert berechnet, mit dem weitergearbeitet wird.

CRM114 ist besonders erwähnenswert, da eine Filterrate von 99.87% erreicht wird (Verfahren wie unter [40], [33], [41] beschrieben kommen auf 97%).

Musik – IMMS

Ein völlig anderes Lernsystem ist das **Intelligent Multimedia Management System** [11], ein XMMS–Plugin [25] welches versucht, die Musikgewohnheiten des Hörers zu erlernen. Als Lernkriterien stehen dem Programm nur folgende Informationen zur Verfügung:

- Wie oft wurde das Stück schon gespielt?
- Wurde der Titel fertig gehört, also wieviel Prozent wurden angehört.

Das Programm speichert seine Daten in einer SQLite [20] Datenbank. Das Lernverfahren ist einfach aber erstaunlich effizient.

IMMS verwaltet eine Spielliste, in der jedem Titel Punkte gegeben werden, je nachdem wie lange er gespielt wurde.

Titel mit wenigen Punkten werden übersprungen. Wenn ein Titel abgebrochen wird, werden seine Punkte verringert, ansonsten erhöht. Damit nicht immer die gleichen Titel zum Zug kommen [2], wird noch der letzte Spielzeitpunkt festgehalten und nur Titel gespielt, die am längsten nicht zu hören waren.

Damit neue Titel rasch gespielt werden, bekommen sie eine recht hohe Grundpunkteanzahl.

[2] Das Verfahren schaukelt sich ansonsten auf, beliebte Titel werden öfter gespielt und bekommen dadurch mehr Punkte, wodurch Sie noch öfter gespielt werden.

3 Maschinelles Lernen

3.1 Einführung

NAVI [14] soll später die Möglichkeit besitzen zu lernen. Die Vorstellung ist in etwa so: Der Benutzer geht mit seinem Navigationsgerät durch die Strassen und bekommt dort immer wieder viele Informationen zu seiner Umgebung. Das System soll **lernen**, was den Benutzer interessiert und ihm später nur die Details zukommen lassen, die er zu hören wünscht.

Das System soll ein Benutzerprofil erlernen und mit diesem Informationen filtern. Uninteressante Informationen werden so vom Benutzer ferngehalten.

Bevor später auf die Problematiken (siehe 3.1.3) eingegangen wird, beginnt dieses Kapitel mit einer von der schlussendlichen Anwendung losgelösten Einführung in maschinelles Lernen.

Unter **Maschinellem Lernen**, **Künstlicher Intelligenz** oder auch **Artifical Intelligence** ist das gleiche zu verstehen. Die Begriffe stammen aus der Pionierzeit (ca. 1970). Damals war man angesichts schneller Anfangserfolge was Computerintelligenz betrifft sehr optimistisch eingestellt. Einige Jahre später folgte die Ernüchterung und der Begriff **wissensbasiertes System** wurde eingeführt, was eher dem Forschungsstand entspricht.

Wenn man von **Lernen** und **Intelligenz** spricht, hat man immer Menschen vor dem inneren Auge. Bei Computern unterscheidet man grundsätzlich zwischen zwei Arten von Systemen:

schwache KI: Der Computer ist ein Instrument zur Untersuchung von Denkprozessen. Ziel ist es, menschliches Denken möglichst gut zu **simulieren**.

starke KI: Der Computer simuliert nicht das Denken, sondern mit den richtigen Programmen wird ihm **Verstehen** und **Denken** zugestanden.

Um zu testen, ob ein Computer wirklich intelligent ist, wurden im Laufe der Zeit viele Tests erdacht, wie z.B. der Turing–Test [44], Schanks System, Searls Szenario [43], uva.

Definition 3.1.1 *Turing-Test*
*Drei Teilnehmer, zwei Menschen (*A *und* B*) und ein Computer* C*, werden in getrennte Räume geführt. Der Teilnehmer* A *kann mit* B *und* C *mittels einer Datenleitung reden, weiß aber nicht, ob sein gegenüber der Mensch oder der Computer ist.*

Aufgabe von Teilnehmer A ist es nun herauszufinden, wer Mensch ist und wer nicht. Der Mensch B *darf dabei* A *Hinweise geben, der Computer* C *muss probieren,* A *zu täuschen.*

Schafft es der Computer C*, den Teilnehmer* A *zu täuschen, hat er den Turing–Test bestanden.*

Der Turing–Test veranschaulicht zwar das Ziel, an dem die Forschung arbeitet, hat aber das Problem, dass die Lernfähigkeit eines Systems damit nicht bewertet wird. Er gilt heutzutage nicht mehr als alleiniges Kriterium, um zu beurteilen ob ein Computer intelligent ist.

Des weiteren soll hier kurz erwähnt werden, dass die perfekte Nachbildung eines denkenden Menschen eigentlich nicht das Ziel ist, welches man erreichen möchte. Denn Computer sollen keine Fehler machen, wenn Sie aber eine perfekte Nachbildung sind, lassen sich diese nicht vermeiden.

3.1.1 Agenten

Agenten stellen eine recht moderne Sicht auf wissensbasierte Systeme dar.

Definition 3.1.2 *Ein Agent ist eine Entität, welche mit seiner Umwelt interagiert. Um mit seiner Umwelt zu kommunizieren, benutzt ein Agent:*

Percepte: *Sie dienen dem Agenten dazu, Informationen über seine Umwelt zu sammeln. Sensoren bilden Informationen aus der Umwelt in das Weltbild des Agenten ab (z.B. eine Videokamera).*

Effektoren: *Mit Effektoren kann der Agent seine Umwelt beeinflussen (z.B. Räder).*

Agenten besitzen oft noch einen Wissensspeicher, in den sie erworbenes Wissen ablegen, um später darauf zurückgreifen zu können.

Ein Agent hat immer ein Ziel. Um sein Ziel zu erreichen tätigt er Aktionen. Jede Aktion verändert seine Umwelt oder auch nur das Weltbild des Agenten. Falls ein Agent immer sinnvoll entscheidet, spricht man von einem **rationalen** Agenten (Ich betrachte in dieser Arbeit nur diese Art.).

Ein kurzes Beispiel soll veranschaulichen, wie man sich einen Agenten vorstellen kann:

Beispiel 1 *Unser Agent sei ein Roboter. Er besitzt eine Videokamera, um zu navigieren und ein Fahrwerk, mit dem er sich bewegen kann. Das Ziel des Agenten ist es, ein Zimmer zu reinigen. Dazu bewegt er sich systematisch durch das Zimmer und merkt sich, wo er schon gereinigt hat. Sobald er an jeder Position des Zimmers einmal war, nimmt er an, das Zimmer vollständig gereinigt zu haben.*

In diesem kurzen Beispiel habe ich schon zwei wichtige Annahmen vorweggenommen:

- Das Ziel des Roboters ist es, das Zimmer möglichst rasch zu reinigen. Über die Qualität wurde keine Aussage gemacht, genauer gesagt hat der Roboter in diesem Beispiel gar nicht die Möglichkeit festzustellen oder zu **messen**, wie stark verunreinigt das Zimmer ist.
- Der Roboter beeinflusst seine Umwelt nur positiv. Er hinterlässt selbst keinen Schmutz. Anders gesagt: Seine Räder sind blitzblank und hinterlassen keine schwarzen Streifen, welche der Roboter selbst reinigen sollte.

Das Beispiel sollte kurz veranschaulichen, dass es einfach ist, einen Agenten zu erschaffen. Meist jedoch ist es schwer, das Ziel eindeutig zu definieren und für den Agenten ist es oft schwer zu prüfen, ob er sein Ziel erreicht hat. Im Beispiel hätte das Ziel auch einfach nur sein können: Besuche jede Stelle im Zimmer einmal, denn bei der Bewegung wird automatisch gereinigt.

Es ist wichtig dafür zu sorgen, dass ein Agent sein Ziel erreichen kann. Im obigem Beispiel könnte es für den Agenten unmöglich gemacht werden, indem z.B. verschiedene Teile des Zimmers für den Roboter nicht zugänglich sind (durch Möbel verstellt).

3.1.2 Einfache Suchverfahren

Ein klassisches Gebiet von wissensbasierten Systemen sind Entscheidungssysteme (z.B. Spiele).

Man geht von einem oder mehreren Anfangs– und Endzuständen aus. Jede Aktion des Agenten ändert den aktuellen Zustand und ermöglicht es, verschiedene Neue zu erreichen. Falls der Agent einen Finalzustand erreicht hat, ist seine Aufgabe vollendet.

Man kann sich das Ganze als Graph vorstellen, in dem jede Kante eine Entscheidung ist und jeder Knoten einem Zustand entspricht.

Aus dieser Sicht ergeben sich zwei Arten von Suchverfahren:

Informierte Suche: Der Agent kennt den kompletten Graphen, und kann mittels Algorithmen aus der Graphentheorie einen optimalen Weg berechnen.

Uninfomierte Suche: Der Graph ist entweder zu groß oder nicht vollständig bekannt. Der Agent kennt also nur den aktuellen Zustand und im besten Fall umliegende Zustände. Er muss also die Möglichkeiten abwägen und schätzen, welche Entscheidung die beste ist.

Technisch interessanter ist die uninformierte Suche, denn hier ergeben sich eine Reihe von Problemen.

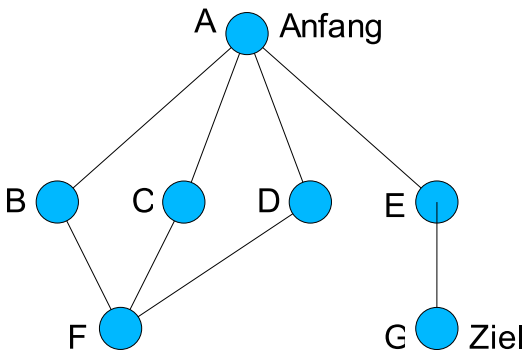


Abbildung 3.1: Entscheidungsbaum

- Im ersten Augenblick geht man davon aus, dass Zustände einmalig sind oder der Agent es vermeidet, den gleichen Zustand mehrfach zu erreichen (im Kreis zu gehen). Es ist ein Problem, solche Kreiswanderungen festzustellen, wenn der Graph zu viele Zustände hat, oder das Gedächtnis des Agenten nicht ausreicht, sich die besuchten Knoten zu merken.
- Der Agent weiß bis kurz vor dem Ende nicht, wie weit er von seinem Ziel entfernt ist. Meist ist es so, dass es mehrere Zielzustände gibt, die im Graphen keineswegs gehäuft sein müssen.
- Es kann Sackgassen im Graphen geben, also Entscheidungen durch die ein Finalzustand niemals erreicht werden kann.

Um die Sache zu veranschaulichen noch ein kleines Beispiel:

Beispiel 2 *Jede Figurenposition auf einem Schachbrett ist ein Zustand. Jeder Zug eine Aktion, die den Zustand verändert.*

Jedes Schachspiel kann also auf einen solchen Graphen umgelegt werden, indem es eine Startposition gibt, eine Reihe von Spielzuständen und Zustände in denen kein Sieg mehr erreicht werden kann.

Da es 64 Felder gibt, gibt es also ca. 2^{64} *Zustände (die Fälle in denen Figuren geschlagen werden, unterschlage ich). Es ist also für die aktuelle Technik nicht möglich, alle Zustände zu erfassen.*

Der Agent hat mehrere wichtige Grundaufgaben zu erfüllen (ganz abgesehen davon, dass er einen Gewinnzustand erreichen will), z.B. soll er Zyklen vermeiden und ein Unentschieden erkennen können. Die Orientierung im Graphen findet der Agent, indem er eine Position und den Abstand *zum Finalzustand abschätzt (normalerweise indem die Figuren und ihre Positionen bewertet werden).*

3.1.3 Probleme

Auf die wichtigsten Grundlagen von **wissensbasierten Systemen** welche zum Verständnis von NAVI [14] wichtig sind, wurde bislang eingegangen. Ich möchte an dieser Stelle die grundlegenden Grenzen und Probleme aufzeigen, welche mit den bisher vorgestellten Mitteln nicht zu beheben sind.

Komplexität

Oft reduziert sich das gesuchte Verhalten eines Agenten auf einen Suchalgorithmus. In jedem Fall müssen aber zuerst die Eingabedaten aufgearbeitet werden. Dies kann sehr schnell rechenintensiv werden z.B. Echtzeit Bildverarbeitung, Spracherkennung, usw. In vielen Fällen werden zur Aufbereitung der Perzepte eigene Agenten definiert.

Unschärfe, Fehlertoleranz

Eingabedaten (Perzepte) sind oft mit Fehlern behaftet. Der Agent sollte sich durch solche Fehler nicht beeinflussen lassen. Schwierig wird es, wenn die Fehler sich direkt auf die Sicht des Weltbildes eines Agenten auswirken.

Wissensrepräsentation

Es ist schwierig erfahrenes Wissen zu speichern.

Logik: Wissen wird als Verknüpfung logischer Zusammenhänge angesehen. Jede neue Information wird an bestehendes Wissen angeknüpft.

Es kann in einfacher Weise von einer Information auf eine andere geschlossen werden. Dazu müssen die Informationen aber eine gewisse Struktur mitbringen oder aber auch entsprechende mathematische Operationen eingeführt werden. z.B.
$< Kleidung, Essen, Computer >= besitz(Mensch)$

Die Hauptnachteile dieser Art der Wissensaufbereitung sind:

- Fehlerhafte Informationen führen dazu, ganze Teilbereiche von erworbenen Wissen in Widersprüche zu versetzen.
- Lernen ist komplex: Der Agent muss das Wissen erst in den korrekten Kontext bringen. Probleme sind vorhersehbar, falls Verknüpfungen nicht erkannt werden und später zu Fehleinschätzungen führen, die dann vielleicht *ungünstige* Informationen produzieren.

Wahrscheinlichkeiten: Es wird gleich vorgegangen, wie bei der Darstellung mittels **Logik**. Es wird aber zusätzlich die Wahrscheinlichkeit gespeichert, ob die Information korrekt ist.

Die Wahrscheinlichkeiten können im Laufe der Zeit dynamisch angepasst werden, um Fehler beim Lernen zu erkennen und die entsprechenden *Wissensgebiete* als fehlerhaft zu markieren.

Im Prinzip sind die Probleme immer noch die gleichen wie bei der rein logikbasierten Darstellung: Das gesammelte Wissen neigt immer noch dazu mit der Zeit zu verfallen. Es ist also entsprechendes positives Feedback notwendig, um diesem Verfall entgegen zu steuern. Meist wird dazu einfach erworbenes Wissen mit der Zeit wieder gelöscht.

neuronale Netze: Sie sind eine ganz andere Art Wissen zu speichern. Die Bezeichnung soll zeigen, dass es sich hierbei um einen Versuch handelt, natürliches Lernverhalten nachzustellen.

Die Idee zu neuronalen Netzen kam früh, doch ein Problem, welches immer noch nicht gelöst wurde, und auch in naher Zukunft nicht gelöst sein wird ist, dass neuronale Netze vollständig parallel arbeiten. Je mehr Wissen gespeichert werden soll, desto größer muss das neuronale Netz sein, was den Rechenaufwand dramatisch erhöht.

4 Lösungsstrategie

4.1 GPS–Navigation

Zur Positionsbestimmung wird in NAVI [14] **GPS** [38] [1] als Navigationssystem verwendet.

Das GPS–Satelliten–System ist zur Zeit das funktionsfähigste und am weitesten verbreitete Positionsbestimmungssystem für Navigationshilfen [2]. Deshalb wird auch dieses zu entwickelnde System diese Technologie benutzen.

Die Auflösung beträgt ca. 6–30m, dies ist für unsere Zwecke gerade ausreichend.

Aus dieser Einschränkung heraus wollen wir uns beim Navigationssystem damit begnügen, die Position ungefähr festzustellen. Dabei gehen wir davon aus, dass blinde und sehbehinderte Menschen wie Sehende ohne größere Probleme in der Lage sind, einem Straßenverlauf selbstständig zu folgen.

Da GPS kontinuierlich Positionsdaten liefert, kann aus diesen die Geschwindigkeit und Richtung der Bewegung abgelesen werden.

Falls in Zukunft bessere Positionsbestimmungssystem benutzbar werden, z.B. UMTS [23], Galileo [6] etc., kann das System darauf angepasst werden, um vielleicht von der höheren Auflösung zu profitieren.

Datendienste wie WLAN oder UMTS werden es in naher Zukunft außerdem ermöglichen, die Auflösung von GPS auf ca. 1–2m zu verbessern (DGPS) [3].

Probleme von GPS

GPS beruht auf einem Netz von 24 Satelliten. Jeder Satellit sendet seine Kennung und ein Zeitsignal. Anhand dieser Informationen (der Verzögerung) kann man die aktuelle Position berechnen (Längen–/Breitengrad und die Höhe).

Es gibt mehrere Einflüsse welche sich auf die Qualität von GPS auswirken. Das System ist **Wetter** [36] abhängig, bei Bewölkung wird der Empfang schlechter und die Positionsdaten ungenauer (Abweichung bis zu 360cm).

Das Navigationssystem hat lokal keinen Referenzpunkt zur Verfügung, ist also auf die Positionsdaten von GPS angewiesen und deren Aussage über Toleranz. Wie sich die angegebene Toleranz auf die aktuellen Daten auswirkt, ist für das Endgerät nicht ermittelbar (z.B. könnten alle Koordinaten 3 Meter nach Osten verschoben sein).

[1] GPS = Global Positioning System
[2] Mit weit verbreitet sind GPS–Empfänger gemeint. GPS als solches ist weltweit verfügbar.
[3] DPGS = Differential GPS

Die Sendefrequenz von GPS ist recht hoch (1.2–1.5GHz) [38] wodurch **Reflexionen** von Häuserfassaden zum Problem werden. Enge Strassen sind auch schlecht, da hier die Sicht auf die Satelliten teilweise verdeckt wird, und so die Genauigkeit sinkt.

4.2 Taktile Ausgabe

SB–Personen ersetzen das Fehlen des Seh–Sinnes zum größten Teil durch das Gehör, weshalb bei der "Führung" zum gewünschten Zielpunkt akustische Richtungskommandos wie "rechts", "links" usw. nur ungünstig durch Sprache ausgegeben werden können. Der Benutzer soll seine Aufmerksamkeit, also sein Gehör auf Dinge richten können, welche für Ihn unter Umständen lebensnotwendig sind (z.B. Baustellen, Autos, Fahrradfahrer, . . .) [29] [32].

Diesen Ausgabepart sollen deshalb "Tabs" (kleine Vibratoren) an den Armen, wie man sie z.B. von Handys kennt, übernehmen. Soll also ein Abbiegen erzwungen werden, so meldet dies der entsprechende Tab durch eine Vibration an den Benutzer.

Näheres zu dieser Art der Benutzernavigation kann in der zweiten Arbeit zum System NAVI [14] von Günther Harrasser [35] nachgelesen werden.

Abbildung 4.1: Vibratoren, Taktile–Ausgabegeräte

4.3 Karten, Routenplanung

Ein wichtiger Teil in meiner Arbeit ist die Umwelt und ihre Darstellung in Datenbanken und Karten.

Karten können, wie schon im Abschnitt 2.2 erwähnt als Schichtenmodell angesehen werden.

4.3.1 Meta–Karte

Die Meta–Karte beschäftigt sich mit Informationen, welche nichts mit der Routenfindung zu tun haben. Hier finden sich hauptsächlich Informationen zu speziellen Punkten auf der Karte.

In dieser Arbeit werde ich noch öfter auf die **Meta–Daten** zurückkommen, zunächst hier eine Erklärung, was man sich darunter vorstellen kann.

Es gibt in einer Straßenkarte, wie man sie gedruckt kennt, eine Menge an Informationen:

- Straßennamen
- Art der Straße, Autobahn, Bundesstraße, Einbahnstraße, ...
- Brücken
- wichtige Gebäude
- etc. (Bushaltestellen, U–Bahn,...)

Abb. 4.2, S. 24 zeigt einen kleinen Auszug einer Karte von München Schwabing um zu veranschaulichen, wie viele Informationen in der Karte eingezeichnet wurden. Was zu sehen ist, sind Strassen und öffentliche Verkehrsmittel, Gebäude usw., aber in der Karte fehlen alle Hinweise auf Geschäfte, Restaurants, etc.

Dem Benutzer soll nun die Umgebung, welche er ja nicht sehen kann, durch die Meta–Daten näher gebracht werden. Wir benutzen Sprachausgabe, um den Benutzer über alle Gebäude und andere interessante Informationen zu informieren. Sobald die Karte mit Meta–Informationen ergänzt wurde, kann man sehr schnell feststellen, dass es für den Benutzer viel zu erfahren gibt.

Datenmodell

Der Begriff “Meta–Daten” ist sehr abstrakt beinhaltet aber auch den Hinweis, dass es sich um eine Vielzahl von verschiedenen Informationen handelt welche verarbeitet werden sollen.

Informationen die in NAVI [14] aufbereitet werden können (Abb. 4.4, S. 26) müssen folgende zwei Eigenschaften besitzen:

Koordinaten: Die Position der Information ist wichtig. Da das gewählte Datenmodell es erlaubt, wird nicht zwischen Straßenknotenpunkten und Informationknoten unterschieden. Es können Meta–Daten an alle Knoten der Karte (Abb. 4.3, S. 25) angefügt werden.

Abbildung 4.2: Meta Karte, München Schwabing

Ausdehnung: Das System muss feststellen können, ab wann es sinnvoll ist dem Benutzer die Information zugänglich zu machen. Der Radius könnte auch anhand der Bewegungsgeschwindigkeit skaliert werden. Je schneller sich der Benutzer bewegt, desto größer der Radius.

Alle weiteren Informationen sind vom aktuellen Lernmodell abhängig. Dieses wird unter Abschnitt 4.4.2 näher behandelt.

4.3.2 Straßenkarte

Eine geographische Straßenkarte besteht im wesentlichen aus **Strassen, Kreuzungen, Plätze** [26].

Kreuzungen und Plätze sind im allgemeinen Verbindungspunkte von Straßen (ein Platz kann jedoch selbst eine größere Ausdehung haben) und besitzen eine geographische Position (Längen-/Breitengrad).

Sie besitzen zusätzlich auch eine Reihe von interessanten Meta–Informationen. Größe (Anzahl der Spuren), Art (Einbahn, Radweg, etc.), Länge und Name der Straße sind für Benutzer in erster Linie von Interesse.

Mathematisches Graphenmodell

Straßen sind im allgemeinen nicht **gerade** sondern gekrümmt und geknickt. Dies wird im Modell dargestellt, indem die Straße in Abschnitte unterteilt wird. Auf Kanten und Knoten reduziert, kann dann eine Straßenkarte als Graph abgebildet werden.

Definition 4.3.1 *Die Straßenkarte ist ein ungerichteter Graph aus Knoten und Kanten,* $G = (V, E, w)$ *(*V *= Ecken,*E *= Kanten und einer Kantengewichtsfunktion* $w : E \rightarrow R+$*).*

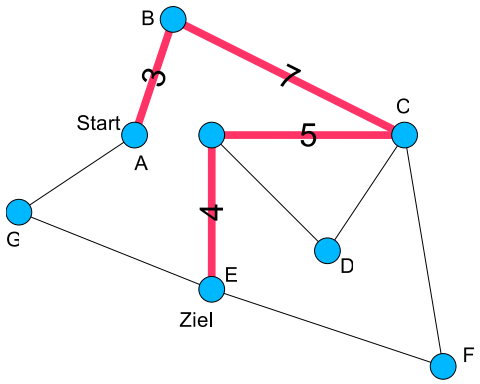


Abbildung 4.3: Straßenkarte, Graphenmodell

In unserem Falle entsprechen **Plätze** und **Kreuzungen** den **Knoten**, die **Strassen** den **Kanten** und die **Gewichtsfunktion** bildet die **Länge** ab.

Diese ganze Datenstruktur wurde in ein SQL–Schema (SQLite [20]) übernommen. Dies ermöglicht es, ohne den Sourcecode später nocheinmal zu ändern, beliebig komplexe Straßengebilde zu sichern (Abb. 4.4, S. 26).

Unser benutztes Datenmodell erlaubt auch, nicht planare Straßenkarten abzubilden. Dies ist im ersten Augenblick wenig sinnvoll, erlaubt es aber auch U–Bahnen und andere Verkehrsmittel in die Karte zu übernehmen, die an keinen Straßenverlauf gebunden sind.

Diese Informationen ermöglichen es, das **Navigationsproblem** [35] zu lösen und einen Routenplaner zu bauen. Im Projekt NAVI [14] ist dies mittels der Boost Library [1], und dem Algorithmus von Djsktra [31] durchgeführt worden (siehe 5.2).

4.4 Informationsfilter

Um dem Benutzer die Umgebung zu erklären (siehe 2.2) sind viele einzelne Informationen notwendig. Wie aus den Anforderungen hervor geht, soll das Gehör des Benutzers nicht zu sehr belastet werden, deshalb soll die Informationsmenge, die an den Benutzer weiter gegeben wird, verringert werden.

Eine Filterung nach Kategorien ist eine Möglichkeit, für SB–Personen jedoch nicht optimal. Um gut nach Kategorien filtern zu können, müssen zuerst manuell Filter erstellt werden. Dies wird ohne visuelle Ausgabe bei einer großen Anzahl von Filtern sehr schnell zu einem Problem [35].

Um nicht verschiedene Filterregeln aufstellen zu müssen, versucht NAVI [14] den Informationsfilter an die Bedürfnisse des Benutzer anzupassen. Dazu muss der Benutzer dem System mitteilen wo seine **Interessen** liegen.

Dies geschieht während der Navigation. Sobald eine Information an den Benutzer weitergegeben wurde, kann dieser dem System mitteilen, ob die Information **sinnvoll** war, oder **nicht**. Aus diesem Feedback wird ein lernender Filter generiert.

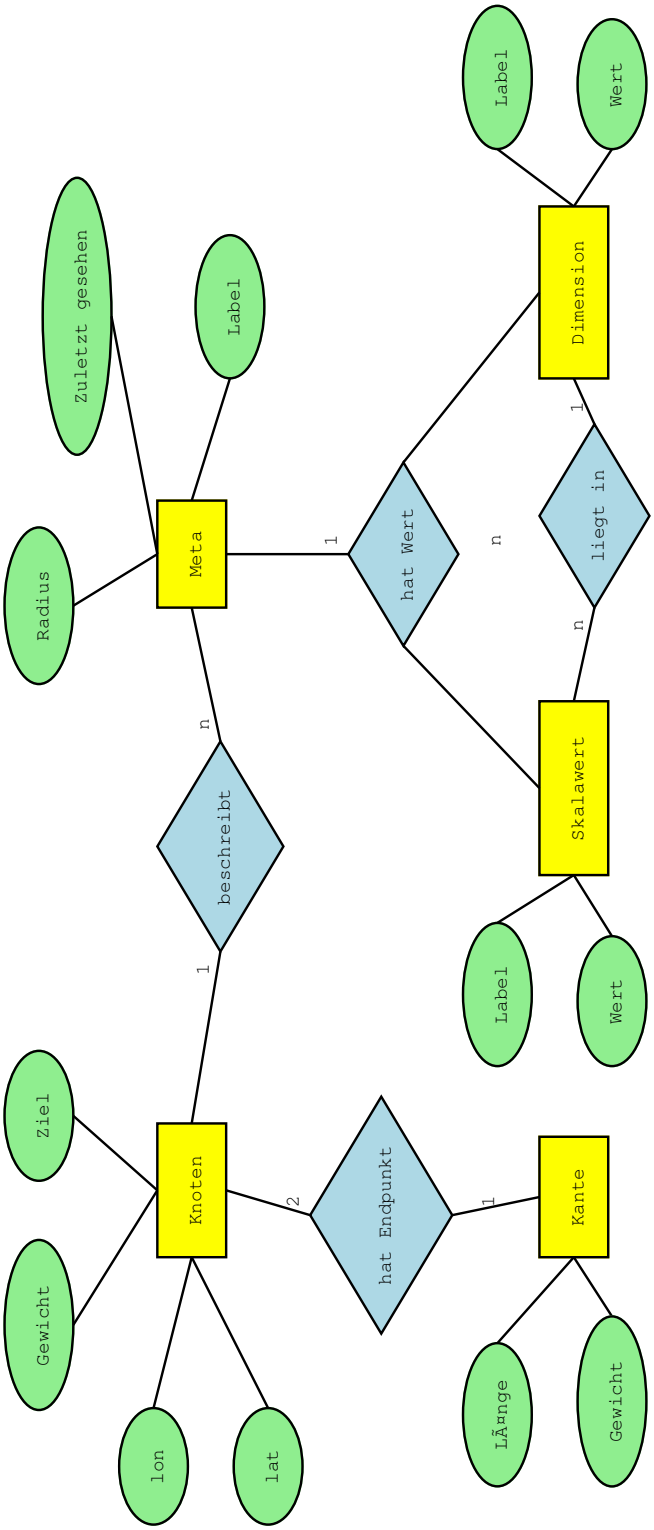


Abbildung 4.4: NAVI [14] Sql Schema

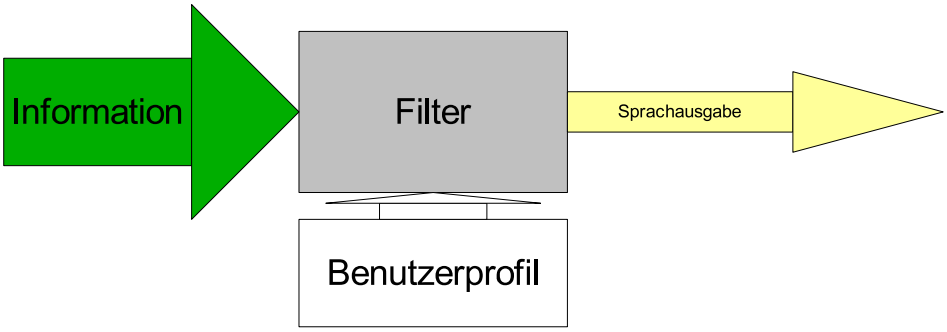


Abbildung 4.5: Filter, Funktionsweise

4.4.1 Vermeidung von Wiederholungen

Erstes Problem bei der praktischen Realisierung ist die Geschwindigkeit mit der sich der Benutzer bewegt. Er bewegt sich zu langsam. Kommt er in die Nähe eines Wegpunktes [4], wird ein ein Hinweis zu diesem Wegpunkt ausgegeben. Dann noch einer und noch einer ...bis er sich wieder weit genug vom Wegpunkt entfernt hat. Falls er inzwischen in den näheren Bereich eines oder mehrerer anderer Wegpunkte gekommen ist, wiederholt sich das Problem.

Lösung hierfür ist die Einführung einer Positions–Hysterese (Abb. 4.6, S. 28). Erst wenn der Benutzer wirklich in der Nähe des Ziels war, und dann wieder genug Abstand gewonnen hat, soll das Ereignis nocheinmal ausgegeben werden.

Ähnliches kann man sich für den zeitlichen Verlauf überlegen.

Konkret gelöst wurde das Problem bei NAVI [14] jedoch anders. Ich gehe davon aus, dass der Benutzer rein zeitlich an den Informationen seiner Umgebung interessiert ist. Der Benutzer weiß im Normalfall, wo er sich gerade befindet und will Informationen über seine Umgebung in zeitlich groß genug dimensionierten Abständen immer wieder hören. Dazu wird direkt im StreetMap–Service, (siehe 5.2.1) vor dem Weiterleiten der Information an den Filter–Service in der Datenbank ein Zeitstempel aktualisiert.

Weitergeleitet werden nur Informationen die ein gewisses **Alter** erreicht haben.

4.4.2 Klassifizierung der Eingabe

Die beiden Lernverfahren (Bayes, SVM), welche beim Projekt NAVI [14] zum Einsatz kommen können, sind Verfahren, welche eine Eingabe klassifizieren, also einer Zielmenge zuweisen. In unserem Fall beschränken wir uns dabei auf zwei Zielmengen: **Ja** und **Nein**.

Der Ablauf im allgemeinen ist folgender:

1. erhalte Eingabevektor
2. klassifiziere Eingabevektor

[4] Wegpunkte sind Stellen auf der Karte, zu denen es Meta–Daten gibt.

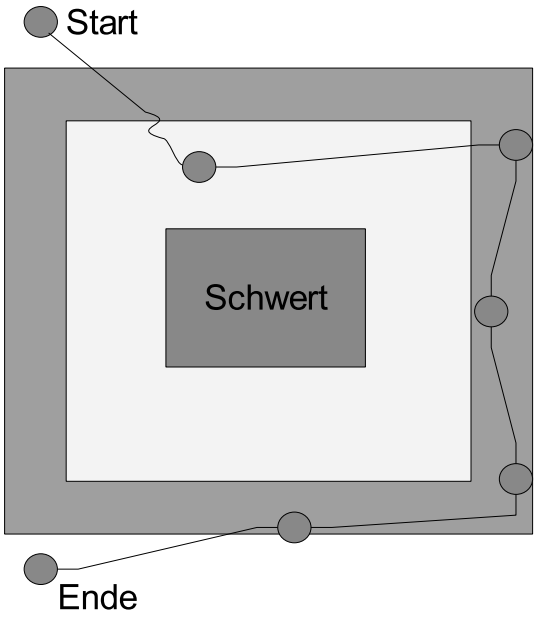


Abbildung 4.6: Szenario – Positions–Hysterese

3. korrigiere das Ergebnis

Ein Eingabevektor ist eine Zusammenfassung der Eingabe zu einem Vektor [5].

Mehr Eingabedimensionen (Features) bedeuten eine genauere Bestimmung der Eingabedaten. Bei der Wahl dieser muss bei den meisten Verfahren darauf geachtet werden, dass es nicht zu Widersprüchen kommt. Überschneidungen der Wertemengen sind ebenfalls problematisch [6].

Beispiel 3 *Es soll Geld klassifiziert werden, also ermittelt, ob die Währung Euro ist. Dabei beschränken wir uns auf die Münzen. Als Eingabedimensionen (Features) wählen wir:*

- *Durchmesser*
- *Farbe*
- *Gewicht*
- *Dicke*

Für jede Dimension kann ein Wert angegeben werden. Aus diesen ergibt sich der Vektor.

$$z.B.\ euro = \begin{pmatrix} 2cm \\ Weißgold \\ 10g \\ 1.5mm \end{pmatrix}$$

[5] Die verschiedenen Dimensionen des Eingabevektors sind in der engl. Literatur als *Features* bekannt

[6] Durch Überschneidungen der Wertemengen von Klassifizierungsdimensionen werden Abhängigkeiten zwischen diesen hergestellt, dies stellt eine implizite Reduzierung [37] der Anzahl der Dimensionen dar.

Definition 4.4.1 *Alle Vektoren haben während des gesamten Lernverfahrens eine fixe Anzahl von Dimensionen.*
$\mathbf{x} \in \mathbf{N}^n, n\ Anzahl\ der\ Dimensionen$
Im allgemeinen können alle Dimensionen verarbeitet werden, für die wir eine bijektive Funktion finden, welche die Eingabedimensionen auf die Menge $\mathbf{N}$ *abbildet.*
$i \in 1...n, f_i(a) : \mathbf{A} \rightarrow \mathbf{N}$

Jeder Eingabevektor wird im Laufe des Verfahrens einer der Zielmengen zugeordnet. Eine externe Instanz, meist ein Benutzer, muss im Anschluss dem System Feedback geben und diese Zuordnung im Zweifelsfalle korrigieren.

4.4.3 Lernalgorithmen

Bayes

Grundlage für das Bayes Lernverfahren ist das 1763 veröffentlichte Theorem von Thomas Bayes. Beim Lernen nach Bayes geht es in unserem Fall darum, zwei Gruppen zu klassifizieren. Hier eine kurze auf die spezielle Anwendungsweise bezogene Einführung in das Theorem.

Es geht um das Zufallsereignis X und ob es eintreten wird oder nicht.

- Herr A, der in 75% aller Fälle recht hat, meint **Nein**.
- Herr B, der in 60% aller Fälle recht hat meint **Ja**.

Mit welcher Wahrscheinlichkeit wird das Ereignis X eintreten?

Wie man schnell bemerkt, ist das Problem unterbestimmt, in nachfolgender Tabelle steht Spalte **A** für Mann A, Spalte **B** für Mann B, Spalte **R** für den tatsächlichen Ausgang.

A	B	R	$P(X)$
n	n	n	p_0
n	n	j	p_1
n	j	n	p_2
n	j	j	p_3
j	n	n	p_4
j	n	j	p_5
j	j	n	p_6
j	j	j	p_7

Hieraus kann man sofort drei Gleichungen ablesen:

1. Die Summe aller Ereignisse ist 1.

$$\sum_{i=0}^{7} p_i = 1$$

2. Aus der Aussage von Mann A folgt:

$$p_0 + p_1 + p_2 + p_3 = 0.75 = u$$

(u steht in den weiteren Berechnungen für die Glaubwürdigkeit von Mann A, v für die von Mann B)

3. Aus der Aussage von Mann B folgt:

$$p_2 + p_3 + p_6 + p_7 = 0.60 = v$$

Man kann diese Gleichungen umschreiben, dann wird es deutlicher, dass sie unterbestimmt sind. Es werden folgende Variablen gewählt:

$$A = p_0 + p_7, B = p1_+p_6, C = p_2 + p_5, D = p_3 + p_4$$

$$\begin{aligned} A + B + C + D &= 1.00 \\ A + C &= 0.75 = u \\ A + D &= 0.6 = v \end{aligned}$$

Wir nehmen an, dass die beiden Ereignisse (die Schätzungen von Mann A und Mann B) stochastisch unabhängig voneinander sind.

Des weiteren ordnen wir die Verteilung von **Ja** und **Nein** der Variable x zu, falls **Ja** und **Nein** gleichverteilt sind, ergibt sich für $x = 0.5$.

Dadurch kann obiges unterbestimmtes Gleichungssystem gelöst werden, und es entsteht:

$$P(X) = \frac{u(1-v)x}{(1-u)v(1-x) + u(1-v)x} \tag{4.1}$$

Allgemeiner gilt: Falls sich N Leute sicher sind dass sowohl das Ereignis X eintreffen wird, und dazu ihre Glaubwürdigkeit mit r_n angegeben wird, als auch die Ereignisse stochastisch unabhängig sind und das Ereignis gleichverteilt ist, ergibt sich folgendes:

$$P(X) = \frac{(r_1)(r_2)\dots(r_N)}{(r_1)(r_2)\dots(r_N) + (1-r_1)(1-r_2)\dots(1-r_N)} \tag{4.2}$$

Falls man den Faktor x explizit in die Gleichung aufnehmen möchte, entsteht:

$$P(X) = \frac{x(r_1)(r_2)\dots(r_N)}{x(r_1)(r_2)\dots(r_N) + (1-x)(1-r_1)(1-r_2)\dots(1-r_N)} \tag{4.3}$$

Diese Formel kann benutzt werden, um Wissen bzw. unsichere Eingaben mit einer Glaubwürdigkeitsabschätzung zu erweitern und dann auf die *Gesamtglaubwürdigkeit* zu schließen.

Anwendung von Bayes

Nun zurück zu unseren Eingabevektoren (siehe 4.4.2) und Dimensionen. Wir bilden zwei gleichgroße Hashes für unsere Eingaben und ordnen unsere Dimensionen je einem Bucket zu.

Jede neue Eingabe wird dem einen oder anderen Hash zugeordnet. Indem man auszählt, wie sich die Eingabe auf die Buckets verteilt ist es möglich für jede Dimension eine Wahrscheinlichkeit anzugeben. Diese sind mittels der Bayes–Formel 4.3 zu kombinieren und eine Gesamtwahrscheinlichkeit zu errechnen.

SVM

Support Vector Maschinen [42] [28] [30] sind ein recht junges Hilfsmittel, um lernende Maschinen zu erstellen.

Grundsätzliche Überlegung ist es, dass die Vektoren, mit denen die Maschine arbeitet, geometrisch interpretiert werden [7]. Sie werden also räumlich aufgefasst: Jeder Vektor bestimmt einen Punkt im Raum und besitzt eine Marke, welche die zugehörige Menge angibt.

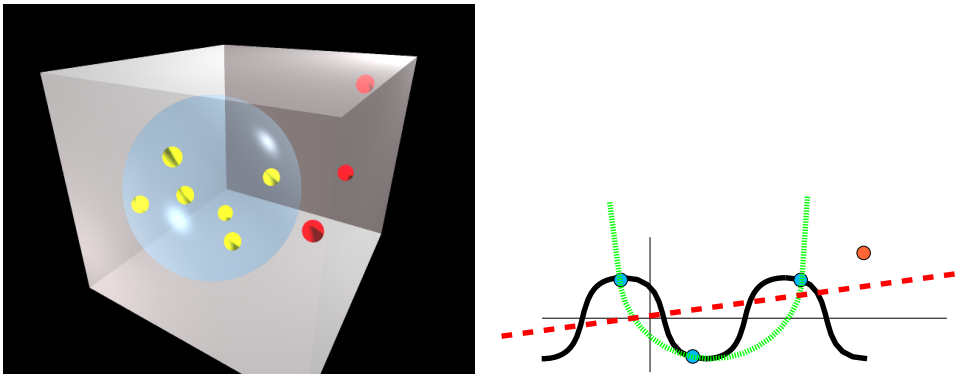
Ideal wäre es nun, wenn sich alle Vektoren, die zu einer Menge gehören, an einer Stelle im Raum sammeln, sodass man diese **einfangen** kann (z.B. mit einer Kugel wie in Abb. 4.7(a), S. 32).

Falls nun die Anordnung im Raum komplizierter wird, könnte man aufwendigere geometrische Gebilde benutzen, um die Mengen zu klassifizieren. Allerdings ist die Lernqualität dann keineswegs besser. Abb. 4.7(b), S. 32 soll dies veranschaulichen. Es ist einfach, eine Funktion zu finden, welche durch alle blauen Punkte verläuft. Nur die Gerade würde in diesem Beispiel mehr über den roten Punkt aussagen, als die Kurve. Höhere Komplexität muss nicht das Lernverhalten verbessern.

In der Praxis [30] wird deswegen ein anderer Weg beschritten. Zum einen werden Fehler bewusst erlaubt, zum anderen wird nur eine Ebene benutzt um den Raum zu teilen (unter der Annahme, dass es nur zwei Mengen zu klassifizieren gibt).

Es wird zuerst die Maschine wie gewohnt trainiert, also die Schnittebene ermittelt. Im Anschluss kann dann klassifiziert werden. Wie man sich überlegen kann, ist das Lernen recht aufwendig, das Klassifizieren aber billig.

[7] Die geometrische Auffassung ist nur für das einfache Gedankenmodell sinnvoll. Praktisch werden wesentlich mehr als 3 Dimensionen verwendet.



(a) einfache Veranschaulichung (b) Komplexitäts Problem

Abbildung 4.7: SVM

4.4.4 Unterschiede beim Klassifizieren

Support Vector Maschinen und Algorithmen mit **Bayes** arbeiten beide mit Eingabevektoren.

Es ist nicht möglich, die gleichen Eingaben auf beide Maschinen anzuwenden und dabei sinnvolle Ergebnisse zu erreichen. Die Dimensionen (Features) und deren Beschaffenheit sind in beiden Fällen komplett unterschiedlich.

Bayes

Wie schon beschrieben, beruhen Algorithmen, die nach Bayes (siehe 4.4.3) funktionieren, auf der Wahrscheinlichkeitsrechnung.

Ausgabe der Berechnung ist stets die Wahrscheinlichkeit, dass die gewünschte Frage zutreffend ist oder nicht. Die Anzahl der Dimensionen geht in die Berechnung nicht ein. Ebenso ist es unerheblich, wie die Dimensionen zueinander stehen. Für die Berechnung ist nur das Auftreten eines Wertes erforderlich und wie oft er zuvor klassifiziert wurde.

Anders ausgedrückt: das Verfahren benötigt keine **Ordnung** im mathematischem Sinne auf den Eingabedaten, sondern nur **Häufungen**.

SVM

Support Vector Maschinen arbeiten in einem N–dimensionalen Raum, welcher durch verschiedene Achsen aufgespannt wird.

Aufgrund dieser Voraussetzung ergibt sich für jede Achse ein "**Kleiner**" und ein "**Größer**". Die Werte innerhalb einer Dimension benötigen eine mathematische Ordnung, ansonsten kann eine SVM nicht funktionierten.

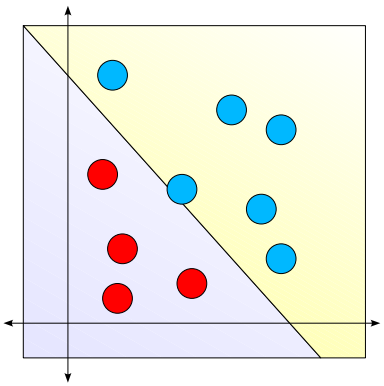


Abbildung 4.8: SVM – Klassifizierung in zwei Dimensionen

Abb. 4.8, S. 33 stellt einen Schnitt dar, wie er zwischen jeder Dimensionenkonstellation machbar ist. Der Schnitt wurde zuvor berechnet und teilt die Eingabewerte **immer** in zwei Mengen.

Bei NAVI [14] tritt nun das Problem auf, dass z.B. als Eingabevektor die Daten einer Bank klassifiziert werden sollen. Nun möchte man im Benutzerprofil jedoch auch die Essgewohnheiten ermitteln und hat dafür eine Dimension im Eingabevektor vorgesehen. Die Dimension Essen soll die X–Achse in Abb. 4.8, S. 33 sein, welchen Wert auf dieser Achse wähle ich für die Bank? (siehe 6)

5 Konkrete Umsetzung

Nun zur konkreten Umsetzung von NAVI [14] : Unser Ziel war es einen Prototypen zu bauen, welcher leicht erweiterbar ist und einfach zu warten.

Ein Problem bei der Implementierung war die Vielzahl von Bibliotheken, welche wir zum Einsatz brachten, SQLite, Boost, Joystick, ...

Bei der eigentlichen Implementierung haben wir viele Ziele gut erreicht. Dabei haben wir auf dem Framework DWARF [27] aufgebaut, welches eine zusätzliche Abstraktionsschicht von CORBA [15] darstellt.

5.1 Bibliotheken, Framework

5.1.1 CORBA

CORBA war schon zu Anfang für die Entwicklung des Projektes Voraussetzung, da es die Grundlage für das Framework DWARF [27] darstellt. CORBA steht für Common Object Request Broker Architecture und wurde 1991 von der Object Management Group (OMG) [15] eingeführt.

Es ist in erster Linie ein Modell zur objektorientierten Programmierung. Spezieller Wert wurde auf folgende Punkte gelegt:

- Plattformunabhängigkeit (Corba ist für Windows, Unix, Linux, ... verfügbar)
- Netzwerktransparenz
- Programmiersprachenunabhängigkeit

Normalerweise muss der Programmierer darauf achten, welches Betriebssystem mit welcher Programmiersprache auf welcher Hardwareplattform zum Einsatz kommt. Falls das Programm auf mehreren Rechnern funktionieren soll, muss das der Programmierer selbst koordinieren und die Netzwerkebene selbst realisieren.

CORBA stellt Schnittstellen zur Verfügung, um diese Probleme zu lösen. Es arbeitet streng objektorientiert. So wird jedem Objekt ein eindeutiger Identifikator zugeordnet. Die verschiedenen Programmteile können anhand dieses Identifikators netzwerk– und plattform transparent immer auf das Objekt zugreifen.

Um die Schnittstellen einheitlich zu formulieren, wurde eine eigene Sprache geschaffen: IDL (Interface Definition Language), sie kann verschieden übersetzt werden, um das Interface in unterschiedlichen Programmiersprachen äquivalent zu repräsentieren.

Die Eigenschaften von CORBA sind ideal, um komplexe, verteile Programme zu entwickeln.

5.1.2 DWARF

DWARF [27] steht für Distributed Wearable Augmented Reality Framework und wurde an der TU–München als grundlegendes Framework entwickelt um Augmented Reality Applikationen einfacher umzusetzen. DWARF [27] stellt dem Anwender eine auf Corba basierende Middleware zur Verfügung.

Zum Konzept von DWARF [27] gehört es, unabhängige Programmteile (Services) zu erstellen, welche mit einer möglichst allgemeinen aber einfachen Schnittstelle ausgestattet sind. Daten werden über ein Nachrichten (Events) basiertes Kommunikationssystem vermittelt. Dienste können Nachrichten erstellen, welche an einen oder mehrere andere Services adressiert sind. DWARF [27] übernimmt die korrekte und effiziente Zustellung dieser.

Das Verschalten oder anders formuliert das Auffinden eines geeigneten Empfängers für eine Nachricht übernimmt ebenfalls DWARF [27] zur Laufzeit. Services kommunizieren über IP basierte Dienste untereinander und müssen weder auf dem gleichen Rechner ausgeführt werden noch auf dem gleichen Betriebssystem aufsetzen.

Ziel des Frameworks ist es, für möglichst viele allgemeine Nachrichtentypen Services zu entwickeln, die dank DWARF [27] und CORBA mit verschiedenen Programmiersprachen entwickelt wurden und wiederverwendbar sind.

Servicemanager

Pro Rechner [1] wird ein **Servicemanager** benötigt. Dieser ist zentraler Bestandteil von DWARF [27] . Wenn ein Service gestartet wird, meldet sich dieser am Servicemanager an, dadurch registriert der Servicemanager den Servicenamen, seine **Needs** und seine **Abilities**. Andererseits kann der Servicemanager Services auch bei Bedarf automatisch starten (falls der Service dafür entsprechend konfiguriert wurde).

Services

Services sind eigenständige Programme, welche über ihre **Needs** und **Abilities** mit den anderen DWARF [27] –Komponenten kommunizieren.

Abilities: Abilities sind Datenschnittstellen, welche vom Service zur Verfügung gestellt werden. Abilities besitzen einen

Typ , welcher einen IDL–Datentypen referenziert, eines oder mehrere

Attribute , welche die Semantik des Datentypen für die aktuelle Anwendung genauer spezifizieren und einen

Namen , welcher eine möglichst aussagekräftige Beschreibung des Interfaces darstellen sollte.

[1] DWARF [27] setzt auf den slpd [16] auf, welcher im Normalfall die Netzwerkkommunikation mit Multicast abwickelt.

Needs: Ein **Need** ist eine eingehende Schnittstelle. Die Daten für diese Schnittstelle werden von anderen Services über deren Abilities zur Verfügung gestellt. Needs bestehen aus einem

Typ , welcher einen IDL–Datentypen referenziert, einem

Prädikat , welches die Semantik des angeforderten Datentyps genauer spezifizieren und einem

Namen , welcher eine möglichst aussagekräftige Beschreibung des Interfaces darstellen sollte.

DWARF [27] beherrscht mehrere Kommunikationsprotokolle und Verschaltungstypen, in NAVI [14] setzen wir auf rein eventbasierte Kommunikation (Sender–Verbraucher Prinzip).

Der Servicemanager versucht nach dem Start der Services jedem **Need** eine **Ability** zuzuordnen. Eine Verknüpfung (Verschaltung) zwischen Need und Ability zweier Services findet statt, falls deren Typ, Attribut und Prädikate einander entsprechen. Die Verschaltung kann sich zur Laufzeit dynamisch ändern/anpassen, da neue Services gestartet bzw. laufende Services gestoppt werden können.

5.1.3 Torch

Die Bibliothek TORCH ist in der dritten Überarbeitung verfügbar [21]. Das Ziel der Entwickler war es, eine möglichst einfach zu benutzende Umgebung zu schaffen, um maschinelles Lernen umzusetzen.

Auf die spezielle Anwendung in NAVI [14] gehe ich im Abschnitt 5.2.2 ein, hier nur eine Einstimmung in die Möglichkeiten von TORCH im Zusammenhang mit Support Vector Machines (SVM). Wer sich mit der Bibliothek näher beschäftigen will, kommt um das HowTo (siehe 5.2.2) der Torch–Entwickler nicht herum. Dies ist aber recht knapp gehalten, deswegen hier noch ein paar NAVI–spezifische Ergänzungen.

Aufbau von TORCH

Die Bibliothek unterteilt maschinelles Lernen in folgende Punkte:

Eingabedaten: Hierunter werden die Daten verstanden, welche die WB–Maschine [2] klassifizieren soll.

Trainingsdaten: Trainingsdaten bestehen aus zwei Teilen: dem Eingabevektor (siehe 4.4.2) und dem Trainingsziel (welcher Klasse der Eingabevektor angehören soll). Diese werden der WB–Maschine durch einen passenden Trainer beigebracht. Eine trainierte Maschine kann Eingabedaten klassifizieren.

Trainer: Dieser erstellt anhand der Trainingsdaten eine WB–Maschine welche mit diesen Daten arbeitet. Der Vorgang des Lernens ist mit den meisten Algorithmen sehr zeitaufwendig (NAVI verwendet eine SVM [3], diese hat eine Lernkomplexität von $O(n^3)$).

Measurer: Ähnlich wie der Trainer ist der Measurer WB–Maschinen spezifisch, dieser kann alle möglichen Eigenschaften der WB–Maschine messen (Fehler, Anzahl der Klassen, etc.). Der Trainer benutzt während des Trainings einen oder mehrere Measurer um feststellen zu können, ob der Trainingsvorgang abgeschlossen werden soll.

Das Erstellen einer lernenden Maschine erfolgt, indem man obige Teile initialisiert und nacheinander aufruft.

1. Zuerst werden Trainingsdatensätze benötigt. Diese werden in einem DataSet abgelegt, dann für die Maschine angepasst.
2. Nun kann ein Trainer initialisiert werden, diesem werden die künftigen Eigenschaften der Maschine mitgeteilt.
3. Nun wird mit Hilfe des Trainers die eigentliche Maschine ins Leben gerufen. Der Trainer benutzt die zuvor aufbereiteten Eingabedaten, um die Maschine entsprechend zu trimmen.

[2] WB–Maschine = wissensbasierte Maschine
[3] Support Vector Maschine

4. Es können nun Eingabedaten der Maschine übergeben werden, die sie dann klassifiziert. Es obliegt dem Benutzer, Schlussfolgerungen daraus zu ziehen.

Obige Schritte können natürlich kombiniert werden. So ist es mit der Bibliothek TORCH ohne Probleme möglich, mehrere Lernmaschinen mit verschiedenen Parametern parallel zu initialisieren. Die Auswertung der Ergebnisse bleibt trotzdem dem Benutzer überlassen (siehe 3.1.3).

Sequence

Sequences in Torch entsprechen dem, was hier in der Arbeit als Eingabevektor bezeichnet wird (siehe 4.4.2). Technisch gesehen stellen sie ein Array aus real–Zahlen [4] mit zwei Dimensionen dar.

DataSet

Eine sehr wichtige Datenstruktur in Torch sind DataSets. Sie beinhalten alle Daten, welche in den verschiedenen Arbeitsschritten verarbeitet werden, und bestehen im wesentlichen aus zwei Arrays, den Inputs und den Targets (beide vom Typ Sequence).

Ein Fallstrick besteht darin, dass diese Strukturen etwas trickreich zu erstellen sind, und dann für die verschiedenen Lernverfahren gesondert angepasst werden müssen [5].

Anders als im TORCH HowTo vielleicht heraus gelesen wird, liefert das DataSet Objekt nur ein Interface, um auf die Daten zugreifen zu können. Die eigentlichen Daten müssen zuvor mit einer der Schwesterklassen [6] geladen werden. Dabei ist es möglich, die Daten auf der Festplatte, im Speicher oder gemischt zu verwalten.

DataSets liefern die Möglichkeit, einen Zugriff auf Substrukturen zu erlauben. Das heißt, dass es möglich ist, in einem DataSet Datensätze auszublenden oder auch umzusortieren (dies wird mit Hilfe eines Map–Arrays vom Typ Integer durchgeführt). Dies kann öfter hintereinander gemacht werden. Diese Mappings werden in einem Stapel verwaltet, der vom Benutzer selbst wieder "aufgeräumt" werden muss.

Während TORCH–Datenstrukturen im allgemeinen ihre Kontrollstrukturen gesondert im Speicher behalten [7], sollte man sich des weiteren bewusst sein, dass die eigentlichen Nutzdaten (Eingabedaten, Zieldaten, Trainingsdaten) **nie** von TORCH kopiert werden. Sie bleiben auch bei den verschiedenen Lernalgorithmen, die diese verändern, im Speicher erhalten. Der Programmierer muss die Konsistenz der Daten selbst wieder herstellen.

[4] Der Datentyp real wird je nach Prozessorarchitektur als double oder float realisiert. Auf x86 Prozessoren wird mit double gearbeitet.
[5] Die Datenstruktur bleibt gleich, es ändert sich nur die Interpretation.
[6] MemoryDataSet, MatDataSet, DiskDataSet stehen dem Benutzer zur Verfügung um seine Daten zu verwalten.
[7] Falls ein neues Objekt aus einem Alten erstellt wird, werden im neuen Objekt die Zeiger auf die Nutzdaten des alten Objektes wiederverwendet.

5.2 NAVI

Am Ende der Realisierungsphase des Projektes NAVI [14] entstand folgender Prototyp.

Es war keine Zeit mehr, das System auf einen Handheld–Computer zu portieren. NAVI [14] wurde aber mit dem Gedanken erschaffen, später portierbar zu sein.

Derzeitige Hardwareanforderungen [8]:

- GPS–Empfänger, welcher als Ausgabe NMEA–0183 [13] beherrscht.
- Tabs, für Taktile Ausgabe (Abb. 4.1, S. 22)
- USB Joypad
- Notebook 256 MB RAM, Pentium 4 2.4 GHz, mit folgenden Schnittstellen:
  - Serielle Schnittstelle R232 für GPS Empfänger
  - Parallele Schnittstelle (IEEE 1284) für Taktile–Ausgabe
  - USB 1.0/2.x für Joypad–Eingabe
  - Soundkarte für Sprachausgabe

Als Software bringen wir GNU/Linux [7] zum Einsatz. Benötigt wird mindestens die Kernelversion 2.4.15. Wir verwendeten die Distribution **Debian Linux unstable** vom 18.11.2003 mit einem 2.6.0–test9 Kernel.

Bei der Wahl der Bibliotheken haben wir uns entschlossen nur reine OpenSource Produkte zu wählen. Dadurch gewährleisten wir, dass unser Projekt NAVI [14] später weiterentwickelt und vielleicht sogar kommerziell umgesetzt werden kann.

Die Linux Distribution Debian hat eine sehr umfangreiche Softwaresammlung und eine konservative Einstellung zu freien Softwarelizenzen [3]. Dies stellt jedoch sicher, dass keine Unsicherheiten bezüglich der Lizenzen in Zukunft zu erwarten sind.

Vorausgesetzte Software Pakete, welche zum Betrieb erforderlich sind:

Grafische Oberfläche: Als Debug–Plattform für sehende Entwickler, haben wir uns für das Kommon Desktop Environment (KDE) [12] in der Version 3.1.4 entschieden (Abb. 5.3, S. 44). Die Oberfläche ist einfach zu verwenden und durch Entwicklerprogramme wie den QT–Designer [17] können Programmoberflächen schnell generiert werden. KDE beruht auf der Bibliothek QT [18] 3.2.1, von der Firma Trolltech.

Die Softwareteile von NAVI [14] , in denen eine grafische Oberfläche Verwendung findet, sind funktional in eigene Bestandteile ausgegliedert. Dadurch können diese Debug–Ausgaben später ohne große Änderungen entfernt werden [9].

[8] Wir haben mit dieser Hardware das System entwickelt und alle Vorführungen im Rahmen des Projektes mit dieser durchgezogen. Die Bibliotheken sollten portierbar sein und die Prozessor- und Speicheranforderungen um ein Vielfaches niedriger als von uns hier angegeben.

[9] In unserer Grundüberlegung gehen wir davon aus, dass blinde und sehbehinderte Menschen keinen Nutzen aus einer visuellen Oberfläche ziehen, sehr wohl aber sehende Entwickler

Joypad API: Als Eingabegerät haben wir uns für einen Joypad entschieden, da Joypads unter Linux einfach anzusprechen sind, kostengünstig und viele Tasten besitzen. Standardschnittstelle ist die libjsw [24] Version 1.5.0.

Wir haben uns für diese Bibliothek entschieden, da sie uns einen einfachen Zugriff auf die Benutzereingaben ermöglicht. In dem dafür zuständigen Programmteil muss nur an einer Stelle abgefragt werden, ob ein Joypad–Event aufgetreten ist. Wenn das der Fall ist, werden alle gedrückten Knöpfe ausgelesen und somit kann der aktuelle Status der Eingabe verarbeitet werden. Falls kein Joypad verfügbar ist, kann mittels **readline** [8] die Eingabe auch über eine Konsole erfolgen.

Datenbanksystem: Als Datenbanksystem haben wir uns für die speicherschonende Variante SQLite [20] Version 2.8.6 entschieden.

Bei SQLite wird kein externes DBMS [10] benötigt. Die Daten werden in einer einzigen Datei abgelegt, auf die während der Benutzung mit normalen SQL92-Anfragen zugegriffen werden kann. Sie ist für viele kleine Anwendungen durch die resourcenschonende Implementierung ideal. Bezüglich der Geschwindigkeit kann sie mit anderen DBMS vergleichen, aber auch auf Handheld–PCs eingesetzt werden.

Algorithmen: Zur Implementierung der Wegesuch–Algorithmen setzten wir auf die Boost [1] STL Headers Version 1.30.2. Diese implementieren alle gängigen Wegesuchverfahren äußerst effizient.

Bei NAVI [14] werden die aus der Datenbank ausgelesenen Daten in einen Boost-Graph konvertiert. Dieser Graph wird während der Programmausführung zum Berechnen der Navigation im Speicher gehalten. Die Wegesuchalgorithmen bestimmen auf dessen Datenstruktur aufbauend den aktuell kürzesten Pfad zum Ziel.

SVM Lernverfahren: Die Bibliothek Torch3 [21] liefert alles was gebraucht wird, um Lernalgorithmen anzuwenden. Zum Einsatz kam die Version 0.0–2.

Lernverfahren beruhen auf vielen Algorithmen, welche untereinander Daten austauschen. Torch3 liefert diese mit einer gemeinsamen Datenbasis und ermöglicht so das schnelle Austauschen von Subalgorithmen, um so das Lernverhalten optimal auf die Aufgabenstellung abzustimmen.

GPS: Die GPS Daten werden vom GPS–Dämon [9] Version 1.14 aufbereitet und dem Projekt NAVI [14] zur Verfügung gestellt. Der GPS–Dämon wird als einzelne Software nicht mehr weiterentwickelt, ist jedoch als Teilprojekt des Programms GPSDrive [10] verfügbar.

GPS Daten können auf eine Vielzahl von Arten empfangen und aufbereitet werden. Der gpsd liefert uns eine einheitliche Schnittstelle zur Abfrage dieser Daten. Des weiteren ermöglicht er das Generieren von Testläufen, indem er gespeicherte GPS–Daten zur Verfügung [11] stellen kann.

Sprachsynthese: Die kleine Version des Paketes Festival, Festival–Lite [4] ist speziell für den Einsatz mit wenig Ressourcen gedacht und wurde bei uns in der Version 1.2 verwendet.

[10] Datenbank Management System

[11] Die Entwicklung GPS basierter Software ist aufwendig, wenn man für jeden Test einen Feldversuch starten muss.

FLite ist nach unserem Wissensstand derzeit die einzige freie Sprachsynthesebibliothek [12], welche auch auf einem PDA lauffähig ist (kompilierte Größe: 175954 Byte).

5.2.1 Komponentensicht

NAVI [14] besteht aus mehreren Services. Jeder Service besteht aus einem oder auch mehreren Objekten. Zur Kommunikation nutzen wir das DWARF [27] –Framework, welches wiederum selbst auf CORBA basiert. Der Datenaustausch erfolgt mit Hilfe von Events.

Aufgrund der Problemstellung ergaben sich folgende NAVI–Komponenten (Services):

GPSPosition verschickt die aktuelle Position des Benutzers.

StreetMap enthält die Datenbasis und erledigt die Routenplanung.

Filter erstellt ein Benutzerprofil bezüglich der Umgebungsdaten.

JoyPadInput nimmt die Benutzereingabe entgegen.

TactileOutput gibt Richtungsänderungen an den Benutzer mittels Vibration weiter.

SpeechOut gibt Informationen mittels Sprachsynthese an den Benutzer weiter.

Serviceübersicht

Die in Abschnitt 5.2.1 aufgelisteten Services sind über das DWARF–Framework miteinander verschaltet (Abb. 5.2, S. 43). Jeder Service hat dabei seine spezielle Teilaufgabe zu erledigen und wird rein anhand seiner Needs und Abilities ausgewählt, dadurch ist es möglich nicht nur Services durch andere Komponenten auszutauschen, sondern auch Needs und Abilities in andere Komponenten aufzuteilen bzw. umzuverteilen (siehe 5.1.2).

Wie in Abb. 5.2, S. 43 dargestellt ist, benutzt NAVI eine Reihe von Datenstrukturen zur Kommunikation:

string stellt eine Abfolge von ASCII-Zeichen dar. Die Semantik, in der der Datenstrom zu sehen ist, wird über das Semantik–Attribut angegeben:

- **Speech:** Der String enthält Daten in englischer Sprache [13], welche für Sprachausgabe geeignet sind.
- **Vibration:** Die Daten stellen eine einfache Meta-Sprache dar, die in Vibrationsausgabe umgesetzt werden kann.

InputDataString wird bei NAVI für die Übermittlung von Benutzereingaben über das Joypad verwendet. Der in dieser Datenstruktur enthaltene String entspricht dabei den gedrückten Knöpfen auf dem Eingabegerät.

[12] Einschränkung bei der Bibliothek ist leider derzeit die Qualität. Weiters ist die Sprachausgabe aktuell nur auf englische Sprache abgestimmt.

[13] Diese Einschränkung besteht derzeit hauptsächlich, weil die Bibliothek FLite [4] nur englische Sprachausgabe unterstützt.

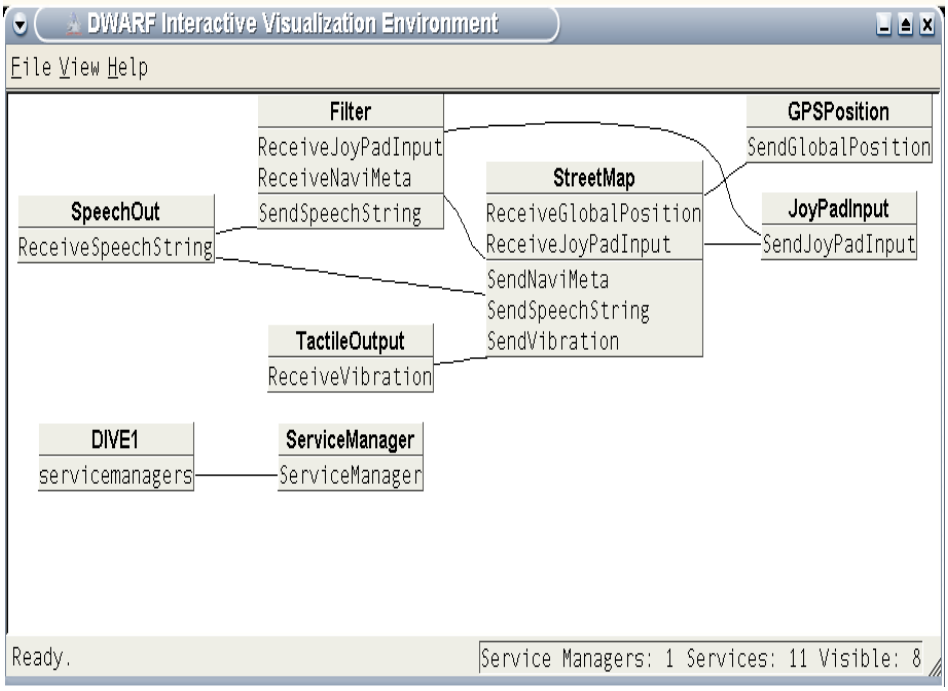


Abbildung 5.1: DWARF – NAVI mit DIVE–Debugger betrachtet

GlobalPosition besteht aus Breiten und Längengraden der aktuellen Position, welche mit Hilfe von GPS ausgewertet wurden. Des weiteren werden die aktuelle Richtungsangabe, Bewegungsgeschwindigkeit, Höhe und das Kartendatum mit übertragen.

NaviMeta ist eine NAVI–spezifische Datenstruktur, welche GPS–Daten und Umgebungsinformationen in sich vereint. Die Umgebungsdaten werden anhand verschiedener Kriterien beschrieben. Näheres siehe unter Abschnitt 3.1.

Ablauf

Das zeitliche Verhalten der einzelnen Komponenten, welche untereinander Events austauschen, wird hier auf zwei Abbildungen verteilt dargestellt:

1. Abb. 5.4(a), S. 45 zeigt den zeitlichen Verlauf bei Verwendung des Menüs, das dem Benutzer vom StreetMap–Service als Interaktionsmöglichkeit zur Verfügung gestellt wird.

   Jeder Zyklus beginnt damit, dass der Benutzer eine Eingabe über das Steuerkreuz am Joypad tätigt. Der StreetMap–Service wertet die Eingabe aus und verschickt ein entsprechendes Event als Feedback über den SpeechOut–Service (Sprachausgabe) an den Benutzer. Dieser kann sich anhand der Rückgabe im Benutzermenü orientieren.

2. Abb. 5.4(b), S. 45 spiegelt den Ablauf während der Navigation wieder. Hier beginnt der Ablauf mit einer Positionsmeldung des GPSPosition–Service, welche den aktuel-

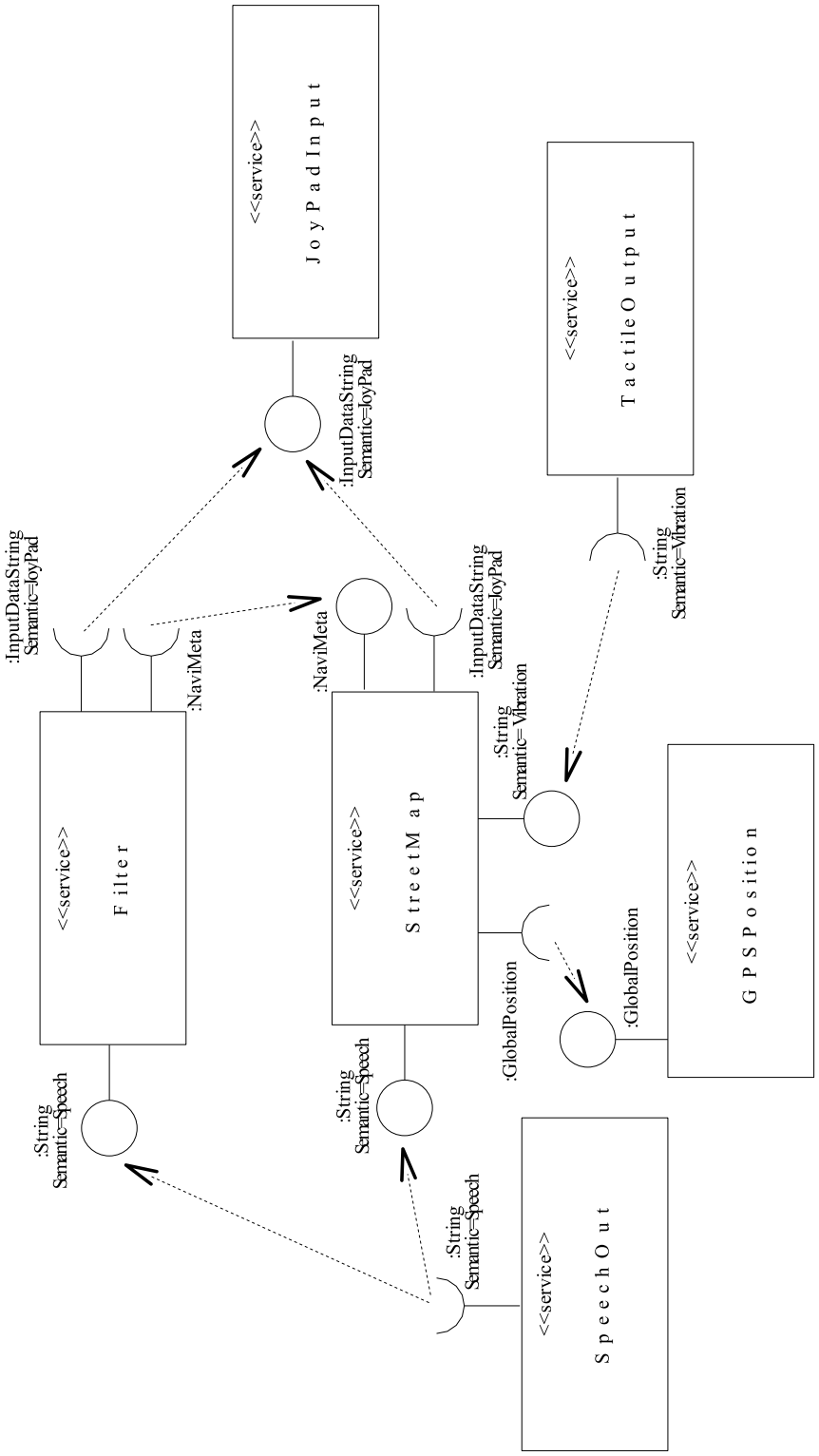


Abbildung 5.2: UML Serviceübersicht

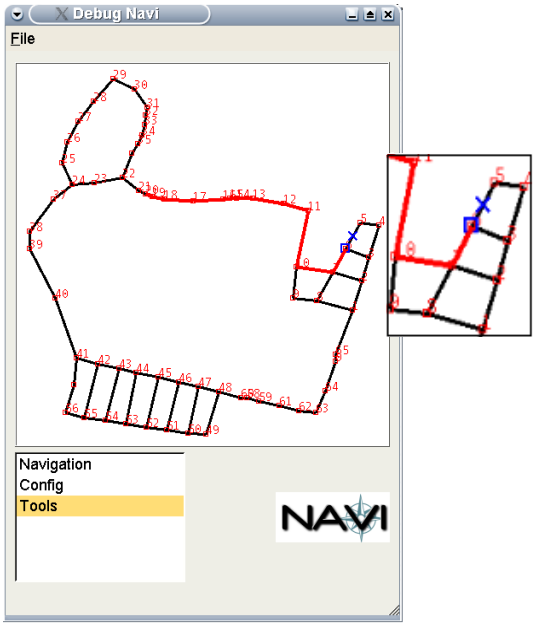
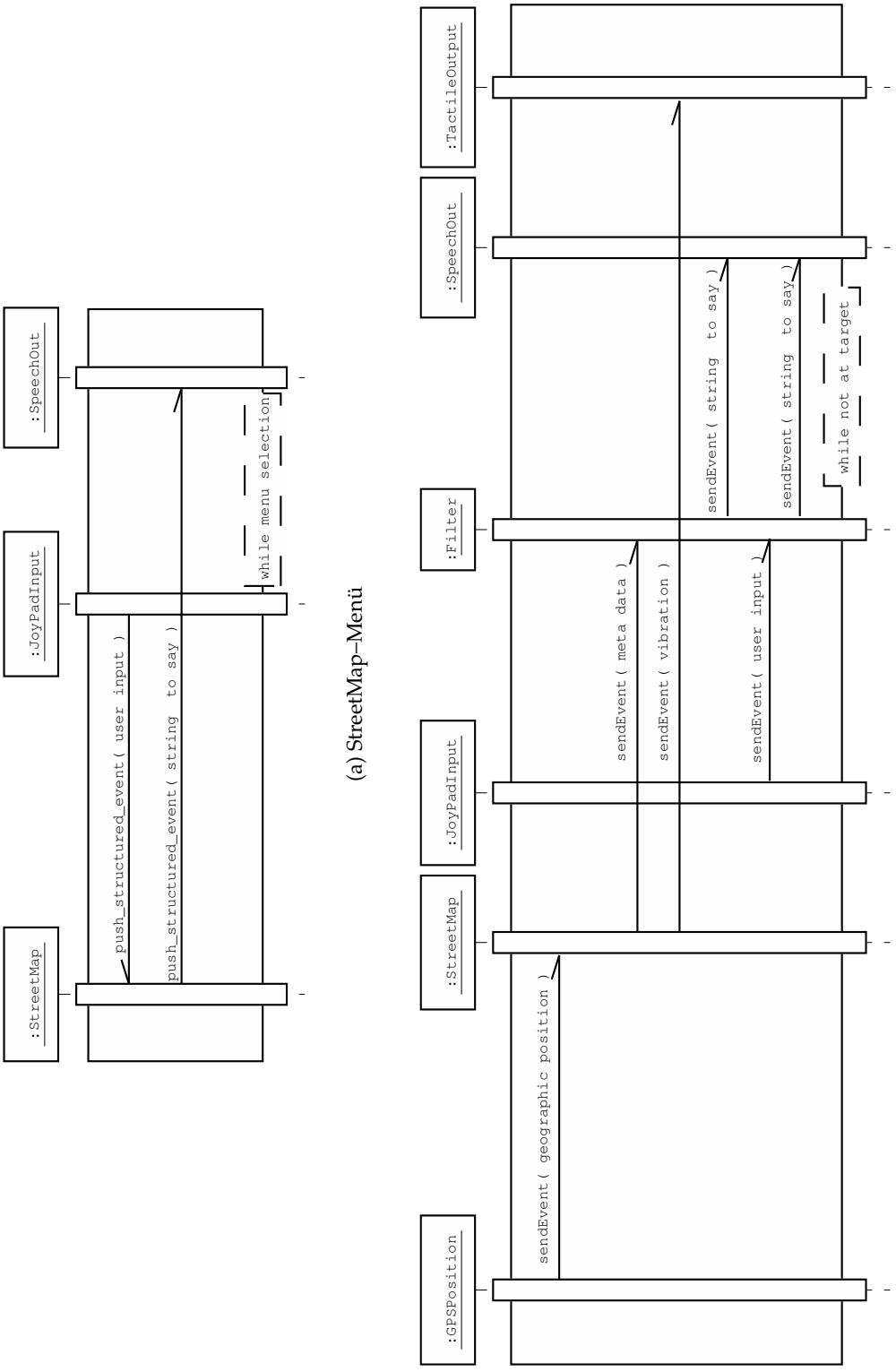


Abbildung 5.3: NAVI – KDE Debugausgabe

len Standort und die Bewegungsrichtung des Benutzers beinhaltet. Der StreetMap–Service ergänzt bei Bedarf diese Information um die Umgebungsdaten (NaviMeta Struktur) und verschickt diese an den Filter–Service. Wenn eine Richtungsänderung dem Benutzer bekannt gegeben werden soll, wird ein entsprechendes Event an den TactileOutput–Service geschickt. Der Filter leitet anhand seines internen Entscheidungsprozesses Umgebungsinformation an den SpeechOut–Service weiter und erwartet Feedback vom Benutzer. Falls der Benutzer sich entschließt, entsprechende Rückmeldung an den Filter–Service zu geben, wird die erfolgreiche Verarbeitung dem Benutzer über den SpeechOut–Service bekannt gegeben.



(a) StreetMap–Menü

(b) Navigation

Abbildung 5.4: UML Ablaufdiagramme

5.2.2 Filter–Service

An dieser Stelle mehr Informationen zur konkreten Realisierung des Filter–Services. Das Lernverhalten wird in Abschnitt 6 näher erläutert.

Der Filter–Service wurde als autonomer Agent entwickelt (siehe 3.1.1), es ist aus der Sicht des Filters nicht relevant, welche Daten er filtern soll. Diese Kapselung macht es möglich, den Filter–Service auch in anderen DWARF [27] Projekten einzusetzen. Die aktuelle Struktur läßt es nicht zu, dass Profilinformationen von anderen Services verwendet werden können [14] (siehe 7.1).

Interaktion mit DWARF

Der Filter–Service beruht wie alle anderen Services auf dem DWARF [27] –Framework und ist in C++ geschrieben. Wie in Abb. 5.5, S. 47 zu sehen ist, besitzt der Service zwei **Needs**:

ReceiveNaviMeta: Dieses Need dient dazu, Umgebungsinformationen vom StreetMap–Service zu empfangen. Jedes Event entspricht genau einer Meta–Information (z.B. Gebäude). Die Datenstruktur, wie unter Abb. 5.6, S. 47 zu sehen, ist eine Zusammenfassung von den empfangenen GPS–Daten und den Event Daten. Außerdem beinhaltet die Struktur schon komplett den String (NaviMetaInformation:label), welcher an die Ausgabe weitergeleitet werden soll.

Der Eingabevektor (siehe 4.4.2) wird in der Corba Sequenz **NaviMetaScale** abgebildet. Jede Dimension entspricht einem Teil der Sequenz. *Die Anzahl der übermittelten Dimensionen, könnte von Event zu Event differieren [15], die SVM [16] läßt dies aber nicht zu.*

Der Filter muss also nur anhand der Meta–Informationen entscheiden, ob er die Informationen aus diesem Event weiterleiten will.

ReceiveJoyPadInput: Der Filter–Service benötigt derzeit vom Benutzer nur zwei verschiedene Eingaben als Feedback: **Information ist interessant** und **Information ist unerwünscht**.

Der Filter–Service kann Eingaben des Joysticks direkt auswerten. Die Interface Unterstützung ist vollständig. Es könnten also auch direkt noch andere Parameter des Filter–Services damit gesteuert werden, wie z.B. Reset der Lernmaschine, Algorithmus der Lernmaschine usw.

Als **Ability** besitzt der Service nur die Möglichkeit der Stringweitergabe an den SpeechOut–Service. Wie schon erwähnt ist dieser vom Streetmap–Service vorgegeben. In der aktuellen Implementierung werden auch einige Debugausgaben über dieses Interface geschickt.

[14] Dazu sollten die Needs und Abilities des Services allgemeiner gestaltet werden, und nicht auf der NaviMeta Datenstruktur beruhen.

[15] Jedes Event wird getrennt zu allen anderen behandelt. Dies kann in späterer Entwicklung dazu verwendet werden, um Eingabestrukturen von verschiedenen Sendern zu verarbeiten, welche dann auch verschieden viele Dimensionen aufweisen dürfen. Nur pro Sender muss diese Anzahl konstant bleiben.

[16] Der Filter–Service hat als Demonstration nur eine SVM implementiert, andere Lernverfahren können einfach nachgerüstet werden.

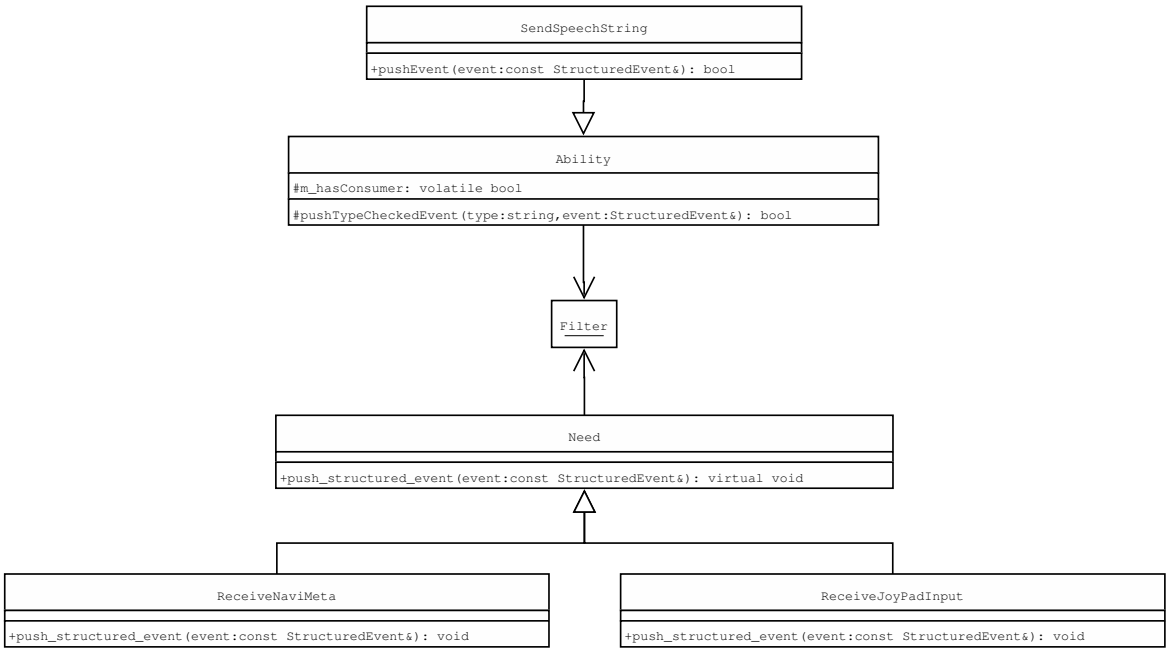


Abbildung 5.5: UML – Needs/Abilities des Filter Service

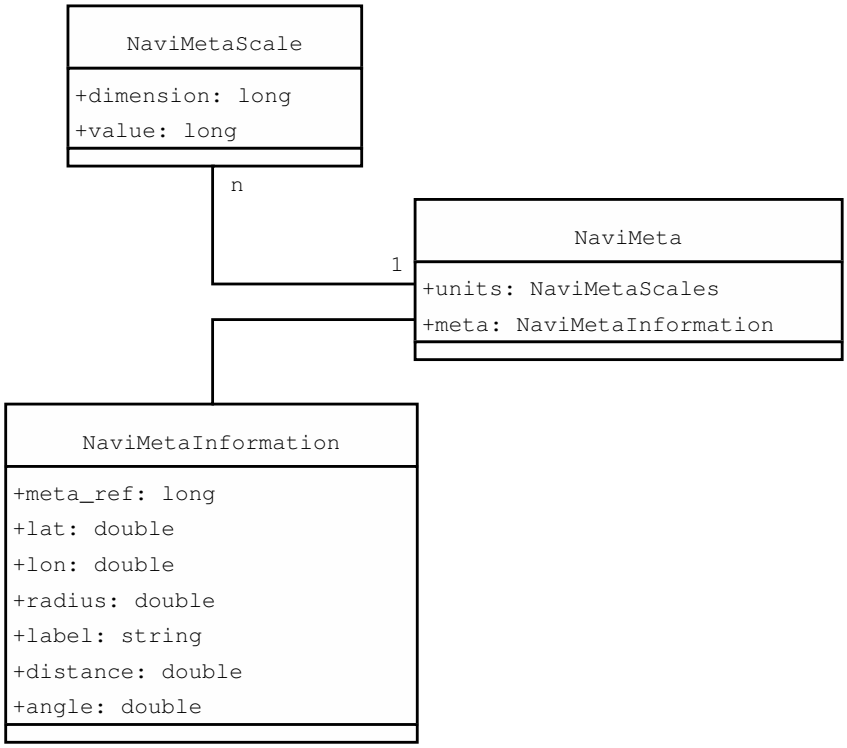


Abbildung 5.6: UML – NaviMeta Datenstruktur

Interaktion mit TORCH

Der Filter–Service selbst besteht aus zwei Teilen, wie man mit Abb. 5.7, S. 49 gut sieht.

FilterServiceSVM: Dieses Objekt wird mit dem Namen **brain** im Filter–Service referenziert. Derzeit implementiert dieses Objekt nur eine SVM. Das Interface sollte jedoch auch einfach für andere Lernverfahren wiederverwendbar sein.

Der Zustand der Maschine wird im Objekt vollkommen autonom verwaltet, Parameter zur SVM–Maschine können nur bei der Erzeugung angegeben werden.

Das Objekt hält die Trainingsdaten intern in einem Ring–Buffer fixer Größe, falls neue Daten eingefügt werden, wird der Buffer solange dynamisch erweitert, bis die maximale vorgegebene Größe erreicht wird. Ab diesem Zeitpunkt werden dann alte Einträge selbständig überschrieben.

Als Interface reichen folgende Funktionen aus (die anderen sind hauptsächlich für Debugzwecke):

add: Fügt einen neuen Eingabevektor als Trainingsdatensatz der Maschine hinzu. Für die Datenkonsistenz muss das externe Programm sorgen. Die Maschine akzeptiert alle Eingabevektoren (fehlerhafte Dateneingaben wirkt sich nur auf die Klassifizierung künftiger Eingaben aus, mehr siehe Abschnitt 6).

ask: Stellt der Maschine eine Frage, welche diese mit einer **real** [17] Zahl beantwortet. Negative Zahlenbereiche entsprechen **false**, positive **true**.

Filter: Das Filter–Objekt kommuniziert zum einen mit DWARF, zum anderen verwaltet es das FilterServiceSVM–Objekt. Abb. 5.8, S. 49 veranschaulicht die Arbeitsweise.

- Der Filter bekommt serielle Events. Die Bearbeitung der Events kann jedoch unter Umständen lange dauern (Klassifizieren ist in $O(n^2)$, Lernen in $O(n^3)$, Sprachausgabe dauert einige Sekunden). Deshalb werden Metadaten in eine Warteschlange gestellt, welche begrenzte Länge hat (zu alte Metadaten sind für den Benutzer nicht interessant).
- Benutzereingabe ist nur zum letzten ausgegeben Event zu erwarten.

  Es genügt also, sich ausschließlich das letzte Event zu merken, bis der Benutzer seine Eingabe getätigt hat. Die Eingabe kann [18] dazu führen, dass der Filter die SVM mit den neuen Daten trainiert, was einige Zeit in Anspruch nimmt.

  Während die SVM lernt, blockiert der Filterservice. Dieses Verhalten scheint nicht zielführend zu sein, läßt sich aber auf keine andere Weise realisieren, denn solange die SVM lernt, kann sie nicht klassifizieren.

  Die Meta–Datenwarteschlange dient dazu, mehrere Meta–Daten, welche in kurzer Abfolge empfangen werden zwischenzuspeichern, bis der SpeechOut–Service sie abgearbeitet hat.

[17] Der Datentyp **real** wird von der Bibliothek Torch [21] eingeführt.

[18] Es ist nicht sinnvoll machbar nach jeder Eingabe die SVM neu zu trainieren, da das Trainieren sehr rechenaufwendig ist. Es wird deshalb gewartet bis der Benutzer eine kleine Anzahl von Eingaben getätigt hat bevor die SVM trainiert wird..

Abbildung 5.7: UML – Filter–Service Objektübersicht

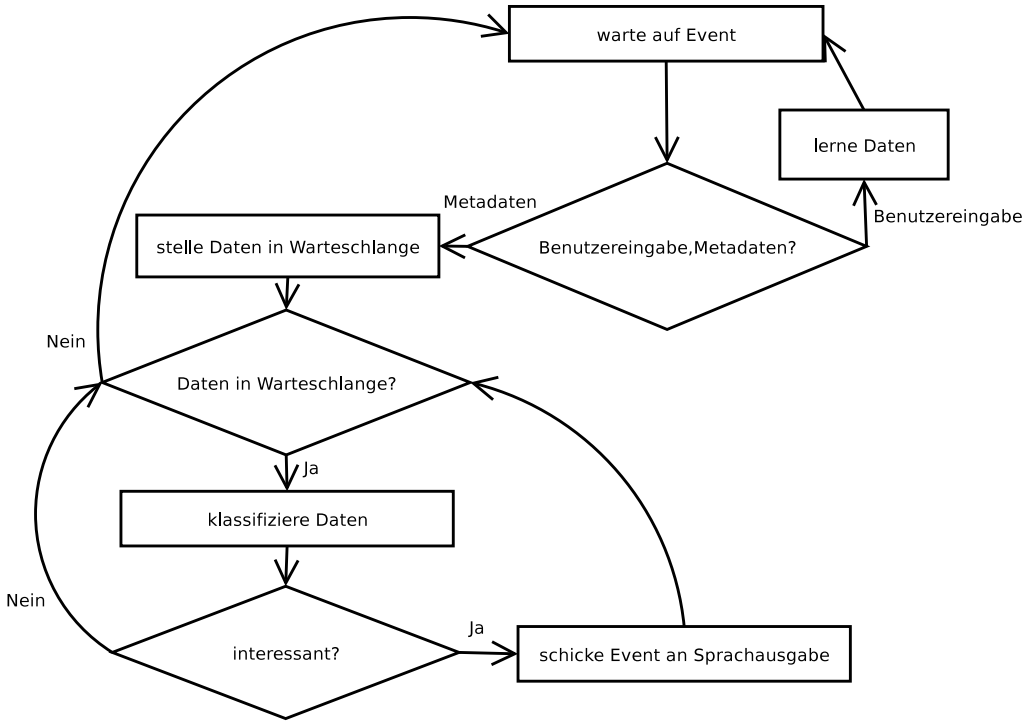


Abbildung 5.8: Ablaufdiagramm – Filter–Service

5.2.3 SVM Parameter

Diskretes Lernmodell

NAVI [14] besitzt ein diskretes Lernmodell (siehe 4.4.4). Die Dimensionen der SVM sind fix vorgegeben, aber auch die Werte, welche in den verschiedenen Dimensionen gültig sind.

Tab. 5.1 zeigt ein Lernmodell wie es in NAVI [14] zum Einsatz kommt. Es gibt in diesem Beispiel drei Dimensionen mit je drei Werten. Andere Werte sind für die verschiedenen Dimensionen nicht gültig.

Daraus ergibt sich, dass bzgl. der Eingabedaten keine Unschärfe zu erwarten ist.

Benutzereingabe

Der einzige Unsicherheitsfaktor welcher Fehler verursachen kann, ist die Benutzereingabe. Auf diese kann aber kein Einfluss genommen werden. Gerade wenn erst wenige Datensätze vorhanden sind wirken sich Entscheidungen des Benutzers sehr stark auf die SVM aus (siehe 6).

Wahl des Kernels

Das Lernverhalten der eingesetzten SVMs hängt nur noch vom **Kernel** ab. Im Falle von NAVI [14] ist die Wahl des Kernels sehr einfach [42].

Das diskrete Lernmodell passt gut mit einem **Gaussian Kernel** [30] zusammen.

$$k(x,z) = e^{\frac{\|x-z\|^2}{2\sigma^2}} \quad x,z = Eingabevektoren \tag{5.1}$$

σ ist der einzige Parameter welcher gewählt werden muss. Dieser gibt die Unschärfe der SVM an. Am Beispiel von Tab. 5.1 betrachtet, wie sehr legt der Benutzer drauf Wert zwischen "Morgen" und "Mittag" zu unterscheiden.

Die Werte der Dimensionen müssen, um mit der SVM bewertbar zu sein, auf Zahlen abgebildet werden. Wird hierfür einfach die Zeilennummer genommen, ergibt sich ein **Abstand von 1** zwischen benachbarten Werten.

Tabelle 5.1: Navi – Lernmodell

Nr.	Essen	Uhrzeit	Kleider
0	Buffet	Morgen	Sportbekleidung
1	Fastfood	Mittag	Alltagskleidung
2	Restaurant	Abend	Abendbekleidung

Soll es keine Unschärfe im Lernverhalten geben, muss

$$\sigma = 1$$

gewählt werden.

6 Ergebnisse, Beispiel

In diesem Abschnitt soll der Filter–Service von einer anderen Seite betrachtet werden. Es wurden schon einige Anmerkungen zu Problemen in der praktischen Realisierung gemacht.

Anhand eines Beispieles sollen hier die restlichen Überlegungen, welche sich im Laufe der Realisierung ergaben und durchdacht wurden, erläutert werden.

6.1 Szenario

NAVI [14] ist ein Navigationssystem für Blinde und Sehbehinderte, gerade deshalb wird hier ein ganz anderes Testszenario verwendet. Die Sicht aus einer anderen Perspektive ermöglicht es, neue Ansätze zu geben und eine einfachere Bewertung des Systems.

Der Vorteil in diesem komplett unabhängigen Modell ist die Freiheit, den Testbenutzer und seine Umgebung beliebig vorzugeben. So wurde versucht in diesem Beispiel auf möglichst alle bisher behandelten Probleme einzugehen und deren erarbeitete Lösung aufzuzeigen.

Ritter

Es war einmal ein blinder **Ritter**, dieser bekam von einer Fee ein Navigationssystem Names NAVI [14] geschenkt. Da er tierlieb war, benutzte er **Vibrationssporen**, um sein Pferd zu steuern.

Seine Aufgaben in unserer Welt sind:

1. Die Prinzessin finden und retten.
2. Ein Schwert zu erhalten, um gegen den Drachen zu kämpfen.
3. Sich auf der Reise nicht von anderen Frauen (Hexen) ablenken zulassen.

Karte

Auf Abb. 6.1, S. 53 ist die Route des Ritters eingezeichnet. Sie beginnt bei 1- einer Fee und endet bei 7- der Prinzessin. Hier eine Übersicht der Wegpunkte:

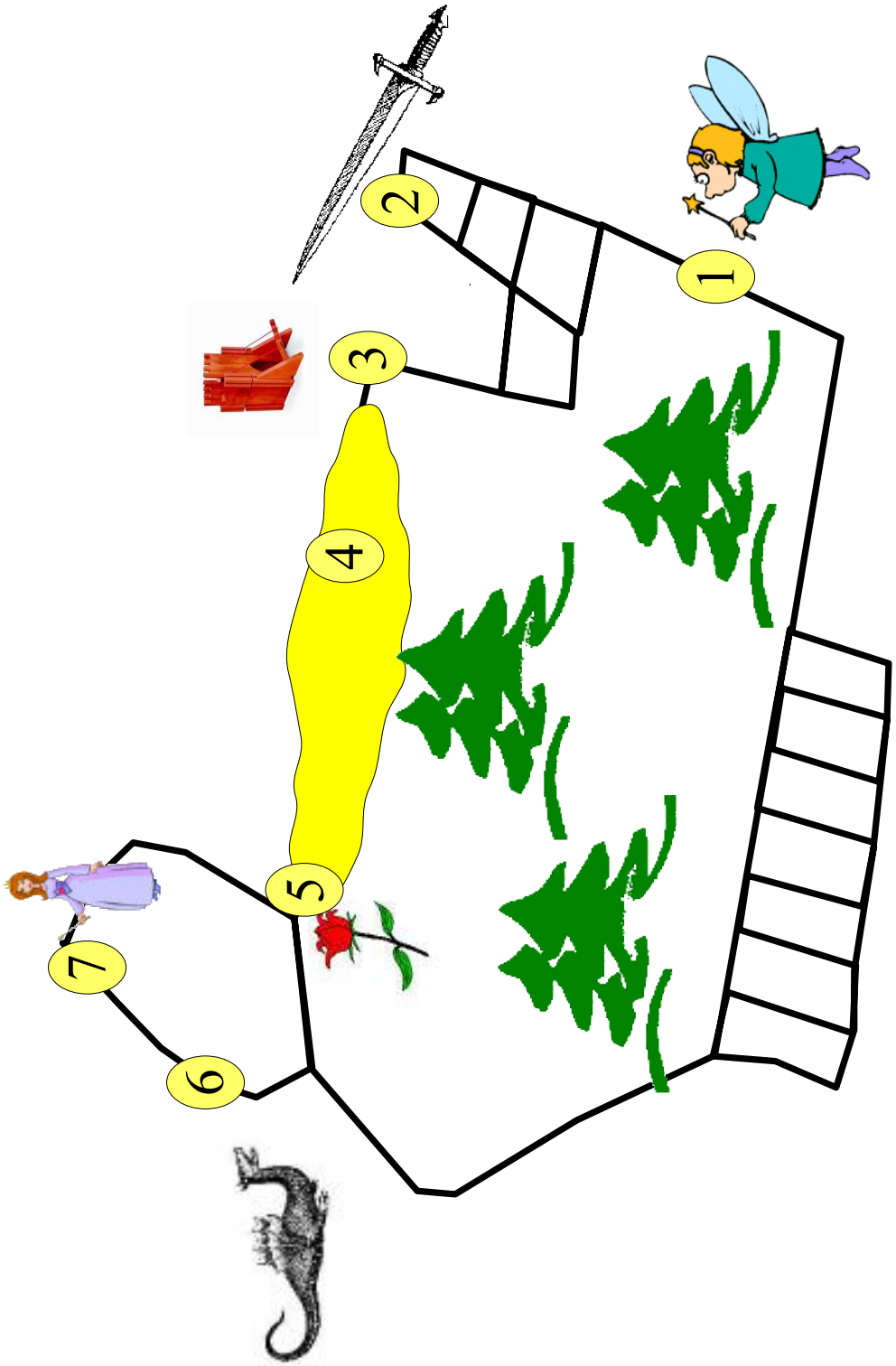


Abbildung 6.1: Szenario – Karte

Wegpunkt	Beschreibung
1	Start
2	Schwert
3	Stadteingang
4	Händler in der Stadt
5	Hexen
6	Drache
7	Prinzessin

Der Routenplaner und die taktile Ausgabe werden in diesem Szenario nicht näher behandelt, hierzu verweise ich auf die Parallelarbeit von Günther Harrasser [35].

6.2 Lernmodell

Wie in Abschnitt 4.3.1 und Abschnitt 4.4.2 erläutert, benötigt das System ein Lernmodell, um ein Benutzerprofil des Ritters erstellen zu können.

Auswahl der Dimensionen

Der Filter–Service nutzt in dieser Implementierung eine SVM (siehe 4.4.3). Tab. 6.1 zeigt von links nach rechts die Dimensionen und von oben nach unten die Werte dieser.

Konkret werden die Werte dieser Tabelle nach **N** abgebildet. Dazu wurde einfach die Zeilennummer der Tabelle bei 0 beginnend verwendet. **dummy** ist also in allen Dimensionen die 0. Der Wert 0 ist für alle Dimensionen besonders wichtig, aber auch für viele Probleme verantwortlich. Wie gleich folgt, sind nicht alle Gebäude und Personen allen Dimensionen zuordbar.

Die SVM benötigt in jeder Dimension einen konkreten Wert, falls also kein Wert der Dimension zutrifft oder die ganze Dimension nicht anwendbar ist wird der Wert 0 (dummy) verwendet.

Die Dimensionen wurden gewählt um den Ritter gut zu charakterisieren, andere Anwendungen führen natürlich zu anderen Dimensionen. Unser Ritter teilt sein inneres Weltbild anhand **Essen**, **Trinken**, **Geschäfte**, **Orte**, **Frauen** und **Gefahren** ein. Mehr Dimensionen würden den Ritter besser beschreiben, die Datenerfassung wird jedoch schnell ein Problem, denn für jedes Gebäude und jede Person müssen alle Dimensionen angegeben werden [1].

Hinweise zur Wahl von Dimensionen

Bei den Dimensionen muss auf Widerspruchsfreiheit geachtet werden. SVMs machen von jeder Dimension zu jeder anderen einen Ebenenschnitt (siehe 4.4.4). Falls sich nun die Dimensionen ungünstig *widersprechen*, fällt dieser Schnitt für diese Dimensionen ungünstig aus.

[1] Bei der technischen Präsentation wurden 15 Wegpunkte gewählt, also $15 * 7 = 105$ Einträge in der Datenbank erzeugt. Eine Stadt wie München welche Schätzungsweise $2 * 10^5$ Wegpunkte besitzt mit einem Lernmodell von 10 Dimensionen, benötigt $2 * 10^6$ Datensätze welche großteils von Hand eingegeben werden müssen.

Tabelle 6.1: Szenario – Lernmodell

Nr.	Essen	Trinken	Geschäft	Spezieller Ort	Gebiet	Frauen	Gefahr
0	dummy	dummy	dummy	dummy	dummy	dummy	dummy
1	Leckerei	Fluss	Händler	Durchgang	Bauernhof	Hexe	keine Gefahr
2	Snack	Brunnen	Marktplatz	Straße	Ansiedlung	Mädchen	wenig Gefahr
3	kleine Mahlzeit	Bar		Runenstein	Siedlung	nettes Mädchen	Gefahr
4	grosse Mahlzeit	Pub		Drachenhöhle	kleine Siedlung	Prinzessin	viel Gefahr
5	Rittertafel	Rittertafel		magischer Stein	Dorf	Königin	
				Schloss			

Tabelle 6.2: Szenario – Beispiel für ein alternatives Lernmodell

Nr.	Essen	Trinken	Frau	Gefahr
0	Ja	Ja	Ja	Ja
1	Nein	Nein	Nein	Nein

Dimensionen, welche alle Wegpunkte (Meta–Daten) gleichermaßen beschreiben sind natürlich von Vorteil, wie z.B. die Dimension **Gefahr**, die sich zu allen Eingaben korrekt zuordnen läßt. Es wäre auch möglich, sehr kleine Dimensionen (wenige Werte) einzuführen und so nur grundsätzlich zu entscheiden, ob eine Information der Dimension zugehörig ist oder nicht (Tab. 6.2).

Für eine große Menge von Meta–Daten ist das alternative Profil allerdings zu grob und sortiert sehr schnell alle Daten als **uninteressant** aus. Wenn eine Dimension wenige Unterteilungen besitzt, kann man dies auch als einfach halbieren/dritteln/...der Dimension auffassen. So ist die Wahrscheinlichkeit, dass viele Werte dem **uninteressanten** Bereich zugeordnet werden, größer, z.B. falls eine Dimension aus zwei Werten besteht und einer davon für **uninteressant** befunden wurde (Ebenenschnitt der SVM), betrifft das schon 50% der Meta–Daten (je nach Verteilung der Meta–Daten auf die Werte).

Wegpunkte

Jeder Wegpunkt benötigt einen Vektor. Nach diesem wird entschieden, falls der Ritter in die Nähe des Ziele gelangt, ob er von NAVI [14] auf den Wegpunkt aufmerksam gemacht wird. Nachdem der Ritter dem Filter–Service Feedback gegeben hat, wird der jeweilige Vektor mit dem Feedback zusammen in das Ritterprofil übernommen (siehe 5.2.2).

Tab. 6.4(a) beschreibt die Zuordnung des Lernmodells zu den Meta–Daten. Die Dimension **Spezieller Ort** ist auf jeden Fall ein Problem für die SVM. Ein echtes besser/schlechter existiert nicht.

Ritterprofil

Der Ritter soll sich wie ein Ritter verhalten, dies ist eine wichtige Komponente des Szenarios. Falls der Ritter zufällige Entscheidungen trifft, ist das Lernverhalten des Filter–Services nicht mehr nachvollziehbar.

Schlimm hierbei sind Widersprüche. Die Fee von **Punkt 2** hat viel Ähnlichkeit mit der Prinzessin von **Punkt 7**. Wenn der Ritter sich nun entscheiden sollte, am **Punkt 2** dem Navigationssystem mitzuteilen, "Die Fee interessiert mich nicht.", ändert sich das Weltbild des Ritters grundlegend (die Prinzessin wird uninteressant).

Der Ritter hier ist als guter Ritter erzogen (vorgegeben) worden und hat deshalb das in Tab. 6.4(b) angegebene Profil.

Nr.	Name	Essen	Trinken	Geschäft	Spezieller Ort	Gebiet	Frauen	Gefahr
1	Start	dummy	dummy	dummy	dummy	Ansiedlung	dummy	keine Gefahr
2	Schwert	dummy	dummy	dummy	magischer Stein	Ansiedlung	Mädchen	keine Gefahr
3	Stadteingang	dummy	dummy	dummy	Durchgang	Dorf	dummy	keine Gefahr
4	Händler in der Stadt	Snack	dummy	Händler	Straße	Dorf	dummy	keine Gefahr
5	Hexen	dummy	dummy	dummy	Straße	Dorf	Hexe	Gefahr
6	Drache	dummy	dummy	dummy	Drachenhöhle	Bauernhof	dummy	viel Gefahr
7	Prinzessin	dummy	dummy	dummy	Schloss	Ansiedlung	Prinzessin	wenig Gefahr

(a) Lernmodell

Interessant?	Essen	Trinken	Geschäft	Spezieller Ort	Gebiet	Frauen	Gefahr
Nein	dummy	dummy	dummy	dummy	dummy	dummy	dummy
Ja	dummy	dummy	dummy	Schloss	Ansiedlung	Prinzessin	viel Gefahr
Ja	dummy	dummy	dummy	dummy	dummy	Prinzessin	keine Gefahr
Nein	dummy	dummy	dummy	dummy	Dorf	Hexe	Gefahr
Ja	dummy	dummy	Marktplatz	Strasse	Dorf	dummy	keine Gefahr

(b) Ritterprofil

Tabelle 6.4: Szenario

- Er mag schöne Frauen in Schlössern.
- Hexen in Dörfern hingegen mag er nicht.
- Auf dem Marktplatz könnte er vielleicht Blumen für die Prinzessin erwerben, also mag er diesen auch.

Erwartungen bzgl. des Ritters

Die Strecke, welcher der Ritter zu bewältigen hat, besteht aus sieben Punkten. Aus Lernsicht gesehen besteht sein Profil aus fünf Grundaussagen. Falls er nun an allen Punkten die dort verfügbare Information von NAVI [14] angeboten bekommt, und er auch Feedback gibt, besteht am Ende sein Profil aus 12 Datensätzen.

Im Verhältnis zu den sieben Dimensionen des Lernmodells sind das sehr wenige Eingaben. Der Ritter neigt bei zu widersprüchlichem Verhalten dazu, im Chaos zu vergehen. Ein gutes stabiles Lernverhalten wird erst bei ausreichend vielen Eingabedaten erreicht [2] (ca. 100 mal mehr Eingabevektoren als Dimensionen [42]).

6.3 Weg durch das Szenario

Wegpunkt 1, NAVI

Der Ritter macht sich auf den Weg, er reitet die Strecke in der Reihenfolge ab, wie schon auf Abb. 6.1, S. 53 beschrieben.

Wegpunkt 2, Schwert

Wie schon unter Abschnitt 6.2 besprochen, wartet an diesem Wegpunkt eine Fee auf den Ritter, welche ihm ein Schwert geben will.

Der Ritter bekommt die Information, dass am Wegpunkt eine Fee auf ihn wartet, natürlich ausgegeben, denn sein Grundprofil (siehe 6.2) ist dem entsprechend modelliert. Er hat aber die Möglichkeit, seine Einstellung zu ändern, indem er nach der Ausgabe NAVI [14] mitteilt, dass die Fee **uninteressant** für ihn ist.

Falls er sich für **uninteressant** entscheidet, verschiebt sich in der SVM vor allem bei der Dimension **Frauen**, die Klassifizierungslinie zugunsten von Hexen (auf der Skala nach links).

Wegpunkt 3, Stadteingang

Sein vorgegebenes Profil tendiert dazu, das **Dorf** zu mögen. So bekommt er auch diese Information. Die Schwelle zum nicht mögen ist jedoch sehr knapp (Tab. 6.4(b)), zwei **Ja** zu einem **Nein** in der Dimension **Gebiet**.

[2] Die SVM verkraftet Fehler in den Trainingsdaten, vernünftige Fehlerraten sind aber erst bei grossen Eingabemengen erreichbar.

Die Entscheidung an diesem Wegpunkt hat enormen Einfluss auf die Wegpunkte 4 und 5. Entscheidet sich der Ritteri, das **Dorf** als nicht interessant anzusehen, müssen andere Dimensionen dieser Wegpunkte sehr positiv bewertet sein, damit der Ritter sie mitgeteilt bekommt.

Wegpunkt 4, Händler und Marktplatz

Im Dorf trifft der Ritter bei **4** einen Händler. Falls der Ritter in Kauflaune ist, und den Händler als **interessant** einstuft, bekommt er am Dorfende noch einen Marktplatz angeboten, auf welchem er der Prinzessin Blumen kaufen könnte (dieser kommt nicht in der Auflistung der Wegpunkte vor).

Wegpunkt 5, Hexen

Die Hexen werden, falls der Ritter sich wie ein Ritter benimmt, und am Anfang seines Weges (Wegpunkt 2, Schwert) die **Fee** als **interessant** eingestuft hat, ignoriert.

Falls der Ritter jedoch auf dem Rückweg vom Drachen nocheinmal bei den Hexen vorbeikommen sollte, und der Drache positiv im Profil des Ritters aufgenommen wurde, werden die Hexen nicht mehr ignoriert, da sich das Verhalten des Ritters bzgl. **Gefahr** geändert hat (der Ebenenschnitt der Dimension Gefahr wurde zugunsten des Wertes **Gefahr** verschoben).

Wegpunkt 6, Drache

Wie in Tabelle (Tab. 6.4(b)) zu sehen ist gibt es zwei **Ja** entgegen zu zwei **Nein** in der Spalte **Gefahr**.

Über den Drachen direkt gibt es im Ritterprofil nichts zu erfahren. Da der Ritter jedoch dazu neigt, die Gefahr zu mögen, wird ihm der Drache als Information angeboten (Tab. 6.4(b)).

Wegpunkt 7, Prinzessin

Falls der Ritter auf seinem bisherigen Weg **Frauen** nicht schlecht bewertet hat, wird ihm die Prinzessin als Information angezeigt.

Fazit

Der Ritter könnte den Weg noch ein paar mal abgehen, um so mehr Eingabedaten zu sammeln. Solange er sich nicht regelmäßig widerspricht, wird bald jede Information, welche nicht seinem Profil exakt entspricht, von NAVI [14] unterdrückt.

Falls er sich doch häufig widersprechen sollte kann es passieren, dass alle Informationen unterdrückt werden. Um diesen Effekt zu vermeiden wurde der Filter–Service (siehe 5.2.2) mit einem Ringbuffer als Trainingsdatensatzspeicher ausgestattet, so dass nicht beliebig viel Information als Grundlage zur Bewertung der Umwelt herangezogen wird (Grundeinstellung sind 300 Datensätze).

Des weiteren wurde eine kleine Heuristik eingebaut, welche zu erkennen versucht, ob NAVI [14] jegliche Information ungewollt unterdrückt. Der Filter–Service zählt die Anzahl der unterdrückten Informationen und gibt nach 5 (Grundeinstellung) Unterdrückungen auf jeden Fall eine Information an die Ausgabe weiter, um so dem Ritter zu ermöglichen, sein Profil zu ändern.

7 Zusammenfassung und Aussichten

Anhand des NAVI [14] Prototypen und der Aufarbeitung des Themas ist soweit klar geworden, wo die Probleme bei der Informationsfilterung zu suchen sind.

Qualität der Eingabedaten

Hauptproblem bleibt die Person, welche das System nutzt. Unabhängig vom Eingabeverfahren ist die Qualität des Lernverfahrens hauptsächlich von der Konsistenz der Eingaben abhängig (das Verhalten und die Qualität des Lernverfahrens wird hier als optimal angenommen).

Widerspricht sich der Benutzer immer wieder, ist das Lernverfahren nutzlos und mit Sicherheit schlechter als fix vorgegebene Regeln, welche die Umgebungsdaten nach Kategorien aufbereiten.

Informationsüberfluss

Meta–Daten sind auf der Karte nicht gleichmäßig verteilt. In einer Stadt unterscheidet sich die Innenstadt vom Gewerbegebiet grundlegend.

Unter der Annahme, dass der Benutzer sinnvolles Feedback liefert, hängt das Lernprofil stark von der Position des Benutzers ab. Dies ist im eigentlichen Sinne kein Problem, falls der Benutzer sich jedoch hauptsächlich NAVI [14] in der Innenstadt trainiert hat, dauert es einige Zeit bis sich das Verhalten im Gewerbegebiet ebenfalls angepasst hat.

Informationsmangel

Wie im Ritter Szenario (siehe 6.1) gut zu sehen war, ist die Konsistenz des Profils sehr schlecht wenn nur wenige Eingabedaten vorliegen. In der Praxis kann man dieses Problem lösen, indem man Profile für verschiedene Bevölkerungsgruppen vorfertigt, z.B. Männer, Frauen, ... und so dem Benutzer erspart die SVM selbst mit genügend Eingabedaten zu trainieren, damit diese stabile Entscheidungen trifft.

7.1 Aussichten

Anwendung der Lerndaten auf alle Meta–Daten

In NAVI [14] wird davon ausgegangen, dass die Informationsklassifizierung in Echtzeit berechnet werden muss. Ein anderer Ansatz wäre es, mehrere Lernverfahren zu kombinieren, und die Benutzereingaben nur zu sammeln um diese später auszuwerten.

Sobald der Nutzer sich eine neue Karte auf seinen PDA spielt könnte man das Benutzerprofil auf die Meta–Daten dieser Karte anwenden. So könnte man alle Daten im voraus filtern, oder auch nur grob gewisse Kategorien filtern.

Auf diese Weise hätte der Lernalgorithmus während der Navigation weniger Informationen zu klassifizieren, und könnte so mehr Rechenleistung für andere Aufgaben verwenden.

Meta–Daten erzeugen

Wie schon kurz erwähnt ist es ein Problem, eine Karte komplett mit passenden Meta–Daten zu bestücken (siehe 6.1). Problematisch ist es auch, im Nachhinein das Lernmodell zu ändern, da dann alle Meta–Daten mit geändert werden müssten. Schlimmer noch aber ist, dass die bis dahin erstellten Profile dann nutzlos sind und den Benutzern neu trainiert werden müssen.

Eine interessante Möglichkeit wäre es, anstatt einer zentraler Datenbasis auf mehrere Datenquellen zu setzen.

Via **Funk** könnte die Datenbasis in Echtzeit zusammengestellt werden. Der Benutzer begibt sich in die Nähe eines Senders und bekommt von diesem alle Informationen zu seiner Umgebung. So könnte jede Firma, Geschäft, etc. seine Sender aufstellen, um so die Nutzer mit Daten zu versorgen.

Störungen und Datenschutz sind bei dieser Variante eine Herausforderung, der Vorteil ist aber sicher die Aktualität der Daten.

Standard Profile

Es ist mühsam ein komplettes Benutzerprofil zu trainieren. Einfacher wäre es den Nutzern verschiedene Profile zur Verfügung zu stellen, welche sie dann nur noch anpassen müssten.

Interessant wäre es, wenn diese Profile dann für andere wieder frei zugänglich wären.

Neben der schnelleren Trainingszeit ermöglicht dies auch neue Geschäftsfelder, z.B. ein Touristenführer würde um ein vielfaches persönlicher. Auch wäre es möglich gewisse Geschäfte zu bevorzugen (Werbung) und so die Benutzer in ihrer Auswahl zu beeinflussen.

Mehrschichtiges Lernen

Anstatt einer Lernmaschine ist es durchaus denkbar, mehrere zu verwenden. Diese könnten dann verschiedene Teilaufgaben erledigen.

Die Kontextbezogenheit wird bei mehrfacher Nutzung ein Problem. Vom Benutzer werden sofortige Ergebnisse erwartet, wenn er z.B.angibt, dass das Restaurant **uninteressant** ist, wäre es gut, nicht fünf Sekunden später ein anderes auszugeben.

Dies widerspricht aber dem **langfristigen Lernziel**, denn vielleicht ist nur gerade an diesem Tage zu dieser Uhrzeit das Restaurant nicht von Bedeutung, normalerweise aber schon.

Eine zweite Maschine, welche ein langfristiges Benutzerprofil erstellt, ist also durchaus wünschenswert.

Ausgabetext direkt filtern

NAVI [14] erstellt ein Benutzerprofil mit dem die Ausgabe gefiltert wird. Ein anderer Ansatz wäre es, nicht ein Lernmodell zu erstellen, sondern einfach nur den Text zu klassifizieren, welcher mittels Sprachausgabe dem Benutzer vermittelt wird.

Es ist dann nicht möglich Aussagen zu den Verhaltensweisen des Benutzers zu machen, dafür muss auch kein Lernmodell erstellt werden und die Meta–Daten müssen nur korrekt benannt sein. Mehrere Sprachen lassen sich auf diese Weise aber nur schwer unterstützen.

Einfluss des Benutzerprofils auf die Routenplanung

Mit etwas Aufwand ist es möglich, Profilinformationen in die Routenplanung zu integrieren. Bevorzugt der Benutzer es, zu Fuß zu gehen? Oder ist ihm ein Taxi lieber? Kann die Route so geplant werden, dass auf dem Weg eine Kleinigkeit gegessen werden kann?

7.2 Status

Lernalgorithmen

NAVI [14] ist derzeit auf die Verwendung einer SVM beschränkt. Die Struktur sollte es aber einfach ermöglichen, andere Lernverfahren zu implementieren (Bayes).

Profil

Das erlernte Profil wird derzeit nicht gesichert. Technisch gesehen wäre dies kein größeres Problem. Es wurde jedoch darauf verzichtet, da es zuvor sinnvoll wäre mehr Benutzeroberfläche zu erstellen (Kartenauswahl, Datensicherung, ...).

Geschwindigkeit

Lernen ist teuer. Derzeit blockiert der Filter–Service während die SVM trainiert wird. Für ein kommerzielles System wäre das nicht akzeptabel. Abhilfe könnte geschaffen werden, indem mehr als ein Filter–Service verwendet wird, aber alle Services die gleiche Datenbasis verwenden.

Lernmodell

Eine Studie zur Ermittlung eines guten Lernmodells, welches für Städte geeignet ist, wäre sinnvoll. Das Problem der Meta–Daten Erstellung bleibt jedoch bestehen (siehe 6.2).

A Appendix

A.1 IDL – DWARF Filter Interface

Hier die Auflistung der IDL–Interfaces des Filter–Services.

A.1.1 NaviMeta

```
#include <DWARF/DwarfCommon.idl>

module DWARF {

        // learn modell data structure
        struct NaviMetaScale {
                long dimension;
                long value;
        };

        // meta–data information
        struct NaviMetaInformation {
                long meta_ref; // for debug, which database object
                double lat;      // for debug, geographic latitude
                double lon;      // for debug, geographic longitude
                double radius; // for debug

                string label;      // string to speak
                double distance; // relative distance from current position
                double angle;      // relative angle to current motion vector
        };

        // modell vector of meta–data
        typedef sequence<NaviMetaScale> NaviMetaScales;

        // composed navi data
        struct NaviMeta {
                NaviMetaScales units;
                NaviMetaInformation meta;
        };
};
```

A.1.2 InputDataString

```
#include <DWARF/DwarfCommon.idl>
//
// Enum used by all InputDataXXX idls
// State: The value represents current state
// Change: The state has changed and the value represents new state
//
enum InputDataEventType { InputDataState, InputDataChange };

// This idl contains the InputData data type for strings needed by DWARF
// component that send discrete string input data (e.g. "submit")
// IMPORTANT: The id has to be send in the header field event.header.fixed_header.event_name
// InputDataEventType enum is defined in InputData.idl

struct InputDataString {
        // type of event
        // InputDataState: The value represents current state
        // InputDataChange: The state has changed and the value represents new state
        InputDataEventType eventType;

        // string value
        string stringValue;
};
```

A.2 Service – XML Beschreibung

Der DWARF–Servicemanager benötigt eine Beschreibung der Service Interfaces.

A.2.1 Filter–Service

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">

<service name="Filter" startOnDemand="false" stopOnNoUse="false" startCommand="

  <ability name="SendSpeechString" type="string">
    <attribute name="StringSemantic" value="Speech"/>
    <connector protocol="PushSupplier"/>
  </ability>

  <need name="ReceiveNaviMeta" type="NaviMeta">
    <connector protocol="PushConsumer"/>
  </need>

  <need name="ReceiveJoyPadInput" type="InputDataString" predicate="(InputDataS
    <connector protocol="PushConsumer"/>
  </need>

</service>
```

A.3 SVM – Test

TORCH liefert ein kleines Beispielprogramm, welches einige Funktionen der Bibliothek demonstriert.

Das Programm ermöglicht es, eine SVM zu trainieren und im Anschluss zu testen. Als Eingabedateien können Textdateien verwendet werden.

Um Vergleiche anzustellen, kann der Zustand der trainierten Maschine gesichert werden. Da NAVI [14] die gleichen Algorithmen benutzt, könnte so eine Beispiel Maschine trainiert und der Zustand dieser direkt in den Filter–Service übernommen werden.

A.3.1 Parameter

```
SVMTorch III (c) Trebolloc & Co 2002

This program will train a SVM for classification or regression
with a gaussian kernel (default) or a polynomial kernel.
It supposes that the file you provide contains two classes,
```

```
except if you use the '-class' option which trains one class
against the others.

#
# usage: ./svm [options] <file> <model>
#

Arguments:
  <file>        -> the train file (<string>)
  <model>       -> the model file (<string>)

Model Options:
  -c <real>     -> trade off cst between error/margin [100]
  -std <real>   -> the std parameter in the gaussian kernel [exp(-|x-y|^2/std^2
  -degree <int> -> if positive, use a polynomial kernel [(a xy + b)^d] with the
  -a <real>     -> constant a in the polynomial kernel [1]
  -b <real>     -> constant b in the polynomial kernel [1]
  -rm           -> regression mode
  -eps <real>   -> eps tube in regression [0.7]
  -class <int>  -> train the given class against the others [-1]

Learning Options:
  -e <real>     -> end accuracy [0.01]
  -m <real>     -> cache size in Mo [50]
  -h <int>      -> minimal number of iterations before shrinking [100]

Misc Options:
  -seed <int>   -> the random seed [-1]
  -load <int>   -> max number of examples to load for train [-1]
  -dir <string> -> directory to save measures [.]
  -bin          -> binary mode for files

#
# or: ./svm --kfold [options] <file> <k>
#

Arguments:
  <file>        -> the train file (<string>)
  <k>           -> number of folds (<int>)

Model Options:
  -c <real>     -> trade off cst between error/margin [100]
  -std <real>   -> the std parameter in the gaussian kernel [10]
  -degree <int> -> if positive, use a polynomial kernel [(a xy + b)^d] with the
  -a <real>     -> constant a in the polynomial kernel [1]
  -b <real>     -> constant b in the polynomial kernel [1]
  -rm           -> regression mode
```

```
  -eps <real>   -> eps tube in regression [0.7]
  -class <int>  -> train the given class against the others [-1]

Learning Options:
  -e <real>     -> end accuracy [0.01]
  -m <real>     -> cache size in Mo [50]
  -h <int>      -> minimal number of iterations before shrinking [100]

Misc Options:
  -seed <int>   -> the random seed [-1]
  -load <int>   -> max number of examples to load for train [-1]
  -dir <string> -> directory to save measures [.]
  -bin          -> binary mode for files

#
# or: ./svm --test [options] <model> <file>
#

Arguments:
  <model>       -> the model file (<string>)
  <file>        -> the test file (<string>)

Misc Options:
  -load <int>   -> max number of examples to load for test [-1]
  -dir <string> -> directory to save measures [.]
  -bin          -> binary mode for files
```

A.3.2 Beispiele

Hier wird eine Maschine mit 9 Klassen trainiert, als Eingabedatei wird **data/traindata.gz** verwendet. Der Status der SVM wird in **model** abgelegt.

```
Linux_OPT_FLOAT/svm -std 1650 -class 9 data/traindata.gz model
Linux_OPT_FLOAT/svm --test model data/test_data.gz
```

A.4 Filter – Trainingsdaten

Der Filter–Service kann Eingabedaten direkt aus einer Datei laden. Damit kann die erste Lernphase unterdrückt werden.

Bei der Vorführung des System am 23.11.2003, wurde folgende Datei verwendet. Die Daten erklären sich aus dem verwendeten Lernmodell. Dieses ist anhand der SQL–Datenbank einfach zu verstehen (siehe NAVI–Source, `scripts/` Verzeichnis).

Format

Erste Zeile von links nach rechts:

- Größe des Eingabevektors
- Anzahl der Eingabevektoren

Nach dem Eingabevektor folgt seine Klasse (0,1).

Datei

```
7 9
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0
0 0 0 6 2 4 4 1
0 0 0 0 0 4 1 1
0 0 0 0 0 4 3 1
0 0 0 0 5 1 3 0
0 0 0 0 5 1 6 0
0 0 2 2 5 0 1 1
3 3 3 3 3 3 3 1
```

Literaturverzeichnis

[1] *Boost, a universal C++ Template Library*. http://www.boost.org/.

[2] *CRM114,the Controllable Regex Mutilator*. http://crm114.sourceforge.net/.

[3] *Debian, Debian Policy Manual*. http://www.debian.org/doc/debian-policy/.

[4] *Flite, a small fast run time synthesis engine*. http://www.speech.cs.cmu.edu/flite/.

[5] *Fraunhofer FIT, Nomadic Information System*. http://www.fit.fraunhofer.de/gebiete/nomad-is/index_en.xml.

[6] *Galileo, European Satellite Navigation System*. http://europa.eu.int/comm/dgs/energy_transport/galileo/index_en.htm/.

[7] *GNU/Linux*. http://www.kernel.org/.

[8] *GNU/readline*. http://www.gnu.org/directory/readline.html.

[9] *GPSd*. http://wwwbruegge.in.tum.de/projects/lehrstuhl/twiki/bin/view/DWARF/ProjectNaviGPSD.

[10] *GPSDrive*. http://freshmeat.net/projects/gpsdrive/.

[11] *IMMS, Intelligent Multimedia Management System*. http://freshmeat.net/projects/imms/.

[12] *KDE, Kommon Desktop Environment*. http://www.kde.org/.

[13] *National Marine Electronics Association*. http://www.nmea.org/.

[14] *Navigation Aid for Visually Impaired*. http://wwwbruegge.in.tum.de/projects/lehrstuhl/twiki/bin/view/DWARF/ProjectNavi.

[15] *Object Management Group (OMG)*. http://www.omg.org/.

[16] *OpenSLP*. http://www.openslp.org/.

[17] *QT Designer*. http://www.trolltech.com/products/qt/designer.html/.

[18] *QT, Trolltech*. www.trolltech.com/.

[19] *SpamAssassin*. http://spamassassin.org/.

[20] *SQLite Database Library*. `http://www.hwaci.com/sw/sqlite/` `http://freshmeat.net/projects/sqlite/`.

[21] *Torch, machine-learning Library*. `http://www.torch.ch/`.

[22] *TREKKER,A GPS system for the Blind and visually impaired*. `http://www.visuaide.com/gpssol.html`.

[23] *UMTS, Universal Mobile Telephone System*. `http://www.umtsworld.com/`.

[24] *UNIX Joystick Driver Wrapper Library*. `http://freshmeat.net/projects/libjsw/`.

[25] *X Multimedia System*. `http://www.xmms.org/`.

[26] N. BARTELME, in Geoinformatik, Modelle, Strukturen, Funktionen, Springer, Feb. 2000.

[27] M. BAUER, B. BRUEGGE, G. KLINKER, A. MACWILLIAMS, T. REICHER, S. RISS, C. SANDOR, and M. WAGNER, *Design of a Component-Based Augmented Reality Framework*, in Proceedings of ISAR 2001, 2001.

[28] B. E. BOSER, I. GUYON, and V. VAPNIK, *A Training Algorithm for Optimal Margin Classifiers*, in Computational Learing Theory, 1992, pp. 144–152.

[29] D. CAPOVILLA, G. HARRASSER, and G. KLINKER, *Entwicklung eines Katalogs mit Rahmenbedingungen für Orientierungs und Navigationshilfen blinder und sehbehinderter Menschen im Strassenverkehr*, 2002.

[30] CRISTIANINI and SCHOLKOPF, *Support Vector Machines and Kernel Methods: The New Generation of Learning Machines*, AIMAG: AI Magazine, 23, 2002.

[31] E. W. DIJKSTRA, *A note on two problems in connexion with graphs.*, Numerische Mathematik, 1, 1959, pp. 269–271.

[32] ESPINOSA, UNGAR, OCHAITA, BLADES, and SPENCER, *Comparing methods for introducing blind and visually impaired people to unfamiliar environments*, Environmental Psychology, 18, 1998, pp. 227–287.

[33] D. ETZOLD, *Improving spam filtering by combining Naive Bayes with simple k–nearest neighbor searches*, nov 2003.

[34] P. GRAHAM, *A Plan for Spam*, 2002.

[35] G. HARRASSER, *Design eines für Blinde und Sehbehinderte geeigneten Navigationssystems mit taktiler Ausgabe*, Master's thesis, Technische Universität München, Department of Computer Science, December 2003.

[36] HOFFMANN-WELLENHOF, LICHTENEGGER, and COLLINS, *GPS: Theory and Practice. 3rd ed.New York*, Springer–Verlag, 1994.

[37] A. KEMPER and A. EICKLER, *Datenbanksysteme: eine Einfuehrung*, Oldenbourg, Muenchen, 2 ed., 1997.

[38] A. F. LANGE, *An Introduction to the Global Positioning System*, in Proceedings of the Thirteenth Annual ESRI User Conference, vol. 2, Palm Springs, CA, 1993, pp. 179–183.

[39] B. M. and WEBER, *Taktile, verbale und motorische Informationen zur geographischen Orientierung Blinder*, Zeitschrift für experimentelle und angewandte Psychologie, 28, 1981, pp. 23–37.

[40] M. SAHAMI, S. DUMAIS, D. HECKERMAN, and E. HORVITZ, *A Bayesian Approach to Filtering Junk E-Mail*, in Learning for Text Categorization: Papers from the 1998 Workshop, Madison, Wisconsin, 1998, AAAI Technical Report WS-98-05.

[41] G. SAKKIS, I. ANDROUTSOPOULOS, G. PALIOURAS, V. KARKALETSIS, C. D. SPYROPOULOS, and P. STAMATOPOULOS, *A Memory–Based Approach to Anti-Spam Filtering for Mailing Lists*, Information Retrieval, 6(1), 2003, pp. 49–73.

[42] B. SCHOELKOPF and A. J. SMOLA, *Learning with Kernels*, The MIT Press, Cambridge, MA, 2002.

[43] J. SEARLE, *Minds, Brains, and Programs*, Behavioral & Brain Sciences, 3, 1980, pp. 417–458.
The classic "Chineese room" argument against AI programs being considered "intelligent".

[44] A. M. TURING, *Computing Machinery and Intelligence*, McGraw–Hill, 1963.

[45] P. WILLIAM S. YERAZUNIS, *Sparse Binary Polynomial Hashing and the CRM114 Discriminator*, 2003.