Diplomarbeit

Entwurf und Realisierung eines Java-Frameworks zur inhaltlichen Erschließung von Dokumenten

von
Thomas Büchner
Arbeitsbereich Softwaresysteme
TU HAMBURG-HARBURG

Betreuer:
Prof. Dr. Florian Matthes

Januar 2002

Abstract

Gegenstand der Arbeit ist der Entwurf und die Realisierung eines Frameworks, das den Vorgang der inhaltlichen Erschließung von Informationsobjekten softwaretechnisch unterstützt.

Die im Framework umgesetzten Erschließungsarten sind die Klassifikation, das Clustering und die Assoziative Suche. Dabei wird, aufgrund der großen wirtschaftlichen Bedeutung, schwerpunktmäßig auf die Erschließung von Textdokumenten eingegangen.

Der Grundgedanke des vorgestellten Modells besteht darin, die zu analysierenden Informationsobjekte in eine hochdimensionale numerische Vektorrepräsentierung umzuwandeln, und anschließend die Vektoren zu analysieren. Da sich das vorgestellte Modell uniform auf verschiedenste Informationsobjekte anwenden lässt, empfiehlt sich eine generische Lösung im Rahmen eines Frameworks. Aus diesem Ansatz ergeben sich synergetische Effekte, da einmal entwickelte Module übergreifend eingesetzt werden können. Ebenfalls ist die Komplexität des Gesamtvorgangs leichter zu bewältigen, wenn man alle beteiligten Einzelschritte voneinander getrennt betrachtet.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ziele der Arbeit 1

1.2 Gliederung der Arbeit 2

2 Erschließen von Informationsobjekten **3**

2.1 Bibliothekarisches Erschließen 3

2.1.1 Formales Erschließen 3

2.1.2 Inhaltliches Erschließen 3

2.2 Volltextsuche 5

2.3 Textklassifikation 5

2.4 Clustering 6

2.5 Assoziative Suche 7

2.6 Schlussfolgerungen 7

3 Ein konzeptuelles Modell der inhaltlichen Erschließung **9**

3.1 Datenmodell 9

3.2 Phasen einer Erschließung 10

3.3 Vektorerzeugung 13

3.3.1 Informationsobjekte 13

3.3.2 Vektorisieren von Features 15

3.3.3 Bildung des Gesamtvektors 17

3.3.4 Zusammenfassung 18

3.4 Vektorvorverarbeitung 19

3.4.1 Ausreißerbehandlung 20

3.4.2 Gewichtung 20

3.4.3 Merkmalsreduktion 20

3.4.4 Normalisierung 22

3.5 Vektoranalyse 23

3.5.1 Klassifikation 23

3.5.2 Clustering 24

3.5.3 Assoziative Suche 24

3.6 Zusammenfassung 24

4 Softwaretechnische Umsetzung - ADF **27**
4.1 Frameworks 27
4.1.1 Klassifizierung von Frameworks 28
4.1.2 Vorteile von Frameworks 29
4.1.3 Nachteile 29
4.2 Anforderungen an das **ADF** 29
4.3 Klassen und Pakete 31
4.4 Vektorrepräsentierung 31
4.4.1 Anforderungen an die Repräsentierung 33
4.4.2 Softwaretechnische Umsetzung 34
4.5 Mediatorinterfaces 38
4.6 Anpassung an Typen von Informationsobjekten 40
4.7 Kontrollfluss 41
4.7.1 Treiber 41
4.7.2 Trainingsphase 42
4.7.3 Produktivphase 46
4.8 Vektormodule 47
4.8.1 Vektorisierer 49
4.8.2 Präprozessoren 50
4.8.3 Vektormaschinen 50
4.9 Services 52
4.9.1 Eigenschaftsservice 53
4.9.2 Logservice 54
4.9.3 Kommunikationsservice 54
4.9.4 Superklasse Initializable 55
4.9.5 Persistenzservice 59
4.10 Katalogmuster 59

5 Zusammenfassung und Ausblick **63**
5.1 Zusammenfassung und Bewertung 63
5.2 Ausblick 64

Abbildungsverzeichnis

3.1 Trainingsphase 11
3.2 Verschiedene Vektormaschinen 11
3.3 Produktivphase 12
3.4 Typen von Informationsobjekten 14
3.5 Featureselektoren 14
3.6 Stringvektorisierer 15
3.7 Stringvektorisierer mit zwei Attributen 16
3.8 Vorverarbeitung von Texten 17
3.9 Vektorerzeugung 18
3.10 Vektorvorverarbeitung in der Trainingsphase 19
3.11 Vektorvorverarbeitung in der Produktivphase 19
3.12 Vektorvorverarbeitung in zwei Schritten 20

4.1 ADF mit verschiedenen Anwendungen 30
4.2 Kommunikation zwischen Framework und Anwendung 30
4.3 Überblick über wichtige Klassen von ADF 31
4.4 Packagediagramm 32
4.5 Konzeptuelles Klassendiagramm Trainingset-Vektor 32
4.6 Vector 35
4.7 Vektorinterfaces 36
4.8 Existierende Vektorimplementierungen 37
4.9 TrainingSet 38
4.10 Existierende Trainingsetimplementierungen 38
4.11 Mediatorinterfaces 39
4.12 Classification 40
4.13 FeatureSelector 40
4.14 LanguageDetector 41
4.15 AssetType 41
4.16 Vererbungshierarchie Driver 42
4.17 Relationship 42
4.18 Ablauf eines Trainings 43

4.19 Einfügeschritt 44
4.20 Sequenzdiagramm des Einfügeschrittes 44
4.21 Trainingsphase 45
4.22 Mögliche Treiber-Meditator Kombinationen 46
4.23 Produktivphase 47
4.24 VectorModule mit Subklassen 48
4.25 Metamodell der Vektorrepräsentierungen 48
4.26 Vectorizer 49
4.27 Sequenzdiagramm buildVectorizer() 50
4.28 Konkrete Vektorisierer 51
4.29 Vererbungshierarchie Präprozessoren 51
4.30 Subklassen von VektorEngine 52
4.31 Eigenschaftsservice 53
4.32 Logservice 54
4.33 Kommunikationsservice 54
4.34 Initializable 55
4.35 Initialisierung eines Objektes 56
4.36 Konfiguration der Services 56
4.37 Konfiguration der Informationsobjekttypen 57
4.38 Konfiguration des Vektortypes 57
4.39 Persistenzservice 59

Kapitel 1

Einleitung

Früher war die Masse von Informationen für den Unternehmenserfolg entscheidend, heute ist es deren Klasse. Es ist von zentraler Bedeutung für ein Unternehmen, den Mitarbeitern einen effizienten Zugriff auf die vorhandene Wissensbasis zu ermöglichen. Nach einer Untersuchung von Deloitte & Touche und Scribe Technologies bei den Top-1000-Unternehmen Großbritanniens verlieren etwa 80% der Mitarbeiter 40 Minuten pro Tag bei der Suche nach individuell benötigten Informationen [Ger00].

Aufgrund der täglich zunehmenden Digitalisierung und Vernetzung nimmt die Menge vorhandener Informationen ständig zu. Daraus folgt eine wachsende Bedeutung des Suchens und Wiederauffindens von Informationen. Unter dem Begriff *Information Retrieval* werden alle Verfahren zusammengefasst, die mit der Aufbereitung, Speicherung und Wiedergewinnung von Informationen zu tun haben. Die vorliegende Arbeit befasst sich schwerpunkthaft mit der automatisierten Aufbereitung von Informationsobjekten. Diese Aufbereitung soll anhand inhaltlicher Kriterien erfolgen, und somit die Lücke zwischen einer manuellen Erschließung und einer Volltextsuche schließen.

1.1 Ziele der Arbeit

Ziel dieser Arbeit ist die Entwicklung eines Java-Frameworks, das die Erschließung von Informationsobjekten unterstützt.

Der Schwerpunkt liegt auf der Erschließung natürlichsprachlicher Texte. Das entwickelte Framework soll sich flexibel an verschiedene Anwendungen anpassen lassen, den komplexen Prozess einer Erschließung in unabhängige Teilprozesse unterteilen und Texte verschiedener Sprachen verarbeiten können. Besonderer Wert wird auf eine performante Umsetzung des Erschließungsprozesses gelegt.

1.2 Gliederung der Arbeit

In Kapitel 2 wird zunächst ein Überblick über verschiedene Arten der Erschließung gegeben. Als Resultat werden die drei im Framework umgesetzten Arten der Erschließung (Klassifikation, Clustering, Assoziative Suche) motiviert.

Anschließend wird in Kapitel 3 das entwickelte und umgesetzte Modell einer inhaltlichen Erschließung vorgestellt.

Die Umsetzung dieses Modells im Framework **ADF** wird in Kapitel 4 dargestellt. Das Kapitel 5 fasst die gewonnenen Resultate zusammen und gibt einen Ausblick auf mögliche Anschlussarbeiten.

Kapitel 2

Erschließen von Informationsobjekten

2.1 Bibliothekarisches Erschließen

Erschließung ist Zugänglichmachung. In Anlehnung an den angelsächsischen Usus wird die Tätigkeit häufig auch als "*Indexing, Indexieren*" bezeichnet. Im klassischen Bibliothekswesen ist Erschließung gleichbedeutend mit *Katalogisierung*. Unter Katalog versteht man im allgemeinen Sprachgebrauch ein nach bestimmten Gesichtspunkten (meist alphabetisch oder sachlich) geordnetes Verzeichnis. (siehe [Hac83])

2.1.1 Formales Erschließen

Formales Erschließen meint das Zugänglichmachen von Dokumenten anhand ihrer formalen Eigenschaften (auch Metadaten genannt). Formale Eigenschaften von Dokumenten können sein:

- Namen der Autoren
- Jahr des Erscheinens
- Version des Dokumenten

Formale Erschließung wird erreicht, indem die Metadaten aller Dokumente erfasst und suchbar gemacht werden. Zum Austausch von Metadaten existieren verschiedene Austauschformate, zum Beispiel **MARC** (**MA**chine-**R**eadable **C**ataloging, siehe [Cra84]).

2.1.2 Inhaltliches Erschließen

Im Gegensatz zur formalen Erschließung macht die inhaltliche Erschließung Dokumente anhand ihres Inhalts zugänglich. Inhaltliche Erschließung besteht aus den folgenden drei Aspekten:

- Äußerste Komprimierung der Aussage
- Erkennen des Wesentlichen
- Fassen in Sprache

Inhaltserschließung ist damit ein Vorgang hohen intellektuellen Rangs und Anspruchs. Im klassischen Bibliothekswesen wird inhaltliche Erschließung durch Einsortieren von Dokumenten in *Sachkataloge* erreicht. Es gibt zwei Arten von Sachkatalogen: den *Schlagwortkatalog* sowie den *Systematischen Katalog*.

2.1.2.1 Schlagwortkatalog

Im Schlagwortkatalog werden die Dokumente bestimmten, aus dem Inhalt der Dokumente gewonnenen Schlagworten, zugeordnet. Ein *Schlagwort* ist ein möglichst kurzer, aber genauer und vollständiger Ausdruck für den sachlichen Inhalt eines Werkes. Die Schlagwörter werden unter sich alphabetisch geordnet. Ist die Liste der Benennungen für den Bibliothekar verbindlich, so spricht man von *gebundenen Schlagwörtern*, welche *terminologischer Kontrolle* unterliegen, d.h.:

- eindeutige Definition der mit den Schlagwörtern bezeichneten Inhalte und der zwischen den einzelnen Termini bestehenden Bedeutungsrelationen;
- verbindliche Regeln für die Verwendung der Schlagwörter.

Terminologische Kontrolle wird im allgemeinen hergestellt über das terminologische Regelwerk eines *Thesaurus*. Ein Thesaurus ist eine geordnete Zusammenstellung von Begriffen und ihren Beziehungen, die in einem Dokumentationsgebiet zum Indexieren, Speichern und Wiederauffinden dient.

2.1.2.2 Systematischer Katalog

Im Systematischen Katalog werden die Dokumente ihrem Inhalt entsprechend nach einem *System der Wissenschaften* eingeordnet. Dieses System der Wissenschaften ordnet die einzelnen Bereiche der Wissenschaft in einer bestimmten sachlich-logischen Abfolge an. Der Systematische Katalog vereinigt also sachlich zusammengehörige Literatur und weist sie im Zusammenhang ihres größeren Sachgebietes nach, im Gegensatz zum Schlagwortkatalog, der das Gesamtgebiet einer Wissenschaft in einzelne Teilgebiete aufteilt und diese alphabetisch ordnet. Das vom Systematischen Katalog verwendete Ordnungssystem ist in der Regel ein Werk der Bibliothekare und soll im Zeitverlauf möglichst stabil bleiben, damit eine einheitliche Erschließung der Inhalte aller Werke gewährleistet werden kann.

Mithilfe der vorgestellten Methoden lässt sich ein schneller Zugriff auf die Dokumente ermöglichen. Problematisch ist jedoch der hohe Erschließungsaufwand. Ziel der Überlegungen muss also eine Automatisierung der vorgestellten Verfahren sein.

2.2 Volltextsuche

Eine Alternative zu Erschließung von Dokumenteninhalten bietet die Volltextsuche. Mit Volltextsuche ist gemeint, dass alle Wörter aller Dokumente als relevante Terme aufgefasst werden.

Technisch wird dies durch das Erstellen eines invertierten Volltextindexes realisiert. In diesem sind zu jedem Term die Vorkommen in den Dokumenten verzeichnet. Die Suchanfrage wird nicht direkt auf den Dokumenten, sondern auf dem Volltextindex durchgeführt. Die einfachste Suchanfrage besteht nun in einem einzigen Suchwort. Diese kann durch boolesche Operatoren (UND, ODER, NICHT), Trunkierungen, Wildcards und Abstandsoperatoren erweitert werden.

Die beschriebene Volltextsuche wird oft als boolesches Retrieval bezeichnet. Das Ermöglichen einer Volltextsuche ist heutzutage Standard. Die Leistungsfähigkeit eines Retrievalsystems lässt sich anhand der Parameter *Precision* und *Recall* beurteilen:

- Der Wert des *Recalls* ergibt sich aus dem Anteil der gefundenen, relevanten Dokumente an der Gesamtzahl der relevanten Dokumente des Bestandes. (Vollständigkeit)

- *Precision* beschreibt den Anteil der gefundenen, relevanten Dokumente an der Gesamtzahl aller gefundenen Dokumente. (Präzision)

Im Einsatz boolescher Retrievalsysteme werden im allgemeinen hohe Recallwerte, aber niedrige Precisionwerte erzielt. Dies liegt an den gestellten Anfragen, welche meist nur aus sehr wenigen Termen bestehen. Zum Stellen komplexer Anfragen ist eine hohe Recherchekompetenz erforderlich, um bei spezifischen Themen mit der Kenntnis von Vieldeutigkeiten, Homonymen und Synonymen das Richtige zu finden. Es zeigt sich, dass einzelne Mitarbeiter gut recherchieren können, die meisten sind jedoch damit überfordert, komplexere Abfragen aufzubauen. Den Anspruch, Inhalte von Dokumenten exakt aufzuspüren zu können, kann boolesches Retrieval allein nicht überzeugend erfüllen, da Worte als Zeichenketten wenig Bezug zur Syntax und Semantik haben.

2.3 Textklassifikation

Innerhalb des Fachgebiets Informatik hat sich in den letzten Jahren ein Teilgebiet entwickelt, welches sich mit der Klassifikation von natürlichsprachlichen Textdokumenten befasst. Dieses Teilgebiet firmiert unter dem Namen *Textklassifikation*. Textklassifikation ist die Aufgabe, Textdokumente $D = \{d_1, ..., d_i, ..., d_n\}$ einer Menge von gegebenen Kategorien $C = \{c_1, ..., c_i, ..., c_m\}$ zuzuordnen. Die Analogie zur inhaltlichen Erschließung im Bibliothekswesen ist offensichtlich. Die Schlagworte sind mit den gegebenen Kategorien gleichbedeutend, und die Dokumente sind gleichbedeutend mit den zu erschließenden Werken. Ist

die Gesamtheit der Kategorien eines Unternehmens in einer Struktur hierarchisch angeordnet, wird das im Folgenden mit *Wissenslandkarte* bezeichnet. Die Wissenslandkarte entspricht also dem gebundenen Schlagwortkatalog.

Übertragen auf den Einsatz von Retrieval im Unternehmen bedeutet Inhaltliches Erschließen das Erstellen einer unternehmensspezifischen Wissenslandkarte, in die die vorhandenen Dokumente einsortiert werden. Als Resultat dieser inhaltlichen Erschließung liegen die Dokumente für eine semantische Suche erschlossen vor, sie sind effizient zugreifbar. Nun stellt sich die Frage nach der praktischen Durchführung dieses Erschließungsprozesses. Eine rein manuelle Durchführung erweist sich in der Praxis als kaum durchführbar. Dies liegt zum einen an der großen Menge von Dokumenten, zum anderen entwickelt sich möglicherweise die zugrundeliegende Wissenslandkarte dynamisch (besonders in Branchen, die einer stürmischen Entwicklung unterliegen). Daraus folgt ein großer ständiger Verwaltungsaufwand. Wünschenswert wäre eine geeignete maschinelle Unterstützung dieses Prozesses. Diese maschinelle Unterstützung soll also folgenden Forderungen gerecht werden:

- Zuordnen (Klassifizieren) von Dokumenten zu bestimmten Kategorien anhand semantischer Eigenschaften.
- Geringer Administrationsaufwand, also Lernen von Beispielen anstatt manuelle Konfiguration über Eingabe von Regeln.

Der Vorgang der Textklassifikation erfordert also das Vorhandensein einer Trainingsmenge, das heißt einer Menge von Kategorien mit manuell zugeordneten Dokumenten. Darauf aufbauend "lernt" ein maschineller Klassifikator in einer Trainingsphase. In der sogenannten Produktivphase ist der Klassifikator anschießend in der Lage, gegebene Dokumente den Kategorien zuzuordnen.

2.4 Clustering

In vielen Unternehmen liegt der Dokumentenbestand inhaltlich unerschlossen vor. Eine rein manuelle Ersterschließung zur Gewährleistung effektiven Retrievals bedeutet einen gewaltigen Aufwand. Zur Sichtung der Dokumente ist ein Tool wünschenswert, welches die Inhalte aller Dokumente analysiert und Kategorien von inhaltlich ähnlichen Dokumenten erstellt. Die erstellten Kategorien (im folgenden auch Cluster genannt) sollen die Eigenschaft haben, dass alle Dokumente eines Clusters einerseits ähnlich zueinander sind, andererseits unähnlich zu Dokumenten anderer Cluster.

Diese Aufgabenstellung wird *Clustering* genannt. Eine eng verwandte Aufgabenstellung ist die anschließende Visualisierung der erzeugten Cluster in sogenannten *Wissenslandkarten.* Auf diese Art und Weise ist es möglich, den gesamten Dokumentenbestand bezüglich der enthaltenen Inhalte zu sichten, und anschließend zu katalogisieren.

Auch im Falle einer Clusteraufgabe existiert eine Trainingsphase, in der der Clusterer alle vorhandenen Dokumente analysiert und eine Menge von Clustern bildet. In der sich sofort an die Trainingsphase anschließenden Produktivphase werden die analysierten Dokumente den gebildeten Clustern zugewiesen. Der Unterschied zur Klassifikation besteht darin, dass als Resultat der Trainingsphase die Kategorien synthetisch erzeugt werden, während im Falle einer Klassifikationsaufgabe die Kategorien mit zugeordneten Dokumenten gegeben sind.

2.5 Assoziative Suche

Ein weiteres Instrument zur Unterstützung von Retrieval ist die sogenannte *Assoziative Suche*. Diese findet zu einem eingegebenen Freitext oder bereits gefundenen Dokumenten weitere Dokumente mit ähnlichem semantischen Inhalt. Mit Hilfe dieser Funktionalität ist eine Navigation entlang bereits gefundener Dokumente möglich. Möglicherweise findet ein Benutzer recht schnell **ein** relevantes Dokument, es existieren aber noch einige weitere, die mit den vorhandenen Operationen nicht gefunden werden. Eine assoziative Suche kann folglich den Retrieval-Prozess erheblich erleichtern.

Im Vorlauf einer Assoziativen Suche muss das entsprechende Tool alle vorhandenen Dokumente in einer Trainingsphase analysieren. Anschließend – in der Produktivphase – ist das Tool in der Lage, Assoziative Suchen durchzuführen.

2.6 Schlussfolgerungen

Zusammenfassend sei nochmals die enorme wirtschaftliche Bedeutung eines effektiven Retrievals erwähnt. Zur Unterstützung dieses Retrievalvorgangs existieren die drei vorgestellten Verfahren, die die vorhandenen Dokumente inhaltlich erschließen, und somit effektiver zugreifbar machen. Alle Verfahren bedürfen einer Trainingsphase, in der sie die vorhandenen Dokumente analysieren. In der anschließenden Produktivphase findet die eigentliche Erschließung statt. Ebenfalls zeichnet alle Verfahren die Fähigkeit aus, Dokumente anhand ihrer inhaltlichen Verwandtschaft zu bewerten. Die Umsetzungen der verschiedenen Erschließungsformen liegen demzufolge eng beieinander.

Abschließend sei kurz erwähnt, dass die genannten Verfahren nicht nur im Zusammenhang mit Textdokumenten eine Bedeutung haben, sondern sie sind vielmehr auf beliebige Informationsobjekte übertragbar. Beispielsweise lassen sich Bilder oder Musikstücke Kategorien zuordnen.

Kapitel 3

Ein konzeptuelles Modell der inhaltlichen Erschließung

In diesem Kapitel wird ein konzeptuelles Modell einer inhaltlichen Erschließung vorgestellt. Dieses Modell beinhaltet die Repräsentierung der zu erschließenden Informationen als numerische Vektoren. Die eigentliche Erschließung lässt sich als ein Prozess in drei Schritten beschreiben: Vektorerzeugung, Vektorvorverarbeitung und Vektoranalyse. Orthogonal dazu existieren zwei unterscheidbare Phasen, in denen jeweils die genannten drei Schritte nacheinander ablaufen. Als erstes ist eine Trainingsphase notwendig. In der nachfolgenden Produktivphase erfolgen die Aktionen bzw. Entscheidung der Erschließung.

Der Schwerpunkt der folgenden Ausführungen liegt auf der Ausführung des entwickelten Modells. Die in den einzelnen Phasen des Modells anzuwendenden Algorithmen werden nur kurz erwähnt, zur Vertiefung werden Literaturreferenzen angegeben. Einen guten Überblick über eine große Zahl der erwähnten Algorithmen gibt [Seb99].

Ein ähnlicher Ansatz wie in der vorliegenden Arbeit wird in [Mou96] vorgestellt. Der weitergehende Fokus dieser Arbeit liegt in der Umsetzung des Modells in einem objektorientierten Framework.

3.1 Datenmodell

Im folgenden Kapitel wird die Repräsentierung von Textdokumenten als numerische Vektoren eingeführt. Diese Repräsentierungsform hat eine Reihe von wünschenswerten Eigenschaften:

- Computer können speichereffizient mit numerischen Daten umgehen.
- Verschiedenartige Informationsobjekte lassen sich als Vektoren repräsentieren und anschließend auf die gleiche Art und Weise analysieren.
- Die Vektorrepräsentierung ermöglicht Distanzbestimmungen in verschiedenen Metriken

und somit Aussagen über Ähnlichkeiten zwischen Vektoren (und somit Informationsobjekten).

- Es existiert eine große Anzahl von Algorithmen zur Analyse von Vektordaten (Klassifikation, Clustering).

Ein numerischer Vektor wird als Objekt der Klasse `Vector` repräsentiert. Ein Objekt vom Typ `Vector` verwaltet zusätzlich zu dem numerischen Werten des Vektors die eindeutige ID des repräsentierten Informationsobjekts. Im Falle einer Klassifikationsaufgabe verwaltet ein Objekt vom Typ *Vector* zusätzlich die eindeutig ID der Kategorie, welcher das Informationsobjekt zugehört.

Eine Menge von Vektoren wird von einem `TrainingSet` verwaltet (siehe Abbildung 4.5).

3.2 Phasen einer Erschließung

Wie bereits erwähnt, lässt sich jede Erschließungsaufgabe in zwei Phasen unterteilen:

1. Trainingsphase,
2. Produktivphase.

Die Vorgänge in beiden Phasen verlaufen analog in drei voneinander unabhängigen Teilschritten.

Die Trainingsphase (siehe Abbildung 3.1) ist die Lernphase des Systems, in ihr "lernt" das System die vorhandenen Informationsobjekte und ihre Beziehungen untereinander "kennen". Der erste Schritt der Trainingsphase ist die Erzeugung einer Trainingsmenge aus der Menge aller Informationsobjekte. Dieser Schritt heißt Vektorerzeugung. Es wird ein Vektor pro Informationsobjekt angelegt. An die Vektorerzeugung schließt sich der optionale Schritt der Vektorvorverarbeitung an. Dieser Schritt ist optional, es werden beispielsweise fehlerhafte Vektoren entfernt, oder der Merkmalsraum wird verkleinert. Die bis dato gebildete Trainingsmenge stellt die Grundlage des dritten Schrittes dar, der Vektoranalyse. In ihr erfolgt der eigentliche Lernvorgang.

Im Schritt der Vektoranalyse lernt eine untrainierte Vektormaschine die Merkmale der zu erschließenden Informationsobjekte, und liegt anschließend als trainierte Vektormaschine vor. In diesem Schritt unterscheiden sich die drei Erschließungsformen erstmals voneinander. Dies wird im folgenden durch unterschiedliche einzusetzende Vektormaschinen modelliert. Im Falle einer Klassifikationsaufgabe kommt ein Vektorklassifikator, im Falle einer Clusteraufgabe ein Vektorclusterer und im Falle einer Assoziativen Suche eine Ähnlichkeitssuchmaschine zum Einsatz (siehe Abbildung 3.2). Allen gemein ist, dass sie im Laufe der Trainingsphase eine gegebene Trainingsmenge analysieren. Im Falle einer

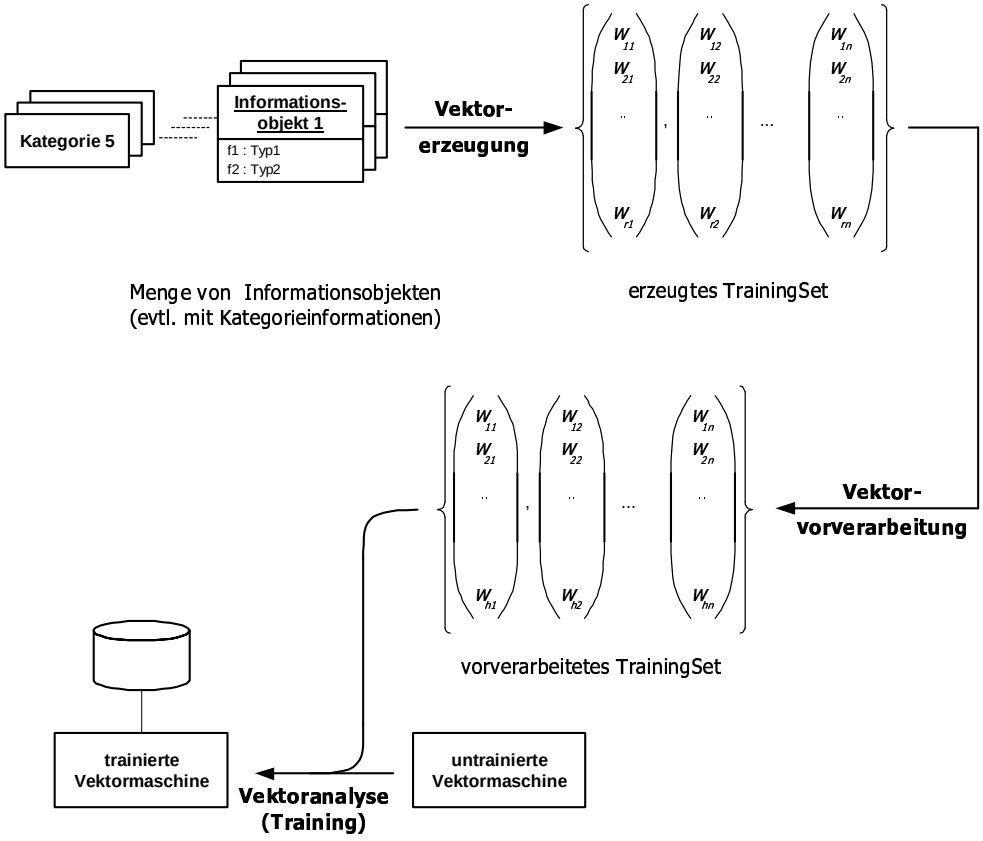

Abbildung 3.1: Trainingsphase

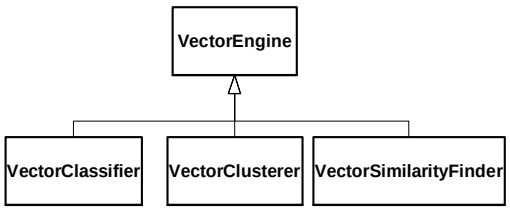

Abbildung 3.2: Verschiedene Vektormaschinen

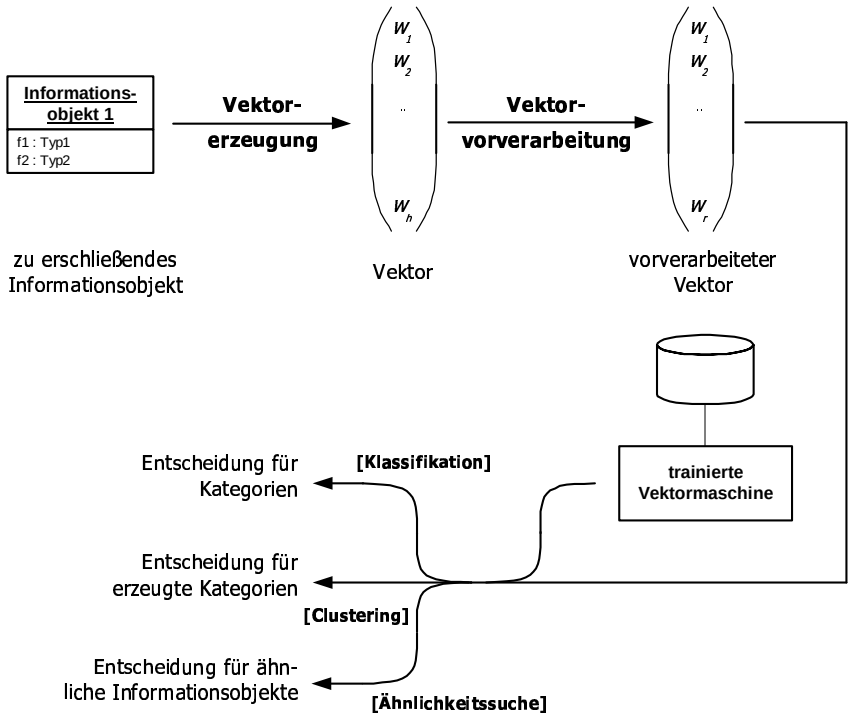

Abbildung 3.3: Produktivphase

Klassifikationsaufgabe werden Informationen über die Zugehörigkeit der Trainingsobjekte zu existierenden Kategorien benötigt.

Mit Abschluss der Trainingsphase sind die Vektormaschinen in einem trainierten Zustand, sie sind nun in der Lage, die jeweilige Erschließung vorzunehmen. Zusammenfassend lässt sich die Trainingsphase also in die drei Teilschritte Vektorerzeugung, Vektorvorverarbeitung und Vektoranalyse unterteilen.

In der nun folgenden Produktivphase (siehe Abbildung 3.3) soll ein bestimmtes Informationsobjekt erschlossen werden. Es laufen die gleichen Teilschritte wie in der Trainingsphase ab. Zunächst wird im Vektorerzeugungsschritt ein Vektor erzeugt, dieser wird anschließend im Vektorvorverarbeitungsschritt vorverarbeitet. Anhand dieses gebildeten Vektors führt die jeweilige trainierte Vektormaschine die entsprechende Vektoranalyse durch.

In diesem Schritt wird das eigentliche Resultat der Erschließung erzeugt. Die Art des Resultates hängt von der gewählten Erschließungsart und damit von der gewählten Vektormaschine ab. Ein Vektorklassifikator fällt Entscheidungen zugunsten existierender Kategorien. Ein Vektorclusterer erzeugt zunächst eine bestimmte Anzahl synthetischer Kategorien und ordnet anschließend zu erschließende Informationsobjekte diesen erzeugten Kategorien zu. Als Resultat einer Assoziativen Suche wird eine bestimmte Anzahl ähnlicher Informationsobjekte zurückgeliefert.

Zusammenfassend kann man – jeweils in Trainings- und Produktivphase – die wesentlichen drei Schritte einer Erschließungsaufgabe erkennen:

1. Vektorerzeugung,
2. Vektorvorverarbeitung (optional) und
3. Vektoranalyse.

Demzufolge gibt es drei Arten von vektorverarbeitenden Objekten, eine Art pro Teilschritt. Diese Objekte werden im folgenden *Vektormodule* genannt (siehe Kapitel 4.8).

Die strikte Unterteilung einer Erschließung in die genannten drei Schritte, und deren unabhängige Umsetzung, bringt die folgenden Vorteile mit sich:

- Verringerung der Komplexität durch Zerlegung eines großen Problems in drei kleinere.
- Die Art des Informationsobjektes ist nur im Schritt der Vektorerzeugung relevant. Implementierungen der folgenden Schritte sind sofort auf alle vektorisierte Informationsobjekte anwendbar.
- Die Art der Erschließung prägt sich nur im Vektoranalyseschritt aus. Demzufolge können alle drei Arten der Erschließung ohne großen Aufwand auf einmal vektorisierte Informationsobjekte angewandt werden.

Die folgenden Abschnitte widmen sich den drei Teilschritten einer Erschließung. Zunächst wird in Kapitel 3.3 die Vektorerzeugung am Beispiel von Textdokumenten behandelt. Im Kapitel 3.4 werden Verfahren der Vektorvorverarbeitung, eingehend auf die speziellen Aspekte der Erschließung natürlichsprachlicher Texte, erläutert. Abschließend werden in den Kapiteln 3.5.1, 3.5.2 und 3.5.3 Möglichkeiten der Vektoranalyse entsprechend der drei Erschließungsarten vorgestellt.

3.3 Vektorerzeugung

Im Vektorerzeugungsschritt werden Vektoren aus Informationsobjekten gewonnen. Die Qualität der gebildeten Vektoren ist für die Qualität der anschließenden Erschließung maßgeblich, die Vektorerzeugung ist demzufolge ein wesentlicher Schritt einer jeden Erschließungsaufgabe. In der Trainingsphase wird aus einer Menge von Informationsobjekten eine Trainingsmenge – ein Vektor pro Informationsobjekt – erzeugt, in der Produktivphase aus einem einzelnen Informationsobjekt ein Vektor.

3.3.1 Informationsobjekte

In diesem Abschnitt soll der Begriff des Informationsobjektes näher beleuchtet werden. Ein Informationsobjekt ist von einem bestimmten Typ und aggregiert eine für die Erschließung relevante geordnete Menge von Features f_i. In Abbildung 3.4 sind fünf verschiedene Typen von Informationsobjekten dargestellt.

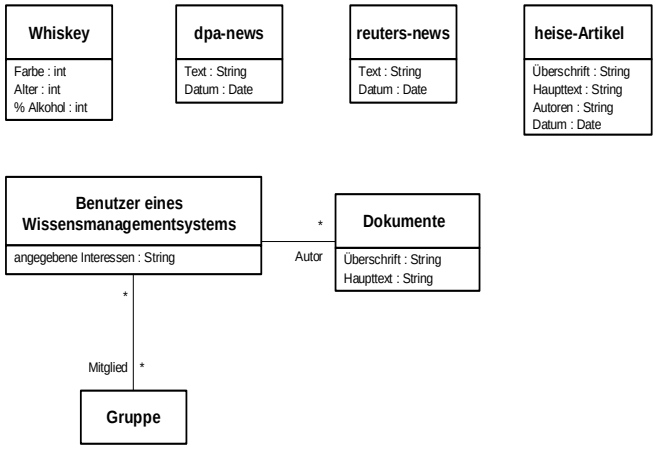

Abbildung 3.4: Typen von Informationsobjekten

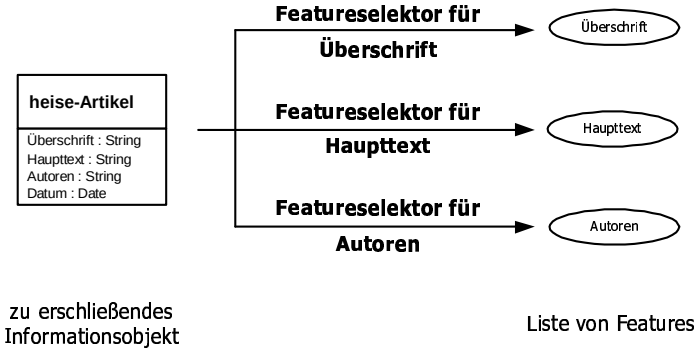

Abbildung 3.5: Featureselektoren

3.3.1.1 Isomorphe Informationsobjekttypen

Typen von Informationsobjekten sind *isomorph*, wenn sie die gleichen relevanten Typen von Features f_i haben. In Abbildung 3.4 sind die Typen *dpa-news* und *reuters-news* isomorph. Da für eine Erschließung nur die Features maßgeblich sind, lassen sich die Informationsobjekte beider Typen auf die gleiche Art und Weise erschließen.

3.3.1.2 Featureselektoren

Sogenannte Featureselektoren extrahieren die Features aus gegebenen Informationsobjekten. Pro relevantem Feature f_i gibt es einen Featureselektor. Als Beispiel ist in Abbildung 3.5 der Informationsobjekttyp *heise-Artikel* mit dazugehörigen Featureselektoren graphisch veranschaulicht. Dabei handelt es sich um Dokumente mit vier Features: einer Überschrift, einem Haupttext, einem Autor und einem Datum. Das Datum ist kein für eine Erschließung relevantes Feature, demzufolge existiert kein Featureselektor der es extrahiert.

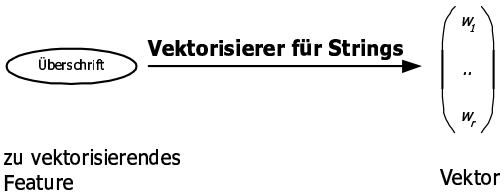

Abbildung 3.6: Stringvektorisierer

Bei Dokumenten des Typs *Benutzer eines Wissensmanagementsystems* aus Abbildung 3.4 können die für eine Erschließung relevanten Features von Featureselektoren aus den über Assoziationen erreichbaren Objekten gewonnen werden.

Der konkrete Typ der jeweils zu erschließenden Informationsobjekte (Art und Anzahl der Features) unterscheidet sich von Erschließungsaufgabe zu Erschließungsaufgabe. Die Features repräsentieren die zu vektorisierenden Informationen. Diese sind im allgemeinen von primitivem Typ (reelle Zahlen, Zeichenketten usw.). Zunächst werden die Features f_i vektorisiert, anschließend wird die resultierende Gesamtvektorrepräsentierung des Informationsobjektes aus den Teilvektorrepräsentierungen gebildet. Dieser Vorgang wird im Kapitel 3.3.3 näher erläutert.

Die Vektorisierung eines Informationsobjektes lässt sich also in die Vektorisierung der Features f_i und die anschließende Bildung einer Gesamtvektorrepräsentierung untergliedern.

3.3.2 Vektorisieren von Features

In diesem Abschnitt wird die Bildung von Vektoren aus Features f_i behandelt. Dies ist die Aufgabe der sogenannten *Vektorisierer*.

Ein Vektorisierer erzeugt einen Vektor anhand der gewichteten Features f_i. Diese Gewichtung wird durch ein $\eta_i \in \Re$ ausgedrückt. Die Aufgabe eines Vektorisierers lässt sich also formal beschreiben durch:

$$\overrightarrow{v} = g(\eta_1 f_i + \eta_2 f_i + .. + \eta_h f_i).$$

In Abbildung 3.6 ist der einfache Fall eines Vektorisierers, der das Feature *Überschrift* vektorisiert, graphisch dargestellt. In diesem Falle ist $\eta_1 = 1.0$. In Abbildung 3.7 erzeugt der selbe Vektorisierer einen Vektor aus zwei Features, es ist $\eta_1 = 2.0$ und $\eta_2 = 1.0$, entsprechend der vermutlich höheren Relevanz des Features *Überschrift*. Der Fall, dass ein Vektorisierer aus mehreren gewichteten Features einen Vektor erzeugt, ist im allgemeinen recht selten und macht auch nur für bestimmte Informationstypen Sinn. Wie dies im speziellen ausgestaltet wird, hängt vom konkreten Vektorisierer ab.

Allgemein kann man sagen, dass diese Form der Bildung eines Vektors angewendet werden sollte, wenn alle verwendeten Features f_i einen (möglicherweise unterschiedlich

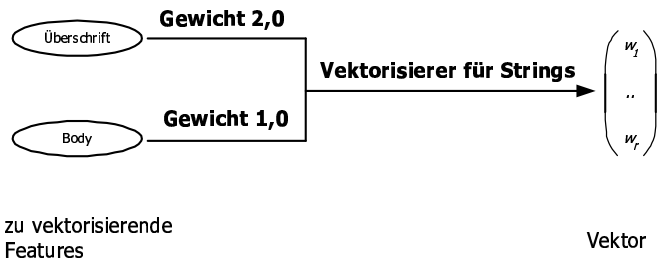

Abbildung 3.7: Stringvektorisierer mit zwei Attributen

wichtigen) Beitrag zu **einem** Aspekt des Informationsobjektes beitragen.

Im anderen Falle – die Attribute leisten voneinander unabhängige Beiträge zum Inhalt des gesamten Informationsobjektes – wird pro Feature ein Vektorisierer eingesetzt.

Wie bereits erwähnt, sind die Features eines Informationsobjektes meist von primitivem Typ. Im einfachsten Falle handelt es sich um eine reelle Zahl. In diesem Falle wird vom Vektorisierer in der Trainingsphase, wie in der Produktivphase, ein Vektor der Dimension eins mit dem entsprechenden Wert erzeugt. Ein weiteres recht einfaches Beispiel ist ein Features, welches diskrete nominale Informationen repräsentiert – beispielsweise das Alter eines Menschen. Mögliche Werte sind: *jung*, *mittel*, *alt*. In der Trainingsphase werden alle Features gelesen und jeder Ausprägung wird ein numerischer Wert zugeordnet, beispielsweise (jung; 0), (mittel; 1) und (alt; 2). Die entsprechende Trainingsmenge mit Vektoren der Dimension eins wird erzeugt, und die Zuordnungsinformationen werden gespeichert. In der Produktivphase erzeugt der Vektorisierer zu jedem Feature ebenfalls einen Vektor der Dimension eins mit dem Wert der entsprechenden Zuordnung.

In der Trainingsphase werden also Vektoren erzeugt und Informationen gesammelt. Anhand dieser Informationen können in der Produktivphase Vektoren erzeugt werden. In jedem Falle stimmen die Dimensionen der in Trainings- und Produktivphase erzeugten Vektoren überein.

3.3.2.1 Textindexierung

Da die Erschließung von natürlichsprachlichen Texten in dieser Arbeit eine Sonderstellung einnimmt, wird im folgenden die Vektorisierung für Features, die natürlichsprachliche Texte in Form einer Zeichenkette repräsentieren, erläutert. In der Trainingsphase liegen n natürlichsprachliche Texte vor. Zunächst werden die Texte vorverarbeitet. Es gibt die in Abbildung 3.8 dargestellten optionalen Möglichkeiten der Vorverarbeitung.

Im ersten Schritt werden die sogenannten Stoppwörter entfernt. Dies sind inhaltsleere Wörter (Artikel, Pronomina, Präpositionen, Adverbien, Konjunktionen), sowie weitere von der Indexierung ausgeschlossene Wörter.

In einem weiteren Schritt, der sogenannten Wortstammbildung werden die vorhande-

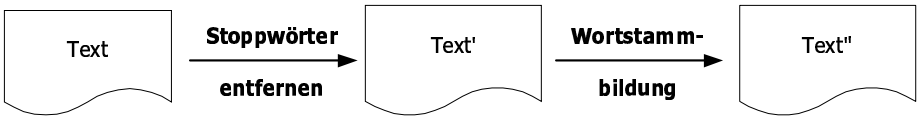

Abbildung 3.8: Vorverarbeitung von Texten

nen konkreten Wortformen auf ihre grammatische Grund- oder Stammform zurückgeführt. Dies kann im einfachsten Fall anhand von Regeln realisiert werden. Für englischsprachliche Texte existieren eine Reihe von gut funktionierenden Algorithmen. Der am weitesten verbreitete ist der Algorithmus von Porter ([Por80]). Aufgrund von Ausnahmen und Mehrdeutigkeiten vieler Wörter ist die Wortstammbildung in der deutschen Sprache durch allgemeine Regeln schwer möglich und fehleranfällig ([WSS97]). Alternativ existiert die Möglichkeit einer wörterbuchgesteuerten Wortstammbildung. Der Nutzen der Wortstammbildung ist jedoch umstritten, da die statistischen Eigenschaften der Terme verändert werden. Sie wird nicht von allen Systemen verwendet.

Aus den vorverarbeiteten Texten werden nun Vektoren gebildet. Formal betrachtet bedeutet dies, dass aus natürlichsprachlichen Texten $d_1, ..., d_i, ..., d_n$ numerische Vektoren $\overrightarrow{v_1}, ..., \overrightarrow{v_i}, ..., \overrightarrow{v_n}$ gebildet werden. Dabei ist:

$$\overrightarrow{v_i}(d_i) = \begin{pmatrix} v_{1i} \\ .. \\ v_{ki} \\ .. \\ v_{ri} \end{pmatrix}.$$

Die Anzahl verschiedener Worte im gesamten Dokumentenraum ist gleich r, und $v_{ki} \in R$ ist ein Maß für den Beitrag des k-ten Terms zum Inhalt des Dokumentes d_i. Im einfachsten Falle entspricht v_{ki} der Häufigkeit des Vorkommens des k-ten Terms im i-ten Dokument.

Es existieren ebenfalls Vorschläge einer ambitionierteren Repräsentierung natürlichsprachlicher Texte. In den durchgeführten Arbeiten ([ADW94], [Lew92]) wurden Gruppierungen von mehreren Wörtern als Indexierungssprache verwendet, die erzielten Ergebnisse zeigten jedoch keine erhöhte Retrievalqualität. Die Erklärung dafür mag sein, dass vorhandene Vorteile durch die semantisch höherwertige Repräsentierung durch Verschlechterung der statistischen Eigenschaften verloren gehen.

3.3.3 Bildung des Gesamtvektors

Der Grundgedanke einer Erschließung im Sinne dieser Arbeit besteht in der Vektorrepräsentierung verschiedenartigster Informationsobjekte. Die Konstruktion eines numerischen Vektors $\overrightarrow{v}$ in Abhängigkeit von einem Informationsobjekt ist dementsprechend eines der

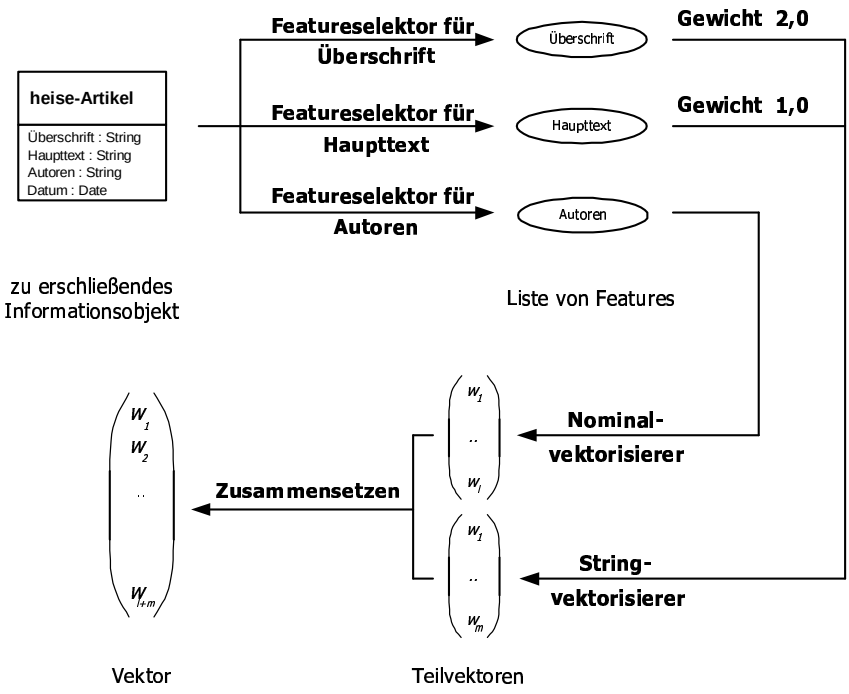

Abbildung 3.9: Vektorerzeugung

Kernkonzepte einer jeden Erschließung. Wie bereits dargelegt, werden zunächst die Attribute A_i von den Featureselektoren extrahiert. Zugeordnete Vektorisierer bilden anschließend Teilvektoren $\overrightarrow{v_j}$, aus denen dann der Gesamtvektor $\overrightarrow{v}$ gebildet wird. Als letztes wird der resultierende Gesamtvektor $\overrightarrow{v}$ durch Zusammensetzen der Teilvektoren $\overrightarrow{v_j}$ gebildet:

$$\overrightarrow{v} = \begin{pmatrix} \overrightarrow{v_1} \\ \overrightarrow{v_2} \\ .. \\ \overrightarrow{v_n} \end{pmatrix}.$$

3.3.4 Zusammenfassung

Der Schritt der Vektorerzeugung lässt sich wiederum in zwei Vorgänge unterteilen. Zuerst werden alle Attribute eines Informationsobjektes vektorisiert. In einem zweiten Schritt wird der Gesamtvektor aus den erzeugten Teilvektoren gebildet. Ein gesamter Beispielvorgang ist in Abbildung 3.9 dargestellt. Das Informationsobjekt besteht aus drei relevanten Features: Überschrift, Haupttext und Autoren. Die Attribute werden mit Hilfe von Featureselektoren gewonnen und anschließend von Vektorisierern zu Vektoren weiterverarbeitet. Da die Überschrift und der Haupttext einen gemeinsamen Beitrag zum Inhalt des Dokumentes beitragen, werden sie mit dem selben Vektorisierer verarbeitet. Die Autorschaft stellt einen zusätzlichen Aspekt dar, sie wird von einem eigenen Vektorisierer

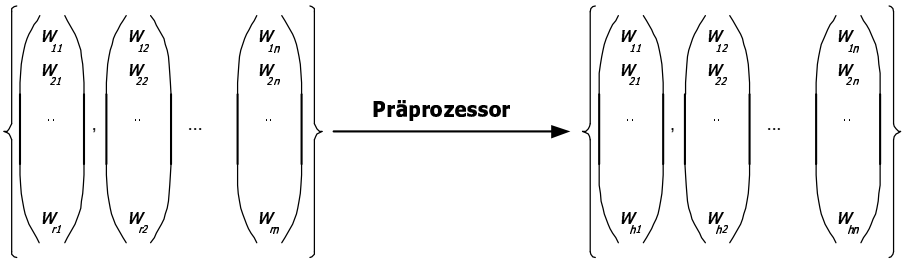

Abbildung 3.10: Vektorvorverarbeitung in der Trainingsphase

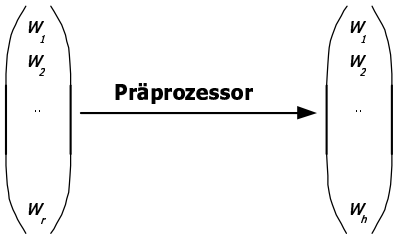

Abbildung 3.11: Vektorvorverarbeitung in der Produktivphase

vektorisiert.

Die entstehenden Teilvektorrepräsentierungen werden in einem letzten Schritt zu einem resultierenden Vektor zusammengesetzt.

3.4 Vektorvorverarbeitung

Im zweiten Schritt einer Erschließungsaufgabe werden die als Resultat der Vektorerzeugung gewonnenen Vektoren vorverarbeitet. Dies geschieht durch sogenannte *Präprozessoren*.

In der Trainingsphase wird vom Präprozessor eine gegebene Trainingsmenge verarbeitet, und eine Trainingsmenge als Resultat zurückgeliefert (siehe Abbildung 3.10). In der Produktivphase besteht die Aufgabe eines Präprozessors darin, einen einzelnen gegebenen Vektor entsprechend zu verarbeiten (siehe Abbildung 3.11).

Es gibt folgende Arten der Vorverarbeitung:

- Entfernung von fehlerhaften Vektoren (siehe Kapitel 3.4.1).
- Einfache mathematische Abbildungen (siehe Kapitel 3.4.2 und 3.4.4).
- Merkmalsreduktion (siehe Kapitel 3.4.3).

Die vorgestellten Arten der Vektorvorverarbeitung sind zueinander orthogonal, es können mehrere Vorverarbeitungsschritte hintereinander durchgeführt werden, die in ihrer Gesamtheit dann die komplette Vektorvorverarbeitung bilden. Beispielsweise ist in Abbildung 3.12 eine Vektorvorverarbeitung in der Produktivphase dargestellt, in der zunächst eine Merk-

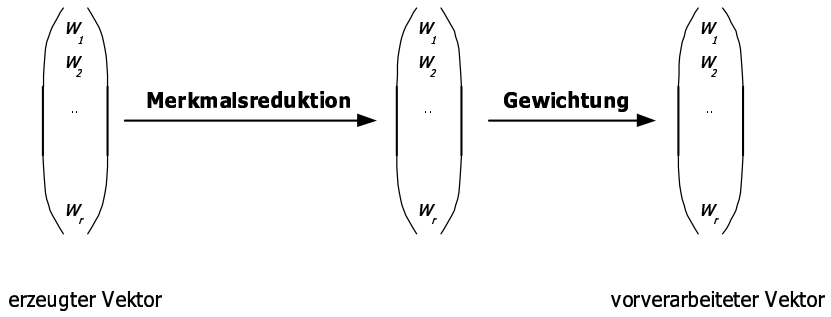

Abbildung 3.12: Vektorvorverarbeitung in zwei Schritten

malsreduktion und anschließend eine Gewichtung stattfindet.

3.4.1 Ausreißerbehandlung

Zunächst müssen eventuelle Ausreißer erkannt werden, anschließend gibt es verschiedene Möglichkeiten der Behandlung. Eine eingehendere Behandlung dieser Möglichkeiten wird in [Run00] durchgeführt.

3.4.2 Gewichtung

Als Resultat des in 3.3.2.1 vorgestellten probabilistischen Ansatzes liegen Vektoren $\overrightarrow{w_i}(d_i)$ vor, deren Werte w_{ki} der Anzahl Vorkommen des k-ten Terms im i-ten Dokument entsprechen. Die gewonnenen Vektoren werden anschließend gewichtet, um den unterschiedlichen Beiträgen der Terme zum Inhalt der Dokumente gerecht zu werden (siehe auch [Sal89]). Eine übliche Wahl ist der *Inverse Document Frequency* Algorithmus:

$$w_{ki} = w_{ki} \log\left(\frac{|D_0|}{\#(d_k)}\right),$$

wobei $|D_0|$ die Menge aller Dokumente ist, und $\#(d_k)$ die Anzahl von Dokumenten in D_0 meint, in denen der Term t_k mindestens einmal vorkommt. Der Hauptgedanke dieses Algorithmus' besteht darin, häufiger vorkommende Terme geringer zu gewichten als seltenere.

3.4.3 Merkmalsreduktion

Die Menge unterschiedlicher Terme in einem großen Dokumentenstamm kann mehrere hunderttausend erreichen. Daraus ergeben sich sehr hochdimensionale Vektoren, deren Analyse selbst mit effizienten Algorithmen undurchführbar ist. Außerdem zeigen Untersuchungen, dass die vorkommenden Terme zu unterschiedlichen Anteilen zum semantischen Gehalt eines Dokumentes beitragen ([Sal89]). Eine Methode zur Lösung dieser Probleme ist die sogenannte Merkmalsreduktion. Es existieren eine Reihe von vorgeschlagenen Algorithmen, die sich folgendermaßen klassifizieren lassen:

- Lokale Merkmalsreduktion: die durchgeführte Reduktion erfolgt spezifisch pro Kategorie.
- Globale Merkmalsreduktion: es wird eine globale Reduktion für alle Kategorien durchgeführt.

Die Entscheidung zugunsten einer lokalen oder globalen Merkmalsreduktion ist unabhängig von der Wahl des verwendeten Algorithmus. Ein weiterer Gesichtspunkte unterscheidet bezüglich der Art der Reduktion:

- Term Selection: es wird eine Teilmenge aller Terme ausgewählt
- Term Extraction: die resultierende Menge von Termen wird als Resultat einer Transformation oder Kombination aus den originalen Termen gewonnen.

3.4.3.1 Term Selection

Die im folgenden vorgestellten Algorithmen benutzen unterschiedliche Kriterien $f(t_k, c_i)$, anhand derer die inhaltliche Relevanz eines Terms t_k in einer Kategorie c_i gemessen wird. Wird eine lokale Merkmalsreduktion durchgeführt, so ist $f(t_k, c_i)$ schon das Kriterium, anhand dessen entschieden wird. Im Falle einer globalen Merkmalsreduktion gibt es die folgenden Möglichkeiten der Bildung eines Kriteriums:

- Summe über alle Kriterien: $f_{sum}(t_k) = \sum_{i=1}^{m} f(t_k, c_i)$,
- gewichteter Durchschnitt: $f_{avg}(t_k) = \sum_{i=1}^{m} f(t_k, c_i)P(c_i)$,
- Maximumbildung: $f_{\max}(t_k) = \max f(t_k, c_i)$.

Liegt der Wert $f(t_k)$ unterhalb einer vorgegebenen Schwelle, werden alle Werte der Dimension k aus der Trainingsmenge entfernt.

Document Frequency (DF) Die *Document Frequency* $f(t_k, c_i) = DF(t_k, c_i) = P(t_k \mid c_i)$ eines Terms t_k und einer Kategorie c_i gibt an, in wie vielen verschiedenen Dokumenten d_i einer Kategorie ein Term t_k vorkommt. Die Grundannahme ist, dass seltene Terme weniger semantische Relevanz haben. DF ist die einfachste Form der Merkmalsreduktion.

Information Gain (IG) Mit dem *Information Gain* eines Terms berechnet man den Informationsgewinn, den das Vorhandensein eines Terms für die Vorhersage der zugehörigen Kategorie mit sich bringt. Der Information Gain eines Terms t_k bzg. der Kategorie c_i berechnet sich zu:

$$f(t_k, c_i) = IG(t_k, c_i) = P(t_k, c_i) \log \frac{P(t_k, c_i)}{P(c_i)P(t_k)} + P(\overline{t_k}, c_i) \log \frac{P(\overline{t_k}, c_i)}{P(c_i)P(\overline{t_k})}.$$

Die Zeitkomplexität der Berechnung der Entropien beträgt $O(rm)$, wobei r die Anzahl verschiedener Terme ist und m die Anzahl vorhandener Kategorien.

Mutual Information (MI) Die *Mutual Information* eines Terms berechnet sich zu:

$$f(t_k, c_i) = MI(t_k, c_i) = \log \frac{P(t_k, c_i)}{P(t_k)P(c_i)}.$$

χ^2 statistic (CHI) Der *Chi-square* Wert eines Terms berechnet sich zu:

$$\chi^2(t_k, c_i) = \frac{g\left[P(t_k, c_i)P(\overline{t_k}, \overline{c_i}) - P(t_k, \overline{c_i})P(\overline{t_k}, c_i)\right]^2}{P(t_k)P(\overline{t_k})P(c_i)P(\overline{c_i})}.$$

Vergleich der Algorithmen Ein Vergleich der Leistungsfähigkeit der Algorithmen wurde in [YP97] durchgeführt. Es ergaben sich sehr gute Resultate für DF, IG und CHI.

3.4.3.2 Term Extraction

Mit Mitteln der *Term Extraction* wird aus einer gegebenen Menge von r Termen eine neue, kleinere Menge von $r\prime \ll r$ synthetischen Termen gewonnen. Der Grundgedanke dieser Ansätze besteht darin, dass sich die originalen Terme aufgrund der Eigenschaften natürlichsprachlicher Texte (Synonyme, Homonyme usw.) als Grundlage einer Repräsentierung nur bedingt eignen. Die als Resultat der Extraktion erzeugten Terme weisen die angesprochenen problematischen Eigenschaften zu einem kleineren Teil auf. Es gibt zwei Ansätze zur Durchführung einer Term Extraction: Term Clustering und Latent Semantic Indexing.

Term Clustering ordnet Terme mit einem großen Grad semantischer Abhängigkeit in Clustern an, und die erzeugten Cluster werden als synthetische Terme verwendet. Es lassen sich zwei unterschiedliche Verfahren unterscheiden:

- unsupervised (siehe [LJ98] und [Lew92]) und
- supervised (siehe [BM98]).

Latent Semantic Indexing erzeugt einen niedrigdimensionaleren Merkmalsraum durch Kombination der originalen Merkmale (siehe [SHP95]).

3.4.4 Normalisierung

Die gewichteten Vektoren werden oftmals normalisiert, um eine Gleichbehandlung langer und kurzer Dokumente zu ermöglichen:

$$w_{ki} = \frac{w_{ki}}{\sqrt{\sum_{s=1}^{r}(w_{si})^2}}.$$

3.5 Vektoranalyse

Die Vektoranalyse ist der jeweils letzte Schritt, sowohl der Trainings- als auch der Produktivphase. Die Vektoranalyse wird von der sogenannten Vektormaschine durchgeführt. In der Trainingsphase wird die Vektormaschine mit einer in Vektorerzeugung und -vorverarbeitung erzeugten Trainingsmenge trainiert. Die Art dieses Trainings unterscheidet sich für unterschiedliche Erschließungsformen. Die eigentliche Erschließung eines Informationsobjektes erfolgt dann in der Produktivphase anhand eines erzeugten Vektors. Die getroffenen Entscheidungen unterscheiden sich ebenfalls für unterschiedliche Erschließungsformen.

3.5.1 Klassifikation

Im Falle einer Klassifikationsaufgabe analysiert der Klassifikator die Zugehörigkeit eines Vektors zu den existierenden Kategorien und versucht Informationen über statistische Verteilungen der Vektoren in den Kategorien zu erlangen. In der Produktivphase einer Klassifikationsaufgabe macht die Vektormaschine Aussagen über die Zugehörigkeit des gebildeten Vektors (und damit des dazugehörigen Informationsobjektes) zu einer der gelernten Kategorien. Im folgenden werden kurz die gebräuchlichsten Klassifikationsalgorithmen beschrieben.

Ein Vergleich der Ergebnisse der verschiedenen Algorithmen im Kontext einer Textklassifikationsaufgabe wurde in [Yan99] und [YL99] durchgeführt.

3.5.1.1 Bayes Klassifikator

Diese Klassifikatoren nutzen das Bayessche Theorem, welches Aussagen über bedingte Wahrscheinlichkeiten trifft. Die Einzelheiten dieser Algorithmen werden in [Sch96] vorgestellt.

3.5.1.2 k-Nearest-Neighbor (kNN)

Ein kNN-Klassifikator findet zu einem gegebenen Vektor die k nächsten Vektoren. Aufgrund der Klassenzugehörigkeit der k Vektoren wird die Klassifikationsentscheidung getroffen. Der einfachste algorithmische Ansatz zur Realisierung eines kNN-Klassifikators ist die Distanzberechnung zwischen allen Trainingsvektoren und dem gegebenen Vektor. Die damit verbundenen Performanceprobleme sind offensichtlich. Für niedrigdimensionale Räume (Dimension$<\tilde{}20$) existieren Algorithmen die in der Trainingsphase Datenstrukturen aufbauen, wodurch ein effizienter Zugriff ermöglicht wird (siehe [KS97] und [BEK 98]).

Für hochdimensionale Räume ist bisher keine effiziente Lösung des Problems bekannt, es existieren jedoch Ansätze zur Lösung des Problems der Approximate Nearest Neighbor Search (siehe [Kle97], [KOR98] und [IM98]).

3.5.1.3 Support Vector Machine

Der Grundgedanke der Support Vector Machines besteht in einer bestmöglichen (auch nichtlinearen) Separierung des Raums. Die Details dieses Verfahrens werden in [Bur98] dargelegt. Die erzielten Resultate zeigen das große Potential dieses Algorithmus auf (siehe dazu [Joa98b], [Joa98a], [Joa99], und [SBS98]).

3.5.2 Clustering

Ist die Erschließung eine Clusteraufgabe, so analysiert die Vektormaschine die Verteilung der Vektoren in der erzeugten Trainingsmenge. Informationen über Zugehörigkeiten zu Kategorien sind nicht vorhanden und werden auch nicht benötigt, gilt es doch genau diese zu synthetisieren. Resultat des Trainings ist eine trainierte Vektormaschine, die intern Cluster gebildet hat, und die Vektoren der Trainingsmenge diesen Clustern zuordnen kann.

In der Produktivphase wird zunächst pro Cluster eine neue Kategorie erzeugt. Anschließend werden die Vektoren aller Informationsobjekte vom Vectorclusterer analysiert und den neu erzeugten Kategorien zugeordnet. Die Vektormaschine ist ebenfalls in der Lage, neue vektorisierte Informationsobjekte den Kategorien zuzuordnen.

Verwendete Algorithmen werden u.a. in [Run00] und in [WF99] beschrieben.

3.5.2.1 Visualisierung

Eine weitere Aufgabenstellung ist die Visualisierung der gefundenen Cluster und der enthaltenen Dokumente. Ein solches Vorgehen wird in [MH00] und [DK98] beschrieben.

3.5.3 Assoziative Suche

Im Falle einer Assoziativen Suche werden Informationen über Zugehörigkeiten zu Kategorien ebenfalls nicht benötigt. Die Ähnlichkeitssuchmaschine "merkt" sich in der Trainingsphase die Verteilung der Vektoren der Trainingsmenge. In der Produktivphase findet die Vektormaschine zu einem gebildeten Vektor (einem zu erschließenden Informationsobjekt) eine Reihe von Vektoren (Informationsobjekten), deren Abstand entsprechend eines bestimmten Distanzmaßes minimal ist.

Die zur Realisierung benötigte Funktionalität ist die eines kNN-Klassifikators, es werden jedoch die gefundenen Vektoren (Informationsobjekte) direkt zurückgeliefert.

3.6 Zusammenfassung

Eine Erschließungsaufgabe unterteilt sich in zwei Phasen, die Trainings- und die Produktivphase. In beiden Phasen lassen sich drei voneinander unabhängige Teilschritte abgrenzen: die Vektorerzeugung, die Vektorvorverarbeitung und die anschließende Vektoranalyse. Konkrete Erschließungsaufgaben unterscheiden sich hinsichtlich der Art der zu er-

schließenden Informationsobjekte und der konkreten Erschließungsaufgabe (Klassifikation, Clustering oder Assoziative Suche). Die Auswirkungen dieser beiden Abhängigkeiten auf eine konkrete Umsetzung lassen sich einzelnen Teilschritten zuordnen. Der Teilschritt der Vektorerzeugung hängt von der Art der zu erschließenden Informationsobjekte ab. Eine Implementierung des Teilschrittes der Vektoranalyse hängt von der Art der Erschließung ab. Die Vektorvorverarbeitung ist ein generischer Teilschritt.

Eine softwaretechnische Umsetzung der genannten Erschließungsformen sollte die genannten drei Teilschritte einer Erschließung voneinander abgrenzen.

Kapitel 4

Softwaretechnische Umsetzung - ADF

Als softwaretechnische Umsetzung der im vorangegangenen Kapitel geschilderten Schritte einer Erschließungsaufgabe empfiehlt sich ein Framework. Im Rahmen dieser Diplomarbeit wurde ein objektorientiertes Framework konzipiert und in Java implementiert. Das Framework trägt den Name **ADF** - **A**sset **D**iscovery **F**ramework.

In diesem und im nächsten Kapitel wird das Design des **ADF** vorgestellt. Zunächst folgen in Abschnitt 4.1 einige Betrachtungen zu Frameworks im allgemeinen. Im Abschnitt 4.2 werden die an das Framework gestellten Anforderungen geschildert.

Anschließend werden die Umsetzungen der drei Teilschritte einer Erschließungsaufgabe in ADF erläutert.

4.1 Frameworks

Zentrale Ziele der Software-Entwicklung sind Wiederverwendbarkeit und Wartbarkeit. Frameworks stellen einen Beitrag zur Verwirklichung dieser Ziele dar. Ralph E. Johnson definiert ein Framework folgendermaßen [JF88]:

"Ein Framework ist eine Menge von Klassen, die ein abstraktes Design für Lösungen zu einer Familie verwandter Probleme enthalten[..]"

Ein Framework grenzt sich von einer Klassenbibliothek dadurch ab, dass zwischen den Klassen vielfältige Beziehungen existieren. Diese Beziehungen stellen das abstrakte Design dar. Konkrete Anwendungssysteme werden im wesentlichen durch Komposition und Konkretisierung existierender Klassen des Frameworks erstellt. Daraus ergibt sich, dass der Entwurf, der durch die abstrakten Klassen des Frameworks und deren Beziehungen untereinander gegeben ist, wiederverwendet werden kann. Ein Framework ist also eine vorgefertigte Struktur eines konkreten Systems.

Weiterhin verfügen Frameworks über eine eigene Ablaufsteuerung, der Grundsatz lautet: *"The Framework calls you, don't call the framework."* Der eigene erstellte Code spezifiziert lediglich das Verhalten des Frameworks an bestimmten Stellen, die Verantwortlichkeit zur Ablaufsteuerung liegt beim Framework.

4.1.1 Klassifizierung von Frameworks

Frameworks lassen anhand verschiedener Gesichtspunkte klassifizieren. Eine Möglichkeit ist die Klassifizierung nach der Problemdomäne, die das Framework abbildet [Nem97]:

- **Application Frameworks** bilden das architektonische Gerüst einer Anwendung. Beispielhaft seien Application-Frameworks für grafische Benutzeroberflächen (GUIs) genannt, welche Standard-Funktionalitäten, die von allen Anwendungen mit grafischer Benutzeroberfläche verwendet werden, unterstützen. Ein Application-Framework ist unabhängig von der Problemdomäne. Ein Beispiel ist das Application-Framework Open Windows Look (OWL) von Borland.

- **Domain Frameworks** liefern eine Lösung für ein bestimmtes Anwendungsgebiet, beispielsweise das Finanzwesen.

- **Support Frameworks** kapseln Dienste für die betriebssystemnahe Programmierung (z.B. Dateizugriff, Datenbankanbindung oder Kommunikationsdienste).

Eine weitere Klassifikationsmöglichkeit ist die Art und Weise der Anpassung des Frameworks an konkrete Aufgabenstellungen (siehe [JF88]). Die Punkte, an denen ein Framework angepasst wird werden **Hot Spots** genannt. Es existieren folgende Arten von Frameworks:

- **White-Box Frameworks** werden durch Konkretisierung vorhandener abstrakter Klassen instanziiert. Zu erstellende konkrete Subklassen spezifizieren das Verhalten des Systems durch Überschreiben von Methoden. Vorteilhaft ist die große Flexibilität von White-Box Frameworks, andererseits ist eine detaillierte Kenntnis über die Klassenstruktur notwendig.

- **Black-Box Frameworks** stellen eine Menge direkt zu benutzender Klassen zur Verfügung, die zur Instanziierung des Frameworks lediglich ausgewählt und konfiguriert werden müssen. Es ist kein Wissen über die innere Struktur des Frameworks notwendig, andererseits sind Black-Box Frameworks nicht so flexibel anpassbar wie White-Box Frameworks.

- **Glass-Box Frameworks** sind eine Kombination aus den genannten Frameworktypen. Zur Instanziierung werden sowohl Komposition als auch Konkretisierung verwendet.

4.1.2 Vorteile von Frameworks

- Der Einsatz von Frameworks erlaubt die Wiederverwendung von Programmentwurf und -code auf einer höheren Ebene, als dies andere Strategien erlauben. Dies führt zu deutlich verringerten Entwicklungszeiten für konkrete Anwendungen (Reduced time to market).
- Die auf Basis eines Frameworks erstellten Anwendungen unterscheiden sich nur in relativ geringen Teilen ihres Codes. Die Wartung des Codes beschränkt sich somit auf kleinere überschaubarere Programmteile.
- Aufgrund der Wiederverwendung des Frameworkcodes verbessert sich dessen Qualität im Laufe der Zeit.

4.1.3 Nachteile

- Die Entwicklungszeit eines Frameworks liegt deutlich über der einer normalen Applikation. Die Entwicklung eines Frameworks lohnt sich nur dann, wenn eine Reihe von Anwendungen auf Basis des Frameworks erstellt werden sollen.
- Aufgrund der vorhandene Eigendynamik eines Frameworks ist die Einarbeitungszeit relativ lang. Es existiert keine allgemein anerkannte Art und Weise der Dokumentation eines Frameworks.

4.2 Anforderungen an das ADF

1. **Anpassbarkeit an verschiedenste Anwendungen**

   Das erstellte Framework **ADF** soll den Prozess der Erschließung von Informationsobjekten unterstützen. Die Grundidee der Erschließung besteht in der Erzeugung einer Vektorrepräsentierung der Informationsobjekte, deren Vorverarbeitung und anschließender Analyse. Die vorgestellte Vorgehensweise ist auf beliebige Typen von Informationsobjekten anwendbar. Das erstellte Framework **ADF** soll sich leicht an Anwendungen, die um Erschließungsfunktionalitäten erweitert werden sollen, anpassen lassen. Dies kann man sich wie in Abbildung 4.1 dargestellt vorstellen. Die offensichtliche Kommunikation zwischen Framework und Anwendung im Verlauf einer Erschließung beschränkt sich auf zwei Fälle. Die Anwendungen liefern die zu erschließenden Informationsobjekte, und das Framework liefert die Resultate der Erschließungsaufgabe an die Anwendung zurück (siehe Abbildung 4.2).

2. **Modulare Unterstützung des Erschließungsprozesses**

   Da sich ein Erschließungsprozess, wie in Kapitel 3.2 dargestellt, in drei Schritte unterteilen lässt, müssen die drei Schritte Vektorerzeugung, Vektorvorverarbeitung und Vektoranalyse voneinander abgegrenzt und einzeln unterstützt werden.

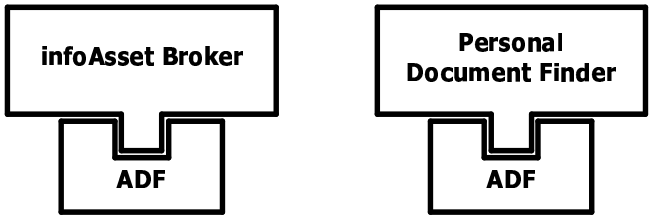

Abbildung 4.1: ADF mit verschiedenen Anwendungen

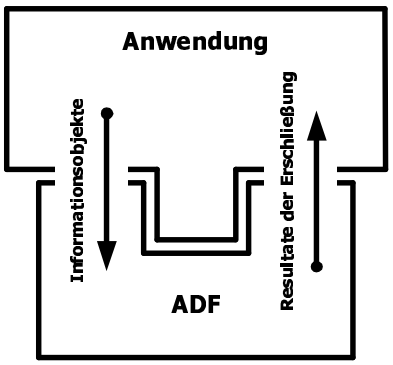

Abbildung 4.2: Kommunikation zwischen Framework und Anwendung

3. **Multilingualität**

   Eine weitere Forderung ergibt sich aus der Sonderstellung, die die Erschließung natürlichsprachlicher Texte einnimmt. Es ist wünschenswert, Texte verschiedener Sprachen transparent zu erschließen. Das System soll erkennen, in welcher Sprache ein Text verfasst ist, und entsprechend unterschiedlich behandeln.

4. **Verschiedene Informationsobjekttypen**

   Weiterhin soll das Framework die Fähigkeit haben, mehrere verschiedene isomorphe Typen von Informationsobjekten (siehe Kapitel 3.3.1.1) transparent zu erschließen. Die verschiedenen Typen haben alle die gleichen Typen von Features, demzufolge lassen sie sich in der gleichen Art und Weise erschließen.

5. **Performance**

   Die Akzeptanz eines durch Erschließung unterstützten Retrievalprozesses hängt entscheidend von der möglichen Arbeitsgeschwindigkeit und vom Ressourcenverbrauch ab. Die Notwendigkeit effizienter Algorithmen und entsprechender Datenstrukturen liegt auf der Hand.

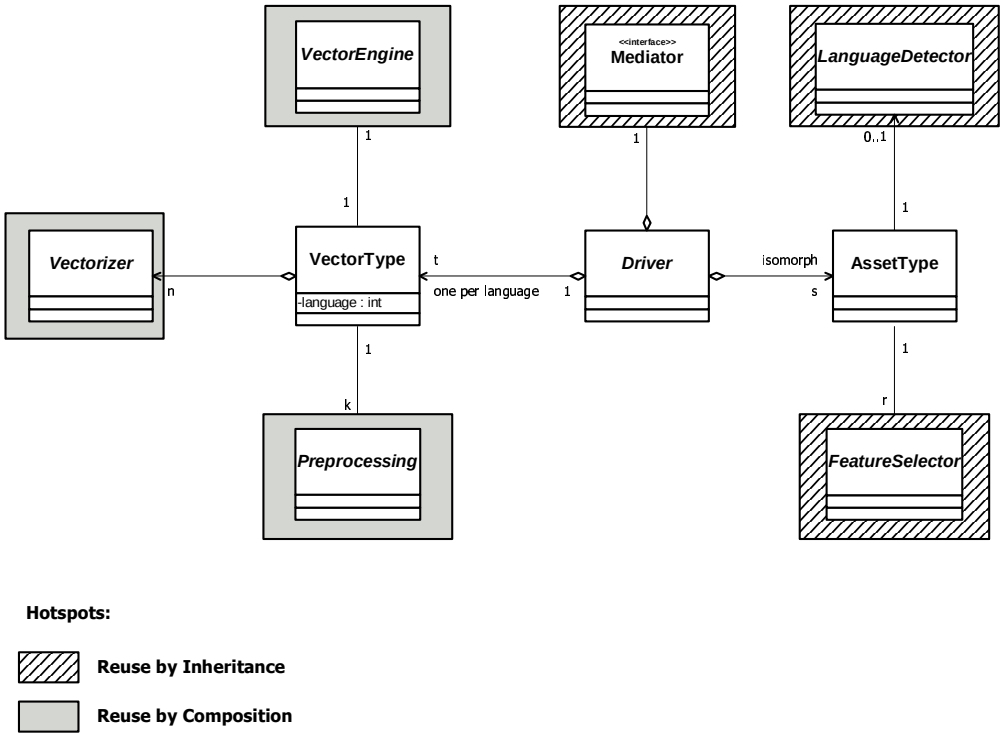

Abbildung 4.3: Überblick über wichtige Klassen von ADF

4.3 Klassen und Pakete

Einen groben Überblick über die wichtigsten Klassen von **ADF** gibt das UML-Klassendiagramm in Abbildung 4.3. Die im Diagramm dargestellten Klassen und ihre Abhängigkeiten werden im folgenden näher erklärt. Das Diagramm soll im folgenden als Überblicksdarstellung dienen. Die zwei verschiedenen Arten von Hot Spots sind gekennzeichnet.

In Abbildung 4.4 sind die entstandenen Java-Packages und ihre Abhängigkeiten untereinander dargestellt.

4.4 Vektorrepräsentierung

Der Ansatz einer Erschließung von Informationsobjekten mit **ADF** besteht in der Repräsentierung von Informationsobjekten durch numerische Vektoren. In diesem Abschnitt wird die softwaretechnische Umsetzung dieser Vektorrepräsentierung erläutert.

Ein Vektor wird in **ADF** durch ein Objekt der Klasse `Vector` repräsentiert. Die im Verlauf einer Trainingsphase erstellte Menge von Vektoren wird durch ein Objekt der Klasse `TrainingSet` repräsentiert. Dies ist in Abbildung 4.5 dargestellt. Ein Vektor kann sowohl allein, als auch als Teil einer Trainingsmenge existieren.

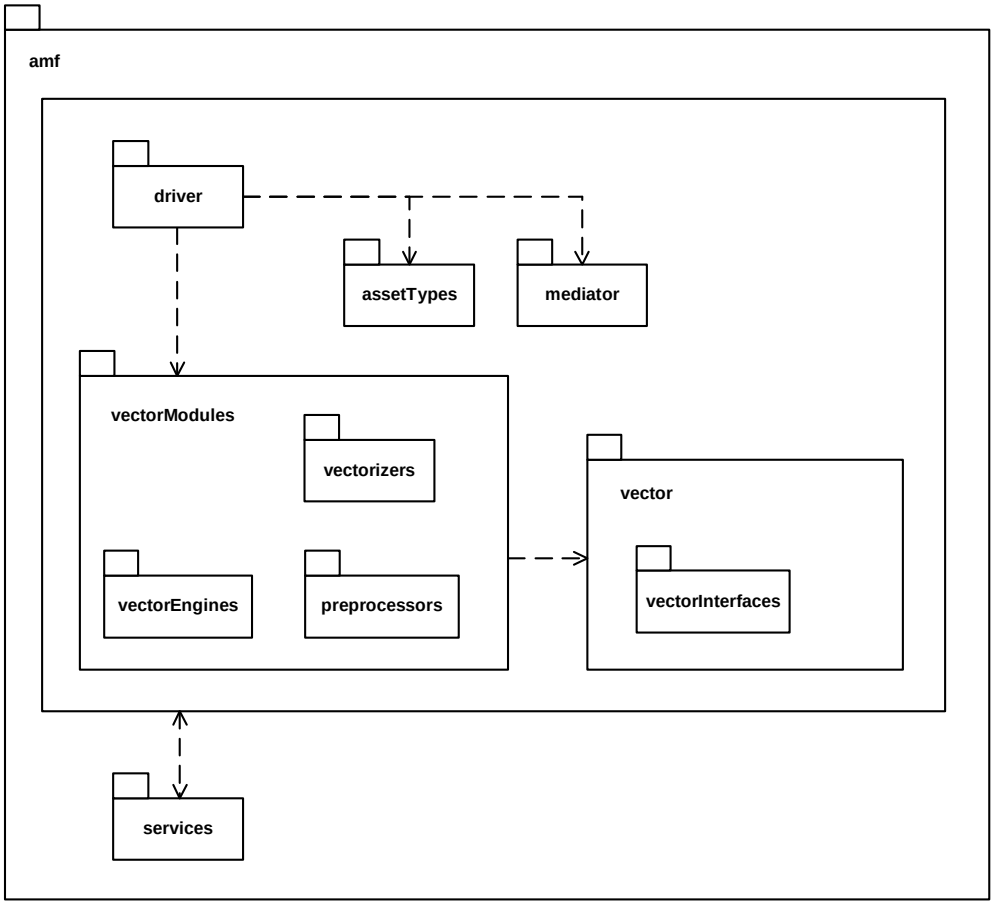

Abbildung 4.4: Packagediagramm

Abbildung 4.5: Konzeptuelles Klassendiagramm Trainingset-Vektor

4.4.1 Anforderungen an die Repräsentierung

Aufgrund der zentralen Bedeutung der Vektorrepräsentierung werden hohe Anforderungen an eine effiziente Umsetzung gestellt. Eine effiziente Realisierung hängt im wesentlichen von den verwendeten Datenstrukturen und Algorithmen ab.

Konzeptuell besteht ein Vektor $\overrightarrow{w}$ aus einer geordneten Menge von reellen Zahlen:

$$\overrightarrow{w} = \begin{pmatrix} w_1 \\ .. \\ w_k \\ .. \\ w_r \end{pmatrix}.$$

Für die Elemente w_i gilt ganz allgemein $w_i \in R$. Die an eine Datenstruktur gestellten Anforderungen lassen sich in die zwei Teilaspekte Platzbedarf und Zeitbedarf untergliedern.

Der Platzbedarf hängt in erster Linie von der Repräsentierung der Elemente des Vektors ab, darauf wird im folgenden in Kapitel 4.4.1.1 näher eingegangen. Weiterhin ergeben sich Optimierungsspielräume aus dem Grad der Spärlichkeit des zu repräsentierenden Vektors. Im Falle der Erschließung von Textdokumenten ist mit sehr spärlich besetzten hochdimensionalen Vektoren zu rechnen. Dies kann durch entsprechende Datenstrukturen ausgenutzt werden.

Der zweite Aspekt ist der Zeitbedarf für bestimmte Operationen. Mögliche Operationen sind einerseits das Einfügen von Elementen beim Erzeugen der Vektoren und andererseits das Zugreifen auf Elemente des Vektors zur anschließenden Manipulation.

Es besteht im allgemeinen ein Zielkonflikt zwischen verschiedenen Kriterien, es ist nicht möglich, eine Datenstruktur gemäß aller Kriterien gleichzeitig zu optimieren. Daraus ergibt sich die Forderung nach verschiedenen Repräsentierungen.

4.4.1.1 Verwendeter Datentyp

Zunächst ist eine Entscheidung über die Repräsentierung der Elemente w_i zu treffen. Diese Entscheidung hat in erster Linie Einfluss auf den benötigten Platzbedarf der Vektorrepräsentierung. In Java existieren vier verschiedene primitive Datentypen zur Repräsentierung ganzzahliger numerischer Zahlen (`long`, `int`, `short` und `byte`) und zwei primitive Datentypen zur Repräsentierung von Fließkommazahlen (`double` und `float`). Diese Datentypen unterscheiden sich durch den Platzbedarf und den abbildbaren Wertebereich bzw. die erreichte Genauigkeit. In **ADF** existieren Implementierungen von Vektoren für die primitiven Datentypen `double`, `float` und `short`.

4.4.1.2 Datenstruktur

Die zweite zu treffende Entscheidung ist vom verwendeten Datentyp unabhängig und betrifft die Art und Weise der Speicherung der Elemente. Die naheliegendste Lösung ist die

Speicherung in einem Array. Diese Lösung erfüllt sicherlich fast alle Forderungen nach effizientem Zugriff und Manipulation der Elemente. Im Falle eines spärlich besetzten Vektors ist eine solche Datenstruktur jedoch platzineffizient. In einem solchen Falle empfiehlt es sich, lediglich die Elemente mit Werten ungleich 0 zu speichern. Hierfür gibt es wiederum verschiedene Möglichkeiten, auf die jedoch an dieser Stelle nicht weiter eingegangen werden soll. In Abhängigkeit von der verwendeten Datenstruktur unterscheiden sich die effizienten Zugriffsmöglichkeiten auf die repräsentierten Daten.

Als Schlussfolgerung lässt sich feststellen, dass eine hohe Flexibilität bezüglich Datentyp, interner Datenstruktur und Zugriffsoperationen einer softwaretechnischen Umsetzung der Vektorrepräsentierung gefordert ist.

4.4.2 Softwaretechnische Umsetzung

Die Herausforderung einer softwaretechnischen Umsetzung der Vektorrepräsentierung besteht darin, dass einerseits aus Performancegründen ein Höchstmaß an Flexibilität bezüglich der verwendeten Datentypen, Datenstrukturen und Zugriffsoperationen zu gewährleisten ist. Andererseits benötigen vektorverarbeitende und vektorerzeugende Module feststehende Zugriffsmöglichkeiten, und es muss Typsicherheit gewährleistet werden. Diese Ziele erfüllt das in **ADF** implementierte Konzept der Vektorrepräsentierung.

Der Grundgedanke besteht darin, eine breite Vektorschnittstelle zur Verfügung zu stellen, die alle möglichen Methoden bzgl. aller Datentypen zur Verfügung stellt. Weiterhin existieren viele verschiedene Vektorimplementierungen, die ihrerseits nur eine kleine Menge der Zugriffsmethoden unterstützen. Benötigt werden die Zugriffsmethoden von Vektormodulen. Eine konkrete Konfiguration des Frameworks besteht aus einer Abfolge bestimmter Module einerseits, und zugewiesenen Vektorimplementierungen andererseits. Jedes Vektormodul benötigt wiederum eine kleine Menge aller von der Vektorschnittstelle zugewiesenen Methoden. Typsicherheit wird gewährleistet, indem im Moment der Initialisierung des Frameworks die Kompatibilität von Vektormodulen und zugewiesenen Vektorimplementierungen überprüft wird.

In Gestalt der Zugriffsmethoden treffen die Vektorimplementierungen und die Vektormodule aufeinander. Aus Gründen der Übersichtlichkeit werden Zugriffsmethoden, die ein bestimmtes Zugriffsszenario unterstützen, zu einem `VectorInterface` gebündelt. Die Zusammenhänge zwischen einer Vektorimplementierung, den Vektorschnittstellen und den Vektormodulen sind im konzeptuellen Diagramm in Abbildung 4.25 dargestellt.

4.4.2.1 Abstrakte Klasse `Vector`

Alle Vektorrepräsentierungen sind vom Typ `Vector`. `Vector` ist eine abstrakte Klasse (siehe Abbildung 4.6), die zunächst keinerlei numerische Daten verwaltet. Dies wird von konkreten Subklassen geleistet.

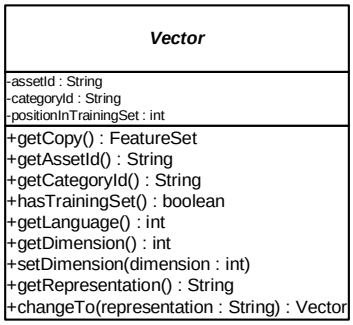

Abbildung 4.6: Vector

4.4.2.2 Vektorinterfaces

Der Zugriff auf die verwalteten Daten erfolgt anhand sogenannter Vektorinterfaces. Ein Vektorinterface definiert eine Menge von Operationen, die ein bestimmtes Zugriffsszenario und eventuell einem bestimmten Datentyp unterstützen. Die momentan in **ADF** existierenden Vektorinterfaces sind in Abbildung 4.7 dargestellt. Beispielsweise stellt das Vektorinterface `FullDouble` Methoden zur Verfügung, die eine Speicherung in einem Array von `double`s nahe liegen. Das Vektorinterface `SparseDouble` unterstützt anhand der angebotenen Methoden eine spärliche Speicherung der Elemente des Vektors als `double`.

Die abstrakte Klasse `Vector` implementiert alle diese Schnittstellen, indem eine `UnsupportedOperationException` geworfen wird, siehe Abbildung 4.7.

4.4.2.3 Vektorimplementierungen

Eine konkrete Vektorimplementierung speichert die Daten in einem bestimmen Datentyp und in einer bestimmten Datenstruktur. Demzufolge unterstützt sie eine bestimmte Menge von Zugriffsszenarien, und damit eine Menge von Vektorinterfaces. Die Methoden dieser Interfaces werden von der Implementierung mit der entsprechenden Funktionalität überschrieben. Eine konkrete Vektorimplementierung wird durch ihren Klassennamen eindeutig identifiziert. Dieser wird von der Methode `getRepresentation()` zurückgeliefert.

In Abbildung 4.8 sind die existierenden konkreten Subklassen dargestellt. Zum Beispiel unterstützt die Klasse `FullDoubleVector` die Vektorschnittstellen `GetDouble`, `SetDouble`, `Weight`, `Normalize`, `RemoveFeatures` und `FullDouble`. Die Klasse `SparseSortedDoubleVector` unterstützt die Vektorschnittstellen `GetDouble`, `Weight`, `Normalize`, `RemoveFeatures` und `SparseDouble`.

Der Vorteil des vorgestellten Mechanismus besteht darin, dass die entsprechenden Vektormodule alle Zugriffsoperationen ohne vorherige Casts durchführen können. Als Nachteil entsteht zunächst Typunsicherheit. Dies wird durch den in Kapitel 4.8 erläuterten Mechanismus vermieden.

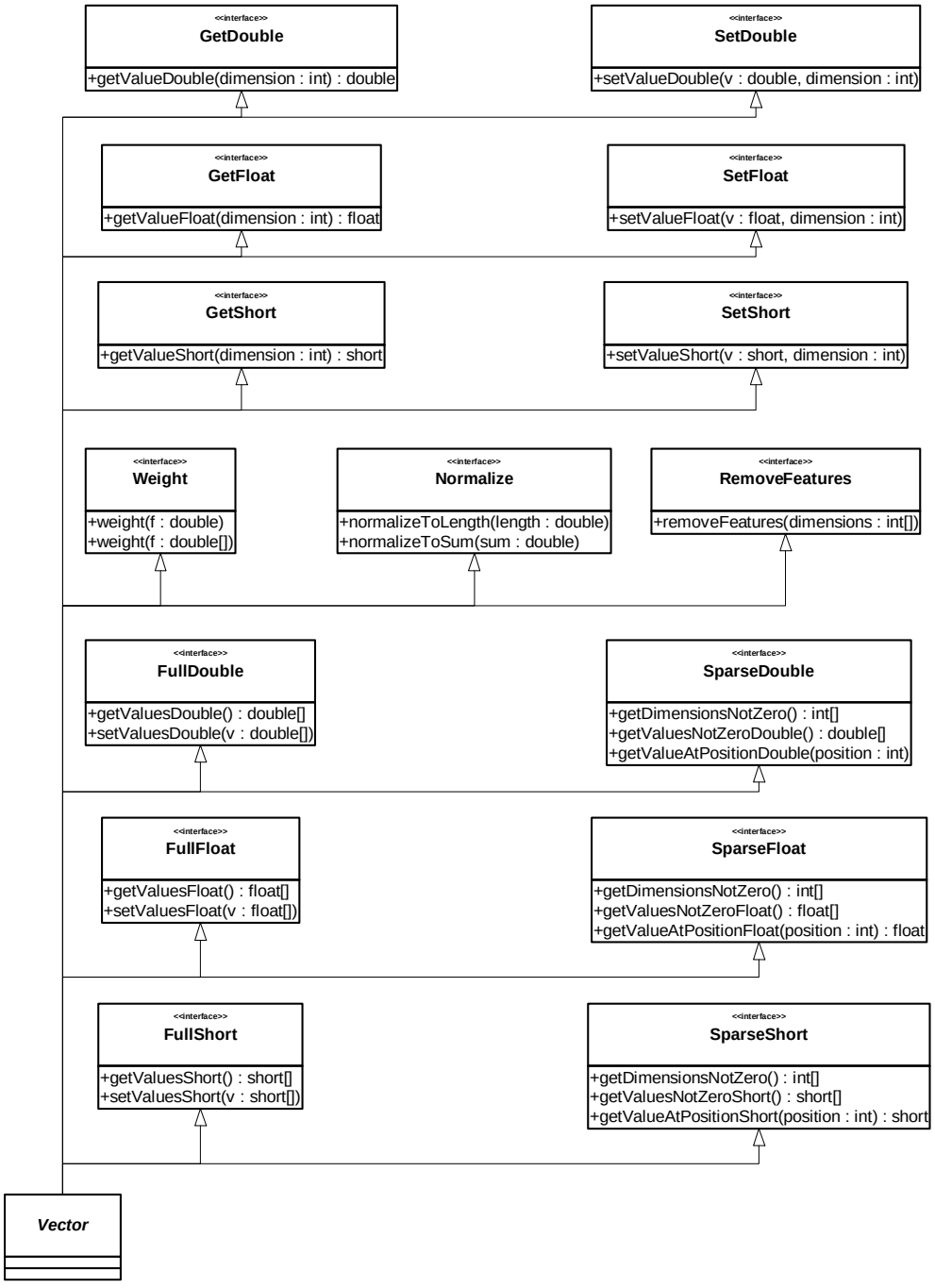

Abbildung 4.7: Vektorinterfaces

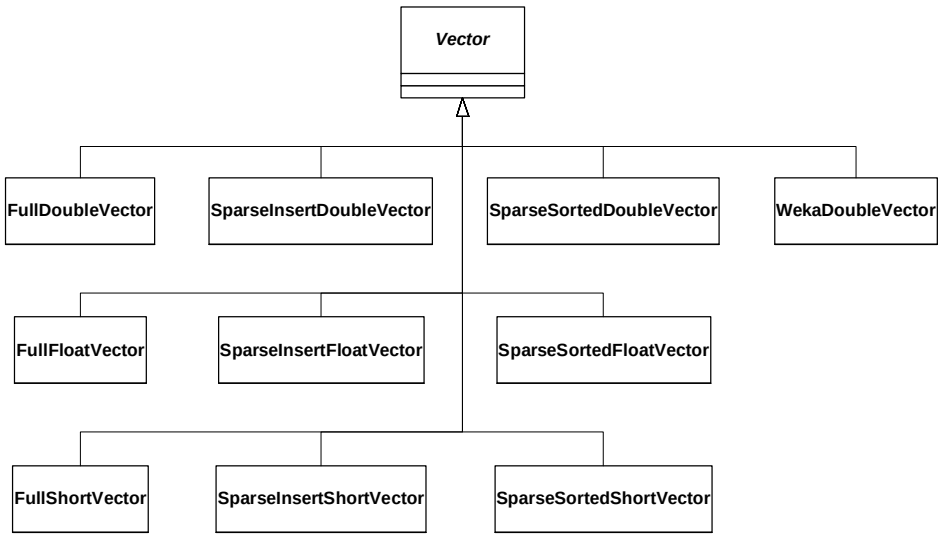

Abbildung 4.8: Existierende Vektorimplementierungen

4.4.2.4 Wechsel der Repräsentierung

Die im Verlauf einer Erschließung an den Vektoren durchzuführenden Operationen unterscheiden sich bezüglich ihrer Zugriffscharakteristiken erheblich. Beispielsweise werden bei der Erzeugung der Vektoren nur Werte eingefügt, ein Zugriff auf die Werte ist nicht erforderlich. Dieser Zugriff wird im weiteren Erschließungsverlauf wichtig, ein weiteres Einfügen unterbleibt. Demzufolge wird die Unterstützung vieler unterschiedlicher Vektorschnittstellen im Laufe einer Erschließung gefordert.

Dies lässt sich durch eine breite Unterstützung von Vektorschnittstellen von Implementierungen oder durch Umwandlung zwischen spezialisierten Implementierungen zwischen den verschiedenen Schritten realisieren. Die zweite Alternative ist der ersten im allgemeinen vorzuziehen, da eine einmalige Umwandlung effizient gestaltet werden kann, während eine Unterstützung von Vektorinterfaces um jeden Preis den ineffizienten Zugriff über weite Strecken einer Erschließung beinhaltet. Die Umwandlung von Vektorimplementierungen ist ein wesentlicher Bestandteil einer Erschließung in **ADF** und wird im Kapitel 4.8 weiterführend behandelt.

4.4.2.5 Trainingsmenge

Im Verlauf eines Trainings werden aus einer Menge von Informationsobjekten eine Menge von Vektoren gewonnen. Diese werden wie in Abbildung 4.5 in einer Trainingsmenge verwaltet. Ein `TrainingSet` ist ebenfalls eine abstrakte Klasse mit den in Abbildung 4.9 dargestellten Eigenschaften. Die eigentliche Funktionalität wird von konkreten Subklassen wie in Abbildung 4.10 dargestellt zur Verfügung gestellt. Diese Subklassen verfügen lassen sich ebenfalls anhand ihres Klassennamen eindeutig identifizieren. Ein Wechsel der

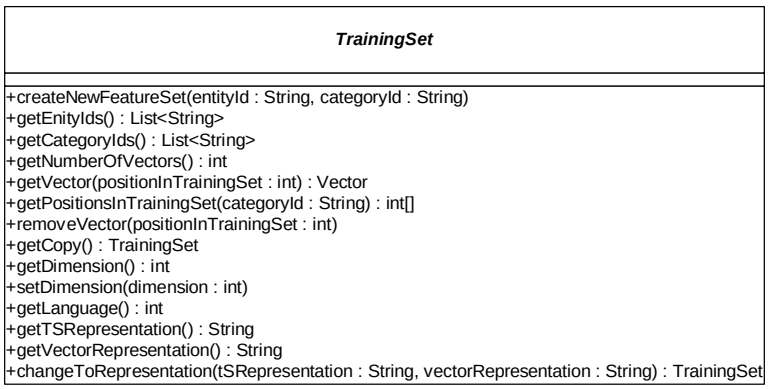

Abbildung 4.9: TrainingSet

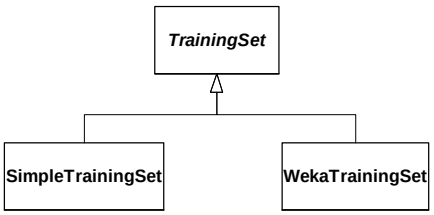

Abbildung 4.10: Existierende Trainingsetimplementierungen

Repräsentation der Trainingsmenge ist nicht möglich.

4.5 Mediatorinterfaces

Ein weiteres wichtiges Designziel von **ADF** ist die flexible Anpassbarkeit an verschiedenartigste Anwendungen. In diesem Kapitel wird das Konzept der frameworkgesteuerten Kommunikation zwischen **ADF** und einer Anwendung vorgestellt. Es existieren zwei verschiedene Schnittstellen zwischen einer Anwendung und dem Framework **ADF**, welche sich darin unterscheiden, von wem die Kommunikation ausgeht.

Einerseits wird **ADF** von der Anwendung gesteuert, die Anwendung veranlasst die einzelnen Erschließungsvorgänge (Trainingsphase, Produktivphase). In diesem Fall ist also die Anwendung der aktive Part. Die Schnittstelle ist in diesem Fall ein Treiberobjekt, welches von der Anwendung gerufen wird (siehe Kapitel 4.7). Diese Art der Kommunikation tritt nicht im Laufe eines Erschließungsvorgangs auf.

Andererseits benötigt das Framework im Laufe eines Erschließungsvorgangs Informationen von der Anwendung bzw. übermittelt Resultate (siehe Abbildung 4.2). Diese Kommunikationsschritte werden vom Framework initiiert, gemäß dem Motto: *"The Framework calls you, don't call the framework."*

In diesem Abschnitt wird der zweite Kommunikationsaspekt, der während des Er-

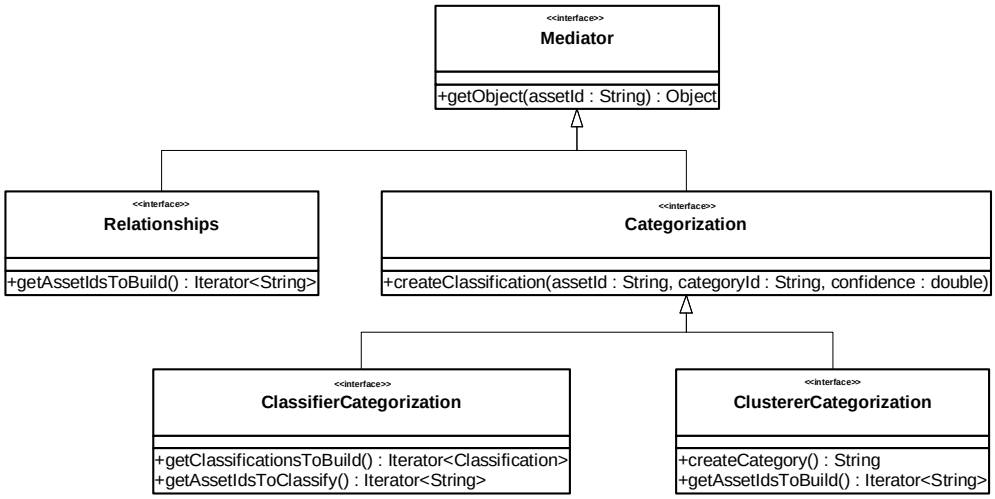

Abbildung 4.11: Mediatorinterfaces

schließungsvorgangs auftritt, behandelt. Die Idee besteht darin, Interfaces zu definieren, die von potentiellen Anwendungen implementiert werden. Die Anpassung des Frameworks an Anwendungen erfolgt in diesem Punkt also durch *Reuse by Inheritance*.

Die zu implementierenden Interfaces heißen Mediatorinterfaces. Wie in Kapitel 3.5 aufgeführt, unterscheiden sich die drei verschiedenen Erschließungsformen in der Art der erzielten Resultate. Dementsprechend unterscheiden sich die Mediatorinterface von Erschließungsform zu Erschließungsform. Es gibt drei verschiedene Interfaces, die alle von `Mediator` erben (siehe UML-Klassendiagramm 4.11).

Von allen Erschließungsformen wird die Methode `getObject(assetId : String)` benötigt, mit deren Hilfe das Informationsobjekt zu einer eindeutigen ID von der Anwendung angefordert wird.

Ebenfalls benötigen alle Erschließungsformen Zugriff auf die zum Training zu verwendeten Informationsobjekte. Die Details dieses Zugriffs unterscheiden sich jedoch. Im Falle einer Ähnlichkeitssuche und eines Clustering wird dies durch die Methode `getAssetIdsToBuild()` realisiert. Es werden lediglich alle im System vorhandenen Informationsobjekte benötigt. Eine Klassifikationsaufgabe benötigt zusätzlich Informationen über die Kategoriezugehörigkeit der Informationsobjekte, dies wird mittels `getClassificationsToBuild()`, welche `Classification`-Objekte zurückliefert, bewerkstelligt. Die Eigenschaften der Klasse `Classification` sind in Abbildung 4.12 dargestellt.

Im Falle einer Klassifikationsaufgabe ist ein batch-Betrieb möglich, es werden alle verfügbaren Informationsobjekte klassifiziert. Diese werden als Resultat der Methode `getAssetIdsToClassify()` geliefert. Die gewonnenen Resultate werden durch die Methode `createClassification(assetId : String, categoryId : String, confidence : double)` in die Anwendung eingetragen. Als Besonderheit einer Clusteraufgabe müssen zu Beginn der Produktivphase eine Menge synthetisch erzeugter Kategorien in der An-

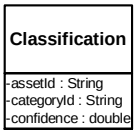

Abbildung 4.12: Classification

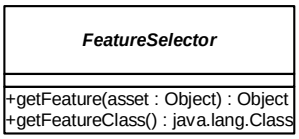

Abbildung 4.13: FeatureSelector

wendung angelegt werden. Dies wird durch die Methode `createCategory()` des Interfaces `ClustererCategorization` erreicht.

Zusammenfassend existiert zu jeder Erschließungsaufgabe eine Schnittstelle, die für eine Anwendung implementiert werden muss. Diese Schnittstellen liefern dem Framework die Informationen über die zu erschließenden Informationsobjekte und informieren die Anwendung über die gefundenen Resultate.

4.6 Anpassung an Typen von Informationsobjekten

Ein weiterer wesentlicher Punkt, in dem sich die Anwendungen unterscheiden, ist der Typ der jeweils zu erschließenden Informationsobjekte. Gemäß Modell (siehe 3.3.1) ermöglicht ein Informationsobjekt den Zugriff auf eine geordnete Menge von Features f_i, die die für eine Erschließung relevanten Informationen beinhalten. Diesen Zugriff realisieren sogenannte Featureselektoren.

In **ADF** wird ein Informationsobjekt durch ein Objekt der Klasse `Object` repräsentiert. Die Umsetzung der Featureselektoren in **ADF** wird durch Subklassen der abstrakten Klasse `FeatureSelector` (siehe Abbildung 4.13) bewerkstelligt. Für jeden neuen Typ von Informationsobjekten muss pro relevantem Feature eine konkrete Subklassen von `FeatureSelector` implementiert werden. Die wichtigste Methode hierbei ist `getFeature(asset : Object)`, welche das Feature aus dem übergebenen Informationsobjekte extrahiert.

Da eine weitere Forderung die Unterstützung verschiedener Sprachen umfasste, besteht die Möglichkeit, einen Sprachdetektor pro Informationsobjekttyp zu definieren. Dieser ist eine Subklasse der abstrakten Klasse `LanguageDetector`, siehe Abbildung 4.14.

Ein Informationsobjekttyp wird in **ADF** durch ein Objekt der Klasse `AssetType` repräsentiert (siehe Abbildung 4.15. Ein `AssetType`-Objekt kann entscheiden, ob ein gegebenes Informationsobjekt zu ihm gehört oder nicht, und es ermöglicht den Zugriff

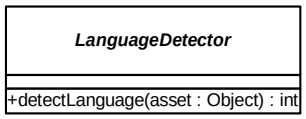

Abbildung 4.14: LanguageDetector

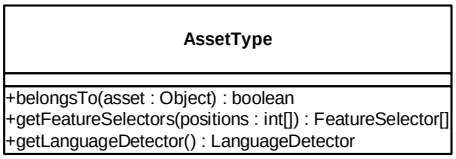

Abbildung 4.15: AssetType

auf die zugeordneten Featureselektoren und evtl. den Sprachdetektor.

4.7 Kontrollfluss

In diesem Abschnitt sollen die grundlegenden Abläufe bzw. Prozesse einer Erschließung in **ADF** diskutiert werden. Es wird das in Kapitel 3 erarbeitete Modell einer inhaltlichen Erschließung, unter Berücksichtigung der in Abschnitt 4.2 vorgestellten Anforderungen nach Multilingualität und der Unterstützung mehrerer Informationsobjekttypen, umgesetzt.

4.7.1 Treiber

Eine zentrale Rolle im Design von **ADF** spielt der sogenannte Treiber. Pro Erschließungsaufgabe existiert genau eine Treiberinstanz. Diese Instanz wird von der Anwendung gehalten und nimmt die Aufträge der Anwendung entgegen.

Ein Treiber wird in **ADF** durch ein Objekt der Klasse `Driver` repräsentiert. In Abbildung 4.3 sind die Beziehungen eines Treibers zu den anderen Klassen zu erkennen. Ein Treiber kann gemäß der Forderung nach Unterstützung mehrerer isomorpher Informationsobjekttypen mehrere `AssetType`-Objekte verwalten. Zu einem konkreten gegebenen Informationsobjekt wird der dazugehörige Informationsobjekttyp durch Abfragen aller registrierten Typen ermittelt.

Eine ebenfalls wichtige Rolle spielt der sogenannte Vektortyp, der durch ein Objekt der Klasse `VectorType` repräsentiert wird. Ein Vektortyp verwaltet alle Vektormodule einer Sprache. Dies sind evtl. mehrere Vektorisierer, evtl. mehrere Präprozessoren und eine Vektormaschine. Zusätzlich verwaltet ein Vektortyp die während eines Trainings erzeugten Trainingsmengen.

Es existiert also ein Vektortyp pro Sprache. Da in **ADF** Informationsobjekte mehrerer Sprachen erschlossen werden können, kann ein Treiber mehrere Vektortypen besitzen.

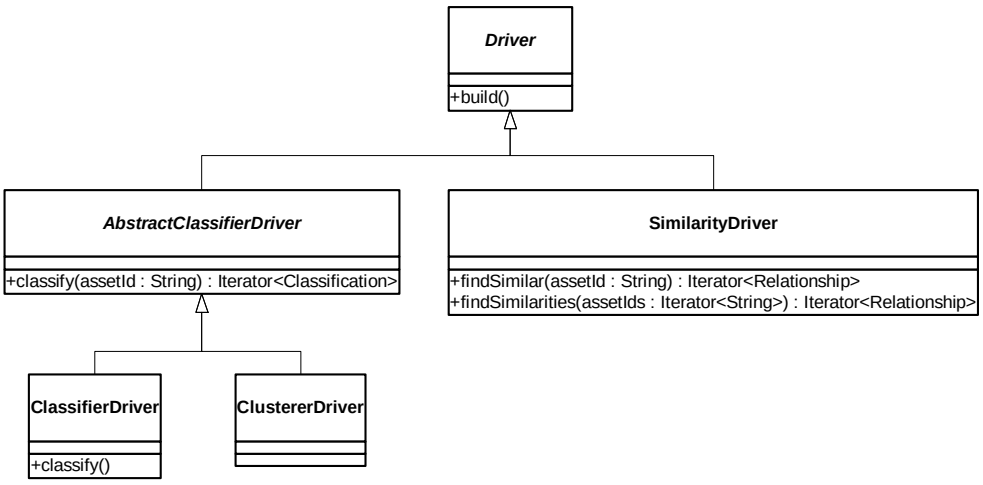

Abbildung 4.16: Vererbungshierarchie Driver

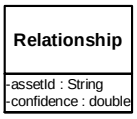

Abbildung 4.17: Relationship

Die wichtigste Aufgabe des Treibers besteht in der Steuerung des Kontrollflusses im Laufe einer Erschließung. Aufgrund der Unterschiede zwischen den drei Erschließungsarten gibt es drei verschiedene Treiber (siehe Abbildung 4.16). Allen Treiber gemeinsam ist die Methode `build()`, welche die jeweilige Trainingsphase startet. Im Falle einer Klassifikations- oder Clusteraufgabe können einzelne Informationsobjekte erschlossen werden, die Resultate werden in Gestalt von `Classification`-Objekten zurückgeliefert. Bei Klassifikationsaufgaben ist ein zusätzlicher batch-Betrieb möglich (siehe auch Kapitel 4.5).

Im Falle einer Assoziativen Suche werden die Resultate einer Erschließung in Form von `Relationship`-Objekten zurückgeliefert (siehe Abbildung 4.17).

4.7.2 Trainingsphase

Zur Unterstützung der Trainingsphase bietet die abstrakte Klasse `Driver` mittels der implementierten Methode `build()` den im Sequenzdiagramm in Abbildung 4.18 dargestellten Ablauf. Es sind die drei Schritte einer Erschließung deutlich erkennbar, wobei der Teilschritt der Vektorerzeugung nochmals in drei Teilschritte unterteilt ist. Dies ergibt sich als Resultat der softwaretechnischen Umsetzung in **ADF** und wird im folgenden näher erläutert.

Eine zentrale Rolle spielt dabei der Vektortyp. Er verwaltet die Vektormodule sowie die erstellten Trainingsmengen jeder Phase des Trainings.

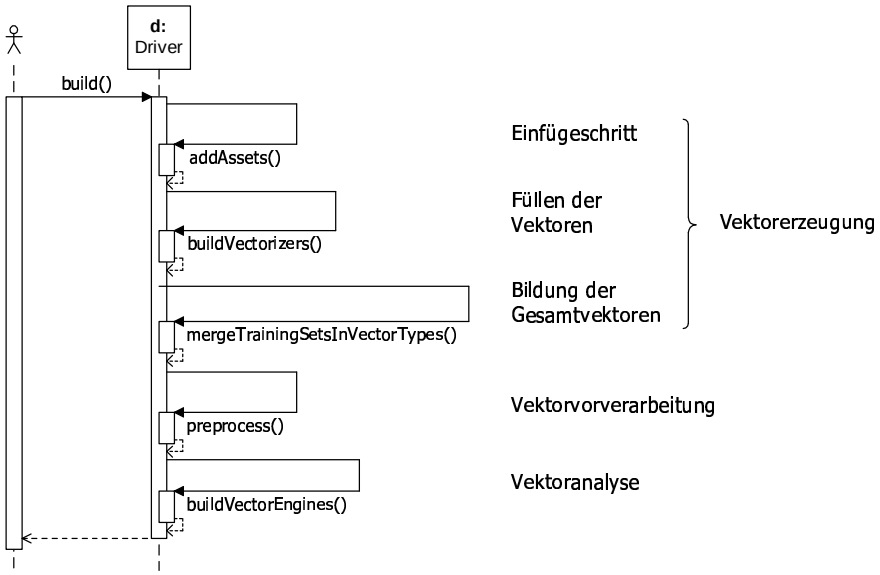

Abbildung 4.18: Ablauf eines Trainings

Die Vektorerzeugung beginnt mit dem Einfügeschritt, dessen Aufgabe darin besteht, die zum Training vorhandenen Informationsobjekte ihren Sprachen entsprechend zu sortieren und eine Trainingsmenge pro Sprache zu erzeugen, welches einen Vektor pro Informationsobjekt enthält. Diese Vektoren enthalten noch keine Werte, sie speichern lediglich die ID des repräsentierten Informationsobjektes und die Sprache. Besitzen die vorhandenen Informationsobjekttypen Sprachdetektoren, so wird die Sprache mit ihnen ermittelt. Existieren keine Sprachdetektoren, so wird eine voreingestellte Sprache gewählt.

Da pro Sprache ein Vektortyp existiert, werden im Verlauf des Einfügeschrittes neue Vektortyp-Objekte erzeugt. Dies geschieht unter Benutzung des Prototyp Musters. Ganz zu Beginn einer Erschließung existiert ein Treiber mit einem speziellen zugeordneten Vektortyp, dem Prototyp. Dieser enthält Prototypen aller Vektormodule und wird nicht zum produktiven Betrieb benutzt, sondern dient lediglich zur Erzeugung von produktiven Vektortypen. Existieren also Informationsobjekte in zwei verschiedenen Sprachen, so werden im Verlauf des Einfügeschrittes zwei produktive Vektortypen erzeugt und pro Vektortyp eine Trainingsmenge mit Vektoren ohne Werte. Dies ist Abbildung 4.19 dargestellt. Im Beispiel existieren Informationsobjekte in zwei verschiedenen Sprachen, nämlich in Deutsch und in Englisch. Als Resultat des Einfügeschrittes existieren zwei Vektortyp-Objekte mit jeweils einer Trainingsmenge. Die Trainingsmengen `t1` und `t2` enthalten die zu erschließenden Informationsobjekte, in `t1` sind die deutschsprachigen Informationsobjekte und in `t2` die englischsprachigen.

Der genaue Ablauf des Einfügeschrittes ist in Abbildung 4.20 in einem Sequenzdiagramm dargestellt. Es wird das Einfügen eines deutschsprachigen Informationsobjektes gezeigt.

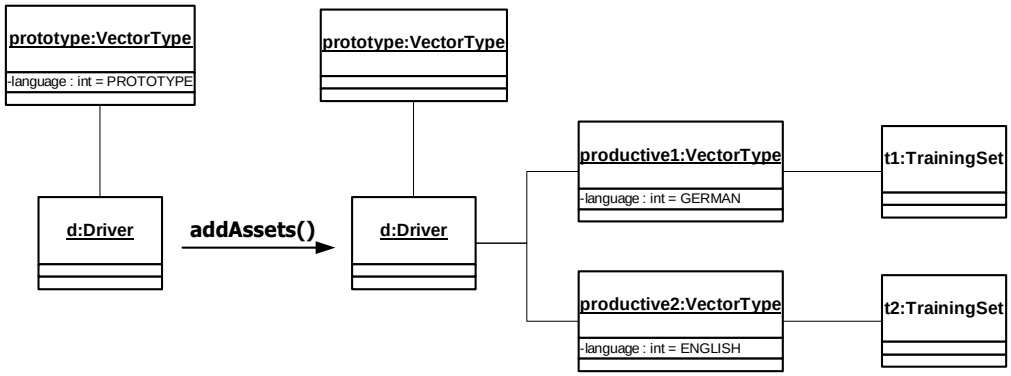

Abbildung 4.19: Einfügeschritt

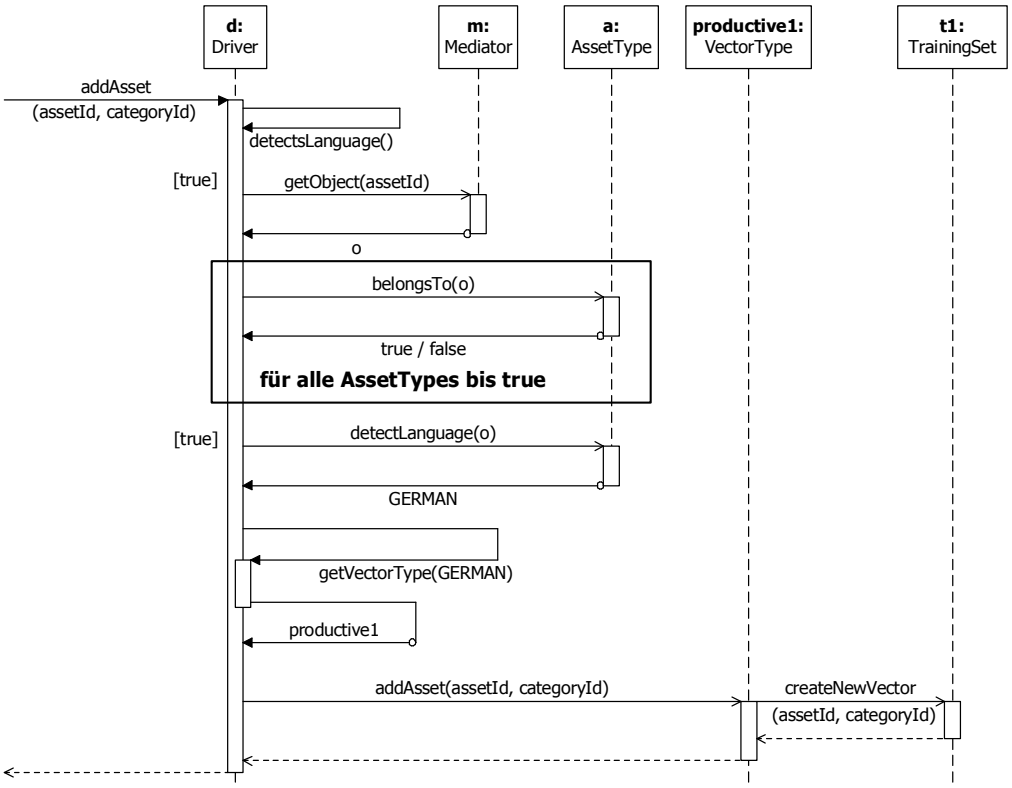

Abbildung 4.20: Sequenzdiagramm des Einfügeschrittes

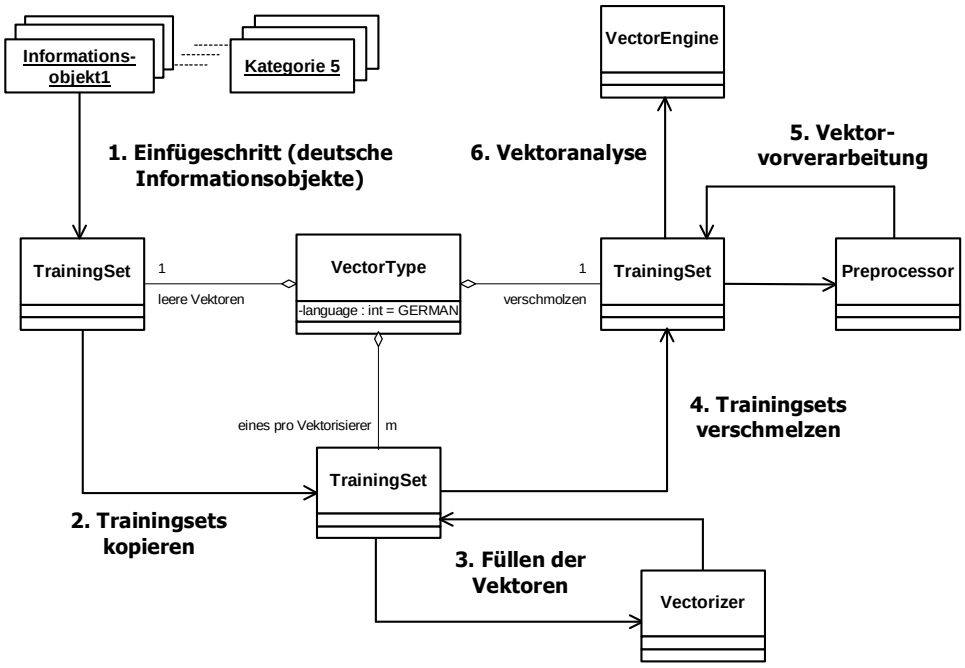

Abbildung 4.21: Trainingsphase

Als nächstes werden die erzeugten Vektoren mit Werten gefüllt. Dies muss für jeden produktiven Vektortyp des Treibers durchgeführt werden. Da ein Vektortyp mehrere Vektorisierer enthalten kann, wird die erzeugte leere Trainingsmenge zunächst für jeden Vektorisierer kopiert. Auf den kopierten Trainingsmengen arbeiten nun die Vektorisierer.

Anschließend findet die Bildung der Gesamtvektoren statt, die von den Vektorisierern gefüllten Trainingsmengen werden wie in Kapitel 3.3.3 besprochen verschmolzen. Der Erschließungsteilschritt Vektorerzeugung ist hiermit vollständig abgeschlossen. Die nun folgenden Teilschritte Vektorvorverarbeitung und Vektoranalyse werden ebenfalls auf allen Vektortypen nacheinander durchgeführt.

Der gesamte Ablauf ist in Abbildung 4.21 für einen produktiven Vektortyp, der deutschsprachige Informationsobjekte verwaltet, mit den entstehenden Trainingsmengen graphisch dargestellt.

Die Teilschritte Füllen der Vektoren, Bildung der Gesamtvektoren und Vektorvorverarbeitung sind generisch, die entsprechenden Methoden sind in `Driver` implementiert. Die genaue Ausgestaltung der Teilschritte Einfügeschritt und Vektoranalyse obliegt den Subklassen von `Driver` entsprechend den unterschiedlichen Erschließungsformen. Gemäß den in Kapitel 4.5 geäußerten Gedanken unterscheiden sich die Zugriffsformen auf die unterschiedlichen Mediatorimplementierungen, und die Vektoranalyse unterscheidet sich ebenfalls. Es sind die in Abbildung 4.22 dargestellten Kombinationen eines Treibers mit einem Mediatorinterface möglich.

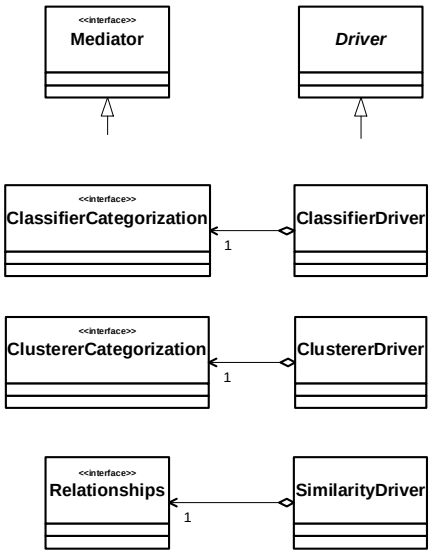

Abbildung 4.22: Mögliche Treiber-Meditator Kombinationen

4.7.2.1 Klassifikationsaufgabe

Im Falle einer Klassifikationsaufgabe müssen zusätzlich zu den *assetIDs* die Kategorieinformationen aufgenommen werden. In der Vektoranalyse werden pro Vektortyp die Vektorklassifikatoren mit den Trainingsmengen trainiert.

4.7.2.2 Clustering

Im Einfügeschritt werden lediglich die *assetIDs* in die leeren Vektoren aufgenommen. Im Teilschritt Vektoranalyse werden zunächst die Vektorclusterer trainiert, und anschließend werden die gefundenen Kategorien mit der Methode `createCategory()` der Schnittstelle `ClustererCategorization` erzeugt.

4.7.2.3 Ähnlichkeitssuche

Soll eine Ähnlichkeitssuche durchgeführt werden, so werden im Einfügeschritt lediglich die *assetIDs* in die leeren Vektoren aufgenommen und zur Vektoranalyse werden die Vektorähnlichkeitssuchmaschinen trainiert.

4.7.3 Produktivphase

Die Aufgabenstellung der Produktivphase besagt, dass ein gegebenes Informationsobjekt erschlossen werden soll. Es ist auch der Fall denkbar, dass eine Erschließung zu einer Menge von Informationsobjekten durchgeführt werden soll, beispielsweise sollen ähnliche Informationsobjekte zu einer Menge von Informationsobjekten gefunden werden.

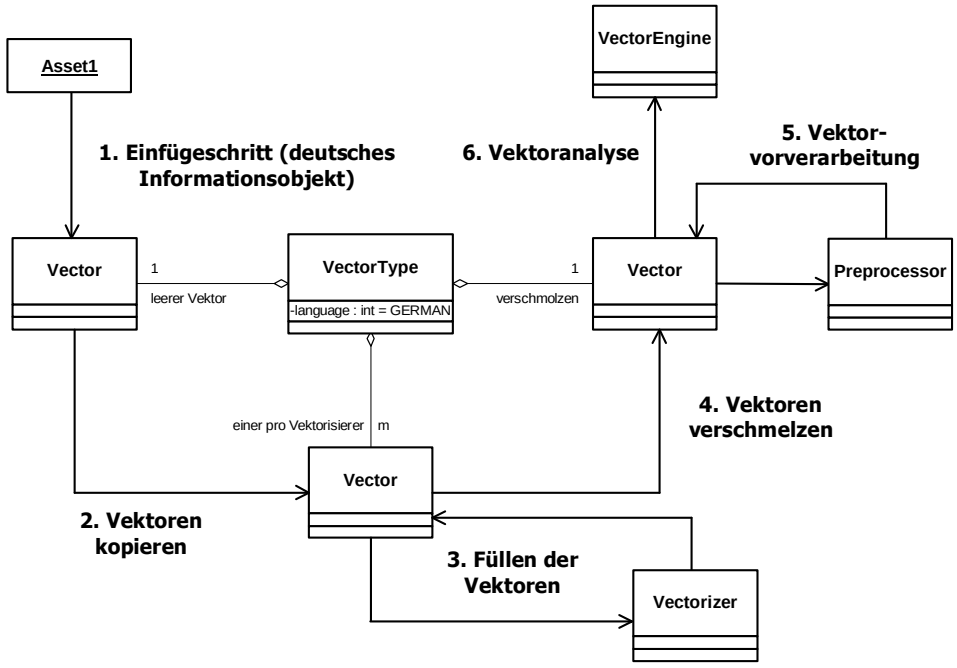

Abbildung 4.23: Produktivphase

Die Abläufe in der Produktivphase sind denen der Trainingsphase sehr ähnlich. Es lassen sich wiederum die fünf Schritte wie in Abbildung 4.18 dargestellt unterscheiden. Die im Laufe der Erschließung gebildeten Vektoren werden wie in der Trainingsphase vom entsprechende Vektortyp verwaltet (siehe Abbildung 4.23).

4.8 Vektormodule

Alle vektorverarbeitenden bzw. -erzeugenden Objekte werden Vektormodule genannt. Entsprechend den drei Schritten einer Erschließungsaufgabe gibt es drei Arten von Vektormodulen. Im Vektorerzeugungsschritt werden von Vektorisierern die Vektoren erzeugt (genauer: mit Werten gefüllt), anschließend von Präprozessoren vorverarbeitet und zuletzt von einer Vektormaschine analysiert. Alle Vektormodule erben von einer gemeinsamen abstrakten Superklasse `VectorModule` (siehe Abbildung 4.24). Allen Vektormodulen gemeinsam ist, dass sie Operationen auf Vektoren durchführen.

Den Gedanken in Kapitel 4.4 folgend, benötigt ein bestimmtes Vektormodul von einem zu bearbeitenden Vektor die Funktionalitäten einer oder mehrerer Vektorinterfaces. Dieser Zusammenhang ist im Metamodell in Abbildung 4.25 dargestellt. Alle in **ADF** existierenden Vektorimplementierungen werden von der Klasse `VectorRepresentations` verwaltet. Eine einzelne Vektorimplementierung implementiert mehrere Vektorinterfaces. Ein Vektormodul benötigt seinerseits von einem Vektor die Funktionalität mehrer Vektorinterfaces.

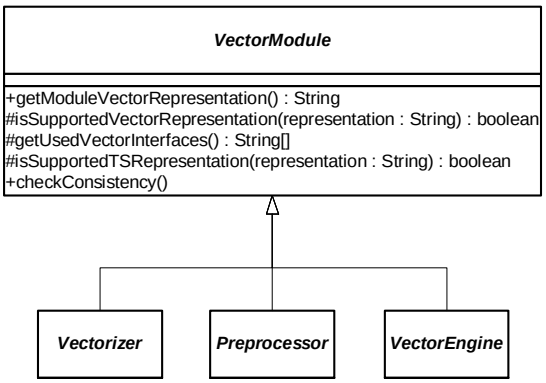

Abbildung 4.24: VectorModule mit Subklassen

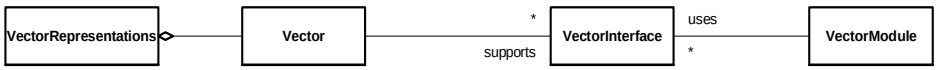

Abbildung 4.25: Metamodell der Vektorrepräsentierungen

Die Abhängigkeit eines Vektormoduls von bestimmten Vektorinterfaces ist zum Zeitpunkt der Implementierung des Vektormoduls bekannt, da das Modul den zu bearbeitenden Vektor mittels der in den Vektorinterfaces definierten Methoden anspricht.

4.8.0.1 Konsistenzüberprüfung

Ein Erschließungsprozess wird in **ADF** durch eine geordnete Abfolge von Modulen modelliert. Ebenfalls werden zu Beginn einer Erschließungsaufgabe die in den Vektormodulen verwendeten Vektorimplementierungen konfiguriert. Die konfigurierten Implementierungen lassen sich durch die Methode `getModuleVectorRepresentation() : String` eines Vektormoduls ermitteln. Die Art und Anzahl der verwendeten Vektormodule und die verwendeten Vektorimplementierungen unterscheiden sich von Erschließungsprozess zu Erschließungsprozess und charakterisieren eine konkrete Erschließungsaufgabe.

Die Idee des implementierten Mechanismus besteht darin, die Konsistenz einer konfigurierten Erschließungsaufgabe vor Beginn der eigentlichen Erschließung zu überprüfen. Eine jede Modulimplementierungen implementiert die Methode `getUsedVectorInterfaces() : String[]` und es wird dynamisch mittels Java-Reflection überprüft, ob die konfigurierte Vektorimplementierung die geforderten Vektorinterfaces tatsächlich implementiert.

Als zweites wird geprüft, ob die laut Konfiguration vorgesehenen Repräsentationswechsel unterstützt werden.

Der vorgestellte Mechanismus ermöglicht die flexible Verwendung verschiedenster Vektorrepräsentierungen im Laufe einer Erschließungsaufgabe. Es ist möglich Vektormodule zu entwickeln, ohne Kenntnis von eventuell später entwickelten Vektorimplementierungen

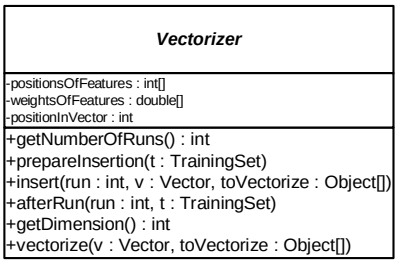

Abbildung 4.26: Vectorizer

haben zu müssen. Die einzige Schnittstelle zwischen Modulentwickler und den Entwicklern von Vektorimplementierungen sind somit die relativ stabilen Vektorinterfaces.

4.8.1 Vektorisierer

Die Aufgabe des Vektorisierers besteht ganz allgemein darin, aus einem oder mehreren gewichteten Features einen Vektor zu bilden. Alle Vektorisierer erben von der abstrakten Superklasse `Vectorizer` (siehe Abbildung 4.26). Die Attribute eines Vektorisierers kennzeichnen seine Position im gesamten zu erzeugenden Vektor (`positionInVector`), sowie die der zu vektorisierenden Features (`positionsOfFeatures`) und die Gewichtungen, mit denen die Features eingehen (`weightsOfFeatures`).

Die genaueren Abläufe unterscheiden sich zwischen Trainings- und Produktivphase.

4.8.1.1 Trainingsphase

Als Voraussetzung einer Vektorisierung in der Trainingsphase wurde eine Trainingsmenge aufgebaut, welches aus einem Vektor pro Informationsobjekt besteht. Diese Vektoren enthalten allerdings keine Werte. Es sind lediglich die eindeutige *assetID* des repräsentierten Informationsobjektes, eventuell Kategorieinformationen sowie die Sprache der Informationsobjekte im Vektor gespeichert. Alle Vektoren der Trainingsmenge sind zwangsläufig in derselben Sprache. Die nun folgenden Abläufe sind in einem Sequenzdiagramm in Abbildung 4.27 dargestellt. Die Vorgänge werden vom Treiber gesteuert. Der Vektorisierer im Sequenzdiagramm führt einen Durchlauf (Run) über die Vektoren der Trainingsmenge durch. Dies lässt sich durch die Methode `getNumberOfRuns()` bestimmen und bedeutet, dass die Informationsobjekte nur einmal vom Vektorisierer benötigt werden. Denkbar sind ebenfalls Vektorisierer, die mehrere Durchläufe über die Informationsobjekte benötigen.

Zunächst wird die Methode `prepareInsertion(t : TrainingSet)` aufgerufen, anschließend beginnt der erste Durchlauf. In diesem wird pro Vektor der Trainingsmenge die Methode `insert(run : int, v : Vector, toVectorize : Object[])` aufgerufen. In `toVectorize[]` sind die Features enthalten. Anschließend wird die Methode `afterRun(run : int, t : TrainingSet)` aufgerufen. Damit ist ein Durchlauf beendet, sollte der Vek-

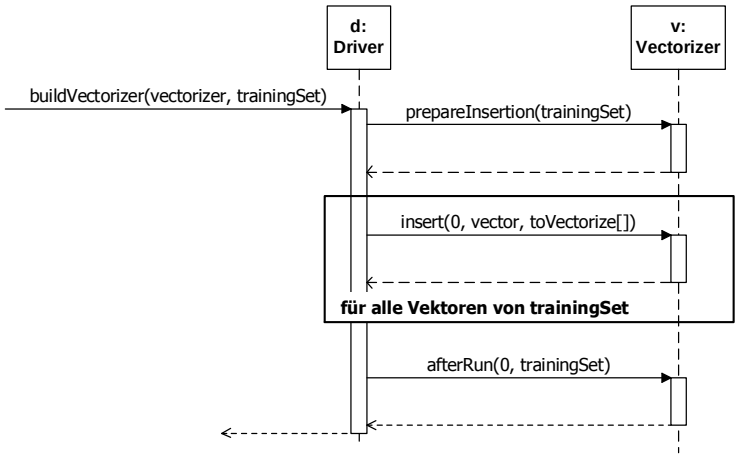

Abbildung 4.27: Sequenzdiagramm buildVectorizer()

torisierer mehrere Durchläufe benötigen, werden die letzten beiden Schritte wiederholt. Alle erwähnten Methoden sind abstrakte Methoden von `Vectorizer`, der Superklasse aller Vektorisierer, und müssen demzufolge von konkreten Vektorisierern implementiert werden.

4.8.1.2 Produktivphase

In der Produktivphase wird zunächst ebenfalls ein leerer Vektor zum zu erschließenden Informationsobjekt erzeugt. Dessen Dimension wird auf den Wert des Methodenaufrufs `getDimension() : int` gesetzt, und anschließend werden in der Methode `vectorize(v : Vector, toVectorize : Object[])` die Werte des Vektors vom Vektorisierer gesetzt.

4.8.1.3 Implementierte Vektorisierer

Alle implementierten Vektorisierer in **ADF** sind in Abbildung 4.28 dargestellt. Der `String-Vectorizer` vektorisiert `String`-Objekte, der `DoubleArrayVectorizer` Arrays von `double`s, der `DoubleVectorizer` `Double`-Objekte, und der `IntegerVectorizer` `Integer`-Objekte.

4.8.2 Präprozessoren

Alle Präprozessoren erben von der abstrakten Superklasse `Preprocessor` (siehe Abbildung 4.29). Diese definiert die abstrakten Methoden `preprocess(t : TrainingSet)`, welche in der Trainingsphase angewandt wird und die abstrakte Methode `preprocess(v : Vector)`, welche in der Produktivphase benutzt wird.

4.8.3 Vektormaschinen

Die abstrakte Superklasse aller Vektormaschinen heißt `VectorEngine`. Es werden keine Methoden in `VectorEngine` definiert, da sich die Signaturen von Erschließungsart zu Er-

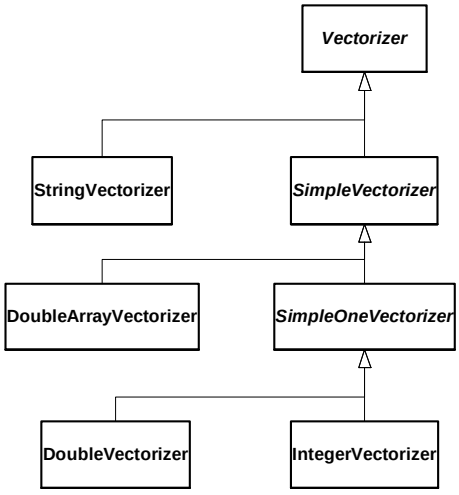

Abbildung 4.28: Konkrete Vektorisierer

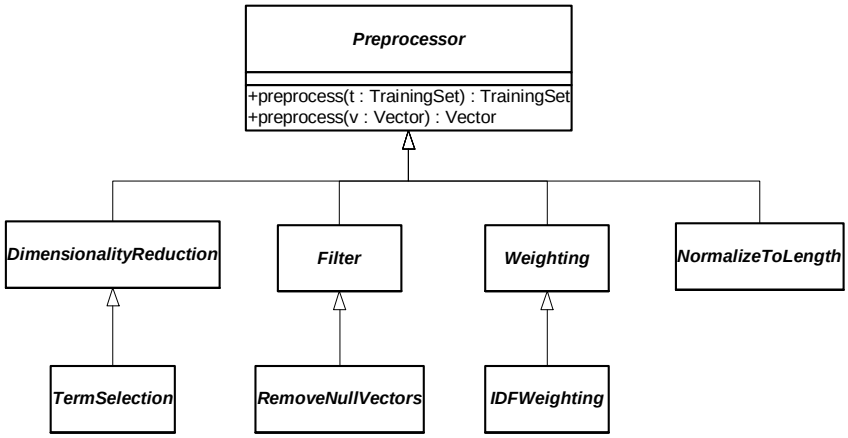

Abbildung 4.29: Vererbungshierarchie Präprozessoren

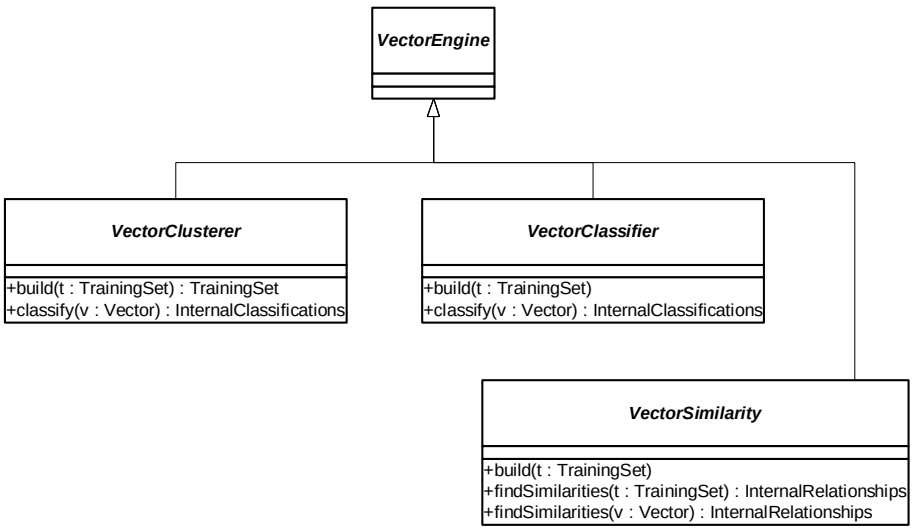

Abbildung 4.30: Subklassen von VektorEngine

schließungsart unterscheiden (siehe Abbildung 4.30). Entsprechend den drei unterstützten Erschließungsarten gibt es drei abstrakte Subklassen.

4.9 Services

Die Aufgabe des Frameworks **ADF** besteht darin, die Realisierung von Erschließungsaufgaben zu unterstützen. Im vorangegangenen Kapitel wurden die wesentlichen an einer Erschließung beteiligten Objekte sowie ihre Beziehungen untereinander, vorgestellt.

Zur praktischen Umsetzung der vorgestellten Kernfunktionalität von **ADF** sind eine Reihe von Mechanismen notwendig, die in diesem Kapitel vorgestellt werden. Diese Mechanismen werden im folgenden *Services* genannt. Es gibt vier verschiedene Services:

1. Eigenschaftsservice,
2. Logservice,
3. Kommunikationsservice und
4. Persistenzservice.

An den Namen der Services lässt sich der jeweils abgedeckte Aufgabenbereich ablesen. Die Services schaffen eine Umgebung für die Objekte, die die eigentliche Kernfunktionalität realisieren.

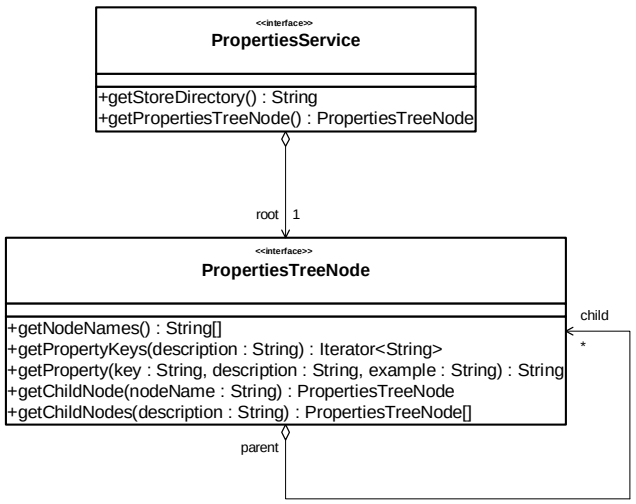

Abbildung 4.31: Eigenschaftsservice

4.9.1 Eigenschaftsservice

Die Aufgabe des Eigenschaftsservice besteht darin, die Objekte von **ADF** auf einheitliche Weise mit Konfigurationsparametern zu versorgen. Der Eigenschaftsservice spielt demzufolge nur im Moment der Erzeugung eines Objektes eine Rolle.

Eine einzelne Eigenschaft ist ein Schlüssel - Wert Paar, ähnlich den Eigenschaften, die die Klasse `java.util.Properties` zur Verfügung stellt. Im Unterschied zur Klasse `java.util.Properties` sind die Eigenschaften, die der Eigenschaftsservice von **ADF** zur Verfügung stellt, in einer Baumstruktur hierarchisch angeordnet.

4.9.1.1 Eigenschaftsbaum

Die Knoten des Baums werden von der Schnittstelle `PropertiesTreeNode` modelliert, welches Zugriff auf im Knoten angeordneten Eigenschaften und die Kinderknoten ermöglicht. Dies ist in Abbildung 4.31 in einem Klassendiagramm dargestellt. Ebenfalls hat ein Knoten des Eigenschaftsbaums einen Namen, und er kennt die Namen der Knoten, die zwischen der Wurzel und ihm liegen. Die Begründung dafür sowie die Anwendungsszenarien werden in Kapitel 4.9.4 vorgestellt. In den Abbildungen 4.36, 4.37 und 4.38 ist ein Beispielbaum dargestellt, der von einem Eigenschaftsservice geliefert werden kann. Auf die Abbildungen wird ebenfalls in Kapitel 4.9.4 näher eingegangen.

Die Methoden, die ein konkreter Eigenschaftsservice in **ADF** zur Verfügung stellt, sind von der Schnittstelle `PropertiesService` (siehe Abbildung 4.31) festgelegt. Im Wesentlichen liefert ein Eigenschaftsservice die Wurzel des Eigenschaftsbaumes.

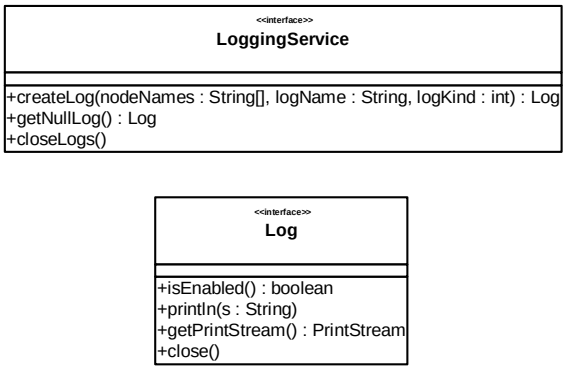

Abbildung 4.32: Logservice

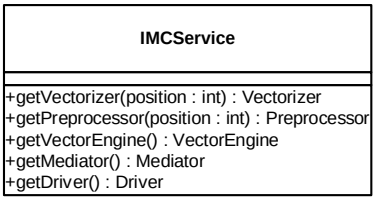

Abbildung 4.33: Kommunikationsservice

4.9.2 Logservice

Die Aufgaben des Logservice in **ADF** werden durch die Schnittstelle `LoggingService` (siehe Abbildung 4.32) spezifiziert. Die wichtigste Methode erzeugt Logobjekte, die die Schnittstelle `Log` implementieren.

4.9.3 Kommunikationsservice

Eine Erschließung in ADF wird in die drei Schritte Vektorerzeugung, Vektorvorverarbeitung und Vektoranalyse zerlegt, und von den entsprechenden Vektormodulen nacheinander realisiert. Die Unterteilung in drei Schritte bringt eine Reihe von Vorteilen mit sich. Der größte Vorteil ist sicherlich die Verringerung der Komplexität des Gesamtvorgangs durch Aufteilen in mehrere voneinander unabhängige Teilschritte.

Diese gewollte Unabhängigkeit kann jedoch in einigen Fällen einen Nachteil darstellen, wenn nämlich Teilschritte voneinander abhängen. Beispielsweise erzeugt ein Vektorisierer als Nebenprodukte Informationen, die für die anschließenden Merkmalsreduktion benötigt werden. Aus Performancegründen ist also ein Mechanismus wünschenswert, der in diesen Fällen einen Austausch zwischen den Vektormodulen ermöglicht. Dieser Mechanismus wird vom Kommunikationsservice realisiert.

Die Umsetzung erfolgt durch ein Objekt der Klasse `IMCService` (siehe Abbildung 4.33). IMC steht hierbei für **I**nter-**M**odule-**C**ommunication. In einer Instanz von **ADF** existiert

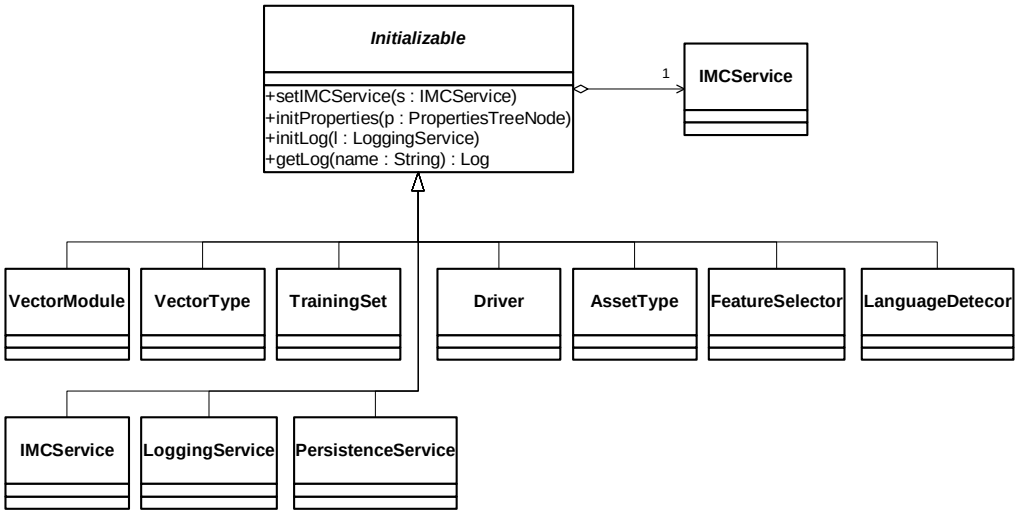

Abbildung 4.34: Initializable

genau eine Instanz des Kommunikationsservice, die in allen Vektormodulen gehalten wird.

4.9.4 Superklasse Initializable

Bisher wurden die Services Eigenschaftsservice, Logservice und Kommunikationsservice einzeln vorgestellt, die Details der Benutzung dieser Services wurden offen gelassen. Diese Details werden von der abstrakten Klasse `Initializable` (siehe Abbildung 4.34) realisiert. Wie in der Abbildung zu sehen ist, erben alle wesentlichen Klassen von **ADF** von `Initializable`.

Allgemein gesagt stellt `Initializable` allen Subklassen eine Infrastruktur zur Verfügung. Die gelieferten Dienste lassen sich den drei vorgestellten Services zuordnen. Dementsprechend erfolgt die Initialisierung eines konkreten Objektes vom Typ `Initializable` in drei Teilschritten, welche im Sequenzdiagramm in Abbildung 4.35 dargestellt sind. Zunächst wird die Instanz des Kommunikationsservice gesetzt, anschließend werden die Konfigurationsparameter eingelesen, und als letztes werden die Logobjekte initialisiert.

4.9.4.1 Initialisieren von ADF

Beim Lesen der Konfigurationsparameter wird dem zu initialisierenden Objekt ein Knoten des Eigenschaftsbaums übergeben. Dieser Knoten enthält die für dieses Objekt benötigten Eigenschaften. Die Grundidee ist nun, dass das Objekt zunächst die für sich interessanten Eigenschaften ausliest. Anschließend extrahiert das Objekt einen Kindsknoten des übergebenen Knoten und übergibt diesen Knoten einem weiteren zu initialisierenden Objekt. Auf diese Art und Weise wird der gesamte Objektgraph von **ADF** initialisiert.

Ein wesentlicher Vorteil dieses Verfahrens ist die gewonnene Flexibilität. Es lassen sich

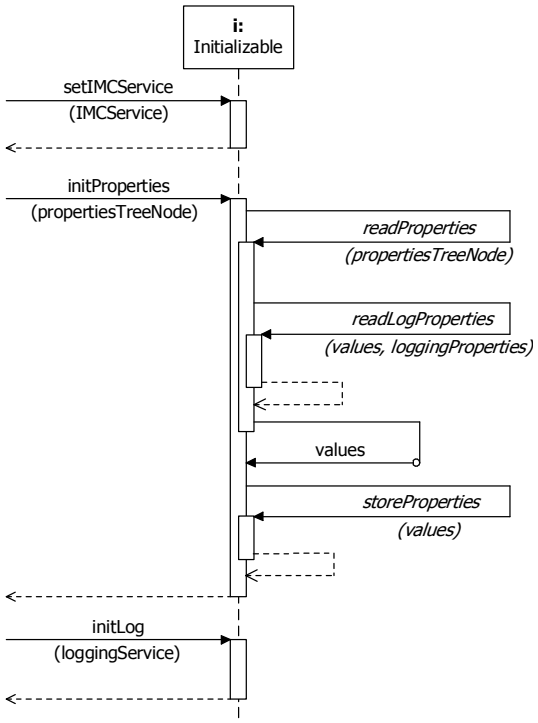

Abbildung 4.35: Initialisierung eines Objektes

die konkreten zu initialisierenden Klassen in den Eigenschaften festlegen und konfigurieren.

Weiterhin ist die gewonnene Einheitlichkeit der Initialisierung von Vorteil. Mit Ausnahme der Mediatorinterfaces und dem Eigenschaftsservice werden alle Klassen von **ADF** auf diese Art und Weise initialisiert und somit auch konfiguriert. Ein beispielhafter Eigenschaftsbaum ist in den Abbildungen 4.36, 4.37 und 4.38 dargestellt. Die Initialisierung beginnt mit der Wurzel des Eigenschaftsbaums beim Treiberobjekt. Das Treiberobjekt benötigt zur Konfiguration selbst keine Eigenschaften. Vom Treiberobjekt aus werden der Kommunikationsservice, der Logservice, der Persistenzservice, die Informationsobjekttypen und der Vektortyp initialisiert. Dementsprechend gibt es zu jedem Objekt einen Kindknoten. Im Beispiel wird als Logservice ein Objekt der Klasse

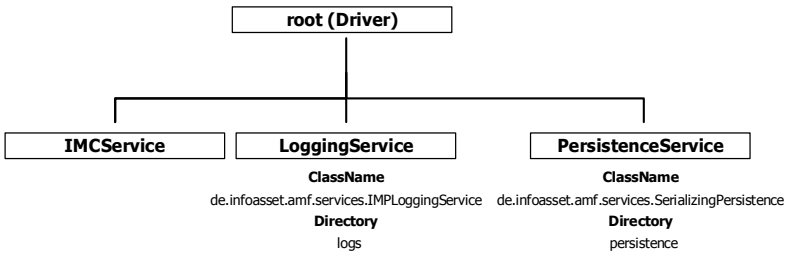

Abbildung 4.36: Konfiguration der Services

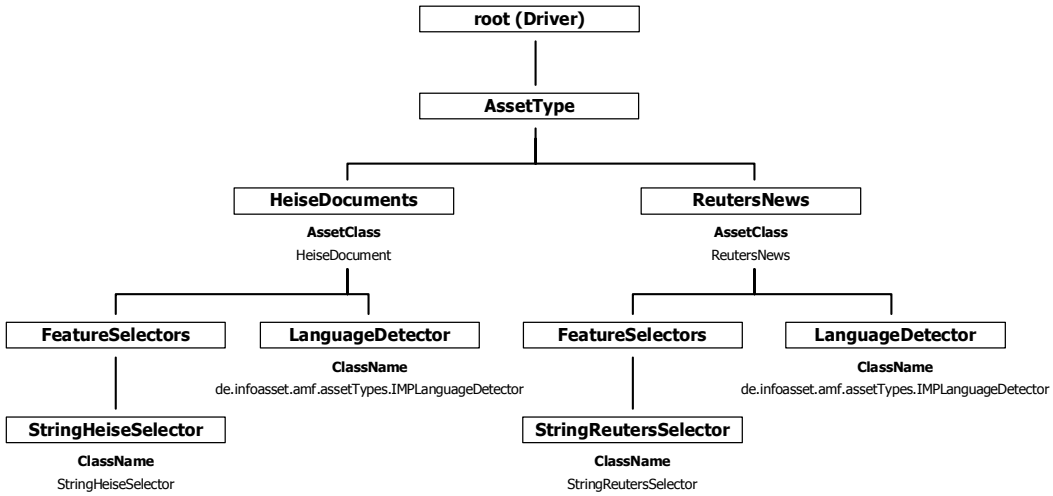

Abbildung 4.37: Konfiguration der Informationsobjekttypen

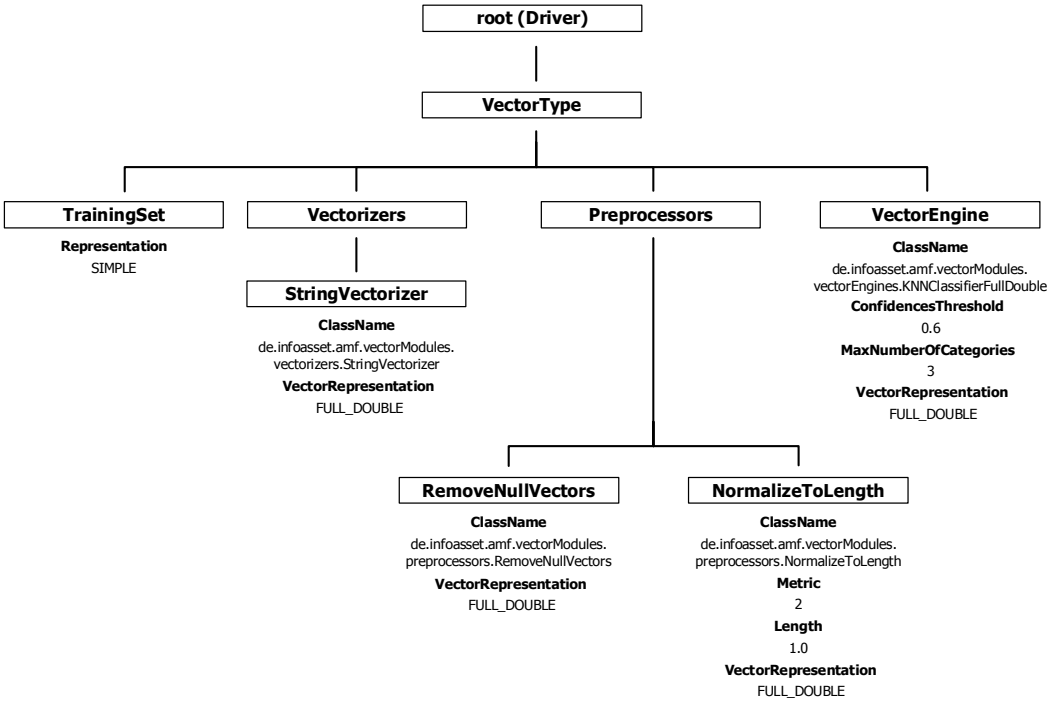

Abbildung 4.38: Konfiguration des Vektortypes

`de.infoasset.amf.services.IMPLoggingService` erzeugt, welches als Eigenschaft das Verzeichnis *logs* zugewiesen bekommt. Die initialisierte Instanz von **ADF** verarbeitet zwei verschiedene Informationsobjekttypen (Heise-Dokumente und Reuters-News) mit einem Vektorisierer, zwei Präprozessoren und einem k-Nearest-Neighbor Klassifikator.

Eine direkte Folge Initialisierungsmechanismus ist bei steigender Anzahl zu konfigurierender Objekte eine hohe Komplexität des Eigenschaftsbaumes. Eine Gefahr besteht darin, dass der Überblick über die Struktur und die konfigurierbaren Eigenschaften verloren geht. Zu diesem Zweck wurde ein Dokumentationsmechanismus entworfen, der die Gesamtheit aller möglichen gelesenen Knoten und Eigenschaften protokolliert. Diese Dokumentation wird von einem Beobachterobjekt erstellt, das alle abgerufenen Eigenschaften und Knoten protokolliert. Aus diesen Informationen wird anschließend der komplette abgerufene Eigenschaftsbaum rekonstruiert. Ermöglicht wird dies durch die Trennung von auslesen und verarbeiten der Eigenschaften (siehe Abbildung 4.35).

4.9.4.2 Logobjekte

Alle Subklassen von Initializable haben mittels `getLog(name : String) : Log` Zugriff auf verschiedene Logobjekte. Der Status dieser Logobjekte wird ebenfalls durch den Eigenschaftsservice festgelegt. Zu diesem Zweck haben alle abgerufenen Knoten des Eigenschaftsbaums, die zur Initialisierung einem Objekt übergeben werden, einen speziellen Kindknoten mit Namen *LogStreams*. In den Abbildungen wurde dieser Knoten aus Gründen der Übersichtlichkeit jeweils weggelassen. Eine Eigenschaft dieses Knotens steht für ein Logobjekt, und der Wert gibt den Status dieses Logobjektes an.

4.9.4.3 Methoden

Das Lesen der Eigenschaften erfolgt zunächst in der Methode `readProperties(p : PropertiesTreeNode) : Object[]` (siehe Abbildung 4.35). Der Zugriff auf Eigenschaften des übergebenen Knoten des Eigenschaftsbaums erfolgt lediglich in dieser Methode. Diese Methode wird zu zwei verschiedenen Zwecken aufgerufen: einerseits zur Generierung der Dokumentation, andererseits zum produktiven Einlesen von Eigenschaften und Kindknoten. Um im Falle der Generierung einer Dokumentation Informationen über die Art der abgerufenen Eigenschaften bzw. Kindknoten zu erhalten werden zusätzliche Informationen übergeben. Das Einlesen einer Eigenschaft erfolgt beispielsweise durch `getProperty(key : String, description : String, example : String) : String`. Anschließend werden von der Methode `readLogProperties(values : List, loggingProperties : PropertiesTreeNode)` die Stati der Logobjekte eingelesen.

Im Falle des produktiven Betriebs werden die gelesenen Eigenschaften anschließend in der Methode `storeProperties(values : Object[])` verarbeitet.

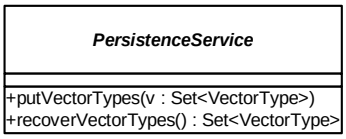

Abbildung 4.39: Persistenzservice

4.9.5 Persistenzservice

Die Aufgabe des Persistenzservice besteht darin, den Status einer Instanz von **ADF** persistent zu speichern. Hauptsächlich dient diese Speicherung dem Bewahren der Ergebnisse eines möglicherweise langwierigen Trainings. Diese Ergebnisse sind in den Vektormodulen repräsentiert. Die Vektormodule einer Sprache werden von einem Vektortyp verwaltet. Demzufolge müssen alle Vektortypen einer Instanz von **ADF** gespeichert werden.

Die Umsetzung wird von Subklassen der abstrakten Klasse `PersistenceService` (siehe Abbildung 4.39) realisiert. Die momentan einzige konkrete Subklasse nutzt den Java-Serialisierungsmechanismus.

4.10 Katalogmuster

Muster sind ein probates Mittel zur Dokumentation von Frameworks (siehe [Joh92] und [Meu99]). Das folgende Katalogmuster beschreibt den Sinn und Zweck von **ADF**.

Name des Frameworks **A**sset **D**iscovery **F**ramework - Framework zur Erschließung von Informationsobjekten

Klassifikation des Frameworks Domain-Framework, Glass-Box

Schlüsselworte Machine Learning, Data Mining, Artificial Intelligence, Classification, Clustering...

Problembeschreibung Infolge der technologischen Entwicklung fallen im täglichen Betrieb eines Unternehmens gewaltige Datenmengen an. Dabei kann es sich beispielsweise um in unterschiedlichen Sprachen verfasste Textdokumente, oder strukturierte Produktdaten handeln. Deren Erschließung ist zwar wünschenswert, aber aufgrund der schieren Menge manuell nicht durchführbar. Mögliche Erschließungsarten sind:

- Klassifikation,
- Clustering/ Visualisierung,
- Assoziative Suche.

Zur Unterstützung dieser Prozesse existieren eine Reihe von Algorithmen aus dem Umfeld des Machine Learning. Unbefriedigend gelöst ist das Problem der softwaretechnischen Einbindung existierender Algorithmen. Diese Einbindung sollte folgende Eigenschaften haben:

- flexible Wiederverwendung von Algorithmen in unterschiedlichen Kontexten,
- transparente Verarbeitung unterschiedlichster strukturierter Informationsobjekte,
- Erschließung von Textdokumenten in unterschiedlichen Sprachen,
- einfache Konfiguration einer Anwendung.

Lösungskonzept Asset Discovery Framework (kurz: **ADF**) ist ein in Java objektorientiert implementiertes Framework. Es unterstützt den Prozess der Umwandlung strukturierter Informationsobjekte in numerische Vektoren, deren Vorverarbeitung und deren anschließende Analyse. Die transparente Handhabung unterschiedlichster Informationsobjekte wird durch flexibel anpassbare Featureselektoren gewährleistet.

Die wesentlichen Schritte einer Analyse sind eine obligatorische Trainingsphase und eine anschließende Produktivphase. Beide Phasen erfordern eine Umwandlung von Informationsobjekten in numerische Vektoren und deren anschließende Verarbeitung. Die numerischen Vektoren werden durch Vektorisierer erzeugt. Die gewonnenen Vektoren werden anschließend von Präprozessoren vorverarbeitet. Die endgültige Analyse erfolgt in den Vektormaschinen.

Die Übermittlung gewonnener Resultate (beispielsweise getroffene Klassifikationen) erfolgt durch anwendungsspezifische Mediatorobjekte.

Beispiele

Textklassifikation im Rahmen eines Wissensmanagementsystems Ein vorhandenes Wissensmanagementsystem soll um Textklassifikationsfunktionalitäten erweitert werden. Der zu erstellende Automatische Klassifikator lernt anhand von Beispielen. Ein Benutzer ordnet einige Dokumente manuell den dazugehörigen Kategorien zu, anschließend lernt das System. Der automatische Klassifikator ist nun in der Lage, bisher unklassifizierte Dokumente automatisch den vorhandenen Kategorien zuzuordnen.

Assoziative Suche auf dem PC Jeder Benutzer eines PC hat eine große Menge von Dokumenten auf seinem System. Diese sind auf der ganzen Festplatte verstreut und somit schwer auffindbar. Das Tool *Personal DocumentFinder* analysiert in einem Trainingsschritt alle auf dem PC befindlichen Dokumente. Anschließend findet es zu einem eingegebenen Freitext die semantisch ähnlichsten Dokumente.

Dokumentation Die Dokumentation des Frameworks besteht in erster Linie aus der vorliegenden Arbeit.

Andere Frameworks Zu nennen ist das WEKA-Projekt der neuseeländischen Waikato-Universität (siehe [WF99]). Im Rahmen dieses Projektes wurde eine umfangreiche Bibliothek von Machine Learning Algorithmen in Java implementiert. Den Charakter eines Frameworks hat das WEKA-Projekt allerdings nicht. Ebenfalls fehlt die Möglichkeit verschiedene Vektorimplementierungen einzusetzen. Dadurch ist eine Anwendung auf die Erschließung natürlichsprachlicher Texte ineffizient.

Kapitel 5

Zusammenfassung und Ausblick

Am Schluss dieser Arbeit werden die Ergebnisse zusammengetragen und bewertet. Es folgt ein Ausblick auf offene Probleme und neu aufgeworfene Fragen.

5.1 Zusammenfassung und Bewertung

Als Resultat dieser Diplomarbeit liegt das in Java implementierte Framework **ADF** vor. Die Anwendungsgebiete von **ADF** liegen in den folgenden Bereichen:

- Implementierung von Algorithmen zur wissenschafltichen Untersuchung deren Eigenschaften, sowie Vergleich verschiedener Ansätze.

- Die produktive Erweiterung bestehender Anwendungen um intelligente Erschließungsfunktionalität.

Die im Framework implementierten Erschließungsarten sind die Klassifikation, das Clustering und die Assoziative Suche. Das dem Framework zugrundeliegende konzeptuelle Modell einer inhaltlichen Erschließung beinhaltet die Repräsentierung von Informationsobjekten als numerische Vektoren. Eine Erschließung besteht aus einer Trainingsphase und einer anschließenden Produktivphase. Diese beiden Phasen bestehen ihrerseits aus drei aufeinanderfolgenden Teilschritten: Vektorerzeugung, Vektorvorverarbeitung und Vektoranalyse.

Die Schwerpunkte beim Design dieses Frameworks liegen auf der Anpassbarkeit des Frameworks an existierende Anwendungen und einer performanten Umsetzung des Erschließungsprozesses. Die Anpassbarkeit wird durch anwendungsspezifische Featureselektoren und Mediatorschnittstellen erreicht. Der Forderung nach Performanz wird durch flexible Vektorimplementierungen Rechnung getragen.

5.2 Ausblick

Zum Abschluss wird ein Ausblick auf mögliche Weiterentwicklungen des Frameworks **ADF** gegeben:

- Der offensichtlichste Bedarf besteht in der Implementierung weiterer Algorithmen. Unter Benutzung der zur Verfügung stehenden Infrastruktur ist eine hohe Produktivität bei der Implementierung neuer Module möglich.
- Aufgrund der hohen Flexibilität des Frameworks bezüglich verschiedener Vektorimplementierungen ist eine Konsistenzüberprüfung notwendig. Diese Überprüfung erfolgt auf dem momentanen Entwicklungsstand größtenteils manuell. Eine Weiterentwicklung dieses zentralen Mechanismus ist wünschenswert.
- Mit **ADF** ist es möglich, die Leistungsfähigkeit verschiedener Algorithmen zu vergleichen. Zu diesem Zweck wird eine Implementierung von Auswertungsfunktionalität benötigt.

Literaturverzeichnis

[ADW94] Apte, Chidanand ; Damerau, Fred ; Weiss, Sholom M.: Automated Learning of Decision Rules for Text Categorization. **In:** *ACM Transactions on Information Systems* 12 (1994), Juli, Nr. 3, S. 233–251. – Special Issue on Text Categorization.. – ISSN 1046–8188

[BEK 98] Berchtold, Stefan ; Ertl, Bernhard ; Keim, Daniel A. ; Kriegel, Hans-Peter ; Seidl, Thomas: Fast Nearest Neighbor Search in High-Dimensional Spaces. **In:** *Proc. 14th IEEE Conf. Data Engineering, ICDE*, IEEE Computer Society, 23–27 Februar 1998. – ISBN 0–8186–8289–2

[BM98] Baker, L. D. ; McCallum, Andrew K.: Distributional clustering of words for text classification. **In:** Croft, W. B. (Hrsg.) ; Moffat, Alistair (Hrsg.) ; Rijsbergen, Cornelis J. (Hrsg.) ; Wilkinson, Ross (Hrsg.) ; Zobel, Justin (Hrsg.): *Proceedings of SIGIR-98, 21st ACM International Conference on Research and Development in Information Retrieval*. Melbourne, AU : ACM Press, New York, US, 1998, S. 96–103

[Bur98] Burges, Chris: A Tutorial on Support Vector Machines for Pattern Recognition. **In:** *Data Mining and Knowledge Discovery* 2 (1998), Nr. 2, S. 121–167

[Cra84] Crawford, Walt: *MARC for Library Use : Understanding the USMARC Formats. Professional Librarian Series.*. White Plains, NY, and London : Knowledge Industry Publications, 1984

[DK98] Deboeck, Guido ; Kohonen, Teuvo: *Visual Explorations in Finance with Self-Organizing Maps*. London : Springer-Verlag, 1998. – 258 p.

[Ger00] Gerick, Thomas: Retrieval: Methoden, Trends, Produkte. **In:** *iX* (2000), Nr. 12, S. 124–128

[Hac83] Hacker, Rupert: *Bibliothekarisches Grundwissen*. K. G. Saur, 1983

[IM98] Indyk, Piotr ; Motwani, Rajeev: Approximate nearest neighbors: towards removing the curse of dimensionality. **In:** ACM (Hrsg.): *Proceedings of the thirtieth annual ACM Symposium on Theory of Computing: Dallas, Texas, May*

23–26, 1998. New York, NY, USA : ACM Press, 1998. – ISBN 0–89791–962–9, S. 604–613

[JF88] JOHNSON, Ralph E. ; FOOTE, Brian: Designing Reusable Classes. **In:** *Journal of Object-Oriented Programming* 1 (1988), Nr. 2, S. 22–35

[Joa98a] JOACHIMS, Thorsten: Making large-scale SVM learning practical. **In:** SCHÖLKOPF, Bernhard (Hrsg.) ; BURGES, Chris (Hrsg.) ; SMOLA, Alex (Hrsg.): *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 1998, S. 169–185

[Joa98b] JOACHIMS, Thorsten: Text categorization with support vector machines: learning with many relevant features. **In:** *Proc. 10th European Conference on Machine Learning ECML-98*, 1998, S. 137–142

[Joa99] JOACHIMS, Thorsten: Transductive inference for text classification using support vector machines. **In:** *Proc. 16th International Conference on Machine Learning*, Morgan Kaufmann, San Francisco, CA, 1999, S. 200–209

[Joh92] JOHNSON, Ralph E.: Documenting frameworks using patterns. **In:** PAEPCKE, Andreas (Hrsg.): *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* Bd. 27. Bd. 27. New York, NY : ACM Press, 1992, S. 63–72

[Kle97] KLEINBERG, Jon M.: Two Algorithms for Nearest-Neighbor Search in High Dimensions. **In:** *ACM Symposium on Theory of Computing*, 1997, S. 599–608

[KOR98] KUSHILEVITZ, Eyal ; OSTROVSKY, Rafail ; RABANI, Yuval: Efficient search for approximate nearest neighbor in high dimensional spaces. **In:** ACM (Hrsg.): *Proceedings of the thirtieth annual ACM Symposium on Theory of Computing: Dallas, Texas, May 23–26, 1998*. New York, NY, USA : ACM Press, 1998. – ISBN 0–89791–962–9, S. 614–623

[KS97] KATAYAMA, N. ; SATOH, S.: The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. **In:** *SIGMOD Record (ACM Special Interest Group on Management of Data)* 26 (1997), Nr. 2. – ISSN 0163–5808

[Lew92] LEWIS, David D.: An evaluation of phrasal and clustered representations on a text categorization task. **In:** BELKIN, Nicholas J. (Hrsg.) ; INGWERSEN, Peter (Hrsg.) ; PEJTERSEN, Annelise M. (Hrsg.): *Proceedings of SIGIR-92, 15th ACM International Conference on Research and Development in Information Retrieval*. Kobenhavn, DK : ACM Press, New York, US, 1992, S. 37–50

[LJ98] LI, Y. H. ; JAIN, A. K.: Classification of Text Documents. **In:** *The Computer Journal* 41 (1998), 1998, Nr. 8, S. 537–546

[Meu99] MEUSEL, Matthias. Mustersysteme zur Dokumentation von Frameworks unter Verwendung von Hypertext. Diplomarbeit Fachhochschule Ulm. 1999

[MH00] MICHAEL HOST, Andreas B.: Visuell gestütztes Retrieval mit Dokumentenlandkarten. **In:** *Wissensmanagement* (2000), Juli/August

[Mou96] MOULINIER, I.: A Framework for Comparing Text Categorization Approaches. **In:** *AAAI Spring Symposium on Machine Learning and Information Access*, 1996

[Nem97] NEMIROVSKY, Adolfo. Building Object-Oriented Frameworks. Taligent Whitepaper: http://www.ibm.com/developerworks/library/oobuilding/index.html. 1997

[Por80] PORTER, M. F.: An Algorithm for Suffix Stripping. **In:** *Program* 14 (1980), Nr. 3, S. 130–137

[Run00] RUNKLER, Thomas A.: *Information Mining*. Braunschweig/Wiesbaden : Vieweg, 2000

[Sal89] SALTON, G.: *Automatic text processing*. Reading, MA : Addison-Wesley, 1989

[SBS98] SCHÖLKOPF, Bernhard ; BURGES, Chris ; SMOLA, Alex: Introduction to Support Vector Learning. **In:** SCHÖLKOPF, Bernhard (Hrsg.) ; BURGES, Chris (Hrsg.) ; SMOLA, Alex (Hrsg.): *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 1998, S. 1–22

[Sch96] SCHÜRMANN, Jürgen: *Pattern Classification*. Wiley-Interscience, 1996

[Seb99] SEBASTIANI, Fabrizio: A Tutorial on Automated Text Categorisation. **In:** AMANDI, Analia (Hrsg.) ; ZUNINO, Ricardo (Hrsg.): *Proceedings of ASAI-99, 1st Argentinian Symposium on Artificial Intelligence*. Buenos Aires, AR, 1999, S. 7–35

[SHP95] SCHUETZE, H. ; HULL, D. ; PEDERSEN, J.: A comparison of classifiers and document representations for the routing problem. **In:** *Proceedings of the Special Interest Group on Information Retrieval*, 1995

[WF99] WITTEN, Ian H. ; FRANK, Eibe: *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, 1999. – ISBN: 1–558–60552–5

[WSS97] WECHSLER, M. ; SHERIDAN, P. ; SCHAUBLE, P.: Multi-language text indexing for internet retrieval. **In:** *Proceedings of the 5th RIAO Conference, Computer-Assisted Information Searching on the Internet*, 1997, S. 217–232.

[Yan99] YANG, Yiming: An evaluation of statistical approaches to text categorization. **In:** *Information Retrieval* 1 (1999), Nr. 1-2, S. 69–90

[YL99] YANG, Y. ; LIU, X.: A re-examination of text categorization methods. **In:** *22nd Annual International SIGIR*. Berkley, August 1999, S. 42–49

[YP97] YANG, Yiming ; PEDERSEN, Jan O.: A comparative study on feature selection in text categorization. **In:** *Proc. 14th International Conference on Machine Learning*, Morgan Kaufmann, 1997, S. 412–420