Entwurf und Implementierung einer objektrelationalen Datenbankerweiterung für die automatische Schreiberklassifikation in historischen Notenhandschriften

Projektarbeit

Universität Rostock
Institut für Informatik

angefertigt von: BSc. Henning Masuch
geboren am: 19. März 1976 in Rostock
Gutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Betreuer: Dipl.-Ing. Temenushka Ignatova
Dipl.-Ing. Andreas Finger
Abgabedatum: 1. Dezember 2005

Zusammenfassung

Unstrukturierte Daten, wie z.B. digitale Bilder, beinhalten eine Vielfalt von Informationen, die mit Hilfe inhaltsbasierter Zugriffsverfahren für das Retrieval genutzt werden können. Dabei werden Inhalte etwa digitaler Bilder durch sogenannte Features, wie z.B. Farbverteilung oder Textur, repräsentiert. Diese Features können nun wieder benutzt werden, um unstrukturierte Daten miteinander zu vergleichen bzw. zu klassifizieren.

Für das Projekt „eNoteHistory“ existiert bereits eine Datenbank zur Verwaltung digitaler historischer Notenhandschriften. Neben verschiedenen Suchmöglichkeiten nach Eigenschaften wie z.B. Titel, Signatur des Werkes und Komponist soll eine automatische Schreiberidentifikation unbekannter Notenhandschriften implementiert und mittels objektrelationaler Erweiterungen in das Datenbanksystem integriert werden.

Hierzu wurden existierende Standards aus dem Bereich der Bildverarbeitung und des Data Mining sowie vorhandene Erweiterungen wie der „Intelligent Miner for Data“ und das Tool „Weka“ auf ihre Verwendbarkeit untersucht und berücksichtigt. Auf dieser Basis wurde eine vorhandene Implementierung zur Bildanalyse und Klassifikation zu speziellen Datenbanktypen mit entsprechender Funktionalität weiterentwickelt.

Inhaltsverzeichnis

1 Einleitung

eNoteHistory ist ein Gemeinschaftsprojekt des Instituts für Musikwissenschaften der Universität Rostock, des Lehrstuhls für Datenbank- und Informationssysteme der Informatik (DBIS) und des Fraunhofer Instituts für Grafische Datenverarbeitung (IGD). Ziel des Projektes ist die mit elektronischen Hilfsmitteln unterstützte Schreibererkennung in historischen Notenhandschriften.
Die Rostocker Universitätsbibliothek verfügt über eine umfangreiche Sammlung historischer Notenschriften. Diese „Rostocker Sammlung“ stammt vorwiegend aus dem 18. Jahrhundert und umfasst Werke zahlreicher Komponisten aus den verschiedensten Gebieten Europas. Zu jener Zeit wurden Kompositionen durch sogenannte Kopisten vervielfältigt. Die namentliche Zuordnung des Kopisten zu den jeweiligen Notenhandschriften ist dabei für die Musikforschung von großer Bedeutung. Sie ermöglicht sowohl eine Eingrenzung, wann eine Komposition geschrieben bzw. abgeschrieben wurde, als auch Aussagen über deren Verbreitung.
Es soll also ein möglichst einfach zu bedienendes Werkzeug entwickelt und bereitgestellt werden, welches Musikwissenschaftler bei der musikhistorischen Forschungsarbeit unterstützt. Die Webpräsentation und die bereits verfügbaren Hilfsmittel des Projektes sind unter http://www.enotehistory.de zu finden.

Für die Schreibererkennung kann neben dem verwendeten Papier, der Tinte und dem Wasserzeichen vor allem auch die Handschrift des Schreibers genutzt werden. Diesem Ansatz liegt die Annahme zugrunde, dass jeder Schreiber eine individuelle Handschrift besitzt, welche ihn charakterisiert. Anhand dieser ist dann eine Zuordnung einer Notenschrift, deren Kopist noch nicht bekannt ist, zu Schreibern, deren Schriftstil bereits analysiert ist, möglich.

Im Rahmen des Projektes *eNoteHistory* werden für diese Analyse zwei verschiedene Ansätze verfolgt. Auf der einen Seite ein semiautomatisches und auf der anderen Seite ein vollautomatisches Analyseverfahren.

Der semiautomatischen Analyse liegt eine von Musikwissenschaftlern ausgearbeitete Einteilung der Merkmale einer Notenschrift in 13 Klassen zugrunde. Diese Klassen beschreiben Notenschlüssel, Taktvorzeichen, Viertelpausen, Schriftneigung, Kaudierung, Form der Fähnchen, Schreibgewohnheiten, Laufweite, Bebalkung, Vorzeichen, Form der Notenköpfe, Schlusszeichen und Rastral der jeweils vorliegenden Notenschrift. In diesen Klassen gibt es insgesamt ca. 80 Merkmale, die den Schreiber charakterisieren. Die Merkmale einer Klasse sind hierarchisch (baumartig) strukturiert, vgl. Abbildung 1 und [DBIS05].

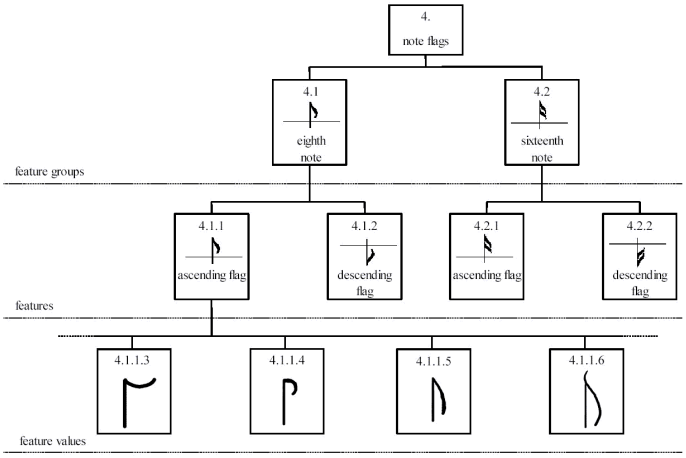

Abbildung 1: Baumhierarchie in der Merkmalsgruppe der Notenfähnchen

Die einzelnen Merkmale werden dabei manuell, also per Augenschein, analysiert. Hierfür ist eine Webschnittstelle auf dem Server des Projektes vorhanden. Die so definierten Merkmale (Features) werden systemintern als Vektoren dargestellt (Featurevektor). Mittels spezieller Funktionen ist die Berechnung der Distanz eines neuen Feature-Vektors zu bereits in der Datenbasis vorhandenen Feature-Vektoren (FV) möglich. Unterschreitet dieser Abstand einen bestimmten Schwellwert, ist ein möglicher Schreiber für die analysierte Notenschrift gefunden.
Dem Nutzer der Webschnittstelle werden schließlich auf Basis dieser Klassifikation alle in Frage kommenden Schreiber mit den jeweils ermittelten Distanzen präsentiert, so dass eine Zuordnung der neuen Notenhandschrift unterstützt wird. Nähere Informationen über die semiautomatische Analyse sind u.a. in [Mil04] und [DBIS05] zu finden.

Die vollautomatische Analyse wurde durch den Projektpartner IGD entwickelt. Hier basiert die Extraktion der Merkmale auf speziellen Bildverarbeitungsalgorithmen. Im Rahmen dieser Arbeit soll die Möglichkeit untersucht werden, diese Algorithmen mit dem Datenbanksystem zu vereinigen und somit eine komfortablere und ggf. effizientere Nutzung dieser Techniken zu ermöglichen.
Dazu werden im folgenden Kapitel 2 zunächst vorhandene Programme und Standards für die Speicherung von Bilder in Datenbanken und die automatische Erkennung von Mustern in großen Datenbeständen auf ihre Verwendbarkeit für die Aufgabenstellung untersucht. Anschließend erfolgt in Kapitel 3 eine Analyse der bereits vorhandenen Implementierung des IGD. Auf Grundlage dieser Voraussetzungen kann in Kapitel 4 ein Konzept für die Realisierung der automatischen Analyse entworfen, und dessen Implementierung in Kapitel 5 beschrieben werden.

Teil I
Grundlagen

2 Vorhandene Standards und Programme

2.1 Bildverwaltung in Datenbanksystemen

Das Vorhandensein ständig wachsender elektronischer Sammlungen digitalisierter Bilder führt zu einem Bedarf an Werkzeugen, welche eine effektive Verwaltung und Organisation dieser Bilder ermöglichen. Gegenwärtig existieren jedoch nur begrenzte Möglichkeiten zur Suche bestimmter Bilder in einem Datenbestand.

Die zur Zeit am weitesten verbreitete Methode ist, die Bilder mit einem Titel und Stichworten zu versehen. Ebenso können die Bilder in bestimmte Kategorien eingeordnet und auf diese Weise klassifiziert werden. Diese Metadaten können ggf. noch um physische Attribute des digitalen Bildes, wie das Speicherungsformat (JPEG, GIF) oder die Abmessung (Pixel) angereichert werden. Auf der Basis der Metadaten können dann unter Verwendung von Information-Retrieval-Techniken Anfragen formuliert und anhand eines Rankings sortierte Ergebnisse präsentiert werden. Allerdings erfordert diese Methode einen hohen Aufwand an menschlicher Arbeitsleistung und ist somit nur für eine begrenzte Menge an Bildern einsetzbar.

Eine andere Art der Suche nach Bildern ist unter den Schlagworten „Content Based Image Retrieval“ (CBIR) oder auch „Query By Image Content“ (QBIC[1]) bekannt. Hierbei handelt es sich um eine inhaltsbasierte Suche anhand der Bildinformationen selbst, nicht anhand von hinzugefügten Meta-Informationen.
Eine solche Suchanfrage könnte z.B. „Finde Bilder von Brücken “ oder „Finde Bilder von Goethe “ lauten. Aufgrund der Verschiedenartigkeit der so gesuchten Bilder sind derartige Anfragen i.d.R. zu komplex. Aus diesem Grund beschränken sich aktuelle CBIR-Systeme in erster Linie auf die Auswertung sogenannter *low-level features*, wie beispielsweise der Durchschnittsfarbe oder der Farbverteilung eines Bildes oder dem Vorhandensein einer charakteristischen Textur. Basierend auf der Auswertung einzelner oder einer Kombination dieser Merkmale ist die Suche nach Bildern, die beispielsweise einem gegebenen Bild oder einer Skizze ähnlich sehen, möglich. Diese Art der Anfrage ist als „query by example“ bzw. „query by sketch“ bekannt.

[1] vgl. Webpräsentation auf http://wwwqbic.almaden.ibm.com/

2.1.1 Proprietäre Produkte

Neben der Möglichkeit, Bilder in ihrer digitalen Form einfach als unstrukturierte Binärdaten (sogenannte BLOB 's) abzulegen, gibt es bereits eine Vielzahl verschiedener Produkte der unterschiedlichen Datenbank-Systeme für die Verwaltung von Bildern in Datenbanken. Hierbei werden die digitalen Bilder i.d.R. durch einen eigenen, speziellen Datentyp repräsentiert, welcher neben der Möglichkeit zur Speicherung der Binärdaten auch Funktionen zur inhaltsbasierten Suche und ggf. zur Bildbearbeitung bereitstellt.

Da im Rahmen des Projektes eNoteHistory das Datenbanksystem *IBM Universal Database* eingesetzt wird, sollen nachfolgend exemplarisch die Funktionen des IBM-Produktes *DB2 Image Extender* kurz vorgestellt werden.
Andere Produkte, wie beispielsweise das *Excalibur Image DataBlade Module*, das *Image Foundation DataBladeModule*[2] oder auch *Oracle* 8*i interMedia*[3] bieten ähnliche Funktionen an. Ein ausführlicher Vergleich dieser Produkte ist unter [Sto03] nachzulesen.

Der **DB2 Image Extender** wird von IBM als Teil des DB2 UDB Audio, Image, and Video Extender Pakets [IBM01] angeboten. Er stellt dem Nutzer des Datenbanksystems den Datentyp `DB2Image` zur Verfügung. Dieser Datentyp kapselt dabei die Bilddaten selbst und die dazugehörigen Attribute. Gespeichert werden die Daten in administrativen Tabellen (auch als Seitentabellen bekannt), der Zugriff auf die Daten erfolgt über ein sogenanntes *handle*, welches jeweils die Tabelle und das Tupel identifiziert. Das Bild selbst kann dabei entweder als *Binary Large Object* (BLOB) oder als Datei im lokalen Dateisystem gespeichert werden.
Die Funktionen zur Bearbeitung der Bilder reichen von Methoden zur Typkonvertierung, über Skalierung und Drehung des Bildes bis zu Farbinvertierungen, Reduzierung der Farbtiefe und Änderung der Komprimierung. Diese Methoden sind jedoch nicht als jeweils eigenständige, orthogonal anwendbare Funktionen (z.B. UDF) implementiert, sondern zu einer einzelnen Operation zusammengefasst. Der Vorteil hierbei liegt in der Performance, da die Bilddaten nur einmal gelesen, dann bearbeitet und zurückgeschrieben werden. Ein Funktionsaufruf zum Einfügen eines neuen Bildes im *Graphic Interchange Format* (GIF), welches auch in diesem Format gespeichert, aber auf 50% seiner ursprünglichen Größe skaliert werden soll, würde folgendermaßen aussehen:

```
DB2Image(CURRENT SERVER, 'image.gif', 'GIF', 'GIF', '-s
0.5', '', '')
```

[2] Sowohl das *Excalibur Image DataBlade Module* als auch das *Image Foundation DataBladeModule* sind Erweiterungen des Datenbanksystems *Informix Dynamic Server*.

[3] *Oracle* 8*i interMedia* in Kombination mit *Oracle* 8*i Visual Information Retrieval* stellt die Funktionalität für die Verwaltung und Bearbeitung von Bildern in *Oracle* 8*i* und *Oracle* 9*i*-Datenbanksystemen zur Verfügung.

Bei der inhaltsbasierten Suche in Bildern werden insgesamt vier verschiedene Merkmale unterstützt: die Durchschnittsfarbe (*average color*), das Farbhistogramm (*positional color*), die positionsabhängige Farbe (*positional color*) und die Textur (*texture*).

Die Durchschnittsfarbe (*average color*) wird dabei folgendermaßen berechnet: Für jede Komponente des Farbwertes wird dessen Mittelwert über alle Bildpunkte (Pixel) der Bildes errechnet. Für den RGB-Farbraum sind die Komponenten rot, grün und blau. Die Durchschnittsfarbe besteht nun aus diesen einzelnen, gemittelten Farbanteilen. Dieser einzelne Wert hat entsprechend nur eine relativ geringe Aussagekraft hinsichtlich der Charakteristik eines Bildes. So hat beispielsweise ein Bild, welches je zur Hälfte rot und grün ist die gleichen Durchschnittsfarbe wie ein vollständig braunes Bild.
Für die Berechnung der positionsabhängigen Farbe (*positional color*) wird das Bild zunächst in eine Vielzahl kleinerer Rechtecke zerlegt (Regionen) und anschließend wird für jede dieser Regionen die Durchschnittsfarbe, wie zuvor beschrieben, bestimmt.
Ein Farbhistogramm (*color histogram*) beschreibt die relative Häufigkeit der verschiedenen Farbwerte der einzelnen Pixel des Bildes. Dafür wird der gesamte Farbraum in ein 64-Farben-Spektrum aufgeteilt und jeder Bildpunkt einem dieser Farbräume zugeordnet.
Die Textur (*texture*) dient der Suche nach bestimmten Mustern in einem Bild. Dafür wird die Grobheit, der Kontrast und die Direktionalität des Bildes gemessen. Die Grobheit gibt dabei die Größe sich wiederholender Elemente, der Kontrast vorhandene Helligkeitsvariationen in einem Bild und die Direktionalität die Richtung, in welche ein Abbild dominiert, an.

Anhand eines dieser oder einer gewichteten Kombination aller Merkmale kann die Distanz (*score*) zwischen zwei verschiedenen Bildern berechnet werden. Die Bedeutung des *score*-Wertes ist dabei nicht genauer dokumentiert; er kann somit lediglich für eine Sortierung der Ergebnisse verwendet werden. Für weiterführende Informationen und Beispiele zur Anwendung sei neben den bereits genannten Quellen auch auf [Kul04] verwiesen.

2.1.2 Standardisierung

Der Standard SQL:1999 erweitert das relationale Datenmodell von SQL-92 um einige objektorientierte Aspekte. Ziel dieser sogenannten objektrelationalen Erweiterungen ist die damit verbundene Möglichkeit, Anwendungslogik besser in die Datenbank integrieren zu können. Dieses geschieht im Wesentlichen durch die Definition benutzerdefinierter Datentypen (*user defined types, UDT*), typisierte Tabellen und Sichten sowie Typ-, Tabellen- und Sichtenhierarchien [Tue03].
Die Spezifikation von SQL:1999 ist dabei in einen Kern und verschiedene Zusatzpakete aufgeteilt, wobei der Kern nur vergleichsweise wenige neue

Sprachkonstrukte gegenüber SQL-92 aufweist. Die meisten objektrelationalen Erweiterungen sind in den einzelnen Zusatzpaketen enthalten. Eines dieser Pakete (PKG009) ist für die Verarbeitung von Multimedia-Daten konzipiert. Dieses Framework (ISO/ IEC 13249 SQL/MM) ist der Versuch einer Standardisierung der im vorigen Kapitel exemplarisch vorgestellten proprietären Erweiterungen der verschiedenen Datenbanksysteme.

Der Standard SQL/MM besteht aus mehreren Teilen. Der erste Teil (Part 1: Framework [ISO02]) gibt dabei zunächst einen Überblick über den gesamten Standard. Jeder weitere Teil ist dann ein Paket für eine bestimmte Art von Mediendaten und besteht aus einer Menge von Datentypen, Funktionen und Methoden auf Basis von SQL:1999.
Teil 2 des Standards (SQL/MM Full-Text) bietet beispielsweise durch Bereitstellung des Datentyps `FullText` Methoden für boolesche oder lineare Suche in Textdokumenten an. Ein weiterer Teil des Standards (SQL/MM Spatial) dient der Verwaltung von Grafikdaten und stellt Routinen für die Manipulation, die Suche und den Vergleich von räumlichen Daten zur Verfügung.
Im Folgenden soll nun der 5. Teil des Standards SQL/MM [ISO03], welcher die Art der Speicherung und Bearbeitung von Bildern in Datenbanken definiert, ausführlicher vorgestellt werden.

Dabei dient **SQL/MM Still Image** nicht der Standardisierung der Bilder und Formate selbst, sondern vielmehr der Schnittstelle für deren Handhabung in Datenbanksystemen. Der Standard ist in insgesamt 10 Abschnitte aufgeteilt. Die Abschnitte 1 bis 4 bestehen dabei vor allem aus Definitionen, Bezeichnungen und einer Einordnung des Standards in den Kontext weiterer Standards und Konzepte. Auf dieser Basis wird dann in Abschnitt 5 der Datentyp `SI_StillImage` einschließlich seiner Funktionen und Methoden definiert. In Abschnitt 6 werden anschließend die Bildmerkmale (*feature types*) und die dazu gehörigen Operationen beschrieben. In den verbleibenden Abschnitten 7-10 werden u.a. die Status Codes (`SQLSTATE`s) aufgelistet und das Informations- und Definitionsschema beschrieben.
Nachfolgend soll nun der in Abschnitt 5 und 6 definierte Datentyp `SI_StillImage` und dessen *feature types* kurz vorgestellt werden.

Der Datentyp `SI_StillImage` stellt einen Container für die Speicherung von Bilddaten zur Verfügung, der folgende Attribute besitzt: `SI_content`, `SI_contentLength`, `SI_format`, `SI_height` und `SI_width`. Das Attribut `SI_content` speichert dabei die kompletten Bilddaten bestehend aus den Pixeldaten und dem jeweiligen Dateiheader des Bildes in Form eines *Binary Large Object*s. Der Wert `SI_contentLength` gibt die Größe von `SI_content` an und entspricht somit der Größe der Bilddatei in bytes. Die verbleibenden 3 Attribute sind sogenannte ableitete Attribute, welche aus den Bilddaten ermittelt werden. `SI_format` gibt das Format des Bildes an. Es muss sich

hierbei jeweils um ein von der entsprechenden Implementierung unterstütztes Format handeln. Die Attribute `SI_height` und `SI_width` schließlich geben die Abmessungen des Bildes in Pixeln an, sofern sich diese Werte für das aktuelle Format ermitteln lassen.

Basierend auf diesen Attributen werden durch den Datentyp verschiedene Methoden zur Verfügung gestellt. Neben den sogenannten Observer-Methoden, welche den aktuellen Zustand eines Attributes zurückliefern, sind weitere Methoden vorgesehen, welche alle jeweils einen Wert vom Typ `SI_StillImage` zurückliefern.
Die Methode `SI_StillImage` mit der Signatur `SI_StillImage(BINARY LARGE OBJECT [, CHARACTER VARYING])` dient zum Erstellen eines neuen Bildes und kann deshalb als *Konstruktor* bezeichnet werden. Parameter dieser Methode sind die Binärdaten des zu speichernden Bildes und optional eine Angabe des vorliegenden Formats.
Die Methode `SI_setContent(BINARY LARGE OBJECT)` dient zum Aktualisieren eines bereits in der Datenbank vorhandenen Bildes. Die abgeleiteten Attribute des Bildes werden beim Aufruf dieser Methode automatisch aktualisiert. Des weiteren gibt es eine Methode zur Änderung des Formates, in dem das Bild gespeichert ist: `SI_format(CHARACTER VARYING)`. Welche Formatkonvertierungen möglich sind, ist von der jeweiligen Implementierung abhängig. Sie werden in der Sicht `SI_IMAGE_FORMAT_CONVERSIONS` des Information Schema hinterlegt.
Schließlich ist noch eine Methode zur Erzeugung von Vorschaubildern vorgesehen: `SI_Thumbnail([INTEGER, INTEGER])`. Diese Methode nimmt optional die gewünschten Abmessungen des Bildes entgegen.

In Abschnitt 6 des Standards SQL/MM-5 werden die Merkmale (*feature types*), die zur inhaltsbasierten Suche genutzt werden können, beschrieben. Folgende Datentypen werden definiert: `SI_AverageColor`, `SI_ColorHistogram`, `SI_PositionalColor` und `SI_Texture`. Es werden somit dieselben *low-level features* unterstützt, die bereits für den DB2 Image Extender exemplarisch vorgestellt wurden.
Diese vier Merkmalstypen sind jeweils vom abstrakten Datentyp `SI_Feature` abgeleitet. Für diesen Datentyp ist die Methode `SI_Score(SI_StillImage)` definiert. Der Rückgabewert dieser Methode gibt an, wie gut das als Parameter übergebene Bild dem jeweiligen Merkmal entspricht. Dazu wird die Methode bei den abgeleiteten Datentypen jeweils durch eine spezielle Implementierung überschrieben.
Wie bereits beim DB2 Image Extender ist auch in diesem Standard die Suche anhand einer Kombination der einzelnen Basismerkmale möglich. Hierzu dient der Datentyp `SI_FeatureList`. Dieser Datentyp stellt zusätzlich zu der Methode `SI_Score` (wie vor) die Methoden `SI_Append` und `SI_FeatureList` bereit. Dabei wird die Methode `SI_FeatureList(SI_Feature, DOUBLE)` ge-

nutzt um basierend auf einem Merkmal eine neue Kombination von Merkmalen zu initiieren und `SI_Append(SI_Feature, DOUBLE)` dazu, einer bereits vorhandenen Kombination ein weiteres Merkmal hinzuzufügen. Beide Methoden erwarten als zweiten Eingabeparameter einen Wert für die Gewichtung des jeweils hinzugefügen Merkmals.

Eine ausführliche Beschreibung und Kommentierung des Standards ist unter [Sto01] zu finden. Eine prototypische Implementierung des Standards soll im folgenden Abschnitt kurz beschrieben werden.

2.1.3 DB2 UDB Still Image Extender

Das wesentliche Ziel der Entwicklung dieses Prototyps bei IBM war die Implementierung einer zum Standard SQL/MM kompatiblen Datenbankerweiterung (Extender) anstelle des derzeit angeboten DB2 Image Extenders. Außerdem sollten die Performance optimiert werden und gleichzeitig die erforderlichen Änderungen an der Datenbank-Engine und damit sowohl Aufwand, als auch Kosten möglichst minimal bleiben.

Die Implementierung des Prototyps basiert dabei nicht auf der des vorhandenen Extenders, sondern besteht aus einem Paket von speziellen strukturierten Datentypen und benutzerdefinierten Funktionen. Dabei stammt die Funktionalität für die Bildbearbeitung einerseits und die Unterstützung der inhaltsbasierten Suche andererseits aus den Projekten ImageMagick und QBIC.

Bei ImageMagick[4] handelt es sich um eine Bibliothek von Bildbearbeitungsalgorithmen. Diese ist als Open Source Software unter einer GPL-kompatiblen Lizenz frei verfügbar. Dem Nutzer stehen eine Vielzahl von Funktionen zum Erzeugen, Bearbeiten und Speichern von Grafiken zur Verfügung. Darüber hinaus wird das Lesen, Schreiben und Konvertieren zwischen einer Vielzahl an Bildformaten unterstützt. ImageMagick kann dabei entweder als Kommandozeilen-Programm oder als Bibliothek für Programme anderer Programmiersprachen verwendet werden. So sind Schnittstellen u.a. für C, C++, Java, Perl und PHP vorhanden.

QBIC [IBM02] hingegen ist ein Projekt von IBM mit dem Ziel der Entwicklung von Methoden für die inhaltsbasierten Suche in Bilddokumenten. Bereits der DB2 Image Extender nutzt diese Funktionen bei der Auswertung von Anfragen auf der Grundlage der bereits mehrfach beschriebenen *low-level features*.

Unter [Sto02] sind schließlich die erforderlichen Arbeitsschritte beschrieben, die für die Implementierung der benutzerdefinierten Funktionen (UDF) nötig

[4] vgl. Webpräsentation unter http://www.imagemagick.org/

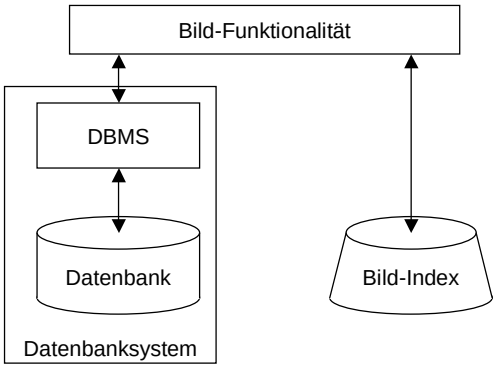

Abbildung 2: Schemat. Darstellung des Still Image Extenders nach [Sto01]

sind. Nach Installation dieser UDF's stehen dem Nutzer des Datenbanksystems bereits eine Vielzahl an Methoden für die Verwaltung und Manipulation von Bilddokumenten zur Verfügung, z.B. `SI_rotate`, `SI_scale` und `SI_crop`. Jede dieser UDF's ist dabei als separate C++-Funktion implementiert.
Durch die Definition eines entsprechenden benutzerdefinierten Datentyps `SI_StillImage` können anschließend die zuvor implementierten Methoden zu einer dem SQL/MM-Standard konformen Datenbankerweiterung, dem DB2 UDB Still Image Extender, zusammengefasst werden.
Somit können nunmehr die Funktionen in einer SQL-Notation angesprochen werden. Nachfolgend werden exemplarisch die erforderlichen Anweisungen zum Anlegen einer neuen Relation zur Speicherung von Bildern, dann zum Einfügen eines neuen Bildes in die Datenbank und schließlich zur Manipulation dieses Bildes vorgestellt:

```
CREATE TABLE t(id INTEGER, img SI_StillImage)
INSERT INTO t VALUES(1, SI_StillImage(SI_loadImage('test.gif')))
UPDATE t SET img=img..SI_shear(10, 0) WHERE id=1
```

Für die Implementierung der UDF's werden neben den eigentlichen Bildverarbeitungsroutinen der ImageMagick-Bibliothek noch einige Hilfsfunktionen benötigt. Dazu gehören neben einer Infrastruktur für die Behandlung von Fehlern und Funktionen für die Umwandlung der Bilddaten in BLOB's und umgekehrt auch Methoden zur Verwaltung des Scratchpads[5]. Der vollständige Quellcode des DB2 Still Image Extenders ist ebenfalls unter [Sto02] zu finden.

Um die inhaltsbasierte Suche in den Bilddokumenten zu optimieren, ist das Anlegen spezieller Indexstrukturen erforderlich. Die Indexdaten werden hier

[5] Als *scratchpad* (deutsch: Notizblock) wird in diesem Zusammenhang der bei der Ausführung von UDF's genutzte Zwischenspeicher bezeichnet.

in externen Dateien gespeichert (vgl. Abbildung 2). Eine interne Speicherung dieser Daten wäre effizient ohne Änderungen am Datenbanksystem selbst nicht möglich [Sto01].
Zur Verbesserung der Performance zum Zeitpunkt der Anfrage wird für jedes der vier Basismerkmale ein separater Index angelegt. Durch eine Sortierung der Bildidentifikatoren im Index anhand des jeweiligen Score-Wertes ist eine effiziente Bearbeitung, beispielsweise von Top-N-Anfragen[6], möglich.

2.1.4 Bewertung

Die vorgestellten Produkte und Standards stellen für die inhaltsbasierten Suche in Bildern lediglich sogenannte *low-level-features* zur Verfügung. Diese sind zwar für die Erkennung einander optisch ähnlicher Bilder ausreichend, aber für die Extraktion von handschrift-spezifischen Merkmalen völlig unzureichend. Die Funktionalität, wie sie das im Kapitel 3.1 beschriebene Framework des Projektpartners IGD bereitstellt, muss zusätzlich in das System integriert werden.

Eine Erweiterung des DB2 Image Extenders ist aufgrund der Art der Implementierung und Speicherung der Daten nicht möglich. Hingegen ist eine Erweiterung des DB2 UDB Still Image Extenders um die zuvor beschriebene Funktionalität möglich. Dafür sprechen insbesondere folgende Aspekte:

1. Die vorhandene Infrastruktur zum Lesen und Schreiben der Binary Large Objects sowie die Fehlerbehandlungsroutinen könnten genutzt werden.

2. Die Funktionalität zur Bearbeitung von Bildern wird zwar gegenwärtig nicht benötigt, aber für eine mögliche zukünftige Nutzung stünden diese dann zur Verfügung. Insbesondere die automatische Erzeugung von Vorschaubildern könnte im Rahmen der Webpräsentation des Projektes genutzt werden.

Folgende Punkte sprechen allerdings gegen eine Erweiterung des Extenders:

1. Die Funktionen des DB2 UDB Still Image Extender ist in der Programmiersprache C++, das Framework zur Analyse der Notenhandschrift hingegen in Java geschrieben. Die Nutzung derselben internen Hilfsfunktionen (z.B. Verwaltung des *Scratchpads*, Laden und Schreiben der BLOB's) ist somit nicht möglich.

2. Der Standard SQL/MM und entsprechend auch dessen prototypische Implementierung unterstützen lediglich eine datenbankinterne Speicherung der Bilder mittels des Datentyps `BLOB`. Eine externe Speicherung der Daten, wie sie beispielsweise der DB2-Datentyp `DATALINK` zur

[6] Top-N-Anfragen liefern jeweils nur die ersten n Tupel der Ergebnisrelation.

Verfügung stellt, ist nicht vorgesehen. Aufgrund der Größe der Bilddaten (bis zu 50 MB je Bild) ist eine externe, eventuell verteilte Datenhaltung wünschenswert. Dieses ermöglicht der Datentyp `DATALINK` durch die Referenzierung der Dateien über deren URL.

Aufgrund des zu erwartenden hohen Implementierungsaufwandes einer Erweiterung des DB2 UDB Still Image Extenders um die Funktionalität zur externen Speicherung der Daten wird die Erstellung einer eigenen Datenbankerweiterung (Extender) avisiert. Das Interface des damit bereitgestellten Datentyps wird sich dabei an dem des Standards SQL/MM orientieren. Ebenso wird sich die Art der Implementierung an der des DB2 Still Image Extenders orientieren. Auch das Problem der Indexierung von speziellen Attributen kann bei Bedarf auf die in Abschnitt 2.1.3 beschriebene Art und Weise gelöst werden.

2.2 Data Mining

Für den Begriff „Data Mining“ (DM) existieren eine Vielzahl an Definitionen, die sich insbesondere in den verschiedenen Anwendungsbereichen unterscheiden. Allgemein lässt sich DM als die Anwendung von Algorithmen auf Datenbestände mit dem Ziel, vorhandene Muster und Zusammenhänge zu erkennen, charakterisieren. In [FPS96] wird DM als ein Teilschritt des „Knowledge Discovery in Databases“ (KDD)-Prozesses verstanden. Bevor eine genaue Einordnung des Data Mining in den KDD-Prozess erfolgen kann, soll zunächst die Bedeutung und der Zusammenhang der Begriffe „Daten“ , „Informationen“ und „Wissen“ im hier verwendeten Sinne näher betrachtet werden.

Daten sind eine (maschinen-) lesbare und bearbeitbare Repräsentation von Informationen. Daten werden dadurch zu Informationen, indem sie in einem bestimmten Zusammenhang verwendet werden. Ein numerischer Wert kann beispielsweise abhängig von seinem Kontext als Telefonnummer oder als Größe bzw. Masse eines Objektes verstanden werden.
Information im pragmatischen Sinne führt zu einem Gewinn an Wissen bzw. ermöglicht die Verringerung von Ungewissheit. Somit kann die Information als eine Teilmenge von Wissen verstanden werden. Wissen allerdings ist weit mehr als eine Menge von Informationen. Wissen umfasst darüber hinaus begründete Verfahren zur Ableitung neuen Wissens aus bereits bekanntem.

Ziel des KDD-Prozesses ist somit das Entdecken neuen Wissens anhand der beispielsweise in Datenbanken vorhandenen Informationen. Teilschritte dieses Prozesses sind u.a. die Auswahl und Bereinigung der Daten, die Anwendung von Data Mining-Algorithmen und schließlich die Interpretation und Anwendung des gewonnenen Wissens (vgl. Abbildung 3).

In der Literatur werden für die Anwendung von DM- bzw. KDD-Verfahren

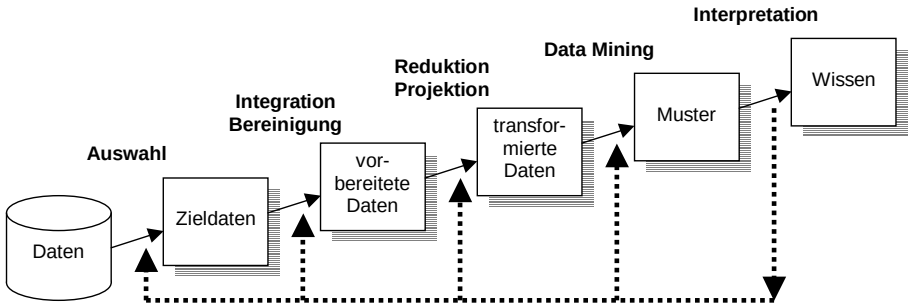

Abbildung 3: Schritte des KDD-Prozesses, nach [FPS96]

oft Szenarien aus dem betriebswirtschaftlichen Umfeld angeführt. So sollen beispielsweise aus den gespeicherten Warenkörben von Kunden deren Einkaufsgewohnheiten ermittelt werden oder Kundengruppen entsprechend eines ähnlichen Kaufverhaltens für Marketingzwecke segmentiert werden.

Im Projekt *eNoteHistory* werden dieselben statistischen Verfahren für die Schreiberidentifikation in historischen Notenhandschriften genutzt. Hier liefern Bildverarbeitungsalgorithmen Informationen u.a. über die Form der Taktstriche, Notenhälse und -köpfe. Diese werden in Form mehrerer sogenannter Feature-Vektoren in einer Datenbank gespeichert. Ziel des KDD-Prozesses ist es hier, den Zusammenhang zwischen diesen Daten und dem Schreiber des dazugehörigen Notenblattes zu erkennen. Dieses Wissen kann dann für die Klassifikation bis dato nicht zugeordneter Notenschriften zu einem Schreiber genutzt werden.

Im folgenden Abschnitt sollen nun allgemeine Begriffe und Techniken aus dem Bereich des Data Mining erklärt werden. Anschließend werden einige Standards aus dem KDD-Umfeld sowohl hinsichtlich der Repräsentation von Daten als auch hinsichtlich der Umsetzung des DM- bzw. KDD-Prozesses vorgestellt. Darauf aufbauend werden zwei konkrete Werkzeuge näher betrachtet, in Abschnitt 2.2.3 der IBM DB2 Intelligent Miner und in Abschnitt 2.2.4 das Java-Framework „WEKA“. Abschließend sollen die vorgestellten Standards und Programme bezüglich ihrer Eignung für das in Kapitel 4 aufzustellende Konzept untersucht und bewertet werden.

2.2.1 Grundlagen

In diesem Abschnitt sollen insbesondere die Ein- und Ausgabeparameter von DM-Algorithmen genauer betrachtet werden. Die Eingabewerte stellen dem Algorithmus die vorhandenen Informationen zur Verfügung, und das Ergebnis ist ein Modell zur Darstellung der in den Daten erkannten strukturellen Muster.

Die Eingabedaten werden durch eine Menge von Instanzen beschrieben. In der Terminologie von [WF00] wird das, was gelernt werden soll, auch als „Konzept“ bezeichnet. Eine einzelne Instanz stellt somit ein Beispiel für ein Konzept dar, und das Ergebnis des DM-Algorithmus, also das Modell zur Beschreibung der Daten, wird als Beschreibung des Konzepts bezeichnet.

Jede Instanz wird durch die konkreten Werte ihrer Attributmenge beschrieben. Die einzelnen Attribute können dabei verschiedene Typen sein, z.B. können diese Attribute in kontinuierliche und diskrete Werte unterschieden werden.
Zu den kontinuierlichen Attributen gehören die Typen *ratio* und *interval*. Der Unterschied zwischen diesen beiden Typen ist, dass für *ratio*-Typen ein fester Nullpunkt bekannt ist und für *interval*-Typen nicht. Für Attribute vom Typ *ratio* sind somit auch alle mathematischen Operationen sinnvoll, für interval-Typen ist hingegen nur die Differenz, also die Abstand voneinander sinnvoll. Ein Attribut vom Typ *ratio* könnte beispielsweise der Preis eines Gegenstandes oder der Umsatz eines Unternehmens sein - Datumsangaben hingegen sind ein Beispiel für ein *interval*.
Diskrete Attribute können nur bestimmte Werte einer endlichen Menge annehmen. Hier kann zwischen *nominalen* und *ordinalen* Typen unterschieden werden. Der Unterschied zwischen diesen beiden Typen besteht in der Ordnung, die bei *ordinalen* Typen auf der Menge definiert ist und bei nominalen Typen nicht. Ein Beispiel für eine *ordinale* Attributmenge ist die verbale Beschreibung der Temperatur: „kalt“ < „warm“ < „heiss“. Für eine Menge von Schreibern von Notenschriften hingegen lässt sich eine solche Ordnung nicht konstruieren. Das Attribut für eine solche Menge ist deshalb vom Typ *nominal*.
Eine Menge von Instanzen gleichen Typs, also mit der gleichen Attributmenge, lässt sich dann beispielsweise tabellarisch darstellen. Weitere Anforderungen an die Attribute der Instanzen sind von der jeweils gewählten Lernmethode abhängig.

Grundlegend können zwei verschiedene Arten der Beschreibung von Mustern in Datenbeständen unterschieden werden: prädiktive und deskriptive Modelle. Deskriptive Modelle beschreiben allgemeine Eigenschaften der Daten, prädiktive Modelle hingegen erlauben Schlussfolgerungen und einen Ausblick auf künftige Entwicklungen basierend auf den vorhandenen Daten. Zu den deskriptiven DM-Techniken gehören beispielsweise Assoziationsregeln und die Clusteranalyse, zu den prädikativen Verfahren gehören u.a. die Klassifikation und die Regression. Nachfolgend soll je ein Vertreter der beiden Techniker, die Clusteranalyse und die Klassifikation, näher betrachtet werden.

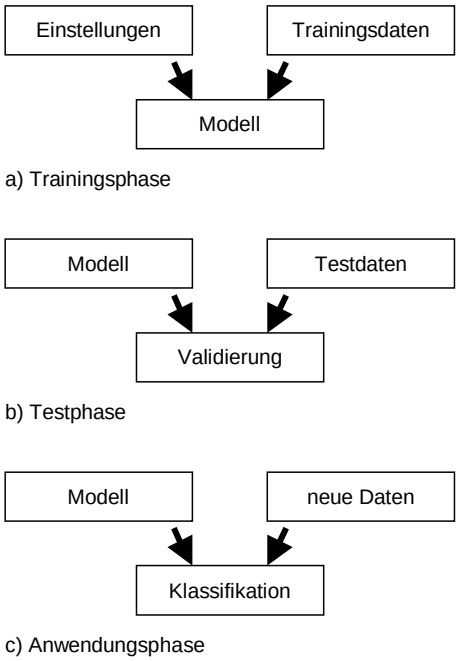

Abbildung 4: Schritte des Klassifikationsprozesses, nach [Sch00]

Klassifikationsverfahren ermöglichen die Zuordnung von Objekten bzw. Instanzen zu verschiedenen vorgegebenen Klassen. Die Eingabedaten sind dabei bereits jeweils einer bestimmten Klasse zugeordnet. Diese Klassenzugehörigkeit wird durch ein spezielles, nominales Attribut, dem sogenannten Klassenattribut, beschrieben. Der Wert dieses Klassenattributes wird auch als *Klassenwert der Instanz* bezeichnet. Ziel des DM-Algorithmus ist die Erstellung eines Modells, welches die Zuweisung des Klassenwertes zu einer neuen Instanz ermöglicht.

Der Klassifikationsprozess kann in mehrere Phasen eingeteilt werden. In Abbildung 4 ist er als 3-stufiger Prozess dargestellt, in [HK01] hingegen werden die Test- (b) und die Anwendungsphase (c) als ein Schritt verstanden.
Der erste Schritt, die Trainingsphase (a) umfasst die Erstellung des Modells oder auch Konzeptes zur Beschreibung der Daten. Dazu wird eine ausgewählte Menge von Instanzen, die sogenannten *Trainingsdaten* anhand ihrer Attribute, insbesondere des Klassenattributes, analysiert. Die verschiedenen Algorithmen für diese Analyse und die sich daraus ergebende Art des Modells werden später noch ausführlicher diskutiert. Weiterhin ist offensichtlich, dass die Auswahl und Vorbereitung der Trainingsdaten von entscheidender Bedeutung für die Qualität des Modells sein wird. Auch hierzu werden nachfolgend noch weitergehende Betrachtungen angestellt.

Im zweiten Schritt, der Testphase, wird dann die Genauigkeit des zuvor konstruierten Modells untersucht. Dazu wird eine Menge von Instanzen, die sogenannte Testmenge, gegen das Modell getestet. Dabei wird jeweils das anhand des Modells ermittelte Klassenattribut mit dem tatsächlichen Klassenattribut der Instanz verglichen. Die Genauigkeit (*accuracy*) des Modells ergibt sich dann als Quotient der richtig klassifizierten Instanzen zur Gesamtzahl getesteter Instanzen. Damit diese Genauigkeit nicht zu optimistisch bestimmt wird, sollten die Trainings- und die Testmenge disjunkt sein. Ist die so bestimmte Genauigkeit des Modells ausreichend, kann es schließlich für die Klassifikation neuer Objekte bzw. Instanzen im Rahmen der Anwendungsphase genutzt werden.

Die Auswahl und Vorbereitung der Trainingsdaten ist entscheidend für die Genauigkeit des Modells. Aufgrund eines Mangels an deterministischen Verfahren werden im Allgemeinen die Instanzen der Trainingsmenge zufällig ausgewählt. Gerade deshalb kommt der Testphase die zuvor beschriebene große Bedeutung zu, da durch diese nachträgliche Überprüfung die Wahrscheinlichkeit, nicht-repräsentative Instanzen für die Modellbildung verwendet zu haben, minimiert werden kann.
Bevor die ausgewählten Test-Instanzen jedoch für die Modellbildung genutzt werden, erfolgt das sogenannte *Preprocessing* der Daten (vgl. Schritt 3 und 4 des KDD-Prozesses, Abbildung 3). Ziel dieser Schritte ist sowohl die Verbesserung der Genauigkeit des Modells, als auch die Optimierung des Klassifikationsprozesses in Hinblick auf Effizienz und Skalierbarkeit. Auf eine ausführliche Beschreibung einzelner *Preprocessing*-Algorithmen wird an dieser Stelle verzichtet und auf [HK01] bzw. die dort genannten Literaturhinweise verwiesen.

Für die Klassifikation von Instanzen existiert eine Vielzahl verschiedener Techniken, wie z.B. Entscheidungsbäume, Bayesische Klassifikation, Neuronale Netze oder instanzbasierte Klassifikation. Abhängig von der jeweiligen Technik wird dann das Modell entsprechend in Form von Klassifikationsregeln, als Baum oder als mathematische Formeln dargestellt. Im folgenden Abschnitt soll die Idee, welche Entscheidungsbäumen zu Grunde liegt, kurz erläutert werden.

Entscheidungsbäume sind eine spezielle Form von Entscheidungsregeln, welche hierarchisch die Abfolge der einzelnen Regeln veranschaulichen. An jedem Knoten des Baumes wird ein spezielles Attribut ausgewertet, die Kanten stellen dabei mögliche Werte des Attributs dar. Bei der Klassifikation einer Instanz wird der Baum von der Wurzel beginnend durchlaufen bis ein Blatt erreicht wird. Die Blätter des Baumes geben dann das Klassenattribut der Instanz an. In Abbildung 5 ist beispielhaft ein Entscheidungsbaum zur Klassifikation von Notenhandschriften dargestellt. Die Attribute, die in

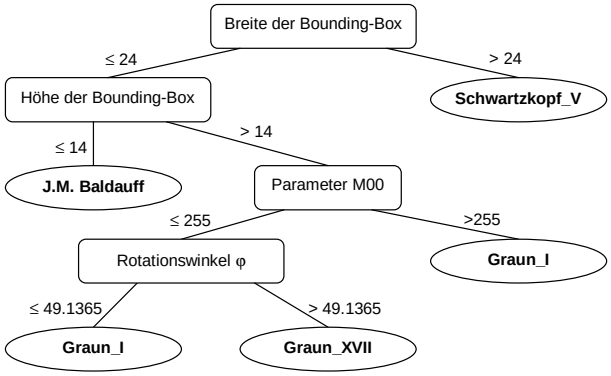

Abbildung 5: Entscheidungsbaum vom Typ J48 nach [Urb05]

den Knoten ausgewertet werden, sind die extrahierten Merkmale der eingescannten Notenblätter (vgl. Tabelle 1).
In [HK01] ist der grundlegende Algorithmus zur Erstellung eines Entscheidungsbaumes nachzulesen. Allerdings weist dieser Algorithmus eine Vielzahl an Freiheitsgraden auf. Wenn beispielsweise das Hinzufügen eines neuen Knotens erforderlich wird, so muss entschieden werden, welches Attribut an diesem Knoten ausgewertet wird. Für die Auswahl des günstigsten Attributes gibt es verschiedene Ansätze. Ebenso ist die Anzahl möglicher Kindknoten eines Knoten im allgemeinen Fall nicht festgelegt. Beim „binären Entscheidungsbaum“ hingegen ist diese Anzahl auf genau 2 festgelegt. Entsprechend gibt es eine Vielzahl konkreter Implementierungen von Entscheidungsbäumen, welche für ihren jeweiligen Anwendungsfall optimiert wurden.

Die Clusteranalyse wird zur Erkennung von Gruppen (Klassen, Cluster[7]) einander ähnlicher Objekte in einer gegebenen Menge von Objekten genutzt. Ziel des Clustering ist die Klassen so zu bestimmen, dass die Ähnlichkeit der Objekte innerhalb einer Klasse maximal und die Ähnlichkeit von Objekten unterschiedlicher Klassen minimal wird. Voraussetzung für derartige Verfahren ist somit das Vorhandensein eines Ähnlichkeitsmaßes für die einzelnen zu analysierenden Instanzen.
Im Gegensatz zur Klassifikation sind bei der Clusteranalyse die einzelnen Klassen nicht durch ein spezielles Attribut der Trainingmenge vorgegeben, sondern die Cluster werden durch das Verfahren automatisch bestimmt. Man spricht deshalb bei der Clusteranalyse auch vom unüberwachten Lernen (*unsupervised learing*) und bei der Klassifikation vom überwachten Lernen (*supervised learning*).
Die Verfahren zur Cluster-Analyse können anhand der verwendeten Algo-

[7]engl. cluster = Haufen, Gruppe, Büschel, Traube
Auf eine Übersetzung des Begriffs wird verzichtet. Unter einem Cluster wird im hier verwendeten Sinn eine Anhäufung, eine Gruppe von Instanzen verstanden werden.

rithmen u.a. in partionierende, hierarchische, dichtebasierte, modellbasierte oder grid-basierte Methoden eingeteilt werden. Für eine ausführliche Beschreibung der einzelnen Verfahren wird auf die Literatur verwiesen: [HK01], [Alp04] und [WF00].

2.2.2 Standards und Austauschformate

Bevor in den folgenden Abschnitten konkrete Data-Mining-Werkzeuge vorgestellt werden, sollen hier vorab die durch diese Programme genutzten Dateiformate ARFF und PMML kurz charakterisiert werden. Ebenso wird eine kurze Einführung in den Standard SQL/MM-Part 6: Data Mining gegeben, welcher u.a. durch den IBM DB2 Intelligent Miner for Data umgesetzt wird.

Das Attribute-Relation File Format (ARFF)[8] ist ein Dateiformat zur Beschreibung von Instanzen mit einer Menge von Attributen. Es wurde an der Universität von Waikato für das WEKA-Projekt (vgl. Abschnitt 2.2.4) entwickelt. ARFF-Dateien bestehen jeweils aus einem Header gefolgt von einem Datenbereich. Der Header enthält zum einen den Namen der gespeicherten Relation und zum anderen die Liste der Attribute. Es können sowohl numerische oder nominale Werte, als auch Zeichenketten oder Datumsangaben als Attribute gespeichert werden. Der Header einer ARFF-Datei könnte beispielsweise wie folgt aussehen:

```
% Kommentar
@relation NameDerRelation
@attribute NumerischesAttribut numeric
@attribute Aufzählung {element1, element2, element3}
@attribute Zeichenkette string
```

Im anschließenden Datenbereich, der durch das Symbol `@data` eingeleitet wird, werden die Attribute der einzelnen Instanzen durch Kommata getrennt dargestellt. Fehlende Werte werden durch ein „?“ symbolisiert:

```
@data
2.4, element3, "Zeile 1"
?, element1, "Zeile 2"
1.5, ?, "Zeile 3"
```

Zusammenfassend kann festgehalten werden, dass das ARFF-Format zur Speicherung „flacher“ Datenstrukturen in ASCII-Dateien genutzt werden kann.

[8] vgl. http://www.cs.waikato.ac.nz/~ml/weka/arff.html

Das Predictive Model Markup Language (PMML) ist ein XML-basiertes Dokumentenformat zur Beschreibung von DM-Modellen. PMML wurde 1996 an der University of Illinois, Chicago entwickelt und wird seit 1998 von der Data Mining Group (DMG) weitergeführt. Die Syntax der PMML wird seit Version 2.1 (aktuell: Version 3.0) durch ein XML Schema definiert. Die allgemeine Struktur eines PMML-Dokumentes sieht folgendermaßen aus:

```
<?xml version="1.0"?>
<PMML version="3.0"
xmlns="http://www.dmg.org/PMML-3_0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<Header copyright="Example.com"/>
<DataDictionary> ... </DataDictionary>

... a model ...

</PMML>
```

Dem Wurzelelement folgen dabei ein *Header*, optional ein *MiningBuildTask*, das *Data Dictionary*, optional ein *Transformation Dictionary*, die Beschreibung eines oder mehrerer Modelle und schließlich optional eine oder mehrere *Extensions*.

Das *Header*-Element enthält u.a. Copyright-Informationen und ggf. eine Beschreibung des Modells und einen Hinweis auf die Anwendung, die das Modell erzeugt hat, sowie eine Versionsnummer und einen Zeitstempel.
Das Element *MiningBuildTask* dient der Beschreibung der Konfiguration des Trainingslaufes, welcher die Modellinstanz erzeugte. Diese Information ist für die Verwendung des Modells nicht zwingend nötig, kann aber für die Pflege oder Veranschaulichung hilfreich sein. Die genaue Struktur des Inhalts dieses Elementes ist in PMML 3.0 nicht genauer definiert.
Das *Data Dictionary* enthält die Definition der Felder, die in den beschriebenen Modellen genutzt werden. Es werden sowohl der Typ als auch der Gültigkeitsbereich der einzelnen Felder festgelegt. Im *Transformation Dictionary* können anschließend verschiedene Umrechnungen der Daten für die Anwendung in den spezifischen Modellen definiert werden, beispielsweise die Abbildung kontinuierlicher in diskrete Werte.
Danach erfolgt die eigentliche Beschreibung des DM-Modells. Folgende Arten von Modellen werden u.a. aktuell unterstützt: Assoziationsregeln, Clustering-Modelle, Entscheidungsbäume, Kombinationen einfacher Modelle, Naïve Bayes Modelle, Neuronale Netze und Regressionsmodelle.

Eine ausführliche Beschreibung der Syntax sowie Beispiele konkreter Modelle sind auf der Webseite der DMG[9] zu finden. Für eine ausführliche Evalu-

[9] http://www.dmg.org/pmml-v3-0.html

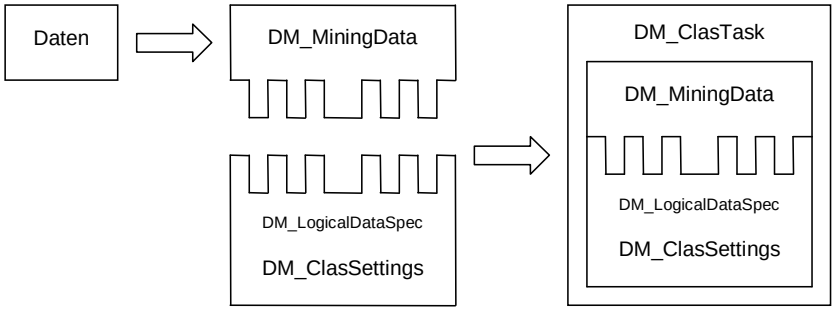

Abbildung 6: UDT's für die Vorbereitung des DM, nach [ISO01]

ierung und den Vergleich von PMML mit anderen Standards sei auf [Thi02] verwiesen.

Der Standard SQL/MM-Part 6: Data Mining [ISO01] ist ebenso wie der in Abschnitt 2.1.2 beschriebene Teil 5: Still Image ein Teil des SQL/MM-Pakets. Analog zu dem bereits vorgestellten Teil 5 erfolgt in den ersten Kapiteln des Standards zunächst eine Einordnung der Norm sowie die Definition verschiedener Begriffe. Anschließend werden in Kapitel 4 die unterstützten DM-Konzepte, der Ablauf des DM-Prozesses und die Abbildung dieses Prozesses auf konkrete UDT's beschrieben.
Folgende DM-Techniken sind vorgesehen: Assoziationsregeln und Clustering als beschreibende sowie die Klassifikation und Regression als vorhersagende Konzepte. Der DM-Prozess wird dabei wie bereits in Abschnitt 2.2.1 beschrieben in eine Trainings-, eine Test- und eine Anwendungsphase unterteilt (vgl. Abbildung 4). Die nachfolgend beschriebenen UDT's modellieren diesen Prozess.

In einem ersten Schritt werden alle für einen DM-Trainingslauf erforderlichen Informationen zusammengestellt. Hierzu werden die allgemeinen Einstellungen des DM-Prozesses mit einer Menge von Trainingsinstanzen zu einem Datentyp zusammengeführt. In Abbildung 6 ist dieser Schritt am Beispiel einer Klassifikation dargestellt. Die Einstellungen (*Settings*) auf der einen Seite umfassen dabei u.a. eine Beschreibung der einzelnen Eingabefelder für die Trainings-, Test- und Anwendungsphase (*logical data specification*). Wesentlich ist dabei die für das DM erforderliche Typisierung der Felder, beispielsweise als nominale oder ordinale Werte. Der Datentyp `DM_MiningData` auf der anderen Seite beinhaltet keine realen Daten, sondern lediglich eine Referenz auf die entsprechenden Tabellen oder Sichten. Auf diese Weise können Methoden des *Preprocessing*, wie z.B. eine Transformation oder Auswahl der Daten, integriert werden.
Auf der Basis dieser abstrakten Zusammenstellung aller wesentlichen Informationen über das Modell kann die Berechnung des Modells selbst (Trai-

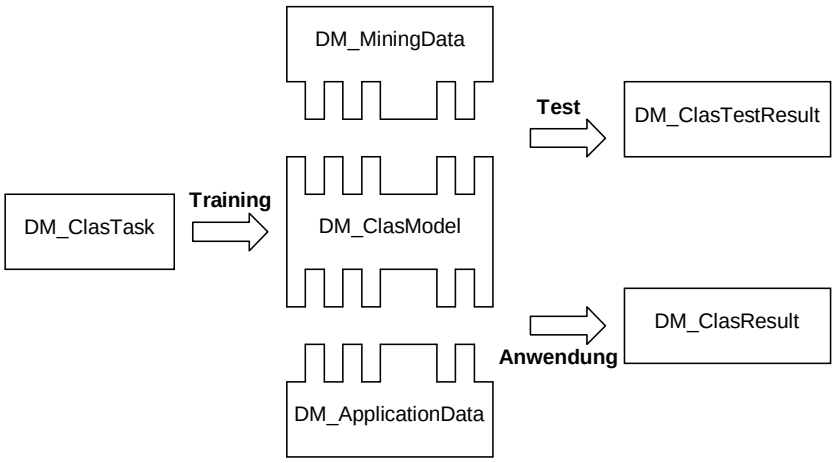

Abbildung 7: UDT's für die einzelnen Phasen des DM, nach [ISO01]

ningsphase) sehr einfach regelmäßig und automatisiert neu durchgeführt werden.

Das so erstellte konkrete DM-Modell stellt sowohl Methoden zum Testen als auch für die Anwendung des Modells zur Verfügung. In Abbildung 7 ist eine Einordnung der einzelnen UDT's in den Kontext der DM-Phasen für das Beispiel einer Klassifikation dargestellt.
Das Ergebnis eines Testlaufes unterscheidet sich dabei von dem einer Anwendung darin, dass hier das Modell auf eine Menge von Instanzen (Testmenge) angewendet wird. Außerdem kann in diesem Fall das berechnete mit dem tatsächlich vorhandenen, also erwarteten Ergebnis verglichen werden. Die Genauigkeit (*accuracy*) des Modells kann so bestimmt werden. Die Spezifikation der Testdaten erfolgt analog zu der der Trainingsdaten durch die abstrakte Beschreibung mittels des Datentyps `DM_MiningData`.
Für die Anwendung des Modells auf eine neue Instanz steht der Datentyp `DM_ApplicationData` zur Verfügung, welcher die konkreten Werte des Zustands kapselt. Für das Ergebnis der Anwendung des Modells ist ebenfalls ein dedizierter Datentyp definiert, da ein DM-Modell neben atomaren auch komplexe Typen zurückliefern kann.

In den Kapiteln 5 bis 9 des Standards werden schließlich die genauen Schnittstellen in Form der UDT's und Methoden für die einzelnen, konkreten Modelle definiert.

2.2.3 IBM Intelligent Miner

Mit der *IBM DB2 Intelligent Miner* Produkt-Familie besteht die Möglichkeit, DM-Techniken direkt in das Datenbanksystem zu integrieren. Der *DB2 Intelligent Miner* (IM) unterstützt dabei sowohl den Standard SQL/MM

Part 6: Data Mining als auch das Datenformat PMML, welche im vorigen Abschnitt vorgestellt wurden.
Wie der Standard SQL/MM unterscheidet auch der IM zwischen der Erstellung des Modells in einem ersten Schritt und der Nutzung des Modells in einem zweiten. Hierzu dienen die Tools *IM Modeling* und *IM for Scoring*. Darüber hinaus werden der *IM for Visualization* für die grafische Darstellung und der *IM for Data* für spezielle Anwendungen angeboten.
Entsprechend dem Standard SQL/MM werden die darin beschriebenen Techniken (Assoziationsregeln, Clustering, Klassifikation und Regression) unterstützt. Der IM for Data stellt noch einige Konzepte zur Verfügung.

Das *IM Modelling* Modul wird für die Erstellung des DM-Modells genutzt. Für die Vorbereitung der Daten werden Hilfsmittel zur Datenauswahl und zur Verknüpfung von Daten aus mehreren Tabellen oder Sichten sowie für die Filterung der Daten angeboten.
Das DM-Modell schließlich wird im PMML-Format dargestellt. Es kann in diesem Format entweder exportiert oder als XML-Objekt (*Character Large Object, CLOB*) in der Datenbank gespeichert werden.

Das *IM for Scoring* Modul basiert auf einem solchen, im PMML-Format gespeicherten DM-Modell und ermöglicht den Zugriff und die Nutzung dieses Modells mit SQL-Befehlen. Das Modell kann entweder aus einer externen Datei oder aus der Datenbank eingelesen werden.

Mit Hilfe des *IM for Visualization* schließlich können die Ergebnisse einer DM-Analyse in einem Java-basierten Browser grafisch dargestellt werden. Eingabeparameter auch dieses Moduls ist ein DM-Modell im PMML-Format.

2.2.4 WEKA

Das „Waikato Environment for Knowledge Analysis“ (WEKA) ist eine Hilfsmittel bei der Abbildung der einzelnen Phasen des KDD-Prozesses. Es wird an der „University of Waikato“ (Neuseeland) entwickelt und stellt Methoden und Werkzeuge vom *Preprocessing*, über das Data Mining bis zur grafischen Darstellung der Ergebnisse zur Verfügung.
Das Projekt basiert auf dem Gedanken, dass es ein universelles DM-Programm für alle Anwendungsszenarien nicht geben kann. Aus diesem Grund wurde eine Sammlung aktueller DM- und *Preprocessing*-Algorithmen in einem Framework zusammengestellt, in dem der Zugriff auf die einzelnen Methoden der unterschiedlichen Algorithmen durch ein gemeinsames Interface vereinheitlicht ist. So können vorhandene Algorithmen und Techniken in einer flexiblen Art und Weise für die Anwendung auf neue Probleme einfach zusammengestellt werden.

Das Standard-Eingabeformat für Daten ist das ARFF Format, welches in Abschnitt 2.2.2 bereits vorgestellt wurde. Zur Vorbereitung dieser Daten stehen eine Vielzahl sogenannter *Filter* zur Verfügung, beispielsweise für die Auswahl bestimmter Attribute oder die Diskretisierung bzw. Normalisierung numerischer Werte.
Weiterhin sind Methoden für eine Vielzahl an DM-Algorithmen, von Assoziationsregeln, über das Clustering bis zu Klassifikation und Regression, vorhanden. Die Beschreibung des Konzeptes kann u.a. in Form von Entscheidungsbäumen, Klassifikations- und Assoziationsregeln, Regeln mit Ausnahmen und mit Relationen oder als Cluster erfolgen. Eine grafische Darstellung dieser Ergebnisse ist ebenso wie eine Visualisierung der Eingabedaten möglich.

Die Nutzung des WEKA-Tools ist auf vielfältige Weise möglich. Zunächst können hier die drei verschiedenen grafische Benutzungsoberflächen (*Graphical User Interface*, GUI) genannt werden. Dabei handelt es sich um den *Explorer*, das *Knowledge Flow Interface* und das *Experimenter*. Der *Explorer* ist die einfachste Methode zur Nutzung von WEKA. Eine ausführliche Beschreibung der Funktionalität des Explorers ist unter [WF00] zu finden. Das *Knowledge Flow Interface* ist insbesondere für die Verarbeitung großer Datenströme geeignet, da es im Gegensatz zum Explorer nicht alle Anwendungsdaten gleichzeitig im Hauptspeicher verwaltet. Das dritte GUI ist das *Experimenter*, welches bei Klassifikations- und Regressionsprobleme für die Bestimmung der optimalen Methoden und Parameter genutzt werden kann. Eine vierte, allerdings nicht-grafische Benutzungsoberfläche ist die Kommandozeile, über die mittels spezieller Befehle der Zugriff auf die Funktionalität des WEKA-Systems möglich ist.

Das gesamte Framework ist in der Programmiersprache Java geschrieben und wird unter der GNU General Public License (GPL) verbreitet. Entsprechend ist der Quellcode des gesamten Systems als sogenannte *Open-Source-Software* verfügbar. Unter Beachtung der Lizenzbestimmungen ist damit sowohl eine Erweiterung des Frameworks, als auch das Einbinden der Software in eigene Projekte möglich.

2.2.5 Bewertung

Für die automatische Schreiberklassifikation in historischen Notenhandschriften können DM-Algorithmen eingesetzt werden. Die in dieser Arbeit untersuchten Programme IM und WEKA stellen beide entsprechende Analyseverfahren zur Verfügung und sind somit grundsätzlich für die Anwendung geeignet.

Im folgenden Kapitel 3 wird die vorhandene Implementierung des Projektpartners IGD vorgestellt, die sowohl die Extraktion der Feature-Vektoren als

auch die Schreibererkennung mittels DM-Techniken umsetzt. Während der Erstellung dieser Implementierung wurden bereits verschiedene Klassifikationsverfahren und Parameter verglichen. Die Beschreibung des Modells in Form eines Entscheidungsbaumes des Typs LMT mit entsprechenden Parametern stellte sich dabei als optimale Lösung dar. Dieser spezielle Typ wird jedoch nur durch das WEKA-Framework, nicht durch den IM unterstützt.

Sowohl der Standard SQL/MM Part 6 und der IM als auch das WEKA-Tool erfordern die Beschreibung einer DM-Instanz durch tupelartige Datenstrukturen (eine Zeile einer Tabelle bzw. ARFF). Für die Klassifikation einer Notenhandschrift werden jedoch viele einzelne Tupel aus verschiedenen Relationen verwendet, die jeweils einen Notenkopf, einen Notenhals etc. beschreiben. Diese Merkmale werden einzeln klassifiziert und anschließend werden diese Teilergebnisse zu einem Gesamtergebnis der Klassifikation zusammengefasst. Somit werden neben Techniken zur Klassifikation auch solche Algorithmen benötigt, die die Teilergebnisse bewerten und zusammenfassen.
Durch die Möglichkeit, individuelle Anwendungen auf der Basis des WEKA-Frameworks zu erstellen, kann der gesamte DM-Prozess durch eine auf dem WEKA-Tool basierende Implementierung einfacher umgesetzt werden.

Aufgrund der oben genannten Aspekte wird die Integration einer eigenen, speziellen DM-Implementierung auf Grundlage des WEKA-Frameworks in die Datenbankumgebung angestrebt.

3 Analyse der vorhandenen Implementierung

Für die automatische Schreiberidentifikation in historischen Notenhandschriften wurde durch das IGD ein umfangreiches Java-Framework entwickelt. Dieses stellt die Funktionalität für die Extraktion der Merkmale mittels Bildverarbeitungsalgorithmen und die Klassifikation der Schreiber mittels Data Mining-Techniken zur Verfügung.
Der prinzipielle Arbeitsablauf dieser Implementierung ist in Abbildung 8 dargestellt. Die Datenbank wird bei dieser Umsetzung lediglich als Speichermedium für die einzelnen Merkmale genutzt. Sowohl die Extraktion der Merkmale als auch die Klassifikation erfolgen durch separate, externe Programme.

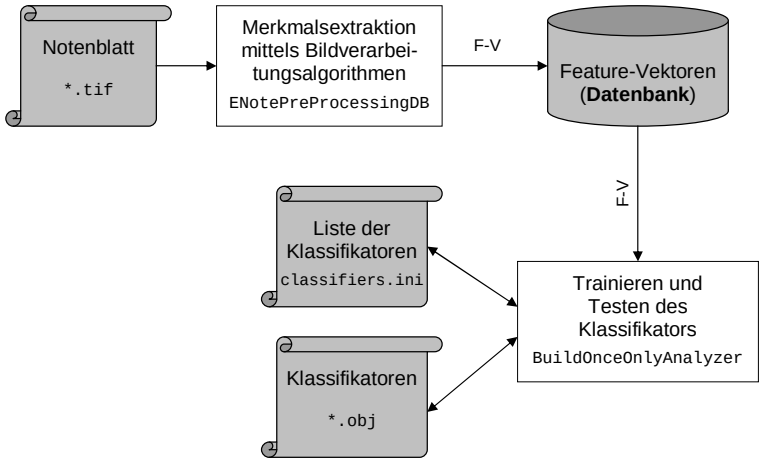

Abbildung 8: Arbeitsablauf der vorhandenen Implementierung

Ziel dieser Arbeit ist es, diese Funktionalität in eine Datenbankumgebung zu portieren, um dadurch einen einfacheren und effizienteren Zugriff auf dieses Werkzeug anzubieten. Um das hierfür erforderliche Konzept erarbeiten zu können, ist zunächst eine Analyse der vorhandenen Implementierung erforderlich. Dabei muss vor allem herausgearbeitet werden, welche Änderungen an dem vorhandenen Frameworks notwendig sind und welche Schnittstellen zwischen den einzelnen Arbeitsschritten definiert werden können.

Im Abschnitt 3.1 wird zunächst die Implementierung der Bildverarbeitungsalgorithmen, welche die Merkmale eines eingescannten Notenblattes extrahiert, untersucht. Danach wird in Abschnitt 3.2 die Umsetzung der Klassifikation basierend auf den Feature-Vektoren beschrieben.

3.1 Merkmalsextraktion mittels Bildverarbeitungsalgorithmen

Die historischen Notenhandschriften wurden eingescannt und stehen als Dateien im TIFF-Format zur Verfügung [Urb05]. Für die Analyse dieser sogenannten Digitalisate wurden Algorithmen zur Bildanalyse in der Programmiersprache *Java* implementiert. Die einzelnen Phasen der Analyse werden im folgenden Abschnitt betrachtet. Anschließend werden in Abschnitt 3.1.2 verschiedene Klassen, die die Analyse einzelner Digitalisate implementieren, betrachtet. In Abschnitt 3.1.3 wird danach die Darstellung der extrahierten Merkmale in der Datenbank vorgestellt. Abschließend werden in Abschnitt 3.2.6 die Ergebnisse zusammengefasst.

3.1.1 Ablauf der Analyse

Die Analyse der Digitalisate ist als ein mehrstufiger Algorithmus implementiert. Der erste Schritt ist die Bildvorverarbeitung, bei dem vor allem eine Trennung des Vorder- und Hintergrundes des Bildes erfolgt. Dafür ist u.a. das Herausfiltern von Flecken und Vergilbungen der historischen Handschriften erforderlich. Außerdem muss der optimale Winkel für eine Drehung des Digitalisates ermittelt werden, damit das Notensystem erkannt werden kann. Der zweite Schritt des Algorithmus ist die Erkennung einzelner Primitive. Hierzu zählen neben den Notenlinien die Taktstriche, die Notenhälse und -köpfe sowie der Notenschlüssel. Im nächsten Schritt, der Objekterkennung, werden die einzelnen Primitive zu Objekten zusammengesetzt.
Im vierten und letzten Schritt werden schließlich die erkannten Objekte pixelgenau vermessen und so charakterisiert. Die Objekte werden dabei durch Ellipsen-Parameter beschrieben. Die einzelnen Attribute und deren Bedeutung sind in Tabelle 1 zusammengestellt.

Attribut	**Beschreibung**
centerVertical, centerHorizontal	Koordinaten des Schwerpunktes
M00, M10, M01	Momente, Anzahl Pixel = Größe der Fläche
m02, m20, m11, m03, m30	zentrale Momente, normiert auf den Schwerpunkt
Ia, Ib	Haupt- und Nebenträgheitsachse
Ra, Rb	Radien der passenden Ellipse
phi	Orientierung der Ellipse in Grad
x, y, width, height	Position und Größe der Bounding Box

Tabelle 1: Momente und EllipseFitting-Parameter

Eine detailierte Beschreibung der einzelnen Schritte ist in [Urb05] und [SB05] zu finden.

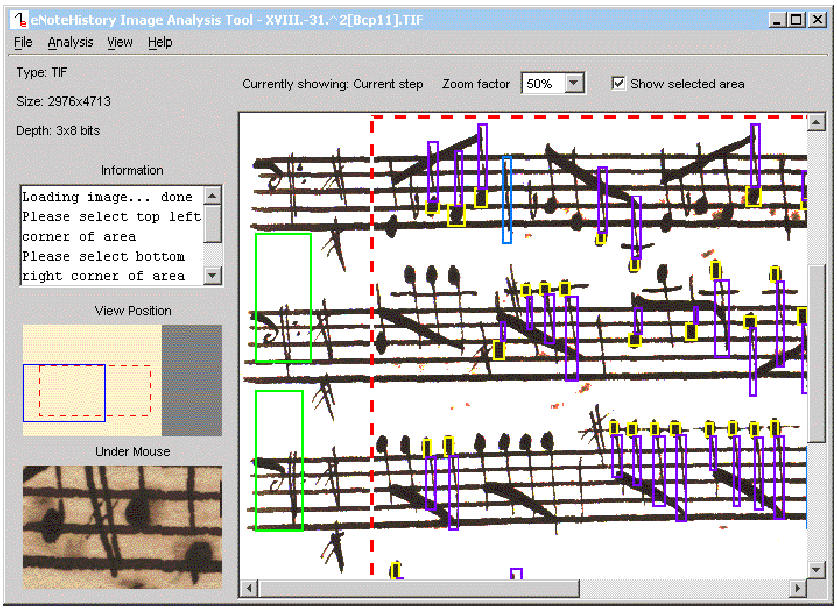

Abbildung 9: Weiterentwickelter GUI-Prototyp, aus [Urb05]

3.1.2 Analyse der Implementierung

Eine detailierte Betrachtung der Implementierung der Bildverarbeitungsalgorithmen an sich ist für diese Arbeit nicht erforderlich. Es werden nur die Klassen, die die Einstiegspunkte in den Extraktionsprozess und deren Ergebnis betrachtet. Die übrige Funktionalität des Frameworks wird als *Black Box* betrachtet.
Folgende Einstiegspunkte für die Extraktion der Merkmale sind vorhanden: eine grafische Benutzungsoberfläche (GUI), eine Applet-Variante der GUI und zwei nicht-grafische Varianten.

Die Klasse `ENoteImageAnalyzer` stellt eine grafische Oberfläche zur praktischen Erprobung der Bildverarbeitungsmethoden zur Verfügung. Wie in Abbildung 9 zu erkennen ist, werden neben dem Originalbild (unten links) auch der untersuchte Bereich des Bildes (*Region of interest*, ROI) und die *Bounding Boxes* der erkannten Objekte dargestellt.
Durch Aufruf der `main`-Methode der Klasse werden zunächst einige Operatoren der JAI-Bibliothek registriert und anschließend wird eine neue Instanz der Klasse `ENoteGui_Swing` erzeugt und initialisiert.

Die Klasse `ENoteImageAnalyzerApplet` stellt dieselbe Funktionalität wie die Klasse `ENoteImageAnalyzer` zur Verfügung, jedoch mit dem für ein *Applet* erforderlichen *Interface*. Die Implementierung beider Klasse ist ansonsten identisch.

Die Klasse `ENotePreProcessingOnly` dient dazu, für viele Bilder das *Preprocessing* durchzuführen und die vorbereiteten Bilder wieder zu speichern. Die extrahierten Merkmale werden nicht ausgegegeben oder gespeichert. Zum Ausführen dieser Klasse werden als Parameter ein oder mehrere Verzeichnisse angegeben, deren TIFF-Dateien bearbeitet werden sollen:

```
java -mx384M ENotePreProcessingOnly "BeispielVerzeichnis"
```

Das Argument `-mx384M` für die *Java Virtual Machine* soll das Auftreten eines `java.lang.OutOfMemoryError`s verhindern. Für die einzelnen Bilder wird die ROI abhängig von der Orientierung des Bildes und des innerhalb des Quellcodes gewählten horizontalen und vertikalen Randabstandes festgelegt. Mit diesen Einstellungen wird dann ein `ENoteImageFeatureVector`-Objekt erzeugt. Auf Grundlage dieses Objektes wird schließlich eine Instanz der Klasse `ENoteImageAnalysis` erzeugt und deren Methode `run` aufgerufen, welche das *Preprocessing* auslöst. Das Speichern der veränderten TIFF-Dateien erfolgt implizit durch diesen Methodenaufruf.

Die Klasse `ENotePreProcessingDB` führt wie `ENotePreProcessingOnly` das *Preprocessing* für viele Bilder durch. Allerdings werden hier die Bilddateien nicht durch die Angabe eines Verzeichnisses im Dateisystem, sondern durch den Zustand der Datenbank bestimmt. Die extrahierten Merkmale werden anschließend in der Datenbank gespeichert.
Zur Umsetzung dieser Funktionalität wird zunächst eine Verbindung zur Datenbank aufgebaut. Dann wird die Menge aller Bilder der Tabelle `IMAGES.PAGE_IMAGES` ermittelt, für die im Schema `IPFV` keine extrahierten Merkmale gespeichert sind. Diese Bilder werden dann jeweils in ein temporäres Verzeichnis kopiert und analysiert. Die Definition der ROI und die Extraktion der Merkmale erfolgt wie bei der Klasse `ENotePreProcessingOnly` (siehe oben). Die berechneten *Feature-Vektoren* werden schließlich im Datenbank-Schema `IPFV` gespeichert und die temporären Bilddateien gelöscht.

3.1.3 Datenmodell der Feature-Vektoren

Für die Verwaltung der eingescannten Notenblätter und ihrer Metainformationen wurde die Datenbank *enote* angelegt. Zur logischen Abgrenzung der einzelnen Tabellen wurden vier verschiedene Schemata verwendet. Eines dieser Schemata ist das Schema `IPFV`, welches zur Speicherung der extrahierten Merkmale genutzt wird. Das relationale Datenmodell ist in Abbildung 10 dargestellt. Eine ausführliche Beschreibung der einzelnen Tabellen und Attribute sowie die ER-Diagramme der Schemata sind in [DBIS05] zu finden.
Für die Erstellung eines Konzeptes zur automatischen Extraktion der

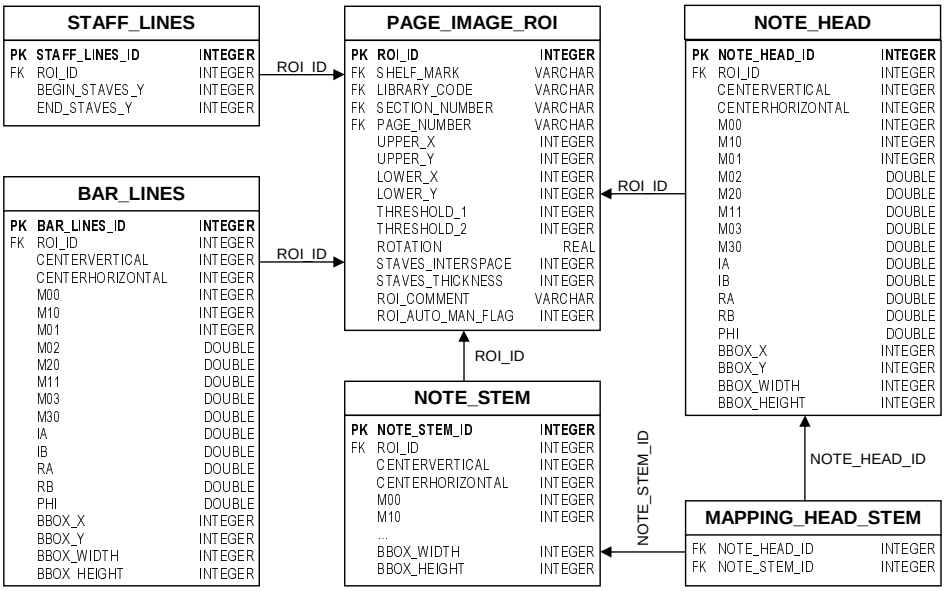

Abbildung 10: Relationales Datenmodell des Schemas IPFV

Feature-Vektoren sind v.a. folgende Aspekte von Bedeutung: für jedes gespeicherte Notenblatt können eine oder mehrere ROI's gespeichert werden. Die ROI beschreibt den Bereich des Notenblattes, der für die Analyse der Feature-Vektoren untersucht wird. Eine automatische Erkennung dieses Bereiches ist noch nicht implementiert. Aus diesem Grund wird zur Zeit die ROI in den Klassen `ENotePreProcessingOnly` und `ENotePreProcessingDB` allgemein durch ihren Abstand vom Seitenrand beschrieben. Deshalb wird typischerweise eine ROI je Notenblatt gespeichert. Zu jeder ROI können mehrere Notenlinien, Taktstriche, Notenköpfe und Notenhälse gespeichert werden. Die Zuordnung von zusammen gehörenden Notenköpfen und -hälsen erfolgt durch die Tabelle `MAPPING_HEAD_STEM`.

3.1.4 Zusammenfassung

Eine Integration der Klasse `ENotePreProcessingDB` in die Datenbankumgebung in Form einer *Stored Procedure* (SP) ist relativ einfach möglich. Hierfür wäre lediglich eine Änderung des Methodenaufrufes und der Aufbau der Verbindung zur Datenbank erforderlich. Außerdem wäre es sinnvoll, statt für alle noch nicht analysierten Notenblätter nur für ein ausgewähltes die Merkmale zu extrahieren. Parameter einer solchen SP wäre also der Primärschlüssel des zu analysierenden Notenblattes. Optional könnten auch die Breite des Randes zur Beschreibung der ROI als Parameter einer solchen Methode verwendet werden. Die SP würde wie die Klasse `ENotePreProcessingDB` die extrahierten Merkmale in den entsprechenden

Tabellen des Schemas `IPVF` speichern.

Eine Implementierung der Extraktion der Feature-Vektoren als UDF stellt sich hingegen als etwas aufwendiger dar. *User Defined Functions* können im Gegensatz zu *Stored Procedures* keine Daten direkt in Tabellen schreiben. Deshalb wird das Ergebnis der Funktion in Form eines Datentyps des Datenbanksystems zurückgegeben. Das Ergebnis dieses Funktionsaufrufes kann dann als Attributwert eines Tupels verwendet und durch SQL-Befehle in die Datenbank geschrieben werden.
Das verwendete Datenbanksystem IBM DB2 jedoch unterstützt keine tabellenwertigen Typen, so dass die Datenstruktur der Feature-Vektoren serialisiert und z.B. in Form eines BLOB zurückgegeben werden müsste.

3.2 Klassifikation mittels DM-Techniken

Das *Package* `ENoteDataMining` enthält eine Vielzahl einzelner Klassen, welche für das *Knowledge Discovery* erforderlich sind. Im Gegensatz zu der im vorigen Abschnitt beschriebenen Extraktion der *Feature-Vektoren* ist bei diesem Teil des Frameworks eine Betrachtung des Prozesses als „Black-Box“ nicht möglich. Um die Funktionalität der Klassifikation in die Datenbankumgebung portieren zu können, müssen mehrere Klassen modifiziert und ggf. der gesamte Arbeitsablauf neu organisiert werden.

Die Klasse `ENoteHistoryDataMiningRuntimeApplication` stellt derzeit den Einstiegspunkt in den DM-Prozess dar. Um die Implementierung dieser Klasse verstehen zu können, ist zuvor die Analyse der durch diese Klasse verwendeten Komponenten erforderlich. Im Rahmen der Untersuchung der einzelnen Klassen dieses *Packages* konnten diese den unterschiedlichen Phasen des KDD-Prozesses zugeordnet werden. Entsprechend wird in Abschnitt 3.2.2 zunächst das *Preprocessing*, in Abschnitt 3.2.3 der zentrale DM-Prozess und danach in Abschnitt 3.2.4 das *Postprocessing* vorgestellt, bevor in Abschnitt 3.2.5 schießlich der Gesamtprozess beschrieben werden kann.

Nicht verwendete Klassen: Einige Klassen des Frameworks werden im Rahmen des hier beschriebenen Prozesses nicht benötigt. Hierzu gehören die Klassen `J48Outputter`, `IterativeTreeBuilder`, `IterativeTreeBuilderTuple` und `ScribesMapper`, welche durch keine andere Klasse des Frameworks verwendet werden.
Die Klassen `EllipseFittingObjectAnalyzer`, `Experimeter` und `J48Analyzer` wurden während der Entwicklungsphase u.a. für die Bestimmung des optimalen Klassifikationsverfahren und dessen Parameter genutzt. Das *Interface* `Analyzable` beschreibt die Schnittstelle dieser drei Klassen und wird deshalb ebenso wie die Klasse `ResultFormatter`, welche nur durch den `EllipseFittingObjectAnalyzer` verwendet wird, nicht weiter benötigt.

Die Klasse `MinimumInstancesAvailablePerWriterComputer` wird zwar von der im Rahmen des *Preprocessing* (vgl. Abschnitt 3.2.2) beschriebenen Klasse `ArffTransformer` verwendet, aber nur in einer Methode, die durch den `EllipseFittingObjectAnalyzer` aufgerufen wird. Entsprechend kann auch auf eine detailierte Betrachtung dieser Klasse verzichtet werden.

Bezeichnungen, Konventionen: Bei der Bezeichnung von Variablen und Methoden im Quellcode wurden einige Namen verwendet, die evtl. einer Erklärung bedürfen. Mit *Signature* wird das Digitalisat eines Notenblattes bezeichnet. Der Begriff *Analyser* wird als Oberbegriff für ein Lernkonzept, wie z.B. einen Klassifikator, verwendet.

3.2.1 Klassen anderer Pakete

Die Klasse `ENoteEllipseFitting` aus dem Package `ENoteOperator` kapselt die in Tabelle 1 dargestellten *EllipseFitting*-Parameter. Der Konstruktor dieser Klasse erwartet ein Bild in binärer Darstellung als Eingabe und berechnet die zu diesem Bild gehörenden Attribute. Weitere Methoden dieser Klasse sind u.a. `getGenericArffHeader`, `getGenericArffHeaderEllipse` und `getGenericArffHeaderNoteStem`, die jeweils Teile des Headers einer ARFF-Datei (vgl. Abschnitt 2.2.2) liefern. Hierzu gehört v.a. die Deklaration der *EllipseFitting*-Attribute.

Die Klasse `ENotePropertyHandler` aus dem Package `ENoteGUI` wird von der Klasse `BuildOnceOnlyAnalyzer` benötigt und wird deshalb hier ebenfalls kurz beschrieben. Die Implementierung beruht auf der Java-Klasse `java.util.Properties` und ermöglicht das Laden und Speichern von Einstellungen (*Properties*[10]) in einer externen Datei. Die Einstellungen werden in der Form Schlüsselwort (*key*) → Eigenschaft (*property*) dargestellt. Das Laden oder Anlegen einer neuen Datei erfolgt durch den Konstruktor, das Speichern der aktuellen Einstellungen durch die Methode `saveProperties`. Darüber hinaus sind Methoden zum Lesen (`getProperty`) und Ändern (`setProperty`) einzelner Eigenschaften, sowie zum Auflisten aller Schlüsselwörter (`getKeys`) vorhanden.

3.2.2 Preprocessing

Im Vorfeld des eigentlichen DM sind verschiedene Schritte zur Auswahl und Vorbereitung der Daten erforderlich. Hierzu gehören zunächst die *InstancesGetter*-Implementierungen, welche das Laden der Feature-Vektoren aus dem Dateisystem oder aus einer Datenbank ermöglichen. Anschließend werden die geladenen Attribute normiert und schließlich über das `ARFF`-Format in Instanzen des WEKA-Tools transformiert.

[10] engl. Properties: Eigenschaften, Eigentum, Besitz

Die Klasse `ArffTransformer` stellt die Funktionalität zur Erstellung von ARFF-Dateien und von Instanzen für die Analyse zur Verfügung. Zum überwiegenden Teil werden diese Methoden jedoch von der Klasse `EllipseFittingObjectAnalyzer` verwendet, welche für die aktuelle Implementierung der Klassifikation nicht verwendet wird. Im aktuellen Framework wird lediglich die Methode `prepareHeader` genutzt. Diese erzeugt den *Header* der ARFF-Datei einer Instanz für ein bestimmtes Merkmal. Parameter dieser Methode sind eine Liste aller Schreiber und dieses Merkmal. In Abhängigkeit hiervon wird durch Nutzung der externen Klasse `ENoteEllipseFitting` (vgl. Abschnitt 3.2.1) der *Header* generiert.
Diese Methode wird durch die vorhandenen Implementierungen des *Interfaces* `InstancesGetter` verwendet, um die Methode `makeInstancesFromArff` der Klasse `InstancesMaker` nutzen zu können.

Die Klasse `InstancesMaker` ermöglicht die Erstellung von Instanzen der Klasse `weka.core.Instances`, welche die Grundlage der DM-Algorithmen sind. Hierzu wird die Methode `makeInstancesFromArff` verwendet, welche eine Zeichenkette im ARFF-Format als Eingabeparameter erwartet. Eine weitere Methode dieser Klasse ist `join`, mit der eine Menge von *Instances* zu einer vorhandenen Menge hinzugefügt werden können.

Das *Interface* `InstancesGetter` definiert eine allgemeine Schnittstelle für Datenquellen, die Instanzen für die Analyse liefern können. Folgender Ablauf der Methodenaufrufe ist während des Klassifikationsprozesses vorgesehen: Zuerst werden die Methoden `getAllScribesAvailable`, `getAllFeaturesAvailable` und `getAllSignaturesAvailable` aufgerufen, die jeweils ein *Array* aller verfügbaren Schreiber (*Scribe*), Merkmale (*Feature*) bzw. Notenblätter (*Signature*) zurückliefern. Für jedes Merkmal und jedes Notenblatt werden dann die folgenden Schritte bearbeitet:

- Um abzuschätzen, wieviele Instanzen je Schreiber für die Klassifikation eines bestimmten Notenblattes verwendet werden sollen, kann die Methode `getAmountOfInstancesPossibleForEveryScribe` genutzt werden. Diese Methode erwartet als Parameter eine Menge von Schreibern, ein spezielles Merkmal und ein bestimmtes Notenblatt (Test-Instanz). Zurückgegeben wird eine Abbildung der Form *Schreiber* $\rightarrow$ *Anzahl der vorhandenen ROI's* zu diesem Schreiber, für das angegebene Merkmal und unter Ausschluß der Test-Instanz. Die Abbildung wird in Form eines `ClassificationResult`-Objektes geliefert.

- Dann wird die Methode `getTrainingInstancesForScribes` aufgerufen, welche die Instanzen liefert, die für die Erstellung des Klassifikators genutzt werden. Parameter dieser Methode sind eine Menge von Schreibern, ein bestimmtes Merkmal, die Test-Instanz und die Anzahl

von Instanzen, die je Schreiber geliefert werden sollen. Das Ergebnis dieser Methode ist vom Typ `weka.core.Instances`.
Wie zuvor wird auch bei dieser Methode die Test-Instanz als mögliche Trainingsinstanz ausgeschlossen.

- Schließlich wird die Methode `getInstancesForSignature` aufgerufen, welche die Testinstanzen für den Klassifikator liefert. Parameter dieser Methode sind wieder eine Menge von Schreibern, ein bestimmtes Merkmal und eine Test-Instanz. Zurückgegeben werden die Daten des angegebenen Merkmals für die Test-Instanz, sofern sich der zugehörige Schreiber in der Menge der angegebenen Schreiber befindet. Der Typ des Rückgabewertes ist auch hier `weka.core.Instances`.

Für das *Postprocessing* bereits sind darüber hinaus zwei weitere Methoden deklariert. Die Methode `getScribeForSignature` liefert den Schreiber eines bereits klassifizierten Notenblattes. Die Methode `getImageSignaturesUsedInTrainingSetFor` liefert ein *Array* aller Notenblätter des angegebenen Schreibers, die für das Trainieren des Klassifikators verwendet wurden. Wird hierbei kein Schreiber angegeben (NULL), so werden alle für die Trainingsphase verwendeten Notenblätter zurückgegeben.

Zwei Implementierungen dieser Schnittstelle sind innerhalb des Frameworks vorhanden. Die Klasse `FileSystemInstancesGetter`[11] kann genutzt werden, um die Eingabedaten aus externen `ARFF`-Dateien einzulesen. Die Klasse `DBSystemInstancesGetter`[12] implementiert das *Interface* für das Einlesen von Instanzen aus einer Datenbank.
Auf Details der einzelnen Implementierungen sowie die genutzten Hilfsklassen soll hier nicht weiter eingegangen werden.

Die Klasse `Preprocessor` wird für die Vorbereitung der Instanzen genutzt. Eine Instanz dieser Klasse besitzt die beiden boolschen Attribute `limitAmountOfInstances` (Standardwert: *true*) und `randomizeInstances` (Standardwert: *false*). Zum Ändern und Lesen dieser Attributwerte sind entsprechende Getter- und Setter-Methoden vorhanden.
Zum Starten des eigentlichen *Preprocessing* wird die Methode `preProcess` aufgerufen. Diese Methode erwartet eine Menge von Instanzen und die Angabe eines Merkmals als Parameter und liefert eine Menge aufbereiteter Instanzen zurück. Im Rahmen dieser Aufbereitung wird zuerst durch Nutzung der Klasse `InstancesLimiter` (siehe unten) die Menge der Instanzen

[11] `FileSystemInstancesGetter` nutzt die Klasse `ImagesPerWriter`, welche die Merkmale aller Notenblätter eines Schreibers kapselt und die Klasse `EllipseFittingsPerImage`, die eine Menge von *Feature-Vektoren* verwaltet.

[12] Die Klasse `DBConnection` stellt eine Verbindung zur Datenbank her und stellt basierend darauf Methoden für Anfrage- und Updateoperationen bereit. Diese Klasse wird durch `DBSystemInstancesGetter` entsprechend genutzt.

verringert. Danach werden die Attribute der verbliebenen Instanzen relativiert[13]. Dieses geschieht durch die Definition entsprechender Filter aus dem WEKA-Framework.
Durch den Filter `AddExpression` wird dem vorhandenen Datensatz ein neues Attribut hinzugefügt. Der Berechnung des Attributwertes erfolgt durch Angabe einer mathematische Formel. Auf diese Weise werden die vorhandenen Attribute umgerechnet (relativiert). Anschließend werden die ursprünglich vorhandenen Attribute durch Nutzung des `Remove`-Filters aus den Datensätzen entfernt.

Die Klasse `InstancesLimiter` wird durch den `Preprocessor` verwendet, um die Anzahl der Instanzen je Schreiber einander anzugleichen. Dafür wird die Methode `limitAmountOfInstances` aufgerufen, welche zunächst prüft, ob das Klassenattribut vom Typ *nominal* ist und mehr als eine Klasse durch die Instanzmenge beschrieben wird. Ist das der Fall, wird die Gesamtmenge der Instanzen in einzelne Instanzmengen der unterschiedlichen Klassen zerlegt. Dann wird für jede dieser Klassen die Anzahl der Instanzen ermittelt. Anschließend wird die kleinste Zahl an Instanzen einer Menge ermittelt (n) und alle Mengen werden dann auf n Instanzen reduziert. Ist der Parameter `randomize` mit dem Wert *true* belegt, wird eine zufällige Auswahl der ursprünglichen Instanzmenge zurückgegeben, ansonsten verbleiben genau die ersten n Instanzen in der Menge.

3.2.3 Analyse

Die Erstellung des Klassifikators ist durch die Implementierung sogenannter *Analyzer* umgesetzt. Die entsprechenden Klassen sind zum überwiegenden Teil im *Package* `Analyzer` abgelegt und werden nachfolgend beschrieben.
Da für die Analyse eines Notenblattes eine Vielzahl einzelner Klassifikationen erforderlich sind, müssen die Teilergebnisse dieser Klassifikationen ebenfalls verwaltet und ausgewertet werden. Hierfür steht die Klasse *ClassificationResult* zur Verfügung, welche im Anschluß beschrieben wird.

Das Package `Analyzer` stellt konkrete Implementierungen für die Klassifikation zur Verfügung. In Abbildung 11 ist die Implementierungshierarchie der einzelnen Klassen dargestellt. Nachfolgend sollen die Attribute und Methoden dieser Klassen und der Klasse `AnalyzerFactory` kurz charakterisiert werden.
Neben diesen Klassen sind in dem Package spezielle Klassen für die Signalisierung bestimmter Ausnahmesituationen (*Exceptions*) implementiert, auf deren konkrete Bedeutung jedoch nicht weiter eingegangen wird.

[13] Auf eine Beschreibung der Umrechnung der einzelnen Attribute wird hier verzichtet. Die jeweiligen Formeln sind aus den in Quellcode definierten Filtern ersichtlich.

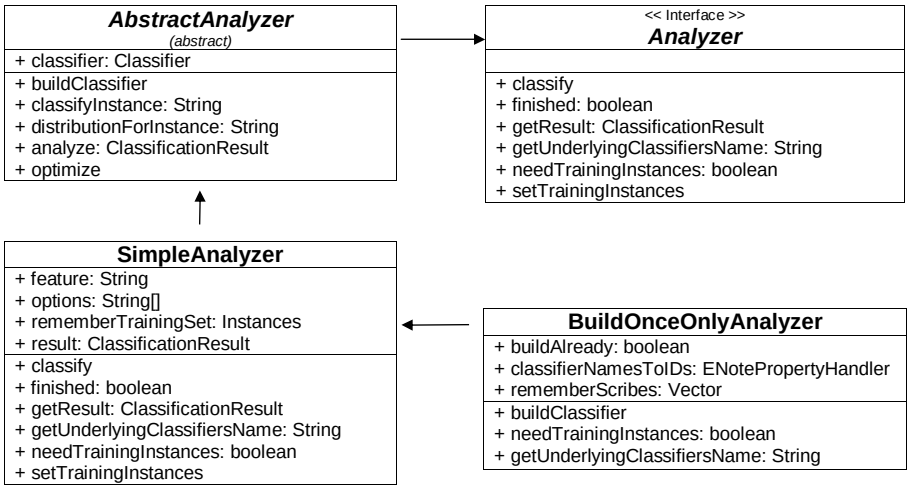

Abbildung 11: Hierarchie der Klassen im Package `Analyzer`

Die Schnittstelle `Analyzer` definiert die Methoden, die für die Nutzung eines *Analyzers* erforderlich sind. Folgender Ablauf der Methodenaufrufe ist dabei vorgesehen:
Zuerst wird die Methode `needTrainingInstances` aufgerufen. Parameter dieser Methode sind eine Menge von Schreibern und ein Merkmal. Zurückgegeben wird ein boolscher Wert, der aussagt, ob für die Klassifikation dieser Schreiber anhand des ausgewählten Merkmal möglich ist, oder ob der Klassifikator zunächst trainiert werden muss. Dementsprechend muss anschließend ggf. die Methode `setTrainingInstances` aufgerufen.
Danach kann die Methode `classify` aufgerufen werden, die das Laden oder die Erstellung eines Modells anstößt und dann die übergebenen Testinstanzen anhand des gewählten Merkmals klassifiziert. Rückgabewert dieser Methode ist ein `ClassificationResult`. Die Methode `finished` gibt Auskunft darüber, ob dieser Klassifikationsprozess bereits abgeschlossen und somit schon ein Ergebnis verfügbar ist.
Durch Aufruf der Methode `getResult` kann zu einem späteren Zeitpunkt erneut auf das Ergebnis der Klassifikation zurückgegriffen werden. Die Methode `getUnderlyingClassifiersName` liefert dem Nutzer den Namen des verwendeten Klassifikators.

Die abstrakte Klasse `AbstractAnalyzer` erweitert die Schnittstellenbeschreibung um konkrete Methodenimplementierungen und das Attribut `classifier`, welches eine Instanz des verwendeten Klassifikators referenziert. Die im *Interface* deklarierten Methoden werden hier jedoch noch nicht implementiert.
Die Methode `analyze` klassifiziert die übergebenen Testinstanzen anhand des ebenfalls übergebenen Klassifikators und des angegebenen Merkmals.

Das Ergebnis wird in Form eines `ClassificationResult` zurückgeliefert. Für die Umsetzung dieser Funktionalität werden u.a. die Methoden `classifyInstance` und `distributionForInstance` verwendet.
Weiterhin ist die Methode `buildClassifier` implementiert. Diese erwartet als Parameter die Instanz einer konkrete Implementierung eines Klassifikators[14], die Parameter für diesen Klassifikator und eine Menge von Trainingsinstanzen. Damit wird dann die gleichnamige Methode des Klassifikators aufgerufen und deren Ergebnis im Attribut `classifier` gespeichert.
Außerdem ist eine Methode `optimize` implementiert, welche einen bereits vorhandenen Klassifikator optimieren soll. Diese Implementierung ist jedoch noch nicht ausreichend getestet und wird deshalb nicht verwendet.

Die Klasse `SimpleAnalyzer` erweitert die abstrakte Klasse `AbstractAnalyzer` und implementiert die Methoden des *Interfaces* `Analyzer`. Außerdem werden zusätzliche Attribute eingeführt: `feature` beschreibt das Merkmal, für das der Klassifikator gebaut wird, `options` speichert die Parameter für den Klassifikator, `rememberTrainingSet` die Menge der Trainingsinstanzen und `result` das Ergebnis der Klassifikation der Testinstanzen.
Zum Erzeugen einer Instanz der Klasse `SimpleAnalyzer` stehen drei verschiedene Konstruktoren zur Verfügung, welche als Parameter die Instanz einer konkreten Klassifikatorimplementierung und optional dazugehörige Klassifikationsparameter erwarten.

Die Klasse `BuildOnceOnlyAnalyzer` schließlich erweitert den `SimpleAnalyzer` um die Attribute `buildAlready`, `classifierNamesToIDs` und `rememberScribes`. Der Wert von `buildAlready` gibt an, ob der Klassifikator bereits konstruiert wurde oder noch erstellt werden muss.
Das Attribut `classifierNamesToIDs` ist vom Typ `ENotePropertyHandler` (vgl. Abschnitt 3.2.1) und dient dazu, auf bereits vorhandene Klassifikatoren zurückzugreifen. Hierzu wird zunächst zur Identifizierung jedes Klassifikators eine sogenannte *Id* aus dem betrachteten Merkmal, der Menge der berücksichtigten Schreiber und dem Typ des Klassifikators gebildet. Diese *Id* wird als Schlüssel für die Speicherung der Klassifikatoren (*Id* $\rightarrow$ Klassifikator) verwendet. Alle bereits vorhandenen und im Dateisystem gespeicherten Klassifikatoren werden auf diese Weise in der Datei `classifiers.ini` verwaltet. Über das Attribut `classifierNamesToIDs` wird der Zugriff auf diese Relation bereitgestellt.
Das Attribut `rememberScribes` dient dazu, die bei der Erstellung des

[14] Konkrete Implementierungen eines Klassifikators sind Klassen, die die abstrakte Klasse `weka.classifier.Classifier` erweitern. Hierzu gehören alle Klassifikations- und Regressions-Implementierungen des WEKA-Frameworks, wie z.B. die Klassen `DecisionTable`, `Id3`, `J48`, `LMT`, `MultiLayerPerceptron`, `NaiveBayesSimple` und `OrdinalClassClassifier`.

Klassifikators berücksichtigten Schreiber zu speichern. Dieser Wert wird, wie gerade beschrieben, beispielsweise für die Berechnung der *Id* benötigt.

Zum Erzeugen einer Instanz des `BuildOnceOnlyAnalyzer`s stehen dieselben Konstruktoren wie für den `SimpleAnalyzer` zur Verfügung. Zusätzlich zur Implementierung der Oberklasse wird hier noch der Wert des Attributs `classifierNamesToIDs` initialisiert. Hierfür wird die Methode `checkForClassifierFiles` aufgerufen.

Darüber hinaus werden in dieser Klasse einige Methodenimplementierungen der Oberklassen überschrieben. Neben der Methode `getUnderlyingClassifiersName`, in der lediglich der Name der Klasse gegen „BuildOnceOnlyAnalyzer“ ausgetauscht wird, betrifft das die Methoden `needTrainingInstances` und `buildClassifier`.
Zusätzlich zu der Implementierung in der Klasse `SimpleAnalyzer` wird hier in der Methode `needTrainingInstances` zuerst überprüft, ob der jeweils benötigte Klassifikator im Dateisystem schon vorhanden ist und somit nicht neu berechnet werden muss. Analog wird die Methode `buildClassifier` in dieser Klasse derart erweitert, dass ein neu erstellter Klassifikator auch im lokalen Dateisystem gespeichert wird und somit auch für zukünftige Berechnungen genutzt werden kann.

Ziel der Erweiterung des `SimpleAnalyzer`s zum `BuildOnceOnlyAnalyzer` ist es, eine mehrfache Erstellung desselben Klassifikator zu vermeiden. Statt dessen wird jeder Klassifikator nur einmal gebaut.

Die Klasse `DataMiningRuntimeClassificationThread` implementiert das *Interface* `Analyzer` und erweitert die Klasse `java.lang.Thread`. Mit Hilfe dieser Klasse kann eine andere Implementierung des *Interfaces* `Analyzer` in Form eines nebenläufigen Prozesses (*Threads*) ausgeführt werden. Dem Konstruktor wird dafür eine Instanz dieser Implementierung übergeben. Die Ausführung wird durch die Methode `run` gestartet.
Außerdem sind alle Methoden des *Interfaces* `Analyzer` vorhanden, die jeweils den Methodenaufruf an die gleichnamig Methode der konkreten Implementierung weiterreichen.

Die Klasse `AnalyzerFactory` verwaltet eine Menge von Instanzen der Klasse `SimpleAnalyzer` mit unterschiedlichen Klassifikatoren. Standardmäßig sind bereits die Entscheidungsbäume `J48`, `LMT` und `REPTree` sowie die Klassifikationsregeln `OneR`, `PART` und das Bayes-Netz `NaiveBayesUpdateable` verfügbar. Weitere Klassifikatoren können mit der Methode `addAnalyzer` hinzugefügt werden. Eine Liste der vorhandenen Instanzen wird von der Methode `getAvailableAnalyzerNames` geliefert, eine Instanz selbst durch die Methode `getAnalyzer`.

Die Klasse `ClassificationResult` wird für die Verwaltung der Ergebnisse einzelner Teilklassifikationen genutzt. Während des DM-Prozesses wird für sämtliche Teilinstanzen und Merkmale eine Klassifikation durchgeführt, die jeweils eine Klasseninstanz als Ergebnis liefert. Die Klasse `ClassificationResult` verwaltet eine Abbildung der Form *Klasseninstanz → Anzahl der Klassifikationen*. Zum Hinzufügen neuer Teilergebnisse sind drei verschiedene Implementierungen der Methode `add` vorhanden: zum Inkrementieren der Anzahl für eine Klasse um 1, zum Inkrementieren der Anzahl für eine Klasse um einen beliebigen Wert und zum Vereinigen zweier `ClassificationResult`-Objekte. Zum Vergleichen zweier `ClassificationResult`-Objekte kann die Methode `equals` genutzt werden. Außerdem sind Methoden vorhanden, um eine nach der Anzahl sortierte Liste aller Klasseninstanzen (`getClasses`) oder die zu einer Klasse gehörende Anzahl (`getAmountOfInstances`) zu erhalten.
Die Methode `isConclusive` liefert eine boolsche Aussage darüber, ob das Ergebnis der Klassifikation schlüssig ist. Grundlage dieser Aussage sind heuristische Verfahren. Durch Angabe einer *Conclusion Policy* kann das verwendete Verfahren ausgewählt werden. Bei einem dieser Verfahren wird das Ergebnis genau dann als schlüssig bezeichnet, wenn mindestens 66% der einzelnen Klassifikationen dieselbe Klasseninstanz als Ergebnis liefern. Bei einer anderen Variante muss die Anzahl der am häufigsten klassifizierte Instanz mindestens 1,75-mal so groß sein, wie die der zweithäufigsten.
Das Gesamtergebnis der Klassifikationen, also die am häufigsten klassifizierte Instanz, kann schließlich durch Aufruf der Methode `getFirstClass` erhalten werden.

Im vorhandenen Framework wird für jedes untersuchte Merkmal ein eigenes `ClassificationResult`-Objekt angelegt. Deren Ergebnisse werden im Zuge des *Postprocessing* zu einem Gesamtergebnis zusammengefasst.

3.2.4 Postprocessing

Die Schritte, die nach der Erstellung des Modells erfolgen, werden als *Postprocessing* (Nachbearbeitung) bezeichnet. Im Framework des IGD betrifft dieses zum einen das *Voting* und zum anderen die Erstellung von Statistiken über die Testergebnisse, anhand derer Aussagen über die Qualität des Modells ermöglicht werden.

Die Aufgabe des *Votings* ist es, aus mehreren Klassifikationsergebnissen den Klassenwert der untersuchten Instanz zu ermitteln. Die einzelnen Teilergebnisse können z.B. aus der Klassifikation mit unterschiedlichen Typen von Klassifikatoren, typischerweise jedoch vor allem aus der Untersuchung der verschiedener Merkmale resultieren.

Das *Interface* `Voting` definiert hierfür die Methode `vote`, die als Eingabeparameter ein *Array* von `ClassificationResult`'s erwartet und ein einzelnes `ClassificationResult`-Objekt zurückliefert. Dieses enthält genau eine Instanz, die den Schreiber der Notenhandschrift identifiziert. Falls der *Voting*-Prozess ergebnislos verlaufen ist, wird ein leeres `ClassificationResult`-Objekt zurückgegeben, falls ein Fehler aufgetreten ist wird `NULL` zurückgegeben.

Konkrete Implementierungen dieser Schnittstelle sind im *Package* `ENoteVoter` zu finden: der `MostAppearancesFirstVoter`, der `SimpleMajorityVoter` und der `UnanimousVoter`. Auf die Unterschiede der einzelnen *Voter* soll an dieser Stelle nicht genauer eingegangen werden.

Die Klasse `VotingFactory` stellt einen Container für die unterschiedlichen Voting-Implementierungen bereit. Standardmäßig werden die drei genannten *Voter* verwaltet, durch die Methode `addVoter` können weitere Implementierungen dem Container hinzugefügt werden. Die einzelnen *Voter* werden durch einen Namen identifiziert. Anhand dieses Namens kann die Instanz eines *Voters* durch Aufruf der Methode `getVoter` abgerufen werden.

Die Qualität des Modells kann anhand der Ergebnisse der Testphase bestimmt werden. Diese Statistiken werden durch die Klasse `DataMiningPostProcessing` aufbereitet. Für die Ausführung dieser Klasse werden zwei weitere Klassen benötigt, welche zunächst betrachtet werden sollen.

Die Klasse `DataMiningPostProcessingData` stellt einen Container für ein Tupel bestehend aus einer *Signatur* (entspricht einem Notenblatt), einem dazugehörigen Schreiber, dem betrachteten Merkmal und der Ergebnismenge des Klassifikators dar. Neben dem Konstruktor, der als Parameter diese vier Attribute erwartet, sind sogenannte Getter-Methoden für die Attribute implementiert.

Die Klasse `DataMiningPostProcessingTuple` verwaltet eine Liste ganzzahliger Werte (Integer). Diese werden als *Indizes* bezeichnet. Für die Arbeit mit dieser Liste werden eine Vielzahl an Methoden bereitgestellt, z.B. zum Hinzufügen eines neuen *Index* zur Liste oder zum Konkatenieren zweier Listen. Weiterhin gibt es Methoden zum Vergleichen zweier Listen (`equals`), zum Ermitteln der Anzahl an Indizes (`size`) oder des größten Indexwertes (`getBiggestIndex`). Das Vorhandensein eines konkreten *Index* in der Liste kann mit `contains` erfragt und die gesamte Liste in Form eines *Arrays* kann mittels `toIntegerArray` ausgegeben werden.

Die Klasse `DataMiningPostProcessing` stellt in Form der Methoden `printStats` und `printStatsMultiple` statistische Informationen über die Genauigkeit der Klassifikation (*accuracy*) zur Verfügung. Der Konstruktor dieser Klasse erwartet als Parameter die Instanz einer *Voting*-Implementierung. Zum Hinzufügen neuer Klassifikationsergebnisse wird die Methode `addResult` verwendet. Parameter dieser Methode sind die Signatur, der Schreiber, das untersuchte Merkmal und die Ergebnismenge der Klassifikation. Diese Informationen werden intern in einer Liste von `DataMiningPostProcessorData`-Tupeln gespeichert.

3.2.5 Gesamtprozess

In diesem Abschnitt soll der Gesamtprozess, also das Zusammenwirken der in den vorigen Abschnitten vorgestellten Komponenten beschrieben werden. Den Einstiegspunkt in diesen Prozess stellt die Klasse `ENoteHistoryDataMiningRuntimeApplication` dar. Grundlage des Data Mining-Prozesses selbst hingegen ist die Klasse `DataMiningRuntime`. Diese Klasse bildet dafür den im *Interface* `ScribesManager` beschriebenen Prozess ab. Aus diesem Grund soll nachfolgend zunächst dieses *Interface* und dessen Implementierung beschrieben werden.

Ziel der Verwendung eines *ScribesManagers* ist es, nicht alle Klasseninstanzen (Schreiber) in einem Schritt zu bearbeiten, sondern in mehreren. Dieses führt sowohl zu einer Verbesserung der Genauigkeit der Klassifikation als auch zu einer besseren *Performance* des Gesamtprozesses.

Die Schnittstelle `ScribesManager` beschreibt den dafür vorgesehenen Arbeitsablauf. Der *ScribesManager* wird durch Aufruf der Methode `setScribes`, mit der Menge aller Schreiber als Parameter, initialisiert. Damit können die einzelnen Durchläufe des Klassifikationsprozesses beginnen. Die Methode `finished` liefert dann eine Aussage darüber, ob das Ergebnis dieses Prozesses vorliegt. Im positiven Fall kann dieses Ergebnis durch Aufruf der Methode `getResult` abgerufen und das *Postprocessing* durchgeführt werden.
Durch Aufrufen der Methode `getNextScribesToBeUsedForTraining` erhält die Anwendung eine Teilmenge der ursprünglichen Schreiber und initiiert nun die Trainings- und anschließend die Testphase für genau diese Teilmenge. Das Ergebnis dieser Phasen wird mittels der Methode `setResultForScribes` im *ScribesManager* abgelegt. Diese Schritte werden für alle Teilmengen der Schreiber durchlaufen.

Die Klasse `SimpleScribesManager` implementiert diese Schnittstelle auf eine sehr einfache Art und Weise. Sie dient lediglich als

Container für eine Menge an Schreibern und ein dazugehöriges Klassifikationsergebnis. Dieses Ergebnis vom Typ *ClassificationResult* wird durch die Methode `setResultForScribes` gesetzt und durch die Methode `getResult` zurückgeliefert. Die Methode `finished` liefert *true*, wenn ein Ergebnis innerhalb des *ScribesManagers* vorhanden ist. Die Methode `getNextScribesToBeUsedForTraining` liefert keine echte Teilmenge, sondern die gesamte, mit `setScribes` übergebene Menge der Schreiber zurück. Diese Implementierung dient somit vielmehr einer formalen Umsetzung der Schnittstellenbeschreibung, als einer tatsächlichen Realisierung der Idee eines *ScribesManagers*.

Die Klasse `DataMiningRuntime` implementiert die für das DM erforderliche Funktionalität. Hierfür werden intern u.a. folgende Attribute benötigt: jeweils ein `InstancesGetter` für die Trainings- und die Testmenge, ein `classIndex` zur Beschreibung des Klassenattributes, das zu analysierende Merkmal (*Feature*), ein `PreProcessor`, eine Menge von `Analyzer`n und `ScribesManager`n sowie eine `Voting`-Instanz. Außerdem ist für jeden `Analyzer` ein `ClassificationResult` vorhanden.
Die `InstancesGetter`, der `PreProcessor`, die `Analyzer` und die `Voting`-*Policy* werden durch den Konstruktor angegeben. Für das Klassenattribut und das zu untersuchende Merkmal gibt es entsprechende Getter- und Setter-Methoden. Die `ScribesManager` werden ebenfalls durch eine spezielle Methode (`setScribesManager`) übergeben.
Nachdem auf diese Weise alle erforderlichen Parameter ausgewählt wurden, kann die Klassifikation einer Instanz der Testmenge mit der Methode `process` gestartet werden. Der Fortschritt dieses Prozesses kann mit der Methode `finished` überprüft werden. Liefert diese Methode den Wert *true* zurück, kann das Ergebnis der Klassifikation mit `getResult` abgerufen werden. Je nach dem gewählten *Level of Detail* wird dabei entweder die Menge der `ClassificationResult`-Objekte oder das Ergebnis des *Votings* auf dieser Menge zurückgegeben.

Beim Aufruf der Methode `process` werden zuerst mittels des `InstancesGetter`s die Daten des ausgewählten Merkmals für das zu untersuchende Notenblatt (*Signature*) aus der Testmenge geladen. Danach werden diese Daten mittels des *Preprocessors* vorbereitet. Für die einzelnen *Analyzer* wird anschließend jeweils ein eigener Klassifikations-*Thread* erzeugt und initialisiert. Falls die Methode `needTrainingInstances` des jeweiligen Klassifikators den Wert *true* liefert, muss der Klassifikator erst mit einer Trainingsmenge erstellt werden, bevor die eigentliche Klassifikation erfolgen kann.
Für die Erstellung der verwendeten Trainingsmenge wird zuerst die Anzahl an ROI's je Schreiber berechnet. Anschließend werden die Trainingsinstanzen geladen (`getTrainingsInstancesForScribes`) und vorbereitet. Die

Anzahl an Instanzen je Schreiber wird dabei auf die minimale Anzahl an ROI's eines Schreibers, maximal jedoch 250, beschränkt.
Sobald alle *Threads* mit der Klassifikation fertig sind, werden deren Ergebnisse zusammengetragen und im zugehörigen `ClassificationResult`-Objekt gespeichert. Der Arbeitsablauf der `process`-Methode variiert von der beschriebenen Art und Weise ein wenig, wenn `ScribesManager` erzeugt wurden. Für diesen Fall ist der im entspechenden *Interface* (siehe oben) beschriebene Ablauf implementiert.

Die Klasse `ENoteHistoryDataMiningRuntimeApplication` stellt derzeit den Einstiegspunkt in die automatische Analyse dar. Zuerst werden jeweils ein `InstancesGetter` für die Trainings- und für die Testmenge an Notenblättern erzeugt. Danach wird ein *Preprocessor* erzeugt und initialisiert (`LimitationStatus: true`, `RandomizationStatus: false`). Anschließend wird die Menge an *Analyzer*n erzeugt. Dieser wird jedoch nur eine Instanz des `BuildOnceOnlyAnalyzer`s hinzugefügt. Schließlich wird eine Instanz des `SimpleMajorityVoter`s als *Voting-Policy* erzeugt.
Mit diesen Parametern wird dann eine Instanz der Klasse `DataMiningRuntime` erzeugt und initialisiert. Zur Verwaltung und Auswertung der einzelnen Teilergebnisse der Testphase wird außerdem eine Instanz der Klasse `DataMiningPostProcessors` erstellt.
Nach Abschluss dieser Vorbereitungen kann die eigentliche Analyse beginnen. Für jedes Notenblatt und jedes Merkmal wird dabei die `process`-Methode der `DataMiningRuntime` durchlaufen. Deren Ergebnisse werden gespeichert und am Ende durch den `DataMiningPostProcessor` ausgewertet.

3.2.6 Zusammenfassung

Durch das analysierte Framework des IGD wird lediglich die Trainings- und Testphase implementiert. Die Anwendungsphase ist derzeit nicht umgesetzt. Eine Implementierung dieser Phase könnte jedoch analog zur Testphase erfolgen. Der Nachteil dabei ist allerdings, dass keine klare Trennung der einzelnen Phasen vorhanden ist. Dieses äußert sich vor allem darin, dass dann auch für die Klassifikation eines Notenblattes Trainingsinstanzen spezifiziert werden müssten. Erst die Klasse `BuildOnceOnlyAnalyzer`, auf die kein direkter Zugriff existiert, entscheidet dann, ob die Klassifikation mit einem bereits vorhandenen Modell erfolgen kann oder ob zunächst ein neues Modell auf Basis der Trainingsinstanzen erstellt werden muss. Die einzelnen Modelle werden ausschließlich durch die Klasse `BuildOnceOnlyAnalyzer` verwaltet. Ein expliziter Zugriff auf die einzelnen Modelle ist nicht möglich. Entsprechend ist es auch nicht möglich, für eine Menge an Schreibern mehrere Modelle zu erzeugen und für den jeweiligen Anwendungsfall zu entscheiden, welches Modell für die Klassifikation genutzt werden soll.

Teil II
Konzeption und Umsetzung

4 Konzept

In diesem Kapitel soll herausgearbeitet werden, in welcher Art und Weise eine Integration der automatischen Schreibererkennung in die Datenbankumgebung möglich ist. Dazu werden nachfolgend teilweise unterschiedliche Ansätze verfolgt und deren Eignung zur Umsetzung der gegebenen Aufgabenstellung untersucht.

4.1 Extraktion der Bildmerkmale

Wie in Kapitel 3.1 beschrieben, existiert eine Implementierung des Projektpartners IGD, welche mittels Bildverarbeitungstechniken die speziellen Merkmale der Notenblätter extrahiert.

Im folgenden Abschnitt 4.1.1 wird zunächst die Möglichkeit einer Weiterentwicklung des vorhandenen, relationalen Datenmodells für die automatische Extraktion der Merkmale der Notenhandschriften verfolgt. Dabei sollen die Vor- und Nachteile dieses Modells gegenüber einer objektrelationalen Modellierung herausgearbeitet werden. Nach Abschluss dieser Vorbetrachtungen kann dann die konkrete Formulierung des Konzeptes erfolgen.

4.1.1 Vorbetrachtungen

Der entscheidende Vorteil einer relationalen Speicherung der extrahierten Merkmale ist, dass das vorhandene Datenmodell des Schemas `IPFV` (vgl. Abbildung 10) beibehalten werden kann. Das Modell müsste lediglich um die Funktionalität zur Extraktion der Merkmale und einige aktive Komponenten erweitert werden.
Die Funktionalität des IGD-Frameworks kann durch Verwendung einer *Stored Procedure* integriert werden. Wie bereits in Abschnitt 3.2.6 beschrieben, kann eine solche Implementierung mit relativ geringem Aufwand erfolgen. Zur Automatisierung der Extraktion können *Trigger* verwendet werden.

Die erforderlichen aktiven Komponenten sind schematisch in Abbildung 12 dargestellt. Beim Einfügen eines neuen Bildes in die Datenbank startet ein *Trigger* (a) die Ausführung einer *Stored Procedure*, welche die Koordinaten der ROI für den allgemeinen Fall berechnet. Das Ergebnis dieser Berechnung wird dann als neue Zeile in die ROI-Tabelle eingetragen. Diese INSERT-Anweisung (d) löst dann wiederum die Ausführung einer zweiten *Stored Procedure* aus, die die Feature-Vektoren des Digitalisats extrahiert und in den verschiedenen Tabellen speichert.

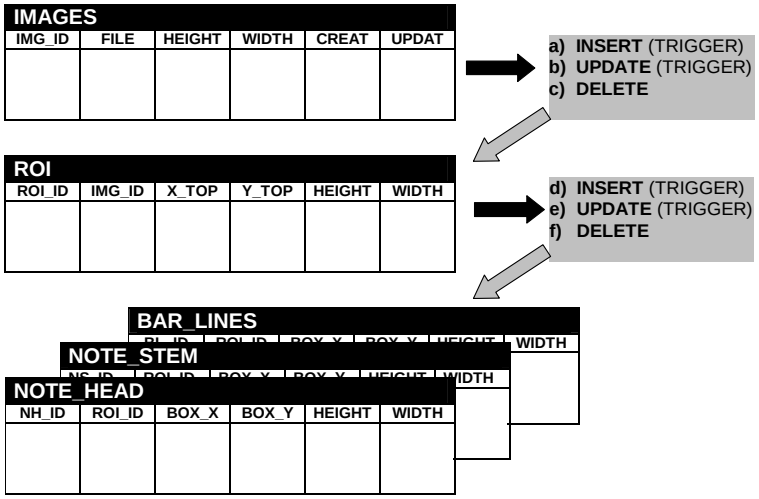

Abbildung 12: Aktive Komponenten eines relationalen Datenmodells

Allerdings kann so der zu analysierende Bereich des Bildes, die ROI, nicht direkt angegeben werden, sondern die Feature-Vektoren würden zuerst immer für die Standardparameter ermittelt werden. In einem zweiten Schritt könnten die Attribute der ROI durch einen entsprechenden UPDATE-Befehl geändert werden, so dass die Feature-Vektoren entsprechend aktualisiert würden (e). Allerdings wäre in diesem Fall eine zweimalige Extraktion der Feature-Vektoren erforderlich. Da die Extraktion der Merkmale sehr rechenaufwendig ist, sollte von einer solchen Modellierung abgesehen werden.
Um dieses zu umgehen, könnte in der Tabelle der Digitalisate zusätzlich ein Attribut (*Auto/Man-Flag*) aufgenommen werden, welches einen bestimmten Wert aufweisen muss, damit der *Trigger* die Ausführung der *Stored Procedure* startet.

Neben den bereits beschriebenen aktiven Komponenten für das Einfügen und Ändern von Datensätzen könnte das Löschen abhängiger Tupel durch die Definition der Randbedingung *kaskadierendes Löschen* realisiert werden.

Bewertung einer relationalen Modellierung: Neben den in den Vorbetrachtungen bereits genannten Vorteilen weist eine in der oben beschriebenen Art modellierte relationale Speicherung (mit aktiven Komponenten) auch einige Nachteile auf. Hierzu gehört neben teilweise umständlichen *Work-Arounds*, wie z.B. dem *Auto/Man-Flag*, vor allem die mangelhafte Berücksichtigung eines wichtigen Qualitätsmerkmals von Software: der Nutzerfreundlichkeit. Voraussetzung für die Benutzung einer so modellierten Datenbank ist eine detailierte Kenntnis über die interne Arbeitsweise des Systems. Eine intuitive Bedienung, z.B. über spezielle Methoden ist nicht möglich. Statt dessen muss der Nutzer wissen, welche Attribute der einzelnen

Tabellen für die Steuerung des Systems erforderlich sind (z.B. die Koordinaten der ROI oder das *Auto/Man-Flag*) und welche Ergebnisse der automatischen Extraktion der Merkmale sind (z.B. die Attribute `THRESHOLD_1` und `ROTATION`).

Durch die objektrelationalen Möglichkeiten des Standards SQL/99 besteht die Möglichkeit, das Datenbanksystem durch die Definition eigener Datentypen mit dedizierter Funktionalität konzeptuell zu erweitern. Diese benutzerdefinierten Datentypen (UDT) können zur Speicherung komplexer Objekte verwendet werden.
Durch die Modellierung eines solchen Datentyps können die oben genannten Nachteile einer relationalen Speicherung behoben werden. Der Nachteil der Standard- oder Nicht-Standard-ROI beim Einfügen neuer Bilder kann durch die Deklaration unterschiedlicher *Konstruktoren* des Datentyps behoben werden. Wird der Konstruktor ohne Parameter aufgerufen, wird die Standard-ROI berechnet; soll eine andere ROI verwendet werden, wird deren Beschreibung dem Konstruktor als Parametern übergeben.
Auf die Attribute des Datentyps wird durch entsprechende Getter- und Setter-Methoden zugegriffen. Für Attribute, die nicht durch den Nutzer verändert werden sollen, wird keine Setter-Methode definiert.

Ein weiterer Vorteil dieser Modellierung ist, dass sämtliche Daten des Objektes in einem Datentyp enthalten sind. Die gesamten Daten über die ROI und deren Feature-Vektoren eines Notenblattes werden zusammenhängend gespeichert. Eine aufwendige Rekonstruktion der Datenstruktur über Schlüssel-Fremdschlüsselbeziehungen (wie im Schema `IPFV`, vgl. Abbildung 10) ist nicht erforderlich.
Der Nachteil einer solchen objektrelationalen Speicherung der Objekte ist, dass die Zugriffsunterstützung von Attributen durch Indexe des DBMS nicht mehr möglich ist. Wird für die Attribute eines UDT's ein Index benötigt, so muss dieser individuell implementiert werden (vgl. Abschnitt 2.1.3).

Fazit: Der wesentliche Nachteil der objektrelationalen Speicherung ist, dass einige Techniken des Datenbanksystems (v.a. Indexe) auf die Attribute eines UDT's nicht angewendet werden können. Die Suche nach bestimmten Attributen der Feature-Vektoren wird derzeit jedoch auch nicht benötigt, so dass der Verzicht auf die Möglichkeit, spezielle Zugriffpfade anzulegen, in Kauf genommen werden kann. Vielmehr werden i.d.R. alle Merkmale einer Notenhandschrift benötigt. Diese liegen bei einer objektrelationalen Speicherung stets zusammenhängend (als BLOB) vor und müssen nicht durch Verbunde aufwendig rekonstruiert werden. Aus diesem Grund soll die Erstellung eines objektrelationalen Modells auf Basis eines speziellen UDT's erfolgen.

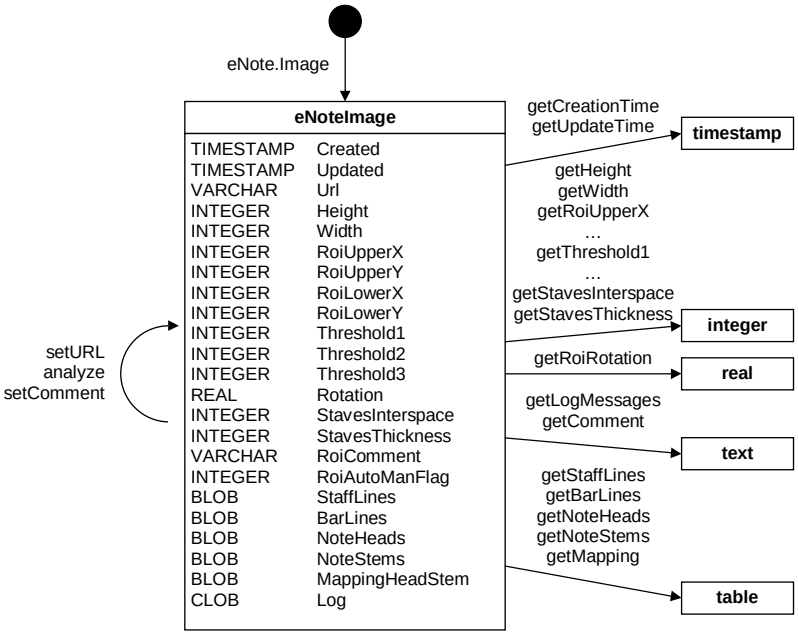

Abbildung 13: Vereinfachtes Modell des Datentyps `eNoteImage`

4.1.2 Modellierung eines dedizierten Datentyps

Nachfolgend soll konzeptuell ein Datentyp modelliert werden, der für die Verwaltung der Bilder einschließlich der automatisch extrahierten Merkmale geeignet ist. Dieser Datentyp wird sich in Struktur und Definition an dem in Kapitel 2.1.3 vorgestellten Datentyp `SI_StillImage` des *DB2 UDB Still Image Extenders* [Sto02] orientieren.

Im vorhandenen relationalen Datenmodell sind für die Speicherung des Digitalisats und die Speicherung der zugehörigen ROI zwei Tabellen vorhanden. Da i.d.R. genau eine ROI je Digitalisat benötigt wird, werden die Attribute dieser beiden Relationen für den hier zu modellierenden Datentyp miteinander verschmolzen. Für den Spezialfall, dass für ein Digitalisat mehrere ROI's untersucht werden sollen, müssen entsprechend viele Instanzen dieses Datentyps erzeugt werden. Da die Binärdaten des Digitalisats nicht im Datentyp gespeichert werden, ist diese Vorgehensweise trotzdem effizient hinsichtlich des verwendeten Speicherplatzes.
Neben den Attributen dieser beiden Tabellen wird der Datentyp Methoden zum Starten der Extraktion der Merkmale und Attribute für die Kapselung der Feature-Vektoren enthalten. Die einzelnen Attribute und Methoden des Datentyps, der mit `eNoteImage` bezeichnet werden soll, sind in Abbildung 13 dargestellt.

4.1.3 Attribute des Datentyps `eNoteImage`

Das analysierte Bild wird nicht in binärer Form gespeichert, sondern lediglich durch seine URL referenziert. Diese Modellierung hat zwar den Nachteil, dass Änderungen an den Binärdaten selbst nicht überwacht werden können[15], aber den Vorteil, eine verteilte Speicherung der Daten zu ermöglichen. Dieser Punkt wurde bereits in Abschnitt 2.1.4 diskutiert.
Um die Zeitpunkte der Erstellung und ggf. späteren Veränderung der Attribute des Datentyps nachvollziehen zu können, werden entsprechende Zeitstempel gespeichert. Außerdem ist ein Attribut für einen Kommentar vorgesehen. Zur Beschreibung der ROI werden die Koordinaten ihrer Eckpunkte gespeichert. Das Attribut `RoiAutoManFlag` zeigt an, ob die Position der ROI manuell oder automatisch festgelegt wurde.

Des Weiteren wird das Ergebnis der automatischen Analyse gespeichert. Neben den allgemeinen Attributen über den Drehwinkel und das Notenliniensystem sowie einigen Schwellwerten werden auch die kollektionswertigen Feature-Vektoren verwaltet. Außerdem ist das Attribut `Log` vorgesehen, welches z.B. Debug-Informationen aus dem Analyseprozess des IGD-Frameworks enthalten kann.

4.1.4 Methoden des Datentyps `eNoteImage`

Der Konstruktor des Datentyps erwartet als Parameter die URL des Digitalisats. Optional können Argumente zur manuellen Definition der ROI und ein Kommentar übergeben werden. Die Signatur dieser Methode sieht deshalb folgendermaßen aus:

`eNote.Image(` *filename* `[,` *options* `[,` *comment* `]] ).`

Somit kann zur Laufzeit entschieden werden, ob beim Einfügen eines neuen Bildes (`filename`) die Standardwerte oder spezielle Parameter (`options`) für die ROI verwendet werden sollen. Der Zeitpunkt der Ausführung dieser Methode wird in den Attributen `created` und `updated` festgehalten.

Die Methode `setURL` ermöglicht die nachträgliche Änderung der URL eines Digitalisats. Wie dem Konstruktor können auch dieser Methode optional Parameter zur Definition der ROI und für das Kommentar-Attribut übergeben werden. Der Aufruf dieser Methode wird durch eine Aktualisierung des Attributes `updated` dokumentiert.

[15] Eine Zugriff auf im Dateisystem gespeicherte Dateien kann durch Verwendung des Datentyps `DATALINK` mittels des *Datalink-Managers* kontrolliert werden. In Abschnitt 5.2.2 wird jedoch beschrieben, warum dieser Datentyp nicht zur Anwendung kommen kann.

Die Methode `analyze` startet eine erneute Extraktion der Feature-Vektoren. Wird die Methode ohne Parameter aufgerufen, erfolgt die Analyse für die bereits definierten ROI-Koordinaten. Dieses ist sinnvoll, wenn sich z.B. der Inhalt des referenzierten Bildes geändert hat. Alternativ kann die Methode aufgerufen werden, um die Position der ROI neu zu definieren. Auch die Ausführung dieser Methode führt zu einer Aktualisierung des Attributes `updated`.

Die Methode `setComment` kann dafür genutzt werden, das Kommentarfeld unabhängig von den zuvor vorgestellten Methoden zu ändern. Der Aufruf dieser Methode wird jedoch nicht durch eine Änderung der Zeitstempel dokumentiert.

Die *Getter-Methoden* (vgl. Abbildung 13, rechts) ermöglichen den Zugriff auf die einzelnen Attribute des Datentyps. Für die kollektionswertigen Attribute, die als BLOB's gespeichert sind, werden entsprechende Methoden zur Deserialisierung der Daten benötigt.

4.1.5 Zusammenfassung

Die Digitalisate und deren Feature-Vektoren können durch Definition des Datentyps `eNoteImage` nunmehr als Attribute einer Tabelle gespeichert werden, z.B.

```
CREATE TABLE enote.images(
      id      INTEGER NOT NULL PRIMARY KEY
              GENERATED ALWAYS AS IDENTITY,
      image   ENOTE.IMAGE)
```

Das Einfügen eines neuen Digitalisats in diese Tabelle erfolgt durch den folgenden SQL-Befehl:

```
INSERT INTO enote.images(image)
VALUES (ENOTE.IMAGE('http://www.enotehistory.de/beispiel.tif'))
```

Die ROI wird in diesem Fall für die Standardparameter berechnet.

Um einen bequemen Zugriff auf die einzelnen Attribute, insbesondere auf die kollektionswertigen Feature-Vektoren zu ermöglichen, sollten entsprechende Sichten definiert werden. So können die Daten auf eine dem vorhandenen Schema `IPFV` ähnliche Art dargestellt werden.

4.2 Automatische Schreiberklassifikation

Die automatische Klassifikation der einzelnen Digitalisate erfolgt anhand ihrer Merkmale, die mit den zuvor vorgestellten Werkzeugen extrahiert werden können. Der Klassifiktion liegt dabei die Annahme zu Grunde, dass jedes Notenblatt (Digitalisat) von genau einem Schreiber geschrieben wurde. Typischerweise werden je Notenblatt ungefähr 200 Notenköpfe, 200 Notenhälse und 10 Notenliniensysteme erkannt. Deren geometrische Eigenschaften werden durch die Feature-Vektoren beschrieben. Neben diesen Merkmalen einzelner Objekte werden auch allgemeine Eigenschaften des Digitalisats, wie der Abstand der Notenliniensysteme, berücksichtigt.
Bei der Klassifikation eines Notenblattes werden die geometrischen Eigenschaften der einzelnen gefundenen Objekte miteinander verglichen. Die räumliche Anordnung der Objekte wird hingegen nicht berücksichtigt.

4.2.1 Vorbetrachtungen

Die vorhandene Implementierung des IGD basiert auf dem durch das *Interface* `Analyzer` und dessen Implementierung `BuildOnceOnlyAnalyzer` vorgegebenen Arbeitsablauf. Dieser erwartet zunächst die Angabe einer Trainingsmenge von Instanzen. Anhand der darin implizit enthaltenen Menge an Schreibern wird entschieden, ob ein entsprechender Klassifikator bereits vorhanden ist und geladen werden kann. Ist das nicht der Fall, muss ein Klassifikator auf Grundlage der Trainingsinstanzen erstellt werden.
Eine direkte Portierung dieser Implementierung in das Datenbanksystem wäre relativ einfach. Hierfür wäre zunächst die Erstellung eines neuen `InstancesGetter` zum Laden der Instanzen erforderlich, die Klasse `BuildOnceOnlyAnalyzer` müsste derart verändert werden, dass die Klassifikatoren statt im Dateisystem nun in der Datenbank abgelegt werden und schließlich müsste die Funktionalität der Klasse `ENoteDataMiningRuntimeApplication` in Form einer Stored Procedure dem Datenbanknutzer zugänglich gemacht werden.

Eine derartige Umsetzung hat jedoch mehrere Nachteile:

- Für die Klassifikation eines Notenblattes müsste jedes Mal eine Trainingsmenge explizit angegeben werden. Alternativ könnte eine automatische Auswahl der Trainingsmenge implementiert werden. Dann hätte der Nutzer jedoch keinen Einfluss auf die Trainingsmenge, was ebenfalls nicht wünschenswert ist.

- Die Erstellung des Klassifikators wird während der Analyse eines Notenblattes vorgenommen, wenn die übergebene Trainingsmenge einen neuen Schreiber beinhaltet. Wünschenswert wäre jedoch die Trennung der Erstellung des Klassifikators von der eigentlichen Analyse. Dann

könnte das Bauen des Klassifikators zu Zeiten durchgeführt werden, wenn der Server weniger stark ausgelastet ist.

- Ein neuer Klassifikator wird nur dann erstellt, wenn sich die Menge der Schreiber ändert. Es kann jedoch auch gewünscht sein, einen neuen Klassifikator zu erstellen, wenn für vorhandene Schreiber neue Instanzen in die Datenbank eingefügt wurden.

Außerdem wäre es günstig, wenn verschiedene Varianten eines Klassifikators, auch für die gleiche Schreibermenge, verfügbar sind. Die Auswahl des für den Anwendungsfall jeweils günstigsten Klassifikators könnte beispielsweise anhand der Ergebnisse der Testphase erfolgen.
Aus diesem Grund wird statt einer direkten Portierung der Implementierung die Erstellung eines eigenen UDT's angestrebt. Dieser erstellt und testet einen Klassifikator auf Grundlage einer gegebenen Trainings- und Testmenge und kapselt ihn. Darauf basierend werden Methoden für die Anwendungsphase des DM bereitgestellt.
Die Trainingsphase kann dann beispielsweise nachts durchgeführt werden, wenn der Server vermutlich geringer ausgelastet ist. Außerdem können verschiedene Instanzen eines solchen Datentyps in einer typisierten Tabelle der Datenbank gespeichert werden.
Diese Änderung des Arbeitsablaufes erfordert jedoch einen höheren Implementierungsaufwand. Die erforderlichen Änderungen werden in Kapitel 5 beschrieben. In den folgenden Abschnitten sollen nun die Anforderungen an den Datentyp formuliert werden. Zum Vergleich wird das Konzept des Standards SQL/MM, Part 6 betrachtet.

4.2.2 Definition eines dedizierten Datentyps

Zur Vorbereitung der Trainingsphase müssen im Standard SQL/MM die Trainingsinstanzen und verschiedene Einstellungen (Modellparameter, logische Datenbeschreibung) definiert werden. Mit diesen Informationen kann dann das Modell gebildet werden. Bei dieser speziellen Umsetzung sind jedoch die meisten Einstellungen in der Implementierung bereits berücksichtigt und müssen deshalb nicht explizit angegeben werden. Optional können bei der Erstellung des Datentyps der Typ des Klassifikators und dazu gehörige Optionen übergeben werden. Für die Initialisierung der Trainingsphase selbst ist dann lediglich die Übergabe der Trainingsinstanzen erforderlich.

Anhand des in der Trainingsphase erstellten Modells und einer Menge von Testinstanzen kann anschließend eine Validierung des Modells erfolgen. Eingabeparameter für die Testphase ist somit ausschließlich die Menge der Testinstanzen. Für die Klassifikation kann auf diese Phase auch verzichtet werden. Sie liefert jedoch wertvolle Aussagen über die Qualität des Modells und damit der anschließenden Klassifikation.

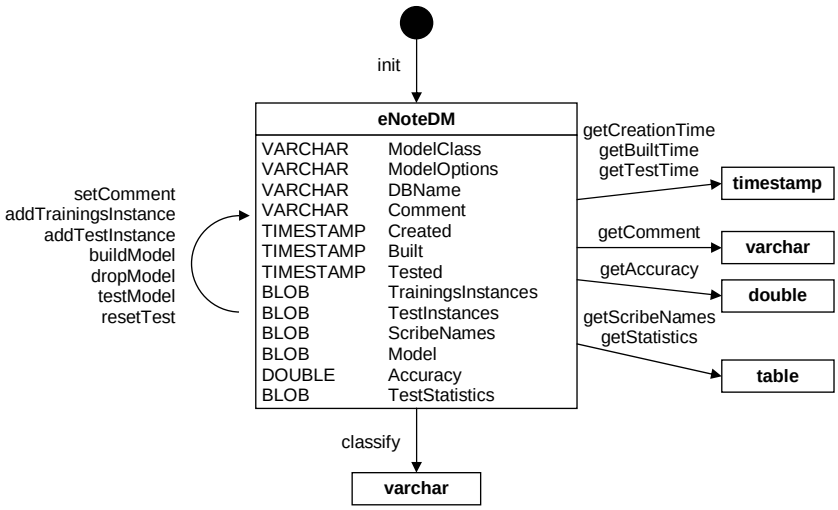

Abbildung 14: Vereinfachtes Modell des Datentyps `eNoteDM`

Die dritte und letzte Phase des Prozesses ist die Anwendungsphase. Hier werden mittels des Modells Instanzen klassifiziert. Eingabeparameter ist also eine konkrete Instanz (hier: ein Notenblatt, repräsentiert durch seine Feature-Vektoren).

Im Standard SQL/MM wird, um eine wiederholte Modellbildung zu vereinfachen, zwischen dem Schritt der Modellbeschreibung (hier als Vorbereitung der Trainingsphase beschrieben) und den anschließenden Phasen des DM unterschieden. Bei der Umsetzung des DM-Prozesses für das *eNoteHistory*-Projekt kann jedoch auf die abstrakte Modellbeschreibung verzichtet werden, denn die Beschreibung spezieller Einstellungen ist nicht erforderlich (siehe oben) und bei Verwendung derselben Trainingsinstanzen ist keine erneute Modellbildung nötig.

Aus diesem Grund wird lediglich ein UDT `eNoteDM` für die Realisierung des *KDD-Prozesses* modelliert. Dieser kapselt die erforderlichen Attribute und stellt basierend hierauf die Methoden für die einzelnen Phasen zur Verfügung. Die einzelnen Attribute und Methoden dieses UDT's sind in Abbildung 14 schematisch dargestellt und sollen nachfolgend ausführlicher beschrieben werden.

4.2.3 Attribute des Datentyps `eNoteDM`

Zur Beschreibung der unterschiedlichen Modelle ist ein Attribut `comment` vorgesehen. Hier können menschlich lesbare Informationen über den Verwendungszweck o.dgl. in Form einer Zeichenkette gespeichert werden. Für den Typ und spezielle Parameter für des Klassifikators sind ebenfalls

Attribute vorhanden. Sind diese nicht gesetzt, werden Standardwerte verwendet. Die Tabelle, in denen die Digitalisate für die Trainings- und Testphase gespeichert sind, wird ebenfalls durch ein eigenes Attribut beschrieben.
Des Weiteren werden verschiedene Zeitstempel gespeichert, die den Zeitpunkt der Erstellung des DM-Objektes sowie der Erstellung und Validierung des Modells wiedergeben.
Ein Verweis auf die zu verwendende Menge der Trainings- und Testinstanzen wird ebenfalls gespeichert. Da das verwendete Datenbanksystem IBM DB2 kollektionswertige Attribute nicht unterstützt, werden diese Daten serialisiert und in Form von BLOB's gespeichert.
Die Menge der durch dieses Modell klassifizierbarer Schreiber wird ebenfalls gespeichert. Sie entspricht genau der Menge der Schreiber der Trainingsinstanzen. Um einen schnellen Zugriff auf diese Menge zu ermöglichen, wird sie explizit gespeichert.
Der Entscheidungsbaum, der durch den *Analyzer* generiert wird, wird auch in Form eines BLOB gespeichert (Model).
Die Genauigkeit des Modells, welches während der Testphase ermittelt wird, wird ebenso wie eine ausführliche Statistik über die Testergebnisse bereitgehalten.

4.2.4 Methoden des Datentyps `eNoteDM`

Die Methode `init` agiert als Konstruktor des Datentyps. Als Parameter können dieser Methode der Typ und die Parameter des Klassifikators in Form jeweils einer Zeichenkette übergeben werden:

```
eNote.dm_init( [classifier, options] ).
```

Der Zeitpunkt der Ausführung der Methode wird als Wert des Attributes `created` gespeichert. Die übrigen Attribute des Datentyps haben nach dieser Initialisierung den Zustand `NULL`.

Die *Getter-Methoden* `getCreationTime`, `getBuiltTime`, `getTestTime`, `getComment`, `getAccuracy`, `getScribeNames` und `getStatistics` liefern direkt den Zustand der jeweiligen Attribute des Datentyps zurück. Dabei ist zu beachten, dass die Methoden `getScribeNames` und `getStatistics` jeweils Tabellen als Ergebnis liefern. Die zugehörigen Attribute werden jedoch intern als BLOB's gespeichert. Hier sind deshalb entsprechende Methoden zur Konvertierung erforderlich.

Die Methode `addTrainingsInstance` dient dazu, die für die Modellbildung erforderlichen Trainingsinstanzen anzugeben. Eine solche Instanz besteht jeweils aus der Spezifikation eines Digitalisats und des zugehörigen

(bereits bekannten) Schreibers. Das Hinzufügen von Trainingsinstanzen ist möglich, solange kein Modell vorhanden ist.

Die Methode `addTestInstance` dient analog dazu, die Instanzen für die Testphase zu spezifizieren. Das Hinzufügen von Testinstanzen ist möglich, solange das Modell noch nicht getestet wurde. Außerdem können nur solche Schreiber von Notenhandschriften hinzugefügt werden, für die das Modell auch trainiert wurde, d.h. die in der Menge der Trainingsinstanzen enthalten sind.
Der Standard SQL/MM und somit auch der IBM Intelligent Miner sieht eine andere Art der Verwaltung der Trainings- und Testinstanzen vor. Dort wird jeweils der Verweis auf eine Tabelle oder Sicht der Datenbank sowie eine logische Beschreibung der einzelnen Attribute dieser Tabelle (*Mapping*) angegeben. Konkret bedeutet das, dass für jedes Modell eine Sicht (selten direkt eine Tabelle) sowohl für die Trainings- als auch für die Testinstanzen definiert werden muss. Um dieses zu vermeiden, sollen im Datentyp `eNoteDM` die Trainings- und Testmenge intern verwaltet werden.

Mit der Methode `buildModel` wird die Erzeugung des Modells angestoßen. Voraussetzung für den Aufruf der Methode ist, dass eine Menge von Trainingsinstanzen angegeben wurde. Nach erfolgreicher Beendigung der Methode ist der Entscheidungsbaum (Attribut: `Model`) und die Menge der Schreiber (Attribtut: `ScribeNames`) verfügbar.

Die Methode `testModel` initiiert den zweiten Schritt des Klassifikationsprozesses, die Testphase. Voraussetzung hierfür ist zum einen das Vorhandensein eines Modells und zum anderen die Definition von Testinstanzen (siehe oben). Als Ergebnis der Ausführung dieser Methode liegen anschließend die Testergebnisse in Form der Attribute `accuracy` und `TestStatistics` vor.

Die Methode `resetTest` setzt die mit der Methode `testModel` ermittelten Testergebnisse wieder zurürck. Dieses ist erforderlich, um z.B. neue Testinstanzen zu definieren und damit einen neuen Testlauf zu starten.

Die Methode `dropModel` verwirft analog das Ergebnis der Methode `buildModel`, um anschließend z.B. die Menge Trainingsinstanzen zu ändern und eine neue Modellbildung zu starten.

4.2.5 Zusammenfassung

Die verschiedenen DM-Modelle können nun als Instanz des Datentyps `enote.dm` in der Datenbank erstellt, gespeichert und genutzt werden. Die Anwendung des Datentyps und die Syntax der SQL-Befehle soll in diesem Abschnitt anhand eines Beispiels gezeigt werden.
Für die Verwaltung der DM-Modelle wird eine spezielle Tabelle angelegt und in diese wird anschließend ein neues DM-Objekt eingefügt:

```
CREATE TABLE datamining(
      id     INTEGER NOT NULL PRIMARY KEY,
      model  ENOTE.DM )

INSERT INTO datamining(id, model) VALUES (1, enote.dm_init())
```

Anschließend können diesem Objekt ausgewählte Trainingsinstanzen hinzugefügt werden. Der erste Parameter der Methode gibt dabei die *Id* des Digitalisats in der Tabelle `enote.images` (vgl. Abschnitt 4.1.5) und der zweite den Namen dessen Schreibers an.

```
UPDATE datamining
SET model=model..addTrainInstance(123, 'scribename')
WHERE id=1
```

Sind alle Trainingsinstanzen definiert, kann das Modell gebaut werden:

```
UPDATE datamining
SET model=model..buildModel()
WHERE id=1
```

Das Hinzufügen der Testinstanzen erfolgt analog den Trainingsinstanzen (siehe oben). Sollen nicht nur einige ausgewählte, sondern mehrere bzw. alle gespeicherten Digitalisate für die Tests verwendet werden, muss der UPDATE-Befehl entsprechend oft rekursiv aufgerufen werden. Dieses kann entweder durch ein externes Programm (z.B. via JDBC) oder durch eine SQL-Routine unter Verwendung des Cursor-Konzeptes erfolgen.
Das Testen des Modells anhand der definierten Testinstanzen erfolgt mittels der Methode `testModel`. Anschließend kann das Attribut `accuracy` des Datentyps abgefragt werden:

```
UPDATE datamining SET model=model..testModel() WHERE id=1

SELECT model..accuracy FROM datamining WHERE id=1
```

Die Klassifikation eines bereits in der Datenbank gespeicherten Digitalisats mit der *Id* 123 erfolgt dann durch den Befehl

```
SELECT model..classify(image)
FROM datamining, enote.images
WHERE datamining.id=1 AND enote.images.id=123
```

Ein noch nicht in der Datenbank vorhandenes Digitalisat kann durch Kombination der Methode `classify` des Datentyps `enote.dm` mit dem Konstruktor des Datentyps `enote.image` klassifiziert werden:

```
SELECT model..classify(enote.image('http://url/image.tif'))
FROM datamining
WHERE id=1
```

5 Implementierung

Die Umsetzbarkeit des in Kapitel 4 aufgestellten Modells wird anhand einer prototypischen Implementierung gezeigt. In diesem Abschnitt werden die für diese Implementierung erforderlichen Schritte beschrieben. Aufgabe dieser Beschreibung ist es, die einzelnen Arbeitsschritte nachvollziehen zu können. Auf eine detailierte Beschreibung der einzelnen Methoden oder bzw. umfangreiche Abbildungen des Quellcodes wird jedoch verzichtet. Hierfür sei auf die der Arbeit beiliegende CD verwiesen.

5.1 Allgemeines

Bevor in den folgenden Abschnitten 5.2 und 5.3 die Implementierung der beiden UDT's beschrieben wird, sollen nachfolgend einige für beide Datentypen erforderlichen Grundlagen dokumentiert werden.

5.1.1 Packagestruktur

Für die Klassen zur Extraktion der Feature-Vektoren und für das Data Mining wurde das *Package* `de.enotehistory` angelegt. In diesem sind weitere Unterverzeichnisse vorhanden, die sowohl die Klassen des vorhandenen Frameworks als auch neu implementierte Klassen enthalten. In der folgenden Tabelle 2 ist diese Struktur dokumentiert.

Package	**Inhalt**
`analysis`	enthält die Klassen des vorhandenen *Packages* `ENoteAnalysis` sowie die neu erstellte Klasse `AnalysisResult` (siehe Abschnitt 5.2.1)
`basics`	Klassen des bisherigen *Packages* `ENoteBasics`
`datamining`	enthält die weiterhin erforderlichen Klassen des vorhandenen *Packages* `ENoteDataMining` und die zusätzlich impementierten Klassen `InstancesSet`, `DB2InstancesGetter` und `DB2Connection` (siehe Abschnitt 5.3.1)
`datamining.analyzer`	enthält die Klassen des *Packages* `ENoteDataMining.Analyzer`; die Klasse `BuildOnceOnlyAnalyzer` wurde durch `LoadAndSaveAnalyzer` ersetzt (vgl. Abschnitt 5.3.1)
`datamining.voter`	bisher *Package* `ENoteDataMining.ENoteVoter`
`gui`	bisher *Package* `ENoteGUI`
`operator`	bisher *Package* `ENoteOperator`
`udf`	enthält die neu implementierten Klassen mit den *User Defined Functions* (siehe Abschnitt 5.3.1)

Tabelle 2: Struktur des *Packages* `de.enotehistory`

5.1.2 Logging und SQLSTATE's

Da die zu implementierenden *User Defined Functions* durch die Laufzeitumgebung des Datenbanksystems ausgeführt werden, ist die Ausgabe von Informationen über den Ablauf und Zustand einer Methode nur auf eingeschränkte Art und Weise möglich. Ausnahmesituationen werden, wie in Java üblich, durch *Exceptions* signalisiert und durch das Datenbanksystem entsprechend ausgegeben. Eine Einordnung des jeweiligen Fehlerfalls in Kategorien kann durch die Ausgabe von `SQLSTATE`'s erfolgen. Ein solches Konzept wird im Rahmen dieser Arbeit jedoch nicht erstellt (siehe Abschnitt 6).

Besonders während der Entwicklungphase des Systems ist die Ausgabe sogenannter *Debug*-Informationen erforderlich. Im vorhandenen Framework erfolgte dieses direkt auf der Standard-Ausgabe (`System.out`). Im Rahmen dieser Arbeit wird für die Umsetzung dieser Funktionalität die Java-Bibliothek Log4J[16] von *Apache* verwendet. Damit ist die Generierung von *Logging*-Informationen verschiedener Prioritätsstufen (Level) möglich: von `DEBUG`, über `INFO` und `WARN` bis zu `ERROR` und `FATAL`. Diese Nachrichten werden durch einen *Appender* gesammelt und weiterverarbeitet. Bei der Entwicklung des Prototyps wurde ein `FileAppender` verwendet, der die Log-Nachrichten in eine Textdatei speichert. Außerdem können durch den *Appender* sowohl das Format der einzelnen Log-Informationen als auch das Mindestlevel der zu verarbeitenden Nachrichten eingestellt werden.
Im Konzept beider UDT's ist die Speicherung der Logging-Informationen als Attribut des Datentyps vorgesehen. Im Rahmen der prototypischen Implementierung ist diese Speicherung nicht umgesetzt, doch durch die Verwendung der Bibliothek Log4J (siehe oben) kann diese Veränderung durch Modifikation des *Appenders* relativ einfach erfolgen.

5.2 Umsetzung des Datentyps `eNote.Image`

5.2.1 Java-Implementierung

Auf der Basis der in Abschnitt 4 formulierten Anforderungen die wurde Klasse `ImageAnalyzerUDF` als *User Defined Function* erstellt. Die Grundlage der Implementierung dieser Klasse bilden v.a. die in Abschnitt 3.1.2 vorgestellten, bereits vorhandenen Einstiegspunkte in die Analyse.

Diese UDF erwartet als Parameter die URL des zu analysierenden Digitalisats und eine Zeichenkette, welche die Position der ROI beschreibt. Diese Beschreibung erfolgt durch Angabe des Abstandes der ROI vom Seitenrand des Bildes in der Form: `"MARGIN_TOP, MARGIN_BOTTOM, MARGIN_LEFT, MARGIN_RIGHT"`. Ist diese Zeichenkette `NULL` oder leer, werden die Standard-

[16] vgl. http://logging.apache.org/log4j/docs/

werte für die Definition der ROI verwendet. Das Ergebnis dieser Methode ist ein Tupel mit den Attributen des Datentyps `eNote.Image`:

$$\textit{analyzeImage}(\texttt{url}, \texttt{roi\_args}) \rightarrow (\texttt{width}, \texttt{height}, ..., \texttt{mapping\_blob})$$

Für die interne Speicherung eines solchen Tupels wurde die Hilfsklasse `AnalysisResult` erstellt. Diese kapselt die Attribute des Tupels und stellt für den Zugriff auf diese entsprechende Getter- und Setter-Methoden bereit.

Die Extraktion der Feature-Vektoren selbst erfolgt durch das Framework des IGD. Folgende Veränderungen mussten an der vorhandenen Implementierung vorgenommen werden:

- Die Klasse `org.fhg.igd.media.jai.OperationRegister` registriert Operatoren, die durch die verwendete Bibliothek JAI verwendet werden. Beim Aufruf einer UDF erzeugt das Datenbanksystem eine Instanz der entsprechenden Klasse und führt den Methodenaufruf durch. Dabei erfolgt u.a. das Registrieren der genannten Operatoren. Wird zu einem späteren Zeitpunkt die UDF erneut aufgerufen, würde der Versuch, die Operatoren erneut zu registrieren, das Anstoßen einer Ausnahmebehandlung (*Exception*) bewirken. Um dieses zu verhindern wurde die Klasse `OperationRegister` durch einen *try-catch*-Blockes um die entsprechenden Anweisungen ergänzt

- Die Klasse `eNoteEllipseFitting` kapselt die *EllipseFitting*-Parameter der erkannten Notenköpfe, Notenhälse und Taktstriche. Diese Parameter müssen für die Speicherung innerhalb des Datentyps `eNote.Image` als BLOB serialisiert werden (vgl. Abbildung 15). Auf diesem Grund wird in dieser Klasse zusätzlich die Methode `writeToByteArray` implementiert, welche die Attribute der jeweiligen Instanz in binärer Form liefert.

- Das Ergebnis der automatischen Analyse des IGD-Frameworks wird durch die Klasse `ENoteImageFeatureVector` verwaltet. Um dessen Attribute in eine Instanz der Klasse `AnalysisResult` zu konvertieren, wurde der Klasse `ENoteImageFeatureVector` die entsprechende Methode `getResult` hinzugefügt.

Um den Zugriff auf die als BLOB gespeicherten, kollektionswertigen Feature-Vektoren zu ermöglichen, werden UDF's erstellt, die die Darstellung der binären Informationen als Tabellen ermöglichen. Die Klasse `EllipseDataUDF` implementiert eine UDF, die eine Tabelle mit den *Ellipse-Fitting*-Parametern zurückliefert. Die Serialisierung der Daten als BLOB ist in Abbildung 15 dargestellt. Die ersten 4 Byte des BLOB stellen die Anzahl der vorhandenen Tupel dar, anschließend werden die einzelnen Attribute der

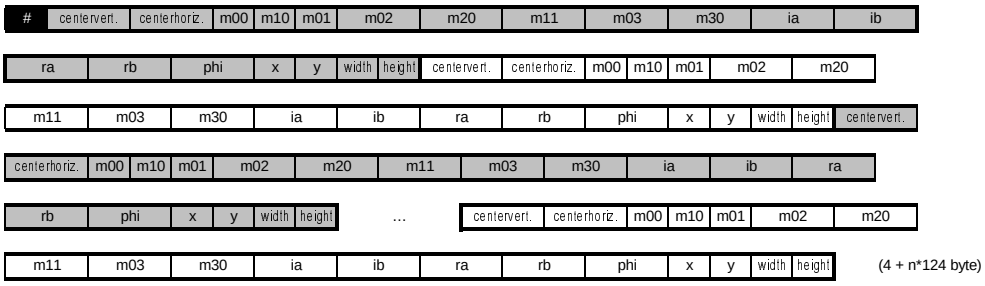

Abbildung 15: Serialisierung der *EllipseFitting*-Attribute

Tupel (insgesamt jeweils 124 Byte) abgelegt.

Analog ermöglichen die Klassen `StaffLinesUDF` und `MappingHeadStemUDF` die Entnestung der Attribute `STAFF_LINES_BLOB` und `MAPPING_BLOB`. Der Aufbau der jeweiligen BLOB's ist in den folgenden Abbildungen 16 und 17 dargestellt.

Abbildung 16: Schematischer Aufbau des BLOB `STAFF_LINES_BLOB`

Abbildung 17: Schematischer Aufbau des BLOB `MAPPING_BLOB`

Für die Implementierung tabellenwertiger UDF's ist die Nutzung des sogenannten *Scratchpads* erforderlich. Für das Auslesen der einzelnen Zeilen der Tabelle ruft das Datenbanksystem die Methode so oft auf, bis das Ende der Tabelle signalisiert wird. Bis dahin wird durch jeden Methodenaufruf eine Zeile der Tabelle durch die Methode geliefert. Das *Scratchpad* speichert hierfür zwischen den einzelnen Methodenaufrufen die aktuelle Zeile in der Tabelle. Die Klasse `Scratchpad` mit ihren Methoden `read():int` und `write(int)` dient dazu, den Zugriff auf das *Scratchpad* zu vereinfachen.

5.2.2 DB2-Implementierung

Um die Funktionalität der in Java implementierten UDF's in der Datenbankumgebung zu nutzen, müssen diese zunächst registriert werden. Dieses erfolgt durch den Befehl `CREATE FUNCTION`, in dem neben der Signatur der Methode und dem Namen der Java-Klasse auch eine Vielzahl an Parametern gesetzt werden können.
Die Methode `analyzeImage` der Klasse `ImageAnalyzerUDF` wird durch fol-

genden Befehl als UDF enote.analyzeImage registriert[17]:

```
CREATE FUNCTION enote.analyzeImage(VARCHAR(255), VARCHAR(255))
RETURNS TABLE (
    width INTEGER, height INTEGER,
    upper_x INTEGER, upper_y INTEGER,
    lower_x INTEGER, lower_y INTEGER,
    threshold_1 INTEGER, threshold_2 INTEGER,
    threshold_3 INTEGER, rotation REAL,
    staves_interspace INTEGER,
    staves_thickness INTEGER,
    roi_comment VARCHAR(255),
    roi_auto_man_flag INTEGER,
    staff_lines_blob BLOB, bar_lines_blob BLOB,
    note_head_blob BLOB, note_stem_blob BLOB,
    mapping_head_stem_blob BLOB, log CLOB)
EXTERNAL NAME '...ImageAnalyzerUDF!analyzeImage'
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
SCRATCHPAD 10
...
```

Analog werden die Klassen zur Entnestung der komplexen Attribute registriert:

```
CREATE FUNCTION enote.getEllipseTable(BLOB)
RETURNS TABLE (
    id INTEGER,
    centervertical DOUBLE, centerhorizontal DOUBLE,
    m00 INTEGER, m10 INTEGER, m01 INTEGER,
    m02 DOUBLE, m20 DOUBLE, m11 DOUBLE,
    m03 DOUBLE, m30 DOUBLE,
    ia DOUBLE, ib DOUBLE, ra DOUBLE, rb DOUBLE,
    phi DOUBLE, bbox_x INTEGER, bbox_y INTEGER,
    bbox_width INTEGER, bbox_height INTEGER )
EXTERNAL NAME '...EllipseDataUDF!getEllipseData' ...

CREATE FUNCTION enote.getStaffLinesTable(BLOB)
RETURNS TABLE (
    staff_line_id INTEGER,
    begin_staves_y INTEGER,
    end_staves_y INTEGER)
EXTERNAL NAME 'de.enotehistory.udf.StaffLinesUDF!getStaffLines'
...
```

[17] Die Befehle zum Erzeugen der UDF's und UDT's werden so dargestellt, dass die Signatur der jeweiligen Methoden ersichtlich und somit die Nutzung der Typen und Methoden ermöglicht wird.

```
CREATE FUNCTION enote.getMappingTable(BLOB)
RETURNS TABLE (
    note_head_id INTEGER, note_stem_id INTEGER)
EXTERNAL NAME '...MappingHeadStemUDF!getMappingHeadStem'
...
```

Anschließend kann der Datentyp `eNote.image` definiert werden. Dieses erfolgt durch den Befehl `CREATE TYPE`. Die Methoden des Datentyps werden durch den Befehl `ADD METHOD` deklariert und durch `CREATE METHOD` implementiert.

```
CREATE TYPE enote.image AS (
    url VARCHAR(255),
    height INTEGER, width INTEGER,
    roi_upper_x INTEGER, roi_upper_y INTEGER,
    roi_lower_x INTEGER, roi_lower_y INTEGER,
    threshold_1 INTEGER, threshold_2 INTEGER,
    threshold_3 INTEGER, rotation REAL,
    staves_interspace INTEGER,
    staves_thickness INTEGER,
    roi_comment VARCHAR(255),
    roi_auto_man_flag INTEGER,
    staff_lines_blob BLOB, bar_lines_blob BLOB,
    note_head_blob BLOB, note_stem_blob BLOB,
    mapping_blob BLOB, log CLOB,
    created TIMESTAMP, updated TIMESTAMP)
INSTANTIABLE
NOT FINAL
...

ALTER TYPE enote.image
    ADD METHOD analyze(url VARCHAR(255), options VARCHAR(255))
        RETURNS enote.image
        LANGUAGE SQL
        ...
    ...

CREATE METHOD analyze(url VARCHAR(255), options VARCHAR(255))
    FOR enote.image
    RETURN
        SELECT enote.image()..url(url)..height(i.height)..
        ...
        created(self..created)..
        updated(current timestamp)
        FROM table(enote.analyzeImage(url, options)) as i,
        WHERE url IS NOT NULL

...
```

Die URL des referenzierten Digitalisats wird als einfache Zeichenkette gespeichert. Zur Verwaltung derartiger Referenzen auf Dateien stellt das Datenbanksystem den Datentyp DATALINK zur Verfügung. Allerdings kann dieser Datentyp nur dann in anderen Datentypen verwendet werden, wenn diese als typisierte Tabellen gespeichert werden. Die Definition eines Konstruktors für einen zusammengesetzten Datentyp, der eine Attribut vom Typ DATALINK besitzt, ist nicht möglich. Aus diesem Grund wird bei der Definition des Datentyps eNote.Image auf die Verwendung dieses speziellen Typs verzichtet und statt dessen das Attribut als Zeichenkette (VARCHAR) gespeichert.

Abschließend werden die Konstruktoren des Datentyps erzeugt. Diese rufen mit den übergebenen Parametern url und optional options die UDF enote.analyzeImage auf und weisen das Ergebnis dieses Methodenaufrufs den jeweiligen Attributen des Datentyps eNote.Image zu.

```
CREATE FUNCTION enote.image(url VARCHAR(255))
    RETURNS enote.image
    LANGUAGE SQL
    NO EXTERNAL ACTION
    RETURN
        SELECT enote.image()..url(url)..height(i.height)..
        ...
        created(current timestamp)..
        updated(current timestamp)
        FROM table(enote.analyzeImage(url, '')) as i,
        WHERE url IS NOT NULL

CREATE FUNCTION enote.image(u VARCHAR(255),o VARCHAR(255))
    RETURNS enote.image
    ...
```

5.2.3 Sichten

Die einzelnen analysierten Digitalisate seien in Form des Datentyps eNote.Image in einer Tabelle enote.images gespeichert.

```
CREATE TABLE enote.images(
    id INTEGER NOT NULL PRIMARY KEY ...,
    image ENOTE.IMAGE)
```

Um eine relationale Darstellung der Feature-Vektoren analog zu dem vorhandenen Schema IPFV zu erhalten, können verschiedene Sichten basierend auf dieser Tabelle definiert werden:

```
CREATE VIEW enote.imagedetails AS
    SELECT id,
```

```
        image..height as height, ...
        image..getcreationtime() as created,
        image..getupdatetime() as updated
    FROM enote.images

CREATE VIEW enote.note_head AS
    SELECT a.id as image_id, b.id as noteheadid,
        centervertical, centerhorizontal, m00, m10, ...
    FROM enote.images a,
        table(enote.getEllipseTable(a.image..note_head_blob)) b
    WHERE a.image..note_head_blob is not null
```

5.3 Umsetzung des Datentyps `eNote.DM`

5.3.1 Java-Implementierung

In diesem Abschnitt werden die zur Umsetzung des Konzeptes erforderlichen Änderungen des in Abschnitt 3.2 vorgestellten Frameworks beschrieben. Dazu gehört sowohl die Implementierung eigener, als auch die Anpassung (Erweiterung) bereits vorhandener Klassen.

Beschreibung der Trainings- und Testinstanzen: Entsprechend dem in Kapitel 4 erstellten Konzept handelt es sich bei den Trainings- und Testinstanzen jeweils um ein Tupel, bestehend aus einer Referenz auf das Digitalisat und einer (namentlichen) Beschreibung des zugehörigen Schreibers. Diese Tupel müssen innerhalb des Datentyps `enote.dm` gespeichert und verwaltet werden können. Es werden somit Methoden benötigt, die diese Tupelmenge in eine flache Datenstruktur serialisieren und auch wieder entnesten können. Außerdem sind Methoden erforderlich, die das Hinzufügen neuer Tupel zu der Menge ermöglichen und dabei unterschiedliche Integritätsbedingungen berücksichtigen.

Die Klasse `InstancesSet` kapselt eine solche Tupelmenge, wie sie sowohl für das Trainings- als auch für die Testinstanzen benötigt werden. Intern wird diese Menge als `Hashtable` dargestellt, die die Daten in Form einer Abbildung *Digitalisat-ID* → *Schreiber* speichert. Auf diese Weise kann jedem Digitalisat nur genau ein Schreiber zugeordnet werden.
Zum Erstellen einer Instanz dieser Klasse sind zwei Konstruktoren verfügbar, von denen einer eine neue (leere) Menge erzeugt und der andere die Menge aus einem BLOB rekonstruiert. Umgekehrt lassen sich die Daten der Menge mit der Methode `getBinaryData` in eine binäre Darstellung umwandeln. Zum Hinzufügen eines neuen Tupels zu der Menge steht die Methode `addInstance` zur Verfügung. Um die Menge der in der Tupelmenge enthaltenen Schreiber zu erhalten, kann die Methode `getScribesArray` verwendet werden.

Die Klasse `InstancesSetUDF` implementiert die *User Defined Function* für die Klasse `InstancesSet`. Die Methode `addInstanceToBlob` erhält die als BLOB gespeicherte Menge der Trainingsinstanzen, die *Id* eines hinzuzufügenden Digitalisats und eine Zeichenkette für den Schreiber des Notenblattes und liefert die neue Menge der Trainingsinstanzen als BLOB zurück. Wird die *Id* eines bereits enthaltenen Digitalisats angegeben, wird dieses Tupel nicht hinzugefügt.
Die Methode `addInstancesToBlob2` wird für die Verwaltung der Testinstanzen verwendet. Da es nicht sinnvoll ist, ein Modell auf die Klassifikation von Schreibern zu testen, für die es nicht trainiert wurde, wird das Hinzufügen solcher Instanzen hier nicht erlaubt. Aus diesem Grund muss dieser Methode neben den oben für die Methode `addInstancesToBlob` beschriebenen Parametern auch die Trainingsinstanzen als Eingabewert erhalten.
Neben den Trainings- und Testinstanzen wird auch eine Liste der verfügbaren Schreiber im Datentyp `enote.dm` gespeichert. Mit der Methode `getScribeListBlob` kann dieser Menge ein weiterer Schreiber hinzugefügt werden. Dabei werden Duplikaten automatisch eliminiert. Um diese als BLOB gespeicherte Liste als Tabelle darstellen zu können, wurde die Methode `getScribesTable` implementiert.

Veränderung des Packages `Analyzer`: Um die in Abschnitt 5.1.2 beschriebenen *Logging*-Mechanismen unterstützen zu können, wurde der Klasse `AbstractAnalyzer` das Attribut `log` hinzugefügt. Dieses kann durch die ebenfalls hinzugefügte Methode `setLogger` initialisiert werden. Wenn dieses erfolgt ist, werden alle bisherigen an die Standardausgabe ausgegebenen Meldungen an den *Logger* gesendet.

Wie bereits in den Abschnitten 3.2.3 und 3.2.6 beschrieben, dient die Klasse `BuildOnceOnlyAnalyzer` dazu, die vorhandenen Klassifikatoren als binärer Form im Dateisystem zu sichern und bei Bedarf wieder zu laden. Dieser Zugriff erfolgt implizit beim Aufruf des Konstruktors. Konnte dabei ein entsprechender Klassifikator geladen werden, liefert die Methode `needTrainingInstances` den boolschen Wert *false* zurück, andernfalls müssen zunächst Trainingsinstanzen spezifiziert und der Klassifikator gebaut werden.
In der Beschreibung des Konzeptes wurde jedoch ein anderes Modell für die Verwaltung der Klassifikatoren gewählt (vgl. Abschnitt 4.2.1). Dabei wird in der Trainingsphase das Modell basierend auf den Trainingsinstanzen erzeugt. Dieses Modell wird dann als ein Attribut des Datentyps `enote.dm` gespeichert. In der anschließenden Test- und Anwendungsphase wird dann dieses Modell geladen und für die Klassifikation der Instanzen genutzt. Diese Funktionalität stellt die Klasse `LoadAndSaveAnalyzer` bereit, die die Klasse `BuildOnceOnlyAnalyzer` ersetzt. Im Übrigen bleibt die in Abbildung 11 dargestellte Implementierungshierarchie erhalten.

Die Klasse `LoadAndSaveAnalyzer` stellt zwei Konstruktoren bereit, die beide als Parameter einen Klassifikator des WEKA-Frameworks, die dazu gehörigen Optionen, die Menge der klassifizierten Schreiber und das untersuchte Merkmal erwarten. Außerdem erwartet der für die Trainingsphase genutzte Konstruktor als Parameter die Menge von Trainingsinstanzen. Der für die Test- und Anwendungsphase verwendete Konstruktor hingegen erwartet als fünften Parameter das (bereits vorhandene) Modell in binärer Form. Mit diesen Daten können die Konstruktoren den Klassifikator entweder neu erstellen oder rekonstruieren. In beiden Fällen sollte nach Aufruf des Konstruktors die Methode `needTrainingInstances` den Wert *false* liefern, so dass der *Analyzer* zum Klassifizieren von Testinstanzen genutzt werden kann.
Da die Methode `classify` der Klasse `SimpleAnalyzer` unabhängig davon, ob das Modell bereits erzeugt wurde, die Methode `buildClassifier` aufruft, wird diese Methode durch eine speziellere Implementierung der Klasse `LoadAndSaveAnalyzer` überschrieben.

Implementierung des DM-Prozesses: Im Framework des IGD wird eine Kombination der Trainings- und Testphase durch die Klassen `ENoteHistoryDataMiningRuntimeApplication` und `DataMiningRuntime` implementiert. Die Klasse `DataMiningUDF`, welche unterschiedliche *User Defined Function*s für die Trainings-, Test- und Anwendungsphase bereitstellt, basiert auf dem in diesen Klassen verwendeten Arbeitsablauf. Allerdings wird kein *ScribesManager* verwendet, und auf die Verwendung von *Threads* wird ebenfalls verzichtet, da keine parallele Bearbeitung verschiedener Klassifikatorinstanzen benötigt wird.
Für das Laden der Test- und Trainingsinstanzen aus der Datenbank wurden die Klassen `DB2InstancesGetter` und `DB2Connection` implementiert, die nachfolgend kurz beschrieben werden.

Die Klasse `DB2Connection` erzeugt beim Aufruf des Konstruktors eine Verbindung zur Datenbank. Da diese Klasse durch eine UDF aufgerufen wird, welche bereits durch das Datenbanksystem ausgeführt wird, kann diese Verbindung direkt hergestellt werden. Außerdem werden beim Aufruf des Konstruktors bereits sämtliche SQL-Anfragen bereits vorbereitet.
Der *Logging*-Mechanismus ist wie für die Klasse `AbstractAnalyzer` (siehe oben) implementiert. Um die Feature-Vektoren eines Digitalisats aus der Datenbank abzurufen, stellt diese Klasse entsprechende Methoden bereit. Die als BLOB gespeicherten Werte werden sofort de-serialisiert und beispielsweise als Menge von `ENoteEllipseFitting`-Instanzen zurückgegeben.

Die Klasse `DB2InstancesGetter` implementiert das in Abschnitt 3.2.2 beschriebene *Interface* `InstancesGetter` für die speziellen Anforde-

rungen des umzusetzenden Konzeptes. Der Konstruktor der Klasse erhält als Parameter eine Instanz der Klasse `InstancesSet`, durch die eine Menge von Digitalisaten und ihren Schreibern beschrieben wird. Das Ergebnis einzelner Methoden wird aus Performancegründen bereits durch den Konstruktor berechnet, so dass später nur noch das „materialisierte “ Ergebnis zurückgegeben werden muss. Die Methoden `getTrainingInstancesForScribes` und `getInstancesForSignature` hingegen rufen nur für die tatsächlich angeforderten Daten die entsprechenden Methoden der Klasse `DB2Connection` auf und erzeugen damit die entsprechenden Instanzen für den Klassifikator.

Die Klasse `DataMiningUDF` implementiert jeweils eine UDF für die Trainings-, die Test- und die Anwendungsphase des DM-Prozesses. Außerdem sind Hilfsmethoden für die UDF's vorhanden, wie beispielsweise die Methode `getLogger`, die ein Instanz des Loggers (vgl. Abschnitt 5.1.2) erzeugt, oder die Methode `getPreProcessor`, die den gleichen Preprozessor für alle Phasen des DM liefert.

Die Methode `getClassifierInstance` erhält als Parameter eine Zeichenkette, die den Typ des Klassifikators beschreibt. Diese Zeichenkette wird als Attribut des Datentyps gespeichert. Mögliche Typen für den Klassifikator sind J48, REPTree, OneR, PART und der Standardklassifikator LMT. Weitere Klassifikatoren können problemlos durch Erweiterung dieser Methode hinzugefügt werden.
Die Methode `getClassifierOptions` konvertiert die ebenfalls in Form einer Zeichenkette übergebenen Optionen für den Klassifikator in ein *Array*. Die einzelnen Optionen müssen dafür durch Kommata getrennt werden. Werden keine Optionen gewählt, liefert die Methode die Standardoptionen für den LMT-Klassifikator zurück.

Die Methode `getModel` implementiert die UDF für die Trainingsphase. Einziger Parameter dieser Methode ist die Menge der Trainingsinstanzen. Der Rückgabewert dieser Methode ist das Modell in Form eines BLOB, das als Attribut des Datentyps `enote.de` gespeichert wird. Nach Aufruf dieser Methode wird zunächst eine Instanz des `DB2InstancesGetter` mit den Trainingsinstanzen erzeugt. Dann wird die Anzahl der Instanzen je Schreiber ermittelt und, wie bereits in Abschnitt 3.2.2 für die Klasse `InstancesLimiter` beschrieben, limitiert. Anschließend wird der Klassifikator für jedes verfügbare Merkmal erstellt. Dazu werden zuerst die Trainingsinstanzen geladen und vorbereitet; dann wird der entsprechende Konstruktor der Klasse `LoadAndSaveAnalyzer` aufgerufen, der das Bauen der Klassifikators startet. Anschließend kann das jeweilige Modell als BLOB gespeichert werden.
Der schematische Aufbau des BLOB's zur Speicherung der Modelle ist in Abbildung 18 dargestellt. Als erstes ist die Anzahl der Merkmale (= An-

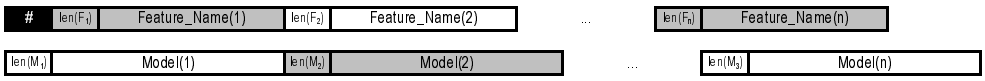

Abbildung 18: Serialisierung der DM-Modelle

zahl der Klassifikatoren) abgebildet. Danach werden die Bezeichnungen der einzelnen Merkmale gespeichert, um eine korrekte Zuordnung der Modelle zu den Merkmalen sicherzustellen. Anschließend werden die einzelnen Klassifikatoren gespeichert. Sowohl für die Namen der Features als auch für die einzelnen Klassifikatoren muss zuerst jeweils die Länge des folgenden Bytestroms geschrieben werden, damit das einzelnen Werte korrekt wieder ausgelesen werden können.

Die UDF für die Testphase des DM wird durch die Methode `testModel` implementiert. Sie erhält das Modell (vgl. Abbildung 18), die Menge der klassifizierten Schreiber und das Menge der Testinstanzen als Eingabeparameter. Rückgabewert dieser Funktion ist die *Accuracy* des Modells. Dieser Wert gibt das Verhältnis der korrekt klassifizierten Instanzen zu deren Gesamtzahl an.
Bei der Bearbeitung dieser Methode wird zuerst ein `DB2InstancesGetter` für die Testinstanzen erzeugt und die Klassifikatoren werden aus dem BLOB rekonstruiert. Danach kann die Klassifikation der einzelnen Digitalisate erfolgen. Dabei werden nacheinander alle unterstützten Merkmale untersucht, indem jeweils die Testinstanzen geladen, vorbereitet und dann klassfiziert werden. Diese Teilergebnisse werden durch die Klasse `ClassificationResult` verwaltet. Nachdem alle Merkmale bearbeitetet wurden, kann mit Hilfe der *Voting Policy* das Ergebnis der Klassifikation aus dem `ClassificationResult` ermittelt und mit dem erwarteten Wert verglichen werden.

Die Methode `analyze` implementiert die UDF für die Anwendungsphase des DM. Im Gegensatz zu den vorigen UDF's wird das zu klassifizierende Digitalisat nicht durch seine Referenz in einer Tabelle der Datenbank, sondern durch seine Feature-Vektoren selbst. Das hat den Vorteil, dass nicht nur bereits in der Datenbank gespeicherte Digitalisate, sondern durch Kombination dieser UDF mit der UDF zu Extraktion der Feature-Vektoren auch andere, extern gespeicherte Notenschriften klassifiziert werden können. Die Deklaration einer solchen Methode des Datentyps `enote.image` erfolgt im folgenden Abschnitt 5.3.2.
Parameter dieser Methode sind somit neben dem Modell und der Liste der Schreiber die unterschiedlichen Feature-Vektoren (i.d.R. als BLOB) des zu klassifizierenden Notenblattes. Als Rückgabewert liefert die UDF eine Zeichenkette mit dem Schreiber, der durch den Klassifikationsprozess für dieses Digitalisat ermittelt wurde.

5.3.2 DB2-Implementierung

In diesem Abschnitt werden die Befehle zum Registrieren der UDF's und des UDT's `enote.dm` im Datenbanksystem vorgestellt. Zuerst werden die UDF's der Klasse `InstancesSetUDF` erstellt:

```
CREATE FUNCTION enote.addTrainInstance(
     BLOB, INTEGER, VARCHAR(255)) RETURNS BLOB
     EXTERNAL NAME '...InstancesSetUDF!addInstanceToBlob'
     ...

CREATE FUNCTION enote.addTestInstance(
     BLOB, BLOB, INTEGER, VARCHAR(255)) RETURNS BLOB
     EXTERNAL NAME '...InstancesSetUDF!addInstanceToBlob2'
     ...

CREATE FUNCTION enote.addToScribesBlob(BLOB, VARCHAR(255))
     RETURNS BLOB
     EXTERNAL NAME '...InstancesSetUDF!getScribeListBlob'
     ...

CREATE FUNCTION enote.getScribesTable(BLOB)
     RETURNS TABLE ( scribe VARCHAR(255) )
     EXTERNAL NAME '...InstancesSetUDF!getScribesTable'
     ...
```

Dann werden die UDF's der Klasse `DataMiningUDF` für die drei Phasen des DM erzeugt:

```
CREATE FUNCTION enote.buildModel(BLOB)
     RETURNS BLOB
     EXTERNAL NAME 'de.enotehistory.udf.DataMiningUDF!getModel'
     ...

CREATE FUNCTION enote.testModel(BLOB, BLOB, BLOB)
     RETURNS DOUBLE
     EXTERNAL NAME 'de.enotehistory.udf.DataMiningUDF!testModel'
     ...

CREATE FUNCTION enote.classify(
     BLOB, BLOB, BLOB, BLOB, BLOB, BLOB, BLOB, INTEGER)
     RETURNS VARCHAR(255)
     EXTERNAL NAME 'de.enotehistory.udf.DataMiningUDF!classify'
     ...
```

Anschließend wird der Datentyp `enote.dm` mit seinen Attributen und Methoden definiert:

```
CREATE TYPE enote.dm AS (
```

```
    comment VARCHAR(255), created TIMESTAMP,
    built TIMESTAMP, tested TIMESTAMP,
    trainingsinstances BLOB,
    testinstances BLOB,
    scribenames BLOB,
    model BLOB,
    accuracy DOUBLE)
INSTANTIABLE
NOT FINAL
...

ALTER TYPE enote.dm
    ADD METHOD buildModel()
        RETURNS enote.dm
        LANGUAGE SQL
        ...
    ...

CREATE METHOD buildModel()
    FOR enote.dm
    RETURN
        SELECT SELF..built(current timestamp)..
        model(enote.buildModel(SELF..trainingsinstances))
        ...
        FROM sysibm.sysdummy1
        WHERE SELF..trainingsinstances IS NOT NULL
        AND SELF..model IS NULL

...
```

Der Konstruktor `enote.dm_init()` wird genutzt, um ein neues (leeres) Data Mining Objekt zu erzeugen.

```
CREATE FUNCTION enote.dm_init()
    RETURNS enote.dm
    SPECIFIC enote.dm_constructor
    LANGUAGE SQL
    NO EXTERNAL ACTION
    RETURN enote.dm()
    ..created(current timestamp)..dbName('enote.images')

CREATE FUNCTION enote.dm_init(
    modelClass VARCHAR(255),
    modelOptions VARCHAR(255) ) RETURNS enote.dm ...

    RETURN enote.dm()..created(current timestamp)
    ..modelClass(modelClass)..modelOptions(modelOptions)
    ..dbName('enote.images')
```

6 Ausblick

Im Rahmen des Projektes *eNoteHistory* soll eine Webschnittstelle entwickelt werden, die dem Nutzer die Funktionalität zur automatische Analyse von Notenhandschriften zur Verfügung stellt. Mit dieser Arbeit wurde die dafür erforderliche Basis geschaffen, indem sowohl für die Verwaltung der Digitalisate und ihrer Feature-Vektoren als auch für die Klassifikation dieser Digitalisate dedizierte Datentypen des Datenbanksystems entwickelt wurden. Der Zugriff auf deren Funtionalität erfolgt durch entsprechende SQL-Anweisungen (siehe Abschnitt 4.1.5 und 4.2.5) beispielsweise via JDBC.

In diesem Abschnitt sollen die noch offenen Punkte im Zusammenhang mit dieser Arbeit aufgezeigt und somit deren Weiterentwicklung hinsichtlich des Projektzieles ermöglicht werden.

Testen des Prototyps: Die mit dieser Arbeit entwickelte Software erfüllt lediglich die Anforderungen einer prototypischen Implementierung, d.h. die Realisierbarkeit des Konzeptes wird gezeigt. Bevor diese Software jedoch außerhalb der Entwicklungsumgebung zum Einsatz kommen kann, sollte sie ausführlich getestet werden.
Die Grundlage dieser Implementierung ist das Framework des IGD. Dieses Framework enthält neben den benötigten Klassen und Methoden auch solche, die für die Funktionalität der UDF's nicht erforderlich wären. So muss beispielsweise die Klasse `ENoteGUI_Swing` eingebunden werden, obwohl keine grafische Oberfläche bereitgestellt wird. Es werden jedoch Methoden dieser Klasse durch Methoden anderer, durch diese Implementierung benötigter Klassen referenziert. Vor allem aus Gründen der Effizienz des Gesamtsystems sollte der Quellcode entsprechend bereinigt und optimiert werden. Dieses betrifft in erster Linie den als *Black Box* eingebundenen Quellcode zur Extraktion der Feature-Vektoren.

Aktualität der verwendeten Klassen: Mit Hilfe der Klasse `ENotePreProcessingDB` wurden die Feature-Vektoren einer Vielzahl an Notenhandschriften bereits extrahiert und im Schema `IPFV` der Datenbank *enote* gespeichert. Einige dieser Digitalisate wurden zum Vergleich auch mit dem Prototyp analysiert. Dabei wurde festgestellt, dass beide Verfahren nicht dasselbe Ergebnis liefern. Die Ursache hierfür liegt vermutlich in unterschiedlichen Versionen des IGD-Frameworks, die jeweils verwendet wurden. Für die Erstellung des endgültigen Systems müssen deshalb zunächst die aktuellen Klassen des IGD-Frameworks ermittelt und eingebunden werden.

Erweiterung des Prototyps: Für die Weiterentwicklung des im Rahmen dieser Arbeit erstellten prototypsichen Implementierung zu einem in der Praxis verwendbaren System sollten folgende Details noch bearbeitetet werden:

- Wie in Abschnitt 5.1.2 beschrieben, wurden bei der Implementierung des Prototyps die Grundlagen für ein einheitliches Logging-Konzept gelegt. Die endgültige Umsetzung eines solchen Konzeptes muss jedoch noch erfolgen. Derzeit schreiben die einzelnen UDF's ihre Log-Ausgaben in separate Dateien, die bei einem erneuten Aufruf der UDF überschrieben werden. Eine alternative Möglichkeit wäre, alle Log-Nachrichten jeweils am Ende einer globalen Log-Datei anzufügen.
  Ebenso sollte ein Konzept für die Verwaltung der Fehler- und Statusmeldungen (SQLSTATE's) entwickelt werden. Gegenwärtig werden Fehlermeldungen lediglich durch die Ausgabe von *Exceptions* signalisiert.

- Neben fixen Einstellungen enthält die protypische Implementierung mehrere zur Laufzeit veränderbarer Parameter. Hierzu zählen beispielsweise das Verzeichnis, in dem temporäre Dateien abgelegt werden sollen, das aktuelle *Loglevel* usw. Diese Einstellungen sollten entweder in einer externen Datei oder einer speziellen Tabelle der Datenbank gespeichert werden. Außerdem werden Methoden benötigt, um diese Werte dieser Parameter lesen und ändern zu können.

TODO's des IGD-Frameworks: In [SB05] werden verschiedene Möglichkeiten zur Weiterentwicklung der DM-Algorithmen des vorhandenen IGD-Frameworks aufgezeigt. Auch die Extraktion der Feature-Vektoren könnte beispielsweise hinsichtlich einer automatischen Erkennung des Notenschlüssels oder der ROI weiterentwickelt werden.

Auswahl der Trainingsinstanzen: Die Auswahl der optimalen Instanzen für das Trainieren des Klassifikators ist ein wesentlicher, noch offener Punkt der automatischen Schreibererkennung. Statt aller vorhandenen Notenhandschriften sollten nur die siginifikantesten Digitalisate für das Bauen des Klassifikators verwendet werden. Ein Algorithmus für die Auswahl dieser Digitalisate ist derzeit jedoch noch nicht vorhanden.

Abkürzungen

	Bedeutung	**eingeführt in Abschnitt**
ARFF	Attribute-Relation File Format	2.2.2
BLOB	Binary Large Object	2.1.1
CBIR	Content Based Image Retrieval	2.1
CLOB	Character Large Object	2.2.3
DBIS	Lehrstuhl für Datenbank- und Informationssysteme	1
DM	Data Mining	2.2
DMG	Data Mining Group	2.2.2
FV	Feature Vector	1
GIF	Graphic Interchange Format	2.1.1
GPL	General Public License	2.2.4
GUI	Graphical User Interface	2.2.4
IGD	Fraunhofer Institut für Grafische Datenverarbeitung	1
IM	DB2 Intelligent Miner	2.2.3
KDD	Knowledge Discovery in Database	2.2
PMML	Predictive Model Markup Language	2.2.2
ROI	Region of Interest	3.1.2
SP	Stored Procedure	3.2.6
UDF	User Defined Function	2.1.1
UDT	User Defined Type	2.1.2
QBIC	Query By Image Content	2.1
WEKA	Waikato Environment for Knowledge Analysis	2.2.4

Literatur

[Alp04] Alpaydin, E.: Introduction to Machine Learning

[Arn02] Arning, A.: Data Mining: Concepts and Implementation Issues.
In: [HLP02]

[Czy05] Czymaj, S.: Konzeptuelle Datenmodellierung für die inhaltsbasierte Suche in Kollektionen von digitalen Bildern

[DBIS05] Lehrstuhl Datenbank- und Informationssysteme: Beschreibung des eNoteHistory Projektes

[FPS96] Fayyad U.; Piatetsky-Shapiro, G.; Smyth, P.: From Data Mining to Knowledge Discovery in Databases
http://www.kdnuggets.com/gpspubs/aimag-kdd-overview-1996-Fayyad.pdf

[Heu97] Heuer, A.: Objektorientierte Datenbanken (2., aktualisierte und erweiterte Auflage)

[HK01] Han, J.; Kamber, M.: Data Mining, Concepts and Techniques

[HLP02] Heuer, A.; Leymann, F.; Priebe, D. (Hrsg): Datenbanksysteme in Büro, Technik und Wissenschaft, 2001

[IBM01] IBM DB2 Extenders, Family Overview
(*http://www-306.ibm.com/software/data/db2/extenders*)

[IBM02] IBM QBIC Home Page
(*http://wwwqbic.almaden.ibm.com*)

[IBM03] IBM Redbooks: Enhance Your Business Applications

[IM02a] IBM DB2 Intelligent Miner Modelling, Administration and Programming, Version 8.1 (2002)

[IM02b] IBM DB2 Intelligent Miner Scoring, Administration and Programming for DB2, Version 8.1 (2002)

[ISO01] ISO/IEC 13249-6 Information Technology - Database Languages - SQL Multimedia and Application Packages - Part 6: Data Mining (Draft)

[ISO02] ISO/IEC 13249-1 Information Technology - Database Languages - SQL Multimedia and Application Packages - Part 1: Framework (Draft)

[ISO03] ISO/IEC 13249-5 Information Technology - Database Languages - SQL Multimedia and Application Packages - Part 5: Still Image

[Kul04] Kulke, S.: IBM DB2 Image Extender (*http://dbis.informatik.tu-cottbus.de/data/pub/skript/mmdb_ws0405/MMDB_ibm_db2_image_extender.pdf*)

[Mil04] Milewski, L.: Integration von Clustering-Classification-Techniken in eine objektrelationale Datenbankumgebung

[SB05] Stahl, R.; Blech, M.: eNoteHistory - Kombination von Bildverarbeitung und Data Mining zur Klassifikation von historischen Notenhandschriften (2005)

[Sch00] Schwenkreis, F.: SQL/MM Part 6: Data Mining

[Sto01] Stolze, K.: SQL/MM Part 5: Still Image - The Standard and Implementation Aspects. In: [HLP02]

[Sto02] Stolze, K.: A DB2 UDB still image extender (*http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0504stolze/*)

[Sto03] Stolze, K.: Still Image Extensions in Database Systems - A Product Overview (*Datenbank-Spektrum 2/2002*)

[Thi02] Thilo, M.: Evaluierung des Schnittstellen-Standards Predictive Model Markup Language (PMML) für Data Mining

[Tue03] Türker, C.: SQL:1999 & SQL:2003 Objektrelationales SQL, SQLJ & SQL/XML

[Urb05] Urban, B.: Schreiberidentifizierung in historischen Notenhandschriften (2005)

[WF00] Witten, I.; Frank, E.: Data Mining, Practical Machine Learning Tools and Techniques with Java Implementations

Abbildungsverzeichnis